M0090

A GUIDE TO MULTICS FOR SUBSYSTEM WRITERS

Chapter III

Inter-Procedure Communication

Elliott I. Organick

Draft No. 5

February 1968

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER III

INTER-PROCEDURE COMMUNICATION

3.1 INTRODUCTION

Call, save, and return sequences are short sequences of instructions whose execution constitutes the standard way for one procedure segment to communicate with (i.e., transfer to, pass information to, and return from) another procedure segment.

The instructions in these sequences involve the management of the so-called "process stack," a special segment used by all procedures of a given process. Executing the standard call, save and return sequences also insures that all pure procedures are automatically recursive (and also sharable), a deliberately planned by-product. Nonstandard methods of communication, whose use in special cases may improve efficiency, are by no means ruled out in Multics. They may be used by the advanced subsystem writer when appropriate.† The successful execution of any process depends upon flawless management of the stack; hence (independent) translators, including assemblers, that produce target code, are responsible for automatically generating call, save and return sequences. As a result the ordinary user is liberated from what might otherwise be an awesome task. In addition to user procedures, essentially all others, including those of the supervisor, the public library, and commands employ the same intercommunication sequences. A subsystem writer who is going to produce an independent translator must become thoroughly familiar with details of this chapter, and of all pertinent MSPM references. The prime reference at this time is BD.7.02. Secondary references are BD.7.03 and BD.9. However, it would seem that any subsystem

†A process may contain one or more groups of related procedure segments, so designed by a subsystem writer that intragroup communication is achieved without the standard call, save and return sequences. While executing within a group, short cuts can result in improved efficiency. At some point however, such a group of segments must interface with system-designed procedures, and here the method of communication must be standard.

writer would need to become somewhat familiar with the process stack, its role and its management.

This chapter begins with a discussion of the process stack and the call, save and normal return sequences for ordinary slave procedures. Argument lists, their structure and creation are then treated. There is a brief summary of the storage structures for several types of arguments and a separate disdussion for function name arguments. Argument lists for calls to internal procedures are considered next. We postpone talk about generation of argument lists for calls on procedures in outer protection rings (outward calls, BD. 9). Chapter 4 will discuss protection rings. A few remarks concerning execute only procedures conclude this chapter

3.2 THE PROCESS STACK

The key to understanding the effectiveness of call, save and return sequences is to first understand the concept of the stack segment. There is a stack segment created for each process.† Each time one segment transfers or returns control to another segment, a frame of data consisting of key information, such as index register contents, A and Q register contents, base register contents and other pointers, are either saved in the stack segment or retrieved or released from it. Moreover, while a procedure segment <b> is in execution, space for all its temporary storage is allocated in the stack segment. When <b> returns to the program which called it, this temporary storage is automatically released by adjusting the stack pointer. The stack is used as a pushdown store by carefully maintaining a current stack pointer. This pointer is kept in the sb←sp base pair. The sb holds the effective pointer, i.e., to word zero of the stack segment, and the sp holds the relative position within the stack segment (current pointer) of word zero for the latest frame of data added to the stack.

†Strictly speaking, as will be seen when we discuss BD. 9, there is actually a stack segment created for every "ring of protection" within the process. The notion of rings and multiple stacks and descriptor segments is purposely delayed until Chapter 4.

Figure 3-1 shows the layout of a typical stack frame. Each frame consists of a header (32 words) and a body. The header for the current frame on the stack is used to save the contents of registers and pointers, etc., which are meaningful to the currently executing procedure at the time it prepares to call on another procedure. The six words shown as not used should be considered as reserved for use of future system services. The body of the frame is of variable length and provides the temporary storage required by the currently executing procedure. For PL/I procedures, for example, space for all variables having the automatic attribute is allocated in the body of a stack frame† as soon as execution of the procedure commences.

Normally‡ the amount of space required for the body of the frame will be determined by the compiler or assembler. Therefore, at the time the frame is created, all the space required for the body can be allocated at one time. Allocation is in units of 8 words so that the immediately following frame header will begin at an internal address that's congruent to zero (mod 8).

Figure 3-2 suggests how the <stack> grows frame by frame as execution moves from <alpha> to <beta>, then to <gamma>.

Figure 3-3 gives a closer look at the frame developed for and during execution in <beta>. We find it convenient to think of the frame header as consisting of seven items, so marked in the figure. Each header item is marked in its upper right corner to denote which sequence, i.e., call or save, is responsible for placing the item in the header. (The item numbers used in Figure 3-3 are used repeatedly in subsequent discussions.)

3.3 THE CALL SEQUENCE

Whenever we wish to transfer from one procedure to the next we would issue what amounts to a standard marco call. For example, the call macro in eplbsa has the form:

call entrypoint (arglist)

---

†Also, the "specifiers and dope" for some types of based controlled variables are kept in the stack frame. See BP.4.00 for more details.

‡Exceptions arise when the stack frame must be extended during execution of the procedure, as discussed in Section 3-4.

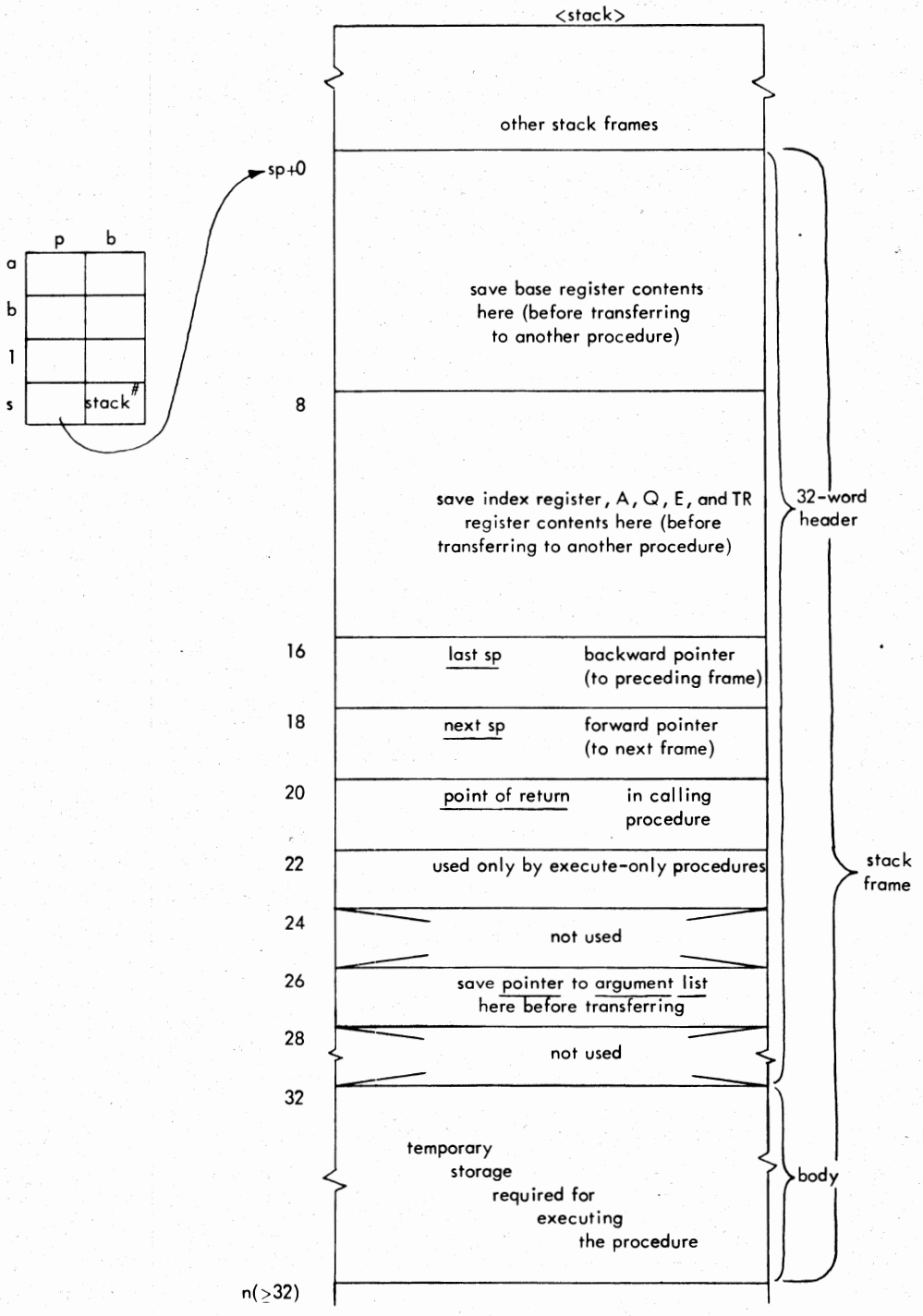


Figure 3-1. Layout of a Typical Stack Frame.

