CAP=PM; Privileged Mode De-Mystified

Jason M. Goertz
System Specialist
Hewlett-Packard

Introduction

Many years have passed since the first HP3000 rolled out the doors at Hewlett-Packard. In that time, an increasingly sophisticated user and software supplier base has emerged. Much of this increased knowledge and sophistication is because of better Computer Science education, as well as the fact that there are more and more people who have more and more years working with the HP3000. Along with this experience has come an increasing use of one feature of the HP3000, that being Privileged Mode.

In spite of this growing sophistication, there is still a large number of people who do not understand what Privileged Mode is. Even the in-house Data Processing departments writing PM code internally and software vendors who are supplying applications using PM (as Privileged Mode will be called from here on) do not always fully understand its consequences and dangers. It is the purpose of this paper to present a description of PM and its implications, and to provide a technical document that describes, in one place, all (or as many as could be found) of the various commands, compiler options, and intrinsics dealing with PM. It is a fundamental part of human nature to be fearful of the unknown. This paper is an attempt to make the largely unknown world of Privileged Mode known, and thus lessen the fear of this potentially powerful tool.

Before starting, some disclaimers are in order. This paper is by no means intended to encourage anyone to use Privileged Mode. It is the opinion and experience of the author, and does not constitute an official statement on the part of Hewlett-Packard. It is intended strictly to be informative so that the reader could form his/her own opinions.

The approach which will be taken in describing the details will be to look at PM from the inside out. That is, examine first what PM is at the hardware level, then look outward to the software which sets and checks for PM.

What is Privileged Mode

At the lowest level, PM is a state in which a process can run, either temporarily or for the duration of the process's execution. While in this mode, the ability to execute a certain set of privileged instructions is given, as well as allowing certain non-privileged instructions to operate in a different fashion than normal, user mode. In addition, procedures can be defined which can only be called when the calling code is running in PM. It is that simple. That is the total definition of PM. No magic, no wires or mirrors, no potions or incantations. However, there are many, many ramifications of this simple definition. Indeed, exactly what these instructions and capabilities are, and how PM can be entered and exited is the whole point of this paper.

The ultimate enforcement of which capabilities will be granted is done by the microcode (ie, hardware) of the machine. To understand how this is possible, we must first understand a little about the hardware architecture of the HP3000.

The CPU of any HP3000 has several hardware registers that are used for various functions. Different CPU types (Series

30,44,64, III) have different registers, but some registers are common to all types. One of these registers is called the Status register, and this contains information regarding the state of the hardware and microcode at a given instant in time. Such things as what code segment is being executed, whether an overflow has occured, whether carry has occured, and other types of information are kept here, and are available to both the hardware and software via this register. Bit zero (highest order bit) of the 16 bits is called the Mode bit, and this is where the fact that the machine is in PM or not is kept. If the bit is on (a one), then the machine is running in PM AT THAT INSTANT IN TIME. The microcode can check this bit at any time to determine whether certain operations are valid. The most common check that is made is when certain instructions are executed which are deemed "privileged". The list is too numerous to mention here, but can be found by consulting the Machine Instruction Set Reference Manual, PN 30000-90022. In the description for each instruction, the various checks that are made are listed. For example, the LOAD instruction lists STOV and BNDV checks. These are, respectively, the STack OVerflow and BouNDs Violation checks. One of the checks which can be made is the MODE check, which is whether or not the Status register Bit 0 is 1. In general, all IO instructions, and any instruction which can reference memory outside of the users stack or code segments have a MODE check on them. A few of these are MFDS (Move From Data Segment), MTDS (Move To Data Segment), LSEA (Load Single from an Extended Address), HALT (Halts the hardware), and SIOP (Start IO Program on HP-IB machines). Rather the list the instructions, it is better to consult this manual on a specific instruction and check if PM is required. When a violation is detected, a trap is executed which produces the message PROGRAM ERROR #6: PRIVILEGED INSTRUCTION.

