Help! Stack Overflow!
Alternatives for reducing stack size on MPE V/E HP 3000s

Lisa Burns Hartman
Hewlett-Packard
Corporate Offices
P. O. Box 10301
Palo Alto, CA 94303

STACK OVERFLOW! Those two words cause even the most experienced HP3000 programmers to groan and shake their heads. Writing a large application and making it work within the HP3000's stack limit of 32K words can be a real challenge. In this article I will suggest some techniques for handling large amounts of data within the constraints of the 3000's stack architecture.

What can cause a Stack overflow condition?

When a program aborts with a STACK OVERFLOW message, it means that the 32KW limit for stack size has been exceeded. If you are running a brand new program and you encounter this message, it means that you have simply placed too many variables on your stack at once. For a COBOL program, this means that your Working Storage is too large for the HP3000 to handle.

But what if you are maintaining an existing application which has run successfully for some time and is now getting a STACK OVERFLOW message? There are several possible reasons this might suddenly occur. Opening additional databases or files will increase your stack size. Each additional open adds between four and 16 bytes to this area. Another programming change which may increase stack size would be a VPLUS form change adding additional fields. VPLUS itself will use additional buffer space for each added field, in addition to the necessary additional room within your program's screen buffer.

Perhaps the most likely reason for an increase in stack size, however, is a new call to an additional subroutine. The linkage area necessary to communicate with the subprogram will add permanent space to the calling program. The local variables for the subprogram will add to the stack size as well. If the subprogram is static, this additional stack space will stay on the stack for the life of the process. If the subprogram is dynamic, local variables will pop off the stack as soon as control is returned to the calling program.

But, you protest, I haven't touched the program! All I did was bring up a new MIT, and now the application blows up. If your application was running very close to the stack limit on one MIT, and you upgrade your machine to the new MIT, this too can produce a stack overflow. MPE intrinsic calls (file intrinsics, IMAGE calls, etc.) are frequently changed for a new MIT. Just as for your own programs, their data usage may go up as well. For example, an FOPEN on UMIT takes more space than it did on T-MIT. Since all application programs call system intrinsics, changing a MIT can produce stack overflows in programs which ran successfully on the previous MIT.

Stack overflow: Quick fixes

Now that you have discovered the stack overflow, you have several options, some easy to implement, some harder. The first thing to try is running your program with the MAXDATA parameter.

```
:RUN MYPROG;MAXDATA=20000
```

The effect of this parameter is to increase the default stack size for your program. The default stack size is set by the compiler and may be inadequate for your program. It will certainly be inadequate for your program if you are using VPLUS screens and intrinsics. Figure 1 shows a diagram of the HP3000 stack structure. VPLUS intrinsics place data in the DL to DB area of your stack. Compiler defaults for the DL to DB area do not allow for this needed additional space. The MAXDATA parameter is therefore required to expand this area for VPLUS applications. Note that the STACK parameter will not expand the DL to DB area and so will not alleviate stack problems for VPLUS users.

The maximum value for the MAXDATA= parameter is 32000. The real word count of this stack area is 31232, but MPE is nice enough to take 32000 and not make you remember that. Setting the value to the maximum will not cause harm, since the stack space will be allocated as needed in 1K increments. Thus, you may wish to go ahead and set MAXDATA to 32000. If you would like to be more conservative, however, and have an idea of how much stack size you really need for your application, experiment with values until you find a minimum value which still avoids the stack overflow. You can do this by beginning with MAXDATA=32000 and working down by 5K increments, MAXDATA=31500, and so on, until you encounter an overflow.

If you are fortunate enough to find a value of 32000 or less which removes the stack overflow, congratulations! I recommend that you re-PREP your program with this same MAXDATA option before releasing it to your users. This will set the stack size for your object code to the MAXDATA value permanently, so that if your user forgets to RUN your program with the MAXDATA option, you will still be safe. Update your PREP job to include this option, so that you will also be safe for future program updates.