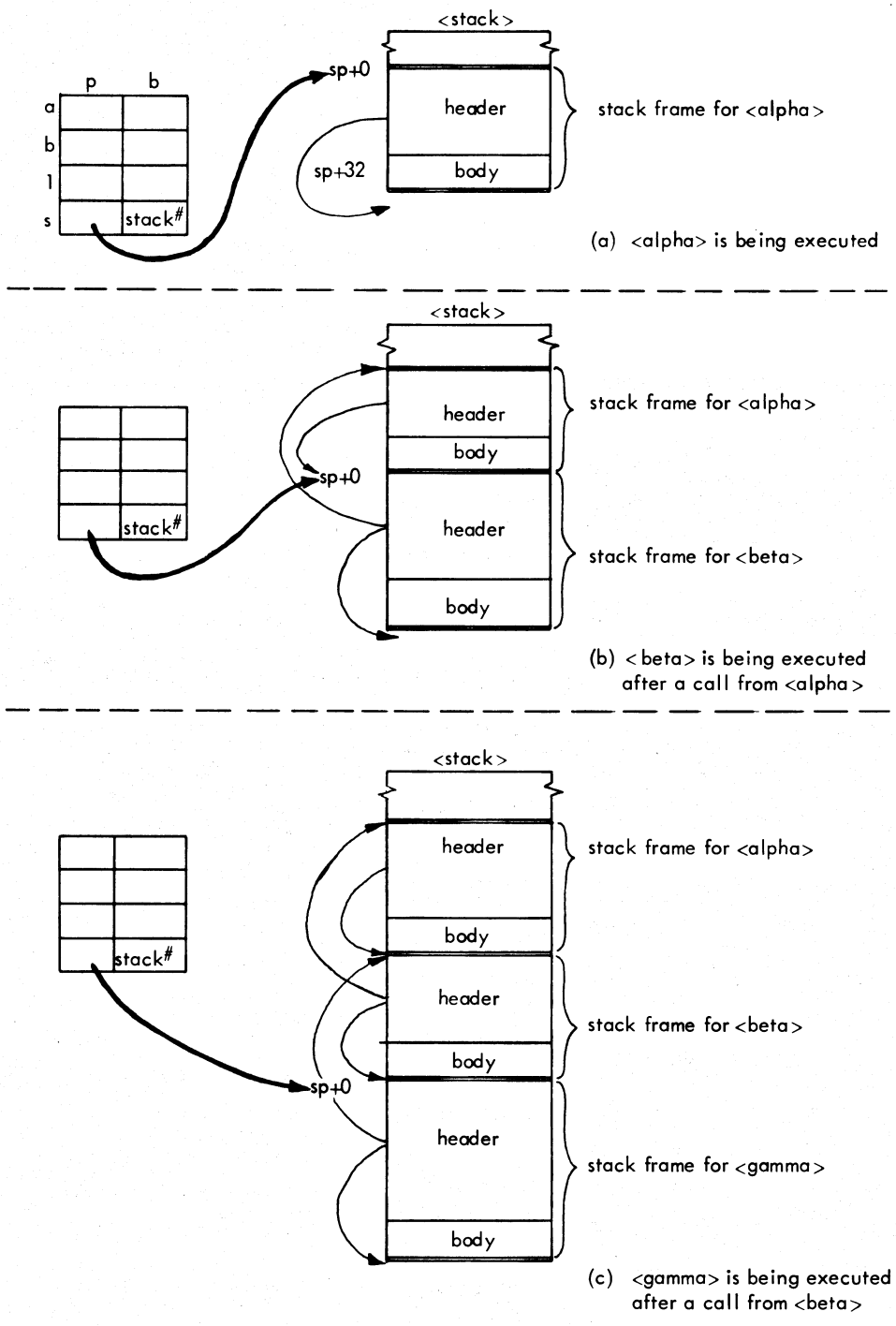


Figure 3-2. Showing Development of <stack> During a Chain of Calls.

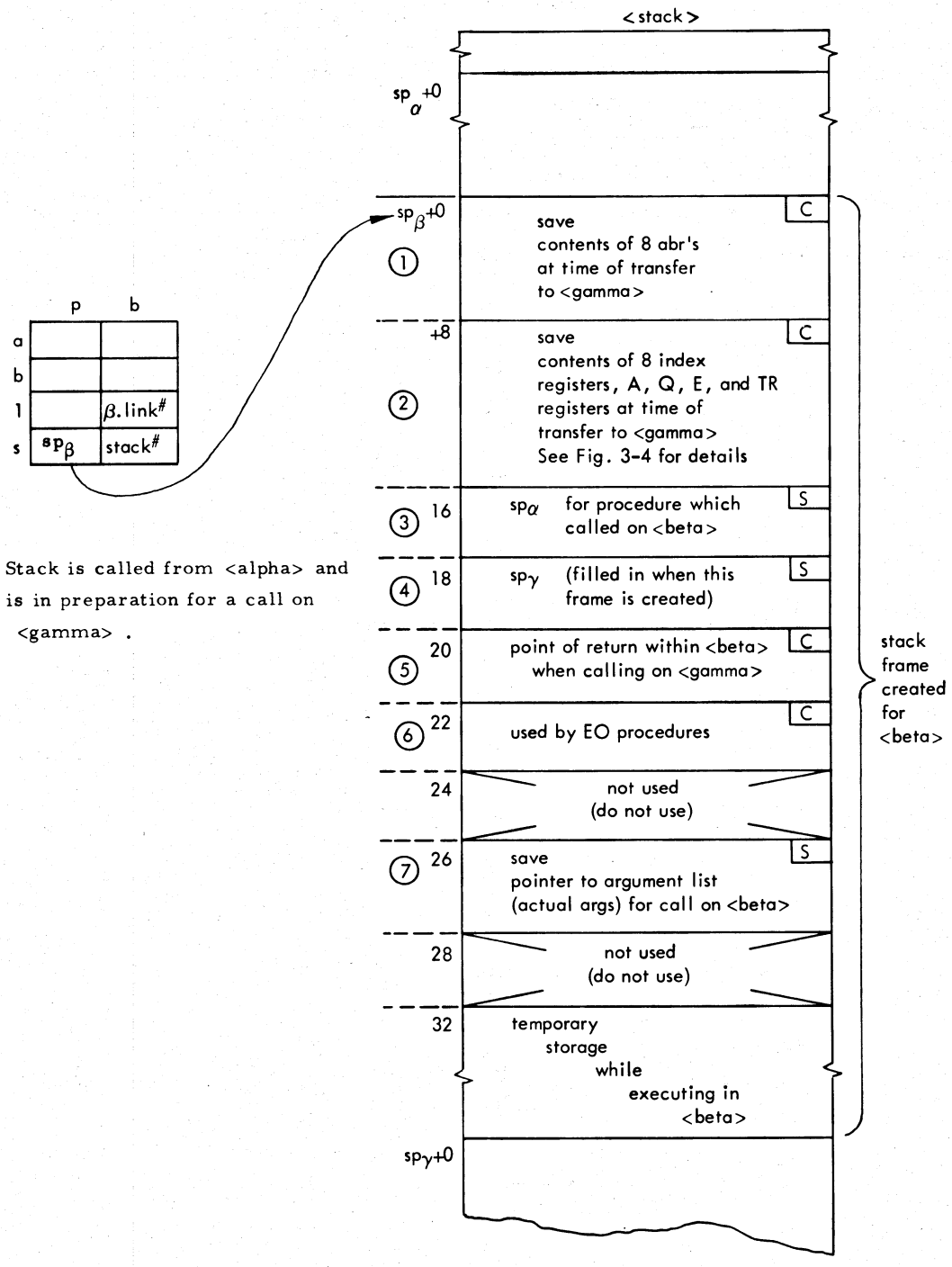


Figure 3-3. Appearance of Stack Frame When Executing in <beta>

Here entrypoint is the entry point of the procedure being called (any type of address may be used for entrypoint), and arglist is the location of the argument list. (Any type of address may be used for arglist, but we shall assume throughout this chapter that argument lists are always kept in the stack.)

For example, if eplbsa encounters:

call< gamma>|[entry2] (sp|arglist)

entrypoint arglist

in processing the code for < beta>, then the expanded sequence will be:

1	stb	sp| 0	save contents of the 8 abr's in sp+0 thru sp+7. Item 1.
2	sreg	sp|8	save contents of the 8 index registers, A, Q, E and TR registers in sp+8 thru sp+15. Item 2. See Figure 3-4.
3	eapap	sp|arglist	place the pointer to the argument list in ab←ap for use by < gamma>. The argument list is kept in temporary storage.
4	stcd	sp|20	save the point of return to <beta> i.e., (ic)+2, and (pbr), and (indicators) in sp+20 and sp+21. Item 5.
5	tra	< gamma>|[entry2]	transfer to called procedure (via the mechanism described in Fig. 2-13).

The first, second and fourth instructions store items 1, 2, and 5 in the frame header.† If we ever return to < beta> and at some later time issue a new call, possibly to some other procedure, new values for items 1, 2, and 5 would be stored in this header.

Item 5 has the format of an its pair

	18——28	29——35
(pbr)	0	its
(ic)+2	(indicators)	0

†In the more elaborate call sequence for execute-only procedures there are also instructions to store Item 6. See BD. 7.03 for such details.

