TITLE: HP 3000/OPTIMIZING ON-LINE PROGRAMS

AUTHOR: ROBERT M. GREEN
ROBELLE CONSULTING LTD.
#130-5421 10TH AVENUE
DELTA, B.C. V4M 3T9
CANADA (604)943-8021

I have identified five general principles which help in optimizing the performance of on-line programs:

* Make each disc access count.

* Maximize the value of each terminal input.

* Minimize the run-time program size.

* Avoid constant demands for execution.

* Optimize for the common events.

FIRST PRINCIPLE: MAKE EACH DISC ACCESS COUNT

Disc accesses are the most critical resource on the HP 3000. The system is capable of performing about 30 disc transfers per second, and they must be shared by system processes (spooling, console operator, etc.), memory management and user programs. (This rate can be increased to 58 per second under the best circumstances, and can degrade to 24 per second when randomly accessing a large file.) Another interesting fact is that a 4096-word transfer takes about the same overhead as a 128-word transfer. Therefore, it is better to read 4096 words in one transfer than to read 128 words 32 times. Another point to remember is that IMAGE database transactions require a lot of immediate disc accesses (from a DBUPDATE, which does one disc write, to a multi-key DBPUT that may require ten or more disc reads/writes).

Some of the operations that consume extra disc accesses on the HP 3000 are:

Increasing the number of keys in a detail dataset, thus causing IMAGE to access an extra master dataset on each DBPUT. Also, making a field a key value means that a DBDELETE/DBPUT is required to change it (which is 10 times slower than a DBUPDATE).

Increasing the program stack size by 8,000 bytes, thus causing the MPE memory manager to perform extra swapping disc accesses to find room in memory for the larger stack.

Improperly segmenting an active program, causing many absence traps to the memory manager to bring the code segments into main memory.

Defining a database or KSAM file with overly large blocksize, thus forcing each user terminal to access a large extra data segment that must be swapped in and out of main memory. (Note: the trade-offs will change when [if?] IMAGE is revised to use shared buffers.)

NOBUF Disc Accesses

When designing your next on-line application, see if there is some way that a random disc file can be used instead of an IMAGE dataset or a KSAM file; Then open that file with NOBUF and access it via directed reads and writes to specific blocks. Normally, when you open a file, the program is assigned an extra data segment to hold the buffer space for the file. Transfers between the file and the program are always done through this extra data segment. When the program requires a record, MPE first checks to see if the record is already in the extra data segment buffers; if so, it is merely transferred from the extra data segment to the user stack. If the block containing the desired record is not in the buffers, MPE issues a read against the disc to bring the block into main memory.

Although this sounds very clever and efficient, it has one major flaw: the extra data segment itself can be swapped. This means that in order to do any file access on a busy system, it may be necessary to read the extra data segment into memory before accessing the data in the disc file. On a heavily loaded system, this could cause a large number of unnecessary disc transfers. NOBUF access does away with all this by providing a direct interface between the user program and the disc files. Blocks are transferred to and from the user stack and the disc without any intervening buffer area. NOBUF is the fastest way to use random disc storage from a user program.

The user program must provide its own buffer space in the stack and call for transfers of data via the block number within the file. When multi-record access is used, it is possible to transfer multiple blocks at a time. The user is responsible for determining which block contains the record that he desires and where within the block the record is located. Simple subroutines can be written to handle this transformation.

A typical use for this kind of file is as a data entry transaction file. As the operator enters the data, it is buffered in the stack until a block is full; then the entire block is written to the disc in one operation. For even better throughput and response time, you might try writing the blocks to the disc with the NO-WAIT option; when this is used, MPE overlaps the write operation to the disc with your next print and read from the terminal. Without NO-WAIT, your program would be suspended until the disc write could be completed by MPE.

(Warning: Be certain that you know when the end-of-file is updated; otherwise, you might find that you have an empty transaction file when the system crashes. I suggest that you move the end-of-file to the limit of the file at the start of the day by writing a null entry in the last record position and then closing the file.) When the transaction file is full (or the day ends), a batch program is used to put the transactions into the final IMAGE dataset or KSAM file. This job can be done in low priority or after hours.

