

## MODULAR DESIGN

One of the most significant features of modern electronic systems is their modularity. A complex circuit is invariably decomposed into several less complex subsystems, called *modules*. The advantage of such an approach is that the modules can be designed and tested independently of the parent system. A module made by one manufacturer can be replaced by one from another manufacturer, as long as the two modules have the same interface to the rest of the system and are functionally equivalent. In the world of software, modularity is an attempt to treat software like hardware by creating software elements called modules. The principal requirement of a software module is that it is concerned only with a single, logically coherent task. For example,  $\sin x$  can be considered as a module—it takes an input,  $x$ , and returns a result (i.e., the sine of  $x$ ). It performs a single function and is *logically coherent* because it carries out only those actions necessary to calculate  $\sin x$ . A module is not just an arbitrary segment of a larger program. An old story is told about the dark ages of programming when a programmer was asked to introduce modularity into his programs. He took a ruler and drew a line after every 75 statements. Those lines were his modules.

A software module is analogous to a hardware element because it has a number of inputs and a number of outputs and can be "plugged into a system." The module processes incoming information to yield one or more outputs. The internal operation of the module is both irrelevant to and hidden from its user, just as the transport properties of electrons in a semiconductor are irrelevant to the programmer of a microprocessor. The advantages of software modules are broadly the same as those of hardware modules. Modules can be tested and verified independently of the parent system and they can be supplied by manufacturers who know little or nothing about the parent system. Figure 1 illustrates the concept of a module.

The disk file monitor (DFM), introduced when we were describing top-down design, can be regarded as such a module. It is entered at one point, with details of the action to be carried out passed as parameters to the module. A return is made from the module to its calling point with parameters passed back as appropriate. It is no more reasonable to enter a module at some other point than it is to drill a hole in a floppy disk controller and to attach a wire to the silicon chip. Indeed, the whole point of modular design is to make the design, production, and testing of hardware and software almost identical; for example, the module for  $\sin x$  takes an input and generates an output,  $\sin x$ , together with relevant status information. Passing information to the inner working of  $\sin x$  or getting information from within  $\sin x$  is quite meaningless to higher level modules.

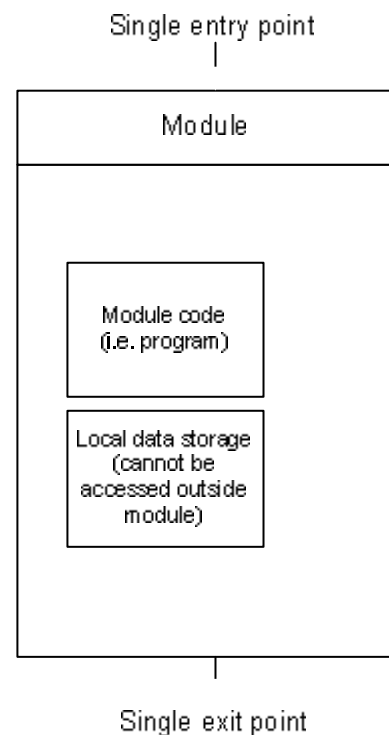


Figure 1

### Module Coupling and Module Strength

A module is sometimes described in terms of two properties: *coupling* and *strength*. Coupling, as its name suggests, indicates how information is shared between one module

and other modules. Tightly coupled modules share common data areas. This situation is regarded as undesirable because it makes it harder to isolate the action of one module from the action of other modules. We can liken this to a person with pneumonia and liver disease: If the person dies, how do you know which illness was the cause?

A module exhibiting loose coupling has data that is entirely independent of other modules. Any data it accesses is strictly private to the module and does not interact with other modules. This condition makes it easier to debug systems because the programmer knows that data associated with a module is not modified by other modules. Therefore, errors can be speedily localized.

The strength of a module is a measure of its "modularity." In the story about the programmer who divided a program up into units of 75 lines, the resulting modularity is very weak because the modules are entirely arbitrary and functions are distributed between modules in a random fashion.

