TO: MSPM Distribution
FROM: D. Widrig
SUBJ: BL.1.03, BL.6.03, and BE.15.02
DATE: 10/02/67

Section BL.1.03 is a major revision of BE.15.02 and completel obsoletes it.

Section BL.6.03 describes the module which initializes the Multics Core Control modules and is a complete replacement of the interim program described in BL.6.03 dated 4/10/67.

Published: 10/02/67
(Supersedes: BL.6.03, 04/10/67)

Identification

Core Control Initializer
T. P. Skinner, M. A. Padlipsky

Purpose

The Core Control Initializer (CCI) is a procedure which must be called by the Multics Initializer before the Basic File System's Core Control Module (CCM) may be utilized. Essentially, the role of the CCM is to tabulate used and unused areas of core memory so that space may be found for new pages when they are brought in from secondary storage; see BG.6. Up to the point in the initialization process at which the CCI is called, contiguous locations in core have been assigned as needed by the early initialization routines, beginning from the Bootload base. After the CCI has operated, the CCM will have the necessary information as to current core usage that will enable it to operate as if Multics were initialized and running.

Discussion

The CCM must be furnished with two basic kinds of information. First, and more straightforward, is information about large blocks of memory which are physically available to the particular Multics system being initialized. Such information is transmitted by the CCI through a call to the CCM entry <u>initcm</u>, which is one of the two basic entries to the CCM for initialization purposes. (From the CCM's point of view, the call to <u>initcm</u> allows the "Core Map"--the primary data base of Core Control--to be initialized.) The CCI derives its information about currently-available physical core from the initialization mechanism's Major Module Configuration Table (see BL.5). If, for example, the middle 64K of memory should be down for some reason, this fact would be reflected in the Major Module Configuration Table and the CCI would not make the affected locations available to Core Control in the call to <u>initcm</u>.

The second kind of information with which the CCM must be furnished is that relating to the current usage of core. Existing segments and page tables must be accounted for, both as to core location occupied and as to "status". Status information, including such items as whether or not the segment is wired down and whether or not the segment belongs to the initialization mechanism, is gleaned from the SLT (see BL.6.02) and from the current descriptor segment. It is passed to the CCM by means of the latter's second basic entry for initialization purposes, assign_at_loc. The CCI must call assign_at_loc to account for each 64-word block of core, whether it is part of a page table, hyperpage, or entire unpaged segment. (Each call of this type creates what Core Control defines as a "group". Each group will remain as one unit until and unless it is freed by a call to the remove entry in Core Control.)

Implementation

The Core Control Initializer comprises a single procedure, init_core_control. Figure 1 presents a block diagram of init_core_control. The major portion of the CCI's work is performed by a repeatedly called internal procedure named core. Figure 2 presents a block diagram of core.

The logic of init_core_control is as follows:

1. Obtain pointers. Pointers to the SLT, the current descriptor segment, and "abs_seg" are necessary. They are gotten through use of the following call:

   ```
   Call slt_manager$get_seg_ptr(name, ptr);
   ```

   (Abs_seg is actually a fictitious segment, being in reality an entry in the descriptor segment indicating an unpaged segment, which is used to point to page tables when the latter must be manipulated; see BL.12. It is not assigned core space.)

2. Call initcm. Available-core information is extracted from the Major Module Configuration Table (see BL.5) and passed to Core Control via a call to initcm (see BG.6.00).

3. Call core for supervisor segments. The segment numbers of the first and last supervisor segments already read in are extracted from the SLT and used as the limits of a loop containing a call to core (see below).

4. Call core for initialization segments. The segment number of the first and last initialization segments already read in are extracted from the SLT and used as the limits of a loop containing a call to core (see below).

5. Return.

The single argument of the core subroutine is a segment number; call it n. The logic of core is as follows:

1. Examine SDW. If n's segment descriptor word is marked directed fault, core returns, thus ignoring the nth descriptor segment entry. If the descriptor is illegal (>4), call panic (see BL.5.00).

2. Get SLT information. The status information for n (wired/unwired), hyperpage size, and descriptor segment switch are extracted from the SLT, to be furnished as arguments in the following call to Core Control.

3. Call assign_at_loc. The information gathered in step 2 is passed to Core Control via calls to assign_at_loc (see BG.6.00) for each hyperpage of n, and for the remaining fractional hyperpage, if any. (The point at issue here is that assign_at_loc accepts assignments for an arbitrary number of contiguous blocks at a single time, so that each call but the last can be for a hyperpage's worth of blocks.) Note that unpaged segments are disposed of in a single call to assign_at_loc. Also, each page table word processed is checked for legality in the same fashion as SDW's are (see step 1).

Figure 1.