But what if you already have MAXDATA set at 32K, and you are still getting an overflow? The second quick fix is to execute your program with the NOCB (no control block) option:

```
:RUN MYPROG;NOCB
```

The PCBX, or Process Control Block eXtention, is used by MPE to manage the files and file equations used by your program (see figure 2a). This area can be very large if your program opens multiple files. The effect of the NOCB option is to move the PCBX area of your stack out to an extra data segment, freeing up more space for your application program's data area (see figure 2b). Be aware, however, that this will mean another data segment which must be CPU resident in order to execute your program. If you are running on a machine which is memory-bound, running your program with the NOCB option may increase memory thrashing and degrade your application's performance. Also, another data segment means another DST (data segment table) entry. For MPE IV and earlier MIT's, the DST is limited to 192 entries. For machines running these MIT's, the NOCB option may cause the error "OUT OF DST ENTRIES". MPE V machines will not be affected by an additional DST entry, since the DST on these machines can be configured for up to 2048 entries. So, for most newer machines with adequate memory running MPE V, the NOCB option poses no threat and may be a quick solution to a sudden stack overflow.

Harder fixes

Suppose that you have tried the above two methods and are still encountering an overflow. You're going to have to work a little harder to solve your problem. The next thing to do is to look at your subprograms. Static subprograms, $CONTROL SUBPROGRAM in COBOL, place data on the stack for the life of your process (see figure 3). This can be a problem, especially if their data areas are large. Dynamic subprograms, $CONTROL DYNAMIC in COBOL, free up stack space once control is returned to the calling program (see figure 4).

There are some things to be careful of when using dynamic subprograms. Remember that since local data areas disappear once a subprogram is exited, care must be taken that any permanent data be passed back to the calling program. Also, initialization routines will need to be executed each time a call is made to the dynamic subprogram.

Subprograms can have another effect on stack space. What if your main program, PROG A, calls a subprogram, PROG B, which then makes its own call to a third program, PROG C? What effect will nesting these subprograms have on your process's stack? Figure 5 illustrates the effect of this nesting. At the point it time when PROG C is being executed, the data areas of all three subprograms will be resident on the stack. This is true whether or not PROG B and PROG C are dynamic. Deep nesting can thus greatly affect stack space. This is another area to examine when checking your subprograms.

Still no luck, huh? You changed your subprograms to dynamic, you eliminated excessive nesting of subprograms, and you are still aborting with a stack overflow. It's time to look at your data areas themselves. Do you duplicate data in several areas in your global data areas (working storage and linkage in COBOL)? Are you passing more data than is necessary to called subprograms? Structured analysis and design techniques can help you identify necessary data flows. Taking the time to examine what exactly is needed by a called module instead of simply passing that variable called "01 Kitchen-sink-data-area" can help reduce stack space.

Hardest fixes for big problems

Some applications are just plain big and need just plain big data areas. If you suspect that this is your situation, you are going to have to work still harder. The first thing to consider is storing needed data outside of your stack. This can be done using extra data segments or using MPE temporary files. In either case, the program which needs access to the externally stored data will have to work harder than if the data were available directly.

An extra data segment (XDS) is an unstructured block of memory associated with your process. It can be used for large data areas, like a report page which is being formatted all at once, or for a table structure which occurs repeatedly. Programs accessing XDS must have the special capability PH enabled. Programming with XDS requires XDS intrinsics, GETDSEG, FREEDSEG, DMOVIN and DMOVOUT. The data segment is created with GETDSEG, and then loaded with data via the DMOVOUT intrinsic. Once loaded, the data in the XDS cannot be accessed directly. To access it, the programmer must bring the data into the stack using the DMOVIN intrinsic, and then manipulate it within the stack (see Figure 6). Since the DMOVIN and DMOVOUT intrinsics work with byte addresses, the programmer must keep an accurate count of where data is located within the XDS. Finally, the programmer should destroy the XDS with the FREEDSEG command. This will avoid problems with creating a data segment which already exists if the program is run a second time from the same session.

