Shared Data File (Workspace Registers) Linkage Techniques

The simplest subroutine call procedure uses the same set of processor registers and data addresses so that a shared data set occurs. This has the advantage of automatic parameter passing from the calling to the called procedure, but it has the disadvantage of limiting what the called procedure may do with the existing data so as not to destroy information vital to the calling procedure. In the 9900 family, the Branch and Link (BL) subroutine branch saves the return value for the program counter in register 11 of the workspace registers and both called and calling procedures use the same workspace. Most of the examples of the previous chapter used the BL call for simplicity. However, if a BL called subroutine must, in turn, call another subroutine with a BL instruction, the return linkage in register 11 is destroyed by the nested call. This problem was avoided in the previous chapter by saving the return linkage in another register prior to the second BL Instruction. Typically, in a subroutine that contains a BL instruction, the contents of register 11 (the primary return linkage) are saved in register 13 prior to the execution of the BL instruction. Then, register 11 is used for the secondary (nested subroutine) return linkage, and register 13 is used for the primary return linkage. The return to the original calling procedure is performed with a B *13 instruction instead of the usual B *11 instruction. This is an example of a more general approach of saving return linkages in a return stack.

Return Linkage Stacks

The basic concept is shown in *Figure 6.10*. As each subroutine is entered, the content of register 11 is saved in an area of RAM whose address is contained in a stack pointer. (Assume register 10 is assigned the stack pointer function.) The stack pointer is then decremented by two to move to the address of the next empty word location in the stack. This is called *pushing* the return linkage to the stack. Thus, to push register 11 to the stack requires

```
MOV   11,*10    Save R11 contents in stack
DECT  10        Decrement Stack Pointer
```

This is used in any subroutine that calls other subroutines with the BL procedure. Pushing the return linkage to the stack is accomplished by the first two instructions in each subroutine. At the last of each subroutine, before the B *11 return instruction occurs, the contents of register 11 must be restored from the stack by *popping* the stack. This requires incrementing the stack pointer (R10) to get back to a location holding a link address, and then moving the contents of the memory location addressed by register 10 to register 11:

```
INCT  10
MOV   *10,11
B     *11
```

