CHAPTER 7

Subroutines and Parameters

We introduced the subroutine in Chapter 3 and made extensive use of it in Chapter 4. In this chapter we focus our attention on two important aspects of subroutines. The first is the way in which a subroutine communicates with the program that called it. The second is the way in which a subroutine employs memory for the temporary storage of its data. Before covering these topics, we review the basic principles of subroutines.

7.1 Basic Principles of Subroutines

A subroutine is a piece of code that can be *called* from some point in a program to carry out a certain task. Although we used the expression *piece of code* to describe a subroutine, it is not just any bunch of consecutive instructions. A subroutine is a *coherent* sequence of instructions that carries out a well defined function. One of the reasons for employing subroutines is that a particular subroutine can be used many times in a program without the need to rewrite the same sequence of instructions over and over again. Programmers also employ subroutines to make a program more readable, as we shall soon see.

Let's look at a simple subroutine called `GetChar` whose function is to read a character from the keyboard and put it into data register D0. In this example, we have provided the assembler listing file that gives the addresses (i.e., locations) of the instructions in order to facilitate our explanation of the subroutine call and the return mechanisms.

```
Address Data         Label    Instruction       Comment field

000FFA  41F900004000          LEA    TABLE,A0   A0 points to destination of data
001000  61000206     NextChr  BSR    GetChar    REPEAT Read a character
001004  10C0                  MOVE.B D0,(A0)+          Store it in memory
001006  0C00000D              CMP.B  #$0D,D0    UNTIL  Character = carriage return
00100A  66F4                  BNE    NextChr
                              .
                              .
                              .
```

```
001102  61000104             BSR    GetChar   Read character and quit if "Q"
001106  0C000051             CMP.B  #'Q',D0
00110A  67000EF4             BEQ    QUIT
                             .
                             .
001208  123900008000 GetChar MOVE.B ACIAC,D1  Read ACIA status
00120E  08010001             BTST   #1,D1
001212  66F4                 BNE    GetChar   UNTIL ACIA ready
001214  103900008002         MOVE.B ACIAD,D0  Get character into D0
00121A  4E75                 RTS              Return
```

In this example we call the subroutine `GetChar` *twice*. The first time is from address 001000_{16} in the loop that stores successive characters in a table. The second time is from address 001102_{16} when we read a single character and test it to see whether it is a letter Q.

The way in which the branch or jump to the subroutine is executed is very simple: the 68000's program counter is loaded with the starting address of the subroutine. When the assembler encounters the line `NextChr BSR GetChar` at address 001000_{16}, the symbolic location `GetChar` is translated into the *difference* between the address of the `BSR` instruction plus two and the address of the actual subroutine. This difference is the offset used by the `BSR` instruction and is computed as $1208_{16} - 1002_{16} = 206_{16}$. The instruction is, effectively, `BSR $206`. Note that the value of the PC used in the offset calculation is the address of the `BSR` *plus 2*, because the PC is automatically incremented by 2 as soon as the `BSR` is read. The following code repeats the above fragment of a program with the branch marked.

```
000FFA  41F900004000         LEA    TABLE,A0
001000  61000206     NextChr BSR    GetChar  ---+
001004  10C0                 MOVE.B D0,(A0)+    |
001006  0C00000D             CMP.B  #$0D,D0     |
00100A  66F4                 BNE    NextChr     |
                             .                  |  1208-(1000+2)=206
001102  61000104             BSR    Getchar     |
001106  0C000051             CMP.B  #'Q',D0     |
00110A  67000EF4             BEQ    QUIT        |
                             .                  |
001208  123900008000 GetChar MOVE.B ACIAC,D0  ---+
```

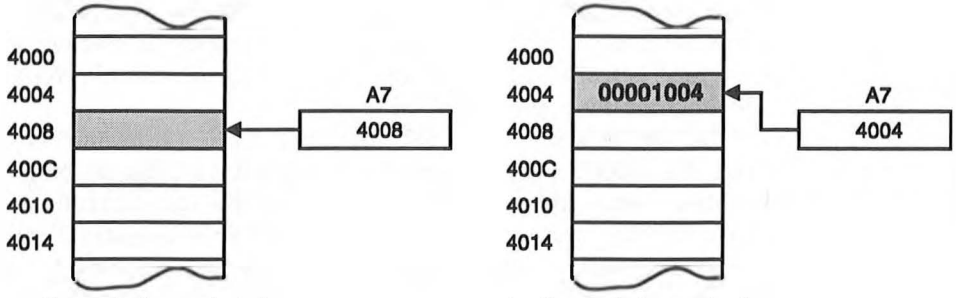
When the program is executed, the 68000 reads the offset from the instruction at location 001000_{16} (i.e., 206_{16}), and adds it to the contents of the program counter (i.e., 1002_{16}) to get 1208_{16}, which is loaded into the program counter to force a jump to the subroutine.

When the assembler next encounters the `BSR GetChar` instruction at address 001102_{16}, the computed offset is different (i.e., $1208_{16} - 1104_{16}$) and the relative branch offset becomes `$104`. By the way, the second subroutine call uses the long form of the `BSR` instruction which takes a 16-bit signed offset. We could have

written BSR.S GetChar which would have taken an 8-bit offset and shortened the code by one word.

A more interesting aspect of subroutines is the way in which a return is made automatically to the appropriate calling point. Whenever a subroutine call is made, the return address (i.e., the location of the next instruction after the call), is pushed onto the stack pointed at by A7. Figure 7.1 demonstrates the action of the first subroutine call in which the return address 1004_{16} is pushed onto the stack — assumed that the stack pointer is initially pointing at address 4008_{16}.

Figure 7.1 Pushing the subroutine return address on the stack



a. State of the stack before pushing a longword on it

b. State of the stack after pushing a longword on it

We can define the effect of the instruction BSR d8, where d8 is an 8-bit offset, in RTL as:

```
[A7]       ← [A7] - 4     Pre-decrement the stack pointer
[M([A7])] ← [PC]          Push the program counter on the stack
[PC]       ← [PC] + d8    Load the program counter with the target address
```

When the subroutine has been executed, an RTS instruction pulls the longword off the top of the stack and deposits it in the program counter to effect the return. The action of the RTS instruction in RTL is:

```
[PC] ← [M([A7])]          Pull return address off the stack
[A7] ← [A7] + 4           Post-increment the stack pointer
```

You can perform a return from subroutine without the use of an RTS instruction by means of an indirect JMP instruction as follows.

```
MOVEA.L (A7)+,A0    A0 now holds the return address
JMP     (A0)        Jump to the return address
```

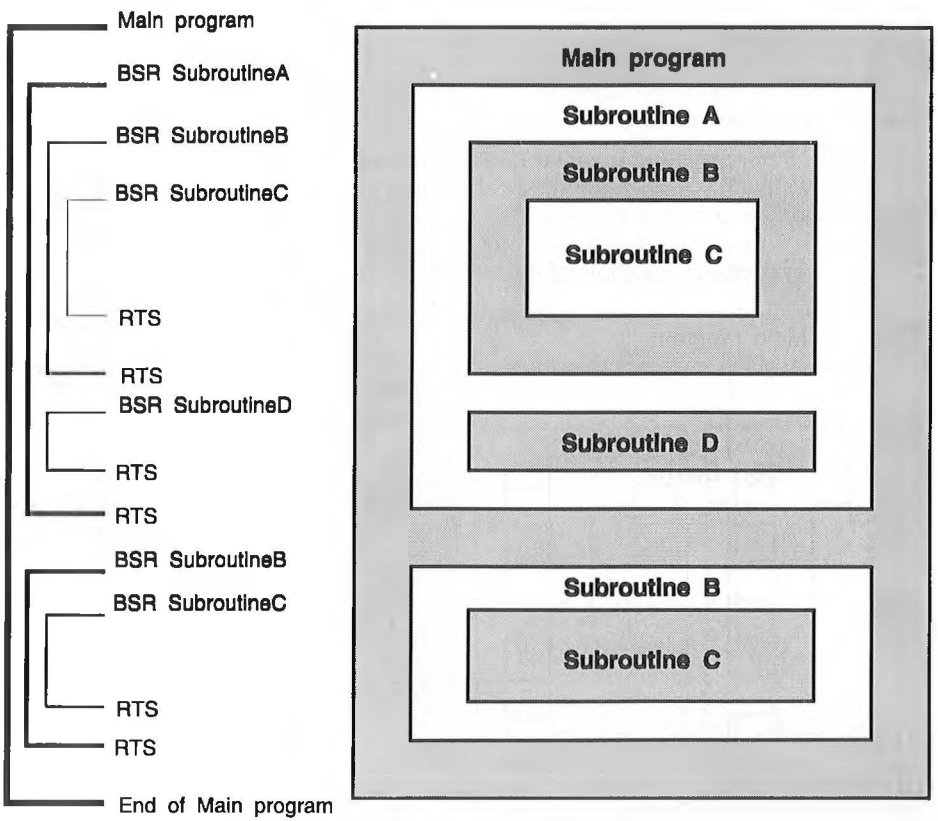
The MOVEA.L (A7)+,A0 instruction reads the contents of memory pointed at by the stack pointer and deposits it in A0. Since the stack pointer points at the return address, the return address is loaded into A0. The post-incrementing ad-

dressing mode steps the stack pointer past the return address and cleans up the stack. The second instruction, `JMP (A0)`, forces a jump to the location whose address is in address register A0. As this is the subroutine return address, a return from subroutine is executed.

Nested Subroutines

If subroutines were called one at a time from a main program and a return made before the next subroutine was called, we would not need to use the stack to store the return address. Any register or fixed location could be used to save the appropriate return address. In practice, a stack is needed to manage subroutine return addresses because subroutines may be *nested*. That is, a main program may call a subroutine and the subroutine may itself call a subroutine, and so on. Figure 7.2 illustrates nested subroutines, in which subroutines B, C, and D are completely enclosed by the subroutine that calls them. For example, from subroutine C you can return only to subroutine B and not to subroutine A or to the main program. We can represent the nesting in Figure 7.2 by the following code:

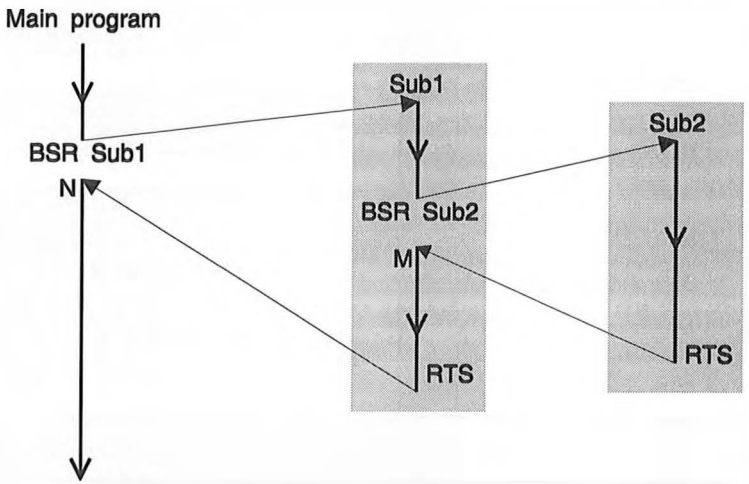
Figure 7.2 Nested subroutines



```
MainProgram  .
             BSR SubroutineA
             .
             BSR SubroutineB
             .
             END
SubroutineA  .
             BSR SubroutineB
             .
             BSR SubroutineD
             .
             .
             RTS
SubroutineB  .
             BSR SubroutineC
             .
             .
             RTS
SubroutineC  .
             .
             RTS
SubroutineD  .
             .
             RTS
```

Consider the two nested subroutines in Figure 7.3. The main program first calls Sub1 and then Sub1 calls Sub2, as the following code demonstrates.

