To: MTB Distribution

From: Melanie Weaver and Steve Webber

Date: January 23, 1980

Subject: New Call/Push/Return Strategy

INTRODUCTION

This MTB proposes a new call/push/return (CPR) strategy for the standard Multics environment. The proposed changes are very large and have many implications. Yet, from recent data, it appears that the changes are warranted. With the current mechanism the cost can be so high the programmers do very unstructured things to avoid the cost. Those programmers who cling desparately to their ideals of well structured software are discouraged by the nontrivial cost in program execution time to use these techniques.

The proposed changes are also needed as we go to highly pipelined hardware. We would like to reorganize the instruction sequences to minimize turbulence in the pipe.

The end result of the project, including extensive performance analysis after the implementation is complete, will give valuable input to design of firmware oriented instructions on any new hardware developed.

EVIDENCE

There are three separate experiments that indicate that a very large amount of CPU time is spent in the CPR overhead. These experiments are (1) a special version of the system (bootloaded on the CISL development machine) that counted the number of calls (MTB-410), (2) a special version of pl1_operators_ used to measure argument list preparation overhead, and (3) data collected with the use of the sim_6180 user ring simulator.

MTB-410 concludes that the overhead of the CPR mechanism in Multics (not including argument list preparation) is about 20%. This number is reinforced with data collected from the simulator. The simulator data is summarized in Table 1. The simulator data also reinforced the findings of the special version of pl1_operators_ that looked into argument list preparation cost.

Table 1
Call/Push/Return Overhead

Script	Total Instructions	Instructions in CPR	Instructions Preparing Arg List
pl1 compile	933447	188617 (20%)	12047 (1.3%)
pl1 compile with -ot	1146476	287484 (25%)	16947 (1.5%)
command script	377342	56388 (15%)	6364 (1.7%)

The simulator data does not include ring 0 runs which are quite different in one respect. A large part of the time spent in ring 0 is spent in the assembly language programs constituting page control, interrupt intercepting (ii), fault intercepting (fim, signaller, etc.), and the traffic controller (pxss). These routines do not labor under the weight of the fully general CPR overhead--indeed much of the reason for placing this code in assembly language was to avoid the CPR overhead. Ignoring these assembly language programs, the rest of ring 0 should behave a great deal like ring 4 programs since it is written in PL/I. Hence, if we can expect to get an N% increase in performance with another CPR strategy, we should expect this same gain in segment control, directory control, and tty interrupt handling.

We can then predict what this improvement might mean to system performance. The proposed PL/I CPR sequence for an external entry (see attached code) uses 27 instructions as compared to the current 44 instructions. This saves about 39% of the CPR overhead for a total non ring 0 saving of about 8%. It is estimated that about 50% of the system's time is spent in ring 0 with about 40% of that in alm, so the total system performance gain should be about 6%. Furthermore, there are additional changes that can be made to the binder to optimize intrasegment calls. These changes are discussed later in this MTB.

A further optimization could be accomplished with a new argument list strategy that uses ITP pointers. However, the gain in this case does not seem worth the effort.

HISTORY OF THE CURRENT APPROACH

If the current CPR strategy is so expensive and more efficient alternatives are available, then why are we still using the current costly scheme? The answer is simply that the implementation and integration costs were thought to be too high to warrant the "expected" gains. Well, the expected gains, due to the recent set of experiments, are higher than previously thought--hence this MTB. However, many of the expensive features were put there for valid reasons. A few of the more important are:

1. The use of an operator segment to implement the algorithms
This is costly not only because it means more instructions getting into and returning from the operators but also because much information must be passed to the operators (in registers) so that operators alone can know where to load and store the values.

This strategy, however costly it is, should not be abandoned. We could not be contemplating the changes proposed in this MTB if we had not isolated the protocols into a few replaceable modules. Had we firmware to play with we might think of placing this kind of logic in firmware, but we would probably not consider very much, if any, in hardware. Similarly, we should not place any in the object modules of the system (except, of course, the system support facilities that are as easily replaced as the operators). Some of the binder optimizations discussed below do put all the CPR logic in the object segment, but users of these will be aware that they may have to rebind if the CPR protocol changes, and these bound segments will be easy to find.

2. Saving and restoring indicators
There is some confusion in the Multics community as to the exact degree to which indicators are saved and restored across a call. On the 645 the stcd and rctd instructions stored and restored the indicators. These instructions were used to effect part of the CPR algorithms. On the 6180 hardware this feature was not provided (due to our error in specifying the instructions). On the 645, saving and restoring the indicators was effectively free. On the 6180 we have to simulate the effect and the cost is sometimes challenged.

The policy on saving and restoring registers and indicators across a call is as follows:
1. All registers needed by a calling program must be saved and restored by the calling program; except
2. Pointer registers 0 (operator pointer) and 6 (stack frame pointer) as well as the indicators are restored from saved values upon return.

This means that if a program wants the indicators (or its operator pointer) restored to some value, the value must be set up in the standard stack frame location prior to the call. PL/I and FORTRAN set up these values when a stack frame is created so that the overhead need not be done on each call. (Note that although these values are restored returning to an ALM program, PR0 does not carry the semantics of an operator pointer and PR0 can therefore be used in such a way that it is restored automatically by the callee, if it has been saved appropriately.)

Hence, indicators are reloaded upon return to a program but unless they were saved just prior to the call, they are not preserved across the call.