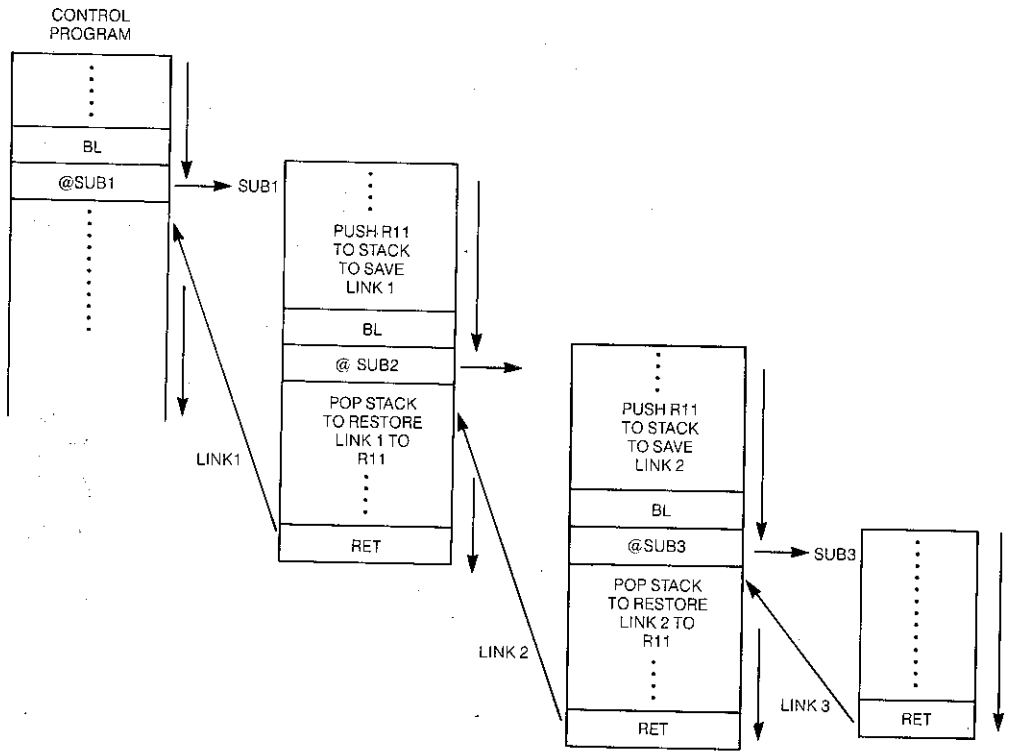

Figure 6.10 *Use of a Stack to Save Subroutine Return Linkages*

The stack is a last-in-first-out or LIFO memory organization and can be used for saving data or program linkage. Many microprocessors save program linkage automatically this way. Such units have a stack pointer register on the processor chip. In such units the return from a subroutine is effected by popping the stack to the program counter. The number of levels to which subroutines and/or data storage can occur in a nested fashion depends on the number of words of memory allocated to the stack function. The stack can be thought of as a temporary memory for saving important data or linkage information so that subroutine data storage requirements will not destroy information vital to the calling subprogram.

Independent Register File (Workspace Register) Linkage Techniques

Another approach to avoiding the subroutines damaging main or calling program information is to provide the subroutine with its own workspace that is totally independent of the calling program's workspace. This is possible in the 9900 family, since the BLWP subroutine call allows the programmer to establish a different workspace pointer value for the subroutine than was used by the calling sequence. The programmer also has the option of using the same workspace pointer for a shared workspace situation. Alternatively, by choosing a workspace pointer value that is within 32 of the calling procedure workspace pointer value, the subroutine workspace will overlap the calling procedure workspace for a partially shared workspace situation.

When an independent or overlapping workspace is used with the BLWP subroutine call in the 9900 family, the calling program registers are available to the called subroutine because register 13 of the called subprogram's workspace contains the workspace pointer value of the calling procedure. Thus, by referring to the new register 13, the address of the old register 0 is available. By using indexed addressing with the new register 13 as the index register, any of the old workspace registers are available for use by the called subroutine. For example, to gain access to the calling program's register 3, a reference in the operand field of the form: (See *Figure 6.11*)

```
@6(13)
```

adds 6 to the contents of register 13 to determine the address of old register 3. If each subroutine's workspace is independent of all other workspaces, there is no problem in losing vital data for one subprogram through the operations of another. However, each subroutine must use some form of indexed addressing using register 13 to obtain access to the data of another subprogram when that is required. Also, this may require an excessive amount of memory for workspace storage. The advantage of the independent

