PRIVILEGED MODE -- HOW TO USE IT SAFELY AND EFFECTIVELY
With Practical Applications

by Joseph C. Felix and Chris Hauck
Software Design Specialists
Educational Computer Systems, Inc.

INTRODUCTION

Privileged mode on the HP3000 is a concept which is often misunderstood by HP3000 users. It's the subject of many conversations and heated arguments. Many people are curious about PM but avoid it out of fear.

The authors feel that when used properly, privileged mode can be a very powerful tool. We openly admit that we don't know everything thing there is to know about privileged mode and MPE. We do feel, however, that countless hours of heuristic programming with a System Tables Manual in one hand and a cold load tape in the other, have given us valuable knowledge which we are only too glad to share.

This paper is intended to serve as a springboard for the experienced programmer who wishes to start experimenting with PM. We also hope to give some new ideas to PM gurus in return for the techniques we have "borrowed" from them.

The paper does touch on some internal MPE tables. Our intent is not to describe the tables in depth; to do so would be a monumental task. We, instead, will explain relationships between tables and describe how to access the tables. Anyone serious about privileged mode programming MUST obtain a MPE System Tables Manual to effectively make use of the techniques and information we supply here.

All information here is accurate to the best of our knowledge. All examples have been tested and they do run on the Q-MIT level C release of MPE. Of course, the authors assume no liability for consequences arising from or out of the use of information contained herein.

WHAT IS PRIVILEGED MODE?

Hewlett-Packard defines privileged mode as follows:

Privileged mode is characterized by the ability to execute privileged instructions and to call segments which have been declared UNCALLABLE.

The normal mode of operation on the HP3000 is called user mode. In order for a program running in user mode to do anything requiring privileged mode (such as input/output), it must call a non-privileged procedure or intrinsic which in turn calls the privileged routines necessary to complete the task. The person running in privileged mode can skip this intermediate step and DIRECTLY call the procedures or execute the machine instructions necessary.

Privileged mode is an attribute of accounts, groups, users, and code segments. In order for a user of group to possess privileged mode capability, the account must have it. Similarly, in order for a privileged program to run, it must reside on a group with PM capability.

WHY USE PRIVILEGED MODE?

Most people realize that one slip while running in privileged mode can destroy file integrity or crash the system, and for them, that's reason enough to avoid PM. However, there are many applications in which privileged mode can be put to good use.

1. It lets you access files with a negative file code - specifically, IMAGE data bases. Hence, you need PM (or at least OP capability) if you want to :STORE or :RESTORE a data base. Sure, DBSTORE works just as well, unless you want to store multiple data bases. Almost all data base utility programs use privileged mode.

2. Privileged mode is extremely useful for overcoming MPE bugs or shortcomings. Have you ever wanted to have a UDC repeat itself? or switch to a different group without logging on again? It's possible; use privileged mode. How about a utility program to show you all users on the system who are accessing a particular file or show all files currently accessed by a particular user? Did you ever want to see your temp files displayed in a :LISTF,2 format rather than that printed by LISTEQ2.PUB.SYS?

3. Another reason for using privileged mode is that it lets you take advantage of "short-cuts" built into MPE by HP for their convenience. Several intrinsics operate differently when called in privileged mode. The privileged user can use machine instructions not available to others for transferring data to and from any data segment on the system. Complex input/output routines become extremely simple. Read on -- we'll show you.

HOW TO IMPLEMENT PRIVILEGED MODE

Before getting into specifics, it's important to know exactly how to use privileged mode. There are basically three different ways of implementing privileged mode.

1. IN A PRIVILEGED PROGRAM -- You type in the program using EDITOR, and compile it as usual. The program must be :PREPped with the parameter CAP=PM in addition to any other capabilities you desire. In order to :RUN the program, you must :SAVE it on a group which has PM capability. If you run it as $OLDPASS or a TEMP file, then the user you are logged on as must have PM.

2. AS A PROCEDURE IN AN SL -- You may choose to write your routine as a procedure or function, omitting the outer block. Compile your procedure as usual, then use the :SEGMENTER to add it to an SL on a group having PM capabilty.

3. DEBUG -- Another way to use privileged mode is in privileged DEBUG. This is often overlooked by people, but is frequently the best way to make small changes, or spot check a system table. Privileged mode DEBUG can be entered by any user posessing PM capabililty simply by typing the :DEBUG command. Once in PM DEBUG, the inexperienced user should be cautious; it is very easy to crash the system while in PM DEBUG.

LANGUAGE CONSIDERATIONS

Once you understand some of the possibilities of using privileged mode, you must choose a language for your program. A program for accessing privileged files can be written in almost any language supported on the HP3000. However, we don't recommend using fourth generation languages, or RPG, or BASIC, or COBOL 68. These languages are awkward to use when writing programs that involve calling intrinsics. COBOL II, FORTRAN, and PASCAL make such programs easier, but we find SPL to be the best choice. SPL is the "native language" of the HP3000. Since the MPE operating system is written in SPL, anything that can be done on the HP3000 can be done in SPL. This is not necessarily true of other languages. SPL provides the programmer with the ability to call intrinsics easily, the ability to write in machine language, and the ability to access absolute memory locations. Since SPL is not taught at many universities, it's hard to find SPL programmers. However, SPL is an easy language to learn and anyone with a fairly decent programming background should have no trouble mastering it. Because of SPL's structure, it makes an excellent language for presenting examples. Anyone familiar with PASCAL, FORTRAN, or COBOL will have little trouble understanding a well- documented SPL example.

WRITING SAFE PM PROGRAMS

We've compiled a list of things to keep in mind when writing privileged mode programs or procedures. Most of these apply to privileged mode programs which access system tables, but many are applicable to programs using privileged files and other types of PM applications.

If you've ever wondered about the "crash potential" of a particular PM program, these guidelines can serve as faily decent criteria for evaluation of a PM program or procedure.

1. TEST YOUR PLANNED PROGRAM WITH PM DEBUG FIRST, IF POSSIBLE. If you're planning to write a program that prints entries from an MPE table, or goes in and modifies one, it's a good idea to try to do it in DEBUG first. By checking your approach step by step, you can spot any flaws in locating the information you desire. Privileged Mode DEBUG is not foolproof, however. Be sure your system manager knows of your activities and by all means, do your experimentation on an unloaded system.