SECOND PRINCIPLE: MAXIMIZE THE VALUE OF EACH TERMINAL READ

Each time a program reads from the terminal, it is suspended and may be swapped out of memory. When the operator hits the carriage return key, the input operation is terminated, and the process must be dispatched again. In order to dispatch a process, MPE must ensure that the data stack and at least one code segment are resident in main memory. If the process is going to access the disc, it may be necessary to make an extra data segment resident also. Unless the computer has enough main memory so that no user segments are ever swapped out, it is desirable to have the process get as much work done as possible before it suspends for the next terminal input (and is swapped out again).

The simplest way to program data entry applications is by prompting for and accepting only one field of data at a time. This is also the least efficient way to do it. The user data stack must be made resident every time the user hits 'return'. (Therefore, the less often the user hits 'return', the larger your stack can afford to be.) Since it is inefficient, fast response time cannot be guaranteed, and the resulting delays are very irritating to operators. They

can never work up any input speed, because they never know when the computer is ready for the next input line. If response time and throughput are the only considerations, it is always preferable to keep the operator typing as long as possible before hitting the 'return' key. Multiple transactions should be allowed per line, with suitable separators, and multiple lines should be allowed without a 'return'.

THIRD PRINCIPLE: MINIMIZE THE RUN-TIME PROGRAM SIZE

The HP 3000 is an ideal machine for optimizing because of the many hardware features available at run-time to minimize the effective size of the program. Even quite large application systems (6000 lines of code) can be organized to consume only a small amount of main memory at any one time. Each executing process on the HP 3000 consists of a single data segment called the "stack", and one or more extra data segments for system storage, such as file buffers. Although a process is always executing some code in a code segment, the code does not properly belong to the process, since one copy is shared by all processes in the system. If a program is to be executed by several terminals, most optimizing should be directed to the data areas (which are duplicated for each user).

Large programs which are not logically segmented make it harder for the memory manager to do its job, and thus cause many disc accesses to be consumed in swapping. In an extreme case, the system can almost be brought to a complete standstill by a very large program executing on many terminals at the same time. The articles listed in Appendix A provide strategies and examples for code segmentation. To simplify a complex problem, follow these guidelines: 1) put initialization, termination and error handling in separate code segments; 2) minimize the number of calls across segment boundaries at run-time; 3) remain in a segment as long as possible; 4) keep segments small (2K-8K words), but don't use too many segments (MPE has a limited overall code segment capacity).

Many more terminals can be supported on a given system if data stack sizes are kept modest (ex: less than 6000 bytes on a 192K-byte machine), and if the code is properly segmented. The simplest way to keep the stack small is to make all data variables local (DYNAMIC in COBOL) and to use global storage only for buffers and control values that must be accessed by all subroutines. The reason that this is so effective is that dynamic local storage is allocated on the top of the stack when the subroutine is entered and is released automatically when the subroutine is left. This means that if the main program calls 3 large subroutines in succession, they all reuse the same space in the stack. The

stack need only be large enough for the deepest nesting situation.

Since the amount of dynamic stack space that will be required by the program is not known at the start of execution, the 3000 provides methods (both automatic and programmatic) to expand the dynamic area. Whenever a stack overflow occurs, MPE automatically allocates more space (up to a MAXDATA limit). Unfortunately, there is no automatic mechanism for reducing the stack size when that additional space is no longer needed. The user application program can include a check in the mainline and shrink the stack back down to the desired size after returning from an oversize subroutine. (See Appendix B for an example.)

The other major way to reduce the size of a data stack is to ensure that constant data items (such as error messages, screen displays) are stored in the code segment instead of the data segment. Since they are never to be modified, there is no logical reason that they must be in the data stack. By moving them to the code segment, one copy of them can be shared by all users running the program. In SPL, this is done by including =PB in a local array declaration or MOVE'ing a literal string into a buffer. In COBOL, constants can be moved to the code segment by DISPLAY'ing literal strings in place of declared data items. In FORTRAN, both FORMAT statements and DISPLAY'ed literals are stored in the code.