The advantages of XDS use is its speed. Since externally stored data is memory-resident, access is very quick. There are some disadvantages, however, and one is the programming complexity mentioned above. An additional wrinkle is that if your data area will not fit in one XDS, which has a maximum size of 32KW, you may need to work with several. This will further complicate your programming. Also, since the use of an XDS means an additional data segment for every user of the application program, this technique has the same problems as the NOCB option. But for MPE V machines with adequate memory, the use of XDS is a good solution.

An alternative to XDS use is the use of MPE temporary files for external data storage. Temporary files have advantage that programming is very simple. Programmers are typically familiar with file intrinsics and are comfortable using them. Like XDS, temporary files can be used for large data areas or for tables. Using the command intrinsic and the BUILD command, the temporary file can be built within the program. Simple reads and writes are used to access the data. The temporary file will be destroyed when the process quits, or it can be destroyed with the PURGE command within the program before the process terminates.

Temporary files have the additional advantage that they are not limited to 32KW and can be expanded as needed. And with disc caching enabled, performance is comparable to XDS use. This is because the cache domain for the temporary file will be memory resident, and reads and writes will be done through memory transfer and will be very fast. However, if you cannot guarantee that your program will run on a machine with caching enabled, MPE file access will significantly affect your program's performance, since accessing the data stored in the file will mean waiting for disc I/O. In this case, you should stick with XDS.

A final technique for solving stack overflow problems is Process Handling. Process Handling (PH) capability allows a process to RUN another program by creating a child process. The child process has its own stack and is independent of its parent (see Figure 7). PH can be used to treat a standalone subroutine, a print program for example, as a separate process. Like XDS, PH has its own set of intrinsics. The parent (calling) program uses the CREATE intrinsic to set up the child process, and then initiates its execution with the ACTIVATE intrinsic. At this point the parent program may use the SUSPEND intrinsic to stop its own processing until the child's function is complete. When the child has completed its task, it wakes up the parent with the ACTIVATE intrinsic, and then SUSPENDs itself. Finally, when the child will not be called again, the parent process destroys the child using the TERMINATE intrinsic.

PH has the advantage that the new child process gets its own stack -- another 32KW of space. There are some significant disadvantages to PH, however, the worst of which is probably the programming complexity described above. Care must be taken to synchronize the parent and child as they ACTIVATE and SUSPEND each other. If this logic is incorrect, data may be lost, or worse, both processes may be SUSPENDed at once, so that nothing will happen at all! It should also be noted that since there is noticeable performance overhead on the first call to the child, the child process should not be TERMINATEd until it is clear that no further calls are needed.

Programming with PH is further complicated by the fact that since the child process is independent of its parent, it must perform its own database opens and file opens, even if the parent has already opened these files. It will then have its own file pointers and database control blocks. And since it cannot make use of linkage areas or passed variables like a subprogram can, the child must use inter-process communication techniques such as XDS or Job Control Words (JCW's) in order to pass data back to the parent.

A final caution about PH is that it will double the number of processes running for a given application. This must be taken into account as far as DST use and PCB entries are concerned, especially on MPE IV and earlier MIT's. Database applications must also remember that there will now be twice as many processes accessing a given database, which may affect IMAGE logging and database locking strategies.

How to avoid stack overflows in the future

The best way to avoid being surprised by stack overflow is to know how much stack you are using in all areas of your application. The process display within the process context of OPT.PUB.SYS can show you the approximate stack usage for a given program. By running the program on one terminal and monitoring the stack use on another, you can see stack size change as you perform different functions within your program.

Figure 8 shows an example of the process display. The process shown has at one point used 27648 words of stack (SIZE). This is its "high-water" mark, the largest the stack has been. Its PCBX area is 1329 words (SYSOV). Its VPLUS area is 7640 words (DL-DB). Its main program has a data area of 10967 (DB-QI), and it has one subprogram with a data area of 3924 words (QI-Q). To determine how large its stack is at present, we need to subtract 3779 (S-Z), the space between the current stack pointer and the high-water mark, from the high-water mark, 27648 (SIZE). Thus, the current stack usage for this program is 23869.

