TO: MSPM Distribution

FROM: G. S. Stoller

SUBJECT: BE.8.05

DATE: 7/24/67

This section supercedes the description of pseudo-process initialization given in BE.7.07 (issued 12/9/66) whenever the two conflict. In particular, parts 8 and 9 of the "Special Segments Descriptions" subsections are superceded by this document.

Segment <.init> is no longer automatically assembled with each run. Users may now request it to be assembled, supply their own text + link for it, or use the <.init> segment on the segment library.

Merge-editor users may fetch text and link for <.init> by inserting the line

```
FETCH      .INIT      TEXT + LINK
```

in the GECOS file.

64.5 driver users must insert a copy of the code of <.init>, as an ordinary EPLBSA assembly, in their data deck. It is not possible to get the text and link of the driver-supplied version of <.init>.

Published: 7/24/67

Identification

Pseudo-Process Initialization

G.S.Stoller

0. Purpose

Quite a bit of initialization of the pseudo-process is done before the user's code is entered. Some of this initialization is done in the 645 LOADER (which is in the 635 support package) and part of it is done in pseudo-process itself.

This document attempts a brief outline of the initialization. It is mainly for anyone who has to make modifications to these segments; a standard user need not concern himself with this document beyond the Summary, except for references made there to other parts of this document.

1. Summary

If the user's code returns all the way back to the initialization module, it will return to <init_disp> which will call <escape> to provide a normal termination to the pseudo-process. This call to <escape> will not destroy or modify any stack frames above <init_disp>'s stack frame. A former standard procedure of terminating with a divide-check to avoid destroying stack frames need not be used.

The bottom stack frame will show a call from <init_disp> to <.init> and the next stack frame will show a call, but not from <.init>. While this is not a common occurrence, it happens here because of the unusual coding of <.init> which performs neither a "save" nor a "return".

2. 645 LOADER

The forward pointer in the bottom frame of the stack is set here (to 40 (8)). This is the stack frame used by <init> and <init_disp>.

"MME1 -1" instructions are stored in the fault vector, <fvector>. for the fault tag 2 fault, an "scu,tra" pair to f2catc is set up.

<ivectr>, the interrupt vector, is initialized to "scu,rcu" pairs. (These result in a "nop" from the supplement on interrupts directed to the pseudo-process.)

3. <init>

Control is passed from the 645 LOADER to segment <init> at location 0, and the pseudo-process begins running. <init>'s attributes are MASPRC, SLVACC. (SLVACC is needed to allow the 645 LOADER to transfer to <init>.)

<init> first sets the control fields of the address base registers to the conventional settings required in the pre-Multics system.

Address base register sb is locked (to the stack segment number). The previous stack frame pointer is set to the null pointer, ITS (-1,1,N).

Second, the sp-pair is set to point to the base of the stack segment; this is the base of the current stack frame. Also, the lp-pair is set to point to <init>'s linkage segment.

Third, <init> prelinks those intersegment references of the linker module (See BE.8.00, pages 3 and 4 of the 10/13/66 issue) that could cause infinite recursive looping on linkage faults in the linker module.

Finally, <init> transfers to <init_disp>. Starting from this point, linkage faults can occur as the linker module can handle them.

The following constraints are imposed by <init>; if not obeyed, and <init> finds this out, it will abort the pseudo-process run.

1) <namtab> must be segment number 3.

2) <stack>, <f2catc>, <linker>,<segman> must be found in <namtab>.

3) All intersegment reference names that are to be prelinked must be of the form "<α>|[α]" where "α" is a symbol of at most 7 characters. Furthermore, the name must begin on an even word boundary.

4) Each attempt to prelink must succeed.

In addition, there are constraints imposed on the coding of <init>:

1) <init> may be entered only once in a pseudo-process. If entered a second time it will abort; hence <init> may not call a procedure that returns to it.

2) No temporary storage (in the stack) has been assigned to <init>.

3) Until the linker module is ready to accept an occurrence of a fault tag 2, <init> may not make any intersegment references (that would ordinarily result in a linkage fault) unless it prelinks the reference.

Currently, <init> transfers (there is no need to call since no return will be made) to <init_disp> right after the linker module is sufficiently prelinked.

<init> is coded under constraints unusual to procedures in the pseudo-process. Hence it is left as soon as possible. Initialization is continued in <init_disp> which is not as constrained as <init>.

4. <init_disp>

This segment is logically just an extension of <init>. It appears as a separate segment for the following reasons:

1) To demarcate the line between code in which a fault tag 2 may not occur and code where it may occur.

2) To allow calls to external procedures. (<init_disp> can be returned to.)

Since <init_disp> is an extension of <init> and can run in the same stack frame (which it does), no save is done when <init_disp> is entered. (Note that <init> does not call <init_disp> either. <init_disp> has taken over <init>'s stack frame, so <init> is not visible in a stack trace.) <init_disp> continues the initialization by dispatching to other procedures before finally calling the user's entry point through <.init>.

<init_disp> first calls <init_esc> to initialize for escaping from the pseudo-process environment to the GECOS environment. (See BE.7.10.) Then it calls <escape> with the null escape number (See BE.7.10 again) so that the links from <init_disp> to <escape> and the links in <escape> are made. Also, <memory> is protected (class is directed fault, see BE.7.00) when <escape> returns.

Now <.init> is called at entry point [1.init], and the user's code is entered.

If <init_disp> is returned to after the above, it again calls <escape>|[escape], but this time with the "finish" escape number. Since the link from <init_disp> to <escape> and all the links needed by <escape> are

already made, no linkage faults will occur at this time. Also, <escape> uses no stack storage. (See BE.7.10.) So, except for the call to <escape> resulting in the address base registers, registers (as in SREG), and control-double information being stored in <init_disp>'s stack frame, the stack is unchanged.

5. <init_esc>

This is described in BE.7.10. It performs the initialization required for the escape mechanism.

6. <.init>

This segment is reduced to one line of actual machine code. In the old terminology, it is a "transfer vector" (of a trivial nature since only one transfer point is provided).

Here is the code of this segment:

```
        name    .init
        entry   ..init
        segref  α,β
..init: tra     β
        end
```

where <α>|[β] is the entry point to the user's code (as specified by the entry line in the GECOS file used by the merge-editor or by the entry control card in the data deck of a 64.5 driver run; See BE.5.02 and BE.6.01). Users may now supply their own text and link of this segment, and not reassemble it each time a pseudo-process run is made. Furthermore, such a segment going to <main>|[start] is on the segment library and will be used if no <.init> is supplied.

A stack trace comes up with a "funny" at the <.init> point when <.init> is coded as bove. The bottom stack frame shows a call from <init_disp>

to <.init>, whereas the next stack frame shows a call from some other segment (not <.init>).

<.init> does not show up here because it does not "call", it transfers. (The same thing happened to <init>).