The overhead of dealing with indicators at all is due to the desire to guarantee that a program can expect a fault if an overflow or underflow condition is detected by the hardware. Were the indicators not restored, a called program could mask overflows and underflows for the caller and change the behavior of the program it returns to. (This could, of course, be effected by changing the saved value of the caller so that when returned to it had the new indicator values.)

Basically the only purpose that restoring indicators serves, then, is to protect the ability to detect overflow and underflow faults. This could be changed so that any programs that wanted this protection could set the indicators each time returned to. To do this would require compiler changes and might well be too large a project. An alternative is to have the hardware do much of the work. The rtcd and stcd instructions, if they worked as on the 645, would solve some of the problem. We would also have to initialize the indicators during the entry sequence (or about that time). One possibility is to have the callsp instruction set the overflow mask, overflow/underflow indicators, and, if we can get the feature, the underflow mask to 0, the most useful initial state. It is also possible that the stcd instruction could set these indicators (after storing all the indicators) but this is less promising as many potential CPR algorithms cannot use the stcd instruction (it would mean embedding knowledge of stack offsets in object programs).

The cost of indicator management as a part of the CPR mechanism is about 1% of the system. The proposal is to get improvements in future hardware, but to live with overhead-in-software until then.

3. Doubly-threaded stack frames
All stack frames currently have both the backward and forward threads. The main reason is to ease debugging. It is clear that a backward pointer is required, but the forward (next) stack frame pointer is not needed, as indicated by the contained proposal.

4. The "Stack-End-Pointer" Concept
Each stack has in its header a pointer to the next available space in the given stack. This pointer is updated on each call and again on each return. It is needed when:
- a new stack frame is created by a called program;
- a call to an inner ring stack is made; and
- the signaller "caps" a stack and creates an environment in

which signal_ can be invoked.
The first use is easily satisfied with alternate techniques.

The inner-ring call of today is an elegant method set up in such a way that a program called in an inner ring need not know that it was called from an outer ring or that it is the first program to establish a frame on the inner ring stack. i.e., the "push" algorithm works identically for intra-ring and inter-ring calls.

The work of capping a stack is done by a privileged ring 0 routine that today uses the stack end pointer in the stack being signalled on. This is convenient although potentially error-prone.

5. Obsolete Code
There is some code executed as part of the CPR facility in maintenance of obsolete features (e.g. version 1 PL/I).

6. Missed Optimizations
As a final note, there were several optimizations that were either missed or postponed due to the development effort necessary to take advantage of them.

THE_NEW_PROPOSAL

The proposed new CPR strategy makes the following changes to the conventions and implementation (each followed by a brief description of the implications):

1. No more support for Version 1 PL/I.

This is not a serious problem since the compiler itself was never released to the field. It did exist at MIT but MIT is willing to (and encourages us to) delete support for it. The only system code still in v1pl1 is now obsolete and no longer used (it should be deleted when we get the time).

2. Single-threaded stack frames.

This will cause a problem to all code, both system and user, that currently decodes stack frames for tracing-like purposes. The system code is isolated in the debuggers, trace_stack, the signalling mechanism, and various other tools.

3. No more stack end pointer.

This will require changes to the signalling mechanism (for capping the stack) and to all gates. Gates will no longer be able to use the "normal" push sequence since it will be streamlined for the much more frequent intra-ring calls.

Rather, gates will be required to lay down the first stack frame using the "stack-begin-pointer" in the header of the inner ring stack. This is not a difficult problem to solve since gates must be in ALM today anyway if they want to take advantage of the call limiter hardware.

The advantages gained by not using the stack-end-pointer discipline are efficiency, simplicity, and less I/O on page zeroes of stacks (they usually don't get modified except for updating the stack-end-pointer).

4. Changes to the format of the call sequence, entry sequence and stack frame.

   The basic premise is to allow the compiler to provide as much information as possible at compile time and at the same time minimize the push overhead.

All of the above changes will require extensive changes to the various operator segments (PL/I, COBOL, etc.) and to a lesser extent, the runtime support facilities.

PL/I CHANGES

The changes proposed for PL/I in the basic inter-segment case are listed below:

- Fill in the argument list header in the object code in a more efficient way instead of in the call operator.

- Get a pointer to the argument list in the object code instead of just the offset in index register 1.

- Get a pointer to the return location while in the object code and store it during the call operator. This replaces transferring to the call operator via a tsx0 and then just filling in the return offset. The return pointer is no longer initialized during the entry sequence.

- Make the compiler-generated offsets of the call and return operators point to the actual operators instead of to the transfer vector. This means that the operators' locations are frozen but it also avoids transfers out of the transfer vector.

- Do not set PR4 to the linkage pointer in the call operator (this was for version 1 PL/I).

- In the object code entry sequence, load a double word containing the stack frame size, original stack frame size (need both fields; one must not be changed during stack

extensions), entry offset, and translator ID instead of just loading the stack frame size.

- Transfer to the entry operator via a pointer in the stack header (as ALM currently does), thus avoiding loading a pointer to the operator transfer vector and transferring out of the transfer vector.

- In the entry operator, store the double word containing the frame size, entry offset, etc. This replaces getting a pointer to the entrypoint, storing it, setting the forward frame pointer, and setting the stack end pointer. Also do not store a copy of the forward pointer at spl4 for use when making temporary frame extensions.