FOURTH PRINCIPLE: AVOID CONSTANT DEMANDS FOR EXECUTION

The HP 3000 is a multiprogramming, virtual memory machine that depends for its effectivenss on a suitable mix of processes to execute. Although the sizes of the segments to be swapped have an effect on performance, this is dependent upon the frequency with which memory residency is demanded. Given the same overall configuration and application program sizes, the system supports many more terminals if each one only executes for 5 seconds every 30 seconds than if each one must execute for 60 seconds at a time. Each additional terminal that is demanding continuous execution (in high priority) makes it harder for the operating system to provide proper response time to all other terminals.

Here are some examples of the kind of operation that can destroy response time if performed in high priority:

EDIT/3000, a GATHER ALL of a 3000-line source file.

QUERY, serial read of 100,000 records

SORT, sorting 50,000 records.

COBOL, compiling on 4 terminals at once.

All of these operations should be done in low priority in batch STREAM Jobs. These Jobs can even be created dynamically by on-line programs. In this way, the on-line user still requests the high-overhead operation, but the system fulfills the request when it has the time.

FIFTH PRINCIPLE: OPTIMIZE FOR THE COMMON EVENTS

In any application where there is a large variation between the minimum and maximum load that a transaction can cause, the program should be optimized around the most common size of transaction. In any application with a large number of on-line functions, it is likely that a small number of functions are used most of the time. In this case, all optimization efforts should be aimed at the commonly used functions and other functions left as is. This is especially feasible on the HP 3000 because of code segmentation and dynamic stacks.

If N is the average number of records in a transaction (i.e, the number of lines on a customer order, maximum is 500), then allow room in your stack for N records. If you only allowed for one record, then there would be unneeded disc thrashing. Alternatively, if you provide room for the maximum number, then the data stack is much larger than actually needed most of the time. Having a larger data stack may cause the system to overload, eliminating the benefits of keeping the records in your stack. It is recommended that room in the stack be allowed for slightly more than the average number, and that a NOBUF disc file be used to "page" this area on very large transactions.

OPTIMIZING CASE STUDY #1: QEDIT

QEDIT is a high-speed, low-overhead source program editor developed by Robelle Consulting Ltd. The primary objective of QEDIT is to provide the fastest possible editing with the minimum possible system load. Other objectives include conservation of disc space, similarity to EDIT/3000 in command syntax, ability to recover the workfile following a system crash or program abort, and increased programmer productivity.

QEDIT and the First Principle: Disc Accesses

In order to reduce disc accesses, QEDIT had to eliminate the overheads of the TEXT, KEEP and GATHER ALL commands of EDIT/3000. These three operations have the most drastic impact upon the response time of the other users. QEDIT attacks the problem of KEEPs by providing an interface library that fools the HP compilers into thinking that a QEDIT workfile is really a "card image" file. As a result, it is never necessary to KEEP a workfile before compiling it. Since KEEPs are rarely used, most TEXTs are eliminated. TEXT is only needed when you want to make a backup or duplicate copy of an existing file. It was anticipated that most users would maintain their source files exclusively in workfile format, so the TEXT'ing of workfiles was optimized (by using NOBUF, multi-record techniques) to be at least 4 times faster than a normal TEXT of a card image file. The GATHER ALL operation is slow because it makes a copy of the entire workfile in another file. QEDIT renumbers up to 12 times faster by doing without the file copy.

Disc accesses during interactive editing (add, delete, change, etc.) were minimized by packing as many contiguous lines as possible into each disc block. The resulting workfile is seldom over 50% of the size of a normal KEEP file or 25% of the size of an EDIT/3000 K-file (workfile). Most QEDIT users maintain all of their source programs in workfile form, since this saves disc space, simplifies operations (there need only be one copy of each version of a source program), and provides optimum on-line performance.