Figure 7.3 Example of nested subroutines

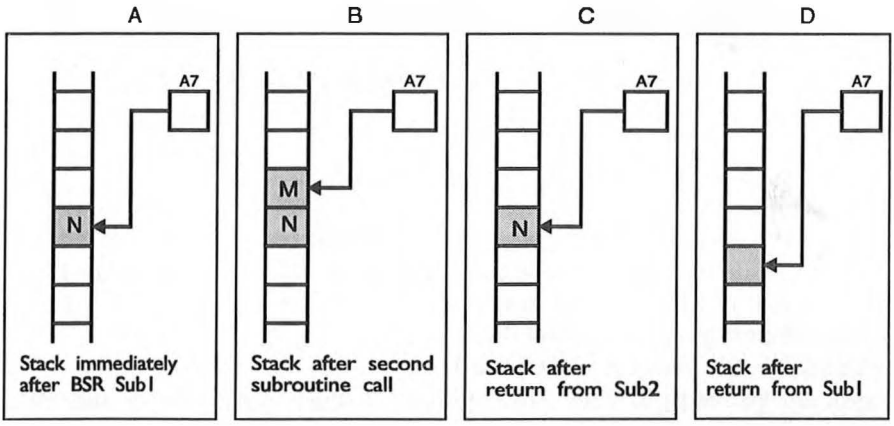


```
MainProgram
                .
                BSR   Sub1
                .
                END
Sub1            .               (subroutine 1)
                .
                .
                BSR   Sub2
                .
                .
                RTS             (return point from subroutine 1)
Sub2            .               (subroutine 2)
                .
                RTS             (return point from subroutine 2)
```

When subroutine Sub1 is called, the return address N is pushed on to the top of the stack — Figure 7.4a. When Sub2 is called from Sub1, return address M is pushed on the stack — Figure 7.4b. Return address, M, is now the new top of the stack. When the second subroutine has been executed to completion, the RTS instruction is executed and a return made to the address at the top of the stack. This return address is M, which is the point in Sub1 immediately after the point at which Sub2 was called. The state of the stack at this point is given in Figure 7.4c. Sub1 is now completed and the RTS at its end loads the address at the top of the stack (i.e., N) into the PC to force a return to the main program. Figure 7.4d illustrates this state, which is the state of the stack before Sub1 was first called.

Subroutines can be nested to any depth by using the stack to save subroutine return addresses, as long as the stack doesn't run out of space for return addresses. The average depth of subroutine nesting is about five. However, systems employing *recursive techniques* may nest subroutines to a much greater depth. A recursive subroutine is a subroutine that calls itself.

Figure 7.4 State of the stack during the execution of the code of Figure 7.3



The term *nesting* implies that one subroutine is entirely enclosed within another, as Figures 7.2 and 7.3 illustrate. In Figure 7.3 a return from Sub2 is made to Sub1. Suppose we write the code in such a way as to return directly from Sub2 to the main program should certain errors arise during the execution of Sub2. Such an operation is possible, as the following code demonstrates.

```
Sub2   .
       .
       BEQ  Exit        Detect a problem here
       .
       .
       RTS              Normal return point to Sub1
Exit   LEA  (4,A7),A7   Move past the normal return point
       RTS              and return directly to main program
```

Sometimes we might think that the detection of an error (or any other event) in a subroutine is best dealt with by returning directly to a higher-level subroutine or to the main program. As you can see from the above code, all you have to do is to skip past the intermediate return addresses on the stack that takes you to the next level of nested subroutine. If you wish to skip two levels of subroutine, you would replace the `LEA (4,A7),A7` instruction by `LEA (8,A7),A7`.

Forcing an exit from a subroutine in this way is regarded as very bad practice indeed. The resulting code is difficult to read; the fundamental rules of structured programming are violated, and the likelihood of introducing a serious bug in the program is increased. We have pointed out that it is possible to skip one or more levels when returning from a subroutine simply to illustrate how the stack functions. The fact that an assembly language *permits* you to do something doesn't make it either right or desirable. You can even force a jump out of a subroutine to any point in the program by means of an unconditional branch or jump (e.g., `BRA Escape` or `JMP PANIC`). Doing this would be a nightmare because the state of the stack would be difficult to determine. The number of return addresses left on the stack would depend on the point from which you jumped. You can always employ the following technique to return from a nested SubB to the main program.

```
       BSR  SubA        Call subroutine A
       .
       .
SubA   .                Subroutine A
       .
       BSR SubB         Call subroutine B
       BCC OK           IF no error in subroutine B THEN continue
       RTS                                          ELSE return
OK     .
       .
       RTS              Normal return point from subroutine A
```

```
SubB   .                  Subroutine B
       .
       .
       ORI   #%00001,CCR Escape from here - Set C-bit for error
       BRA   Rtrn        Now exit
       .
       .
Rtrn   RTS               Normal return
```

In this example the main program calls subroutine A which in turn calls subroutine B. If an error occurs in subroutine B, a return is made to subroutine A. However, subroutine A checks the state of the C-bit immediately after the return and forces a return to the main program if it is set.

Subroutines and Readability

We have said that subroutines are useful because they remove the need to rewrite chunks of code (like an input routine) that are repeated many times in a program. I sometimes use subroutines to improve the readability of a program. Consider the following example.

```
* Calculator program
*
        BSR Initialize   Set up the constants
Again   BSR GetNum1      Read the first number
        BSR GetOpr       Read the operator: one of +,-,*,/
        BSR GetNum2      Read the second number
        BSR Calc         Perform the calculation
        BSR PrintRes     Display the result
        .
        .
GetNum1 BSR GetDigit     Get a digit from the keyboard
        BSR Valid        Test for a valid digit in range 0 to 9
        BCS DigErr       IF not valid digit THEN deal with error
        BSR SaveDigit                       ELSE save it
        .
        RTS
```

This fragment of code demonstrates how you might employ subroutines to make your program highly readable and to comply with the ideas of *top-down design*. The first part of the program consists of a series of subroutines that are called one by one to carry out the function of the program. In this case we might not use, say, the subroutine `Calc` more than once in the program. The only reason for constructing the subroutine called `Calc` is to enhance the readability of the program. For example, if the subroutine `Calc` were replaced by the appropriate

in-line code, anyone reading the program would find it much harder to follow. Even the subroutine `GetNum1` called by the main program is little more than another sequence of subroutine calls. Now that we have covered the basics, we are going to look at how information is passed to and from a subroutine.

7.2 Parameter Passing

We employ a subroutine to carry out a specific action. For example, in a high-level language a subroutine `SIN(X)` might be called to calculate the sine of `X`. A subroutine operates on data that we can call its *input* to produce an *output*. Therefore, a subroutine must transfer information between itself and the program calling it. The only exception occurs when a subroutine is called to trigger an event. For example, a subroutine can be designed to start the motor of a disk drive or to sound an alarm. Simply calling the appropriate subroutine causes the pre-determined action to take place. No explicit data is transferred between the subroutine and the program calling it. In this chapter we deal with two interrelated topics: the way in which information is actually transferred to and from a subroutine, and the way in which the subroutine allocates memory for its own use.

Consider first the application of subroutines to inputting or outputting data. Obtaining a character from a keyboard, or transferring one to a CRT terminal, requires the execution of a number of instructions and is inherently device-dependent (i.e., it involves the control of specific hardware like a keyboard controller). As even a fundamentally simple operation like reading the value of a key might require tens of machine-level operations, input and output transactions are frequently implemented by calling the appropriate input or output subroutine.

Suppose a program calls the subroutine `Put_Char` to display a single character on a CRT terminal. The character is printed by calling the subroutine with `BSR Put_Char`. The entire process of locating the subroutine and returning from it takes place automatically, thanks to the assembler that calculates the relative address, and to the processor's stack mechanism that looks after the return address. In this example, the program has to transfer just one item of information to the subroutine, namely the ASCII code of the character to be displayed.

When only a single character is passed to the subroutine, one of the eight data registers serves as a handy vehicle to transfer the information from the calling program to the subroutine. For example, if the character to be displayed is in a table pointed at by A0 and register D0 is used to carry the character to the subroutine, we can write the following code.

```
MOVE.B  (A0),D0        Pick up the character to be printed
BSR     Put_Char        and print it
```

Programmers frequently transfer data between a program and a subroutine by means of one or more registers, particularly when the quantity of data to be

transferred is either very small or the processor is well endowed with registers. Transferring data via a register permits both *position independent code* and *re-entrancy*. In Chapter 4 we introduced position independent code, PIC, that does not use absolute addresses to refer to operands. You can move a block of PIC from one place in memory to another without recalculating the address of operands. If data is transferred to a subroutine in a data register, position independence is guaranteed because no absolute memory location is involved in the transfer of data. Since, in our example, the character to be output is transferred in D0, we can relocate the program anywhere in memory because D0 does not have an address in memory.

The following fragment of code demonstrates the passing of a parameter via a memory location (in this example it is `HoldChr`).

```
HoldChr DS.B   1              Reserve a location for byte to print
        .
        .
        MOVE.B (A0),HoldChr   Move the character to be printed to
        BSR    Put_Chr        HoldChr and print it
        .
        .
Put_Chr MOVE.B HoldChr,D0     Read the character from HoldChr
        .
        RTS
```

If the program is ever relocated in a different part of memory, it must be recompiled, because the source operand used by `MOVE.B HoldChr,D0` will have a new value. However, before we continue with parameter passing, we have to take a short diversion and look at the re-entrant subroutine.

The only disadvantage in passing information to and from subroutines via registers is that it reduces the number of registers available for use by the programmer. Moreover, the quantity of information that can be transferred is limited by the number of registers. In the case of the 68000, it is theoretically possible to transfer up to 15 long words of data in this way. This total is made up of eight data registers and seven address registers. Address register A7, the stack pointer, cannot itself be used to transfer data.

Re-entrant Subroutines

Re-entrancy is such an important concept that we have a short time-out to introduce it. Suppose you are reading a book and a friend asks to borrow it before you've finished it. You lend the book to your friend who returns it after reading it. You then continue reading from where you left off. This is an everyday example of re-entrancy. All that is needed to make the process re-entrant is for you to remember where you had reached before you lent the book, and for your friend not to rip out pages (i.e., the data should not be corrupted).

A subroutine is re-entrant if a program can begin executing it; if another program can take control of the subroutine and run it *before* the first program has completed it; and if it can be returned to the first program without problems. This sequence is entirely analogous to the example of the book. A necessary requirement for re-entrancy is that the program that takes over the subroutine must leave it in the state in which the program found it.

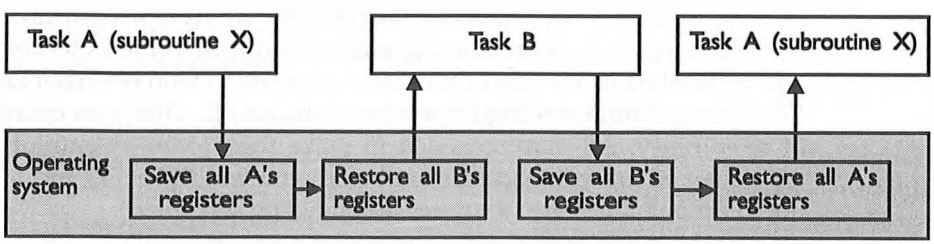
Why is re-entrancy a desirable feature in a computer? More importantly, why should a program wish to borrow a subroutine that is being used by another program? In a *multitasking* system, several programs or *tasks* run *simultaneously*. To be more precise, the operating system switches between user tasks so rapidly that to an outsider (e.g., a human operator) they appear to be running simultaneously. Imagine a situation in which task A is running and using subroutine X to perform a certain function. Suppose the operating system switches tasks while A is still in the middle of subroutine X. Task B also uses subroutine X. When B has finished with subroutine X, it must leave the subroutine in exactly the same condition it found X. Otherwise, the multitasking system simply would not work.

We're not going to deal with multitasking and interrupt processing further at this stage, because they are covered in Chapter 10. What we need to discuss now is the way in which a subroutine can be re-used by another program. Let's return to the example of the output subroutine in which data is transferred to the subroutine in a data register, D0. Re-entrancy is possible as long as the re-use of the subroutine saves the contents of the register employed to transfer the data *before* it is re-used. If this sounds confusing, think about Goldilocks. If she had put the porridge on the back-burner, made herself a new helping, eaten it, washed the dishes, and then restored the original porridge to the front of the stove before the three bears got home, she would have saved herself a lot of trouble.

Consider the situation of Figure 7.5. Initially, task A is running subroutine X and the operating system forces a task switch. In order to preserve task A's *working environment*, all its registers must be saved in a safe place. Before the processor is given to task B, all task B's registers are restored from their safe place. When the operating system intervenes again to switch back to task A, B's registers are saved and A's registers are restored. Task A, which was running subroutine X at the time a switch occurred, is in the same state as before the switch.

It is important to appreciate that task switching should be *invisible* to a subroutine and should be able to occur at any point in the code. Consider the following code in which a task switch takes place at the indicated point (i.e., immediately after the execution of the instruction `BTST.B #0,D1`).

Figure 7.5 Switching in a multitasking environment



```
         MOVE.B   (A0),D0       Pick up the character to be printed
         BSR      Put_Char      and print it
         .
         .
Put_Char MOVE.W   D1,-(A7)      Push D1 on the stack
         MOVE.B   ACIAC,D1      Read status
         BTST.B   #0,D1
         .
         .
         MOVE.W   (A7)+,D1      Restore D1
         RTS
```

Task switching takes place here.
The contents of all registers are
saved and the subroutine can be
re-used by the interrupting program.
When the task is to be run again,
all its registers must be restored.