- Fill in a new variable, stack_frame.operator_return_ptr with a pointer to the base of the object segment. This replaces the text base pointer and the operator return offset. All languages must initialize this variable since it will be the standard location of the stack frame owner's segment number.

- In the return operator, do not reset the stack-end pointer or reset PR7 to point to the stack header.

THE_BINDER

One of the improvements considered in the past but never worked on due to manpower demands is a binder that performs a set of optimizations. These optimizations range from doing inline (nonoperator) calls to sharing stack frames. Some of these binder enhancements are relatively easy to do. These will be considered in this MTB. Another reason for considering them now is that they require minor compiler changes to the call and entry sequences, and that area of the compiler is already affected by the rest of the proposal.

There are three progressively more efficient CPP protocols that can be used between components of bound segments. They are:

1) optimization with operator support,
2) optimization without operator support, and
3) optimization to make components act like quick internal procedures.

It is possible that a single bound segment might contain all three protocols, depending on the attributes of the component entries. The optimizations will cause the binder to be knowledgeable about specific language implementations. It is proposed that these optimizations initially apply only to PL/I and FORTRAN components.

The first protocol is used when the called entry is retained (callable externally) or when it is desired to keep the CPR

mechanism in the operators. The second protocol is used when the called entry is not retained but does require its own stack frame. The third protocol is used when the called component is neither retained nor recursive, so that it can share the stack frames of all of its callers. Protccols 2 and 3 imbed the CPR mechanism into the object segment, so any bound segment using them would have to be rebound if the CPR mechanism changes again. This is much easier than recompilation, however. A new flag in the object map indicating this degree of optimization will both facilitate locating these segments and enable the linker to detect them so it can log access to them or refuse to link to them (or neither) as the user requests.

Protocols 2 and 3 should not be enabled by default because they embed potentially incompatible code in object segments. Instead they will be controlled explicitly. The decision about whether to use protocols 2 and 3 will be determined by the bindfile or a control argument (independently). To enable the binder to perform the transformation, code must be be extended; the compiler must provide space for this at certain places in the object segment. This feature will be controlled by a new compiler option.

Prototype code sequences for protocols 1 and 2 are included in this MTB. Basically, the differences between protocol 1 (binder optimizations with operator suppport) and the proposed standard CPR protocol are:

- Bypass the call operator and the transfers to and from the entry sequence in the object code. Instead, pick up in the calling sequence the double word containing the frame size and set the return pointer in the entry operator. For this the compiler must generate a pad word in the new call sequence.

- Use the transfer vector when transferring to the call/entry operator so that these entries can be traced.

- Do not set PR0 to point to the argument list and then reset it to the operator table. The new entry_from_bound_seg operator uses a different pointer register convention for argument lists.

- Do not load the indicators in the entry operator. Since the caller is in PL/I or FORTRAN, the indicators are guaranteed to be unmasked before the call.

- Do not obtain a new linkage pointer (PR4) value since the current value will be required to be valid. (It is used to get a pointer to the routine to call.)

- If no entry in the called component is retained, a new return operator that does not load the indicators can be used.

CPR protocol 2 (binder optimizations without operator support) differs from protocal 1 primarily in that the CPP code is embedded in the object segment at the entry and return points. This eliminates several instructions transferring to and from the operators.

CPR protocol 3 (optimization to make components act like quick internal procedures) eliminates the overhead of pushing and popping stack frames. Although the details have not yet been worked out, the call/return overhead in this case is reduced to about 7 instructions. However, there are several restrictions to the use of this protocol. First, none of the components that are to use this optimized protocol can be callable from outside the bound segment. Second, the combined stack frame size cannot exceed 16K words. Third, none of the components that share a stack frame can be recursive either internally or within the bound segment (except for the component that does the pushing). To aid the binder, the compilers must turn on a new object map flag if a program is not recursive. All stack references would have to be relocated and those in the first 64 words of the frame special cased. Care must be taken so that large areas of a stack frame are not wasted. The bindfile syntax will be changed to allow control of this feature.

ALM_CHANGES

In ALM, the operations performed by the PL/I entry operator are split between the entry and push operators. The push operator may not be invoked at all or may be invoked in a "subroutine" used by several entries. For this reason, the double word containing the stack frame size and entry offset will be copied by the entry operator. Thus the assembler will have to calculate a stack frame size even if there will be no frame.

ALM programs depend on PR7 always pointing to the base of the stack. Now that the PL/I/FORTRAN return operator will no longer reset PR7, the ALM short_call pseudo-op must be changed to reset it. All ALM programs using the short_call pseudo-op must be reassembled before the new CPR protocol goes into effect. The call pseudo-op already restores all the registers.

Some ALM programs take advantage of the fact that the first instruction of the expanded push pseudo-op is eax7 stack_frame_size by picking the frame size out of the instruction. These programs must be changed before being reassembled with the new pseudo-ops. The proposal is to create a

new ALM pseudo variable, %stack_frame_size, which is flexible and will free these programs from dependence on generated code.

The changes proposed for the ALM operators are described below. There is not as much improvement as in the PL/I case.

- Use the current frame's size instead of the stack end pointer to determine where the next frame begins.

- In the entry operator, store the double word containing the stack frame size, entry offset and translator ID where the new stack frame will be.

