HDY

LNG

North American Response Centers

HP 3000 APPLICATION NOTE #19

STACK OVERFLOWS:

Causes and Cures For COBOL II Programs

HEWLETT PACKARD

December 1, 1986
Document P/N 5958-5824/2649

HP 3000 APPLICATION NOTES *are published by the North American Response Centers twice a month and distributed with the Software Status Bulletin. These notes address topics, where the volume of calls received at the Centers indicates a need for addition to or consolidation of information available through HP support services. You may obtain previous notes (single copies only, please) by returning the attached Reader Comment Sheet listing their numbers.*

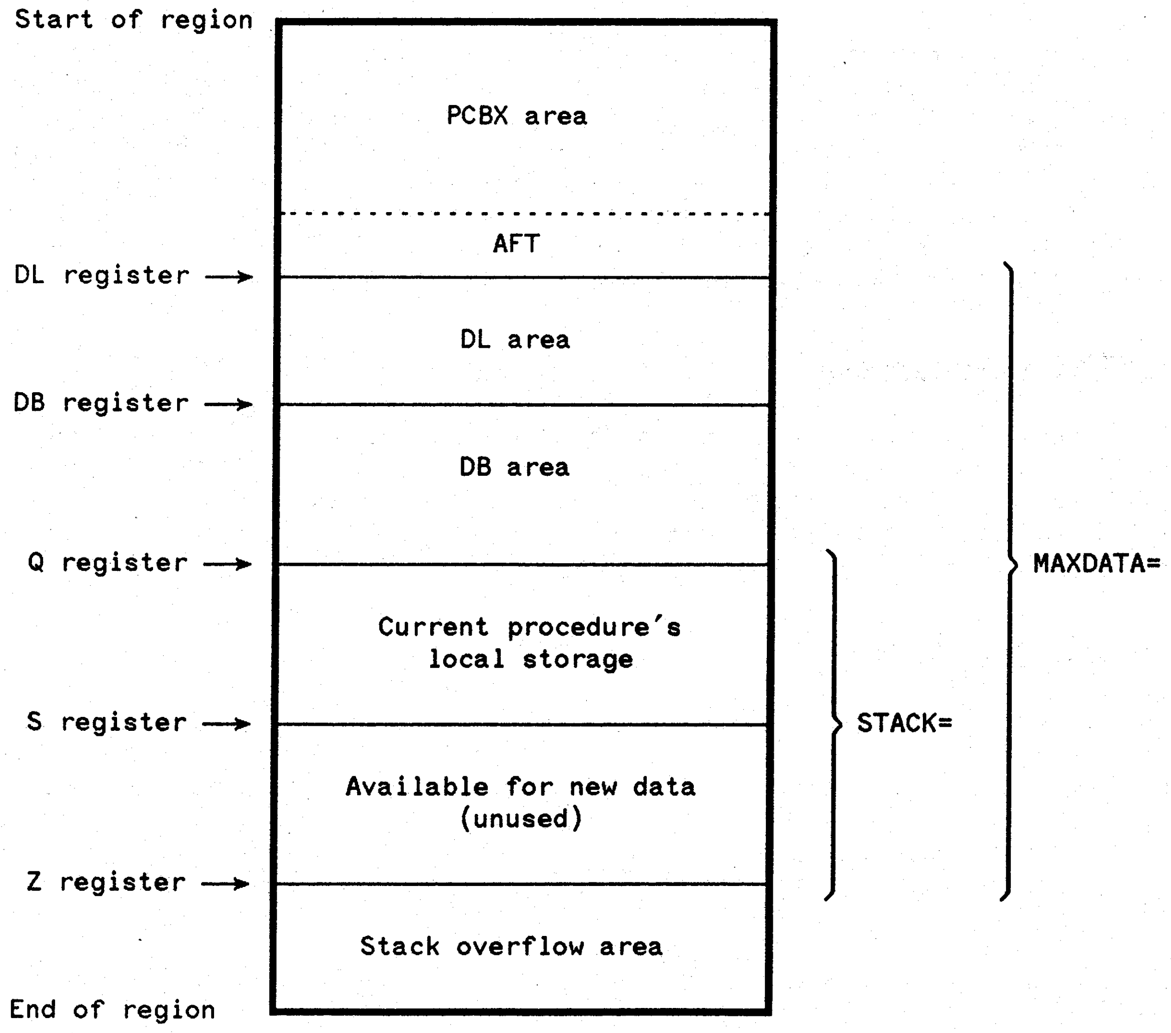
Note #	*Published*	*Topic*
1	*2/21/85*	*HP 3000 Printer Configuration Guide (superseded by note #4)*
2	*10/15/85*	*Terminal Types for HP 3000 HPIB Computers (superseded by note #13)*
3	*4/01/86*	*HP 3000 Plotter Configuration Guide*
4	*4/15/86*	*HP 3000 Printer Configuration Guide - Revised*
5	*5/01/86*	*MPE System Logfile Record Formats*
6	*5/15/86*	*HP 3000 Stack Operation*
7	*6/01/86*	*Cobol II/3000 Programs: Tracing Illegal Data*
8	*6/15/86*	*KSAM Topics: Cobol's Index I/O; File Data Integrity*
9	*7/01/86*	*Port Failures, Terminal Hangs, TERMDSM*
10	*7/15/86*	*Serial Printers - Configuration, Cabling, Muxes*
11	*8/01/86*	*System Configuration or System Table Related Errors*
12	*8/15/86*	*Pascal/3000 - Using Dynamic Variables*
13	*9/01/86*	*Terminal Types for HP 3000 HPIB Computers - Revised*
14	*9/15/86*	*Laser Printers - A Software and Hardware Overview*
15	*10/01/86*	*FORTRAN Language Considerations - A Guide to Common Problems*
16	*10/15/86*	*IMAGE: Updating to TurboImage & Improving Data Base Loads*
17	*11/01/86*	*Optimizing VPLUS Utilization*
18	*11/15/86*	*The Case of the Suspect Track for 792X Disc Drives*

