HP 3000 IUG EDINBURGH

2nd-7th OCTOBER 1983

HP 3000 INTERNATIONAL USERS GROUP

1983 HP 3000 INTERNATIONAL CONFERENCE
EDINBURGH OCTOBER 2–7
THE ASSEMBLY ROOMS AND
MUSIC HALL COMPLEX, GEORGE STREET
EDINBURGH, SCOTLAND

PROCEEDINGS

Architectural Changes for MPE V

David N. Holinstat
Development Engineer
Hewlett-Packard GmbH
Böblingen, Baden-Württemberg
Federal Republic of Germany

ABSTRACT

MPE V supports 255 code segments per program, as well as allowing most MPE Tables to reside anywhere in the first 4Mb of memory. These expansions necessitated several modifications to the HP3000 architecture. Relocation of MPE's Tables required changes to the LST and SST instructions. Removal of the code segment limitations and expansion of the Code Segment Table involved adding a new table, the Logical Segment Transfer Table (LSTT), to the HP3000 architecture. In addition, alterations were made to the PCAL, SCAL, EXIT, IXIT, PARC, ENDP, and XBR instructions, the Segment Transfer Table (STT), and one bit in the Stack Marker.

The presentation will quickly review the original HP3000 architecture, then explain the alterations for MPE V.

ARCHITECTURAL CHANGES FOR MPE V

David N. Holinstat

Since its first introduction, the HP3000 architecture has changed several times. For example, the Series I and its predecessors allowed a total of 255 code segments for the entire machine, including MPE and all users. When the Series II was introduced, the architecture was modified to provide 192 code segments for MPE, plus up to 63 code segments for every program. Series II also provided 64-bit Extended Precision Floating Point, an improvement over the 48-bit precision found on Series I. Introduction of the Series 33 brought an entirely new I/O system to support the HP-IB peripherals.

The most recent change, part of the recently announced Series 42, 48, and 68 systems, will remove many of the current "maximums" caused by the HP3000 architecture itself. These architectural changes, along with the new MPE V operating system, will allow each program to have 255 code segments (private or shared), as well as allowing many critical MPE tables to double or quadruple in size. This will allow many more jobs, sessions, terminals, programs, etc., to be supported as were on MPE IV.

General Table Expansion

"MPE has a table for everything." The MPE Tables manual currently has 20 chapters and is several hundred pages in length. All the data required to control user processes, check security, handle memory and I/O resources, etc., is kept by MPE in its tables. So, as the number of terminals, sessions and jobs supported has grown, HP has needed to expand MPE's tables. However, for reasons which will become apparent, many of MPE's tables must be kept in Bank 0 of memory. And recently, as the number of terminals and sessions allowed quickly grew, it became apparent that Bank 0 is not an unlimited resource!

Historically, MPE has kept its data in an area known as the System Global Area (SYSGLOB), beginning at location %1000 in Bank 0 of memory. When MPE wanted to access this data, it would set DB to 0.%1000. (We write "0.%1000" for Bank 0, loc. %1000.) It could then access tables (which are just arrays that happen to contain MPE's data) using the normal LOAD and STOR instructions (Fig. 1a). These instructions can directly address DB+0 through DB+255, so MPE keeps pointers to the most important tables and a few very commonly used variables in this area (from %1000 to %1377). When DB is set to SYSGLOB, many tables can be accessed with normal LOAD and STOR Indirect instructions. For example, suppose location %1005 points to the I/O Request Queue (IOQ). We can access the 20th word in the IOQ with the following code sequence:

```
       Instruction            Function
   (a) LDXI  20               X := 20;
       LOAD  DB+5,I,X         TOS := ((DB+5) + DB + X)
                                  = (%4000 + %1000 + %24)
                                  = (%5024);
```

[Please see Fig. 2 for the sample SYSGLOB layout to be used in these examples. Note that "(%nnnn)" means the contents of loc. %nnnn. Also, since all Store-type instructions function exactly like the Load-type instructions as far as address calculation goes, all examples use the Load-type instructions.]

It became apparent that it was very uneconomical to always switch the DB register of SYSGLOB when accessing system tables, because the actual act of switching can take significant CPU time. Thus, the Load System Table (LST) and Store System Table (SST) instructions were designed. These instructions reference tables in SYSGLOB without actually switching the DB register (Fig. 1b). So, code sequence (a) can be replaced by sequence (b), without regard to the current content of the DB register:

```
    (b) LDXI  20             X := 20;
        LST   5              TOS := ((SYSDB+5) + SYSDB + X)
                                  = (%4000 + %1000 + %24)
                                  = (%5024);
```

The addressing range of LST is 0-15, because only 4 bits could be spared in the opcode. (Such are the problems with adding instructions after the original design!) So, in order to make the instruction more useful, LST 0 was given a special meaning. When an LST 0 is executed, the offset to the array pointer is taken from TOS, thus giving an even greater range than the other Load instructions. For example, reading the 30th word of the Example Table (pointer at DB+134) could be done as follows:

```
    (c) LDXI  30             X := 30;
        LDI   134            TOS := 134;
        LST   0              TOS := ((SYSDB+(TOS)) + SYSDB + X)
                                  = ((%206) + %1000 + %36)
                                  = (%101400 + %1000 + %36)
                                  = (%102436)
```

Examples (b) and (c) certainly require as many or more instructions as does (a). However, they don't require DB to be set, an operation taking dozens (possibly hundreds) of instructions.

There is one fault, however, with both of the above approaches to system table access. Whether we use the normal LOAD instruction with DB-relative addressing (implying DB set to 0.%1000), or the LST instruction, which addresses relative to SYSGLOB (0.%1000) implicitly, we have one critical limitation: we are always addressing somewhere in Bank 0. This means that any tables addressed using one of the preceding methods must be in Bank 0. When you recall that a bank of memory on the HP3000 contains exactly 65,536 words, you realize that as MPE grows larger, we will have a problem finding space for all of the tables. And the need for space becomes particularly acute when adding terminals and sessions, since each terminal requires one entry in each of several I/O-related tables, and each session requires entries in many job-related tables.

One possible solution would be to use absolute addressing for all tables. There are problems with this, however: First of all, our degrees in Computer Science would all be retroactively revoked! More seriously, an absolute address takes 2 words, one for Bank and one for Offset. This means that the 255 locations in SYSGLOB that are directly addressable could hold only 127 pointers maximum, instead of the 255 possible today. Also, some tables contain cross-references to several other tables. Today these cross-reference pointers are usually SYSGLOB-relative. Thus, a table entry containing pointers to 5 other tables would have to be 5 words longer to hold absolute address pointers to the 5 other tables. If the table has 100 entries, we have just used 500 more words! So, absolute addressing was ruled out to the extent possible.