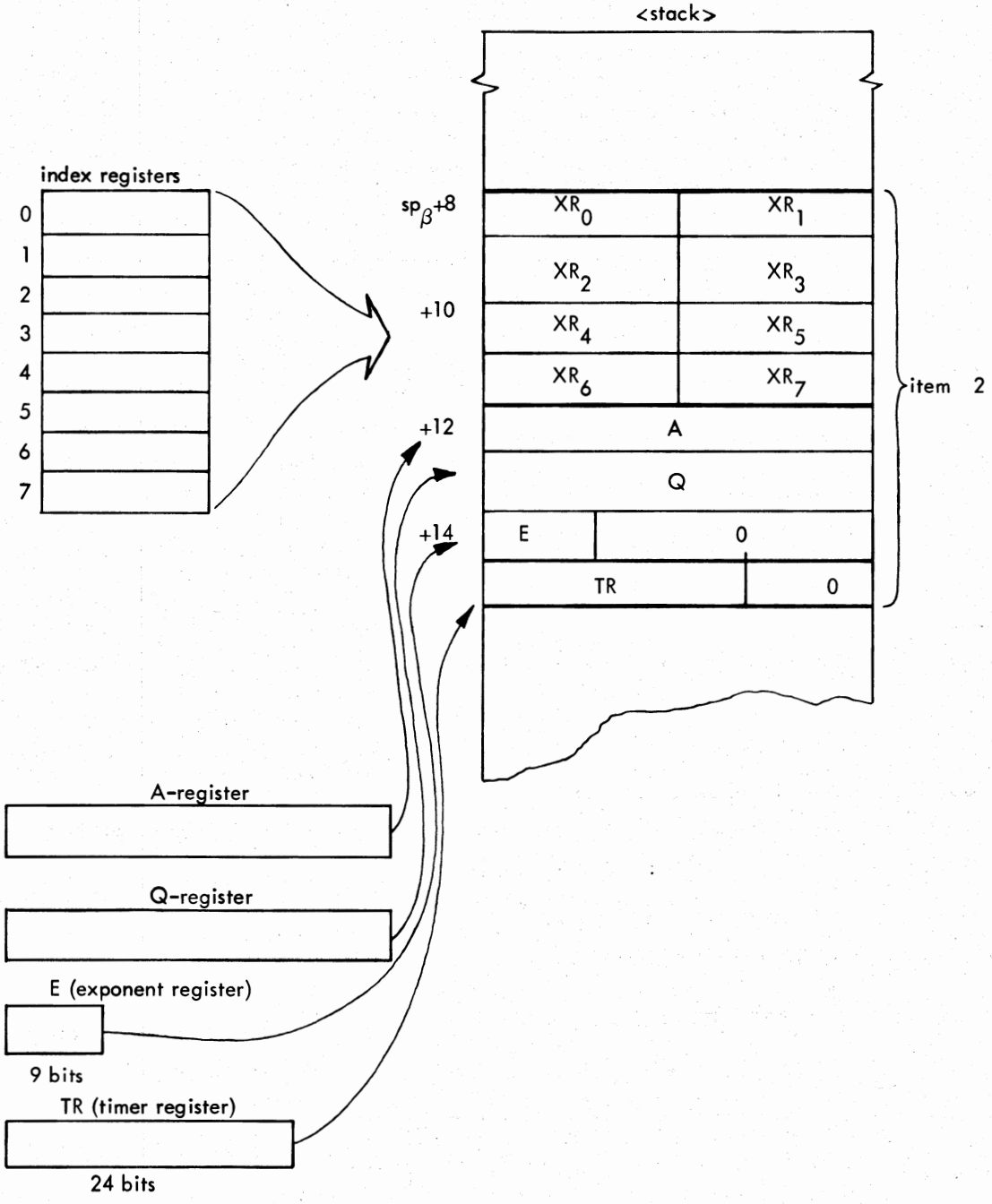


Figure 3-4. Saving Index Registers, A, Q, E, and TR Registers upon Execution of: sreg sp|8.

The tra instruction of the call sequence will be assembled as

tra lp|k,*

The symbol "k" represents an offset within <beta.link>. The ft2 pair at this location is converted by the linker (as explained in Section 2.9) to the its pair:

k

gamma.link#	0	its
dist	0	0

The pair of instructions found at gamma.link#|dist, you will recall, is for the purpose of (1) loading gamma.link# into the lb base register (and normally a zero into lp) and (2) transferring (via another its pair) to gamma#|entry2. If you have forgotten how this mechanism works, you should review the diagrams given in Figure 2-13.

3.4 THE SAVE SEQUENCE

The purpose of the save sequence is to (1) generate a new stack frame of 32 or more words for the just-called procedure, and (2) supply values for items 3, 4, and 7 in the header of the newly formed frame.

Before we explain the details of the save sequence we make the following general observation. If we continue following the example of <beta> calling on <gamma> (as we shall here), the items 3, 4 and 7 about to be established will be those of the <gamma> frame. To see how the corresponding items would have been established in the <beta> frame given in Figure 3-3 we would have to shift the reference point of our example to that of <alpha> calling on <beta>. We prefer instead to continue the "walk" through one complete call, save and return cycle that starts with call.

The save sequence for <gamma> is actually stored in two parts. Part 1 consists of the eaplp, aos, tra, arg guadruplet† and the link, which are kept in <gamma.link>:†

```
dist:   eaplp    -*,ic
        aos      2,ic
        tra      link2-,*ic*
        arg      0
```

†This is the way it's viewed in the MSPM (BD. 7.02)

†Effective 2/12/68 the eaplp, tra pair is replaced by the quadruplet shown here, which includes a usage counter to keep track of the number of times an entry has been used. The aos instruction adds one to the location immediately following the tra instruction. (Drafts 2 and 3 of Chapter 2 do not show this usage counter.)

link2:

gamma#	0	its
entry2	0	0

The effect is to establish the lb←lp pair for pointing to <gamma.link>, and transferring to <gamma>. Part 2 is kept in <gamma> proper, beginning at entry2. This part has the job of creating the new stack frame of size tnew and storing item 7 (the argument list pointer) into it. A new stack frame is said to be created when

(a) the new frame contains a backward pointer item (3).

(b) the new frame contains a forward pointer to the next frame (item 4).

(c) the sb←sp pair is reset to point to the beginning of the new frame.

A crucial Multics requirement is that during the course of executing instructions to create a new frame, there must never be an instant when the beginning of the next frame (i.e., the frame beyond the last fully created one) is undecidable. The reason is simple: The Multics supervisor uses the same stack to store the stateword of this process (i.e., register contents, etc.) in the event of a hardware system interrupt. If such an interrupt occurs during creation of a new stack frame there must be a completely safe way to identify the beginning of the next frame for use in handling the interrupt. The handler always locates safe-to-use storage at a point in the stack beginning at 32 words beyond the beginning of this frame.

The save sequence instructions of Part 2, shown immediately below, are especially designed to meet this objective.

1.	entry2:	eapbp	sp|18,*	Save item 4 of the current frame temporarily in bb←bp. That is to say, let bp temporarily become the stack pointer for the new frame.
2.		stpsp	bp|16	Store current sp value, i.e., sp_β as item 3 of the new frame
3.		eapbp	bp|tnew	Create item 4 for the new frame in bb←bp by adding tnew to what is already in bb←bp (item 4 is a pointer to the frame following the one currently being created).

4.	stpbp	bp|18-tnew	Store item 4 for new frame in position 18 of new frame. (It's stored in sp_γ+tnew+18-tnew or sp_γ+18).
5.	eabsp	bp|-tnew	Form new stack pointer, i.e., sp_γ in sb←sp, by setting the sp part of sb←sp to point to the beginning of the frame for gamma (i.e., the bp part of bb←bp minus the length of the new frame).
6.	stpap	sp|26	Save item 7. I.e., save the argument list pointer left in ab←ap by <beta>.

Note that at any given instant the frame pointed at by the sb←bp pair is the last fully created frame. Upon completion of the fifth instruction in the sequence (eabsp bp|-tnew) the new frame has been fully created. Interrupts occurring any time before or after this instant are treated as shown in Figure 3-5.

The particular instructions used in this sequence depends on the size, tnew, of the frame being created. The value of tnew† can ordinarily be determined by the assembler or compiler.

---

†The maximum value that can be used for tnew in the type 1 instructions of the save sequence is 2^{14}. If a frame having a size in excess of 2^{14} words is to be allocated, two additional instructions may be generated at the end of the sequence:

eabbp	bp|excess	add excess to copy of item 4 and
stpbp	sp|18	store as revised value for item 4.

This works if excess is itself $\leq 2^{14}$. A somewhat slower-to-execute sequence would be needed in building frames whose length ranges up to 2^{18} words. One such sequence might be:

adbbp	excess,du	type 0 instruction: add excess to current contents of bp. The du, or direct upper modifier, indicates the value of excess is in the address field of this instruction.
stpbp	sp|18	store as revised value for item 4.