QEDIT always accesses its workfile in NOBUF mode and buffers all new lines in the stack until a block is full before writing to the disc. Wherever possible in the coding of QEDIT, unnecessary disc transfers have been eliminated. For example, the workfile maintains only forward direction linkage pointers, which reduces the amount of disc I/O substantially. Results of a logging test show that reducing the size of the workfile and eliminating the need for TEXT/KEEP reduces disc accesses and CPU time by 70-90%.

QEDIT and the Second Principle: Terminal Accesses

QEDIT allows multiple commands per line, plus multiple data lines per data line input (i.e, you can enter 7 lines of text without hitting 'return'). All interaction with the terminal is done directly through the READX and PRINT instrinsics.

QEDIT and the Third Principle: Program Size

QEDIT is a completely new program, written in highly structured and procedurized SPL. The resulting program file consists of 7 code segments of 1780 words (decimal) each. Only two code segments are required for most editing commands, while the most common function (adding new lines) requires only one code segment most of the time.

QEDIT uses a minimum data stack and no extra data segments. All error messages are contained in the code, isolated in a separate code segment that need not be resident if you make no errors.

QEDIT and the Fourth Principle: Constant Demands

Most QEDIT commands are so fast that they are over before a serious strain has been placed on the host machine. For example, a 2000-line source program can be searched for a string in four seconds. For those operations which still are too much load, QEDIT provides the ability to switch priority subqueues dynamically. In fact, the system manager can dictate a maximum priority for certain operations such as compiles or TEXT and KEEP commands.

QEDIT and the Fifth Principle: Common Events

The entire design of QEDIT is based on the observation that program editing is not completely random. When a programmer changes line 250, he is more likely to require access to lines 245 through 265 next than he is to lines 670 through 710. This observation dictated the design of the indexing scheme for the QEDIT workfile.

There are many examples of optimizing for the most common events in QEDIT: the blocksize will hold about a screenful of data lines, built-in compiler, fast renumbering command (600 lines per second) in place of a GATHER command, faster TEXT'ing of workfiles than KEEP files (4 to 7 times faster).

Results of Applying the Principles to QEDIT

In less than 7 seconds, QEDIT can text 1000 lines, renumber them and search for a string. Commands are 80% to 1200% faster than EDIT/3000, program size is cut in half, and disc I/O and CPU time are reduced by up to 90%.

In order to measure performance, an editor-callable "procedure" was written that calculates the elapsed time (using TIMER intrinsic) and the processor time (PROCTIME intrinsic) between events. QEDIT measured faster than EDIT/3000 by these percentages:

Renumber	1204% faster
List to printer	115% faster
Find string	613% faster
Change	645% faster
Keep	82% faster
Text from keepfile	44% faster
Text from workfile	733% faster

The more efficient the programming of an operation, the less system resources it consumes. MPE provides a "logging" facility to record the resource usage of programs for later analysis. Both QEDIT and EDIT/3000 were used to perform a typical program maintenance change (edit, compile, correct errors). According to the logging statistics, QEDIT reduced overhead by these percentages:

Physical disc transfers	93% reduction
Disc space required	87% reduction
CPU time	72% reduction
Program size	63% reduction
Total data space	53% reduction
Data stack size	43% reduction

Programming of QEDIT began in March 1977 and user-site testing in September 1977. At the present time (September 1978), there are 20 QEDIT user installations. QEDIT shows what can be accomplished by applying all of these optimizing principles in the design of one system. In any given application system, it may not be possible to take advantage of all five principles; but to whatever extent they can be applied, the resulting system will provide better service than it would have.