- Transfer to the new entry and push operators via the pl1_operators_ pointer in the stack header instead of through a direct pointer in the stack header. The ALM return operator is the same for both old and new code, so transfers to it can remain the same.

- In the push operator, use index register 7 to hold the old frame's size.

- Do not set the stack end pointer.

- Set the operator return pointer.

GATE_CHANGES

Gates can no longer use the standard push and entry operators because their stack frames are not usually contiguous with their callers' frames. When entering an inner ring, the stack can be assumed to be empty, so the stack begin pointer in the stack header can be used to determine the beginning of the gate's frame. However, sometimes a gate call does not cause a ring change; for example, dprint, which uses message segments, running in ring 1. Because of those cases, the gate push code must check whether a ring (stack) change has occurred. If it has, the stack begin pointer is used. Otherwise the new frame is located at the end of the old frame as usual.

The extra overhead of checking for ring changes is partly offset by some optimizations available to gates. Gates do entry and push at the same time, so since there must be a new operator anyway, it can combine both functions. Besides, the standard entry operator does several things that are useless to gates. For example, gates invoke the entry operator in a "subroutine", making the entry offset the same for all entries in a gate segment. If tsp3 is used instead of tsx2 to transfer to the setup "subroutine", PR3 can be used to obtain a more realistic entry offset. However it still will not be the "real" one in the

gate's transfer vector. Other changes are mentioned in the more detailed list below.

Currently, hardcore gates do not use the alm entry operator. They obtain the linkage pointer from a location in the text section that is initialized by init_hardcore_gates, rather than from the LOT. The only justification for this appears to be a saving of one instruction. Gates would be simpler if this mechanism were eliminated. If that cannot be done in general (because fast gates would be one instruction longer, etc.), at least it should be done in the standard hardcore push case. Then the same push/entry operator can be used for both hardcore and non-hardcore gates.

The proposal is to have a new gate push operator which is transferred to via the pl1_operators_ transfer vector. Before the new CPR protocol goes into effect, all non-hardcore gates must be reassembled with new gate macros that use the gate_push operator. Both a temporary version of the operator and the reassembled gates must then be installed. After that, when the system that changes the CPR protocol is installed, all the gates will automatically work correctly. An alternative would be to put the gate push code in the gate macros themselves. In order to be compatible across protocols, the macro gate push code would have to check a flag in the stack header to determine what type of frame to push. If the macros were not compatible, the non-hardcore gates would have to be installed in ring 1 using non-standard procedures just after the system came up. The answering service would not be able to use message segments, etc. (e.g. dprint anything) until this was done. Of the three methods, using a new operator seems the most straightforward.

The list below summarizes the proposed actions related to a gate push.

- Transfer to the setup "subroutine" via a tsp3 instruction.
- Transfer to the gate push operator using the pl1_operators_ pointer in the stack header and the pl1_operators_ transfer vector.
- In the operator, compare the segment number portions of PR6 and PR7. If they are the same, locate the new frame at the end of the current frame as usual. Otherwise, use the stack begin pointer in the stack pointed to by PR7 to locate the new frame.
- Store prev_sp, arg_ptr and operator_return_pointer.
- Use an mlr instruction (to avoid changing registers) to copy the words containing the stack frame size and translator ID from the location following the transfer to the operator.

- Store rel(PR3)-1 as the entry offset.

- Obtain the linkage pointer from the LOT and store it (in hardcore gates too).

- Do not obtain and store the static pointer since it is not needed by gates.

MODULES_TO_BE_CHANGED

Following is a list of most of the system modules that are affected by changes to the CPR protocol. Rather than presenting a complete list of all procedures, it is more useful in some cases to state categories.

hardcore
BOS stack dump
parts of system initialization
all gates
parts of page control
scheduler
the signalling mechanism
outward_call_handler_
pl1_operators_ and friends
cu_
bound_error_handlers_
alm programs that use short_call
alm programs that depend on the push pseudo-op to generate
eax7 stack_frame_size

non-hardcore
all gates
trace_stack
bound_debug_util_
trace
probe
debug
stu_
other language operators
fortran_io_
operator_names_
ol_dump
gate_meters
all compilers and alm
ifd
alm programs that use short_call
private stack switching programs, etc.

In addition, a significant amount of MPM SWG documentation must be updated.

ORDER_OF_INSTALLATION

This section lists the steps involved in implementing the proposed changes. They are listed approximately in chronological order except that many of the changes can be coded ahead of time.

- install current pl1_operators_ with new gate_push operator
- set bit in stack header indicating frame format

- change gate macros to use gate_push (via macro, not pseudo-op)
- reassemble and install all (non-hardcore) gates
- change everything in hardcore that needs changing
- change trace_stack and bound_debug_util_ compatibly
- install trace_stack and bound_debug_util_
- change and install assembler to reset PR7 during short_call
- reassemble and install all alm programs that use short_call, including user programs
- make the rest of the non-hardcore changes incompatibly
- install new system including untested new operators
- install incompatible non-hardcore programs immediately afterword in a special session
- change PL/I, FORTRAN, ALM and COBOL to generate improved call/entry sequences
- change binder to optimize call/push/return ("quick" external procedures may be delayed)
- install pl1_operators_ with debugged new operators
- install binder
- install improved compilers (PL/I, FORTRAN and ALM will now use the new operators)

NEW_CODE_SEQUENCES

This section presents the proposed new code sequences to be used in the operators. The instructions in the operators are indicated by a vertical line in the left margin. All other instructions are in the caller's or callee's object segments. Code in parentheses is not considered to be part of the CPR mechanism. Argument list preparation is not included. The code sequences have not been completely optimized for pipelined hardware. The PL/I versions are prototypes, since there are several PL/I entry operators.