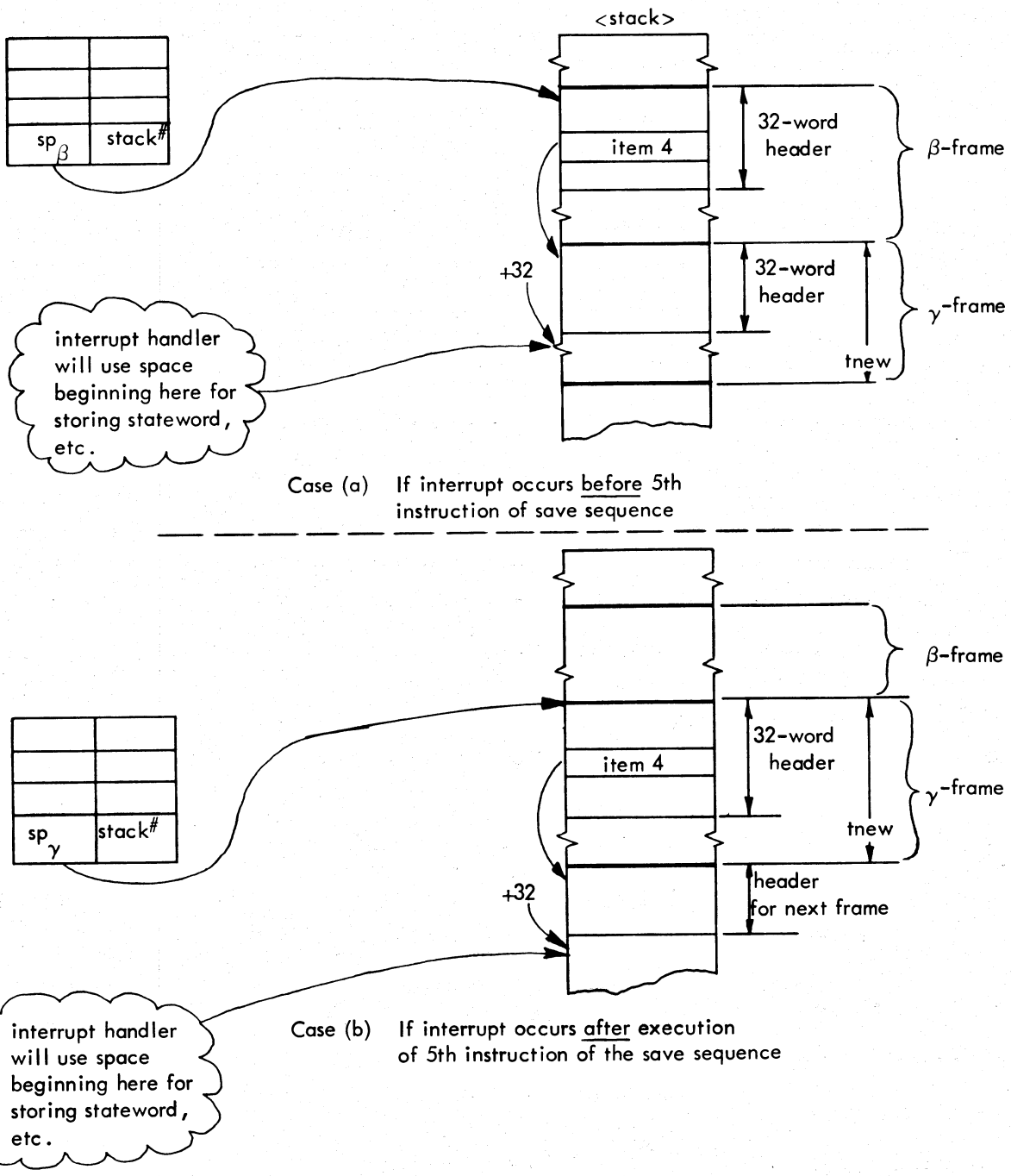


Figure 3-5. Handling Interrupts Before and After 5th Instruction of the Save Sequence

In short, upon completion of the save sequence in <gamma> we've set the sb←sp to point to the beginning of the new frame of tnew words. Item 3 (sp+16) has been set to point backward to the beginning of the preceding frame for <beta>. Item 4 points forward to the beginning of the next frame, and item 7 holds a copy of the pointer to the argument list of the call to <gamma>. If, during execution in <gamma>, additional amounts of temporary stack storage are required, more space may be allocated to the frame simply by altering item 4.[≠]

The instructions:

```
eapbp   sp|18,*     get current value in item 4.

eabbp   bp|extra    increment by extra (which should
                    be a number that's congruent to
                    0 (mod 8) and ≤ 2^14.

stpbp   sp|18       store new value of item 4.
```

would do the job.

Also, by way of summary, the following is the condition of the abr's upon completion of the save sequence in <gamma>.

	p	b
a	sp_β+arglist	stack#
b	sp_γ+tnew	stack#
l	0 (normally)	gamma. link#
s	sp_γ	stack#

temporarily holds a pointer to the first word in <stack> beyond the <gamma> frame

3.5 RETURN SEQUENCES

There are two types of return sequences which can be executed, the normal or standard return to the point of call and an abnormal return, i.e., a return to a program point within an arbitrary procedure which point has been supplied as an argument. We shall discuss only the normal return here.

[≠]To give you some idea where stack extension might be used, you should note that in the original EPL implementation, temporary arrays that are adjustable are allocated space (when their space requirements become known) as extension of the current stack frame.

A subsystems writer should ordinarily implement or employ abnormal returns† only where necessary, i.e., only where the extra overhead is justified.

3.6 THE NORMAL RETURN SEQUENCE

If you've followed what is required in the call and save sequence, the normal return sequence is quite simple to understand. All that's needed for <gamma> to return to <beta> is to

(1) reload the base registers and index registers, etc., (all but the TR register) whose values were saved in the <beta> frame during <beta>'s call on <gamma> and

(2) restore the ic and pbr (and indicators) registers to the values tucked away as item 5 in <beta>'s frame.

Only three GE 645 instructions are actually required. The eplbsa macro call,

return

expands to:

```
ldb    sp|16,*    | reload 8 base registers
lreg   sp|8       | reload 8 index registers,
                  | A, Q, and E registers
rtcd   sp|20      | "restore control word double"
```

The first of these instructions loads the 8 base registers from the location pointed to by the contents of sp|16, which is item 3 of the <gamma> frame. Item 3 is the backward pointer to the top of the <beta> frame. The net effect is to reload the base registers stored in the <beta> frame. As a consequence, the sb←sp pair will now point to the beginning of the <beta> frame instead of the <gamma> frame.

The second instruction in the return sequence will reload all 8 index registers and the A, Q, and E registers using the direct address:

sp|8

since sb←sp now holds the desired pointer. Finally, the third instruction

†These are handled by calls to the unwinder, a supervisory routine described in BD.9.05 and discussed fully in Chapter 5.

recovers all other machine conditions, i.e., (ic)+2, (pbr) and (indicators) which were safe-stored as item 5 in the <beta> frame at the time <beta> called <gamma>. The effect of the rtcd instruction, a very fancy transfer instruction, is to resume execution at a place two words beyond the stcd instruction in the calling sequence to <gamma>.

3.7 BASIC STORAGE STRUCTURE FOR AN ARGUMENT LIST

All compilers and assemblers operating in the Multics environment must produce lists of calling arguments that fall into certain standard patterns. Every operating system requires such standard patterns. In the familiar batch operating system on conventional computers, the argument list is usually supplied as a set of pointers or values immediately following the transfer and save (ic) instructions (e.g., TSX in the IBM 7094). The Multics argument list, however, may be stored in an arbitrary location, but is generally kept in the stack in order to keep the procedure pure and to guarantee that it is recursive.† Moreover, the list consists only of pointers. No actual values are stored in the list.