For more information on QEDIT, contact me directly:

Robert M. Green
Robelle Consulting Ltd.
#130-5421 10th Ave.
Delta, B. C. Canada
V4M 3T9
Phone: (604) 943-8021

OPTIMIZING CASE STUDY #2: APPLYING PAYMENTS

In this accounts receivable system, 24,000 invoices per month are posted to 10,000 customer accounts. The number of unpaid items per customer varies from one or two (a lot of accounts) to 500 (a few major accounts). The A/R are maintained on an open-item basis. That is, the invoices appear on the customer's statement each month until they are matched up with a payment and considered reconciled. About 200-300 cheques are posted to the database each day. The problem is to allow the A/R clerks to "apply" the payments to the proper invoices in the cheapest possible manner. Certain other constraints exist: the machine is a Series I, only dumb terminals are to be used, and the system is already supporting about 17 terminals and seems fully loaded.

The computer system cannot tolerate the overhead to scan down the chain of records for each account (DBGETs) and print them on the screen. There is too heavy a load already. In addition, the software would have to skip over (i.e., set and ignore) a large number of paid invoices to find the unpaid ones.

A/R and the First Principle: Disc Accesses

A/R uses a database to index entries by account and sort them by date, but allows no on-line updates to the database. They are too slow and too hard to control (recover/balance). Updates are only allowed by sequential batch programs.

Each clerk is provided with a transaction disc file for her "ledger", containing copies of her active accounts (14 entries/block). She also has a printout that shows each account and gives its location in the file. The disc file and associated lineprinter report are prepared in batch. The user accesses this file in on-line mode and converts the entries into database transactions.

The transaction file is accessed in NOBUF mode and contains only unpaid invoices. All on-line activity is done into this file, then, at night, those entries which have been marked in the file for application are retrieved from the database and updated.

A/R and the Second Principle: Terminal Accesses

The user input syntax allows (but does not require) many individual instructions to be entered in each input line. This example applies a payment to seven invoices and writes a small adjustment against one invoice:

```
/1ABDEFGH,A(1010-1010,7.50,C)
```

A major design problem was how to refer to the items that are on the customer's account. The invoice number is too long for efficient data entry (and subject to errors). A sequence number could have been assigned to each entry on an account. However, invoices are not paid in sequence; eventually, the sequence numbers would be as large as the invoice numbers. A quick calculation showed that the time required to assign new sequence numbers was prohibitive (because of DB inefficiencies). The scheme settled upon assigned relative position numbers to each unpaid item on a dynamic basis, but these numbers are not actually stored in the database. In order to shorten input, an alphanumeric code was used (A,B,C...Z,A1,...). In retrospect, a pure numeric sequence number might have been better because of the input speed of numeric keypads.

A/R and the Third Principle: Program Size

A/R is written entirely in SPL/3000. Stack sizes are modest (2K-3K decimal or less), and only one disc block is kept in the stack. SPL procedures were created to simulate a mini-file system for the transaction files. The procedures do all deblocking and disc input/output. This simplified coding of the three major programs.

A/R and the Fourth Principle: Constant Demands

There are a few special transactions that can take up to a minute, but they are very rare and can be ignored. Most transactions are very short, and all data is available in memory, or is one disc read away.

A/R and the Fifth Principle: Common Events

This principle was applied heavily to A/R. The most common event is to apply a payment that came in yesterday to an old invoice(s). Also, most accounts have less than 10 outstanding items. Therefore, this system anticipates the next day's requests by creating the batch-file/printout of all the accounts with an unapplied payment. For those accounts that require attention, but have no payment, the clerk loads them into her batch file on-line (rare). The blocksize was picked so that most accounts could fit in one block.

The transaction to apply a payment is:

```
>10117A67/1AGH
            ^Invoice lines
          ^Payment number
        ^Disc location of the account (from printout)
  ^Account number
^Prompt Character
```

Since starting production, we have discovered that usually the account # and location # entered is just the one that sequentially follows the last one. Therefore, the system will someday be changed to allow entry of * for next account.

When converting from the manual system to a pure on-line computer system, the ability to write notes on the customer's account card was lost. After a few months, we found ourselves under heavy pressure to create new types of transactions in the system to handle the many special cases that arose (paid twice, overpaid, short-paid, took credit note twice, etc.). The original design only allowed for four types of transactions: invoice, payment, adjustment and journal entry (a large adjustment with a unique number assigned for control purposes). Rather than clutter up the design, we added the ability to write multi-line comments for any journal entry. With these comments, the A/R clerk can now communicate directly with the customer's accounts payable clerk to explain the problem in English. Since the comments are kept in a separate dataset, indexed by the unique journal entry number, there is no additional overhead on ordinary transactions.