As you can see from the above code, registers A0 and D1 are in use at the time the task is switched. The contents of these registers must be safely saved and retrieved before the task is run again

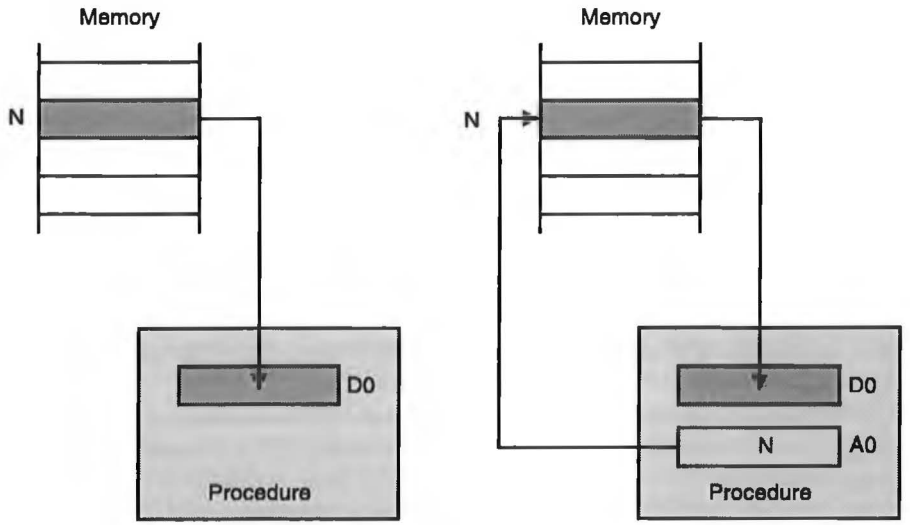
If data is saved in an *absolute* address location, it will be corrupted by the re-use of the subroutine. For example, suppose we store a loop counter in memory location $1234. At some point during the execution of the program, the contents of $1234 indicate how many times a loop has been executed. If an interrupt occurs and another program runs the same subroutine, the new program will also store its loop count in memory location $1234. Consequently, the re-use of the subroutine corrupts any of its data held in fixed or *static* memory locations.

Mechanisms for Parameter Passing

At this point, we are going to talk about how parameters are passed. The two basic ways of passing parameters are transfer by *value* and transfer by *reference.* In the former, the *actual* parameter is transferred, while in the latter, the *address* of the parameter is passed between program and subroutine. Figure 7.6 demonstrates these two concepts.

This distinction between *value* and *reference* is important because it affects the way in which parameters are handled. When passed by value, the subroutine receives a copy of the parameter. Therefore, if the parameter is modified by the subroutine, the *new value* does not affect the *old value* of the parameter elsewhere in the program. In other words, passing a parameter by value causes the parameter to be cloned and the clone to be used by the subroutine.

Figure 7.6 Passing a parameter by value and by reference



(a) Passing a parameter by value. The data in the memory location is copied to the subroutine.

(b) Passing a parameter by reference. The address of the parameter is passed to the subroutine.

When a parameter is passed by address (i.e., by reference), the subroutine receives a *pointer* to the parameter. In this case, there is only one copy of the parameter, and the subroutine is able to access this unique value because it knows the address of the parameter. If the subroutine modifies the parameter, it is modified globally and not only within the subroutine.

The actual mechanism by which information is passed to a subroutine generally falls into one of three categories: a register, a memory location, or the stack. We have already seen how a register is used to transfer an actual value. A region of memory can be treated as a mailbox and used by both calling program and subroutine, with one placing data in the mailbox and the other emptying it. However, it is the stack mechanism that offers the most convenient method of transferring information between a subroutine and its calling program.

Passing Parameters by Reference (i.e., Address)

Let's look at an example of how you might pass parameters to a subroutine by *reference*. Suppose a subroutine is written to search a region of memory containing a text string for the first occurrence of a particular sequence of characters called a substring. The sequence we are looking for (i.e., the substring) is stored as a string in another region of memory. In this example, the subroutine requires four pieces of information: the starting and ending addresses of both the region of text to be searched and the substring to be used in the matching process. Figure 7.7 provides a memory map for this problem.

The information required by the subroutine is the four addresses, 00 1000_{16}, 00 $100B_{16}$, 00 1100_{16}, and 00 1103_{16} that indicate the starting and ending points of the two strings. Note that in this example we are passing the parameters by reference, because the subroutine receives their addresses. We are not passing the actual parameters (i.e., the text string and the substring) themselves. After the subroutine has been executed, it returns the value 00 1007_{16}. While it is possible to transfer all these addresses via four address registers, an alternative technique is to assemble the four parameters into a data block in memory and then pass the address of this block. Figure 7.8 shows how the block is arranged.

The only information required by the subroutine is the address 00 2000_{16}, which points to the first item in the block of parameters stored in memory. The following fragment of code shows how the subroutine is called and how the subroutine deals with the information passed to it. In this example, address register A0 is used to pass the address of the parameter block to the subroutine, and A3, A4, A5, and A6 are used by the subroutine to point to the beginning and end of both the text and the string to be matched. Note that, in practice, we would replace the four `MOVE.L (A0)+,Ai` instructions in the subroutine `MATCH` by a single `MOVEM.L (A0)+,A3-A6` instruction.

```
        ORG     $2000          Set up the parameter block
StrtStrg DS.L   1              Pointer to start of string to be matched
EndStrg  DS.L   1              Pointer to its end
StrtSub  DS.L   1              Pointer to start of substring to be matched
```

Figure 7.7 Memory map of a string-matching problem

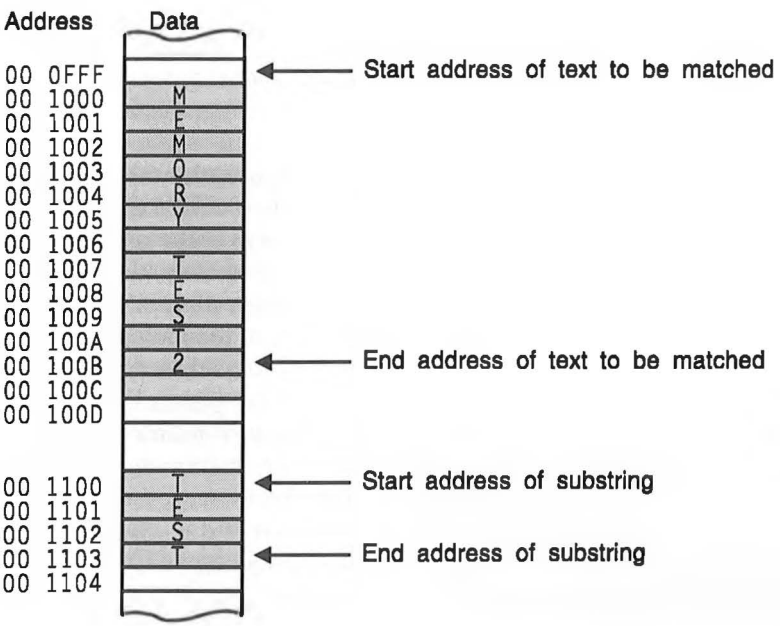
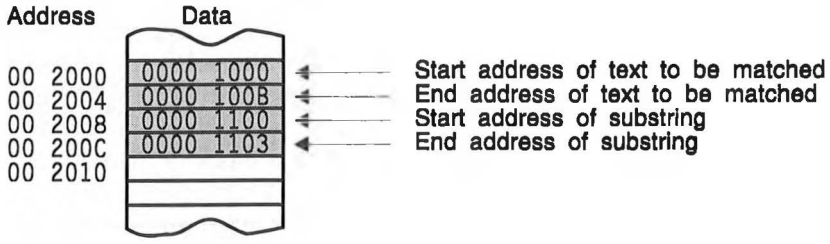


Figure 7.8 The parameter block for Figure 7.7




```
EndSub   DS.L    1              Pointer to its end
         .
         .
         .
         LEA     $002000,A0     Set up address of parameter block in A0
         BSR     MATCH          Call string matching subroutine
         .
         .
         .
MATCH    MOVEM.L A3-A6,-(A7)    Save address registers on stack
         MOVEA.L (A0)+,A3       Put start address of text in A3
         MOVEA.L (A0)+,A4       Put end address of text in A4
         MOVEA.L (A0)+,A5       Put start address of string in A5
         MOVEA.L (A0)+,A6       Put end address of string in A6
         .
         .                      Body of the subroutine
         .
         MOVEM.L (A7)+,A3-A6    Restore address registers from stack
         RTS
```

The four addresses required by the subroutine are obtained by using address register A0 as a pointer to the parameter block. We have passed a parameter to the subroutine by its address (i.e., the address of the parameter block) and have transferred this address in register A0. Unfortunately, this method of storing the parameters in a fixed block in memory cannot always be employed, as such a program cannot be used re-entrantly. Clearly, if the parameters are stored in static memory locations, any attempt to interrupt the subroutine and to re-use it must result in the parameters being overwritten by new values. A much better approach is to use the stack to pass parameters or pointers to parameters. In this case, if the subroutine is interrupted, the new parameters are pushed higher on the stack than the old parameters. When the interrupting program has used the subroutine, a return from interrupt is made with the stack in the same condition as it was immediately prior to the interrupt. Chapter 10 deals with interrupts in more detail.

The Stack and Parameter Passing

The stack is not only useful for storing subroutine return addresses in such a way that subroutines may call further subroutines; it can also be used to transfer information to and from a subroutine. All that need be done is to push the parameters (or their addresses) on the stack before calling the subroutine. Since we are transferring parameters by reference, we will exploit the PEA instruction. The program fragment below shows how this is done for our string-matching algorithm. In this example, we transfer all four parameters by *reference.*

```
        PEA       TEXT_START      Push start and end addresses
        PEA       TEXT_END        of string to search
        PEA       STRING_START    Push start and end addresses
        PEA       STRING_END      of substring used in search
        BSR       STRING_MATCH    Call subroutine for matching
        LEA       (16,A7),A7      Adjust stack pointer
        .
        .
        .
MATCH   LEA       (4,A7),A0       Put pointer to parameters in A0
        MOVEM.L   (A0)+,A3-A6     Get parameters off stack
        .
        .                         Body of subroutine
        .
        RTS
```

The instruction PEA, push effective address, is used to push the address of the four operands onto the stack. We could have used `MOVE.L #TEXT_START,(A7)` to do the same job, but the PEA instruction is much neater. Figure 7.9a shows the initial state of the stack, and Figure 7.9b shows the stack immediately after the four addresses have been pushed. Figure 7.9c shows the state of the stack on subroutine entry, with the return address on top of the parameters.

The first instruction in the subroutine, `LEA (4,A7),A0`, loads address register A0 with the starting address of the last parameter pushed on the stack. We add 4 to the stack pointer because the return address is at the top of the stack. After the subroutine has been called, the instruction `MOVEM.L (A0)+,A3-A6` pulls the four addresses off the stack, and deposits them in address registers A3 to A6 for use as required. Note that these parameters are left on the stack after a return from subroutine is executed (Figure 7.9d).

In the calling program, the instruction `LEA (16,A7),A7` is executed after a return from subroutine has been made. This replaces the contents of the stack pointer with the contents of the stack pointer plus 16. Consequently, the stack pointer is restored to the position it was in before the four 32-bit parameters were pushed on the stack (Figure 7.9a).

Since the subroutine makes use of address registers A3 to A6 to point to the four parameters, any information in these registers is corrupted by the string

matching subroutine. Most programmers would employ a MOVEM instruction to save these registers on the stack before they are used by the subroutine, and then restore them immediately before executing a return from subroutine. The following code demonstrates this point.

