TO: MSPM Distribution
FROM: R. M. Graham
SUBJ: BD.7.02
DATE: 06/30/67

The attached revision of BD.7.02 contains the following changes:

1. Fields for ring crossing information and subsystem ID are defined in the stack frame.

2. The argument descriptions are further defined.

3. Alternate returns have been deleted and abnormal returns added.

Published: 06/30/67
(Supersedes: BD.7.02, 04/04/66;
BD.7.02, 09/20/66)

Identification

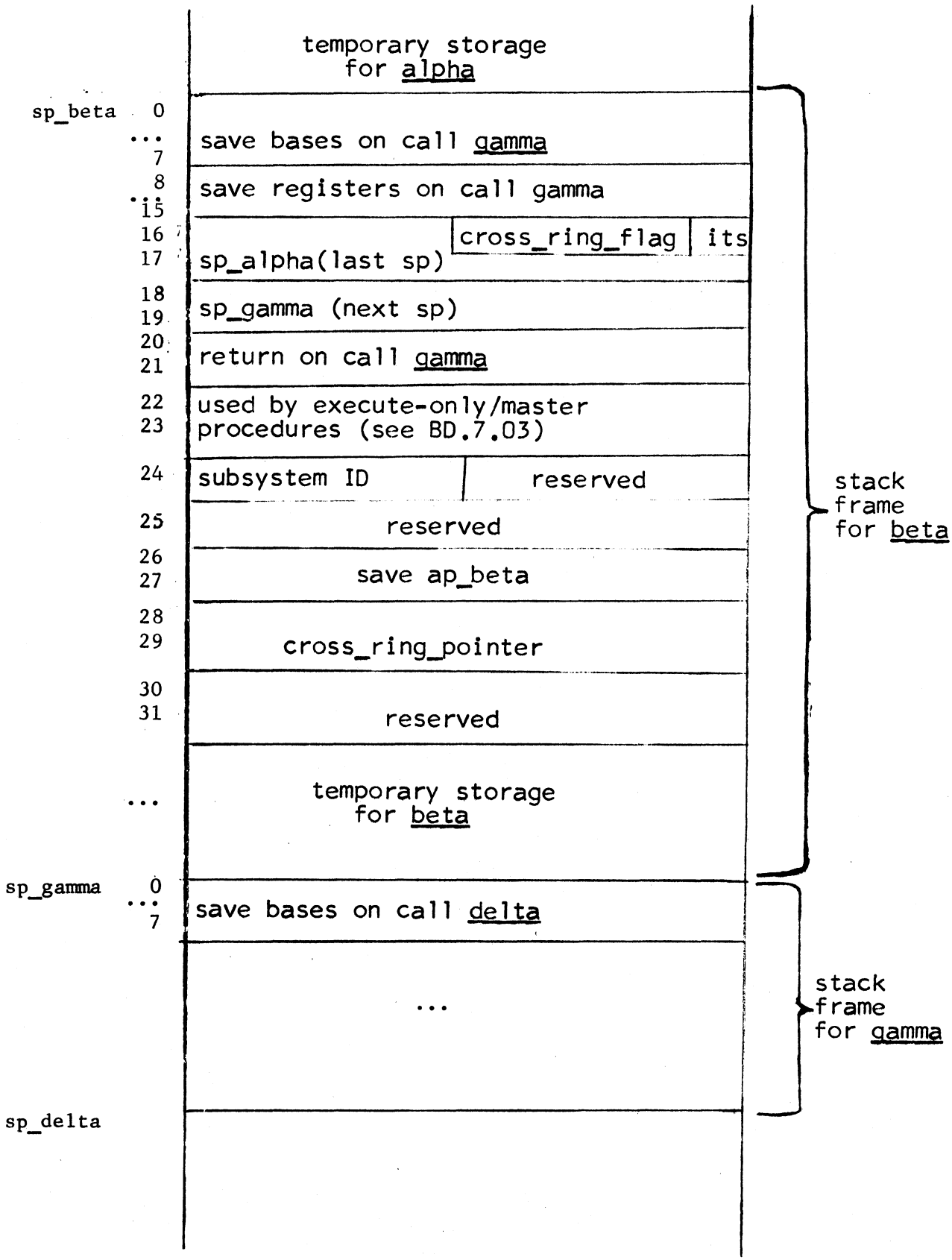
CALL, SAVE, and RETURN Sequences for Ordinary Slave Procedures
R. Montrose Graham

Purpose

Because of the complexity of the hardware, the linkage mechanism, and the stack manipulations, standard call, save, and return sequences for ordinary slave procedures have been adopted (for execute-only and master procedures see section BD.7.03). All translators (including assemblers) must generate these sequences. All supervisor entries, commands, and public library procedures will use these sequences; further, they will assume that other procedures use them.