2. START BY WRITING READ-ONLY PM PROGRAMS, NOT PROGRAMS WHICH GO IN AND CHANGE TABLES. We feel that one of the shortcomings of MPE is the lack of a "Read Only Privileged Mode" capability. The way MPE was designed, PM capability gives the user the ability to access AND CHANGE anything in the operating system. This makes it all too easy to accidentally destroy MPE. Though it may seem obvious, we recommend that the novice PM user stick with programs which read tables and PM files, and stay away from programs which change them.

3. USE TEMPORARILY PRIVILEGED PROGRAMS WHENEVER POSSIBLE. It's always a good idea to minimize the time your program spends running in privileged mode. For this reason, you should use the GETPRIVMODE and GETUSERMODE intrinsics to bounce in and out of privileged mode. Stay in user mode as long as possible, then GETPRIVMODE, do your thing, then GETUSERMODE.

4. ANTICIPATE POTENTIAL BOUNDS VIOLATION SITUATIONS. Since MPE has been in operation, bounds checking for programs running in privileged mode has been minimal (although it's rumored that the Series 64 now does PM bounds checking). Operations which produce bounds violations in user mode execute OK in privileged mode, sometimes destroying data in undesired locations. To prevent this, your program should check DST sizes, and array boundaries, etc.

5. DST CHECKING. Data Segment number 2 is a table containing information about all data segments in the system. It's always a smart move to check this data segment to be sure you're not going to try to access a nonexistent data segment or read past the end of an existing one.

6. NEVER :RELEASE A PM PROGRAM. Any released file can be written to by any user on the system. A released PM program could be potential disaster. Anyone can change this program to do what they want it to, or get into PM debug. We recommend the use of lockwords to help you keep a constant watch on the security of your privileged programs.

7. DISABLE TRAPS WHILE EXECUTING PRIVILEGED CODE. To prevent unwanted interrupts while your privileged program is running, disable arithmetic traps, user interrupts, and control Y handling Interrupts are not normally a problem, as long as your program is prepared to handle them. To keep programs simple and straightforward, we recommend disabling traps.

8. DISABLE THE BREAK KEY. The last thing you want to have happen in your privileged program is to have a user press break and :ABORT in the middle of a table change. By all means, disable the break key; especially if your program is going to do any table or file updating.

9. CHECK THE MPE VERSION OR SYSTEM CONFIGURATION FOR DEPENDANT FEATURES. If you know for sure that your program works on one particular version of MPE, you might want to check for that version before you have your program do its thing. This is particularly important if you suspect that the program may not run on future releases of MPE. Locations %114-%116 in DST #5 (System Global Area) contain the current update, fix level, and version of MPE.

10. YOU MAY WANT TO CHECK THE CPU TYPE. Use the privileged instruction PCN (Push CPU Number) or the callable procedure THISCPU. We recommend THISCPU for the simple fact that PM is not needed to call it. The return values are:

CPU TYPE	PCN RETURNS	THISCPU
II	1	1
30,33	8	2
III	2	3
40,44	3	4
64	4	5

11. USE ENTRY SIZE VALUES IN TABLE HEADERS OR ENTRY NUMBER ZERO OF MOST TABLES. The idea here is to hard-code as little information as possible about system tables within your program. Many tables contain information about themselves such as the number of entries, entry size, etc. By using the values contained in the tables, your program is more likely to run on future releases of MPE. If HP changes the table size, your program will pick it up at run time, and still be OK.

12. KEEP GOOD DOCUMENTATION. The importance of this can not be overstressed. Major items to document are: A. Assumptions your program makes, if any. 1. DSTs of particular tables 2. Sizes of particular tables 3. Relationships between tables B. DEBUG test procedures -- How can you verify the operation of your program using PM DEBUG C. Table names and entries your program uses D. Table names and entries your program changes (if any)

Also, keep your documentation up to date. You'll need it in the next step.

13. Re-evaluate your program after major releases of MPE. HP often restructures tables from one release of MPE to another. Past experience has shown that though the tables may change and data may move around, the information you need is still in there somewhere. Your PM program should require little modification, if any, to go from one release of MPE to another. However, since there is always the possibility of table changes, we recommend that you do take the time to analyze your program to prevent disastrous results.

14. USE SIRS AND SETCRITICAL WHEN APPROPRIATE. MPE handles table contention through the use of SIRs. Certain tables should only be accessed after you've locked (obtained) the corresponding SIR. The MPE System Tables Manual has a list of tables and their corresponding SIRs. It's also important that you get SIRs in the correct order. Never attempt to get a SIR with a higher number than any SIR you already hold. A SIR deadlock could result.

A program running in privileged mode can also call SETCRITICAL and RESETCRITICAL. Once a procedure is "CRITICAL", the system will continue to execute this procedure only, until it calls RESETCRITICAL. This is another way to make sure a table has not changed between successive reads or between a read and an update. Be careful, if your process aborts for any reason while it is CRITICAL, a system failure 311 will result. Likewise, if you get a SIR, you better release it before termination or a system failure 314 will ensue.

MPE TABLES

As mentioned in the introduction, we intend to provide a brief introduction to some MPE tables. Table descriptions are kept purposely brief. Our goal is to acquaint you with some important MPE tables, show useful examples, and encourage further reading on your part.

The tables we'll introduce are divided into three major classes:

1. Job/Process Tables 2. File System Tables 3. I/O Tables

We'll describe the major tables in these classifications and explain the appropriate inter-relationships. Following the table explanations, we'll present examples to aid understanding.

JOB/PROCESS TABLES

THE DATA STACK -- Every process has a stack associated with it. We've all seen the familiar pictures of the stack showing the area from DL to Z. What these pictures don't show is the area beneath DL. This area is called the PCBX and is further divided into three areas:

PXGLOB - area containing JOB TABLE pointers & indices

PXFIXED - misc. information about this process

PXFILE - the file system section of the PCBX

Since a data stack exists for all processes in a job, the PXGLOB area is almost identical in all stacks belonging to the same job. When a process terminates, its data stack goes away. A job or session is just a special type of process. Therefore, if you wish to make any changes to the PCBX area, you must do it on the stack of your main process in order for it to stay until you log off. Since the job-related information in the PXGLOB area is the same for all processes in a job, you can use the PXGLOB area of the process (program) you are running to get to your job tables.