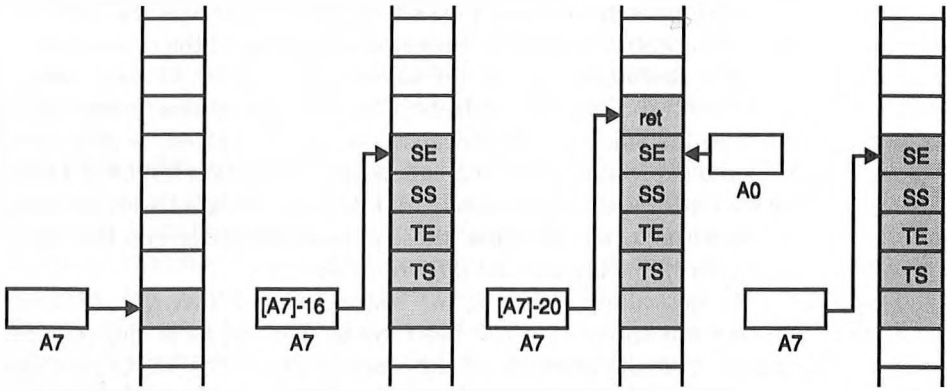
```
MATCH   MOVEM.L  A0/A3-A6,-(A7)   Save working registers
        LEA      (24,A7),A0       Load A0 with pointer to parameters
        MOVEM.L  (A0)+,A3-A6      Get parameters off stack
        .
        .                         Body of subroutine
        .
        MOVEM.L  (A7)+,A0/A3-A6   Restore working registers
        RTS
```

Note that we use an offset of 24 in `LEA (24,A7),A0` because the parameters are buried under the return address and the five registers saved by the `MOVEM.L A0/A3-A6,-(A7)`. Passing parameters on the stack facilitates position independent code and re-entrant programming. If a subroutine is interrupted, the stack builds upward and information currently on the stack is not overwritten.

7.3 The Stack and Local Variables

In addition to the parameters passed between a subroutine and the calling program, a subroutine sometimes needs a certain amount of *local workspace* for its

Figure 7.9 State of the stack during parameter passing



a. State of stack immediately before the parameters are pushed.

b. State of stack after four longword addresses pushed by PEA.

c. State of stack after return address pushed on top. A0 points at the parameters.

d. State of stack after return from subroutine.

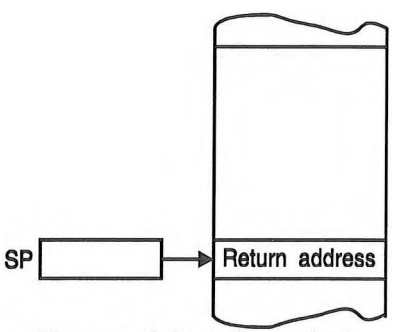
temporary variables. Here, the word *local* means that the workspace is private to the subroutine, and is never accessed by the calling program or by other subroutines that the current subroutine might call. You can sometimes allocate a region of the system's memory space to a subroutine that requires a work area at the time the program is written. You simply reserve fixed locations for the subroutine's variables in a process called static allocation. This approach is satisfactory for subroutines that are not going to be used *re-entrantly* or *recursively*. A subroutine is used recursively if it calls itself.

If a subroutine is to be made re-entrant or used recursively, its local variables must be bound up not only with the subroutine itself, but with the *actual occasion of its use*. In other words, each time the subroutine is called, a new workspace must be assigned to it. For example, suppose a subroutine is being used by task A. Workspace is allocated for use by the subroutine's variables. Suppose now that a task switch takes place while task A is still executing the subroutine and that the subroutine is used by task B. Clearly, task B must be allocated new workspace for its own variables, if it is not to corrupt task A's variables. Once again, the stack provides a convenient mechanism for implementing the dynamic allocation of workspace. This type of storage allocation is called *dynamic* because it is allocated to variables when they are created and then deallocated when the variables are no longer required.

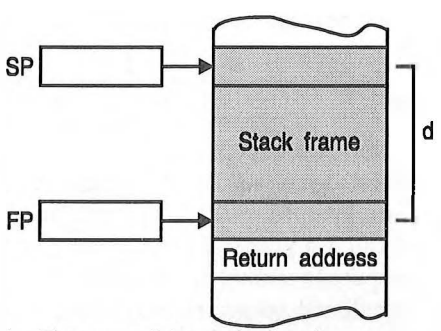
Two items closely associated with dynamic storage techniques for subroutines are the *stack frame* (SF) and the *frame pointer* (FP). The stack frame is a region of temporary storage at the top of the current stack. The frame pointer, which is normally an address register, points to the bottom of the stack frame. Figure 7.10a illustrates the state of the stack after a subroutine call, and Figure 7.10b illustrates the stack frame that has been created on top of the subroutine's return address.

Figure 7.10b also shows how a stack-frame is created merely by moving the stack pointer up by *d* locations at the start of a subroutine. Remember that the 68000's stack grows toward the *low* end of memory, and therefore the stack pointer is *decremented*. It is perhaps unfortunate that we talk of the stack *growing* as the address in the stack pointer gets *smaller*.

Figure 7.10 The stack frame



a. The state of the stack immediately after a subroutine call.



b. The state of the stack after the allocation of a stack frame by executing LEA (-d,SP),SP.

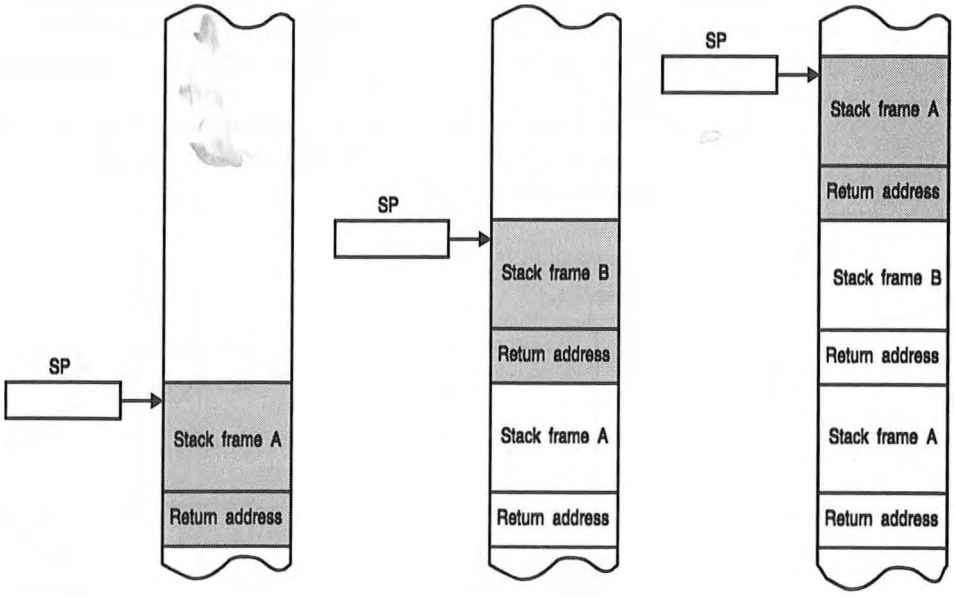
Reserving 100 words of memory is achieved by executing the operation `LEA (-200,A7),A7`. The offset is *200*, as all 68000 offsets are *byte* values. Once the stack frame has been created, local variables can be accessed by any addressing mode that uses A7 as a pointer. However, most programmers access variables via the frame pointer. For example, we could write the code:

```
AnySub  LEA   (-4,A7),A6      Set up A6 as the frame pointer
        LEA   (-200,A7),A7    Create the stack frame
        .
        .                     The subroutine proper
        .
        LEA   (200,A7),A7     Collapse the stack frame
        RTS                   and return from subroutine
```

Before a return from subroutine is made, the stack frame must be collapsed by an `LEA (200,A7),A7` instruction. We shall soon see that the 68000 has two special instructions, `LINK` and `UNLK`, that automatically manage the stack frame.

Figure 7.11 shows how stack frames can be built on top of each other. In this example, subroutine A has a stack frame at the top of the stack while the subroutine is being executed (Figure 7.11a). Suppose that subroutine A calls subroutine B, and that subroutine B also builds a stack frame on top of the stack. This situation is illustrated by Figure 7.11b. Subroutine B accesses data in its stack frame and any data that A was using is safe in its own stack frame.

Figure 7.11 Growth of stack frames



a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to subroutine A

Suppose that subroutine B calls subroutine A and a new stack frame is built on subroutine A's stack (Figure 7.11c). Subroutine A now has *two* stack frames, one for each of its calls. Since the scratchpad data used during each of A's runs is located in *separate* stack frames, these two runs do not interfere with each other. When the second use of subroutine A is completed, the stack frame is collapsed and the situation is the same as Figure 7.11b. When subroutine B has been completed, its stack frame is collapsed, and we are once more at the state of Figure 7.11a.

The Link and Unlink Instructions

The 68000 mechanizes the simple scheme of Figure 7.11 by a complementary pair of instructions, LINK and UNLK (link and unlink). These are relatively complex in terms of their detailed implementation, but are conceptually simple. The great advantage of LINK and UNLK is their ability to let the 68000 manage the stack automatically and to make the memory allocation scheme entirely re-entrant.

If the stack frame storage mechanism is to support re-entrant programming, a new stack frame must be reserved each time a subroutine is called. Since each successive stack frame is of a (possibly) variable size, its length must be preserved. The 68000 uses an address register for this purpose. In the following description of LINK and UNLK, a 16-bit signed constant, *d*, represents the size of the stack frame. At the start of a subroutine LINK creates the stack frame belonging to this subroutine. The following code achieves the desired effect:

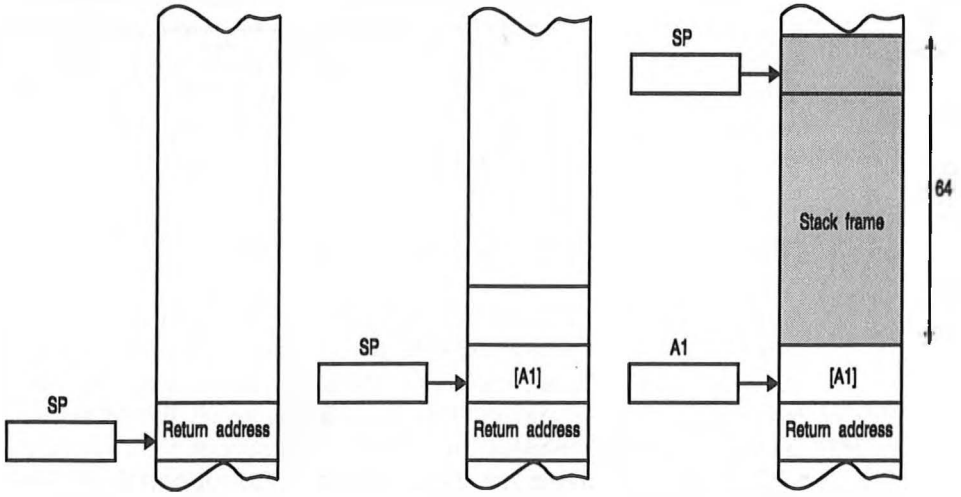
```
Sub1   LINK   A1,#-64        Allocate 64 bytes (16 long words) of storage
         .                   in this stack frame - use A1 as frame pointer
         .
         .                   Body of the subroutine
```

The minus sign in the LINK instruction is needed because the stack grows toward low memory and the stack pointer must be decremented. We can define the action executed by LINK A1,#-64 in RTL terms.

```
LINK: [SP]       ← [SP] - 4     Decrement the stack pointer by 4
      [M([SP])]  ← [A1]         Push the contents of address register A1
      [A1]       ← [SP]         Save stack pointer in A1
      [SP]       ← [SP] - 64    Move stack pointer up by 64 locations
```

Clearly, a LINK instruction does more than simply move the stack pointer to the top of the stack frame. Figure 7.12 shows the stages of the execution of a LINK A1,#-64. The first operation performed by the LINK is to save the old contents of A1 by pushing them on the stack (Figure 7.12b). Pushing the old value of the frame pointer on the stack makes sense because the previous stack pointer can later be restored when the current stack frame is thrown away. Similarly, the old value of the stack pointer is preserved in A1 (Figure 7.12c). In other words, the LINK destroys no information and, therefore, it is possible to undo the effect of a

Figure 7.12 Executing a LINK instruction



a. The state of the stack immediately after the subroutine call

b. The state of the stack after register A1 has been pushed on the stack.

c. The state of the stack after the LINK instruction. The old SP is loaded into A1 and the SP moved up by 64 bytes.

LINK at some later time. Note that in Figure 7.12c the stack pointer is moved to the top of the stack frame, and the region between this point and the saved value of A1 is available to the subroutine. Address register A1 points at the base of the stack frame and can be used to reference data on the stack frame.

The next step is to demonstrate how the work of the LINK instruction is undone at the end of the subroutine. The following code shows how you would use an UNLK instruction before returning from a subroutine.

```
Sub1  LINK   A1,#-64     Allocate 64 bytes (16 long words) of storage
      .                  in this stack frame - use A1 as frame pointer
      .
      .                  Body of the subroutine
      .
      UNLK   A1          De-allocate Subroutine 1's stack frame
      RTS                Return to calling point.
```

The RTL definition of UNLK A1 is:

```
UNLK:   [SP]  ← [A1]
        [A1]  ← [M([SP])]
        [SP]  ← [SP] + 4
```

The stack pointer is first loaded with the contents of address register A1. Remember that A1 contains the value of the stack pointer just before the stack

frame was created. By doing this, the stack frame collapses. The next step is to pop the top item off the stack and place it in A1. This has two effects. It returns both the stack and the contents of A1 to the points they were in before LINK was executed. Following the UNLK, a return from subroutine can be made.