Instead, it was decided to change the format of system table pointers, and change the LST and SST instructions as well. Instead of a 16-bit SYSGLOB-relative pointer, LST and SST will now expect a pointer with the following format:

```
| Base.(0:11)  | Bank   |
|______________|_______|
0            10 11    15
```

Bits (11:5) specify the bank in which the table will be found. The base address is calculated by setting the bank bits (11:5) to zero and adding %1000. Note that the base address is calculated as it currently is, except that the low order 5 bits are set to zero. The new format has several ramifications:

1) The use of 5 bank bits allows MPE tables to be loaded anywhere in the first 32 banks (4 Mb) of memory.

2) MPE tables are required to begin on 32-word boundaries, since the last 5 bits of a table address will always be zero.

3) Compatibility with Series II, III, 30, 33, 40, and 44 is maintained. Separate versions of MPE will not be required.

Point 3 needs some clarification. Beginning with MPE V, INITIAL will always create those tables referenced via LST/SST on 32-word boundaries. However, it will also determine whether it is running on a machine with MPE V level firmware or not. If not, INITIAL will simply locate all such tables in Bank 0. The old LST instruction then sees a 16-bit pointer whose low order bits happen to be 0; this points (correctly) to the beginning of the table. If one were now to install the MPE V firmware and do a COOLSTART, the new firmware will see the same table addresses, this time pointing specifically to Bank 0. So, MPE V will run on both old and new firmware without problem. Of course, if MPE V is configured with larger table sizes than can possibly fit in Bank 0, and is so installed on a machine with the old firmware, you will get the old, friendly "OUT OF MEMORY" message from INITIAL!

<u>Code Segment Table Expansion</u>

Recently some HP3000 users have attempted to run several large applications systems on the HP3000 simultaneously -- and have been rewarded with the message "OUT OF CST ENTRIES--UNABLE TO LOAD PROGRAM TO BE RUN". Upon examination of the configuration, they find that the Code Segment Table is configured to its maximum 192 entries. These users have often asked, "Why doesn't HP just change the maximum size of this silly table?" Unfortunately, expanding the CST is not just a matter of changing the size of an array in 2 or 3 MPE modules and recompiling! It has taken a major revision of MPE and of the 3000 architecture to allow each program 255 code segments. The following section will explain the Code Segment Table (CST) structure, both new and old, as well as the difficulties involved in changing it.

Let's quickly review the concept of a code segment. All memory on the HP3000 is divided into code segments, data segments, and free areas. All program code is kept in code segments, and all data in data segments. But these segments can be located anywhere in memory. So, to begin running a program in code segment 2, we first have to find code segment 2. To do that, we look in the Code Segment Table (CST), where there is a four-word entry for code segment 2 (Fig. 3). Among other things, this entry tells whether the segment is in memory ("Present") or on disc ("Absent"). If present, the CST entry tells where the code segment is located and how long it is. To begin executing code within this segment (after a PCAL or EXIT), the 3000 sets PBank := Bank, PB := Base, P := Base + desired offset into segment, and PL := Base + (Length/4)*4. The format of the Code Segment Table entry hasn't changed much since the original 3000. What has changed is the way in which we find the appropriate entry. In other words, the layout of the CST has changed, but the content remains the same. Changing the layout once more with the release of the new Series 42-48-68 machines has enabled us to remove the 192 entry limitation, as described below.

The first point to discuss is a very important limitation: A code segment is identified by its number, which is between 1 and 255. This limitation comes from the fact that only 8 bits are allowed for the segment number (Seg#) in the Status register. Changing this limitation directly would cause incredible problems! For example, suppose we wanted to quadruple the addressing range to 1024, which would require 10 bits for Seg# instead of 8. First, to hold the extra bits, we would have to make the Status register an 18-bit register. (Those of you with microprocessor experience know how easy it is to find a register chip with 18 bits!) Furthermore, the extra 2 bits would have to go in the stack marker. We can get one bit reasonably easily, but not two; therefore, we would have to go to a 5-word stack marker. Changing to a 5-word stack marker would mean that all of MPE and all user programs in the world (!) would have to be recompiled. Except, of course, SPL programs, which would have to be rewritten! Given that HP likes to brag about compatibility, this "brute force" method of adding more code segments can be quickly rejected.

The Code Segment Table architecture has been modified once before. The Series I and its predecessors had the simplest CST structure. At any given time there were 255 code segments available on the machine, numbered 1 to 255 (Fig. 4). The total number of code segments assigned to MPE and all running programs could never exceed 255. Of course, since the 3000 was introduced with a maximum memory size of 64K words, this was quite reasonable. As the system grew and more users were added, resources became more precious. One result was the first CST expansion, which was introduced with the Series II.

With the advent of the Series II, the Code Segment Table was divided into 2 domains: the system Code Segment Table (CST) with entries for segments 1-191, followed by the Code Segment Table Extension (CSTX) (Fig. 5). In the CSTX are blocks of entries, one block per program. Each block contains entries for segments 193-255 (%301-%377). MPE segments are assigned entries in the system CST, as are user segments that come from a Segmented Library (SL). Program segments have their entries in the CSTX. So, if a program has 6 code segments, there will be a block of entries in the CSTX for segments %301-%306. A program having 63 segments will have a CSTX entry block containing entries for segments %301-%377. A pointer in absolute memory loc. 1 points to the block of CSTX entries in use at any given time; this pointer is updated by the MPE Dispatcher whenever it launches a process. This architecture is the basis for the current CST limitations: 191 system and SL code segments, and 63 code segments per program. One can easily imagine 20 programs running, each with 63 code segments; this is a vast improvement from the 255 code segment maximum on Series I.

However, there is still a problem with the above method. As noted, there is a block of CSTX entries for each program. But only code segments that are part of a program can have entries in the CSTX, because these segments are only accessible when that program is running. Sharable segments--i.e., segments coming from an SL--are not associated with any particular program. Therefore, to allow them to be accessed from any program, they must be given entries in the system CST. Since MPE can use up to 100 of the 191 available entries, there is clearly a rather strict limitation on the number of SL segments a program can use. Unfortunately, large application systems, such as HP's own MM/3000 and HPFA, use many sharable segments in the interest of efficiency: If a code segment is included in 5 different programs (for example, using an RL), then when all 5 programs are running, 5 copies of the segment must reside in memory; but if the segment is put into an SL, it is made sharable, and one copy can be used for all 5 programs. This takes, however, another CST entry. So, for MPE V, a scheme was developed to allow more entries in the CST by introducing the concept of "code segment mapping".

Under the new system, a code segment can be "logically mapped" or "physically mapped". Physically mapped code segments will be located as they are today--with entries from 0 through 255 beginning at the CST Base address stored in absolute loc. 0. There can only be 255 physically mapped code segments, and these will all be reserved for MPE. All user segments will be logically mapped, including subsystems such as EDITOR and COBOL. Program segments will have entries in the CSTX, similar to the current (Series II -> 64) systems. Sharable user segments (i.e., SL segments) will have entries in the CST with physical segment numbers of 256 and greater. A CPU-internal flag will control the current mapping state of the CPU: physical or logical. This will not be held in the Status register, because (as mentioned previously) there is no space available there.