The PXGLOB area has pointers to the following job tables: JMAT, JPCNT, JDT, JIT, and JCUT. The following is a brief description of these tables. Refer to figure 1 for a related diagram.

JMAT -- The JMAT (Job Master Table) is one table presently located at DST %31. This table contains a header entry followed by one entry per logged on job or session. An entry in the JMAT contains (among other things): the job/session number, user, account, group, and job name, input and output device numbers, time and date of logon, PIN of the job/session main process, log on priority, CPU time limit, etc.

JPCNT -- This table (Job Process Count Table) contains a byte for each logged on job or session and is used in the allocation and deallocation of SIRs for jobs.

JDT -- A JDT (Job Directory Table) exists for each job on the system. This is unlike the JMAT and JPCNT which have all jobs and sessions contained in a single table. The JDT is composed of five smaller tables. These five tables contain information regarding data segments, temp files, file equations, line equations, and JCWs used by the job.

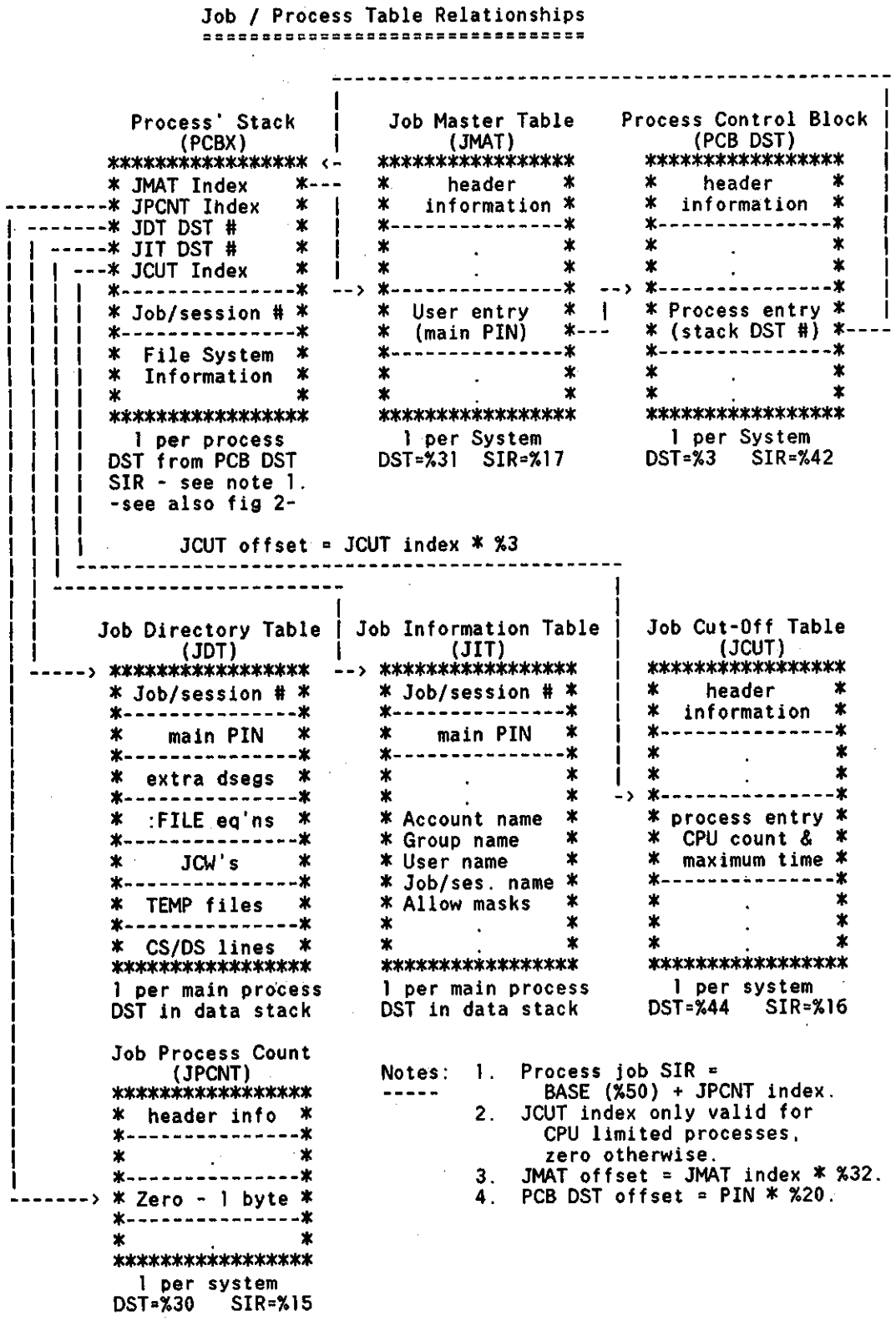
JIT -- The JIT (Job Information Table) is similar to the JDT in that a different data segment is used for each job on the system. A job's JIT contains information such as the job's account security, group security, account name, home group, logon group, user name, job name, local attributes, ALLOWed commands, account- ing information, directory pointers for the account and group, the PIN of the main process, etc.

JCUT -- The JCUT (Job Cutoff Table) is a single table located at DST %44. This table contains one entry for each CPU limited job or session. If your job or session has a limited number of CPU seconds, it will have an entry in this table. When your CPU limit is reached, you will be logged off.