Subroutines often employ several data and address registers. Since this re-use of registers by another subroutine will overwrite and corrupt their contents, it is necessary to save any registers that will be used by the new subroutine and then restore them. Many programmers employ a MOVEM instruction to save registers on the stack at the start of a subroutine. Consider the following fragment of code.

```
Sub1 LINK    A1,#-64              Allocate 64 bytes of storage in this
*                                 stack frame - use A1 as frame pointer
     MOVEM.L D0-D7/A3-A6,-(SP)    Save working registers on the stack
     .
     .                            Body of the subroutine
     .
     MOVEM.L (SP)+,D0-D7/A3-A6    Restore working registers
     UNLK    A1                   De-allocate the subroutine stack frame
     RTS                          Return to calling point.
```

Note that we wrote *SP* rather than *A7* — they are interchangeable. The state of the stack before the subroutine call, after the call and the LINK operation, and after a MOVEM instruction is described in Figure 7.13. In this example, the working registers, D0 to D7 and A3 to A6, are saved on the stack. The stack frame's temporary storage allocation is 64 bytes and address register A1 is used by LINK. Note that we first create the stack frame by the LINK instruction and then push the working registers by means of the MOVEM. That is, the sequence is:

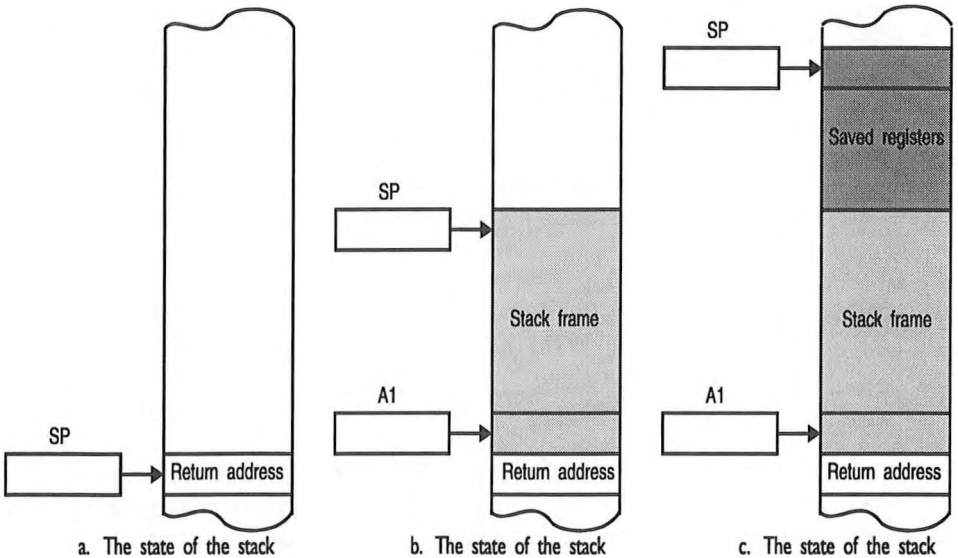
```
LINK    A1,#-64              Create the stack frame
MOVEM.L D0-D7/A3-A6,-(SP)    Save the registers
.
.
MOVEM.L (SP)+,D0-D7/A3-A6    Restore registers
UNLK    A1                   Delete the stack frame
```

Alternatively, we could have chosen the sequence:

```
MOVEM.L D0-D7/A3-A6,-(SP)    Save the registers
LINK    A1,#-64              Create the stack frame
.
.
UNLK    A1                   Delete the stack frame
MOVEM.L (SP)+,D0-D7/A3-A6    Restore registers
```

In this case, the working registers are stacked *before* creating the stack frame. Although both techniques are equally valid, the recommended approach is to

Figure 7.13 Using a MOVEM to save registers



create the stack frame *before* saving the registers. The only reason for adopting this sequence is that it sometimes makes it easier to manipulate the stack frame.

In Figure 7.12 the `LINK` instruction sets address register A1, the frame pointer, to point to the base of the stack frame. Any temporary variables created by the subroutine can be accessed by means of the frame pointer. For example, `MOVE.W D3,(-2,A1)`, stores the low-order word in data register D3 in the stack frame immediately on top of the saved value of A1.

Example of the Use of Local Variables

Consider the following example of a subroutine that uses the `LINK/UNLK` pair. A subroutine, `CALC`, uses three parameters, P, Q, and R. Parameters P and Q are called by *value*, and parameter R by *reference*. This subroutine calculates the value of $(P^2+Q^2)/(P^2-Q^2)$. Although it is not always necessary to save working registers on the stack, we do it in the following example because it is good practice. Having to remember that calling a certain subroutine corrupts the values of D3 and A4 is not as good as ensuring that D3 and A4 are not corrupted by the subroutine. We use the `LINK` instruction to create a stack frame (which can be used by the subroutine as temporary storage).

```
*                              D0 contains value of P
*                              D1 contains value of Q
```

```
MAIN  MOVE.W  D0,-(SP)          Push the value of P on the stack
      MOVE.W  D1,-(SP)          Push the value of Q on the stack
      PEA     R                 Push the reference to R on the stack
      BSR     CALC              Call the subroutine
      LEA     (8,SP),SP         Clean up the stack by removing P, Q, R
      .
      .
      STOP    #$2700            End of program
      .
      .
CALC  LINK    A0,#-14           Establish 3 longword + 1 word frame for the stack
      MOVEM.L D6/A6,-(SP)       Save working registers on the stack
      MOVE.W  (14,A0),D6        Get value of P from stack
      MULU    D6,D6             Calculate P²
      MOVE.L  D6,(-4,A0)        Save P² on the stack frame
      MOVE.W  (12,A0),D6        Now get Q from the stack
      MULU    D6,D6             Calculate Q²
      MOVE.L  D6,(-8,A0)        Store Q² on the stack frame
      MOVE.L  (-4,A0),D6        Get P²
      ADD.L   (-8,A0),D6        Add Q²
      MOVE.L  D6,(-12,A0)       Store P²+Q² on the stack frame
      MOVE.L  (-4,A0),D6        Get P²
      SUB.L   (-8,A0),D6        Subtract Q²
      MOVE.W  D6,(-14,A0)       Store P²-Q² on the stack frame (as a word)
      MOVE.L  (-12,A0),D6       Copy P²+Q² from stack frame to D6
      DIVU    (-14,A0),D6       Calculate (P²+Q²)/(P²-Q²) Note 16-bit arithmetic
      LEA     (8,A0),A6         Get pointer to address of R
      MOVEA.L (A6),A6           Get actual address of R
      MOVE.W  D6,(A6)           Modify R in the calling routine
      MOVEM.L (SP)+,D6/A6       Restore working registers
      UNLK    A0                Collapse the stack frame
      RTS
```

This code is rather cumbersome because we have not optimized the usage of registers. That is, we did not really have to use a stack frame, because the 68000 has sufficient registers to hold all temporary variables. However, a compiler for a high-level language might produce an assembly language output similar to the above code. Figure 7.14 provides a composite illustration of how the stack grows during the execution of this code.

In order to clarify the process we have just described, we will walk through the code and present a snapshot of the stack as the instructions are executed. Before the subroutine is called, the two `MOVE.W <register>,-(SP)` instructions push the *values* of the parameters P and Q onto the stack. The following instruction, `PEA R`, pushes the *address* of R onto the stack. P and Q each take up a word, and R takes up a longword because the address of R is pushed rather than its value. Figure 7.15 illustrates the stack at this stage.

Figure 7.14 State of the stack during the execution of the CALC subroutine

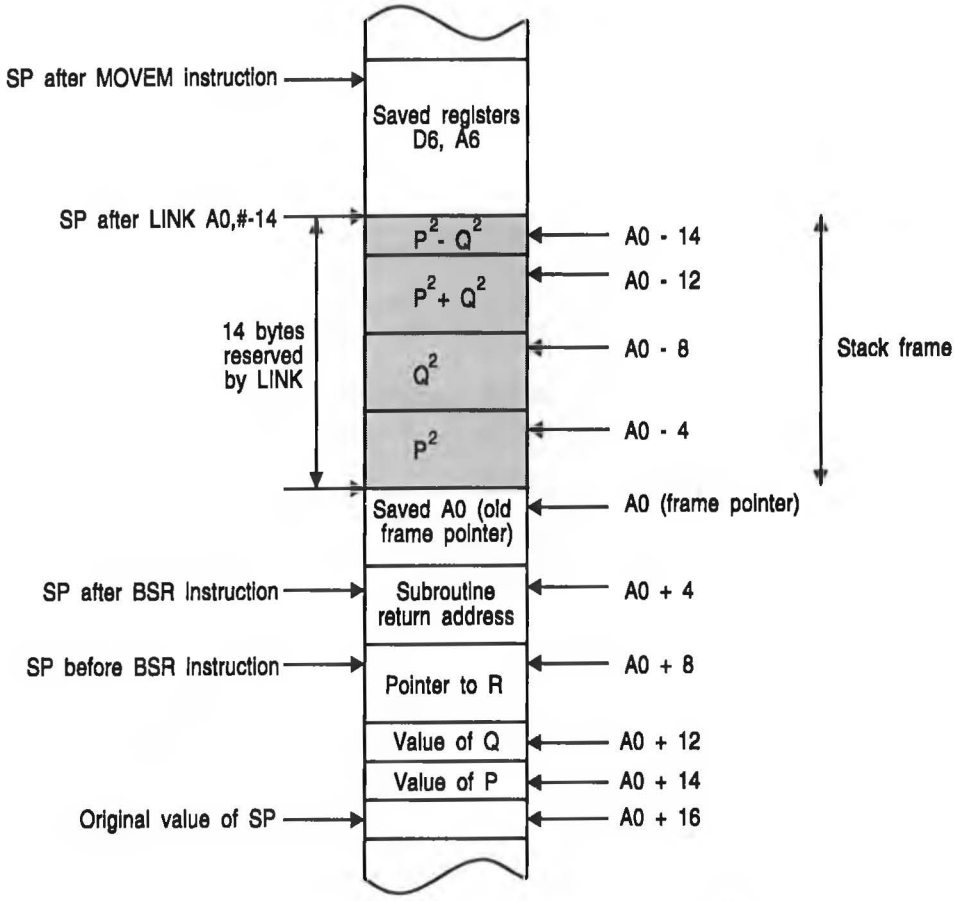
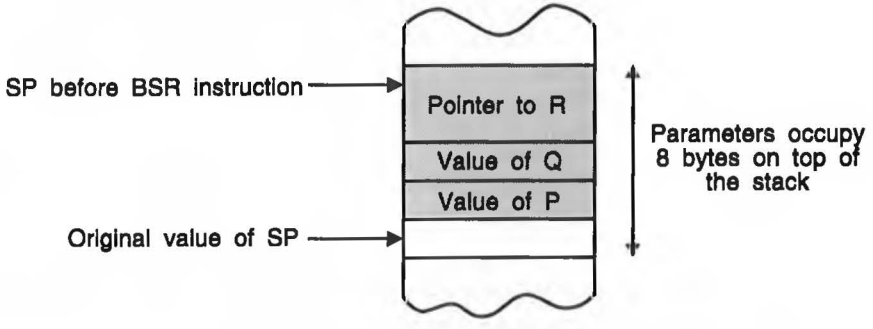


Figure 7.15 State of the stack after pushing the parameters



After calling the subroutine, the `LINK A0,#-14` instruction pushes the old value of A0 (the frame pointer) on the stack and moves the stack pointer up by 14 bytes (Figure 7.16). A0 is loaded with the value of the stack pointer immediately before it was moved up 14 bytes. That is, A0 is the current frame pointer and is now pointing at the base of the subroutine's stack frame. We can access all stack values by reference to A0 — we could also use the SP as a reference. It is better to use A0 (i.e., the FP) rather than the stack pointer to access items in the stack frame, since A0 will be constant for the duration of this procedure, while the stack pointer may be modified. Note that the stack frame is 14 bytes because it holds three longwords (P^2, Q^2, P^2+Q^2) and a word (P^2-Q^2). We save P^2-Q^2 as a word, because the 68000 requires that a divisor be a word value.