When the CPU transfers control to a logically mapped code segment, it looks first at a new table, the Logical Segment Transform Table (LSTT) (Fig. 6). Each program has its own LSTT, which is pointed to by absolute loc. %1221-%1222 whenver that program is running. (The Dispatcher updates these locations, as well as loc. %1223, the number of CSTX segments in the program, whenever a process is launched.) To transfer to logical segment 2, the CPU looks at Entry 2 in the current LSTT, which contains the physical segment number of the target segment (Fig. 7). The CPU then uses this physical CST number to index into the system CST, where the appropriate entry points to the code segment in the usual manner. This physical segment number can be as high as 2047, which is the new maximum number of CST entries.

When the CPU transfers control to a new code segment, it must determine whether the segment is physically or logically mapped. To do this, changes were made to some familiar architectural structures: the STT (Segment Transfer Table) and the Stack Marker.

To explain the differences between the old and new approaches, we should first look at a few examples of the old (current) method. PCAL will be used as an example; refer to Fig. 8 for illustration.

Suppose a PCAL 1 is executed. First, a Stack Marker is written on the stack. Then, the CPU checks to see if 1 (the PCAL operand) is less than or equal to the number of Plabels in the STT (STT Length), found at PL-0. If so, the Plabel at PL-1 is read. Bit 0=0 signifies that this is a local Plabel, in the format shown. If Bit 1 (U)=1 the segment is uncallable, and user mode callers will abort with an STT Uncallable Violation. Finally, Bits 2-14 are a PB-relative offset into the current segment. After bounds checking assures that (PB <= PB + new Delta-P <= PL), PB and P are set to the new values, which transfers control to the new procedure.

Now, suppose a PCAL 5 is executed (Fig. 9). A stack marker is written, and if 5 <= # of Plabels, PL-5 is read. Bit 0=1 signifies that this is an external Plabel. That means the procedure we want is in another code segment, and the first order of business is to find that other segment. The segment number is in the Plabel, bits 8-15. (Note that Seg# is only 8 bits long here as well!) If Seg# is less than 192, the CPU looks in the CST; otherwise it looks in the CSTX. The 4-word CST entry pictured in Fig. 3 begins at the following address:

For CST: Seg# * 4 + CST Base
For CSTX: (Seg# - 192) * 4 + CSTX Base

The CST Entry tells the CPU if the segment is present or absent. If absent, the PCAL stops here, and MPE is awakened via an Absence Trap. If the target segment is present, the CST Entry gives the absolute starting address (Bank and Base) as well as the segment length. Assuming the segment is in memory, the new PL value is calculated, and the Plabel at (new PL - STT#) is read, where STT# is bits 1-7 of the Plabel already read from the current segment. The new Plabel read from the target segment must be in internal format; otherwise the process will abort with an STT Violation. At this point, bounds checking is performed and control transferred as described above.

With MPE V and the new firmware, the STT will look a bit different (Fig. 10). Bit 0 of a Plabel will no longer be used to signify internal/external, but instead to indicate whether the target code segment is physically or logically mapped. The internal/external determination will be made via word 0 of the STT, which will hold "number of local Plabels" as well as "number of Plabels". The local Plabels are always written at the beginning of the STT, so for a PCAL n the determination is made as follows:

$$n = \begin{cases} 0 \text{ -> Use Plabel from TOS; defined to be in external format.} \\ 0 < n <= \#\text{local Plabels - - -> Local format} \\ \#\text{local Plabels} < n <= \#\text{Plabels -> External format} \\ \#\text{Plabels} < n \text{ - - - - - -> STT Violation} \end{cases}$$

When an internal PCAL is performed (calling a procedure in the current segment), PCAL will operate exactly as today, as described previously. But if an external PCAL is done, the CPU will behave quite differently (Fig. 11a). First, the stack marker will be written and the Plabel at PL-n read as before. If bit 0 of the Plabel is 1, signifying a physically mapped segment, the segment number is used to directly access the CST as before, although the allowable range of segment numbers will now be 1-255 instead of 1-191. The CPU's mapping flag is set, and control is transferred. However, if bit 0 of the Plabel is 0, then the CPU must first check memory loc. %1223, which tells how many segments the program file has. If segment number <= # program segments, then the code segment entry will be found in the CSTX entry block for this program (Fig. 11b). (As previously described, memory loc. 1 points to the current CSTX block.) If Seg# > # program segments, then this is a shared (SL) segment (Fig. 11c); furthermore, the Seg# that we have is a logical number. We must look in the LSTT at loc. (2 * logical Seg#) to find the physical Seg#. This physical Seg# is then used to access the CST, which now has 2047 as largest possible segment number. Note that the Status register reflects the logical segment number, but the mapping flag says "logical mapping". This enables the CPU to know where to look for a given code segment.

Suppose the CPU has been executing in logical segment 5, and then does a PCAL to logical segment 6, where both segments are SL segments with entries in the CST. The Mapping flag shows logical mapping, but this is an internal flag, not accessible to the program. To leave seg. 6, the procedure does an EXIT instruction. EXIT finds the return segment number in the old Status register, stored at Q-1 in the stack marker. However, the old Status register merely says "5" for Seg #; this could be an EXIT to either physical seg. 5 or logical seg. 5. How does the CPU know which? With MPE V and the new firmware, a bit in the stack marker has been reassigned to save the mapping flag (Fig. 12). Bit 0 of the Delta-P (Q-2) has always meant that a Control-Y interrupt was pending; this is set by MPE when a Control-Y interrupt is received. Bit 1 signified that a Trace interrupt was pending, again set by MPE. With the new system, Bit 0 will indicate that either a Control-Y interrupt or a Trace interrupt is pending. Because of the way this was defined in the past, it will be easy for MPE to differentiate between the two. Bit 1 of Delta-P will be the old mapping flag. Thus, the stack marker contains both the segment number and the mapping flag. When the EXIT is executed, the CPU can find the appropriate code segment by getting its entry from the CST or CSTX, using the LSTT if returning to a logical segment. The method used is the same as for PCAL, except that the STT is not referenced. There is no need to find a Plabel, because we already know the target segment number and PB-relative return address.

Some of you who see memory dumps from time to time will notice that the Delta-P values of MPE segments (in an MPE V dump) seem unreasonably large. Actually, you are seeing the old mapping flag laid down in the stack marker. MPE segments are usually physically mapped, and Delta-P Bit 1=1 for physically mapped segments--so it looks as if all MPE segments have a Delta-P of %40000 or larger.

You may have noted in Fig. 6 that the LSTT is divided into 2 parts. In the first part is a two-word entry for each logical segment used in this program, containing the physical segment number and a pointer to an "External Label List". The second part contains an External Label List for those shared (SL) segments which are referenced by this program. The reason is, the STT of a sharable segment cannot contain a logical segment number in an external Plabel since more than one program may be sharing that segment.