The other check made by the microcode is done by the PCAL (Procedure CALl) instruction. It is possible to define a procedure with the OPTION UNCALLABLE keywords. When this is done, the PCAL instruction makes sure that the MODE bit is set when calling this procedure. If it is not, a trap is executed which produces the message PROGRAM ERROR #17: STT UNCALLABLE.

It is important to note at this point that it is not the instructions themselves that are particularly dangerous. It is the use of them that can cause problems. Even then, it is usually only two things that cause problems. One is when data is MODIFIED outside of the user's domain. Very rarely does just LOOKING at data outside the stack or code segment cause problems. The other and perhaps more difficult problem to ensure does not happen is providing incorrect information to the machine instructions or uncallable procedures. Something as simple as giving a data segment number of zero to the MFDS instruction will cause a System Failure 16. Or worse, a non-zero DST number which is not currently used. A timing problem in the creation and release of a data segment can be disasterous. It is almost entirely this area which has given PM code a bad name. Even MPE (which runs ENTIRELY in PM) cannot escape this problem.

It must be pointed out that there is a great deal of data which can only be obtained via PM that is perfectly safe to access. An example is data in the PCBX area of the stack. This is an area below the DL register which contains all kinds of file control blocks, extra data segment information, and other things. Since the stack is always there when a process is running, it is very safe to access this data. However, since it is below DL, privileged instructions must be used to access this area. One routine which does this is the JOBINFO procedure in the Contributed Library. This procedure access the PCBX to obtain the Job/Session number. A new intrinsic in MPE-V, also named JOBINFO, will replace this contributed procedure. It is impor- tant to note, however, that the new JOBINFO will do essentially the same work as the contributed version. Which brings up a good point: There is really two kinds of Privileged Mode. They are: 1) The kind HP supports and, 2) the kind HP doesn't support. All this really means is that HP writes MPE and utilities in Privileged Mode and supports it. When someone else writes the code, HP doesn't support it.

Setting the Mode Bit

We have seen that essentially what constitues PM is the MODE bit is set in the STATUS register. But how does this bit get set? Asking this simple question opens a veritable can of worms. We will try to make a systematic analysis of all the various options available to perform this simple bit-twiddle.

At the lowest level, the MODE bit is set by only one machine instruction, that being the PCAL instruction. The decision whether to set or not set the MODE bit on the PCAL by the following rules: 1). If the CST or CSTX entry reflects the fact that this segment is Privileged (Bit 1 of word 0 of the entry), then set the MODE bit on. 2). If the segment being branched to is nonprivileged, but the

MODE bit is currently set, then keep the MODE bit set for the execution of the new segment.

Note that 2) above implies that CODE WRITTEN TO RUN IN USER MODE WILL RUN IN PRIVILEGED MODE. Typically, this will not pose a problem. However, it is a little known fact that ALMOST ALL BOUNDS CHECKING IS TURNED OFF IN PRIVILEGED MODE. An example is the SCAN statement in SPL. If the termination criteria are not found, then normally a bounds violation will occur. In PM, however, the scan will continue beyond the stack. Obviously, this can have disasterous implications if the code has not been well debugged.

The reason for leaving the machine running in PM can be understood if we examine the logic for the EXIT instruction. This instruction can set the MODE bit to zero, but will not set it to one. The logic choices are: 1). If the currently running segment is in PM and the segment being exited to is in user mode, then the MODE bit will be cleared on the exit, returning the machine to a user mode state. 2). If the machine is in user mode at the time of the EXIT and the segment being exited to is privileged, (as indicated in the Status Register saved in the Stack Marker at Q-1), then Privileged Instruction violation will occur. This is done to prevent a possible breach of security. It would be a simple matter for a user without any special capabilities to write a small SPL program to call a procedure (a PCAL) which set bit zero of Q-1 to 1 (the MODE bit). Upon EXITing, the program would be running in PM, and could look at any part of the system, including the directory to get MANAGER.SYS password, or any other part of memory desired. Because the EXIT instruction follows rule 2, above, we see how this possible loophole is closed. We also see why the PCAL instruction follows rule 2 in the previous paragraph. If the procedure should, indeed, be privileged, this is the simplest way to insure that the MODE bit is set correctly.

