HEWLETT-PACKARD GENERAL SYSTEMS USERS GROUP

FEBRUARY, 1980 MEETING

RESOURCE OPTIMIZATION SERIES

Checkstack and Controlling COBOL Stacks

David Greer
Premier Cablevision Ltd.
5540 Cambie St.
Vancouver, B.C.
Canada, V5Z 3A2

I. Introduction

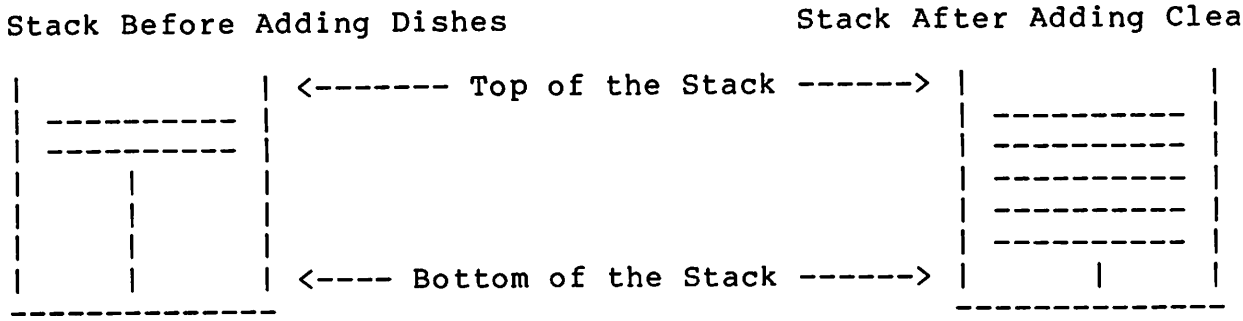
One of the most important resources of the HP 3000 is memory. Checkstack allows the programmer the ability to monitor his data space and hence the amount of memory that his programs use. The less memory your program consumes, the less load it is on the system and the better performance you can expect. Specifically, checkstack gives excellent guidelines on what values should be used in the STACK= parameter of the PREP command.

Note that user memory is broken into two main parts. The first is program code, the size of which can be controlled by the SEGMENTER and compiler directives, but which is another topic altogether. The second type is the data stack, and this paper concentrates on this type of user memory.

In order to describe checkstack, the programmer must be aware of the HP 3000 stack mechanism and how it works. The following is a brief overview of the stack from a COBOL programmer's point of view.

II. The Stack

All user-accessible memory on the HP 3000 is partitioned into variable length stacks. A stack works like a plate stacker in a restaurant. When a plate is taken off the top of the stack it gets smaller and when a new set of clean plates arrives from the kitchen the stack grows larger.



The stack is a very dynamic resource, constantly varying in size as dishes are removed and added. The stack used in memory is very similar to this simple model. When a program requires more memory the stack grows, and when a program releases memory the stack may or may not grow smaller.

Probably the least understood concept among COBOL programmers is just why programs require more or less memory. In general there are three types of COBOL programs.

1. MAINLINE - Indicated by $CONTROL SOURCE in the first control line of the program.
2. SUBPROGRAM - Indicated by $CONTROL SUBPROGRAM in the first control line of the source program.
3. DYNAMIC - Indicated by $CONTROL DYNAMIC in the first control line of the source program.

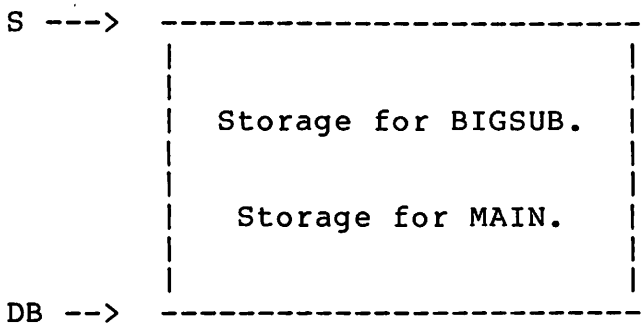
A single source file falls into one of the above three categories, with MAINLINE being the default. SUBPROGRAM and DYNAMIC indicate subroutines which can be called from the MAINLINE or from other subroutines. An MPE program is one or more COBOL source files compiled into a USL file which is then PREP'ed into a program file. It is this program file which you then :RUN.