An alternative to OPT is available for programs which do not call VPLUS intrinsics. For these programs, adding the data areas your main program and subprograms will give you a close estimate of stack space. This data area is shown in octal at the end of each compile listing.

Keep in mind, however, that this total will not reflect the PCBX portion of your stack.

Once you have established your current stack size, be aware of how programming changes and enhancements will affect stack use. Increases in linkage areas and global areas will add space, as will new called modules and additional file opens. These increases must be considered as you design these changes.

If you are dangerously close to the magic maximum value of 32636 words (the PCBX area plus the MAXDATA area) take steps NOW to overcome the problem, not when you blow up! Consider using XDS or temporary files. Rewrite called subprograms as process handled programs. Break up one program into several if possible. Buy yourself some room for growth. You will sleep better at night if you know that your maximum stack use is 22000, not 31999 words.

Of course, the real answer to stack space is to port to Spectrum, where the 32K word limit will be a thing of the past. I can't wait!

HP3000 Data Stack

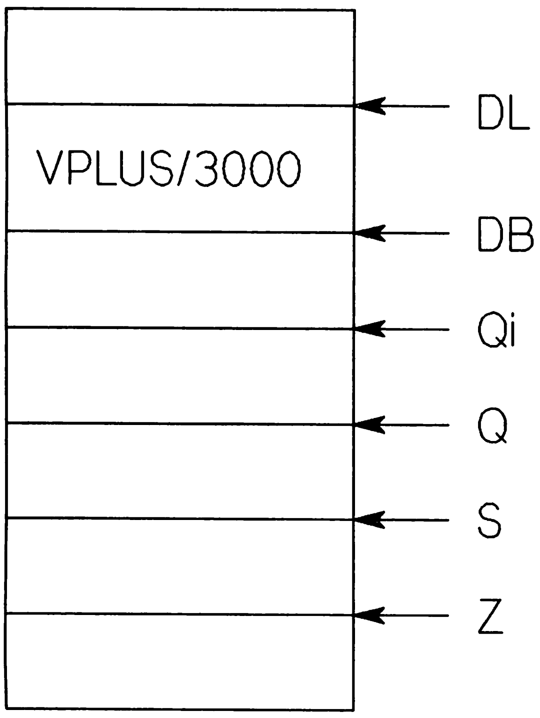


Figure 1

:RUN MYPROG

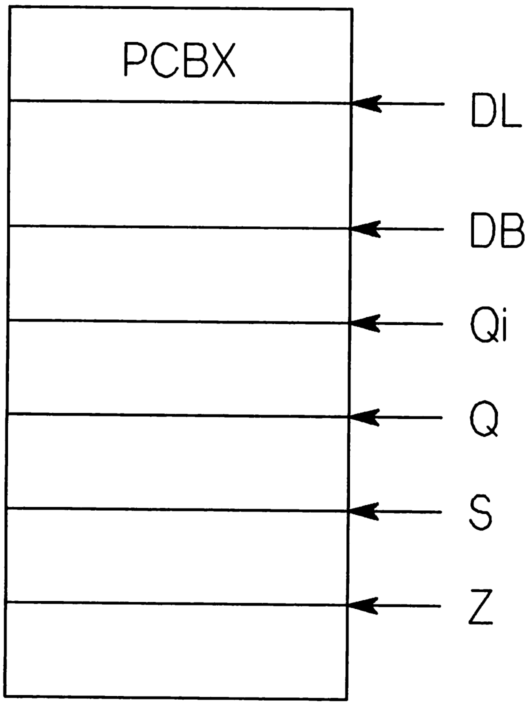


Figure 2a

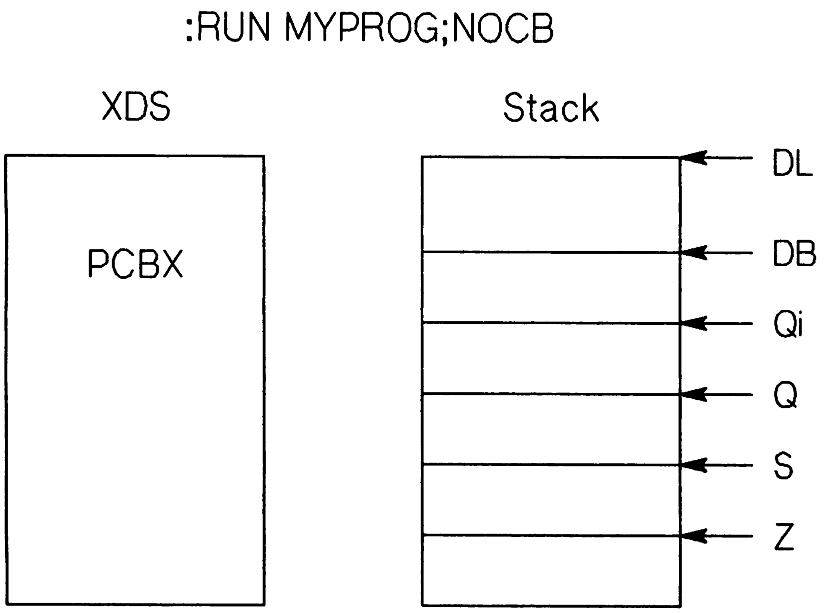
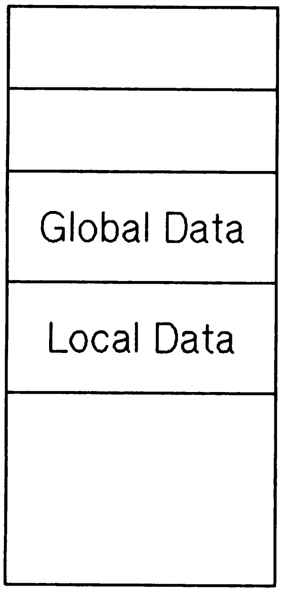