Figure 7.17 illustrates the next stage in the process in which the two working registers, D6 and A6, are saved on the stack above the stack frame. This operation takes up two longwords on the stack. You should appreciate that we save the registers *after* creating a stack frame to simplify the pointer arithmetic. If the saved registers were below the stack frame, the offsets of the parameters on the stack (i.e., P, Q, and R) would depend on the number of registers saved. By putting the registers on top of the stack frame, we do not need to worry about the space taken by the saved registers.

At this stage, Figure 7.17, the work of the subroutine can be carried out and parameters accessed from the stack. Note that we get P and Q as values from the

Figure 7.16 State of the stack after the subroutine call and `LINK A0,#-14`

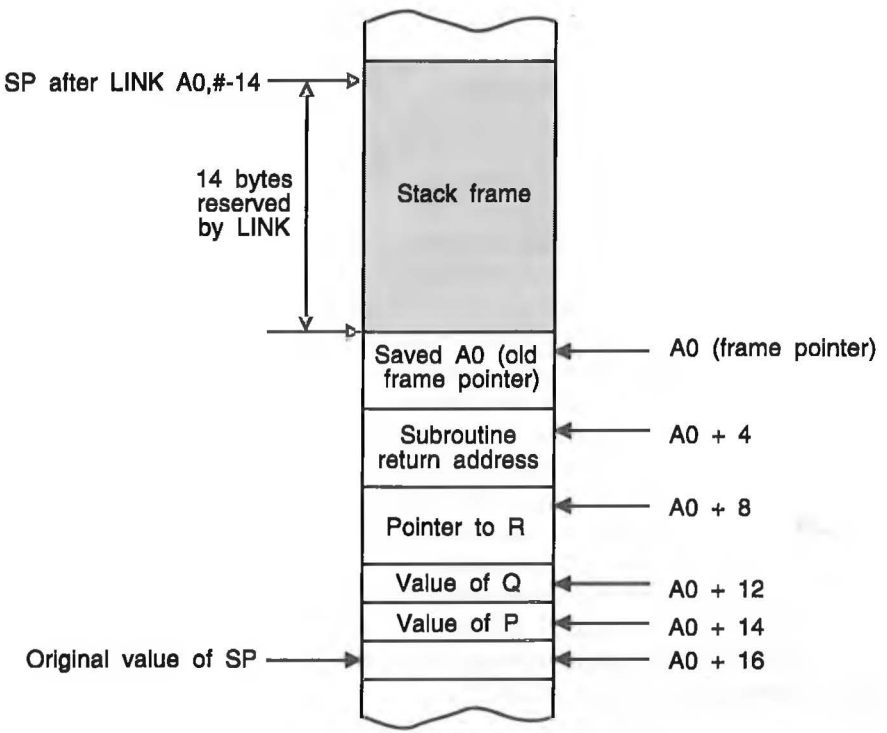
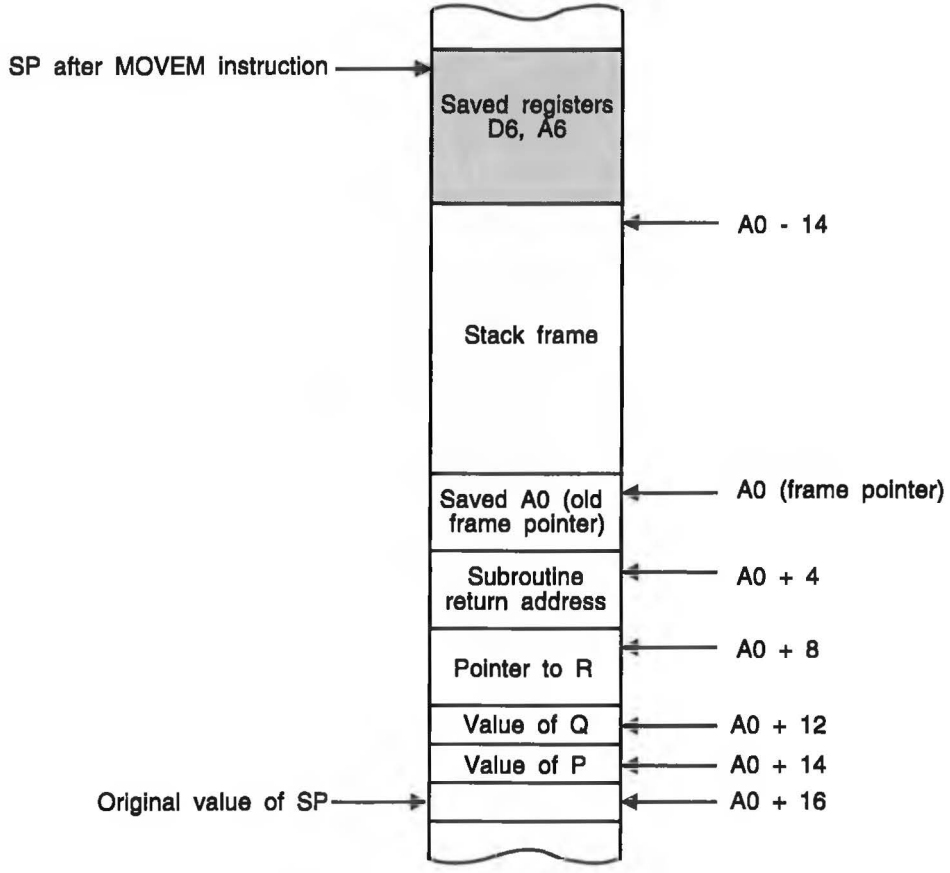


Figure 7.17 State of the stack after saving the working registers



stack, and then access R indirectly by reading its address from the stack. The value of P is accessed by `MOVE.W (14,A0),D6` because A0 points to the base of the stack frame and we have to step past the old A0 (4 bytes), the return address (4 bytes), and R and Q (4 and 2 bytes) to access P itself. The total offset is 14 bytes.

Accessing R, which is passed by reference, is more complex. The instruction `LEA (8,A0),A6` loads the *address* of the parameter on the stack. However, the parameter on the stack is the address of R and we must use the instruction `MOVEA.L (A6),A6` to get a pointer to parameter R itself in A6.

After the procedure has done its work, we first use a `MOVEM.L (SP)+,D6/A6` instruction to restore the contents of D6 and A6 to their original values. We then execute an `UNLK A0` instruction to collapse the stack frame, discard the current frame pointer, and restore the frame pointer to its value immediately before the subroutine was called. However, since three parameters, taking a total of 8 bytes, are left on the stack, we have to clean up the stack by executing an `LEA (8,SP),SP` instruction.

Since the LINK and UNLK instructions are very important to the programmer who is going to write sophisticated 68000 programs, we will use the 68000 simulator to go through the example. Before we can do that, we need to build the subroutine into a test fixture.

We have set up an initial value for the stack pointer, the variables P and Q, the contents of A0, A6, and D6, and have reserved a memory location for R (i.e., 2000_{16}). The contents of registers A0, A6, and D6 have been set to the values $A0A0A0A0, $A6A6A6A6, and $D6D6D6D6, respectively, so that they will stand out when we look at the stack. The initial values of P and Q are 7 and 6, respectively.

The following output is from the Teesside cross-assembler. Remember that the cross-assembler accepts only the original 68000 assembly language form of address register indirect addressing. For example, MOVE -12(A0),D2 must be entered rather than MOVE (-12,A0),D2.

```
Source file: LINK1.X68
Assembled on: 93-02-02 at: 19:59:19
         by: X68K PC-2.0 Copyright (c) University of Teesside 1989,93
Defaults: ORG $0/FORMAT/OPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO

 1  00000400                    ORG       $400
 2  00000400 2E7C00001026       MOVEA.L   #$1026,SP      ;Set up the stack pointer
 3  00000406 203C00000007       MOVE.L    #7,D0          ;Set up P
 4  0000040C 223C00000006       MOVE.L    #6,D1          ;Set up Q
 5  00000412 4DF9A6A6A6A6       LEA       $A6A6A6A6,A6   ;Set up dummy A6
 6  00000418 41F9A0A0A0A0       LEA       $A0A0A0A0,A0   ;Set up dummy A0
 7  0000041E 2C3CD6D6D6D6       MOVE.L    #$D6D6D6D6,D6  ;Set up dummy D6
 8                         *
 9  00000424 3F00               MOVE.W    D0,-(SP)       ;Push value of P on stack
10  00000426 3F01               MOVE.W    D1,-(SP)       ;Push value of Q on stack
11  00000428 487900002000       PEA       R              ;Push reference to R on stack
12  0000042E 6100000A           BSR       CALC           ;Call the subroutine
13  00000432 4FEF0008           LEA       8(SP),SP       ;Clean up stack by removing P,Q,R
14  00000436 4E722700           STOP      #$2700
15                         *
16  0000043A 4E50FFF2     CALC: LINK      A0,#-14        ;Stack frame=3 longwords + word
17  0000043E 48E70202           MOVEM.L   D6/A6,-(SP)    ;Save working registers on the stack
18  00000442 3C28000E           MOVE.W    14(A0),D6      ;Get value of P from stack
19  00000446 CCC6               MULU      D6,D6          ;Calculate P²
20  00000448 2146FFFC           MOVE.L    D6,-4(A0)      ;Save P² on the stack frame
21  0000044C 3C28000C           MOVE.W    12(A0),D6      ;Now get Q from the stack
22  00000450 CCC6               MULU      D6,D6          ;Calculate Q²
23  00000452 2146FFF8           MOVE.L    D6,-8(A0)      ;Store Q² on the stack frame
24  00000456 2C28FFFC           MOVE.L    -4(A0),D6      ;Get P²
25  0000045A DCA8FFF8           ADD.L     -8(A0),D6      ;Add Q²
26  0000045E 2146FFF4           MOVE.L    D6,-12(A0)     ;Store P²+Q² on the stack frame
```

```
27  00000462 2C28FFFC          MOVE.L   -4(A0),D6      ;Get P²
28  00000466 9CA8FFF8          SUB.L    -8(A0),D6      ;Subtract Q²
29  0000046A 3146FFF2          MOVE.W   D6,-14(A0)     ;Store P²-Q² on the stack frame
30  0000046E 2C28FFF4          MOVE.L   -12(A0),D6     ;Read P²+Q² from the stack frame
31  00000472 8CE8FFF2          DIVU     -14(A0),D6     ;Calculate (P²+Q²)/(P²-Q²)
32  00000476 4DE80008          LEA      8(A0),A6       ;Get pointer to address of R
33  0000047A 2C56              MOVEA.L  (A6),A6        ;Get actual address of R
34  0000047C 3C86              MOVE.W   D6,(A6)        ;Modify R in the calling routine
35  0000047E 4CDF4040          MOVEM.L  (SP)+,D6/A6    ;Restore working registers
36  00000482 4E58              UNLK     A0             ;Collapse the stack frame
37  00000484 4E75              RTS
38                        *
39  00002000                   ORG      $2000
40  00002000 00000004    R:    DS.L     1
41           00000400          END      $400

Lines: 41, Errors: 0, Warnings: 0.
```

We will load the program and run it under the trace mode. Every so often, we will examine the state of the stack by using the simulator's MD (memory display function). Remember that the stack grows toward lower addresses.

```
OK, e68k LINKDEMO
[E68k 4.0 Copyright (c) Teesside Polytechnic 1989]

Address space 0 to ^10485759 (10240kbytes), MAXKB is ^4.
Loading binary file "LINK.BIN".
Start address: 000400, Low: 00000400, High: 00000485

>DF

PC=000400 SR=2000 SS=00A00000 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVEA.L  #4134,SP

>TR

PC=000406 SR=2000 SS=00001026 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001026 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.L   #7,D0
```

```
Trace>

PC=00040C SR=2000 SS=00001026 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001026 Z=0
D0=00000007 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.L    #6,D1

Trace>

PC=000412 SR=2000 SS=00001026 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001026 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>LEA       $A6A6A6A6,A6

Trace>

PC=000418 SR=2000 SS=00001026 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001026 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>LEA       $A0A0A0A0,A0

Trace>

PC=00041E SR=2000 SS=00001026 US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001026 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.L    #-690563370,D6

Trace>

PC=000424 SR=2008 SS=00001026 US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001026 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>MOVE.W    D0,-(SP)

Trace>
```

```
PC=000426 SR=2000 SS=00001024 US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001024 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>MOVE.W   D1,-(SP)

Trace>

PC=000428 SR=2000 SS=00001022 US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001022 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>PEA      $00002000

Trace>

PC=00042E SR=2000 SS=0000101E US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=0000101E Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>BSR.L    $0000043A
```