A strong module is one that performs a single task; for example, the  $\sin x$  module discussed previously is strong because it does nothing more than calculate the sine of an input  $x$ . A weak module performs more than one task; for example, a module dealing with input from the keyboard, a disk drive, and a serial interface is relatively weak. It is a module in the sense that it performs logically related functions (i.e., input), but it is weak because several functions have been grouped together in one module.

The strength of a module is important, because strong modules are relatively easy to test as they perform only one function. Equally, a strong module from one supplier is not difficult to replace with a functionally equivalent module from another supplier. A weak module carries out several tasks, making it more difficult to test.

## **Passing Parameters to Modules**

A module implemented as a subroutine must transfer information between the subroutine and the program calling it. The only exception occurs when a subroutine is called to trigger some event; for example, a subroutine can be designed to ring a bell or to sound an alarm. Simply calling the subroutine causes the predetermined action to take place. No communication exists between the subroutine and the program calling it.

Consider now 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 actions and is inherently device dependent. Consequently, input and output transactions are frequently implemented by calling the appropriate input or output subroutine.

Suppose a program invokes the subroutine PUT\_CHAR in order to display a single character on the CRT terminal. The character is printed by executing the instruction BSR PUT\_CHAR and the whole process takes place automatically, due to the processor's stack mechanism, which takes care of the return address. In this example, the program has to transfer just one item of information to the subroutine, namely, the character to be displayed.

In this case, where the program passes only a single character 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 register D7 and register D0 is used to carry the character to the subroutine, the following code may be written:

```
MOVE .B    D7,DO
BSR        PUT_CHAR
```

The preceding method of transferring data between a program and a subroutine by means of one or more registers is popular and is frequently used where the quantity of data to be transferred is very small or the processor is well endowed with registers. The advantage of this method is that it permits both position-independent code and reentrancy. Position independence is guaranteed because no absolute memory location is involved in the transfer of data and reentrancy is possible as long as the reuse of the subroutine saves the registers employed to transfer the data before they are reused.

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 longwords 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.

### **Mechanisms for Parameter Passing**

At this point it is worthwhile mentioning a number of concepts relevant to parameter passing. We need to distinguish between the ways in which parameters are passed and their implementation. The two basic ways of passing parameters are transfer by value and transfer by reference. In the former, the actual parameter is transferred; whereas in the latter, the address of the parameter is passed between program and subroutine. This distinction 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. The clone never returns from the subroutine.

When a parameter is passed by address (i.e., by reference), the subroutine receives a pointer to the parameter. In this case, only one copy of the parameter exists, 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 modifies the parameter globally and not only within the subroutine.

The mechanism by which information is passed to subroutines 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 the calling program and subroutine, with one placing data in the mailbox and the other emptying it. However, the stack mechanism offers the most convenient method of transferring information between a subroutine and its calling program.

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

Suppose a subroutine is written to search a region of memory containing text for the first occurrence of a particular sequence of characters. The sequence we are looking for 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 to be searched and the string to be used in the matching process. Figure 2 illustrates this problem.

The information required by the subroutine is the four addresses, \$00 1000, \$00 100D, \$00 1100, and \$00 1103. Note that we are passing the parameters by reference, because the subroutine receives their addresses; we are not passing the actual parameters (i.e., the text strings) themselves. Eventually, the subroutine returns the value \$00 1007. Although it is possible to transfer all these addresses via registers, an alternative technique is to assemble the four parameters into a block somewhere in memory and then pass the address of this block. Figure 3 shows how the block is arranged.

The only information required by the subroutine is the address \$00 2000, 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, 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 the text and of the string to be matched.