The control record of the source files determines how memory is allocated at run time. MAINLINE and SUBPROGRAM-specified source files cause ALL memory for the program to be allocated at run time, and at least this amount of memory is used for the entire time that the program is running. Contrast this with DYNAMIC routines which cause the stack to grow and, possibly, contract for each call to the procedure. Note that any SUBPROGRAM that initializes all its variables for each call to the procedure can be changed to DYNAMIC without affecting the program logic. The following example should help clarify these points.

```
$CONTROL SOURCE

 PROGRAM-ID.  MAIN.


 PROCEDURE DIVISION.
 00-MAIN.

       CALL "BIGSUB".

DATA AREA IS %000615 WORDS.
```

```
$CONTROL SUBPROGRAM

 PROGRAM-ID.   BIGSUB.


 PROCEDURE DIVISION.
 00-MAIN.

          ...

DATA AREA IS %001005 WORDS.
```

The stack looks like this immediately after :RUNning this program.



The pointers in the diagram represent the following:

DB - Data Base pointer. This points to the bottom of the stack.
S - Top of the stack pointer.

Now I introduce four different routines with DYNAMIC subprograms:

```
$CONTROL SOURCE

 PROGRAM-ID.   MAIN2.


 PROCEDURE DIVISION.
 00-MAIN.

       CALL "SMALLSUB".

       CALL "SMALLERSUB".

DATA AREA IS %000615 WORDS.
```

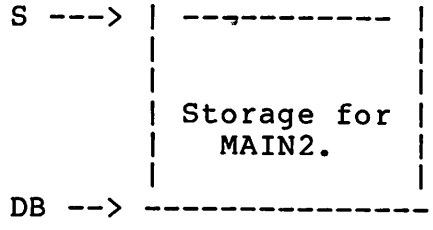
```
$CONTROL DYNAMIC

 PROGRAM-ID.   SMALLSUB.


 PROCEDURE DIVISION.
 00-MAIN.

       ...

DATA AREA IS %001005 WORDS.
```

```
$CONTROL DYNAMIC

 PROGRAM-ID.   SMALLERSUB.


 PROCEDURE DIVISION.
 00-MAIN.

*NESTED CALL

       CALL "SMALLEST".

DATA AREA IS %000520 WORDS.
```

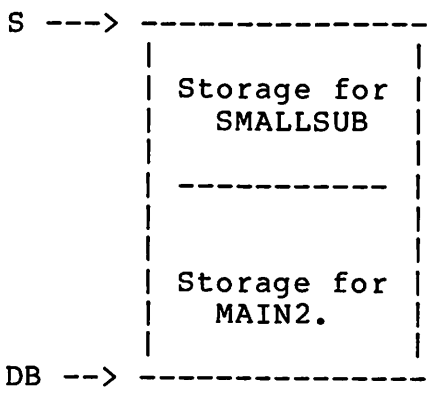
```
$CONTROL DYNAMIC

 PROGRAM-ID.   SMALLEST.

*CALLED FROM SMALLERSUB.

 PROCEDURE DIVISION.
 00-MAIN.

       ...

DATA AREA IS %000310 WORDS.
```

The stack now looks as follows:

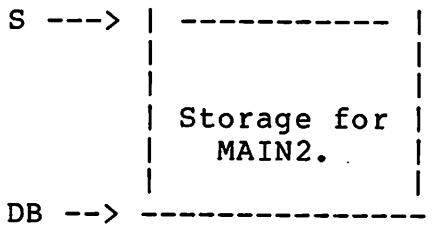
Before the call to "SMALLSUB".



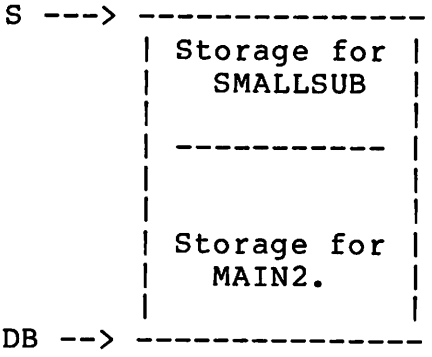
After the call to "SMALLSUB".



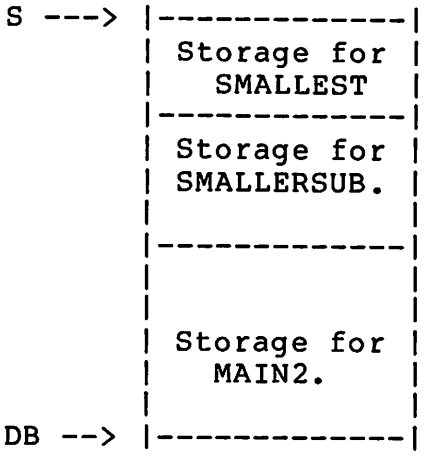
After the return from "SMALLSUB".