STACK OVERFLOWS:
Causes & Cures for COBOL II Programs

This Application Note discusses Stack Overflows and suggests how you can prevent them. Examples and references are oriented towards the COBOL II/3000 programmer, however, much of the information can be also be applied to the other languages on the HP 3000. For a more detailed discription of how the stack operates please refer to *HP 3000 Application Note #6: HP 3000 Stack Operation.*

What is a Stack Overflow?

A stack overflow is an error condition that almost every HP 3000 programmer encounters sooner or later. As its name suggests, stack overflows occur when a process' data stack attempts to 'overflow' its boundaries. This happens when the S register, indicating the Top of Stack (TOS), moves beyond the current end of the stack, pointed to by the Z register. MPE will attempt to increase Z, but can only do so when the DL to (new) Z space is less than or equal to MAXDATA (see below). Otherwise, the process is aborted with a STACK OVERFLOW message.



Current Process' Stack

To MPE, a process' stack is simply a portion of a data segment. It, like any other data segment, is limited to a maximum size of 32767 words. Unfortunately, not all of this space is actually available to the user process. MPE itself uses the area below DL, known as the Process Control Block Extension (PCBX), to hold process specific information not contained in other system-wide tables. The Available File Table (AFT), containing entries for the process' open files, also resides in the PCBX. With another 128 words beyond the Z register reserved for the processing of error conditions (such as stack overflows), the amount of space available to the user process is actually only 31232 words. (This number could be reduced further if a large number of files are opened creating a large number of AFT entries).

In general, the key to avoiding stack overflows is to manage the stack as efficiently as possible. Usually, this means using as little of it as you can at any given time.

The MAXDATA Parameter

The most common method of eliminating stack overflows is to use the MAXDATA parameter at RUN and/or at PREP time. It specifies the maximum size to which the stack (the DL to Z area) can grow. Although MPE does NOT reserve this amount of space in memory when the process is loaded, it does reserve space in Virtual Memory (VM) to hold the stack when it is swapped out of memory. In fact, the only way for a stack to expand past its initial limits is by being swapped out to VM and back.

It is usually best to specify the smallest MAXDATA that will work without aborting your program. Although specifying a larger MAXDATA may solve your STACK OVERFLOW problem, if MAXDATA is larger than the stack will ever grow, VM disc space will be wasted; and because MAXDATA specifies the maximum space from DL to Z, a very large MAXDATA will restrict the size to which the PCBX can expand. This could cause a process which opens many files to abort with a "`No room left in stack segment for another file entry (FSERR 74)`" error indicating no more AFT entries can be created. Therefore, it is best to use the smallest MAXDATA possible. The easiest way to determine this is to experiment and use the smallest value that works.

The STACK Parameter

The STACK parameter is frequently confused with MAXDATA but has a significantly different function. Whereas MAXDATA specifies the maximum size a stack (from DL to Z) can attain, the STACK parameter specifies the amount of stack space you wish the process to start with when it is first loaded into memory. This overrides the initial space allocation computed by the SEGMENTER in the PREP command and, if the space is not needed, it will be wasted.

The NOCB Parameter

If MAXDATA does not solve your overflow problem, the NOCB parameter might. NOCB stands for No Control Block and tells the MPE file system to move its control blocks from the PCBX area of the stack to an extra data segment (XDS). This is done regardless of whether or not space is available in the PCBX. Although using NOCB causes the process to incur slightly more CPU and I/O processing overhead, it sometimes is the only alternative. To run your process NOCB only requires you specify it at RUN and/or PREP time, MPE takes care of the managing the extra data segment holding the file control blocks.