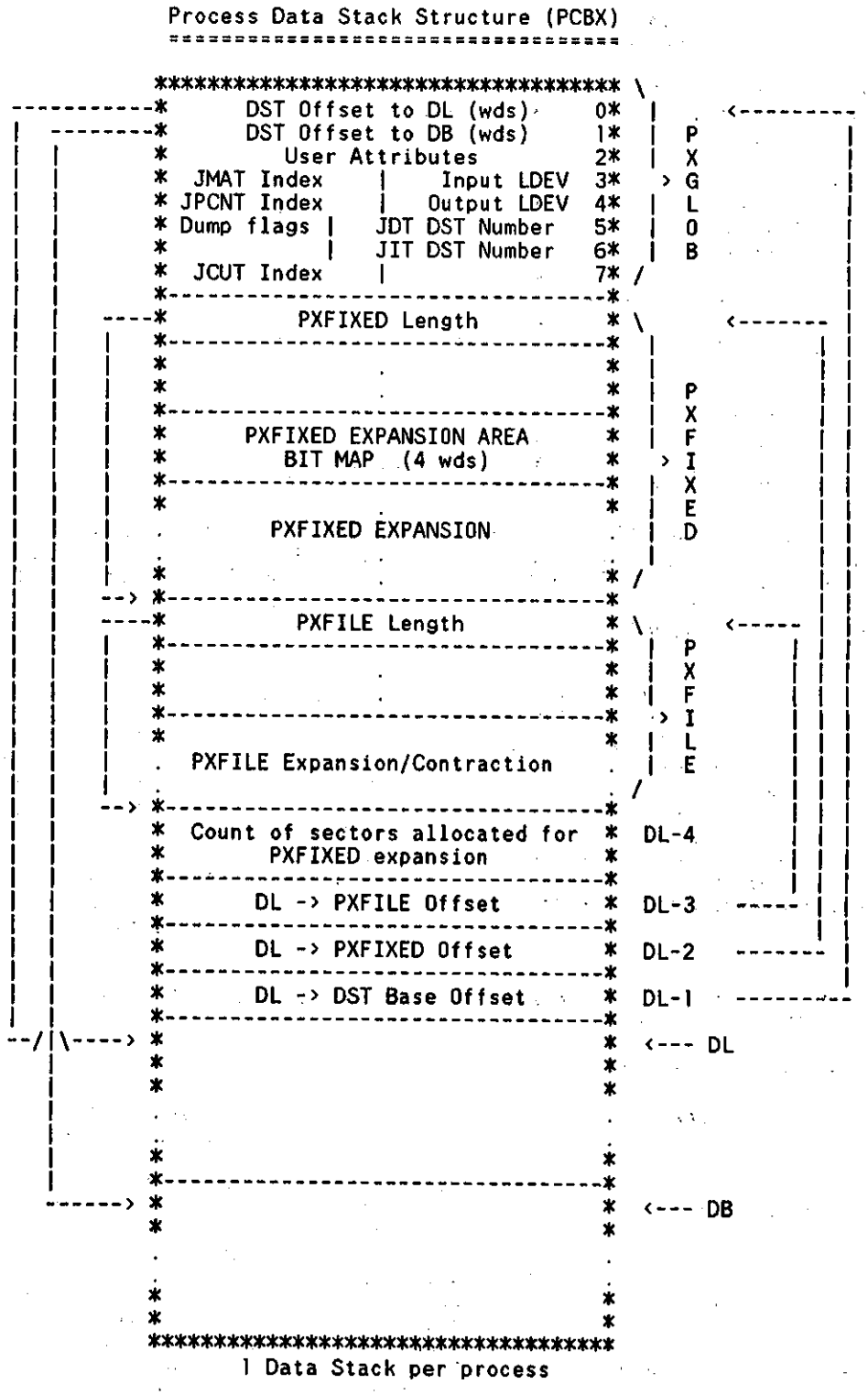
The interesting job tables, in our opinion, are the JMAT, JDT, and the JIT. Of these three, only the JMAT has the format of having one entry per job instead of one table per job. Since the JMAT has the job/session numbers in it, it is the prime starting place for any program looking for a particular session, or job; or a program which will print something for all logged on jobs and sessions. The MPE command :SHOWJOB simply scans the JMAT and reports it in a meaningful format.

An interesting exercise for the novice PM programmer is to write a program to print a SHOWJOB by reading the JMAT. SHOWJOB tells you the day of the week that a person logged on, but did OPERATOR.SYS log on last week or last month? It's in there, but MPE won't tell you.

As mentioned earlier, if you want to change anything in the PCBX area, you've got to do it on the stack of your main process. To do this, you need to know the DST number of your main process' stack. This number is stored in the PCB (Process Control Block) DST (data segment #3) with one entry for each process -- PROCESS not JOB! Remember, a job or session is a special type of process. The PCB entries are indexed by the PIN (Process Identification Number) of the process whose entry you are looking for. The PIN of your job/session main process can be found in your JMAT entry, your JDT, and your JIT. Take your pick. Figure P1 shows a listing of a procedure which will return your main process' PIN and the DST number of your main process' stack. This procedure GET'MAIN calls a procedure GET'DATA which uses the MFDS (Move From Data Segment) machine instruction to retrieve data from any data segment on the system. The comments in the two procedures make them virtually self-explanatory. Figure 1 shows the relationships between the job/process tables. For specifics, refer to the MPE System Tables manual.

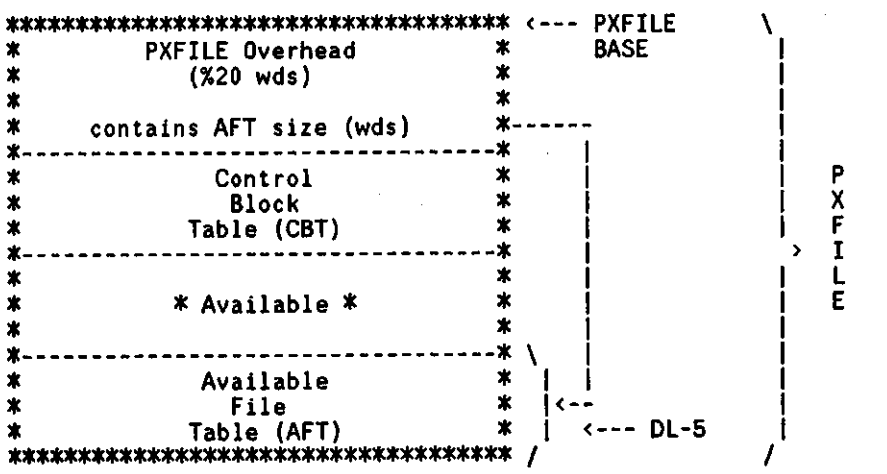

Job / Process Table Relationships
Process' Stack (PCBX)
* JMAT Index *
* JPCNT Index *
* JDT DST # *
* JIT DST # *
* JCUT Index *
* Job/session # *
* File System *
* Information *
1 per process
DST from PCB DST
SIR - see note 1.
-see also fig 2-
Job Master Table (JMAT)
* header *
* information *
* User entry *
* (main PIN) *
1 per System
DST=%31 SIR=%17
Process Control Block (PCB DST)
* header *
* information *
* Process entry *
* (stack DST #) *
1 per System
DST=%3 SIR=%42
JCUT offset = JCUT index * %3
Job Directory Table (JDT)
* Job/session # *
* main PIN *
* extra dsegs *
* :FILE eq'ns *
* JCW's *
* TEMP files *
* CS/DS lines *
1 per main process
DST in data stack
Job Information Table (JIT)
* Job/session # *
* main PIN *
* Account name *
* Group name *
* User name *
* Job/ses. name *
* Allow masks *
1 per main process
DST in data stack
Job Cut-Off Table (JCUT)
* header *
* information *
* process entry *
* CPU count & *
* maximum time *
1 per system
DST=%44 SIR=%16
Job Process Count (JPCNT)
* header info *
* Zero - 1 byte *
1 per system
DST=%30 SIR=%15
Notes: 1. Process job SIR = BASE (%50) + JPCNT index.
2. JCUT index only valid for CPU limited processes, zero otherwise.
3. JMAT offset = JMAT index * %32.
4. PCB DST offset = PIN * %20.