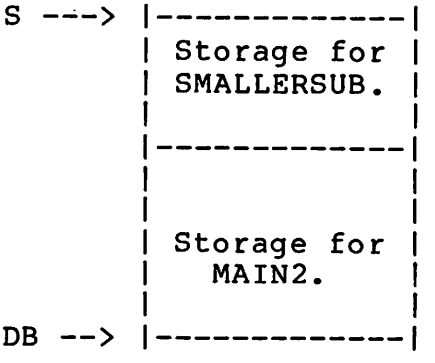
After the call to "SMALLERSUB".



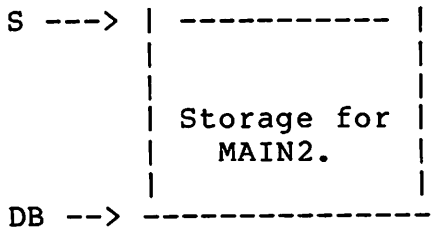
After the call to "SMALLEST".



After the return from "SMALLEST".



Finally we see the stack after the return from SMALLERSUB.

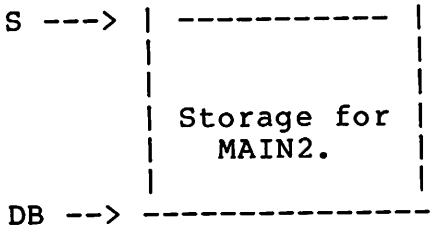


During these subroutine calls, the S pointer has been moving up and down as more or less memory was required. In an ideal situation, this is what we would want, but under MPE it is too expensive to constantly expand and shrink the stack size. Contracting the stack size is relatively inexpensive, but expanding the stack involves a large amount of overhead.

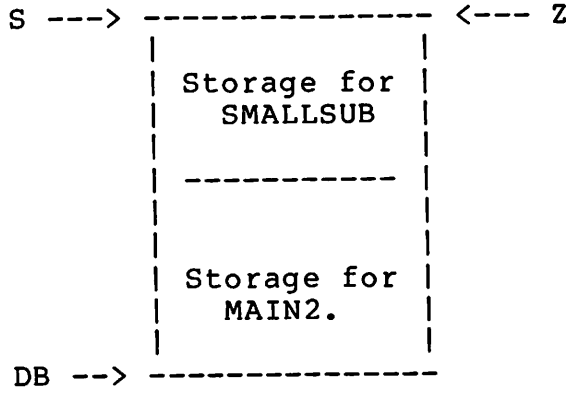
Every time your stack grows, the current data stack, before being expanded, is written out to disc. Then more room is found in memory for the larger stack. When enough memory is found, your stack is copied from disc back into memory. Since this is expensive, MPE won't shrink your stack size; instead, there is another pointer called Z which points to the high-water mark where S has pointed. Note that the amount of memory that you are using is the amount between Z and DB, plus some more, if the program uses VIEW, regardless of where S is.

This is the same example as before but now the Z pointer is included:

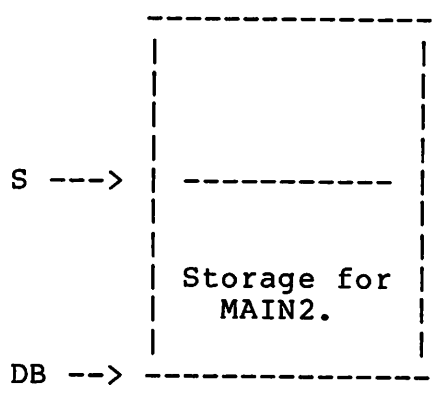
Before the call to "SMALLSUB".



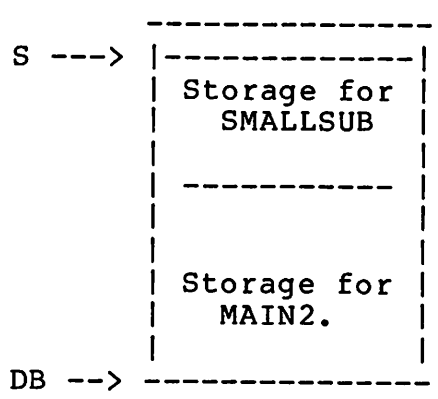
After the call to "SMALLSUB".