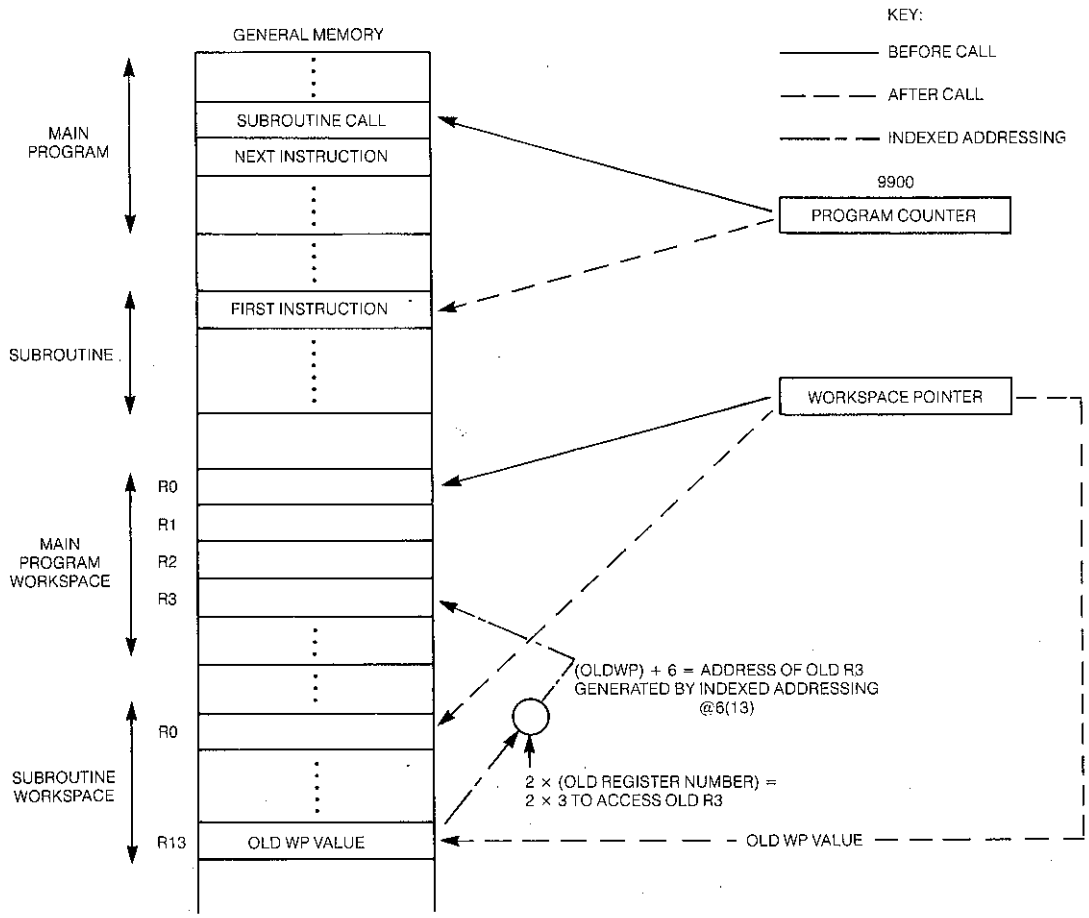

Figure 6.11 *Procedure for Locating Old Workspace Register Values After a BLWP Call with Independent Workspaces*

workspace approach is that subroutines can be nested without destroying return linkages. However, if the BLWP approach is used with the shared subroutine approach, the contents of the last three workspace registers must be pushed to a stack at the first of each subroutine, and popped from the stack at the last of each subroutine to allow subroutine nesting. This could require an excessive amount of program overhead to avoid destroying return linkages. If the stack pointer is decremented before saving (pushing) information to the stack area and incremented after restoring (popping) information from the stack, the following sequence is used to save the contents of registers 13 through 15:

```
DECT  10      Decrement pointer to next empty word
MOV   13,*10  Push R13 to stack
DECT  10      Decrement pointer to next empty word
MOV   14,*10  Push R14 to stack
DECT  10      Decrement pointer to next empty word
MOV   15,*10  Push R15 to stack
```

This sequence is placed at the first of a subroutine, before the next BLWP instruction. At the end of the subroutine the following sequence restores the return linakge and returns to the original calling sequence:

```
MOV   *10+,15 Pop stack to R15, Change pointer
MOV   *10+,14 Pop stack to R14, Change pointer
MOV   *10+,13 Pop stack to R13, Change pointer
```

This popping sequence is the reverse of the pushing sequence. Following this sequence is the RTWP instruction.

A limited amount of subroutine nesting with BLWP calls can be provided by having each successive subroutine's workspace located three words back from the previous workspace, as shown in *Figure 6.12*. With this approach, the return linkage is never destroyed by subroutine operations, and each workspace will pass ten data words to the next subroutine. As the returns from the subroutines are performed, data from previous subroutines is available through the workspace pointer of the main program and the proper index value. Thus, the contents of register 0 of subroutine 2, which is register 3 of subroutine 3, is six words back from the address of register 0 of the main program. This register is addressed by the main program workspace pointer, which is stored in register 10 of the main program workspace and in register 13 of subroutine 1 workspace. This storage occurs when subroutine 0 calls subroutine 1. Thus, once subroutine 1 has been called, register 0 of subroutine 2 can be referred to, using the indexed addressing operand of:

```
@-12(13)
```

The approach of *Figure 6.12* does avoid the need for a stack, but it requires much more thought on the part of the programmer to keep track of the location of the system data and program linkages at any given time.