-- Figure #1 --

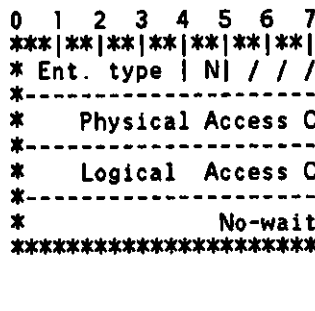

Process Data Stack Structure (PCBX)
DST Offset to DL (wds) 0
DST Offset to DB (wds) 1
User Attributes 2
JMAT Index
Input LDEV 3
JPCNT Index
Output LDEV 4
Dump flags
JDT DST Number 5
JIT DST Number 6
JCUT Index 7
PXFIXED Length
PXFIXED EXPANSION AREA
BIT MAP (4 wds)
PXFIXED EXPANSION
PXFILE Length
PXFILE Expansion/Contraction
Count of sectors allocated for PXFIXED expansion
DL-4
DL -> PXFILE Offset
DL-3
DL -> PXFIXED Offset
DL-2
DL -> DST Base Offset
DL-1
<--- DL
<--- DB
PXFIXED
PXFILE
1 Data Stack per process

-- Figure #2 --

File System Area of Process Data Stack (PXFILE)



The Available File Table (AFT) is indexed in reverse order by file number, i.e. the entry for file #1 is at DL-8 thru DL-5, file #2 is at DL-12 thru DL-9, etc. For unused file numbers, the entry is all zeros. The number of entries in the AFT (# of files) = AFT Size from PXFILE Overhead / 4. There is a 4 word AFT entry for each file of the process, formatted as follows:



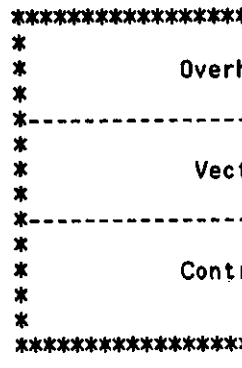
Entry Type: 0-File system
1-Remote file
2,3-DS files
4,5-CS files
6-KSAM files
8-MSG files

N: File is $NULL, no ACB vectors used.

LACB: Only for multi-access files.

PACB: Valid for all files except $NULL.

The PACB and LACB vectors, are indexes into tables known as Control Block Tables (CBT). The general structure of a CBT is:



The ACB vectors of the AFT entry consist of a six-bit vector table entry number & a Data Segment Table number containing the CBT. *All* pointer information for the CBT's are relative to the CBT beginning address. The vector table entry contains a pointer into the Control Block Area which is then the actual LACB, PACB or FCB. The CBT may be located in two places: 1) on a process' data stack, or 2) in a separate segment. Since all pointer addressing is relative to the CBT starting address, care must be taken if the CBT is located on a process' stack. (You can determine if a data segment is a process' stack by checking flags contained in the Data Segment Table (segment #2) entry for that segment). The same repetitive procedure is used when finding the LACB, PACB and FCB.

I/O TABLES

The input/output tables in MPE have a much more complex structure than the job tables. We don't recommend modification of these in any program. The important I/O Tables in our opinion are the LPDT -- Logical/Physical Device Table, DIT -- Device Information Table, ILT -- Interrupt Linkage Table, and the IOQ -- Input Output Queue. We use these tables solely for verification in our programs, to make sure we're communicating with the device we think we are. It's very possible to attempt to read or write beyond the physical limits of a disc if you don't know the type of disc you're using. Also, you wouldn't want to think you're writing to a terminal and actually be writing to a disc. For these reasons, we check device information in the I/O Tables. The System Tables Manual contains an entire section on I/O tables for the interested reader.

All I/O requests are processed through the IOQ Table. A procedure called ATTACHIO handles the interfacing to the IOQ. All our information on ATTACHIO has come from what we have read in contributed programs, so we know what works but not exactly how or why. A page toward the end of this paper explains parameters to ATTACHIO as we know them. Use this procedure with extreme caution.

FINDING OUT MORE ABOUT MPE

So you haven't had enough? You want to find out more? Here's how we continuously learn more about MPE internals.

1. Get yourself an up to date set of manuals. We specifically recommend the following:

A. HP3000 System Reference Manual (HP Part # 30000-90020) This manual is found at most HP3000 sites, but few people bother to read it. It is an excellent source of information on stack operation, code segments, data segments, and the I/O system. Skim the book and make notes (mental or otherwise) on what information is in the book and where to find it in case you need it.

B. HP3000 Machine Instruction Set (HP Part # 30000-90022) Another seldom-referenced manual found in most HP3000 shops, it is often hidden in the same binder as the Sysyem Reference Manual. We recommend that you become familiar with HP3000 machine language for reasons mentioned later. Perhaps the most valuable page in this manual is page A-1. Always look there first -- it's a real time saver.

C. HP3000 Intrinsics Manual (HP Part # 30000-90010) For anyone programming in SPL, this is a MUST. Pay particular attention to what the intrinsics DON'T do and you've got a nice privileged mode program to write.

D. Programming Language reference manual of your choice. As mentioned earlier, we recommend SPL.

E. MPE System Tables Manual -- Another MUST for the serious privileged mode programmer. We hang on to our old System Tables Manuals even though they are now outdated and often totally inaccurate. It's interesting to see the advances made in MPE over the years. Looking back also gives us a good idea as to what types of things are likely to change from one version of MPE to the next.