PROPOSED PL/I INTER SEGMENT CALL SEQUENCE
(Total = 27)

```
        (ldaq     arglist_header)
        eppbp     callee
        eppbb     arglist
        staq      bb|0
        tspab     ap|call_op

|       spriab    sp|stack_frame.return_ptr
|       eppap     bb|0
|       callsp    bp|0

        ldaq      *-N
        tspbp     sb|stack_header.new_ent_op,*

|       ldx0      sp|stack_frame.size
|       sprisp    sp|stack_frame.prev_sp,0
|       eppsp     sp|0,0
|       staq      sp|stack_frame.size
|       epaq      bp|0
|       lprplp    sb|stack_header.lot_ptr,*au
|       sprilp    sp|linkage_ptr
|       spriap    sp|stack_frame.arg_ptr
|       spbpbp    sp|stack_frame.operator_return_ptr
|       eppap     operator_table
|       spriap    sp|stack_frame.operator_ptr
|       ldi       0,dl
|       tra       bp|M

        (random code)
        .
        .
        (end of code)
        tra       ap|returr_op

|       eppsp     sp|stack_frame.prev_sp,*
|       eppap     sp|stack_frame.operator_ptr,*
|       ldi       sp|stack_frame.return_ptr+1
|       rtcd      sp|stack_frame.return_ptr
```

CURRENT CALL SEQUENCE
(Total = 44)

```
        (fld      arglist_head,du)
        eppbp     callee
        eax1      arglist
        tsx0      apicall_op

|       tra        call_ext_out
|       epbpsb     sp10
|       ora        4,dl
|       staq       sb10,1
|       stx0       sp1stack_frame.return_ptr+1
|       eppap      sb10,1
|       eoplp      sp1linkage_ptr,*
|       callsp     bp10

        eax7      stack_size
        eppbp     sb1stack_header.pl1_operators_ptr,*
        tspbp     bp1entry_op

|       tra        ext_entry
|       eppbp     bp1-3
|       epaq       bp10
|       lorplp     sb1stack_header.lot_ptr,*au
|       eppbb      sb1stack_header.stack_end_ptr,*
|       sprisp     bb1stack_frame.prev_sp
|       spriap     bb1stack_frame.arg_ptr
|       eppab      bb10,7
|       spriab     bb1stack_frame.next_sp
|       spriab     sb1stack_header.stack_end_ptr
|       eppsp      bb10
|       sprilp     sp1linkage_ptr
|       spribp     sp1stack_frame.entry_ptr
|       spbpbp     sp1stack_frame.return_ptr
|       spbpbp     sp1text_base_ptr
|       stz        sp1stack_frame.operator_return_offset
|       eppap      operator_table
|       spriap     sp1stack_frame.operator_ptr
|       spriab     sp14
|       ldi        0,dl
|       tra        bp15

        (random code)
        .
        .
        (end of code)
        tra       apireturn_op

|       tra        return_mac
|       epbpsb     sp10
```

```
|          sprisp     sb|stack_header.stack_end_ptr
|          eppsp      sp|stack_frame.prev_sp,*
|          eppsb      sp|0
|          eopap      sp|stack_frame.operator_ptr,*
|          ldi        sp|stack_frame.return_ptr+1
|          rtcd       sp|stack_frame.return_ptr
```

INTRA-SEGMENT CALL SEQUENCE
(WITH OPERATORS)
(Total = 19)

```
        (epplp    spllinkage_ptr,*)
        (ldaq     arglist_header)
        eppbp     callee
        eopbb     arglist
        staq      bbl0
        ldaq      target_frame_size
        tspab     aplbound_call

|       tra       bound_call_entry
|       spriab    splstack_frame.return_ptr
|       ldx0      splstack_frame.size
|       sprisp    splstack_frame.prev_sp,0
|       eppsp     spl0,0
|       staq      splstack_frame.size
|       sprilp    spllinkage_ptr
|       spribb    splstack_frame.arg_ptr
|       spriap    splstack_frame.operator_ptr
|       spbpab    splstack_frame.operator_return_ptr
|       tra       bplX

        (random code)
        .
        .
        (end of code)
        tra       aplreturn_op_no_ind

|       eppsp     splstack_frame.prev_sp,*
|       rtcd      splstack_frame.return_ptr
```

INTRA-SEGMENT CALL SEQUENCE
(NO OPERATORS)
(Total = 15)

```
(epplp     spllinkage_ptr,*)
(ldaq      arglist_header)
eppbb      arglist
staq       bb|0
stcd       splstack_frame.return_ptr
tspbp      callee

ldaq       *-N
ldx0       splstack_frame.size
sprisp     splstack_frame.prev_sp
eopsp      sp|0,0
staq       splstack_frame.size
spbpbp     splstack_frame.operator_return_ptr
sprilp     spllinkage_ptr
spribb     splstack_frame.arg_ptr
spriap     splstack_frame.operator_ptr
(random code)
.
.
(end of code)
eppsp      splstack_frame.prev_sp,*
rtcd       splstack_frame.return_ptr
```