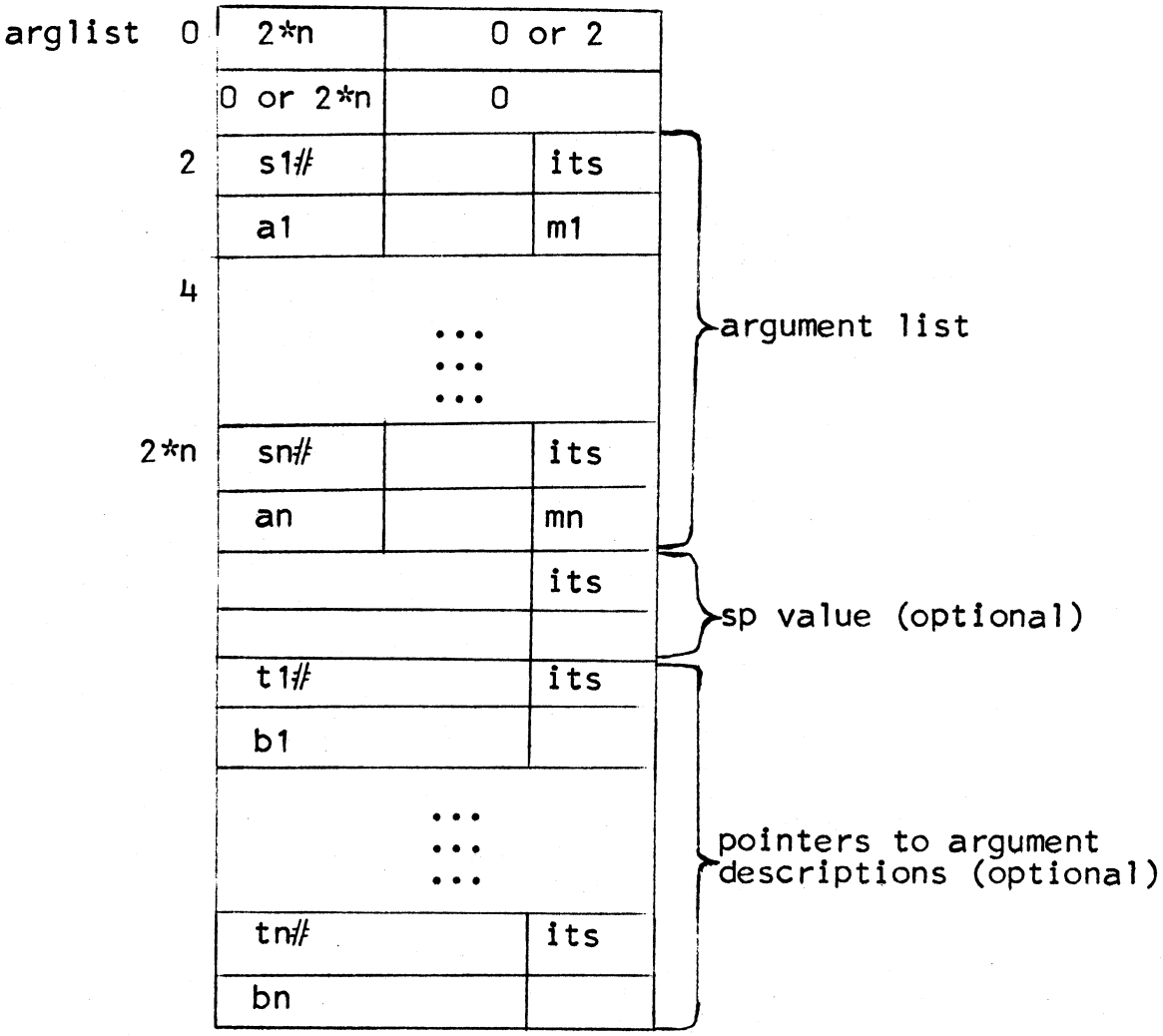
Stack Usage

The standard call, save, and return sequences use the stack. The usage of the various locations in the stack is shown in the diagram. For the purpose of the diagram alpha calls beta, beta calls gamma, and gamma calls delta. Each time a procedure is called, the value of the stack pointer sp (the paired bases sb←sp) is adjusted to point to a new region of the stack segment, called a stack frame, which the called procedures uses. The value of sp must equal zero (modulo 8) since the instructions for storing the bases and registers require addresses which equal zero (modulo 8). When the procedure beta is executing the base pair sb←sp will point to sp_beta (in the diagram). The cross_ring_flag and cross_ring_pointer are used by the protection and abnormal return mechanism (see BD.9.01, BD.9.05). Conceptually there is a single stack per process. All of the active frames in the stack are connected by two threads: a forward thread (sp|18) and a backward thread (sp|16). Actually there is a stack per ring. The frames within a stack are threaded together. The cross_ring_flag, if equal to 1, indicates that control left and reentered this ring immediately preceding that frame, and the cross_ring_pointer indicates where control came from on reentry to this ring. The subsystem ID is an optional 18-bit field available for use by subsystems to further identify this frame.