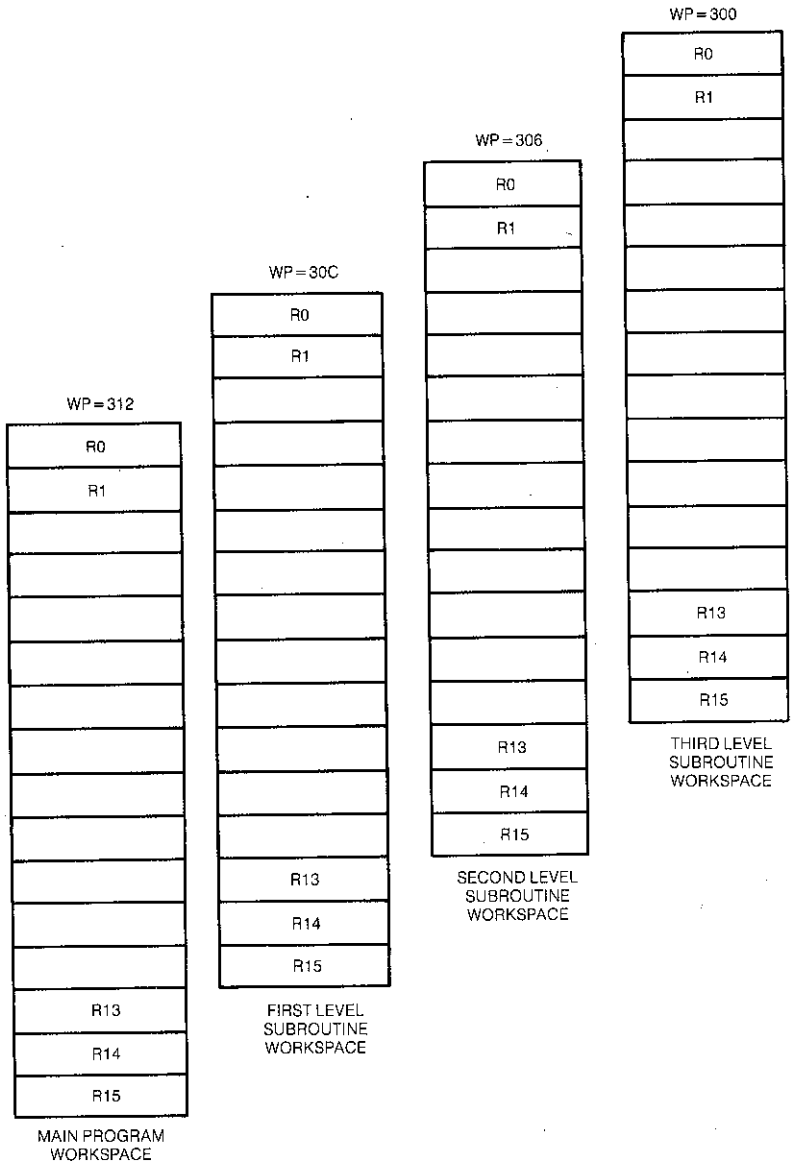

Figure 6.12 Data Structure for Overlapping Workspaces

Parameter Passing Techniques

Shared, independent, and overlapping workspace approaches all provide a technique for passing of data variables and control codes from the calling subprogram to the called subprogram. The XOP subroutine procedure of the 9900 family provides a different form of parameter passing. There, the calling argument in the XOP coding is stored in register 11 of the subroutine's workspace. *Figure 6.13* reviews this procedure using direct addressing. In this case, the data or code in location PARAM is saved in register 11 of the XOP's workspace. The subroutine can then refer to register 11 to locate data that is accessed by both the main and called programs. For example, the address or data in register 11 of the new workspace could represent a single data variable, or it could be the first of a string of data and control codes stored in memory. In the latter case, the list