COMPATIBLE INTER-SEGMENT CALL SEQUENCE
(Total = 35)

```
          (fld      arglist_head,du)
          eppbp     callee
          eax1      arglist
          tsx0      aplcall_op

|         tra        call_ext_out
|         epbpsb     sp10
|         ora        4,dl
|         staq       sb10,1
|         stx0       sp1stack_frame.return_ptr+1
|         eppap      sb10,1
|         callsp     bp10

          eax7      stack_frame_size
          eppbp     sb1stack_header.pl1_operators_ptr,*
          tspbp     bp1entry_op

|         tra        ext_entry
|         ldx0       sp1stack_frame.size
|         stz        sp1stack_frame.size,0
|         stx7       sp1stack_frame.size,0
|         sprisp     sp1stack_frame.prev_sp,0
|         eppsp      sp10,0
|         epaq       bp-3
|         sxl7       sp1stack_frame.original_size
|         spriap     sp1stack_frame.arg_ptr
|         stq        sp1stack_frame.entry_offset
|         spbpbp     sp1stack_frame.operator_return_ptr
|         lprplp     sb1stack_header.lot_ptr,*au
|         sprilp     sp1linkage_ptr
|         spbpbp     sp1stack_frame.return_ptr
|         eopap      operator_table
|         spriap     sp1stack_frame.operator_ptr
|         ldi        0,dl
|         tra        bp15

          (random code)
          .
          .
          (end of code)
          tra        ap_return_op

|         tra        return_mac
|         eppsp      sp1prev_sp,*
|         eppap      sp1stack_frame.operator_ptr,*
|         ldi        sp1stack_frame.return_ptr+1
|         rtcd       sp1stack_frame.return_ptr
```

ALM ENTRY_SEQUENCES

NEW

```
        ldaq    *-N
        eppbp   sb|stack_header.pl1_operators_ptr,*
        tspbp   bp|alm_entry_op

|       ldx7    sp|stack_frame.size
|       staq    sp|stack_frame.size,7
|       epaq    bp|0
|       lprplb  sb|stack_header.isot_ptr,*au
|       sprplb  sp|stack_frame.static_ptr,7
|       lprplp  sp|stack_header.lot_ptr,*au
|       tra     bp|0
```

COMPATIBLE

```
        tspbp   sb|stack_header.entry_op_ptr,*

|       epaq    bp|-1
|       ldx7    sp|stack_frame.size
|       stq     sp|stack_frame.entry_offset,7
|       lprplb  sb|stack_header.isot_ptr,*au
|       sprplb  sp|stack_frame.static_ptr,7
|       lprplp  sb|sb|stack_header.lot_ptr,*au
|       tra     bp|0
```

ALM PUSH SEQUENCES

NEW

```
        eppbp     sb|stack_header.pl1_operators_ptr,*
        tspbp     bp|alm_push_op

|       ldx7      sp|stack_frame.size
|       sprisp    sp|prev_sp,7
|       eppsp     sp|0,7
|       spriap    sp|stack_frame.arg_ptr
|       sprilp    sp|stack_frame.lp_ptr
|       spbpbp    sp|stack_frame.operator_return_ptr
|       tra       bp|0
```

COMPATIBLE

```
        eax7      stack_frame_size
        tspbp     sb|stack_header.push_op_ptr,*

|       ldx7      sp|stack_frame.size
|       sprisp    sp|stack_frame.prev_sp,7
|       eppsp     sp|0,7
|       ldx7      bp|-2
|       stx7      sp|stack_frame.size
|       sxl7      sp|stack_frame.original_size
|       eax7      1
|       sxl7      sp|stack_frame.translator_id
|       spriap    sp|stack_frame.arg_ptr
|       sprilp    sp|stack_frame.lp_ptr
|       spbpbp    sp|stack_frame.operator_return_ptr
|       tra       bp|0
```

ALM RETURN SEQUENCES

NEW

```
        tra         sblstack_header.return_op_ptr,*

|       eppsp       splstack_frame.prev_sp,*
|       eppap       splstack_frame.operator_ptr,*
|       ldi         splstack_frame.return_ptr+1
|       rtcd        splstack_frame.return_ptr
```

CURRENT

```
        tra         sblstack_header.return_op_ptr,*

|       inhibit     on
|       sprisp      sblstack_header.stack_end_ptr
|       eppsp       splstack_frame.prev_sp,*
|       inhibit     off
|       epbpsb      spl0
|       eppap       splstack_frame.operator_ptr,*
|       ldi         splstack_frame.return_ptr+1
|       rtcd        splstack_frame.return_ptr
```

CURRENT ALM ENTRY/PUSH SEQUENCES

ENTRY

```
        tspbp     sb|stack_header.entry_op_ptr

|       eppbp     bp|-1
|       epplp     sb|stack_header.stack_end_ptr,*
|       spribp    lp|stack_frame.entry_ptr
|       epaq      bp|0
|       lprplb    sb|stack_header.isot_ptr,*au
|       sprplb    lp|stack_frame.static_ptr
|       lprplp    sb|stack_header.lot_ptr,*au
|       tra       bp|1
```