temporary storage for alpha
sp_beta
0
...
7
save bases on call gamma
8
...
15
save registers on call gamma
16
17
cross_ring_flag
its
sp_alpha(last sp)
18
19
sp_gamma (next sp)
20
21
return on call gamma
22
23
used by execute-only/master procedures (see BD.7.03)
24
subsystem ID
reserved
25
reserved
26
27
save ap_beta
28
29
cross_ring_pointer
30
31
reserved
...
temporary storage for beta
stack frame for beta
sp_gamma
0
...
7
save bases on call delta
...
stack frame for gamma
sp_delta

Argument Lists

An argument list is a sequence of pointers (its pairs) preceded by a count of the number of arguments. Specifically, the form is:



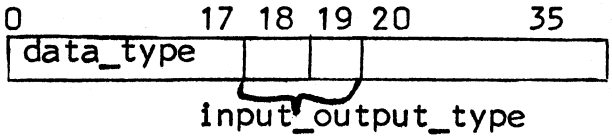
n is the number of arguments. arglist + 2*i and arglist + 2*i + 1 contain an its pair which points to the ith argument (or points to a pointer if the modifier is non-zero).

The ith argument (or pointer to it) is located at <si>|ai, i.e., at relative location ai in segment <si>, whose segment number is si# (see BD.7.00 for description of notation). The argument list must begin at an even location.

If the right half of arglist equals 2, sp value is the value of the stack pointer for the last generation of storage associated with the procedure being called. This is used for a call to an internal procedure which is, in any way, involved in intersegment communication. For example, any call to a procedure entry point which has been passed as a procedure parameter uses this option. The value to put in sp value, in this case, is the second its pair in the procedure parameter. If the right half of arglist is zero, there is no sp value.

If the left half of arglist + 1 is 2*n, a list of pointers to argument descriptions is appended to the argument list. A description of the ith argument is found starting at <ti>|bi. In general, this description will be in the symbol table for the procedure (see BD.1.00), however it may be anywhere. This descriptive information is used in such cases as the callback of the protection mechanism (see BD.9.01) and interprocess calls.

As a minimum requirement, these pointers must point to a word which has the format,



where data_type is a code describing the argument and
input_output_type = 1 if the argument is input only for the called procedure
= 2 if the argument is both input and output for the called procedure
= 0 if the input/output type of the argument is unknown

The interpretation of data_type is defined in BD.1.00. Only those argument types described in BB.2 (scalars and 1-dimensional arrays of scalars) are normally supported in user interfaces to the system. While the above is all that is required, the established convention is that these pointers point to a full segment symbol table entry for the argument.

Call

The standard calling sequence is:

```
stb     sp|0        save bases
sreg    sp|8        save registers
eapap   arglist     establish argument pointer
stcd    sp|20       save return
tra     entrypoint  transfer to called procedure
```

arglist is the location of the argument list. There is no restriction on the type of address which may be used for arglist. The argument list must be generated before the calling sequence is executed.
arglist=0 if no arg list

entrypoint is the entry point of the procedure being called. entrypoint may be any type of address.

Save

The save sequence is distributed between the called procedure and the called procedure's linkage section. Let <lib>|[entry] be the entry point of the called procedure, then the following portion of the save sequence appears in the linkage section of <lib> at <lib.link>|in:

```
in      eaplp     -*,ic       established linkage pointer
        tra       inx-*,ic*   transfer to procedure segment
        ...
        ...
```

inx	lib#		its	points to actual
	entry		0	procedure entry

The remainder of the save sequence appears in the procedure segment itself at <lib>|[entry]:

```
entry   eapbp     sp|18*      establish new, last sp
        stpsp     bp|16
        eapbp     bp|t        establish new, next sp
        stpbp     bp|18-t
        eabsp     bp|-t       establish stack pointer
        stpap     sp|26       save ap for this call
```

The constant t=0 (modulo 8) and must be greater than or equal to 32. t is the number of stack words needed for temporary storage by the called procedure, however in the above sequence t must be less than 2**14. If more space than that is needed the above save sequence may be optionally suffixed by,

```
adbbp     xt,du
stpbp     sp|18
```

where xt may be any 18-bit constant. The total amount of stack reserved will then be t+xt. The definition, in the linkage section of <lib>, of the symbol entry is marked so that the linker will not establish a link directly to the procedure segment <lib>, but to its linkage section at <lib.link>|in (see BD.7.01 for details). After completion of the save, sp (the base pair sb←sp) points to the beginning of the region, in the stack for use by the called routine, lp (the base pair lb←lp) points to the beginning of called routine's linkage section, and ap (the base pair ab←ap) points to the beginning of the argument list for this call.