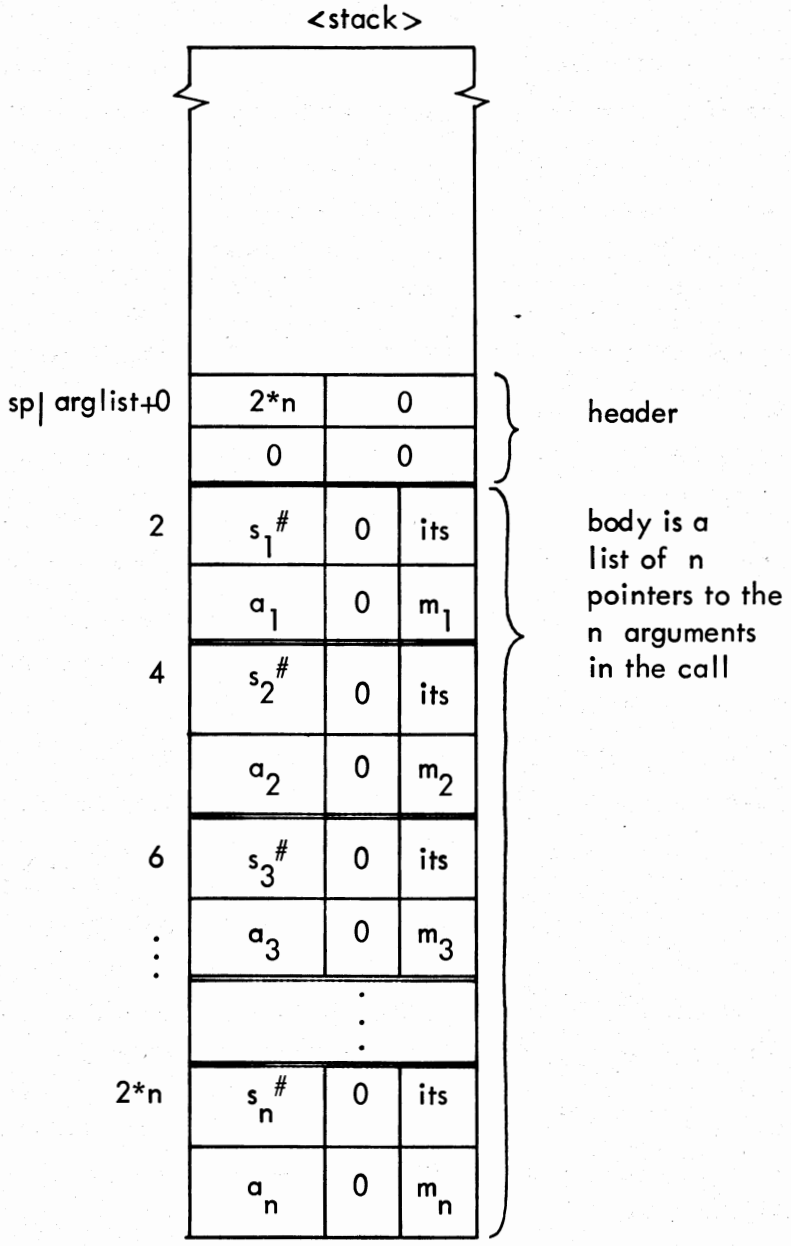
Figure 3-6 shows the "basic" storage structure used for an argument list.≠ It consists of a two-word header, followed by a body composed of n its pair pointers. The length of the body, $2 \times n$ words, is given in the header whose address is sp|arglist.

Each its pair can point independently, either directly or indirectly to a corresponding argument. The its pair normally has a zero modifier, i.e., direct, when the argument is local to the calling procedure. Indirect modifiers, *, are useful when the argument is externally defined as will be explained in a later paragraph.

---

†An argument list must be generated and stored in the stack if at least one of the datum values it points to must be kept in the stack. This means that several copies of the argument list, each pointing to different "generations" of arguments, can be stacked at the same time. By a datum value we mean for example, a variable, label or procedure entry point.

≠This form must be embellished in one of two ways to achieve "**special effects.**" One of the embellished forms is discussed in Section 3.11, the other in Chapter 4.



This structure is for a call to an n-argument procedure segment. The m_i are modifiers which are normally 0 but which may be * (indirect).

Figure 3-6. Basic Storage Structure of an Argument List.

3.8 PUTTING ITS PAIR POINTERS INTO AN ARGUMENT LIST

The creation of its pairs for an argument list and the insertion of them in the list requires one of several coding technique, depending on the kind of argument in the call. Three kinds are recognized here:

(a) argument is locally defined within the calling procedure

(b) argument is a parameter of the procedure, i.e., passed along as an argument by a procedure which called the currently executing procedure

(c) argument is an external symbol

The different coding techniques are alluded to in BD. 7.02 under Notes and Comments. In the remainder of this section we give a small amount of elaboration. Feel free to skip over these details during the first reading. Probably even more detail is needed for subsystem writers who will be writing compilers or assemblers.

We shall sketch how each of the three kinds of argument pointers might be formed by basing our examples on the following hypothetical situation. We imagine source code which shows <a> calling on <b> which in turn calls on <c>. We then focus on the job of the compiler which must construct the code to generate an argument list for a call within <b> on <c>. We further suppose in all instances that this list is to be located at sp|arglistb. Let the i^{th} argument in the call on <c> be given the name xb.

(a) xb is locally defined within <b> and hence its value resides in the stack frame associated with <b>, say at some offset xxb from word zero of the frame. This offset is computable by the compiler. Suitable code to create and store the desired its pair pointer in this case would be:

```
eapbp   sp|xxb            form address of argument
stpbp   sp|arglistb+2*i   store as i^th its pair in arglistb
```

(b) xb is a parameter, say the j^{th} parameter of <b>. In calling on <b> we know that the procedure <a> has provided an argument list with an its pair pointer to this j^{th} argument. Suitable code to form the i^{th} its pair pointer for xb would be:

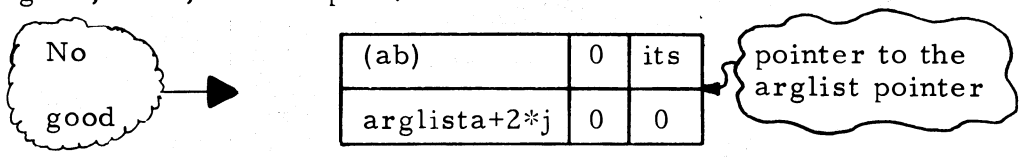
```
ldaq   ap|2*j
staq   sp|arglistb+2*i
```

The ldaq instruction lifts the its pair "bodily" out of the argument list supplied by <a> and puts it into the argument list being constructed at sp|arglistb.

Notice that it would be a mistake to use code such as:

```
eapbp   ap|2*j
stpbp   sp|arglistb+2*i
```

because this instruction pair would store a pointer to the pointer in <a>'s arglist, i.e., the its pair:



where (ab) means the contents of the ab base register.

(c) the i^{th} argument is an external symbol. In this case, the its pair which must be created and put in the argument list includes a segment number and an external symbol, values for which are not known to the compiler at the time it's generating the code that creates this argument list.

For example, suppose the i^{th} argument is to be <data>|[x]. Source code like:

```
eapbp   <data>|[x]
stpbp   sp|arglistb+2*i
```

would, when executed, certainly create the desired its pair, but in so doing would force an ft2 fault to the Linker which must determine data# and x. This is because the generated code will be of the form:

```
eapbp   lp|k,*
stpbp   sp|arglistb+2*i
```

The trouble with this approach is it forces linking at too early a stage. After all, we don't really know if the called program <c> will ever use this argument. So, why link to it during the process of calling <c>? If <c> never uses this argument the early linking could be a costly strategy.

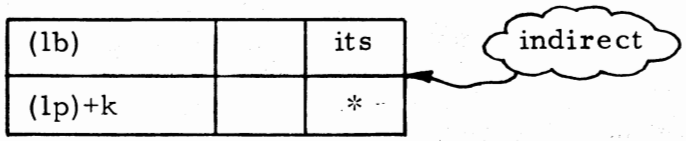
A way to postpone this early linking, is to create an indirect its pair pointer for the argument list.

Coding which could be generated to do the job might look like:

```
eapbp    lp|k              | form address of the ft2 pair
                           | for <data>|[x] in base pair
stpbp    sp|arglistb+2*i   | store the address (i.e., lb|lp+k)
                           | as an its pair
lda      16,dl             | put an indirect code, which is
                           | decimal 16, into lower part of
                           | accumulator. dl means "direct
                           | lower"
orsa     sp|arglistb+2*i+1 | or the accumulator to storage
                           | to form an indirect modifier in
                           | the second word of the its pair
```

An its pointer will then be constructed of the form:

sp|arglistb+2*i



where (lb) and (lp) represent the contents of the lb and lp base registers at the time the eapbp instruction is executed. For a more complete discussion on how to handle such arguments, see BB.2.02.

3.9 STORAGE STRUCTURES FOR DIFFERENT TYPES OF DATA

Thus far we have been discussing lists of pointers to the arguments of a procedure, but we haven't been paying attention to what the arguments themselves look like. Some types of arguments, e.g., integer and real variables are sufficiently simple that their data values are pointed to directly by the pointers in the argument list. Other data types are sufficiently complex in structure that the argument pointers don't point to data values but, alas, to pointers which are part of the storage structure of the individual arguments. The subsystem writer must, of course, keep this in mind in instances where code is being constructed to fetch or store data values via argument list pointers.

At least some of the procedures of every subsystem must interface with Multics system modules. Arguments passed to or from a subsystem procedure and a Multics system module are restricted in type to a subset of PL/I data types.†

†It may surprise you to learn that, although most of the Multics system is written in PL/I, the data types used are a restricted subset within that language. For motivation and full discussion of this point, see BB.2.

Only the following types of arguments may be passed:

(a) All scalars:

arithmetic { integer, real, complex

strings { bit, character

labels (including procedure entry points)

pointers

(b) Any one-dimensional array of the above.