PUSH

```
        eax7      stack_frame_size
        tspbp     sb|stack_header.push_op_ptr

|       spribp    sb|stack_header.stack_end_ptr,*
|       eppbp     sb|stack_header.stack_end_ptr,*
|       sprisp    bp|stack_frame.prev_sp
|       spriap    bp|stack_frame.arg_ptr
|       sprilp    bp|stack_frame.lp_ptr
|       eppsp     bp|0
|       eppbp     sp|0,7
|       spribp    sb|stack_header.stack_end_ptr
|       spribp    sp|stack_frame.next_sp
|       eax7      1
|       stx7      sp|stack_frame.translator_id
|       tra       sp|0,*
```

NEW GATE PUSH SEQUENCE

```
            eppbp     sb|stack_header.pl1_operators_ptr,*
            tspbp     bp|gate_push_op

|           tra       gate_push
|           epaq      sp|0
|           era       sb|stack_header.stack_begin_ptr
|           ana       =-1,du
|           tze       same_ring
|           epplb     sb|stack_header.stack_begin_ptr,*
|           tra       gate_push_join
|same_ring:
|           ldx7      sp|stack_frame.size
|           epplb     sp|0,7
|gate_push_join:
|           sprisp    lb|stack_frame.prev_sp
|           mlr       (pr),(pr)
|           desc9a    bp|0,8
|           desc9a    lb|stack_frame.size,8
|           epaq      bb|-1
|           spriap    lb|stack_frame.arg_ptr
|           spbpbp    lb|stack_frame.operator_return_ptr
|           stbq      lb|stack_frame.entry_offset,60
|           lprplp    sb|stack_header.lot_ptr,*au
|           sprilp    lb|stack_frame.lp_ptr
|           eppsp     lb|0
|           tra       bp|1
```