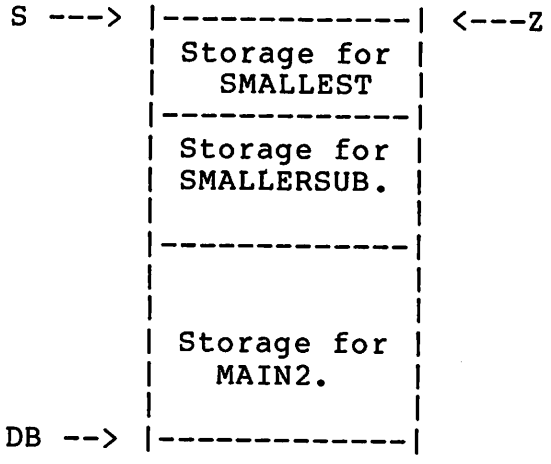
After the return from "SMALLSUB".



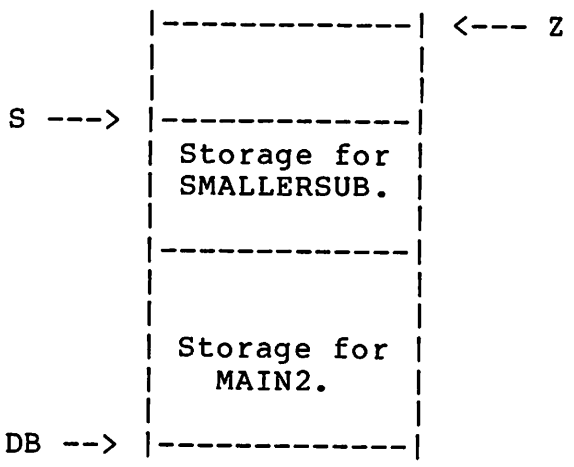
After the call to "SMALLERSUB".



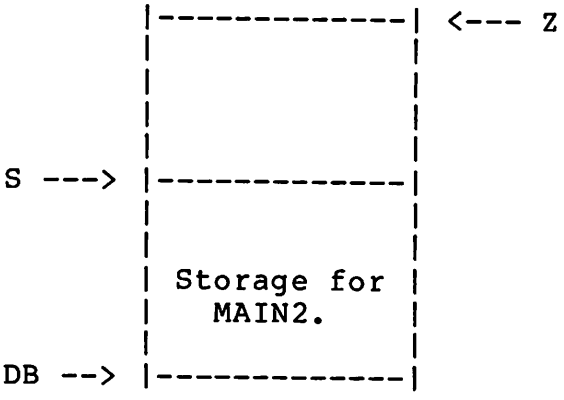
After the call to "SMALLEST".



After the return from "SMALLEST".



Finally we see the stack after the return from SMALLERSUB.



These diagrams should give an idea of how much memory is being taken up by all these various subroutines. The problem is that MPE will expand the amount of stack that the program uses, but it will not shrink the stack. Instead, the programmer must do it explicitly.

III. Checkstack

The type of programs that would use checkstack are those which have a mainline, usually with a small amount of storage, and then have calls to several DYNAMIC subroutines. These subroutines automatically expand the stack as necessary according to how much storage each subroutine needs. Note that if every subroutine is called, then Z will point to the largest amount of storage used, which also includes calls between subroutines. Every call to a subroutine causes the stack to grow, if the amount of storage necessary exceeds Z.

Since it is very tedious to try to keep track of all movements of the stack by hand, Bob Green of Robelle Consulting Ltd. wrote checkstack (available in the contributed library as CHECKSTACK). These SPL routines are called by the COBOL mainline to monitor the stack size. Calling checkstack requires the following in the MAINLINE:

1. Set up a small work space in the working storage section of the MAINLINE.
2. Call checkstack1 at the beginning of the MAINLINE.
3. Call checkstack2 after every procedure call.
4. Just before terminating the program, checkstack3 is called.