The VPLUS 'FAST' Forms File Option

VPLUS uses the DL to DB area of the stack for its "COMAREA EXTENSION". This COMAREA EXTENSION holds the largest save field, input and data buffers for the forms file, the largest form definition, function key labels, a save field table, a field status table, a local forms storage directory, a "protect string" (to indicate protected fields), and the forms file directory. The size of the directory, built by the VPLUS forms file compiler, varies depending upon whether the forms file is a regular forms file or a "fast" forms file. A regular forms file contains "source" records, "intermediate" records, and "code" records. A fast forms file contains only "code" records making it smaller than a regular forms file. Since its directory, which resides in the COMAREA EXTENSION, is also smaller, a "fast" forms file uses 800 words less stack space than a regular forms file. The FORMSPEC utility can be used to create your "fast" forms file.

SORT

When a procedure is called, space is allocated on the stack for its local data items (requiring the stack to expand if there is not enough room). This is also true when a SORT is performed. When the sort is invoked by the SORT verb, all stack space needed for communication with the sort subsystem is allocated. This space is not released until the sort is over. Therefore, if COBOL's INPUT PROCEDURE and OUTPUT PROCEDURE clauses are used, the stack space for any procedures *they* call (IMAGE, VPLUS, user-written, etc) will be allocated *in addition to* that space already being used for/by the sort routines. If your process is nearing the limit of its stack size, an overflow can occur.

As shown in the following example, using the INPUT PROCEDURE and OUTPUT PROCEDURE clauses, any procedure call within INPUT-PROC or OUTPUT-PROC would expand the stack beyond the size already used by SORT.

```
SORT SORT-FILE ASCENDING/DESCENDING KEY IS SORT-KEY
       INPUT PROCEDURE IS INPUT-PROC
       OUTPUT PROCEDURE IS OUTPUT-PROC.
```

The following code in which the USING/GIVING clauses are used will accomplish the same result. In this case, however, INPUT-PROC writes each record to TEMP-FILE instead of RELEASE-ing them to SORT-FILE. Likewise, OUTPUT-PROC reads the sorted records from TEMP-FILE instead of RETURN-ing them from SORT-FILE. Using this technique, procedure calls in INPUT-PROC and OUTPUT-PROC do not consume stack space in addition to that used by SORT. Note, however, that in saving stack space, you are increasing (maybe substantially) execution time of the process.

```
PERFORM INPUT-PROC.
    SORT SORT-FILE ASCENDING/DESCENDING KEY IS SORT-KEY
         USING TEMP-FILE
         GIVING TEMP-FILE.
PERFORM OUTPUT-PROC.
```

Subprograms and $CONTROL DYNAMIC

As mentioned above, each call to a procedure or subprogram causes space to be allocated on the stack, which may cause the stack to be expanded. Obviously, a program which has many nested calls to subprograms can use a lot of stack space and may cause a stack overflow, unless each subprogram releases the space it uses after it finishes executing.

This can be achieved by compiling subprograms with $CONTROL DYNAMIC. This causes stack space to be allocated on an as-needed basis in the dynamic area (Q to Z area) for the procedure each time the call is made. When execution finishes, the stack space used is returned and is now available for use by other subprogram's.

Compiling subprograms with $CONTROL SUBPROGRAM (the default) causes data items to be allocated in the DB area (global area) of the stack; the space is not released when execution finishes. Thus, subsequent subprogram calls to other procedures cannot use any of this space and additional stack space must be obtained.

The only reason to use SUBPROGRAM rather than DYNAMIC is if you must retain subprogram data storage from one call to the next. SUBPROGRAM ensures that its variables will remain intact for the next call to another subprogram by not releasing stack space after the subprogram is executed.

Another $CONTROL option for subprograms is ANSISUB. This allows for space to be obtained and released like the DYNAMIC option. However, between calls to the subprogram variables are retained in a disc file (i.e. when the subprogram is exited the variables are written to disc). When the subprogram is called again, these variables are restored from the file. The additional I/O involved with this option makes it the most inefficient option discussed and it should only be used when you must have stack space for subsequent calls and must retain variables between calls.

Program Design Considerations

Common sense tells us that if we use huge arrays or other large storage areas we increase the probability of a stack overflow. In such cases, it is very important to design your program's storage as efficiently as possible. For instance, when a large table is needed for data validation, consider the use of a KSAM file to contain this information. Extra Data Segments (XDS) can also be used to hold portions of a larger table. With the DMOVIN and DMOVOUT intrinsics you can dynamically swap these portions in and out of memory, allowing a smaller stack resident table to be used.

If the preceeding suggestions do not eliminate the stack overflow, you may need to reconsider the design of your application. One of the ways to reduce the needed stack space is to divide the program into two or more subprograms, with each subprogram having access to only the data that it absolutely needs to accomplish its task.