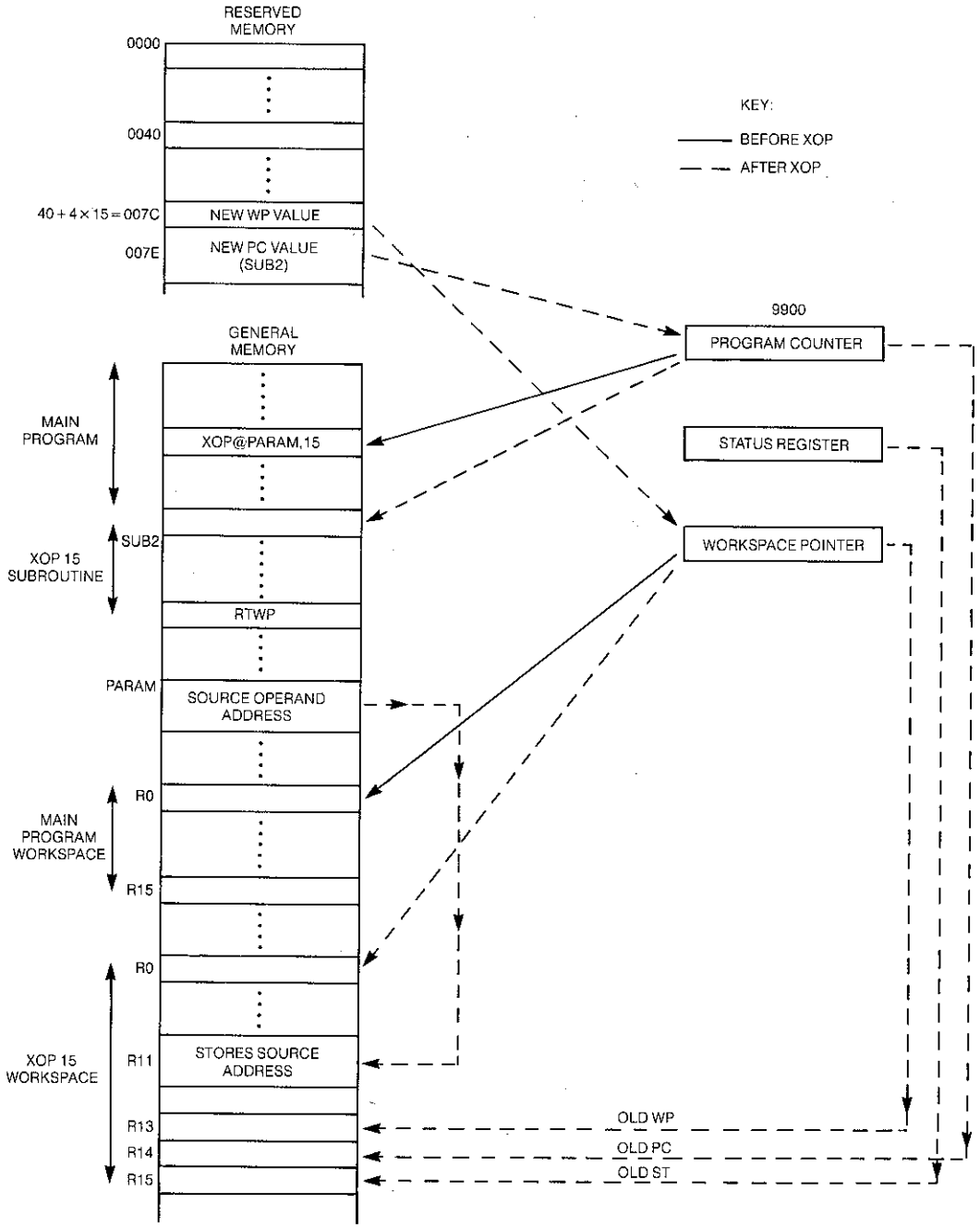

***Figure 6.13** Execution of XOP Instruction with Parameter Passing*

forms a first-in-first-out or FIFO memory passed to the subroutine. The list is constructed in memory by the calling procedure. It sets up the data by the subroutine in order, beginning at the address to be passed to the XOP in register 11. Then, the XOP can sequence through this list beginning at that address, to obtain the data to be processed. The memory organization of the list could also be a LIFO file (last-in-first-out). To do this, the calling procedure accumulates the data in reverse use order, with the last data assembled at the address passed to the XOP. Then, the XOP works through the data, starting at this address.

Some of the techniques discussed thus far can only be used in processors that offer multiple sets of register files like those in the memory-to-memory architecture of the 9900 family of processors. Other processors rely on stack operations to store inactive register values in protected locations while the subroutine uses the registers for its own operations. Then, after the subroutine operations have been completed, the old values can be retrieved from the stack. Further, during the subroutine operations, the old data is available to the subroutine by referring to the stack pointer and adding or subtracting an appropriate offset value, using some form of indexed addressing.

There is another approach that can be used by all processors to pass parameter values to a subroutine. This approach involves placing the values just after the call instruction, as shown in *Figure 6.14*. This approach may be useful for passing control constants to the subroutine if the program is stored in read-only memory. If the program is stored in read-write memory, the parameters could be constants or variables as needed. The idea behind the approach is that the addresses of the parameters are available through the return program counter value which is saved on the return stack in a typical microprocessor. In 9900-based systems, it is saved in register 11 after a BL subroutine jump, or in register 14 after a BLWP subroutine jump. The parameter list can thus be sequentially accessed by using this program counter value as an indirect address, and then incrementing the value after each access to move to the address of the next value. After all values in the list have been accessed, the program counter value must be incremented once more. It then contains the correct address of the instruction to return to when the subroutine ends. As an example, the following coding illustrates a subroutine call that passes two parameters to the subroutine ANG, which may use one or both parameters in its execution:

```
      BL    @ANG     Call subroutine Ang
      DATA  >2000    Parameter 1 is 2000(16)
      DATA  >0D00    Parameter 2 is 0D00(16)
       .
       .             Remaining instructions of calling
       .                procedure
      AORG  >260     Establish starting address of ANG at 260
ANG   MOV   *11+,3   Move parameter into register 3
      C     3,2      Compare parameter to contents of R2
      JEQ   ABS      If equal, go to ABS instruction
      MOV   *11+,4   If not copy 2nd parameter into R4 and
                        change R11 value to return address
      B     *11        and return
ABS   MOV   *11+,4   If R2 contents equal to 1st parameter,
                        copy second parameter into R4, update
      NEG   4          and negate contents of R4
      B     *11        and return
```