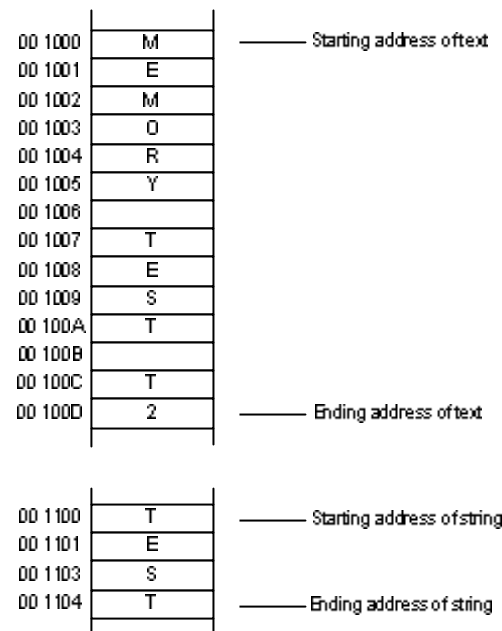
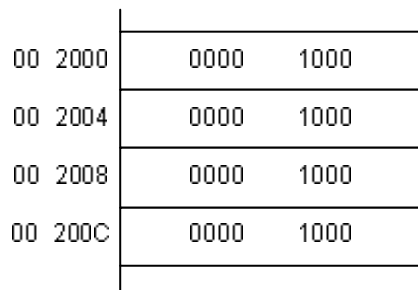


Figure 2



All that the subroutine needs is the starting address of the block of parameters, that is, \$00 2000

Figure 3

```

.
.           Body of subroutine
.
MOVEM.L    (SP)+,A3-A6   Restore address registers from stack
RTS

```

In the preceding example, A0 is used as a pointer to the parameter block to obtain the four addresses required by the subroutine. We have passed a parameter by its address (i.e., the address of the parameter block) and have transferred this address in a register (A0). Unfortunately, this method of storing information in a block whose location is fixed in memory cannot always be employed because such a program cannot be used reentrantly. Clearly, if the parameters are stored in a block with a fixed address (i.e., static memory location), any attempt to interrupt the subroutine and to reuse it must result in the new values overwriting the parameters. 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 onto the stack on top of the old parameters. When the interrupting program has used the subroutine, a return from interrupt is made with the stack in the same condition it was in immediately prior to the interrupt.

### Stack and Parameter Passing

The stack is useful not only for storing subroutine return addresses in such a way that subroutines may call further subroutines but also for transferring information to and from a subroutine. All that needs to be done is to push the parameters (or their addresses) on the stack before calling the subroutine. The following program fragment shows how this is done for our string-matching algorithm. In this example, we transfer all parameters by reference.

```

PEA        TEXT_START    Push text starting address
PEA        TEXT_END      Push text ending address
PEA        STRING_START  Push string starting address
PEA        STRING_END    Push string ending address
BSR        STRING_MATCH  Call subroutine for matching
LEA        16(SP),SP     Adjust stack pointer
.
.
.
STRING_MATCH
LEA        4(SP),A0      Put pointer to parameters in A0
MOVEM.L    A3-A6,-(SP)   Save working registers on stack
MOVEM.L    (A0)+,A3-A6  Get parameters off stack
.
.           Body of subroutine
.
MOVEM.L    (SP)+,A3-A6  Restore working registers
RTS

```

In this example, the instruction PEA (push effective address) pushes the effective address of the operands onto the stack, but instead we could have used MOVE.L #TEXT\_START,-(SP).

Figure 4 illustrates the state of the stack during the execution of the preceding code. The first instruction in the subroutine, `LEA 4(SP),A0`, loads A0 with the starting address of the last parameter pushed onto the stack. We must add 4 to the stack pointer because the return address is at the top of the stack. The instruction `MOVEM.L (A0)+,A3-A6` pulls the four addresses off the stack pointed at by A0 and deposits them in address registers A3 to A6 for use as required. Note that these parameters are left on the stack pointed at by SP after a return from the subroutine is executed.

In the calling program, the instruction `LEA 16(SP),SP` is executed after a return from the subroutine has been made. This instruction 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 onto it.

As we have already pointed out, passing parameters on the stack facilitates position-independent code and permits reentrant programming. If a subroutine is interrupted, the stack builds upward and information currently on the stack is not overwritten.