At this stage we have set up the initial values of A0, A6, D0, D1, and D6. Note that the initial value of the stack pointer, A7, was 1026_{16}, and is now $101E_{16}$ because we have pushed eight bytes on the stack. We will now start executing the program proper using the values we have set up.

```
>MD 1016

001016  00 00 00 00 00 00 00 00 00 00 20 00 00 06 00 07
001026  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00.
```

As you can see, the stack contains the values 00002000_{16}, 0006_{16}, and 0007_{16}. We will continue tracing.

```
>TR

PC=00043A SR=2000 SS=0000101A US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=0000101A Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>LINK     A0,#-14
```

```
Trace>

PC=00043E SR=2000 SS=00001008 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001008 Z=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001008 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
---------->MOVEM.L  A6/D6,-(SP)

Trace>

PC=000442 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
---------->MOVE.W   14(A0),D6
```

We have just pushed registers A6 and D6 on the stack and are about to execute the instruction `MOVE.W (14,A0),D6`. Let's have another look at the current state of the stack.

```
>MD 1000

001000  D6 D6 D6 D6 A6 A6 A6 A6 00 00 00 00 00 00 00 00
001010  00 00 00 00 00 00 A0 A0 A0 A0 00 00 04 32 00 00
001020  20 00 00 06 00 07 00 00 00 00 00 00 00 00 00 00.
```

Starting at the top of the stack and moving down we have: the contents of registers D6 and A6 saved by the `MOVEM.L` instruction, 14 bytes of zero comprising the stack frame itself (the simulator initializes memory to zero), the original value of A0 (the frame pointer), the subroutine return address (00000432_{16}), and the eight bytes pushed on the stack before calling the subroutine. We will continue to trace the program.

```
>TR

PC=000446 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D60007 D7=00000000 C=0
---------->MULU     D6,D6

Trace>
```

```
PC=000448 SR=2008 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000031 D7=00000000 C=0
------>MOVE.L    D6,-4(A0)
```

The instruction MULU D6,D6 squares the contents of D6 (i.e., P = 7) to provide a result 31_{16} (this is $49_{10} = 7^2$).

```
Trace>
PC=00044C SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000031 D7=00000000 C=0
------>MOVE.W    12(A0),D6

Trace>

PC=000450 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000006 D7=00000000 C=0
------>MULU       D6,D6

Trace>

PC=000452 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000024 D7=00000000 C=0
------>MOVE.L    D6,-8(A0)

Trace>

PC=000456 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000024 D7=00000000 C=0
------>MOVE.L    -4(A0),D6

Trace>
```

```
PC=00045A SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000031 D7=00000000 C=0
------>ADD.L     -8(A0),D6

Trace>
PC=00045E SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000055 D7=00000000 C=0
------>MOVE.L    D6,-12(A0)
```

We have now added $P^2 + Q^2 = 49 + 36 = 85 = 55_{16}$. As you can see, this value is currently stored in data register D6.

```
Trace>

PC=000462 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000055 D7=00000000 C=0
------>MOVE.L    -4(A0),D6

Trace>

PC=000466 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000031 D7=00000000 C=0
------>SUB.L     -8(A0),D6

Trace>

PC=00046A SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=0000000D D7=00000000 C=0
------>MOVE.W    D6,-14(A0)
```

At this point, the contents of D6 are $P^2 - Q^2 = 49 - 36 = 13 = D_{16}$.

```
Trace>
PC=00046E SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=0000000D D7=00000000 C=0
------>MOVE.L    -12(A0),D6

Trace>
PC=000472 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000055 D7=00000000 C=0
------>DIVU      -14(A0),D6

Trace>
PC=000476 SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00070006 D7=00000000 C=0
------>LEA       8(A0),A6
```

We have now performed the division of $(P^2 + Q^2)/(P^2 - Q^2) = 85/13$, which is 6 remainder 7. This is the value currently in D6. Before continuing, we will have another look at the stack.

```
>MD 1000

001000  D6 D6 D6 D6 A6 A6 A6 A6 00 0D 00 00 00 55 00 00
001010  00 24 00 00 00 31 A0 A0 A0 A0 00 00 04 32 00 00
001020  20 00 00 06 00 07 00 00 00 00 00 00 00 00 00 00
001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00.
```

We continue tracing. The next two instructions fetch the address of parameter R and then store the result (i.e., 6) at this location.

```
>tr

PC=00047A SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=0000101E A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00070006 D7=00000000 C=0
------>MOVEA.L  (A6),A6
```

```
Trace>
PC=00047C SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00002000 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00070006 D7=00000000 C=0
------>MOVE.W    D6,(A6)
```

Now that the work's all done, we can pack up and go home. The saved registers are first retrieved and the stack frame collapsed.

```
Trace>

PC=00047E SR=2000 SS=00001000 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00002000 A7=00001000 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00070006 D7=00000000 C=0
------>MOVEM.L   (SP)+,A6/D6

Trace>

PC=000482 SR=2000 SS=00001008 US=00000000        X=0
A0=00001016 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001008 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>UNLK      A0

Trace>

PC=000484 SR=2000 SS=0000101A US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=0000101A Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>RTS

Trace>

PC=000432 SR=2000 SS=0000101E US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=0000101E Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
------>LEA       8(SP),SP
```

```
Trace>

PC=000436 SR=2000 SS=00001026 US=00000000        X=0
A0=A0A0A0A0 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=A6A6A6A6 A7=00001026 Z=0
D0=00000007 D1=00000006 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=D6D6D6D6 D7=00000000 C=0
———>STOP      #9984
```

The final step is to examine the contents of memory location 2000_{16}. It should contain the result (i.e., 6).

```
>MD 2000

002000  00 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00.
```

It does, it does. This example should demonstrate three things. The first is how the two instructions `LINK` and `UNLK` work. The second is how the stack can be used to pass parameters to a procedure both by reference and by value. The third is the inherent difficulty in writing assembly language programs. For example, in order to reference parameter P, we had to execute `MOVE.W (14,A0)`, which meant that we had to have a clear picture of the location of P with respect to the frame pointer, A0. The chance of making a mistake in dealing with the stack in programs like this is very great indeed. Such complex stack manipulation is best left to compilers.

7.4 Recursion

There are not a lot of highly original jokes associated with computer science. One of the few is 'Question: What's the dictionary definition of recursion?' 'Answer: See recursion.' An object is said to be defined *recursively* if it is defined in terms of itself. A procedure is said to be recursive if it calls itself.

At first sight, recursion sounds odd, if not ridiculous. Many programmers do not encounter recursion early in a computer science course. They first meet *iteration*, which might be said to be the *opposite* of recursion. The textbook example most commonly used to introduce recursion is the factorial. Factorial n is written n! and defined as:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1.$$

For example, 6! = 6x5x4x3x2x1 = 720. We can immediately write down an iterative procedure to evaluate n!.

```
Factorial_N = 1
I := N
REPEAT
    Factorial_N := Factorial * I
    I := I - 1
UNTIL I = 0
```

This procedure can readily be translated into 68000 code as:

```
*                           D0 contains initial value of N
*                           D1 contains the 16-bit result
*
        MOVE.W  #1,D1       Set initial value of the result to 1
Loop    MULU    D0,D1       REPEAT N_Factorial:= N_Factorial * I
        SUBQ.W  #1,D0               I := I - 1
        BNE     Loop        UNTIL I = 0
        RTS
```

This code provides a correct result, provided that it is less than 65,535. We now look at the recursive definition of factorial n. That is:

$n! = n \times (n-1)!$

This definition is recursive because n! is defined in terms of another factorial value; i.e., (n-1)!. Using the same definition we can evaluate (n-1)! from (n-1) x (n-2)!, and so on. When we reach 1!, we substitute the value 1! = 1, and then evaluate the expression for n!. The pseudocode construct of a recursive procedure to evaluate n! is:

```
Module Factorial(N)
                    IF N = 1 THEN Factorial(N) := 1
                             ELSE Factorial(N) := N*Factorial(N-1)
                    END_IF
End Factorial
```

The procedure `Factorial(N)` calls itself. It is, therefore, a recursive procedure. Let's walk through this procedure for n = 4.

1. The procedure is called initially with n = 4. The `IF` part yields a false value because N is not 1, and the `ELSE` part begins to calculate `4*Factorial(3)`.
2. The `ELSE` part calls `Factorial(3)` recursively. The `IF` part of the procedure yields a false value and the `ELSE` part begins to calculate the expression `3*Factorial(2)`.
3. The `ELSE` part calls `Factorial(2)`. Again, the `IF` part yields a false value and the `ELSE` part begins to calculate `2*Factorial(1)`.

4. The ELSE part calls `Factorial(1)`. Now, the IF part yields a true value and a return from `Factorial(1)` with the value 1 is made.

5. A return is made from `Factorial(1)` to `Factorial(2)`. The expression `2*Factorial(1)` is calculated, which is 2*1 = 2.

6. The return from `Factorial(2)` to `Factorial(3)` results in the evaluation of `3*Factorial(2)`, which is 3*2 = 6.

7. Finally, the return from `Factorial(3)` to `Factorial(4)` results in 4*6 = 24.

As you can see from this walkthrough, the arithmetic is performed after the procedure has been called recursively n times. In order to implement recursion, successive return addresses (and any workspace) are stored on the stack.

We can now write a recursive factorial procedure in assembly language.

```
*         Factorial(N)
*         On entry D0 holds N and on exit D0 holds N!
*
Factor    MOVE.W  D0,-(A7)      Push N on the stack
          SUBQ.W  #1,D0         N := N - 1
          BEQ     ExitFact      IF N = 1 THEN Factorial(N) := 1
          BSR     Factor                ELSE
          MULU    (A7)+,D0              Factorial(N) := N*Factorial(N-1)
          RTS
ExitFact  MOVE.W  (A7)+,D0      Clean up the stack
          RTS                   and return
```

This recursive procedure can be tested by running it with N = 4. The following output is from the cross-assembler. Note that we have included a section that sets up the stack and calls the recursive procedure.

```
Source file: FACT.X68
Assembled on: 92-11-27 at: 16:11:38
        by: X68K PC-1.9 Copyright (c) University of Teesside 1989,92
Defaults: ORG $0/FORMAT/OPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO

  1                     *       Factorial(N)
  2                     *       On entry D0 holds N and on exit D0 holds N!
  3                     *
  4  00000400                    ORG     $400
  5  00000400 4FF81000           LEA     $1000,A7      ;Set up the stack pointer
  6  00000404 203C00000004       MOVE.L  #4,D0
  7  0000040A 61000006           BSR     FACTOR        ;Evaluate 4!
  8  0000040E 4E722700           STOP    #$2700
  9                     *
 10  00000412 3F00        FACTOR: MOVE.W  D0,-(A7)      ;Push N on the stack
 11  00000414 5340               SUBQ.W  #1,D0         ;N := N - 1
```

```
12  00000416 67000008            BEQ      EXITFACT     ;IF N = 0 THEN Factorial_N := 1
13  0000041A 61F6                BSR      FACTOR       ;ELSE
14  0000041C C0DF                MULU     (A7)+,D0     ;Factorial_N:=N*Factorial_N(N-1)
15  0000041E 4E75                RTS
16  00000420 301F        EXITFACT: MOVE.W   (A7)+,D0     ;Clean up the stack
17  00000422 4E75                RTS                   ;and return
18                       *
19          00000400             END      $400

Lines: 19, Errors: 0, Warnings: 0.
```

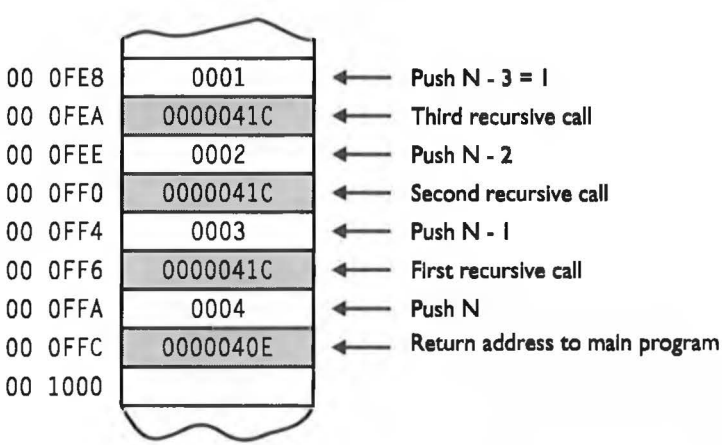
When the procedure is first called, register D0.W, which contains 4, is pushed on the stack and then decremented. The procedure is called again if the contents of D0 are not zero. This sequence is repeated and the stack looks like Figure 7.18. Note that we have drawn the stack in an unusual fashion, with alternate words and longwords. The longwords are the successive return addresses, and the words are the values pushed on the stack by the procedure.