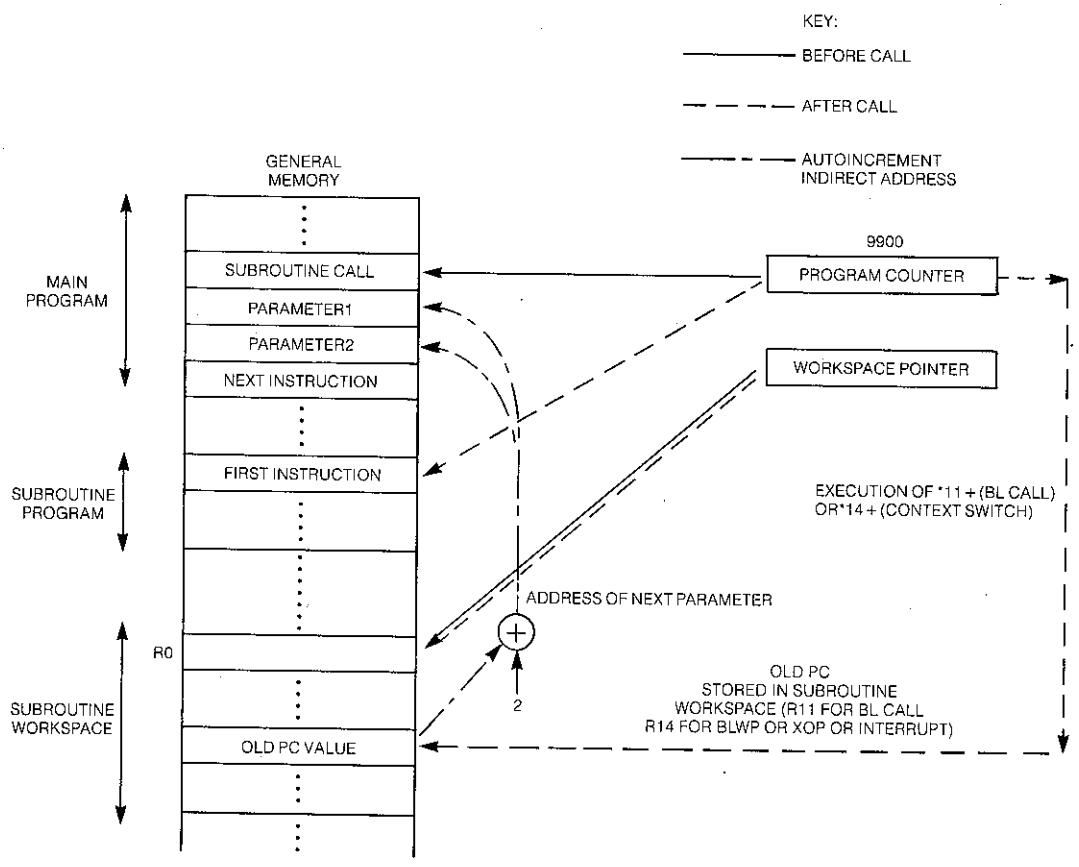

Figure 6.14 *Parameter Passing Using the Old Program Counter Value*

The subroutine autoincrements the address in register 11 through the two parameters and to the address of the instruction immediately after the second parameter. While this example involved only two parameters, there is no limit to the number of parameters that can be passed in this way. The number could be variable, with the first parameter continuing the number of parameters in the list, or the last parameter's being a list termination code. In any case, the programmer must write the subroutine so that the proper return linkage (back to the next instruction of the calling sequence after the parameter list) is established before the subroutine return instruction is executed.

The form of parameter passing and data file organization used in a given programming situation depends on the requirements of the programs and the characteristics of the subroutines involved. The type of subroutine call used and the methods required to maintain return linkages also depend on the characteristics of the subroutines involved, as well as on the number of subroutines that may be nested at any given time. In order to use the parameter passing and data file control techniques discussed in this section, the programmer must analyze each programming requirement in terms of the more general system principles of modularity and structured design.

Such an allocation would require only 28 words of RAM, which may be less RAM storage than would be required by a shared workspace-stack file scheme. In fact, such a shared workspace scheme requires 16 words for registers and 3 words of stack for each level of BLWP call. Four levels of BLWP calls require a stack of only 12 words, for a total RAM commitment of 28 words. This situation is equivalent to using the allocation scheme of *Figure 6.12* for four levels of calls.

Generally, the preservation of linkage, passing of data, provision of a sufficient number of working registers, and allocation of workspace and memory data locations to a module structure can be arranged fairly easily if the modules call only other modules. The situation is complicated if a module should have to call itself, or if a module can be entered before it has completed its operations. Modules like these are called *recursive* modules and *re-entrant* modules, and they sometimes occur in complicated software situations.