Basis for future expansion: since most accounts pay in a simple pattern, the computer will (in batch) pre-apply the payments when creating the transaction file. Then the operator need only take action if the computer has selected incorrectly.

Results of Applying the Principles to A/R

The application maintains 10,000 accounts with 24,000 invoices per month, using two ADM-3 terminals on a CX-3000 with 15-19 other terminals doing less optimized things. A/R staff has been reduced from seven people to two. At the same time, the three terminals used for program development were switched to QEDIT. Response time has actually improved on old applications. At the same time, 2 terminals have been added to the system.

For more information on this example of applying optimizing principles, contact the user site directly:

Gary Nordman, Manager of Systems Development
Malkin & Pinton Industrial Supplies
325 East Fifth Avenue
Vancouver, B.C. Canada
V5T 1H6

APPENDIX A: REFERENCES ON HP 3000 OPTIMIZING

[1] Transaction Processing on the HP 3000 Series III.
Some HP field System Engineers have this internal HP document which describes the internal workings of these software products:
IMAGE
KSAM
FILE SYSTEM
COBOL
FORTRAN

[2] COMMUNICATOR No. 14.
Page 87, Block/Page mode problems.

[3] COMMUNICATOR No. 12.
Segmentation in COBOL

[4] COMMUNICATOR No. 5.
Segmentation for Maximum Efficiency of System-Type Programs.

[5] JOURNAL-3000 Vol 1, No. 6.
KSAM vs. IMAGE
HP 3000 with Front-End Processor
FORTRAN Optimization

[6] JOURNAL-3000 Vol. 1, No. 5.
QEDIT, Quick Program Editing, Small Appetite for Computer Time.

[7] JOURNAL-3000 VOL. 1, No. 4.
Using Extra Data Segments.
Common Programming Errors with IMAGE/3000.

[8] CONTRIBUTED LIBRARY, Vol I/II.
IDEA Program
IDEAII Program
RESP Program
IDLE Program
PROGSTAT PROGRAM

[9] CONTRIBUTED LIBRARY, Vol III.
SOO Program
DBREBILD Program

[10] CONTRIBUTED LIBRARY, "Vol IV".
DBSTAT Program
DBCHANGE Program

[11] SCRUG MEETING LIBRARY, March 1978.
FASTER - An essay on writing programs for greater efficiency.
OVERLORD (See also SOO.)
DBSTAT - Internal efficiency of master datasets.
SHOWVM - Shows virtual memroy.
STACKOPT - Stack optimizing routines.

[12] SCRUG MEETING NOTES, March 1978.
Extra Data Segments and Process Handling
Operator Utilities

[13] INTERNATIONAL USERS MEETING, 1977.
KSAM (see extra data segment size, load times)
IMAGE for the advanced User
Optimizing FORTRAN IV/3000
RPG/3000 Programming Optimization
Data Entry Techniques
Segmentation
MPE II Measurement and Optimization
MPE C Measurement and Optimization

[14] INTERNATIONAL USERS MEETING, February 1975.
Software Optimization Throush Segmentation

[15] INTERNATIONAL USERS MEETING, May 1974.
Program Performance

[16] CCRUG MEETING MINUTES, May 9, 1978.
IDEA Program
DBDRIVER Program

[17] PERFORMANCE GUIDELINES/SERIES III (HP 5953-0533).
Note the extra load of synchronous terminals(p.9) and the dramatic increase in the number of terminals supported when a simple file is used instead of IMAGE/DEL/COBOL.

[18] SPL/3000 FOR COMMERCIAL APPLICATIONS, EFFICIENCY WITH EASE OF MAINTENANCE.
Report available from Robelle Consulting Ltd.