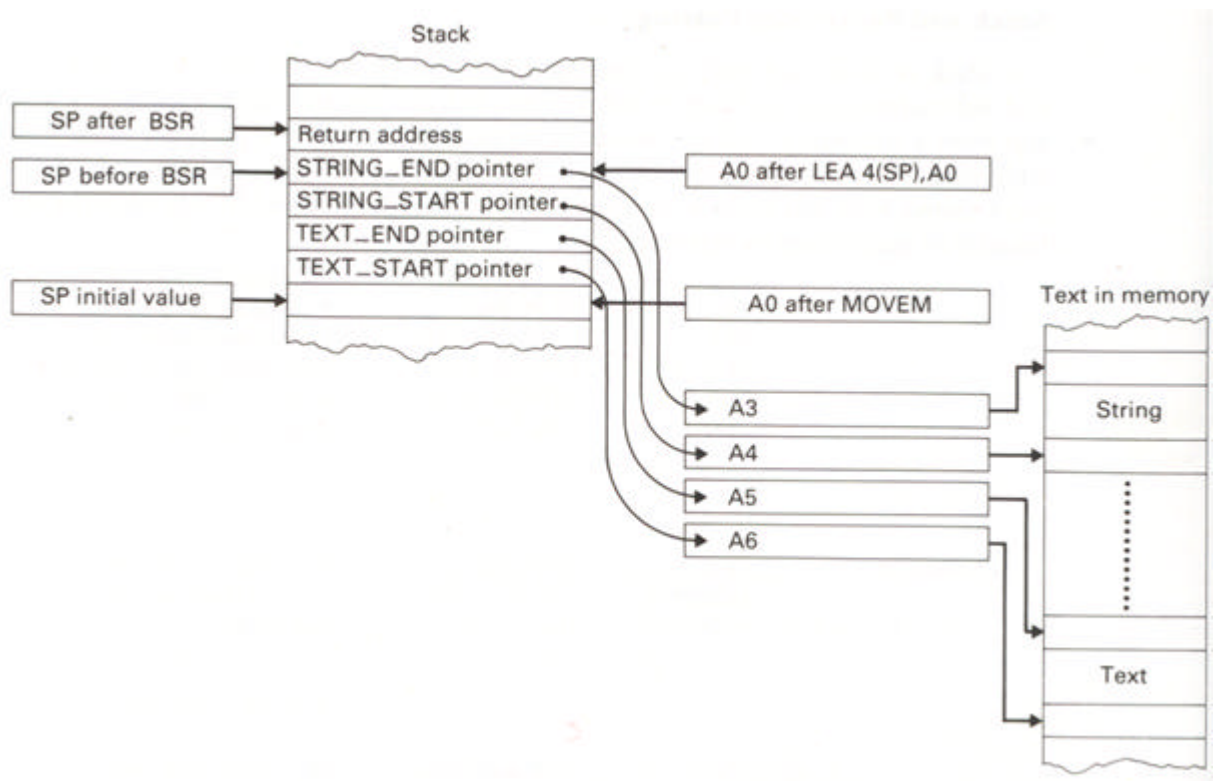


Figure 4

## Stack and Local Variables

In addition to the parameters passed between itself and the calling program, most subroutines need a certain amount of *local workspace* for their temporary variables. The word local means that the workspace is private to the subroutine and is never accessed by the calling program or other subroutines. In a few circumstances, it is quite feasible to allocate a region of the system's memory space to a subroutine requiring a work area at the time the program is written. The programmer simply reserves fixed locations for the subroutine's variables in a process called *static allocation*. This process is entirely satisfactory for subroutines that are not going to be used reentrantly or recursively.

If a subroutine is to be made reentrant or is to be used recursively, its local variables must be bound up not only with the subroutine itself but with the occasion of its use. In other words, each time the subroutine is called, a new workspace must be assigned to it. Once again, the stack provides a convenient mechanism for implementing the dynamic allocation of workspace.