The save sequence displayed above has a critically important, although non-obvious, property. At all times sp|18 contains a valid its pair pointing to the top of the stack. The save sequence prepares the next frame before loading sp with a pointer to the new frame. This property is required because of the following action by the system when an interrupt occurs. Using the contents of sp at the time of the interrupt, the contents of sp|18 is used to find the top of the stack. Then 32 is added to this value (in case the interrupt occurred during preparation of the next frame) to obtain a pointer to free space in the stack which may be used by the system. The save sequence is also designed to execute without causing a fault when the base sb is locked.

Return

The standard return sequence is:

```
ldb       sp|16,*
lreg      sp|8
rtd       sp|20
```

It is possible to return control to some point other than the location immediately following the call, such a return is called an "abnormal" return. The location to be returned to abnormally, abnrtn, is specified by label data (see BB.2) and may be transmitted as an argument or as the contents of an external (global) location. Since abnormal returns may imply ring crossings, the abnormal return sequence is actually a system procedure (described in BD.9.05). To effect an abnormal return,

```
call unwinder (abnrtn)
```

Notes and Comments

The call, save, and return sequences assume the standard pairing of the eight base registers; i.e., ab←ap, bb←bp, lb←lp, and sb←sp. They further assume that at least sb←sp has the same value that it did when the procedure was entered. Once a procedure has been entered, bb←bp may be used freely. lb←lp must be preserved for the standard linkage machinery to work. It is extremely dangerous to change the contents of lb←lp. The base sb will be locked and its contents cannot be changed by the user. It is essentially disastrous to change the contents of sp. The user may generate argument lists by any method that he chooses; however, the following suggestions constitute a rather simple method. The simplest way to generate a pointer to the location place is:

```
eapbp     place
stpbp     pointer
```

pointer must be an even location. This method works for all types of addresses except when place = ap|2*i, which generates a pointer to a pointer. To avoid this, use the sequence,

```
ldaq      ap|2*i
staq      pointer
```

which moves the ith argument pointer. When place is an external reference such as <seg>|e, or <seg>|[x], normal reference to the linkage section, i.e., lp|k,* (see BD.7.01 for details), will cause linking to occur at the time the pointer is generated if the sequence,

```
eapbp     lp|k,*
stpbp     pointer
```

is used. The sequence

```
eapbp   lp|k
stpbp   pointer
lda     16,dl
orsa    pointer+1
```

will generate a pointer, containing an indirect modifier, to the linkage section entry at lp|k, i.e., a pointer of the form:

(lb)		its
(lp+k)		*

where (lb) is the segment number in the external base lb and (lp) is the contents of the internal base lp. Since indirect modifiers may or may not occur in the its pointers of the argument list, the called routine must assume that they do. The safe way to obtain the value of ith argument is:

```
lda     ap|2*i,*
```

if it is a single word scalar, or

```
ldaq    ap|2*i,*
```

if it is a double-word scalar.

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

loads bb←bp with the address of the ith argument. If the argument is an array or string, the accessing is more involved since the argument list entries point to specifiers (see BB.2). It should be noted that the call, save, and return sequences use only the eight base registers. Hence, all registers are preserved across the call in the caller. Further, all registers except for the bases, are preserved in the transition from calling program to called program.

EPLBSA Notes

The bootstrap assembler contains the following built-in macros.

```
 i)  call entrypoint
ii)  call entrypoint (arglist)
iii) save
iv)  return
```

These macros expand into the standard sequences described above. In case (i), no argument list, the assembler substitutes eapap = 0 for eapap arglist in the call sequence given above. In EPLBSA writing,

```
eapbp     <seg>|[ext]
```

will always cause

```
eapbp     lp|k,*
```

to be generated. Writing the pseudo-operation,

```
link  k,<seg>|[x]
```

will define the symbol k to be equal to the relative location in the linkage section of the link for <seg>|[x]. Now the user can write,

```
eapbp     lp|k
stpbp     pointer
lda       16,dl
orsa      pointer+1
```

which will generate a pointer (containing an indirect modifier) to the link for <seg>|[x] without causing linking.