2. The second thing you can do to find out more about MPE is to learn to read HP3000 machine language. You don't need to have every mnemonic mnemorized, but you should get intimate with the Machine Instruction Set manual. The next best thing to knowing something is knowing where to find it. You'll be surprised how fast it comes to you.

3. Thirdly, use your newly acquired knowledge of machine language to set breakpoints in HP supplied code. How does SPOOK open a spool file? Set breakpoints, use DEBUG, step through the code and find out. It's easier than you think.

4. Equally enlightening is the use of so-called "program dumpers". SEGMENTER can tell you a lot about SL.PUB.SYS. Use it to find the location of "interesting" procedures. Numerous contributed programs will tell you what external procedures are used by a program. So, you find out that SPOOK calls a procedure called FSOPEN. Find it in SL.PUB.SYS using SEGMENTER then "decompile" it using DEBUG with the C (CODE) option or the

latest version of DECOMP which lets you look at SL procedures.

5. Look in the contributed library for PM programs. We try to check out all new ones on each tape we get. One rule of thumb: No source -- it gets purged. It's surprising how many HP employees with a knowledge of MPE internals and PM techniques will contribute programs (often anonymously). You can frequently get good ideas or find out how to call those "undocumented uncallables". You can always customize the program to meet your specific needs (now that you know how). One final word of caution: don't assume that all privileged mode contributions are clean. For that matter, don't assume that ANY PM contributions are clean. Treat each one with caution, and evaluate the source thoroughly before you even THINK of typing :RUN.

6. Finally, you've got to write privileged mode programs. You can't learn anything unless you do it. Be prepared for the worst and don't be afraid to try things. That's easy for us to say, we're not in charge of keeping your computer running.

Remember that HP doesn't like for you to use privileged mode (and we don't blame them). When you get system failures, you should remove PM capability from any account having it (except SYS) so you can be ABSOLUTELY sure that your PM program is not the culprit.

HOW TO MAKE YOUR SPL PROGRAM RUN IN PRIVILEGED MODE

If you are writing a program:

No special capabilities are needed to compile it. The USER (and thus the ACCOUNT) needs PM capability in order to PREP with CAP=PM

$CONTROL PRIVILEGED in your source causes a program to start in privileged mode when it is run, unless ;NOPRIV is specified on the :RUN command.

Without $CONTROL PRIVILEGED, you can still call the GETPRIVMODE intrinsic at any time, if and only if your program has been :PREPped with CAP=PM

SAVEd programs must reside on a group with PM capability.

If you're running $OLDPASS or a TEMP file, then the user must have PM capability. This prevents you from naming a temp file X.PUB.SYS and running it.

If you are writing a procedure:

Privileged procedures must reside in an SL on a privileged group.

The USER (and thus the account) needs PM capability to be able to -ADDSL a segment containing a privileged procedure in SEGMENTER

Using OPTION PRIVILEGED, you are running in privileged mode as soon as you call the procedure.

Using $CONTROL PRIVILEGED, you can use GETPRIVMODE to jump into privileged mode when you desire.

OPTION PRIVILEGED puts you into privileged mode whenever you call ANY procedure in the same segment as the procedure declared OPTION PRIVILEGED.

EXAMPLES USING PM DEBUG

As mentioned earlier, PM DEBUG is an excellent way to make small changes to a system table or to take a quick look at one. Here are some examples to illustrate.

To enter privileged DEBUG, make sure you are logged on to a user having PM capability. Then type the MPE command :DEBUG. DEBUG is not very user-friendly and it responds with a ? prompt.

```
:DEBUG

*DEBUG* PRIV.A30.14
?
```

To look at a system table, use the DDA (Display Data Segment) command. The DDA is followed by the data segment number you wish to examine, an optional offset into the data segment, optional number of words to display, and optional display mode. Everything entered into DEBUG and displayed by DEBUG is in octal unless otherwise specified. So, suppose we want to look at the number of seconds allowed for a user to log on. The System Tables Manual tells us that this number is at word %120 in the SYSGLOB (System Global Area). It also tells us that the SYSGLOB is DST #5. So, to display this value in decimal (base 10), you would type:

?DDA5+120,I

and DEBUG will print something like:

DA5+120 +00120

which tells us that all users have 120 seconds to log on after they establish connection with the computer. Shortening this time to a ridiculously low value is one way of keeping people off the system. Similarly, lengthening the amount of time may be desired if you have many new users who have a great deal of trouble typing their :HELLO command. You can change this value on the fly by typing:

MDA5+120,I

Which does the same thing as the DDA command, except that it prompts you for a new value. Remember that your input is octal unless you specify otherwise. To enter a decimal number, prefix your entry with a # sign. For example, to change the log on time to 4 minutes instead of two minutes:

```
You type:       ?MDA5+120,I
DEBUG says:     DA5+120   +00120   :=
```

At the := prompt, enter your new value. Four minutes is 240 seconds. Prefix it with a # symbol like this:

DA5+120 +00120 := #240

and it's changed. Presto. No cold start. You can use this technique to change many things without having to take down the system for a cold start. If you need to change the maximum extra data segment size or the maximum number of extra data segments per process, for example, these numbers are stored at locations DA5+111 and DA5+112 respectively.

For example, increase the maximum number of extra data segments per process to 32.

```
:DEBUG
*DEBUG* A30.14
?DDA5+112,I

DA5+112     +00004

?MDA5+112,I

DA5+112     +00004   :=#32

?R
```

Note that the R command is used to exit DEBUG. In this example, the previous maximum number was 4. This change takes effect immediately; there is no need for a cold load.

Suppose you want to check on a job or session to find out what command it is currently executing (i.e. How far along is your job?) Try the following:

1. Locate the main pin of the desired job or session using the MPE :SHOWQ command. The PIN will be preceded by the letter M and followed by the session number. For example:

C M14 #S639

tells you that session #639's main PIN is 14 (decimal).