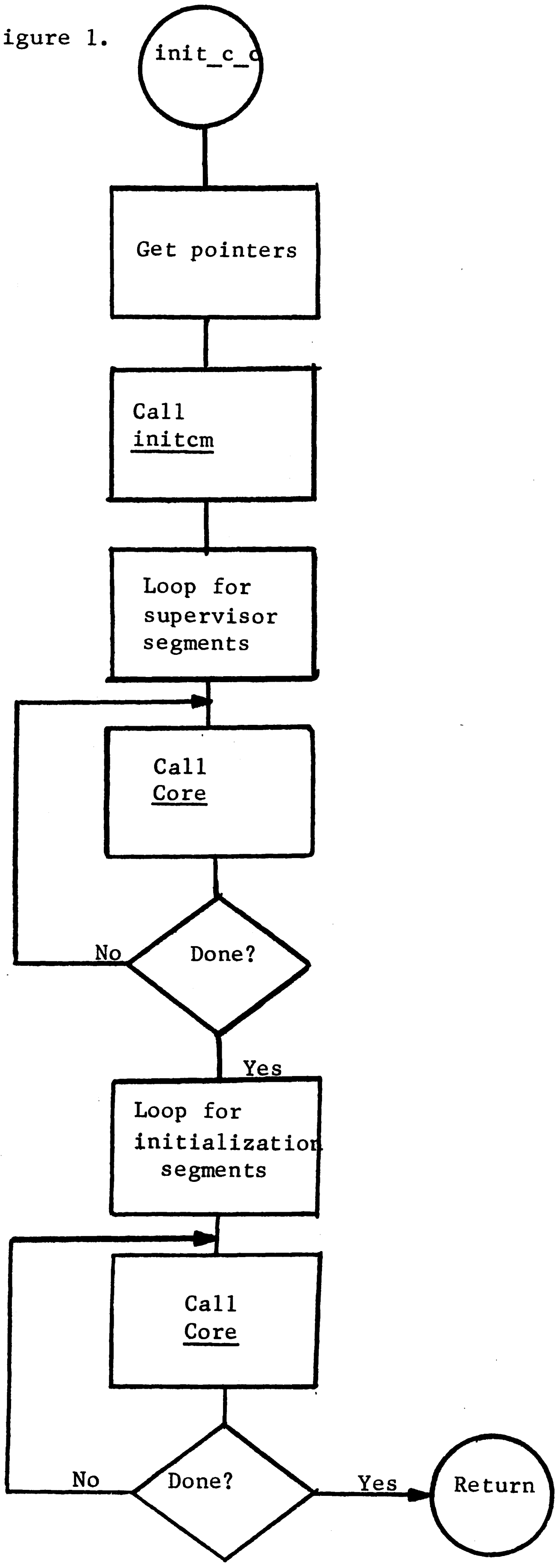
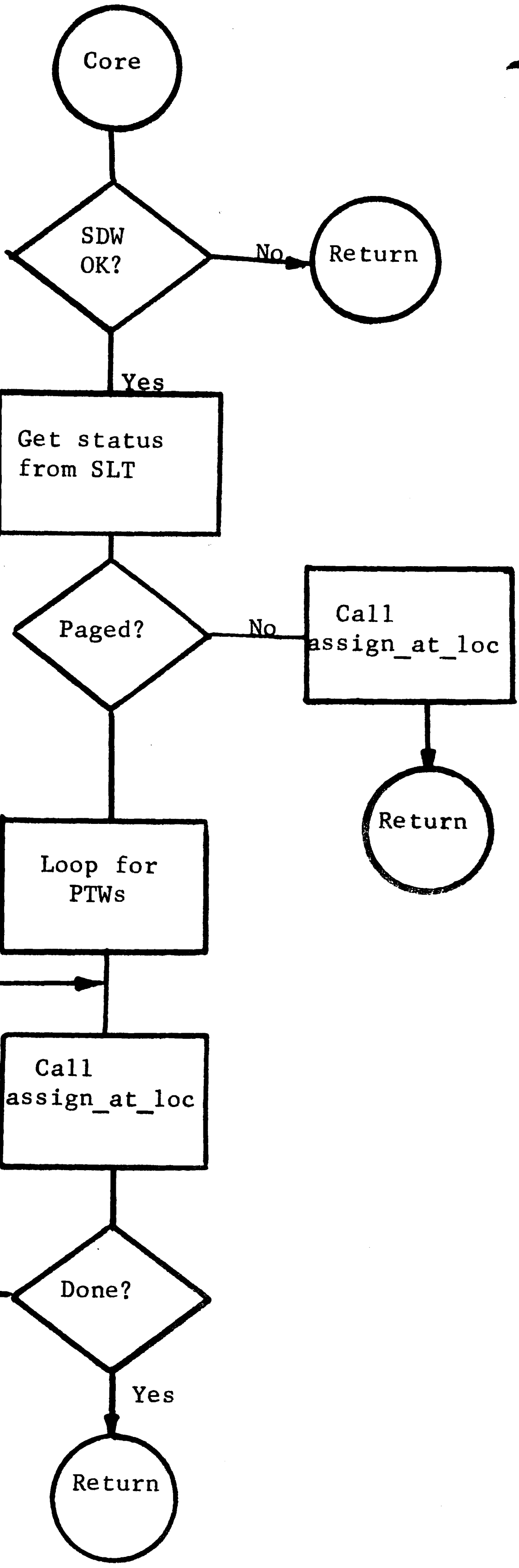


Figure 2.


Core
SDW OK?
No
Return
Yes
Get status from SLT
Paged?
No
Call assign_at_loc
Return
Loop for PTWs
Call assign_at_loc
No
Done?
Yes
Return