MTB-434

```
/*         BEGIN INCLUDE FILE ... stack_frame.incl.pl1 ... */

/* Modified: 16 Dec 1977, D. Levin - to add fio_ps_ptr and pl1_ps_ptr */
/* Modified:  3 Feb 1978, P. Krupp - to add run_unit_manager bit & main_proc bit */
/* Modified: 21 March 1978, D. Levin - change fio_ps_ptr to support_ptr */
/* Modified: 21 December 1979, M. Weaver - change stack frame format */

     dcl     sp                        pointer;                /* pointer to beginning of stack frame */

     dcl     stack_frame_min_length    fixed bin static init (48);


     dcl     1 stack_frame             based (sp) aligned,
               2 pointer_registers     (0:7) ptr,
               2 prev_sp               pointer,
               2 operator_return_ptr   pointer,
               2 return_ptr            pointer,
               2 size                  fixed bin (18) unaligned unsigned,
               2 original_size         fixed bin (18) unaligned unsigned,
               2 entry_offset          bit (18) unaligned,
               2 translator_id         fixed bin (18) unaligned unsigned, /* Translator ID
                                                                             0 => PL/I version II
                                                                             1 => ALM
                                                                             2 => PL/I version I
                                                                             3 => signal caller frame
                                                                             4 => signaller frame */
               2 operator_and_lb_ptr   ptr,                    /* serves as both */
               2 arg_ptr               pointer,
               2 static_ptr            ptr unaligned,
               2 support_ptr           ptr unal,               /* only used by fortran I/O */
               2 on_unit_relp1         bit (18) unaligned,
               2 pad1                  bit (18) unaligned,
               2 pad2                  bit (36) unaligned,
               2 x                     (0:7) bit (18) unaligned, /* index registers */
               2 a                     bit (36),               /* accumulator */
               2 q                     bit (36),               /* q-register */
               2 e                     bit (36),               /* exponent */
               2 timer                 bit (27) unaligned,     /* timer */
               2 pad3                  bit (6) unaligned,
               2 ring_alarm_reg        bit (3) unaligned;
```

```
dcl     1 stack_frame_flags     based (sp) aligned,
          2 pad                 (0:7) bit (72),         /* skip over prs */
          2 xx0                 bit (22) unal,
          2 main_proc           bit (1) unal,           /* on if frame belongs to a main procedure */
          2 run_unit_manager    bit (1) unal,           /* on if frame belongs to run unit manager */
          2 signal              bit (1) unal,           /* on if frame belongs to logical signal_ */
          2 crawl_out           bit (1) unal,           /* on if this is a signal caller frame */
          2 signaller           bit (1) unal,           /* on if next frame is signaller's */
          2 link_trap           bit (1) unal,           /* on if this frame was made by the linker */
          2 support             bit (1) unal,           /* on if frame belongs to a support proc */
          2 condition           bit (1) unal,           /* on if condition established in this frame */
          2 xx0a                bit (6) unal,
          2 xx1                 fixed bin,
          2 xx2                 fixed bin,
          2 xx3                 bit (25) unal,
          2 old_crawl_out       bit (1) unal,           /* on if this is a signal caller frame */
          2 old_signaller       bit (1) unal,           /* on if next frame is signaller's */
          2 xx3a                bit (9) unaligned,
          2 xx4                 (9) bit (72) aligned,
          2 v2_pl1_op_ret_base  ptr,                    /* When a V2 PL/I program calls an operator the
                                                         *  operator puts a pointer to the base of
                                                         *  the calling procedure here. (text base ptr)  */
          2 xx5                 bit (72) aligned,
          2 pl1_ps_ptr          ptr;                    /* ptr to ps for this frame; also used by fio. */

     END INCLUDE FILE ... stack_frame.incl.pl1 */
```

```
/*        BEGIN INCLUDE FILE ... stack_header.incl.pl1 .. 3/72 Bill Silver  */
/*        modified 7/75 by M. Weaver for *system links and more system use of areas */
/*        modified 2/77 by M. Weaver to add rnt_ptr */
/*        modified 1/80 by M. Weaver to define new_frame_format bit */
/*        modified 1/80 by M. Weaver to remove stack_end_ptr and add new entry op ptrs */

dcl   sb     ptr;                                          /* the  main pointer to the stack header */

dcl 1 stack_header       based (sb) aligned,

    2 pad1 (4)           fixed bin,                 /*  (0)  also used as arg list by outward_call_handler  */
    2 old_lot_ptr        ptr,                       /*  (4)  pointer to the lot for current ring (obsolete) */
    2 combined_stat_ptr  ptr,                       /*  (6)  pointer to area containing separate static */

    2 clr_ptr            ptr,                       /*  (8,  10)  pointer to area containing linkage sections
    2 max_lot_size       fixed bin(17) unal,        /*  (10, 12)  DU  number of words allowed in lot */
    2 new_frame_format   bit (1) unal,              /*  (10, 12)  DL  "1"b if stack frame format has changed
    2 main_proc_invoked  fixed bin (10) unal,       /*  (10, 12)  DL  nonzero if main procedure invoked in ru
    2 run_unit_depth     fixed bin(5) unal,         /*  (10, 12)  DL  number of active run units stacked */
    2 cur_lot_size       fixed bin(17) unal,        /*  (11, 13)  DU number of words (entries) in lot */

    2 system_free_ptr    ptr,                       /*  (12, 14)  pointer to system storage area */
    2 user_free_ptr      ptr,                       /*  (14, 16)  pointer to user storage area */

    2 null_ptr           ptr,                       /*  (16, 20)  */
    2 stack_begin_ptr    ptr,                       /*  (18, 22)  pointer to first stack frame on the stack
    2 null_ptr_pad       ptr,                       /*  (20, 24)  */
    2 lot_ptr            ptr,                       /*  (22, 26)  pointer to the lot for the current ring */

    2 signal_ptr         ptr,                       /*  (24, 30)  pointer to signal procedure for current rin
    2 bar_mode_sp        ptr,                       /*  (26, 32)  value of sp before entering bar mode */
    2 pl1_operators_ptr  ptr,                       /*  (28, 34)  pointer to pl1_operators_$operator_table */
    2 call_op_ptr        ptr,                       /*  (30, 36)  pointer to standard call operator */

    2 push_op_ptr        ptr,                       /*  (32, 40)  pointer to standard push operator */
    2 return_op_ptr      ptr,                       /*  (34, 42)  pointer to standard return operator */
    2 return_no_pop_op_ptr ptr,                     /*  (36, 44)  pointer to standard return / no pop operato
    2 entry_op_ptr       ptr,                       /*  (38, 46)  pointer to standard entry operator */

    2 trans_op_tv_ptr    ptr,                       /*  (40, 50)  pointer to translator operator ptrs */
    2 isot_ptr           ptr,                       /*  (42, 52)  pointer to ISOT */
    2 sct_ptr            ptr,                       /*  (44, 54)  pointer to System Condition Table */
    2 unwinder_ptr       ptr,                       /*  (46, 56)  pointer to unwinder for current ring */
```

```
    2 sys_link_info_ptr ptr,                /*  (48, 60)  pointer to *system link name table */
    2 rnt_ptr           ptr,                /*  (50, 62)  pointer to Reference Name Table */
    2 evt_ptr           ptr,                /*  (52, 64)  pointer to event channel table */
    2 assign_linkage_ptr ptr,               /*  (54, 66)  pointer to storage for (obsolete) hes_$assig

    2 ext_entry_op_ptr  ptr,                /*  (56, 70)  pointer to PL/I operator ext_entry */
    2 ext_entry_desc_op_ptr ptr,            /*  (58, 72)  pointer to PL/I operator ext_entry_desc */
    2 ss_ext_entry_op_ptr ptr,              /*  (60, 74)  pointer to PL/I operator ss_ext_entry */
    2 ss_ext_entry_desc_op_ptr ptr,         /*  (62, 76)  pointer to PL/I operator ss_ext_entry_desc *

    2 int_entry_op_ptr  ptr,                /*  (64, 100)  pointer to PL/I operator int_entry */
    2 int_entry_desc_op_ptr ptr,            /*  (66, 102)  pointer to PL/I operator int_entry_desc */
    2 ss_int_entry_op_ptr ptr,              /*  (68, 104)  pointer to PL/I operator ss_int_entry */
    2 ss_int_entry_desc_op_ptr ptr,         /*  (70, 106)  pointer to PL/I operator ss_int_entry_desc

    2 pad2 (26)         bit (36) aligned;   /*  (72, 110)  for future expansion, including new entry o

/*        The following offset refers to a table within the  pl1  operator table.  */

  dcl     tv_offset               fixed bin         init(361)  internal static;  /* (551) octal */

/*        The following constants are offsets within this transfer vector table.  */

  dcl    (call_offset             fixed bin         init(271),
          push_offset             fixed bin         init(272),
          return_offset           fixed bin         init(273),
          return_no_pop_offset    fixed bin         init(274),
          entry_offset            fixed bin         init(275))   internal static;
```