APPENDIX B: SHRINKING THE STACK SIZE

The following SPL code can be added to any program that calls a lot of procedures (or subprograms in COBOL) in order to dynamically optimize the size of the data stack.

CHECKSTACK LIBRARY SUBROUTINES

1. Checks for excessive dynamic stack space after subroutine calls and adjusts the stack size; consists of three routines that are intended to be called from the mainline of an application program that uses many subprograms with varying data requirements.

2. Contents: CHECKSTACK1, CHECKSTACK2, CHECKSTACK3.

3. Parameters: WORKSPACE, 20 bytes of global data in the calling program. The proper COBOL definition is:

```
       01 CHECK-STACK-SPACE .
         05  PRINT-RESULTS-FLAG  PIC  S9(3) COMP VALUE N.
      *          N=0(NO PRINTOUT),1(ON TERMINAL),
      *            2(ON CONSOLE),3(ON BOTH).
         05  FILLER              PIC X(18).
```

HOW TO USE CHECKSTAK:

1. Add the WORKSPACE to the data division of your program and set the desired PRINT'FLAG value(see step 4).

2. At the start of program execution:

```
CALL "CHECKSTACK1" USING CHECK-STACK-SPACE.
```

This call should occur once at the start of the mainline. The purpose is to record the size of the dynamic stack area before any subprograms are called. This size is determined by STACK=XXXX in the :PREP or :RUN commands.

3. After returning from each subprogram call:

```
CALL "CHECKSTACK2" USING CHECK-STACK-SPACE.
```

This call compares the current dynamic stack area with the initial size and if it is over 512 words larger (1024 bytes), reduces it back to the initial.

4. At the end of program execution:

```
CALL "CHECKSTACK3" USING CHECK-STACK-SPACE.
```

This call prints statistics on stack usage on either $STDLIST or the console or both. Format is:

```
  GLOB99  STK99  #OK99  AVG99   #ADJ99    SIZ99

  ^ Global stack size in decimal words
         ^ Initial dynamic stack size
               ^ Number of "OK" subprogram calls
Average stack size per "OK" call^
     Number of times stack was adjusted ^
          Average stack size per adjusted call ^
```

Start with the default value for STACK= (about 800) and a large value for MAXDATA (20000). If all of the subprogram calls are adjusted (i.e., OK=0), increase the STACK= value. Try to find a value where most of the subprogram calls execute without having to shrink the stack afterwards, but not so large that there are no large subprograms left ot adjust.

INSTALLATION OF CHECKSTAK:

1. Type the following SPL source code into the system using QEDIT or EDIT/3000 and create a source file.

2. Compile the source file and make corrections until there are no errors:

```
:SPL SOURCE
```

3. When you have a successful compile, save the USL file, using this command:

```
:SAVE $OLDPASS,USLSPL
```

4. Either COPY the segment called "LIBSEG1" into the USL file of your application program (using the :SEGMENTER commands AUXUSL and COPY) or add it to an SL (:SEGMENTER or :SYSDUMP).

```
$CONTROL LIST,SUBPROGRAM,MAIN=LIBSEG1,ERRORS=9
BEGIN

PROCEDURE  CHECKSTACK1 ( BUF ) ;
    INTEGER ARRAY   BUF;
BEGIN
<< DEFINE STRUCTURE/USE OF BUF >>
DOUBLE ARRAY DBUF (*) = BUF;
DEFINE
  PRINT'FLAG    = BUF#      ,INITIAL'SPACE   = BUF(1)#
 ,SHRINK'COUNT = BUF(2)#   ,OK'COUNT         = BUF(3)#
 ,OK'SPACE   = DBUF(2)#    ,SHRINK'SPACE    = DBUF(3)#
 ;
INTEGER Z,Q;

IF NOT ( 0<=PRINT'FLAG<=3 ) THEN
  PRINT'FLAG := 1; <<DEF>>

PUSH (Z,Q);  Z:=TOS; Q:=TOS;

INITIAL'SPACE := Z - Q;
BUF(2) := 0;
MOVE BUF(3) := BUF(2),(7);

END; <<CHECKSTACK1 >>


PROCEDURE CHECKSTACK2 ( BUF ) ;
    INTEGER ARRAY   BUF;
BEGIN
<< DEFINE STRUCTURE/USE OF BUF >>
DOUBLE ARRAY DBUF (*) = BUF;
DEFINE
  PRINT'FLAG    = BUF#      ,INITIAL'SPACE   = BUF(1)#
 ,SHRINK'COUNT = BUF(2)#   ,OK'COUNT         = BUF(3)#
 ,OK'SPACE   = DBUF(2)#    ,SHRINK'SPACE    = DBUF(3)#
 ;
INTEGER Z, Q, STACKSIZE;
INTRINSIC ZSIZE;

PUSH ( Z,Q ); Z:=TOS; Q:=TOS;
STACKSIZE := Z - Q;
IF STACKSIZE > (INITIAL'SPACE + 512) THEN BEGIN
  ZSIZE ( Q + INITIAL'SPACE );
  SHRINK'COUNT := SHRINK'COUNT + 1;
  SHRINK'SPACE := SHRINK'SPACE + DOUBLE(STACKSIZE);
  END
ELSE BEGIN
  OK'COUNT := OK'COUNT + 1;
  OK'SPACE := OK'SPACE + DOUBLE(STACKSIZE);
  END;

END; <<CHECKSTACK2>>
```

```
PROCEDURE CHECKSTACK3 ( BUF) ;
    INTEGER ARRAY   BUF;
BEGIN
<< DEFINE STRUCTURE/USE OF BUF >>
DOUBLE ARRAY DBUF (*) = BUF;
DEFINE
  PRINT'FLAG    = BUF#      ,INITIAL'SPACE   = BUF(1)#
 ,SHRINK'COUNT = BUF(2)#   ,OK'COUNT         = BUF(3)#
 ,OK'SPACE   = DBUF(2)#    ,SHRINK'SPACE    = DBUF(3)#
 ;
INTEGER ARRAY P(0:38);
BYTE ARRAY P'(*)=P;
INTEGER TERMINAL;
INTEGER GLOBAL'SPACE;
INTRINSIC PRINT,PRINTOP, ASCII,DASCII,WHO,DATELINE;

IF PRINT'FLAG = 0 THEN RETURN;

IF PRINT'FLAG=2 OR PRINT'FLAG=3 THEN BEGIN
  << PRINT IDENTIFYING MESSAGE ON THE CONSOLE >>
  P:="   "; MOVE P(1):=P,(38);
  MOVE P :="CHECK-STACK:";
  WHO(,,,P'(12),P'(21),P'(30),,TERMINAL);
  MOVE P'(39) := "ON";
  ASCII(TERMINAL,10,P'(42));
  P'(20):=P'(29):=".";
  PRINTOP(P,-46,0);
  END;

P:="   "; MOVE P(1):=P,(38);
MOVE P:="GLOB";
PUSH (Q);
GLOBAL'SPACE := TOS;
ASCII(GLOBAL'SPACE,10,P'(4));
MOVE P'(10):="STK";
ASCII(INITIAL'SPACE,10,P'(13));
MOVE P'(19):="#OK";
ASCII(OK'COUNT,10,P'(22));
MOVE P'(28):="AVG";
DASCII(OK'SPACE/DOUBLE(OK'COUNT),10,P'(31));
MOVE P'(37):="#ADJ";
ASCII(SHRINK'COUNT,10,P'(41));
MOVE P'(47):="SIZ";
DASCII(SHRINK'SPACE/DOUBLE(SHRINK'COUNT),10,P'(50));

IF PRINT'FLAG=2 OR PRINT'FLAG=3 THEN
  PRINTOP(P,-56,0);
IF PRINT'FLAG=1 OR PRINT'FLAG=3 THEN
  PRINT(P,-56,0);

END; << CHECKSTACK3 >>
END <<LIBRARY >> .
```