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. Below Figure 5 illustrates a stack frame.

Figure shows how a stack frame is created merely by moving the stack pointer up by  $d$  locations. This can be done at the start of a subroutine. Note that the 68000 stack grows toward the low end of memory and therefore the stack pointer is decremented. Reserving 200 bytes of memory is achieved by `LEA -200(SP),SP`. Once the stack frame has been created, local variables are accessed by any addressing mode that uses A7 as a pointer. Before a return from the subroutine is made, the stack frame must be collapsed by `LEA 200(SP),SP`.

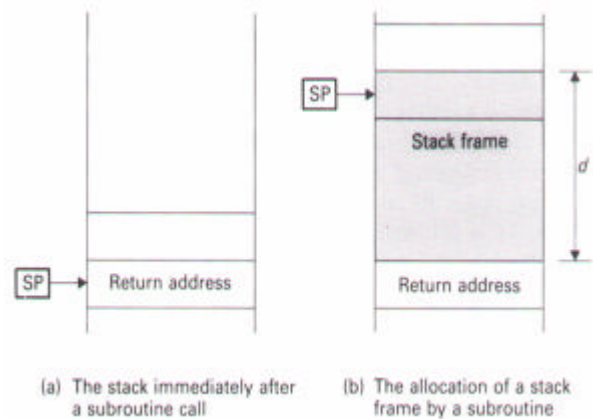


Figure 5

### Link and Unlink Instructions

The simple scheme of the above figure has been mechanized in the 68000 by a complementary pair of instructions, LINK and UNLK. These are relatively complex in terms of their detailed application 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 reentrant.

If the stack frame storage mechanism is to support reentrant programming, a new stack frame must be reserved each time a subroutine is called. As each successive stack frame is of a (possibly) variable size, its length must be preserved somewhere. 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, the registers that must be saved are first pushed on the stack and then the LINK instruction is executed to create the stack frame belonging to *this* subroutine. The following code achieves the desired effect:

```

MOVEM.L    D0-D7/A3-A6,-SP)  Save working registers on the stack
LINK      A1,#-64             Allocate 64 bytes (16 longwords)
                                   of storage on this stack frame

```

In this example, all working registers are saved on the stack, the temporary storage allocation is 64 bytes, and address register A1 is used by LINK. Note that the minus sign is needed because the stack grows toward low memory. The action executed by `LINK A1,#-64` is defined in RTL terms as follows:

```

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

```

Note that the old contents of A1 are not destroyed by this action; they are pushed on the stack. Similarly, the old value of the stack pointer is preserved in A1. In other words, the LINK destroys no information and, therefore, it is possible to undo the effect of a LINK at some later time. The state of the stack before a subroutine call, after the call and the MOVEM operation, and after a LINK instruction is given in Figure 6.

After stage (c) in Figure 6, the programmer can use the stack frame area as required; for example, LEA (SP),A2 loads the first free address of the frame into address register A2 and then A2 can be used as an offset in all references to the stack frame. Equally, all data references can be made with respect to the stack pointer or to A1. Life becomes interesting when the subroutine calls another subroutine with its own stack frame. Figure 7 depicts this situation.