Figure 2b

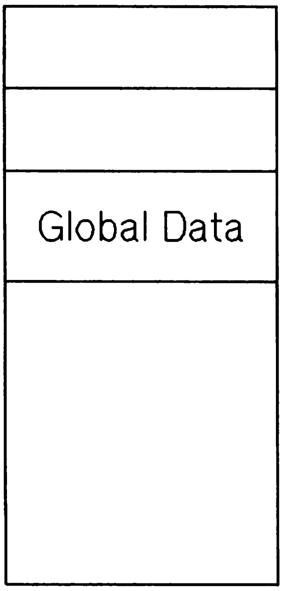
Static Subprogram



Stack after GOBACK from subprogram

Figure 3

Dynamic Subprogram



Stack after GOBACK from subprogram

Figure 4

Nesting Subprograms

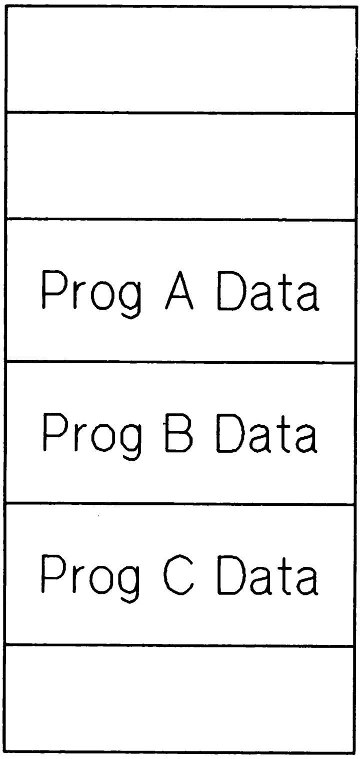


Figure 5

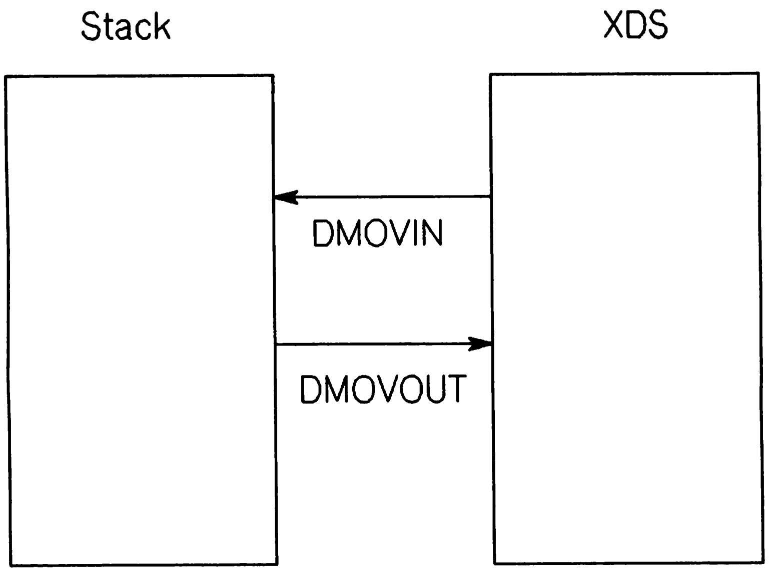


Figure 6

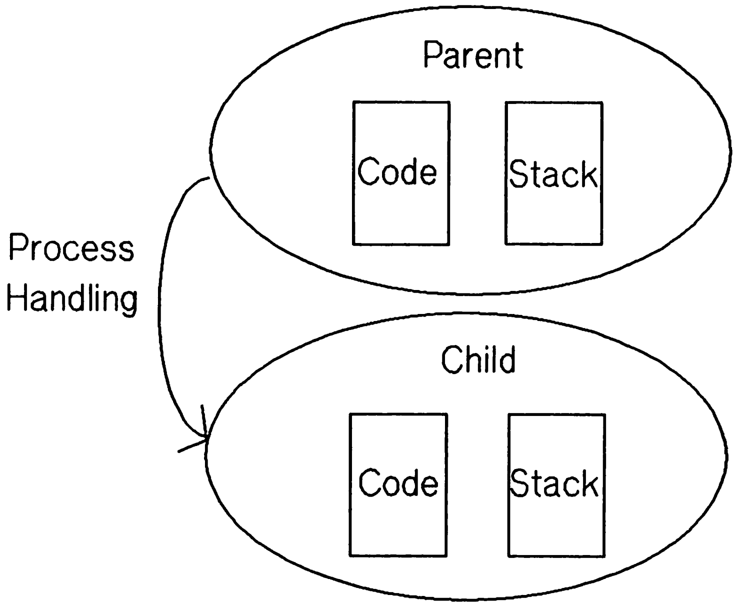


Figure 7

Figure 8

Process display within Process context of OPT.PUB.SYS

```
PIN: 403  MYPROG.MYGROUP.MYACCT        USER:  ME.MYACCT      (S227)  1:12 PM
-------------------------------------------------------------------------------
        STACK INFORMATION         | CPU TIME:    7057 MSEC|STATUS FLAGS:
    DST:   1403* SYSOV:   1329  4.8%|                       |
   SIZE: 27648   DL-DB:   7640 27.6%| PRIORITY: 152         |MRNG  GRIN  LRIN
 MAXDATA: 31223  DB-QI:  10967 39.7%| CAPABILITIES:   ND SF|BIO    I/O  UCOP
 MAX Z-DL: 22955  QI-Q:   3924 14.2%|  BA IA PH DS          |JUNK  TIME  MSG
                   Q-S:      9   .0%|                       |
                   S-Z:   3779 13.7%|                       |
-------------------------------------------------------------------------------
```

A version of this article originally appeared in the Nov. 1987 edition of HP Professional magazine.