2. Enter PM DEBUG and do the following steps. Find the size of a PCB entry. This is located at location 1 in the PCB DST, which is DST #3. The number on most recent versions of MPE is 16 (%20) but check to make sure.

```
:DEBUG
*DEBUG* PRIV.A30.14

?DDA3+1,I
DA3+1    +00016
```

3. Calculate the offset into the PCB DST for this process by multiplying the PIN times the entry size. In this case, our offset is equal to 16*24 which equals 224 (decimal). This number is the starting location of the PCB entry for the process' main PIN. We want to examine word #3 of the

process' PCB to get the DST number of its stack. We can do this manually, or let DEBUG handle the arithmetic as follows:

```
                    ?DDA3+#224+3
OR WE COULD ENTER   ?DDA3+#16*#14+3

                    DA3+343     005020
```

4. Manually extract bits 1:10 from this value to get the stack DST number. In this example, %005020.(1:10) = %120

5. Word number 1 in the stack DST gives us the offset to the stack's DB area.

```
?DDA120+1
DA120+1    000444
```

6. We want to look at the string located at DB+1, so by adding one to the DB offset, and dumping the area in ASCII, we can look at the last command executed by this main process.

```
?DDA120+444+1,30,A
```

The command will be displayed, and it may be incomplete or followed by garbage. We have no way of determining the length of the command except by inspection. The command ends with a carriage return, which is displayed in the ASCII dump as a period.

Granted, this procedure is somewhat complex, and you'd probably be better off writing a program to handle such a task. The example is only intended to show how easily information can be retrieved from system tables.

We have written other programs which do nice things in Privileged Mode. It's not hard to switch to another group without logging on again. This involves verifying that the group exists, looking up its pointers in the directory, and putting these pointers in the right places in the job tables. The name of the group you are logged onto must be changed in the job tables also, or the SHOWME command will be inaccurate. What the contributed programs don't do (for the most part) is update the directory with pertinent information. It may or may not be important to you if your GROUP CPU seconds and CONNECT time values are accurate. We think they should be updated when you switch to a new group. Another item in the directory that needs to be updated and is often ignored is the usage counter. Each account and group within the account has a counter that keeps track of the number of users currently logged on to the account. This prevents someone from purging a group or account while it is in use, leaving the user of it in limbo. With most contributed group and account jumping programs, these group usage counters get out of whack.

Another nice program is one which will allow you to do a "GOTO" while executing a UDC. MPE provides an IF condition, but no GOTO. The most popular application of this is when you have a MENU type environment. Your UDC runs a program (or any number of programs) then when a program ends, you want to start the entire command all over again. To exit the loop, set a JCW in your menu program (perhaps triggered by a function key) and test it using the MPE :IF command. Our program uses the PARM value as a relative index for branching within the UDC. For example, when run with PARM=-5, the program moves the pointer back 5 lines in the UDC. Similarly, PARM=3 skips execution of the next three lines in the UDC. This has proven to be a very powerful enhancement to UDCs.

The simple program operates as follows. It uses our procedure GET'MAIN to locate the DST number of the main process' stack. On the stack, DB+%1635 contains the record number in the UDC file of the next line in the UDC to be examined for execution. The program simply adjusts this value up or down depending on the value of the PARM. Note that because recursion is used when executing UDCs, this will work only on your outermost UDC. For example if your MENU command invokes another UDC, such as RUNP, don't expect the UDCLOOP program to work correctly in the RUNP command.

CONCLUSION

Privileged Mode is not for everybody. We have attempted to explain it here in detail, and to cover a few system tables to get the interested programmer started. We may have tried to cover too much material in this paper, but we want to provide as much information as possible to assist any present or future privileged program writer. We invite questions and/or comments, and we apologize in advance since we may not be able to reply to all of them. Thank you.

Here are some other userful (and sometimes necessary) MPE internal procedures, that we have come across in contributed software programs. These procedures are all found in the system SL, SL.PUB.SYS.

```
  I                IV
FLAG:= GETSIR (SIR'NUMBER);
```

This procedure is used to lock a SIR (system internal resource) to prevent data accessing problems when more than 1 user may be involved. The SIR must be 'unlocked' before termination of the program or a system failure will result. The value of 'FLAG' is passed to the procedure RELSIR.

```
             IV       IV
RELSIR (SIR'NUMBER, FLAG);
```

This procedure is used to unlock a SIR after it has been locked by GETSIR. FLAG is the value returned by the corresponding GETSIR. A locked SIR must be unlocked (released) before ending the program, or a system failure will result.

```
  L
FLAG:= SETCRITICAL;
```

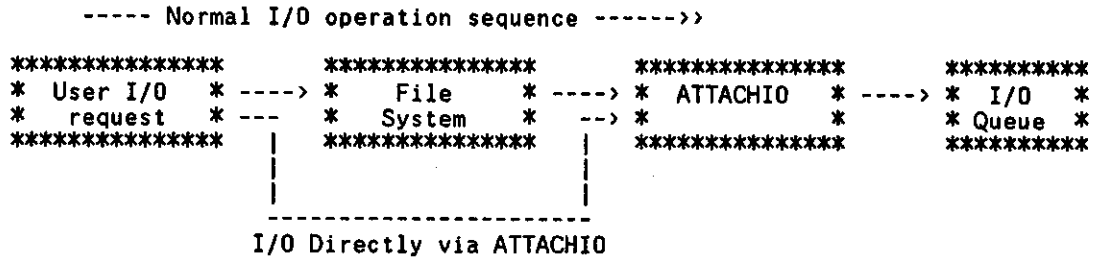
This procedure causes the system to only execute this process until RESETCRITICAL is called. If SETCRITICAL has been called, a RESETCRITICAL must be executed before terminating the program or a system failure will result.

```
                IV
RESETCRITICAL (FLAG);
```

This procedure reverses the critical status of a process after a SETCRITICAL. This must be called if a process has been made critical or a system failure will result. FLAG should be zero. Use of SETCRITICAL & RESETCRITICAL will insure (like the use of SIR's) that no other process will be accessing data you are attempting to modify.

ATTACHIO

Normally, all Input & Output is accomplished via calls to file system intrinsics. However, through the use of privileged mode, all the normal checks and limitations imposed upon the user by MPE may be by-passed. This is achieved by calling an MPE internal procedure called ATTACHIO. ATTACHIO simply formats parameters and queues requests to the device drivers.



The parameters to ATTACHIO are as follows:

```
              IV     IV     IV     IV     IV      IV     IV      IV      IV
  ATTACHIO (LDEV, QMISC, DSTX, ADDR, FUNC, COUNT, PARM1, PARM2, FLAGS)
```

Parameters

LDEV	Logical Device Number for I/O request.
QMISC	Request state & flags. Only observed to be zero.
DSTX	DST of target data area. Only observed to be zero.
ADDR	Address of data buffer to be written or read into.
FUNC	Function Specification. 0 - Read from specified LDEV. 1 - Write from data buffer to LDEV.