Continuing on our journey outward, it was mentioned that the CST or CSTX entry (logically the same) was checked for a PM bit. This bit gets set when the CST or CSTX entry is created. It is best to take each case, CST and CSTX, separately.

The CST table is a fixed length table (192 entries) which contains information regarding segments found in SL files only. MPE-V will change this around in detail, but for the purposes of this discussion, this description will suffice. Each entry is built by one of two software entities. On startup, INITIAL reads the system SL (SL.PUB.SYS) and creates a CST entry for every segment which is a SYSTEM, RESIDENT, or PRIVILEGED segment. Typically, MPE segments will always be SYSTEM and PRIVILEGED. Thus, each segment marked PRIVILEGED will have the PM bit set in the CST. After the system is up and running, and all MPE segments are loaded and CST entries created, the LOADER is the entity that adds any other SL segments to the CST. This is done when a program is loaded which references procedures in a segment in a group, public or the system SL. If that segment is also marked as being PRIVILEGED, then the PM bit is set for that segment also.

For segments that are resident in program files, the rules are the same as for user SL segments, above, except that table that gets built is the Code Segment Table Extension (CSTX) instead of the CST. Also, the information as to whether the segment is privileged or not is kept in record 0 of the program file in a table called the CST re-mapping array. Thus, when a :RUN command is entered by a user, the LOADER reads this table and knows whether or not to set the PM bit in the CSTX. It is important to note that the smallest entity which can be called "privileged" is a segment. This has some important ramifications. If one procedure in a segment is defined as privileged, then ALL PROCEDURES IN THE SEGMENT WILL BE PRIVILEGED. It is therefore a good idea to keep all privileged code together, and not mix the privileged procedures with ones that run strictly in user mode.

Continuing outward, we must ask how the program file or SL file is built. In each case, the answer is the same: the SEGMENTER. Whether used directly via the :SEGMENTER command, or indirectly via a :PREP command, the SEGMENTER is used to transform a USL file segment or segments into a finished, linked Program file or SL segment. It is at this stage that about half of the security implications of PM are realized. Obviously, this is a much higher level than that which the hardware imposes.

When the :PREP (or -ADDSL) command is entered, and the segment (or outer block, in the case of a program) is entered, the SEGMENTER checks two things. First, it checks to see if the proper capability is present. In the case of a program file, this implies that the user has entered CAP=PM on the :PREP command. The second check made, and one key to the implementation of MPE security regarding PM, is that the :PREP'ing user has PM capability assigned. If not, the :PREP (or -ADDSL) fails.

One exception must be noted at this point regarding the SEGMENTER's check of capabilities. In a program file, there are two different kind of segments. There is a special segment, called the OUTER BLOCK,

and there are all the other segments of the program. The Outer Block can be privileged in addition to the other segments. It is also possible for the other segments to be privileged and the Outer Block to not be, or vice versa. In any case, if the Outer Block is privileged, the :PREP will fail if CAP=PM is not specified. A minor point, but one worth mentioning.

The next level of capability checking is done when the program is actually loaded. In fact, the LOADER performs more checking than any other entity except perhaps the microcode. When the program is :RUN, the LOADER first checks to see if any segments are privileged. This is done by scanning the CST remapping array mentioned previously. If any segment is privileged, and the capability word stored in record 0 does not have the PM bit set on, then the load fails with "ILLEGAL CAPABILITY". The capability word is set by the SEGMENTER based upon the CAP= parm of the :PREP command. If none is specified, then IA,BA are assigned, but not if those capabilities do not exist at the group level.