For example, imagine the following procedure in a sharable segment:

```
procedure A;
begin
  B;
end;
```

Imagine that B is also in a sharable (SL) segment. Now, I run PROGX, which has 2 segments and calls A. The loader will set up the STT in PROGX such that segments 1 and 2 are the PROGX segments, Seg. 3 contains A, and Seg. 4 contains B. Now, since A calls B, the STT of A must reflect the fact that B is in Seg. 4. (See Fig. 13) So far, so good---but now you run PROGY, which also calls A, and has only one segment. The loader must assign the number "1" to the PROGY segment, and "2" and "3" to the segments containing A and B. But now, the STT for A must reflect that B is in logical Seg. 3---which is somewhat difficult, since it must also show that B is in logical Seg. 4!

The problem is solved by using "0" as the logical segment number of every sharable segment called by another sharable segment. In other words, the STT of a sharable segment contains 0 as the Seg# for those Plabels referencing other sharable segments. When the CPU encounters "logical seg. 0" during a PCAL, it looks at the LSTT to find the External Label List for the current segment.

Using the above example, when A does a PCAL n to get to B, the CPU finds a 0 for Seg# at PL-n. It then looks in the Status register to get the current logical Seg#, and reads LSTT(2*Seg#+1). It can then index into the External Label List, which contains all the External Labels for this particular execution of the current segment; there it will find the correct Plabel. [The Plabel is actually found at (beginning of External Label List) - n + # local Plabels. This is because only the external Plabels are there, so we add in the number of local Plabels to effectively "skip over" the nonexistent local ones.]

In the preceding examples, only PCAL and EXIT have been discussed. Several other instructions were also changed to support the new CST/STT accessing scheme. These changes are briefly described below:

LLBL - Fetches a Plabel in the same manner as PCAL, accessing the LSTT if necessary.

IXIT - Performs a transfer of control just like EXIT (in fact, it executes the same microcode).

SCAL - Works the same as it always has, but has to know about the new STT format to know if the target Plabel is internal or external.

Interrupts - Interrupts do implicit PCAL's to MPE interrupt routines. These implicit PCAL's work just like ordinary PCAL's.

COBOL 74 instructions - The instructions XBR, PARC, and ENDP are special intructions generated by the COBOL 74 compiler to create more efficient COBOL object code. They provide a method of transferring directly from one code segment relative address to another, using a segmeng # and offset instead of an STT with a Plabel. Hence, these instructions resemble EXIT in their method of control transfer. When the transfer is external (to another code segment), the same considerations are in effect--the LSTT must be consulted if the target segment is logically mapped.

We've discussed the architectural changes necessary to support the advances in the MPE V operating system. However, it should be noted that the bulk of the engineering work went into improvements to MPE itself, not into the firmware changes described here. It is beyond our scope to discuss all of the improvements to MPE, but the list is substantial, and hundreds of "engineer-months" were required to complete the project. Other presentations will discuss methods used to coordinate all the work done on MPE, as well as some of the implications of the new software.

I couldn't close without thanking Dan Mathias of CSY Software R&D for his help, both in helping me to understand the whole mess (!), and in permitting me to use some of his working documents in this paper.

SELECTED HP3000 ARCHITECTURAL DEVELOPMENTS

	Series I	Series II, III	Series 30,33,40,44,64	Series 42, 48, 68
Code Segments (maximum)	255	192 SL (incl. MPE) +63 per program	192 SL (incl. MPE) +63 per program	255 MPE +255 per program (incl. SL)
I/O	Parallel/Differential (SIO)	Parallel/Differential (SIO)	HP-IB	HP-IB
Extended Precision Floating Point representation	48-bit	64-bit	64-bit	64-bit
MPE Memory Resident Tables				
must reside in:	Bank 0	Bank 0	Bank 0	Banks 0 through 31
accessed via:	LOAD/STOR	LST/SST	LST/SST	modified LST/SST

SLIDE I

Ⓐ

STOR **Store TOS into memory. The content of the TOS is stored into the effective address memory location, and is then deleted from the stack.**
Memory opcode 05, bit 6 = 1
Indicators: unaffected
Addressing modes DB+, Q+, Q-, S- relative
Direct or indirect
Indexing available
Traps STUN, BNDV

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	1	X	1	1									

Mode and Displacement

LOAD **Load word onto stack. The content of the effective address location is pushed onto the stack.**
Memory opcode 04
Indicators CCA
Addressing modes P+, P-, DB+, Q+, Q-, S- relative
Direct or indirect
Indexing available
Traps STOV, BNDV

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	X	1										

Mode and Displacement

Machine Instruction Set

Ⓑ

LST **Load from system table. The X register contains a value which is used to index into a table pointed to by the contents of location %1000+K if K is non-zero, or by the contents of location %1000+A if K is zero. The table pointer itself is also relative to location %1000. The data accessed in the table is pushed onto the stack if K is non-zero or replaces A if K is zero.**
Special opcode 00
Indicators CCA
Traps STUN, STOV, MODE
This is a privileged instruction.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	0	0	0	0	0	0	0				

K

SST **Store into system table. The X register contains a value which is used to index into a table pointed to by the contents of location %1000+K if K is non-zero, or by the contents of location %1000+A if K is zero. The table pointer itself is also relative to location %1000. The data contained in A if K is non-zero or in B if K is zero is stored into the calculated address. The stack is then popped by one if K is non-zero or by two if K is zero.**
Special opcode 15
Indicators unaffected
Traps STUN, MODE
This is a privileged instruction

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	0	0	0	1	1	0	1				

K

Ⓒ For the new LST and SST, the phrase "The table pointer itself is also relative to location %1000" is replaced by "The table pointer contains the bank of the table in bits 11:5. The base address is computed by taking the 16-bit table pointer, setting bits 11:5 (the bank address) to 0, adding %1000, then adding X."

Figure 1

Referencing the System Global Area (SYSGLOB)

Example Layout

```
Memory
Loc.      ____
    %1000|
     1001|
         :
     1005|   %4000 (SYSDB-relative pointer to base of IOQ)
     1006|
         :
     1205|
     1206|   $101400 (SYSDB-relative pointer to base of Example Table)
     1207|
         :
     5023|
     5024|   %123456 (20th word of IOQ)
     5025|
         :
   102435|
   102436|   %654321 (30th word of Example Table)
   102437|
         :
         :
end of SYSGLOB|___
```

Examples (a) and (b): Read DB+5 (=%1005) = %4000

Now, compute the effective address

E = %4000 + DB + X = %4000 + %1000 + %24 = %5024

Now, read the word at %5024 (%123456) and push it onto TOS.

(For case (b), substitute SYSDB for DB. SYSDB is always %1000.)

Example (c): Push 134 (=%206) onto TOS and execute an LST 0.

Read SYSDB + TOS (%1000 + %206) = %101400.

Now compute the effective address:

E = %101400 + SYSDB + X = %101400 + %1000 + %36 = %102436.

Now read (E) = %654321 and push it onto TOS.