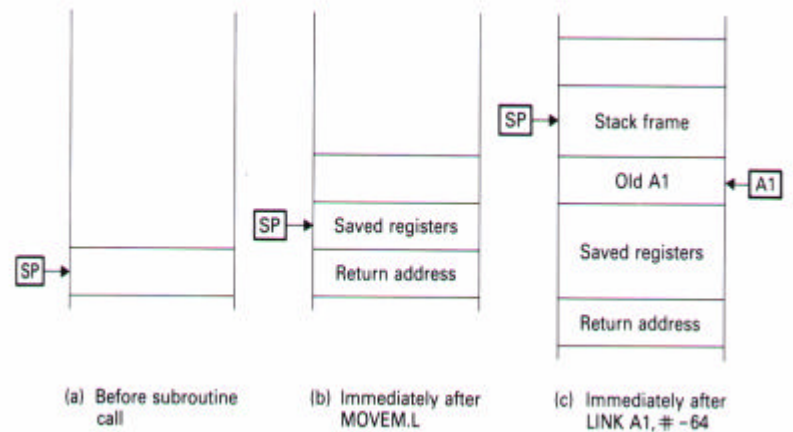


Figure 6

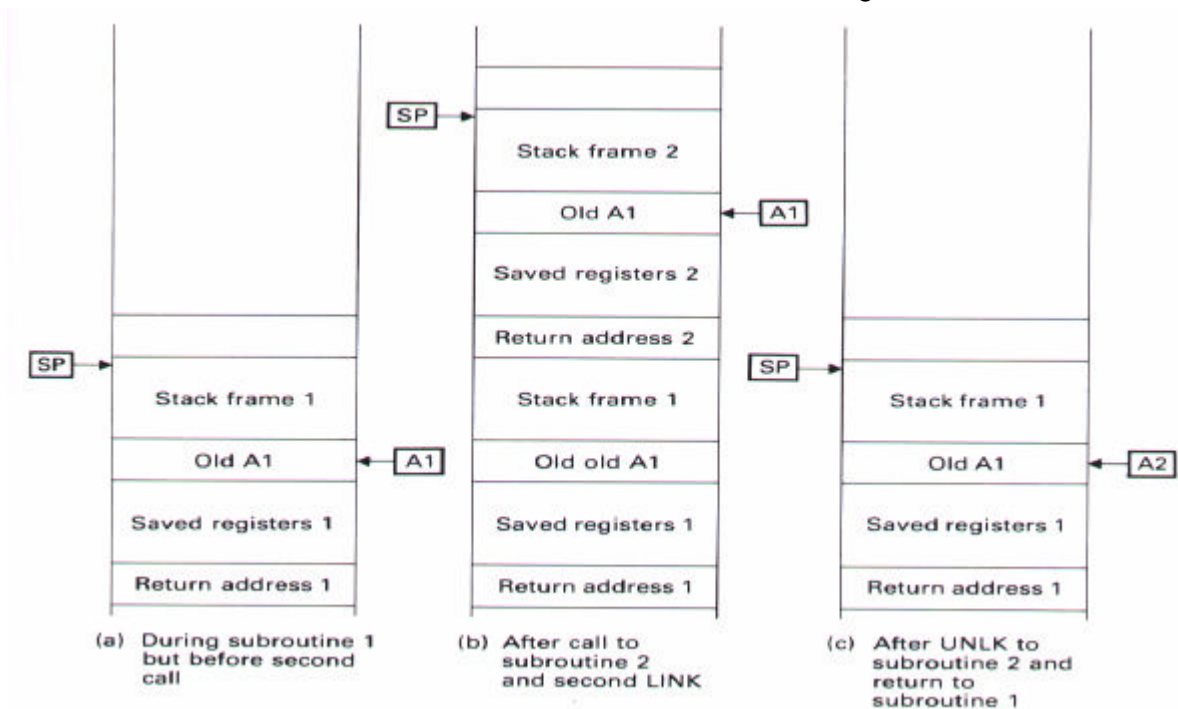


Figure 7

In figure 7 at the initial state (a), subroutine 1 is being executed and has its own stack frame, labeled "Stack frame 1." Let us suppose that a second subroutine is invoked and another LINK A1,#- d executed. The second stack frame can be of any size and is not necessarily related to stack frame 1. This situation is illustrated in Figure 7 at state (b). Register A1 now contains the value of the stack pointer immediately before the creation of stack frame 2; that is, A1 points to the location of the "old A1." The "old A1" is, of course, a pointer to the base of subroutine 1's stack frame, which holds the "old old A1." Because A1 points to the base (i.e., highest address) of the stack frame, all local variables can be accessed by register indirect addressing with displacement, where A1 is the register used to access them.