Assuming that the capability word is in order, a check is made to be sure the capabilities of the program do not exceed that of the group. In other words, if IA,BA,PM are assigned to the program, all three must be present at the group level. (Also at the account level, but this not verified by the LOADER). It is important to note that the same checks are made for any privileged SL segments being loaded as a result of external calls made by the program. The only difference is that the LOADER differentiates between illegal capability of the program file and of the SL file in the error message. (Thank heavens for small favors!!!).

Compiler Options

We have seen so far what is necessary from the machine and operating system point of view to make a program privileged. Various areas were mentioned as having a PM bit set, such as the CST remapping array. However, we have not addressed how these bits magically get set. This section will deal with this subject. Since SPL is the only language that allows full access to PM code and machine instructions, this is the only compiler which will be shown.

As mentioned before, the smallest entity within the system that can be privileged is the segment. In SPL, the way a segment is given PM is by using the OPTION PRIVILEGED statement:

```
PROCEDURE EXAMPLE(A,B,C);
   VALUE A,B,C;
   INTEGER A,B,C;
   OPTION PRIVILEGED;
```

As mentioned before, IF ONE PROCEDURE IS DECLARED PRIVILEGED, THE WHOLE SEGMENT, AND ALL PROCEDURES WITHIN IT, ARE PRIVILEGED. It is therefore a good idea to place all privileged procedures together in one segment by using the $CONTROL SEG= compiler command. This applies equally to program file segments as well as segments that are compiled separately and placed in an SL.

To make a procedure UNCALLABLE, the following option is used:

```
PROCEDURE EXAMP;
  OPTION PRIVILEGED, UNCALLABLE;
```

This causes a bit in the Segment Transfer Table to be set. This table is resident at the end of every code segment, and is used by the PCAL instruction to branch to other segments. If this bit is on and current MODE bit is off, an STT VIOLATION occurs.

As previously mentioned, a program file has a special segment called an Outer Block. A special command is provided in SPL to make this segment privileged:

```
$CONTROL USLINIT,PRIVILEGED,MAIN=OB'
```

This option implies that the Outer Block is privileged from the start of the program, and remains privileged unless turned off. Since PCAL'ing a user mode procedure from a privileged one turns PM on, this would mean that the entire program will be privileged at all times. If this is not desired, then $CONTROL PRIVILEGED should be avoided.

:RUN Options, Intrinsics, and Things That Go BUMP in the Night

Several things need to be mentioned before any discussion of PM can be complete. First, besides the CAP = parm, there is another parameter of the :RUN command which deals with PM, and that is the NOPRIV option. NOPRIV is used by the LOADER to negate the effects of the PM bits in the prog file capability word and the CST Remapping Array. Thus, no part of the program will run in PM. However, if an SL segment is called, this

WILL run in PM. If this were not the case, then normal intrinsics, such as FOPEN, FREAD, etc, would not work. The implications are that if any PM code is executed, then a PRIVILEGED INSTRUCTION VIOLATION will occur. However, this is a good safeguard if development is being done and an added layer of security is necessary during testing.

There are several Intrinsics which behave differently when called from a PM segment. First, the file system will allow two things in PM. One is the ability to open Privileged files (files with a negative file code). This can only occur if the correct filecode is supplied along with being in PM. For filecodes equal or greater than zero, this is not necessary. The second ability granted by the file system is the ability to do nobuf, nowait IO to non-message files. (Message files can be accessed this way without PM).

The next group of procedures are the Data Segment Intrinsics. When called from user mode, this set of Intrinsics (GETDSEG, FREEDSEG, ALTDSEG, DMOVIN and DMOVOUT) checks for DS capability and returns a DST index, which is an index into a local table, not an MPE DST number. When called in PM, however, the check for DS capability is ignored, and the index returned is an actual MPE DST number. This implies that this number could then be used by MFDS, MTDS or MDS instructions later. Also, this must be done if SWITCHDB is to be called. SWITCHDB can only be called in privileged mode.