Figure 2

Locating a Code Segment via the CST Entry

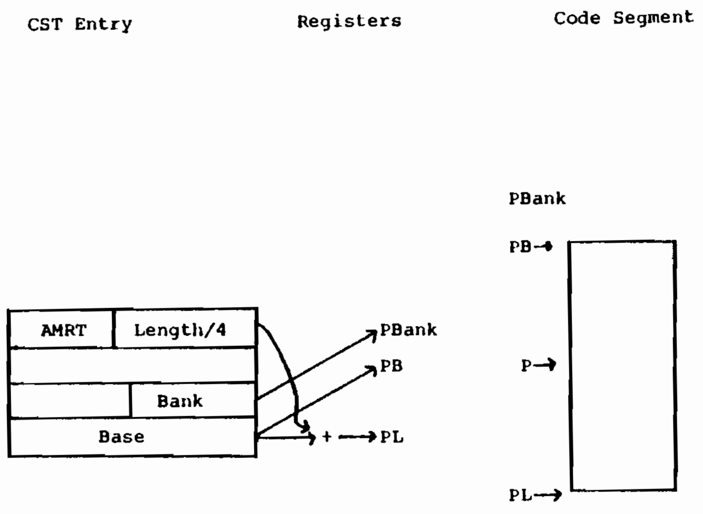
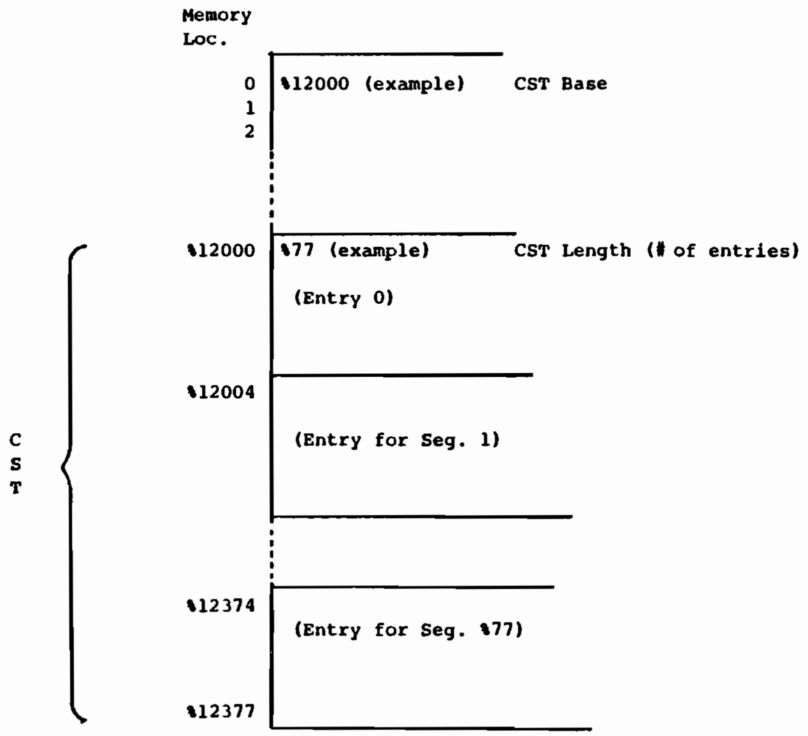


Figure 3

Code Segment Table Structure

Series I and Previous Systems



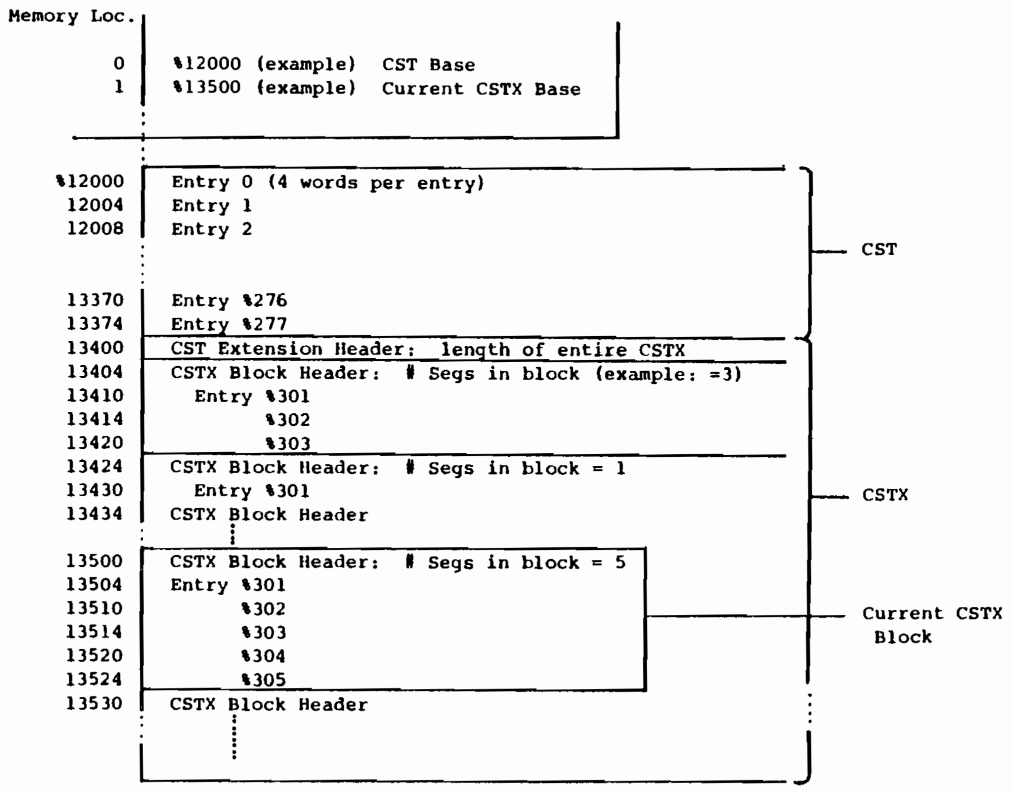
To find entry for Seg. 2, look at CST Base (%12000) + 2 * entry size

= %12000 + %10 = %12010.