COUNT Transfer count. Number of words (or bytes if COUNT < 0) to be read or written.

PARM1 & 2 Zero for terminal I/O. Contains disc address for disc I/O.

FLAGS Observed to be:
1 - Disc I/O.
%401 - Terminal I/O.

ATTACHIO returns two words (double integer) after execution. The first word contains the actual transfer count, exactly like the count returned by FREAD, in the second word, bits 8:8, the general status is given:

0 - Awaiting completion.
1 - Successfully completed.
2 - EOF detected.
3 - Unusual Condition
4 - Irrecoverable error.

This procedure is probably the most powerful (and consequently most dangerous) procedure available to the privileged mode user. Use of this procedure should never be done casually, and every possible check should be performed. (Such as verifying LDEV information from internal device tables, ADDRESS information from device type & subtype tables, etc.).

```
LOGICAL PROCEDURE GET'DATA (DST'NUM, OFFSET, COUNT, ADDR);
VALUE DST'NUM, OFFSET, COUNT;
LOGICAL DST'NUM, OFFSET, COUNT;
LOGICAL ARRAY ADDR;
OPTION CHECK 3;

BEGIN

ENTRY PUT'DATA;

INTRINSIC GETPRIVMODE, GETUSERMODE;

LOGICAL ARRAY ENTRY'BUF (0:3);
LOGICAL WRITE'DATA;

WRITE'DATA:= FALSE;
GO TO LET'S'START;

PUT'DATA:       << ENTRY FOR WRITING TO DATA SEGS >>
WRITE'DATA:= TRUE;

LET'S'START:

GETUSERMODE;    << JUST TO BE SAFE >>

ADDR (0):= ADDR (0);    << MAKE SURE IT WILL FIT >>
ADDR (COUNT - 1):= ADDR (COUNT - 1);

GETPRIVMODE;    << LET'S SEE IF THERE IS SUCH A SEGMENT >>

TOS:= @ENTRY'BUF;
TOS:= 2;
TOS:= 0;
TOS:= 4;
ASSEMBLE (MFDS 4);  << GET DST ENTRY 0 >>

IF DST'NUM > ENTRY'BUF (0) THEN GO RETURN'FALSE; << BAD DST >>

TOS:= @ENTRY'BUF;
TOS:= 2;
TOS:= DST'NUM * 4;
TOS:= 4;
ASSEMBLE (MFDS 4);  << GET DST'S ENTRY >>
```

```
IF ENTRY'BUF (0).(3:13) = 0 THEN GO RETURN'FALSE;  << NOT USED >>
ENTRY'BUF (0):= ENTRY'BUF (0).(3:13) * 4;   << LENGTH OF SEGMENT >>

IF OFFSET + COUNT > ENTRY'BUF (0) THEN GO RETURN'FALSE; << BAD LEN >>

<< LOOKS OK, GET THE DATA >>

IF WRITE'DATA THEN BEGIN
  TOS:= DST'NUM;
  TOS:= OFFSET;
  TOS:= @ADDR;
  TOS:= COUNT;
  ASSEMBLE (MTDS 4);  END;   << WRITE THE DATA >>


ELSE BEGIN
  TOS:= @ADDR;
  TOS:= DST'NUM;
  TOS:= OFFSET;
  TOS:= COUNT;
  ASSEMBLE (MFDS 4); END      << GET THE REQUESTED DATA >>

GET'DATA:= TRUE;
GETUSERMODE;
RETURN;

RETURN'FALSE:

GET'DATA:= FALSE;
GETUSERMODE;
END;

LOGICAL PROCEDURE GET'MAIN (MAIN'PIN, MAIN'STACK'DST);

LOGICAL MAIN'PIN, MAIN'STACK'DST;
OPTION CHECK 3;

BEGIN

INTRINSIC GETUSERMODE, GETPRIVMODE;

LOGICAL POINTER PXGLOB, S'0=S-0;
LOGICAL JIT'DST, PCB'DST, PCB'ENTRY'SIZE, PCBPT;

PCB'DST:= 3;

GETPRIVMODE;

PUSH(DL);          << PUT CONTENTS OF DL ON STACK >>
TOS:= S'0(-1);     << PUT ON CONTENTS OF DL-1 >>
ASSEMBLE (SUB);    << THE DIFFERENCE IS PXGLOB BASE >>
@PXGLOB:= TOS;

JIT'DST:= PXGLOB(6).(6:10);  << GET THE JIT DST NUMBER >>

GETUSERMODE;

IF NOT GET'DATA(JIT'DST, 10, 1, MAIN'PIN)
     THEN GO RETURN'FALSE;
MAIN'PIN:= MAIN'PIN.(8:8);

IF NOT GET'DATA(PCB'DST, 1, 1, PCB'ENTRY'SIZE)
     THEN GO RETURN'FALSE;
PCBPT:= MAIN'PIN * PCB'ENTRY'SIZE;

IF NOT GET'DATA(PCB'DST, PCBPT+3, 1, MAIN'STACK'DST)
     THEN GO RETURN'FALSE;
MAIN'STACK'DST:= MAIN'STACK'DST.(1:10);
```

```
GET'MAIN := TRUE;
RETURN;

RETURN'FALSE:
GET'MAIN := FALSE;
END;
```

Joseph C. Felix is a Software Design Specialist for Educational Computer Systems, Inc. of Cincinnati, Ohio. He has a B.S. from the University of Cincinnati in Information Processing Systems. As one of the founders of Educational Computer Systems, Joe has been with the company since its inception in 1977. He has been writing SPL programs using privileged mode since 1978. His work with HP operating systems dates back to 1974 when he customized TSB, the HP2000 operating system, while employed by Cincinnati Public Schools. In his free time, Joe enjoys amateur radio, community theater, music and electronic tinkering.

Chris Hauck is a Software Design Specialist for Educational Computer Systems, Inc. of Cincinnati, Ohio. He has a B.S. degree in Computer Science from the University of Dayton. He has been using HP computer systems since 1976, beginning with the HP 2000. In 1979, at Educational Computer Systems, he began using SPL and privileged mode on the 3000 and has since written countless privileged mode programs and procedures.