The GETPRIORITY Intrinsic has an option where a user can specify an absolute priority and place a process in a linear queue. Normally, the process will be placed in a circular queue. Obviously, this has far reaching ramifications, as a process in a high, linear queue could cause a lockout of other processes.

Probably the two most commonly used Intrinsics in the realm of PM are the GETPRIVMODE and GETUSERMODE instrinsics. These intrinsics allow a program :PREP'ed with CAP=PM to enter PM for a short time, then leave it. This is definitely the preferred method for performing privileged functions, especially if the amount of PM code needed is small and well contained. Typically, however, if the program is mostly privileged, it is probably better to just let it run in PM all the time. This allows the overhead of the calls to GETPRIVMODE and GETUSERMODE to be eliminated. Note also that using ;NOPRIV will cause these intrinsics to fail.

Deciding on PM

If you are a manager, you may still be uncertain whether or not to buy or use applications which run in Privileged Mode. The following are some guidelines.

If there is a real concern on your part, then discuss it with the vendor. Most vendors who use PM have not decided to do so lightly, as they realize that many people will have doubts and questions like you. Ask them if the PM was really necessary. Was it put in just to be a whiz bang, or is there some real benifit? Now that you've read this paper, you can put your knowledge to work. Is the PM just there to access some special Intrinsic capability, such as NOWAIT IO? If so, then it is probably very safe. Does the code look at MPE? If so, how secure is it? Is it looking at a part of MPE that can change drastically, or is it a part that has remained the same since the Series-CX? What is the reputation of the vendor? Are they noted for being MPE internals oriented, and knowing what is going on? Or did they pick someone off the street, teach him/her how to spell GETPRIVMODE and set them loose?

If you are really uncertain, sometimes you can get an independent consultant or HP involved. Hewlett-Packard SE's or System Specialists will sometimes be willing to look at an application and bless it "safe". Obviously, there is a large problem with liability. It will usually be up to the local management as to whether or not such a decision will be dealt out by one of their SE's. It is definately worth pursuing.

Finally, look at the vendor's support contract closely. If they guarantee that the code will work, then they must feel very good about it, and are confident that what they have written will not need much change, or at least that they will be able to change it if need be.

Conclusion

An attempt has been made to compile a compendium of options regarding Privileged Mode. In doing so, explanations were given so that it can be understood what is occuring during various privileged operations. It is hoped that these explanations would take some of the mystery of PM away, allowing a less fearful at- titude to form in the mind of the reader.

Name: Jason M. Goertz

Title: System Specialist Hewlett-Packard Neely Sales Region, Bellevue, WA. 98006

Jason started working with computers as a senior in high school. While attending Whitman College, he used computers in his course work. During his senior year, Whitman purchased an HP3000 Series II, which served as both an academic and administrative machine. Upon graduation, Jason stayed at Whitman as a Programmer/Data Base administrator, and later as system programmer, where he wrote a job scheduler, BEACON, and a security system, GUARDIAN, both of which employ Privileged Mode extensively. In addition, during this time, he worked on the HPIUG Contributed Library, and wrote or helped write many of the Infobase utilities for LIBS 4,5, 6 and 7. In 1980, Jason started working for the Bellevue HP office as a System Engineer, and within a year had added the role of Field Software Coordinator to his duties. He quickly became the area resource on MPE and internals, and has taught 4 MPE in- ternals classes, both to customers and to other HP employees. In August of 1983, he was promoted to System Specialist, where he concentrates on internals, special projects, and teaching of other SE's. Jason enjoys photography, backpacking and swimming for recreation, as well as travel and just relaxing. Jason's family consists of his wife, Doris, also involved in the HP com- munity, and his black cat, Nerd. All three reside in Redmond, Washington, north of Bellevue.