Standardized storage representations (structures) have been established for each of these types and are given in section BB.2.02 of the MSPM. We provide here a convenient summary of these storage structures. These are the items in Table 3-1. In each case, the graphic form 

denotes the item of the argument pointed to by the pointer in the argument list. Shaded boxes denote parts of the argument containing actual data values. Unshaded boxes denote pointers or "dope."

Shown in Table 3-2 are the storage structure conventions established for the remaining data types within PL/I. More details on these latter items can be found in the MSPM documents which relate more directly to PL/I (i.e., BD.1, B0056, BP.2.01, and BP.2.02).

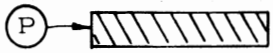
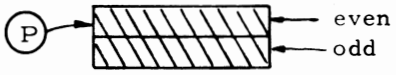
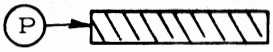
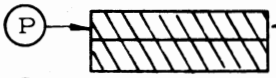
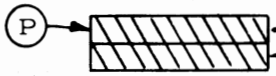
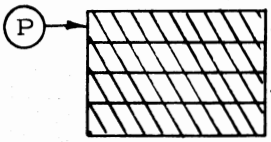
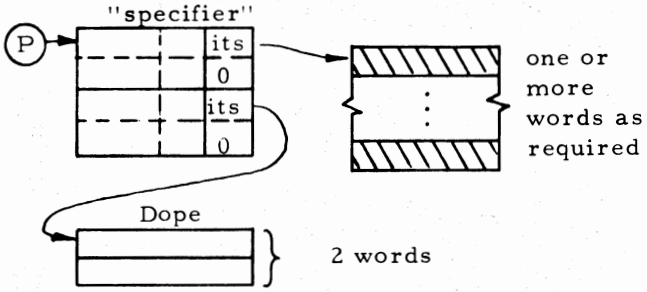
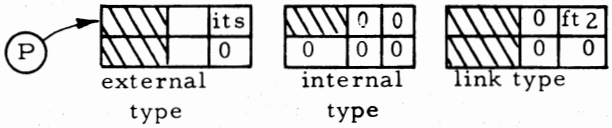
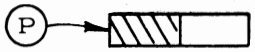
Table 3-2 items should also be of interest to the subsystem writer, but for somewhat different reasons. Thus a subsystem writer who is developing a new language processor, e.g., MAD or ALGOL, may benefit by seeing the way n($\geq$2)—dimensional arrays are structured in the Multics PL/I. While these are perhaps not the best or only methods for representing such data types, two things are worth considering seriously:

(1) These structures are tested and have proven practicable.

(2) Multics will eventually provide an extensive library of subroutines written in PL/I. A subsystem writer who chooses PL/I storage structures can have the automatic by-product of being able to have his subsystem interface easily with (i.e., call directly on) a growing library supported by the Multics staff and the PL/I community. Enough said.

TABLE 3-1

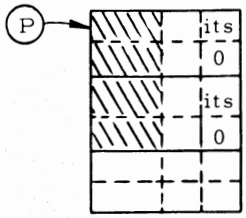
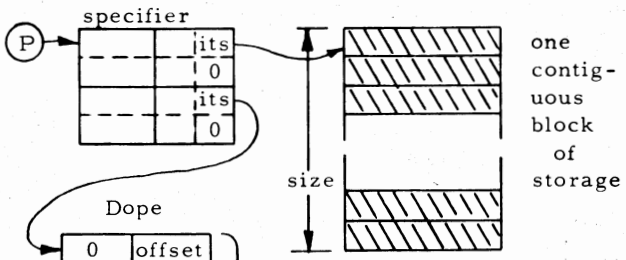
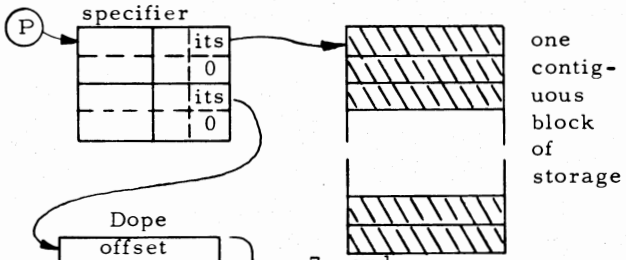
Types of Arguments and their Storage Structures — System-Wide Standards

Type No.†	Argument Type	Storage Structure
	Non-string Scalars	
1	Single-word integer	
2	Double-word integer	
3	Single-word floating-point	
4	Double-word floating-point	
7	Single-word floating-point Complex	
8	Double-word floating-point Complex	
	String Scalars	
9	Non-varying bit strings	
11	Non-varying character strings	
10	Varying bit strings	Same as 9, 11 above except add another its pair to the specifier to point to free storage. See BB. 2.02
12	Varying character strings	
	Program Control Data	
13	Absolute pointer	
14	Relative pointer	

† Numbering here is same as symbol type number (BD. 1, p. 10) used as a code in the segment symbol table.

TABLE 3-1 (continued)

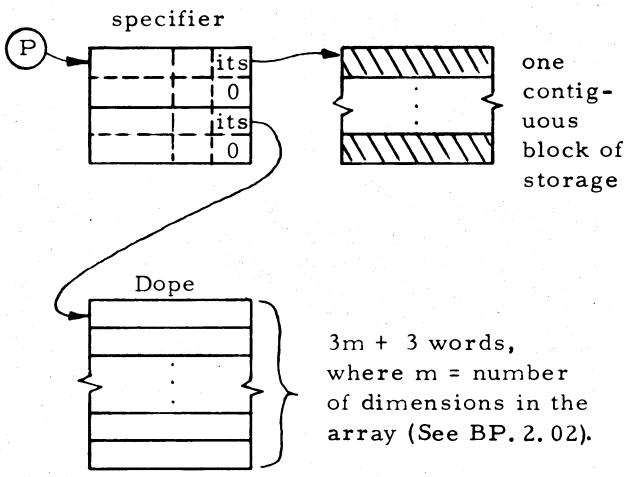
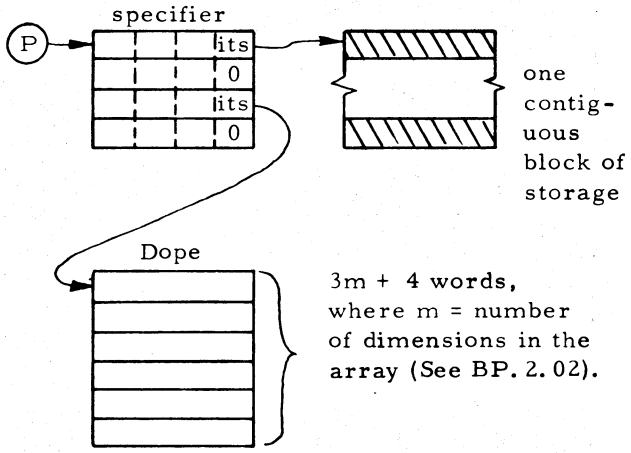
Types of Arguments and their Storage Structures — System-Wide Standards

Type No.	Argument Type	Storage Structure
15	Label	 program point in <x.link> stack pointer‡ not now used
16	Entry	
	same storage structure	
	One-Dimensional Arrays	
17-24	of scalars, types 1-8, and 13-16	 6 words See BB. 2.02 for more details which will reveal its full generality.... id is a nine-bit code that describes the type of the structure and size of the elementary data item
25, 27	of scalars (non varying strings), types 9 and 11	 7 words See BB. 2.02 for more details which reveal its full generality...
26, 28	of scalars (varying strings), types 10 and 12	See details BB. 2.02

‡ to the stack frame that defines the generation of temporary storage appropriate to the program point.

TABLE 3-2

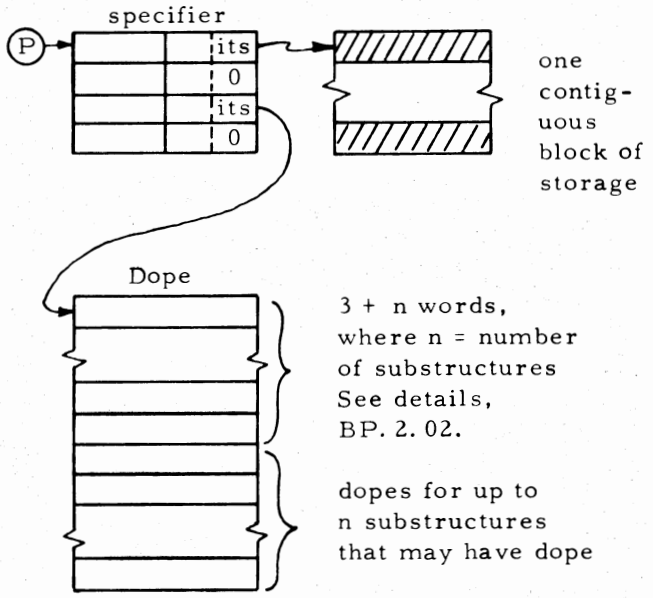
Types of Arguments and their Storage Structures — PL/I Standards Only.