Recursive and Re-entrant Modules

A recursive module is one that can call itself. It can occur in certain mathematical manipulations (the factorial function is a commonly used example) or in certain list-processing schemes. The basic idea is that in the process of executing a certain function, an identical type of function is required. The programmer's alternatives in this situation are to provide as many copies of the function module as there are levels of nested use, or to design a single copy of the module so that it can call itself. The latter approach is to make the module recursive.

A re-entrant module is one that can be entered a second time before it has completed execution resulting from a first entry. In microcomputer systems, this situation is more likely to occur than is the recursive module situation. The re-entrance occurs most often in an interrupt driven system, in which a given interrupt-related subprogram requires the use of a module that is also used by the non-interrupt system software. The basic situation is illustrated in *Figure 6.16*. The main program sequence may use module A, and an interrupt sequence may also require module A. Since interrupts can occur at any time, it is possible that an interrupt will occur when the system is executing module A. In this case, a subroutine jump will be made out of module A (before it has completely executed) into the interrupt sequence, which, in turn, will cause the system to enter module A. Unless module A is designed to be re-entrant, the second entry into module A will likely destroy data and program linkage that existed when the interrupt sequence called the module. Thus, the programmer must either make module A re-entrant, or the interrupt sequence that uses module A must be disabled until the execution of module A by the main system program has been completed. The latter approach is the simplest strategy, but it is feasible only when the response time to the interrupt sequence is not critical. If it is possible that incoming information or outgoing control will be lost by locking out the interrupt sequence during the execution of module A by the main program, this approach is not acceptable, and module A must be designed to be re-entrant.

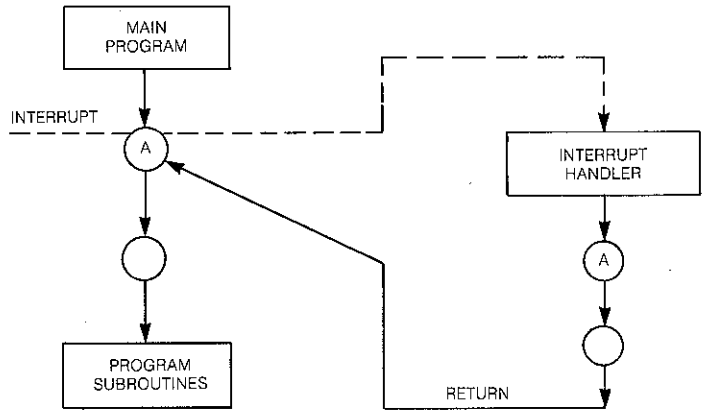

Figure 6.16 *Interrupt Requiring Re-entrant Programming*

In order for a program module to be re-entrant, it must have the following features:

1. The subroutine must not alter data that is common to all calling procedures. That is, subroutine data common to all calling procedures must be read-only data.
2. Data that is related to a calling procedure must be stored in register file and memory data locations that are not used by other calling procedures.
3. The subroutine must not perform operations that will alter its instruction or data codes.

A simple example will illustrate some of these features by comparing a re-entrant and a nonre-entrant version of a commonly used subroutine. The module in this example is one that maintains the filling of a data buffer by the following sequence. In register 1, the module receives from the calling procedure the data to be sent to the buffer. It receives the address of the location within the buffer to receive the data in register 2, and the last address of the buffer in register 3. If, after the data is sent to the buffer and after two is added to the register 1 address, the value in register 1 exceeds the value in register 2, the subroutine sets register 4 to all 1's. This signals the calling sequence of a buffer full condition. If the buffer is not full, register 4 will be cleared before a return to the calling procedure.

Figures 6.17a and *6.17b* show the calling sequence used to access the subroutine BUFF by both the main program and an interrupt sequence. For a general case, the calling procedure is a BLWP subroutine call. *Figure 6.17c* shows a possible program version of the subroutine BUFF. In this example, the subroutine is re-entrant only if the workspace defined by the interrupt context switch is different from the workspace defined by the main program context switch. In fact, each interrupt sequence that can call BUFF must use an independent workspace from the one used by main program calls and from the workspaces used by all other interrupt calls (assuming one interrupt sequence can be