Figure 4

Code Segment Table Structure

Series II through Series 64 (Pre-MPE V)



Memory Loc.
0 %12000 (example) CST Base
1 %13500 (example) Current CSTX Base
%12000 Entry 0 (4 words per entry)
12004 Entry 1
12008 Entry 2
13370 Entry %276
13374 Entry %277
13400 CST Extension Header: length of entire CSTX
13404 CSTX Block Header: # Segs in block (example: =3)
13410 Entry %301
13414 %302
13420 %303
13424 CSTX Block Header: # Segs in block = 1
13430 Entry %301
13434 CSTX Block Header
13500 CSTX Block Header: # Segs in block = 5
13504 Entry %301
13510 %302
13514 %303
13520 %304
13524 %305
13530 CSTX Block Header
CST
CSTX
Current CSTX Block

Figure 5

Logical Segment Transform Table(LSTT)

```
+-----------------------------+
| # of Logical Segments       |
+-----------------------------+
| Length of LSTT              |
+-----------------------------+ ---
| Physical Segment #          |
+-----------------------------+  logical segment 1
| Ptr to External Label List  |
+-----------------------------+ ---
| Physical Segment #          |
+-----------------------------+  logical segment 2
| Ptr to External Label List  |
+-----------------------------+ ---
|             .               |  .
|             .               |  .
|             .               |  .
|             .               |  .
+-----------------------------+ ---
| Physical Segment #          |
+-----------------------------+  logical segment n
| Ptr to External Label List  |    (max 255)
+-----------------------------+ ---
|M| STT #      | SEG #        |
+-----------------------------+  External Labels
|M| STT #      | SEG #        |  from Logical
+-----------------------------+  Segment 1
|              .              |  (if needed)
|              .              |
+-----------------------------+
|M| STT #      | SEG #        |
+-----------------------------+ ---
|              .              |  .
|              .              |  .
|              .              |  .
|              .              |  .
+-----------------------------+ ---
|M| STT #      | SEG #        |
+-----------------------------+  External Labels
|M| STT #      | SEG #        |  from Logical
+-----------------------------+  Segment n
|              .              |  (if needed)
|              .              |
+-----------------------------+
|M| STT #      | SEG #        |
+-----------------------------+ ---
```

Figure 6

Code Segment Table Structure

Series 42, 48, 68 (MPE V)

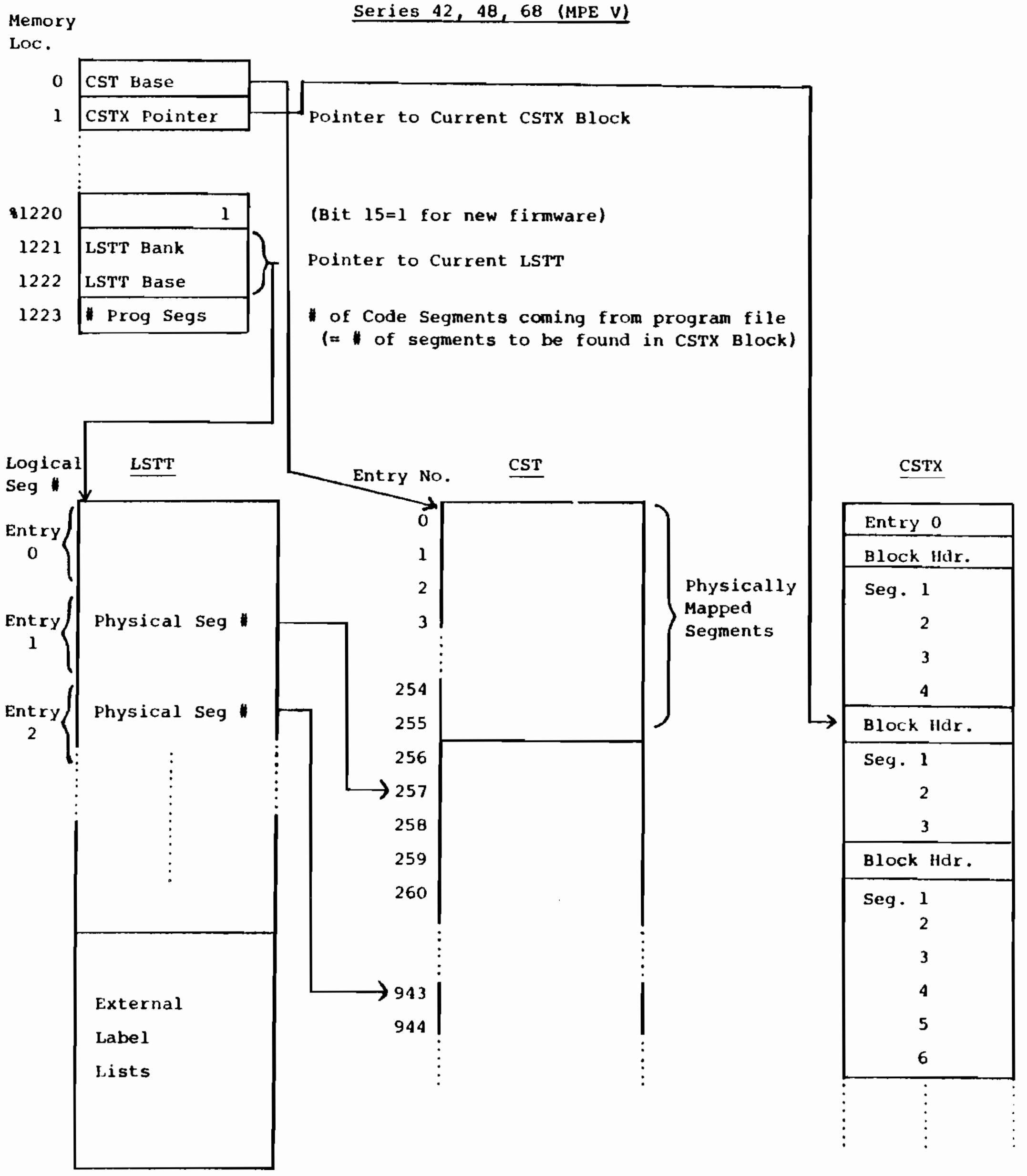


Figure 7

Code Segment and Present STT Structure

```
      +-----------------------------+
PB    |                             |
      |            Code             |
      |                             |
      |                             |
      +-----------------------------+  --
      |                             |   |
      |                             |   |
      +-----------------------------+   |  External
      |1| STT #       | SEG #       |   |  Labels
      +-----------------------------+   |
      |1| STT #       | SEG #       |   |
      +-----------------------------+  --
      |                             |   |
      |                             |   |
      +-----------------------------+   |  Local
      |0|U|  Address                |   |  Labels
      +-----------------------------+   |
      |0|U|  Address                |   |
      +-----------------------------+  --
PL    |0|U|   0       |# of Labels  |
      +-----------------------------+
```

Bit 0 of the label designates whether the entry is an internal or external label. "U" designates whether the local label is callable or uncallable.