The next step is to show how an orderly return from subroutine 2 to subroutine 1 can be made. At the end of subroutine 2, the following sequence is executed:

```

UNLK      A1                Deallocate subroutine 2's stack frame
MOVEM.L   (SP)+,D0-D7/A3-A6 Restore working registers from the stack
RTS                          Return to calling point

```

The RTL definition of UNLK A1 is

```

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

```

We can put this definition more clearly by stating that the stack point is first loaded with the contents of address register A1. Remember that A1 contains the value of the stack pointer just before stack frame 2 was created. In this way, stack frame 2 collapses. The next step is to pop the top item off the stack and place it in A1. This process has two effects: it returns both the stack and the contents of A1 to the points where they were located before LINK was executed.

Following the UNLK, the working registers can be pulled off the stack and a return to subroutine 1 made. The execution of subroutine 1 continues from the point at which it left off.

The key to LINK and UNLK is the use of A1 to hold the base of the stack frame and the storage of the *previous* contents of A1 below the base of the stack frame. The LINK and UNLK instructions have been included to support recursive procedures. A recursive procedure is one that calls itself.

The frame pointer is important because it points to the region of the stack allocated to the current procedure. All parameters passed to and from the local workspace can be accessed by means of the frame pointer. Moreover, the frame pointer can be used to access any local variables created by the procedure.

Consider the following example. A subroutine requires three parameters. Two parameters are called by *value* (P and Q) and one is called by *reference* (R). In this example, we shall use the subroutine to calculate  $R = (P^2 + Q^2) / (P^2 - Q^2)$ . 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 D6 and A6 is not as good as ensuring that D6 and A6 are not corrupted by the subroutine. We use the LINK instruction to create a two-longword stack frame (which the subroutine can use as temporary storage).

```

MOVE.W   D0,-(SP)      Push value of P on stack
MOVE.W   D1,-(SP)      Push value of Q on stack
PEA      R              Push reference to R on stack
BSR      CALC           Call the subroutine
LEA      8(SP),SP       Clean up stack by removing P, Q,
.
.
.
CALC
MOVEM.L  D6/A6,-(SP)   Save working registers on the stack
LINK     A0,#-8         Establish 2 longword frame for the stack
MOVE.W   22(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.L   D6,-8(A0)     Save P2 on the stack frame - twice
MOVE.W   20(A0),D6     Now get Q from the stack
MULU     D6,D6          Calculate Q2
ADD.L    D6,-4(A0)     Store P2 + Q2 on the stack frame
SUB.L    D6,-8(A0)     Store P2 - Q2 on the stack frame
MOVE.L   -4(A0),D6     Get P2 + Q2
DIVU     -6(A0),D6     Calculate (P2 + Q2) / (P2 - Q2)
LEA      16(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
UNLK     A0             Collapse the stack frame
MOVEM.L  (SP)+,D6/A6   Restore working registers
RTS

```

Figure 8 illustrates the structure of the stack corresponding to the problem. Before the subroutine is called, the two `MOVE.W <register>,-(SP)` instructions push the parameters P and Q onto the stack. The following instruction, `PEA R`, pushes the address of R onto the stack. Note that P and Q each take up a word and R takes up a longword because it is an address rather than the value of R.

After calling the subroutine, the return address is pushed on the stack. We also save the working registers, D6 and A6 which take up a total of two longwords on the stack above the return address.

The `LINK A0,#-8` instruction pushes the old value of A0 (the frame pointer) on the stack and moves the stack pointer by 8 bytes. A0 is loaded with the value of the stack pointer immediately before it was moved 8 bytes. That is, A0 is the current frame pointer and is now pointing at the base of the stack frame. We can access all stack values by reference to A0 (we can also use the SP as a reference point). It is better to use A0 (i.e., the FP) than the stack pointer to refer to items in the stack frame, since A0 will be constant for the duration of this procedure, whereas the stack pointer may be modified.