interrupted by another interrupt sequence). The reason for this can be understood by considering the consequences of using the same workspace in the interrupt sequence call as was used in the main program call. In the first place, the return linkage of the main program will be overwritten by the return linkage produced by the interrupt call. Even if this were prevented by saving the linkage on a stack at the first part of BUFF, if the interrupt occurs at any time prior to the RTWP execution, the data or status flags will be incorrectly processed for the main program. For example, if the interrupt occurs during the MOV 1,*2 instruction, that instruction will be executed properly but the INCT 2 instruction will not be executed until the interrupt call loads register 2 with a new address value. After the interrupt sequence has executed properly and the return is made back to the original INCT 2 instruction, the address being incremented will be the interrupt buffer address value left over from the interrupt sequence execution of BUFF, and not the address value for the main program sequence. From that point on, the subroutine will not properly provide the buffer filling operations for the main program. On the other hand, if the transfer vectors of *Figure 6.17* are used, both the main program and the interrupt sequence specify the BUFF location as the program counter value. When the BUFF subroutine is called, the main program uses 330_{16} as the workspace pointer value, but the interrupt sequence uses 300_{16} as a workspace pointer value. Thus, registers 1, 2, 3, and 4 for the interrupt call are not the same as registers 1, 2, 3, and 4 for the main program call. The operations of BUFF when re-entered due to the interrupt will not affect the values of registers 1 through 4 used by the main program. In fact, one of the main advantages of the memory-to-memory architecture and the context switch subroutine procedure is the possibility of endless sets of register (workspace) files, which, in turn, make re-entrant programming a relatively simple matter.

```
        MOV   @DT,1
        MOV   @SC,2
        MOV   @EA,3
        BLWP  @BUFVEC
        MOV   2,@SC
        MOV   4,@FG

BUFVEC  DATA  >330
        DATA  >200
```

a. Main Program

```
        LI    12,INBS
        LDCR  1,0
        MOV   @IC,2
        BLWP  @IEND,3
        BLWP  @IBFVEC
        MOV   2,@IC
        MOV   4,@FLG
        RTWP

IBFVEC  DATA  >300
        DATA  >200
```

b. Interrupt Program

```
        AORG  >200
BUFF    MOV   1,*2
        INCT  2
        C     2,3
        JGT   FUL
        CLR   4
        RTWP
FUL     SETO  4
        RTWP
```

c. Subroutine BUFF

Figure 6.17 *Subroutine BUFF and Calling Sequences to Access It*

When the commonly used subroutine is very short, as is the case of the BUFF subroutine of *Figure 6.17* the programmer may decide to use one copy of the subroutine in the main stream of activity and another copy in the interrupt stream of activity. In this example, this adds eight words to the system program storage requirements. However, even with this short program it is more efficient to use a shared BUFF module that is re-entrant with overlapping workspaces. This saves the extra eight words of program storage, and adds only four words of RAM workspace requirements, assuming the new interrupt workspace register 15 is the same as the old workspace register 11. This provides for independent registers 1 through 4 for each calling procedure, and provides for separate linkage registers 13 through 15 for each calling procedure.

Even with context switching, it is possible to set up a nonre-entrant version of the BUFF program if the programmer is not careful. An example of such a nonre-entrant version is shown in *Figure 6.18*. The only differences between the coding of *Figure 6.17* and the coding of *Figure 6.18* is that the buffer full status flag is stored in register 4 in *Figure 6.17*, and in a direct address location FLAG in *Figure 6.18*. All aspects of both versions work properly with independent registers 1 through 4 and 13 through 15 under a re-entrant condition, except for the establishment of the flag conditions of *Figure 6.18*. For example, if the main program called BUFF, and the CLR @FLAG was executed just before the interrupt, the buffer is not full for the main program. However, if the interrupt called BUFF and encounters a full buffer, it sets the contents of FLAG to all 1's. The FLAG will then contain all 1's when the return to the main program occurs, erroneously indicating a full buffer for the main program. The error in the version of *Figure 6.18* is that the programmer has neglected to establish independent call-oriented storage locations for all variables that can be changed by the program operation. Generally, the programmer must analyze each re-entrant module coding to make sure that all the conditions required for re-entrance are met. Such efforts can pay off in reduced program storage in large systems.

Much of the discussion of this chapter has dealt with the module level of program development, both in terms of interrupt service modules and the general characteristics and considerations in the design of all program modules. The programmer must also be careful to make sure that the overall system program uses these modules in the proper sequence to provide the required system behavior. Some of the characteristics of system programming will be considered in the next section.

```
        AORG   >200
BUFF    MOV    1,*2+
        C      2,3
        JGT    FUL
        CLR    @FLAG
        RTWP
FUL     SETO   @FLAG
        RTWP
```

Figure 6.18 *Non-Reentrant Version of BUFF*