Figure 8

External PCAL Example

(Series II through 64)

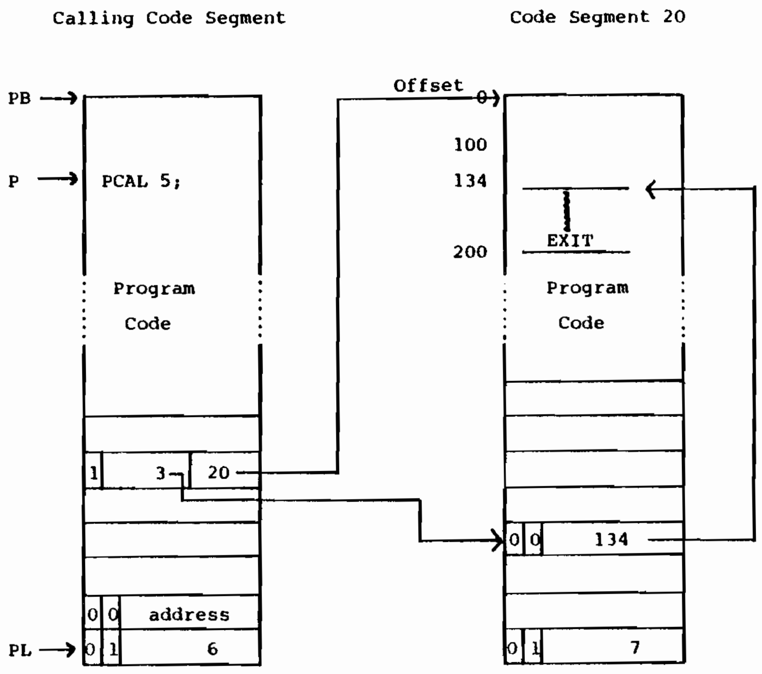


Figure 9

Code Segment and STT Structure
for Logical Mapping

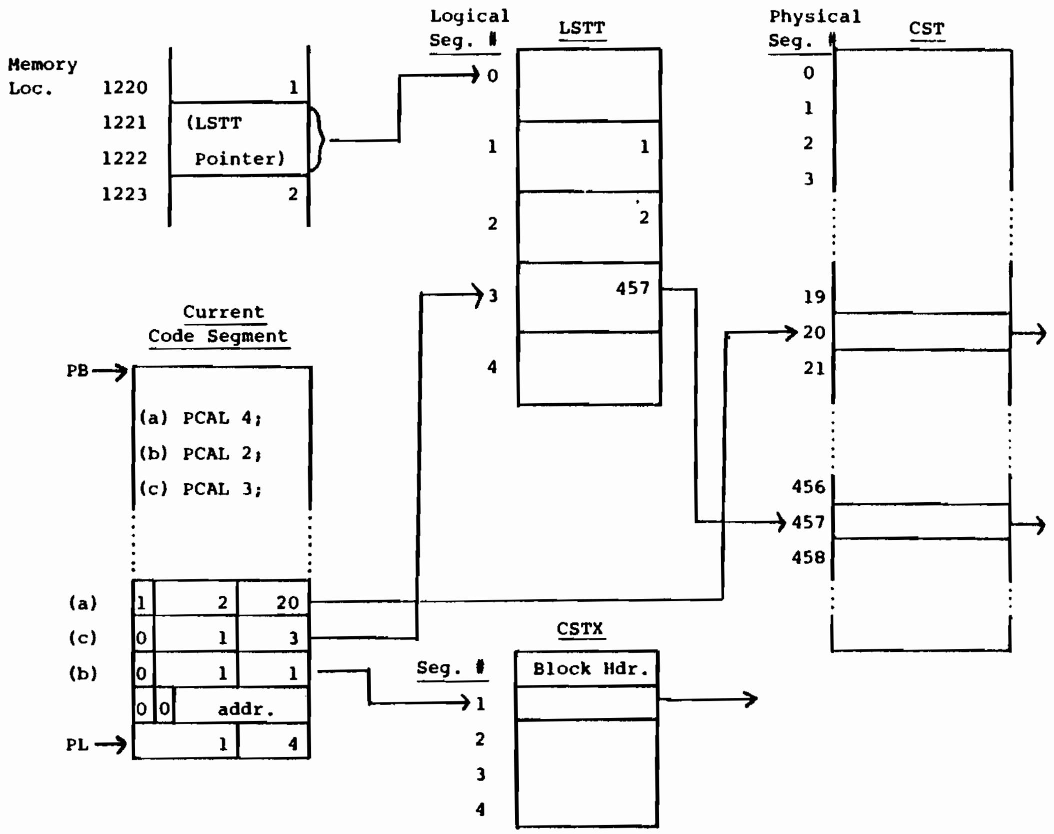
```
       +----------------------------+
PB     |                            |
       |          Code              |
       |                            |
       |                            |
       +----------------------------+   --
       |                            |   |
       |                            |   |
       +----------------------------+   |   External
       |M| STT #       |  SEG #     |   |   Labels
       +----------------------------+   |
       |M| STT #       |  SEG #     |   |
       +----------------------------+   --
       |                            |   |
       |                            |   |
       +----------------------------+   |   Local
       | |U|  Address               |   |   Labels
       +----------------------------+   |
       | |U|  Address               |   |
       +----------------------------+   --
PL     |#Local Labels|# of Labels   |
       +----------------------------+
```

Whether a label is local or external is determined by using the two counts at the head of the STT. "M" is used to designate whether the segment number in the label is a physical CST number or it is a logical CST number and must undergo a logical mapping through the LSTT to obtain the physical CST number (1=physical, 0=logical).

Figure 10

External PCAL Examples

(Series 42, 48, 68)



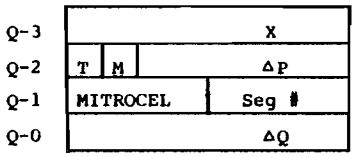
(a) PCAL 4 transfers directly to STT #2 of Seg. 20, because this Plabel specifies physical mapping.

(b) This Plabel specifies logical mapping. Seg. # is 1, which is less than the number of program segments; so, this is a program segment, and we have to look at Entry 1 in the current CSTX Block.

(c) Since the Plabel specifies logical mapping, and 3 is greater than the number of program segments, we look at LSTT Entry 3 to find the physical code segment number; this is seen to be 457. We then look in the CST for Entry 457, which points to the code segment.