At this stage the work of the subroutine can be carried out and parameters can be accessed from the stack. We get P and Q as values from the stack and then access R indirectly by reading its address from the stack. We access the value of P by `MOVE.W 22(A0),D6` because A0 points to the base of the stack frame and we have to step past the old A0 (4 bytes), the saved registers (8 bytes), the return address (4 bytes), and R and Q (4 and 2 bytes) to access P itself. The total offset is 22 bytes. In this case we calculate the value of  $P^2$  and store this longword at address  $-4(A0)$  the temporary storage created by the `LINK A0,-8` instruction. We do this to demonstrate how the stack frame's space can be used to hold temporary variables. Note that the division instruction, `DIVU -66(A0),D6`, has an effective address of  $-6(A0)$ . This corresponds to the low order word of  $P^2 - Q^2$  on the stack since the `DIVU` instruction takes a 16-bit source.

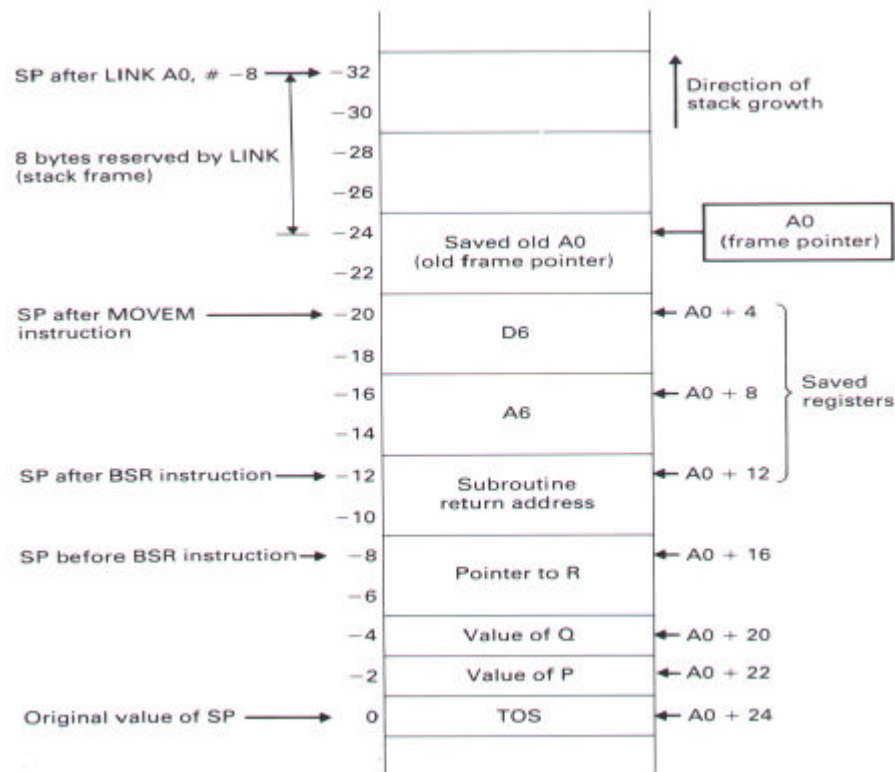


Figure 8

Accessing R, which is passed by reference, is more complex. The instruction `LEA 16(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 the `UNLK A0` instruction collapses the stack frame, discards the current frame pointer and restores the FP to its value immediately before the subroutine was called. Finally, the saved registers are restored from the stack and a return from subroutine is executed. In this example, the stack is in the state it was in before the subroutine call. However, since there are three parameters, taking a total of 8 bytes, left on the stack, we must (should?) clean up the stack by executing an `LEA 8(SP),SP`.

This example should demonstrate three things. The first is how the `LINK` and `UNLK` instructions work. The second is how the stack can be used to pass parameters both by reference and by value. The third is the inherent difficulty in assembly language

programming! For example, in order to reference parameter P we had to execute `MOVE.W 22(A0),D6` 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 mistake in dealing with the stack in programs such as this is very great indeed. Such complex manipulation is best left to compilers.

NOTE: This document on LINK and UNLK instructions of 68000 is taken from the book *Microprocessor Systems Design*, by Alan Clements.