We will use the Teesside simulator to walk through the program itself. After loading the program, we use the DF, display formatted register command, and then the trace command.

```
df

PC=000400 SR=2000 SS=00A00000 US=00000000          X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>LEA       $1000,SP
```

Figure 7.18 Growth of the stack during the calculation of 4!



```
>tr

PC=000404 SR=2000 SS=00001000 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001000 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.L    #4,D0

Trace>

PC=00040A SR=2000 SS=00001000 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001000 Z=0
D0=00000004 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BSR.L     $00000412
```

Here's where we call the procedure FACTOR for the first time.

```
Trace>

PC=000412 SR=2000 SS=00000FFC US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFC Z=0
D0=00000004 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.W    D0,-(SP)

Trace>

PC=000414 SR=2000 SS=00000FFA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFA Z=0
D0=00000004 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>SUBQ.W    #1,D0

Trace>

PC=000416 SR=2000 SS=00000FFA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFA Z=0
D0=00000003 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BEQ.L     $00000420
```

```
Trace>

PC=00041A SR=2000 SS=00000FFA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFA Z=0
D0=00000003 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BSR.S     $00000412
```

And here's our second call — this time it's recursive.

```
Trace>

PC=000412 SR=2000 SS=00000FF6 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF6 Z=0
D0=00000003 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.W    D0,-(SP)

Trace>

PC=000414 SR=2000 SS=00000FF4 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF4 Z=0
D0=00000003 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>SUBQ.W    #1,D0

Trace>

PC=000416 SR=2000 SS=00000FF4 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF4 Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BEQ.L     $00000420

Trace>

PC=00041A SR=2000 SS=00000FF4 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF4 Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BSR.S     $00000412
```

This is the third call.

```
Trace>

PC=000412 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF0 Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.W    D0,-(SP)

Trace>

PC=000414 SR=2000 SS=00000FEE US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEE Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>SUBQ.W    #1,D0

Trace>

PC=000416 SR=2000 SS=00000FEE US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEE Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BEQ.L     $00000420

Trace>
PC=00041A SR=2000 SS=00000FEE US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEE Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BSR.S      $00000412
```

This is the fourth and final call to the recursive procedure.

```
Trace>
PC=000412 SR=2000 SS=00000FEA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEA Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.W    D0,-(SP)
```

```
Trace>

PC=000414 SR=2000 SS=00000FE8 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FE8 Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>SUBQ.W    #1,D0

Trace>

PC=000416 SR=2004 SS=00000FE8 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FE8 Z=1
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>BEQ.L      $00000420
```

Let's have a look at the contents of the stack by using the memory display function.

```
>MD FE8

000FE8  00 01 00 00 04 1C 00 02 00 00 04 1C 00 03 00 00
000FF8  04 1C 00 04 00 00 04 0E 00 00 00 00 00 00 00 00.
```

This memory map is the same as that of Figure 7.18, and shows the alternating sequence of data values and return addresses.

```
>TR

PC=000420 SR=2004 SS=00000FE8 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FE8 Z=1
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MOVE.W     (SP)+,D0
Trace>


PC=000422 SR=2000 SS=00000FEA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEA Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>RTS
```

Now we begin the returns and unwind the stack.

```
Trace>
PC=00041C SR=2000 SS=00000FEE US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FEE Z=0
D0=00000001 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MULU      (SP)+,D0
```

We pop an element off the stack and perform a multiplication.

```
>TR

PC=00041E SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF0 Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>RTS

Trace>
PC=00041C SR=2000 SS=00000FF4 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF4 Z=0
D0=00000002 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MULU      (SP)+,D0
```

Here, we perform the next multiplication.

```
>TR
PC=00041E SR=2000 SS=00000FF6 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FF6 Z=0
D0=00000006 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>RTS

Trace>
PC=00041C SR=2000 SS=00000FFA US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFA Z=0
D0=00000006 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
------>MULU      (SP)+,D0
```

This is the last multiplication. After this, the contents of data register D0 should be 4!, i.e., 24 or $18.

```
Trace>

PC=00041E SR=2000 SS=00000FFC US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00000FFC Z=0
D0=00000018 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
———>RTS

Trace>

PC=00040E SR=2000 SS=00001000 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001000 Z=0
D0=00000018 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
———>STOP      #9984
```

This text does not use recursive techniques elsewhere. We have introduced recursion here because it makes use of the 68000's stack mechanism. Moreover, if the recursive procedure requires local workspace, the 68000's LINK and UNLK instructions can be used to create a stack frame for each recursive call.

Recursive algorithms are found in many branches of computer science, particularly data structures. Many algorithms that are expressed iteratively can also be expressed in a recursive form. For example, consider the algorithm to detect whether a string is a palindrome or not (i.e., the string reads the same left to right as right to left, e.g., hannah). The iterative approach leads to an algorithm like:

```
Module Palindrome
        Set Found := true
        Set LeftPointer to leftmost character
        Set RightPointer to rightmost character
        REPEAT
                IF Char[LeftPointer] ≠ Char[RightPointer]
                    THEN
                    Found := false
                    EXIT
                END_IF
                LeftPointer := LeftPointer + 1
                RightPointer := RightPointer - 1
        UNTIL LeftPointer = RightPointer OR LeftPointer+1 = RightPointer
End Palindrome
```

We can also use a recursive algorithm to detect a palindrome. A string is a palindrome if the outermost two characters are equal and the remainder of the string is a palindrome.

```
Initialize Palindrome
        Left := pointer to start of string
        Right := pointer to end of string
Module Palindrome
        IF (Left < Right)
           THEN RETURN([M(LEFT)]=[M(RIGHT)]) AND Palindrome(Left+1,Right-1)
           ELSE RETURN(TRUE)
        END_IF
End Palindrome
```

The recursive algorithm says that a word is a palindrome if its outermost characters are equal and the remaining characters are a palindrome. One of my students, Roger Allen, wrote the following 68000 recursive procedure to check whether a string is a palindrome. We have added a function call and a mechanism to display the result (using the Teesside 68000 simulator).

```
Source file: PALINDRM.X68
Assembled on: 93-02-02 at: 21:03:08
        by: X68K PC-2.0 Copyright (c) University of Teesside 1989,93
Defaults: ORG $0/FORMAT/CPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO
   1  00000400                      ORG     $400
   2  00000400 4FF81000             LEA     $1000,A7          ;Set up the stack
   3  00000404 487900002000         PEA     PAL_STRT          ;Push the start address of the string
   4  0000040A 487900000464         PEA     PAL_END           ;Push the end address of the string + 1
   5  00000410 61000024             BSR     PALINDRM          ;Test for a palindrome
   6  00000414 4FEF0008             LEA     8(A7),A7          ;Clean up the stack by removing addresses
   7  00000418 43F900000464         LEA     NOT_PAL,A1        ;Let's say it wasn't a palindrome
   8  0000041E 4A40                 TST.W   D0                ;If it wasn't go and say so
   9  00000420 66000008             BNE     P_STRING
  10  00000424 43F900000470         LEA     IS_PAL,A1         ;If it was then say so
  11  0000042A 1219         P_STRING: MOVE.B  (A1)+,D1          ;Pick up the size of the message
  12  0000042C 103C0000             MOVE.B  #0,D0             ;D0 = 0 to print a string
  13  00000430 4E4F                 TRAP    #15               ;Call the O/S
  14  00000432 4E722700             STOP    #$2700
  15                        *
  16  00000436 4240         PALINDRM: CLR.W   D0                ;Clear D0 to assume palindrome
  17  00000438 48E740C0             MOVEM.L D1/A0-A1,-(A7)    ;Save working registers
  18  0000043C 4CEF03000010         MOVEM.L 16(A7),A0-A1      ;Load the start/end address in A0, A1
  19  00000442 B3C8                 CMPA.L  A0,A1             ;Test for "middle" or palindrome
  20  00000444 6C18                 BGE.S   PAL_EXIT          ;If at the middle, we can exit
  21  00000446 1219                 MOVE.B  (A1)+,D1          ;Otherwise, pick up a character
  22  00000448 41E8FFFF             LEA     -1(A0),A0         ;and move the other pointer back
```

```
 23 0000044C 48E700C0            MOVEM.L  A0-A1,-(A7)        ;Save the two pointers
 24 00000450 61E4                BSR      PALINDRM           ;and see if what's left is a palindrome
 25 00000452 4FEF0008            LEA      8(A7),A7           ;Remove the parameters after the subroutine
 26 00000456 B218                CMP.B    (A0)+,D1           ;Compare left character with right character
 27 00000458 6704                BEQ.S    PAL_EXIT           ;If same then all is well
 28 0000045A 303CFFFF            MOVE.W   #-1,D0             ;otherwise set not_palindrome flag
 29 0000045E 4CDF0302    PAL_EXIT: MOVEM.L  (A7)+,D1/A0-A1    ;Restore registers before leaving
 30 00000462 4E75                RTS
 31                      *
 32         00000464     PAL_END: EQU      *
 33 00000464 180A0D497420 NOT_PAL: DC.B     24,$0A,$0D,'It is not a palindrome'
            6973206E6F74
            20612070616C
            696E64726F6D
            65
 34 0000047D 140A0D497420 IS_PAL:  DC.B     20,$0A,$0D,'It is a palindrome'
            697320612070
            616C696E6472
            6F6D65
 35 00002000                      ORG      $2000              ;Let's put the palindrome somewhere memorable
 36 00002000 41424241    PAL_STRT: DC.B     'ABBA'             ;A test palindrome
 37         00000400              END      $400

Lines: 37, Errors: 0, Warnings: 0.
```

Note that the Teesside 68000 simulator's TRAP #15 instruction is used to output a string. The parameter 0 in data register D0 tells the trap handler to output the string pointed at by address register A1 whose length is given by the contents of data register D1.

Summary

- A subroutine is called by means of either a BSR or a JSR instruction. A return from the subroutine to the instruction immediately after the calling point (i.e., after the BSR or JSR) is made by means of an RTS instruction.
- You should leave a subroutine at a single point (i.e., at the RTS instruction). You should never use a BRA, JMP, or Bcc instruction to jump out of a subroutine into another subroutine or the main program.
- The 68000's stack pointed at by A7 automatically handles subroutine return addresses. Subroutines may be nested; that is, one subroutine may call another subroutine. You must not corrupt the stack pointed at by A7; otherwise, the subroutine return mechanism will fail and the system will crash.

- The BSR instruction employs a relative address. Fortunately, the address is calculated automatically by the assembler. If you know that the subroutine is situated within 128 bytes of the calling point, you can use the short form of the branch — BSR.S. If you use a JSR instruction, the address is *absolute* and the code is not position-independent.
- Although subroutines are primarily used to avoid rewriting bits of frequently used code, they can also be used to facilitate top-down design and to enhance program readability.
- Subroutines usually operate on data. Data can be passed to and from a subroutine by putting it in registers, memory, or on the stack. The stack is the preferred transfer mechanism.
- Passing data to and from a subroutine via the stack aids the writing of reentrant subroutines.
- Data may be passed to a subroutine by means of its value or by its address. When it is passed by value, a copy of the parameter is transferred. When it is passed by address (or by reference), the address of the parameter is passed to the subroutine. You can use the push effective address instruction, PEA, to push an address on the stack before calling a subroutine.
- A subroutine frequently requires local variables. These variables are intermediate values required by the subroutine. Local values are not accessed from outside the subroutine and are discarded after the subroutine is exited. Local variables are normally created on the top of the stack.
- The 68000 includes a LINK and UNLK instruction pair that create a stack frame for local variables and collapse the stack frame, respectively.

Problems

1. If the BSR and RTS instructions did not exist, what two instruction sequences would you use to synthesize these instructions?

2. Why is the BSR instruction generally preferred to the JSR?

3. When does a JSR instruction prove invaluable?

4. What is the difference between passing a parameter to a subroutine by value and passing it by reference?

5. What is a *local variable* and why should it be local?

6. Explain how the 68000's LINK and UNLK instructions help to implement a stack frame.

7. Will the following code work?

```
        BSR Sub1          Call subroutine 1
        .
Sub1    BSR Sub2          Call subroutine 2
        .
        .
        RTS               Return from subroutine 1
Sub2    .
        .
        LEA (4,A7),A7     Step past the return address
        RTS               Return directly to the main program
```

8. Why does the LINK instruction invariably take a *negative* literal as a parameter?

9. Write a recursive subroutine to calculate the nth value in the Fibonacci sequence: 1, 1, 3, 5, 8, 13. The first two terms in the sequence are 1, 1, and the value of any other term is given by the sum of the two immediately preceding terms.