Figure 11

New Stack Marker



T = Trace or Control-Y interrupt pending

M = Physically mapped code segment

Figure 12

A Problem with Shareable Segments

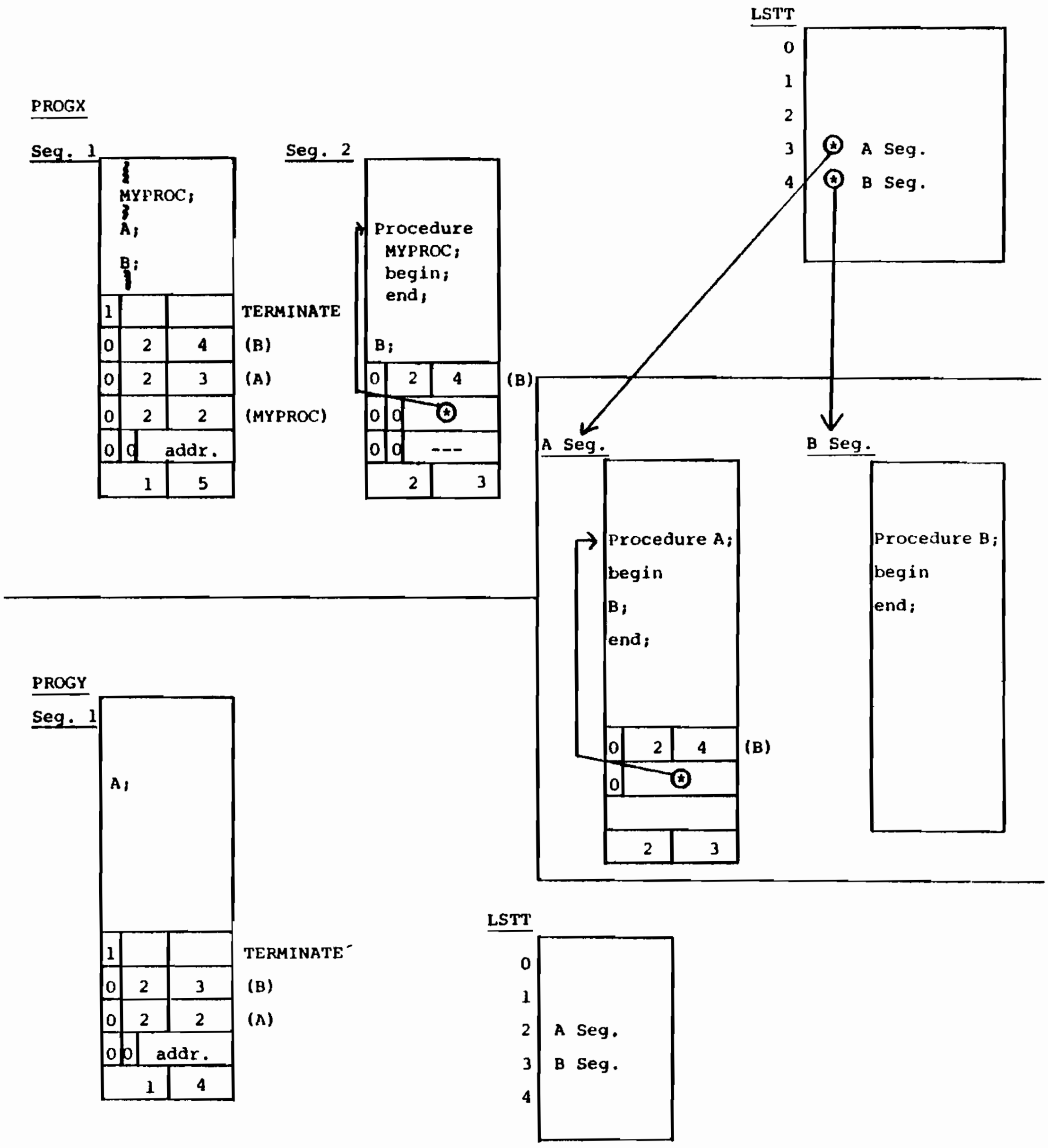
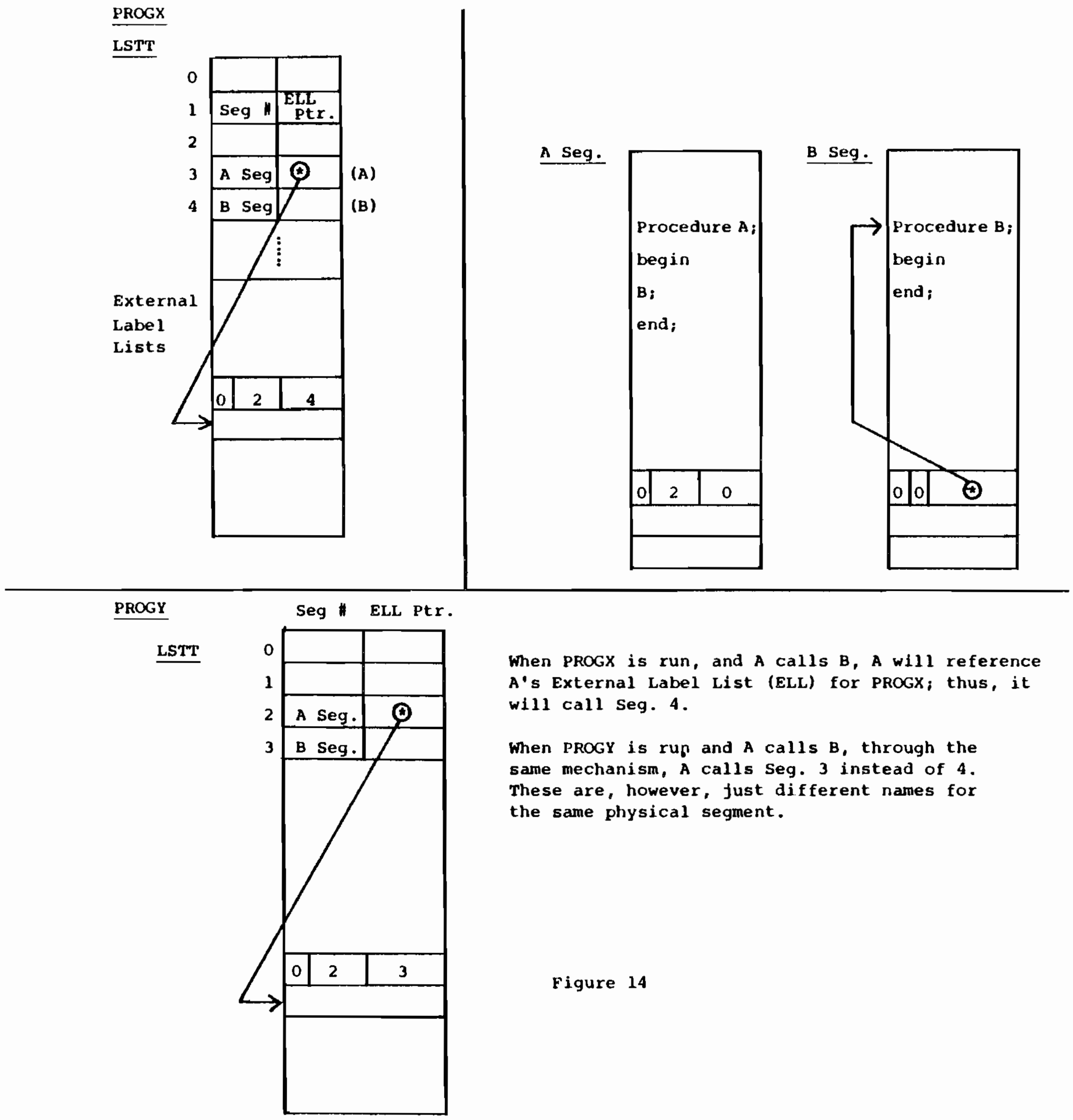


Figure 13



When PROGX is run, and A calls B, A will reference A's External Label List (ELL) for PROGX; thus, it will call Seg. 4.

When PROGY is run and A calls B, through the same mechanism, A calls Seg. 3 instead of 4. These are, however, just different names for the same physical segment.

Figure 14