Type No.†	Argument Type	Storage Structure
	Higher-dimensional Arrays	
17-24	of scalars of types 1-8	
	of non-varying string scalars	
25	bit strings	
27	character strings	
	of varying strings	
26	bit strings	
27	character strings	
	(same as 25 and 27, except we add one more its pair to the specifier which points to free storage)	

† See footnote to Table 3-1

TABLE 3-2 (Continued)

Types of Arguments and their Storage Structures — PL/I Standards Only.

Type No.	Argument Type	Storage Structure
	Structures	
	Arrays of Structures	See BP. 2. 02.

3.10 FUNCTION NAME ARGUMENTS, ORDINARY CASE

A procedure <p> may call on another procedure <r>, passing to it an argument that is a procedure entry. Suppose the argument passed to <r> is the procedure entry <q>|[entry3], in a call equivalent to

call <r>|[entry](arglist1).

In this case the argument list contains a pointer to the entry datum which has a pointer to <q>|[entry3]. Some time later, while executing in <r>, we can have a call on <q>, by referring to a parameter, say t, that corresponds to <q>. Such a call might have the appearance:

call t(arglist2)

where t is declared to be the dummy procedure name, and the arguments, x, y, and z are recognized as external symbols for (say floating point) variables located in the segment <data>. A corresponding sketch is shown in Figure 3-7 using PL/I terminology.

The argument list generated in the call to <r> will have the appearance:

arglist1

2		0
0		0
(sb)	0	its
(sp)+arg	0	0

The argument itself we suppose is located at sp|arg. It has the format:

entry datum

sp|arg+0

q. link#	0	its
entrypair	0	0

normally zero (used only for internal functions)

unspecified error check information

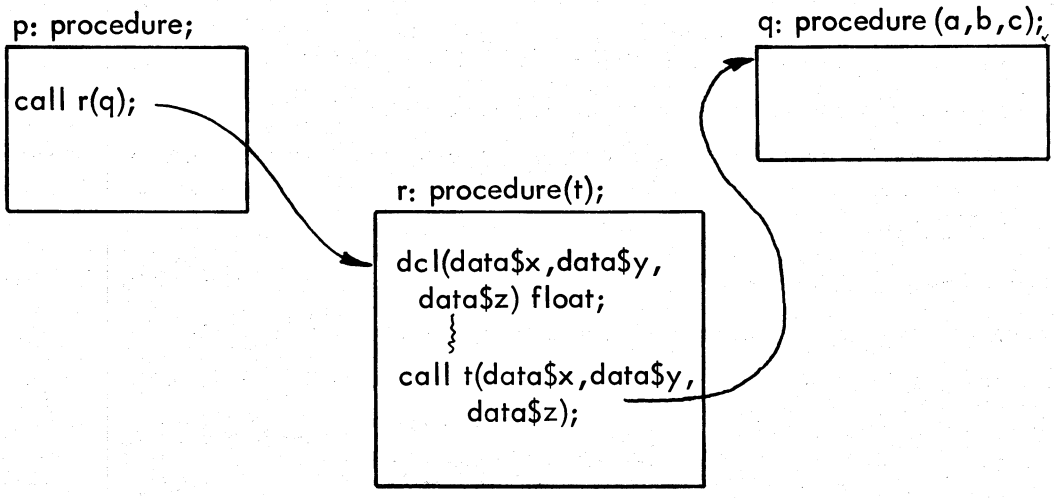


Figure 3-7. Calling an External Procedure Whose Name (q) has been Passed as an Argument.

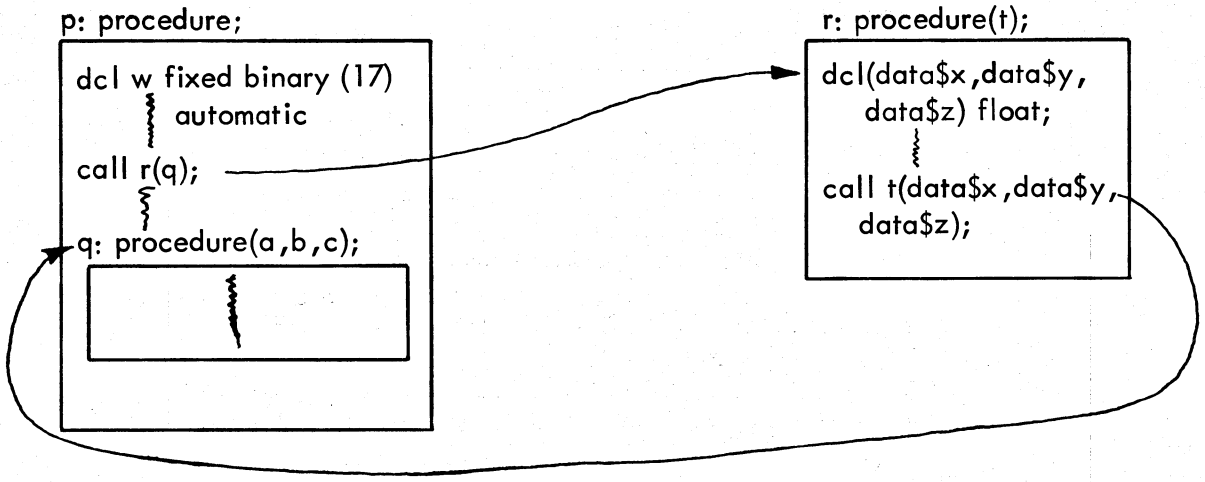


Figure 3-8. Calling an Internal Procedure Whose Name (q) has been Passed as an Argument.

The first its pair points to an offset within <q.link> at which we can expect to find the quadruplet

```
entrypair:  eaplp   -*,ic
            aos     2,ic
            tra     link-*,ic*
            arg     0
```

whose execution basically results in the transfer to <q>|[entry3]. Ordinarily the second pair of words in the entry datum is zero. In certain cases, however, as explained below, it will be used for holding a stack pointer value. The third pair, is to be used for as yet unspecified error checking information.

To generate the call on the dummy t while executing in <r>, the last instruction in the call sequence might be preceded by the instruction:

```
eapbp   ap|2,*
```

to put the address of the function name argument in bb←bp. The standard call sequence would follow, ending with the instruction:

```
tra   bp|0
```

which would cause the transfer to the address found in the first its pair of the function name argument, i.e., ultimately to the desired entry point in <q>.

The argument list that goes with this call on <q> would appear as:

arglist2+0

6		0
0		0
data#	0	its
x	0	0
data#	0	its
y	0	0
data#	0	its
z	0	0

3.11 FUNCTION NAME ARGUMENTS, SPECIAL CASE

Suppose again the procedure <p> calls on <r>, passing to it as an argument an entry point in q. But this time, suppose q is a procedure that's internally defined within <p>. Figure 3-8 illustrates this case using PL/I terminology. We will imagine that the entry point is located at q+entry3 within <p>. Because q is an internal procedure, it may, whenever it is executed, require data values which have been allocated temporary storage in a stack frame created earlier by the containing procedure <p>. Somehow q will have to know how to reach this stack frame. This section explains the Multics conventions which are designed to aid subsystems writers in solving communications problems of this type. Such problems, of course, will occur in subsystems which permit the embedding and or the nesting of internal procedures or blocks within procedure segments.

To continue with our example, suppose then, <p> calls <r> with a call equivalent to

call <r>|[entry1] (arglist1)

The argument list at arglist1 looks just like the one we showed in the preceding discussion. However, the argument itself must now include a stack pointer value for reasons we shall be developing in the next paragraphs. Thus, our argument at sp|arg would now appear as:

entry datum

sp arg+0

p. link#	0	its
entrypair	0	0
(sb)	0	its
(sp)	0	0

Each call sequence that sends control from <p> to <r> must be immediately preceded by code that creates an argument in the above form.

The first its pair points to the entry instructions within < p. link> whose execution would result in a transfer to < p> | q+entry3. (sb) and (sp) refer to values of the sb←sp base pair extant immediately prior to the call. Code to generate the first two of these its pair could be:

eapbp	lp | entrypair	form the address p. link# | entrypair
stpbp	sp | arg	store as the first its pair in the function name argument.
stpsp	sp | arg+2	store stack pointer value as the second its pair.

Proceeding further with our example of Figure 3-8, we now suppose that, while executing in < r> at some point, we wish to execute a call on q via reference to the corresponding parameter t, e. g.,

call t (arglist2)

Here again, t is the dummy procedure name. The arguments x, y and z happen to be externally defined within < data>.

Once the call sequence in < r> to p# | q+entry3 has been completed it must be possible for the computation of q(x, y, z) to proceed successfully. But suppose that while executing in q it becomes necessary to refer to a previously stacked data value such as w that was assigned during prior execution in < p>. Remember that the compiler does not and cannot furnish addresses that are relative to the beginning of the stack segment. It only furnishes addresses relative to the beginning of a stack frame. Therefore, q must know the stack frame pointer that was in use at the time < p> called < r>. Note this is the pointer that would be made a part of the function name argument to be "passed" to < r>. Obviously < r> itself has no need for this stack pointer, but notice that when < r> calls on q, < r> can pass this pointer back to q as an argument. In a sense, the stack pointer must make a "roundtrip" from < p> to < r> and back to < p>. We now see why a function name argument (entry datum) that refers to an internal procedure is designed to include the current stack pointer value.