The following is an example of a call to checkstack.

```
$CONTROL SOURCE

     ....

 WORKING-STORAGE SECTION.

 01  CHECKSTACK-AREA.
     05 PRINT-FLAG                  PIC S9(4) COMP.
     05 FILLER                      PIC X(18).

*
*  Note:  The PRINT-FLAG can have four values:
*         0.  Print no messages, just adjust the Z pointer as
*             necessary.
*         1.  Print messages on the terminal.
*         2.  Print messages on the console.
*         3.  Print messages on both the console and terminal.
*
*  The PRINT-FLAG must be set before the call to checkstack1. Also,
*  the filler area must never be modified by the MAINLINE, as
*  checkstack uses this area for storage.
```

```
PROCEDURE DIVISION.
00-MAIN.

      MOVE 1 TO PRINT-FLAG.
      CALL "CHECKSTACK1" USING CHECKSTACK-AREA.

10-SELECT-FUNCTION.
      ACCEPT FUNC.

      IF FUNC = "A" THEN
          CALL "SUBA"
      ELSE
      IF FUNC = "B" THEN
          CALL "SUBB"
      ELSE
      IF FUNC = "C" THEN
          CALL "SUBC"
      ELSE
      IF FUNC = "E" THEN
          GO TO 20-FINISH-PROGRAM.

      CALL "CHECKSTACK2" USING CHECKSTACK-AREA.

      GO TO 10-SELECT-FUNCTION.

20-FINISH-PROGRAM.
      CALL "CHECKSTACK3" USING CHECKSTACK-AREA.

      STOP RUN.
```

Checkstack1 finds the current Z value of the stack and the amount of global area used by the MAINLINE. Checkstack2 sees if the stack has expanded past the original Z value which is fixed by the STACK= parameter of the PREP command. If it has grown, then checkstack2 reduces the stack size to the original Z size. Checkstack3 then prints certain statistics according to which print value was specified. A typical example of the output from checkstack3 is as follows:

```
GLOB750  STK2500 #OK54  AVE2550  #ADJ5  SIZ4500
```

The parameters are:

GLOB - Global space used by the MAINLINE.
STK - Initial value of Z which is the STACK= parameter on the PREP command.
#OK - Number of calls to checkstack2 where no adjustment to the stack was necessary. This means that no subroutine or group of nested subroutines expanded past the STK value.
AVE - Average size of the stack when checkstack2 was called.
#ADJ - Number of adjustments to the stack. If this number is much larger than the #OK, then the STACK= parameter on the PREP command is too small.

SIZ - Average size of the stack when it was larger than the value that the program was prepped with.

Some analysis of these figures is now necessary. The Z size isn't decreased unless the stack has expanded past Z plus 512 words. This prevents unnecessary shrinking of the stack. The global storage should be as small as possible. Remember that every user must have his own stack, therefore, all the extra storage in every stack causes a large total decrease in the amount of available memory, especially if there are many users of this program.

As with everything on the HP 3000, there is always a trade off between good and evil. There is a cost associated with this expansion and contraction of the stack size. Shrinking the stack is very inexpensive, as it only results in your stack being contracted when your data stack is swapped out and then later swapped in. Expanding the stack is very expensive, since every stack expansion results in the data stack being swapped out to virtual memory and then swapped back in when more memory is found. The net result of this is that you want to PREP your program with an initial stack size that causes the minimum number of adjustments while keeping the STK value as low as possible.

An excellent example of where CHECKSTACK is most useful is when there are a large number of calls to CHECKSTACK that are OK, and then only about 5% of the calls are for adjustments. Also, if the size of the adjustments is much larger than the average, then CHECKSTACK is helping you get optimal performance from your programs.

Appendix

An Application of Checkstack.

The primary application package at Premier consists of a controlling routine which prompts for commands. For each command there is a call to a COBOL subroutine. Each subroutine may then call other COBOL or SPL subroutines.

During July 1978, we examined the stack size of the program using Son of Overlord and found that the average stack size was approximately 12,000 words. Checkstack was added to the program to monitor and adjust the stack size.

The result was drastic, the average stack size being only 8,000 words instead of 12,000. Only one routine was causing the stack to expand to 12,000 words, but without checkstack the stack was remaining at 12,000 words rather than shrinking. We had a net saving of 4,000 words per user, and there were six users at that time. A total savings of 24,000 words of virtual and real memory.

We still felt that the stack size was too large, so every subroutine was checked to see that it started with $CONTROL DYNAMIC instead of $CONTROL SUBPROGRAM. At the same time unnecessary storage (such as long field lists, instead of the "@" sign) was deleted from each subroutine. The routine that was using 12,000 words of storage was examined, and a special effort was made to delete unnecessary storage. The average stack size decreased to 4,000 words and the worst case became 8,000 words. Approximately 2,500 words of this storage was saved just by changing $CONTROL SUBPROGRAM to $CONTROL DYNMAIC in two of the larger routines. The net savings was again 4,000 words per user. Checkstack and some simple changes to our application software reduced the average stack size by 8,000 words per user.