In just a moment we will examine the structural form of the argument list that must be used in calling on q. Before doing so, we might digress

here to ask several questions: What sort of call sequence should be used in calling an internal procedure? Should the standard call sequence be used? In any case, how should execution proceed in an internal procedure like q once it is called? Should there be a save sequence executed to create a separate stack frame for execution of q? Anyone writing a compiler for a language that has an ALGOL-like block structure, e.g., one which permits the nesting of internal procedures and/or PL/I-like begin blocks, must provide his own answers to these questions.

Mechanisms have been developed by the EPL compiler writers for dealing with these problems. They are well documented in MSPM. Other compiler writers may choose to adopt these techniques. If so, they will find the notes in BN.5.01 very helpful. They should keep in mind, however, that the implementation described there is regarded by its developers as somewhat clumsy and subject to improvement.

The notes in BN.5.01 suggest one mechanism for handling the general problem which arises when the called procedure or block may be nested at any depth within a containing external procedure. The bookkeeping becomes more complicated, because successful execution of the called procedure may require knowledge of a (different) stack pointer for each of the "containing" procedures or blocks. That is to say, a mechanism is developed for knowing, when executing within a procedure or block that is nested at level i, where to find stacked data that was generated by containing procedures or blocks at various "shallower" levels $k < i$. In the EPL implementation, begin blocks are treated indistinguishably from internal procedures. To enter a block one issues a standard call sequence, as if it were a procedure. Once called, the internal procedure executes the standard save sequence to create its own stack frame. Among other things that can be stored in this frame is the (set of) stack pointer(s) to the frame(s) for any outer level procedures that the called procedure needs to refer to. This set of stack pointers is referred to as the "display."

The cost associated with a call to a nested procedure or block in the current EPL mechanism unfortunately increases with increase in the depth of nesting. This objection is serious enough that work is progressing on improved techniques which would have the virtue that calling costs are independent of nesting depth.

We are now ready to discuss the structure of the argument list that is used for calling on an internal procedure. This Multics "standard" takes the form of a simple embellishment to the structure of an ordinary argument list. The extended form is shown in Figure 3-9.

In our particular example we see that the PL/I statement

```
call t (data $x, data $y, data $z);
```

corresponds at assembly level to

```
call t(arglist2)
```

with an associated argument list which would look like:

arglist2+0

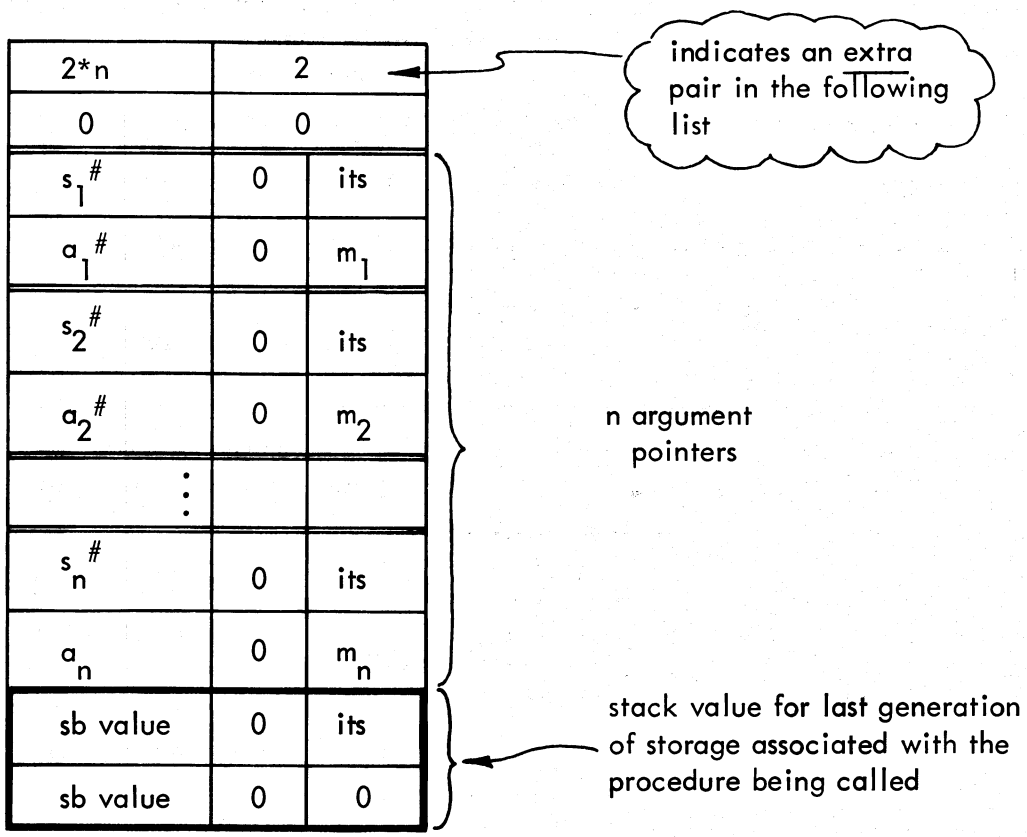
	6		2
	0		0
+2	data#	0	its
	x	0	0
+4	data#	0	its
	y	0	0
+6	data#	0	its
	z	0	0
+8	(sb)	0	its
	(sp)	0	0

copied here from sp|arg+2

In point of fact, eplbsa does not automatically generate the code to construct the argument list in the expansion of the call macro, although an assembler with a more advanced macro capability could be expected to do so. This means that any compiler designed to generate eplbsa code as output must bear the full responsibility for generating code to form the argument list before generating the call macro.

If we were to imagine the use of a more advanced assembler, then conceivably, a source statement like

```
call t(< data>|[ x ] , <data>|[ y ] , <data> | [ z ] )
```



This structure is used when calling on an n-argument internal procedure whose name was previously passed as an argument.

Figure 3-9. Structure for an Argument List.

could be recognized. The assembler would then generate:

(1) the code to form the complete argument list

(2) the call sequence

Such an assembler would have to automatically recognize that t is a dummy that representing an internal procedure in order that it (the assembler) could know to generate code to form the n + 1st its pair of the argument list. A sophisticated assembler could recognize that t is an internal procedure as follows:

Since t is a parameter, the assembler must generate code which, when executed, determines if the entry point that corresponds to t is internal or external. If external there is of course no need to add the n + 1st pointer. The distinction can be achieved by inspecting the second its pair in the entry datum. If it is zero, the entry must be external because otherwise this pair would hold a stack pointer. Code like the following would, if generated, perform this discrimination during execution, and add the its pair as needed, etc.

```
        eapbp   ap|2*i,*          | establish pointer to function name
                                  | arg, assuming it is the ith argument.
                                  |
        ldaq    bp|2              | pick up (sb), (sp) from second its
                                  | pair of this arg.
                                  |
        tze     skip              | bypass†
                                  |
        staq    sp|arglist2+2*n+2 | store stack pointer value as n + 1st
                                  | argument in call on q
                                  |
          .                       |}  adjust word zero of
          .                       |}  argument list
          .                       |
                                  |
skip:                             |
                                  |
                                  |
                                  |
                                  |
```

†In the GE 645, execution of the ldaq instruction causes the zero indicator to be turned on when the loaded double word is zero.

Once inside the called internal procedure, any reference to a local (automatic) variable in the contai ning procedure <p> must be accomplished via the stack pointer argument. In our particular example, a reference to the variable w by the procedure q might be coded something like:

```
eapbp   ap|8,*
lda     bp|w
```

3.12 COMMUNICATION TO AND FROM EXECUTE-ONLY PROCEDURES

Communication to or from execute-only (EØ) procedures justifies separate discussion. The communication process for such procedures is necessarily more complex. Special code must automatically be generated within EØ procedures and in their linkage segments which will control access to them and at the same time allow flexibility in their use and still permit them to be pure, and hence sharable. Thus: Although any call on an EØ procedure must begin execution at word zero, so that the call can be examined for some sort of validity, we still want to allow such procedures to effectively have multiple entries. Moreover, although any return to an EØ calling procedure must actually resume execution at word zero, again, for validation purposes, we still want to permit the writer of the EØ procedure to effectively imply or specify any program point for a normal or alternate return.

You should have little difficulty appreciating the communication objectives and problems, which are clearly set forth in BD. 7. 03. The solutions, in the form of more elaborate call and save sequences are also outlined in BD. 7. 03 although they are perhaps not quite so easy to understand.

Suffice to say that all these special sequences are supposed to be automatically generated by the Multics PL/I compiler and by at least one of the macro assemblers provided in the Multics public library. If you are writing your own target-code producing assembler or compiler and if you want it to be capable of generating EØ procedures, you will find it essential to gain a grasp of all the details in BD. 7. 03. Hopefully, this chapter has prepared you for the challenge.