Avoiding Heap Overflow and Stack Overflow with Pascal/3000

by Christopher P. Maitz
Hewlett-Packard

This paper is an attempt to share with the Pascal/3000 community things that were learned in the development of the Pascal/3000 compiler. The compiler is written in Pascal and is compiled with itself. The solutions we found to avoid heap overflow and stack overflow while using the Pascal/3000 compiler should be helpful to other developers of Pascal systems.

The Pascal/3000 project team has been facing the problem of lack of space during compilation since the early days of the Pascal/3000 compiler's development. We currently use the Pascal/3000 compiler to compile its own source, but before we reached the stage in development when enough features existed in Pascal/3000 to make self-compilation possible, we compiled our source with the "P4" compiler available in the contributed library. The P4 compiler took Pascal source code and produced P-code. An assembler then converted the P-code into SPL/3000 source code, at which point the SPL/3000 compiler was used to produce 3000 code. This painful process was a great motivating factor for us to develop the compiler to the point at which we could compile the compiler with the compiler. As a foreshadowing of future woes, we faced the space problem even with the P4 compiler. My first assignment as part of the Pascal/3000 team was, in fact, to enhance our version of the P4 compiler so that its symbol table would pack two characters per word rather than one, doubling the size of its symbol table.

What exactly is the stack size limitation problem? A program executing on the HP3000 uses a stack as shown in Fig. 1. Primary and Secondary DB contain the global variables. When a procedure is called, its activation record (stack marker and local variables) is pushed onto the stack. The set of stack markers between QI (Q Initial, the stack marker for the outer block) and Q (the stack marker for the currently executing procedure) shows at any given time the dynamic calling sequence of the program. The area between Q and S (the top of stack) represents space allocated for the current procedure's local variables and any expressions undergoing evaluation. Between S and Z (the stack limit) is space for the expansion of S.

The other side of the stack is the area between DL and DB. This area can be thought of as free space as far as the system is concerned. For Pascal/3000 programs, however, this area contains the heap, where storage is allocated dynamically. Whereas the system automatically increases Z as space is needed above S, the Pascal runtime library heap routines control the size of the DL-DB area directly, using the intrinsic DLSIZE. The total size of the regions both above and below DB must not exceed the stack segment size, which is determined when the system is configured. The maximum size is 31232 words. (Actual space available to the program is less, due to process information that the system keeps in the PCBX in the user's stack.)

When the Pascal/3000 compiler requests data to be allocated in the heap and not enough space exists in the stack, the Pascal runtime library traps the condition and the error

SYSTEM RESOURCE EXHAUSTED 1, COMPILE TERMINATED (426) is generated and the compilation is terminated. When the Pascal/3000 compiler runs out of space at the other end of the stack (due to a procedure call, allocation of local variables, expression evaluation), an error also occurs. This error, unfortunately, cannot be trapped as the heap overflow error can. Instead of a Pascal error message, the system aborts the compiler and generates the message

```
ABORT :PASCAL.PUB.SYS.<segment-#>.<segment-offset>
PROGRAM ERROR #20 :STACK OVERFLOW
```

Both of these overflow situations will be referred to simply as the "stack size limitation problem" or "overflow problem" in the rest of the paper.

It is the responsibility of the runtime library to balance the areas on either side of DB. When space is needed in the heap and not enough space is left between the internal Top-of-Heap pointer and DL, DLSIZE is called to increase the area between DL and DB. When space is deallocated through use of the Pascal pre-defined heap routine RELEASE, DLSIZE is called to cut back this area, allowing the system more room with which to increase Z. The stack segment of a Pascal/3000 program is then split between the space used on either side of DB. The compiler has very few static data structures; its structures are almost entirely dynamic, the sizes of which depending significantly on the structure and character of the user's program. The Pascal/3000 compiler as it runs will generally require only a few thousand words between DB and Z, leaving the rest for the heap.

The compiler uses the dynamic structure of the heap to its advantage. When compiling a program the compiler processes the global information first, which includes labels, constants, types, variables and external procedure headers. Even though space is allocated for these dynamically in the heap, it remains allocated throughout the entire compilation since global identifiers can be referenced anywhere in the program.

Processing procedures is very different from processing the outerblock and therein lies the effectiveness behind several key space-saving ideas that will be presented shortly. A procedure's identifier and parameter list are stored in the heap, along with other global information. Before the declarations and the body of the procedure are processed, a MARK is done in the heap which saves the Top-of-Heap pointer (Fig. 2.a). The procedure's declarations and statements are then processed and stored in the heap (Fig. 2.b), but, after the code for the procedure is generated, all of the space allocated since the MARK can be reclaimed using RELEASE (Fig. 2.c). The heap is then returned to its state before the declarations and statements of the procedure. All that remains is the procedure's name and descriptions of its parameters, which is only the information necessary to call that procedure from other points in the program.

With this short introduction to the workings of the compiler, one can see that two ways to make the 3000 stack stretch further would be to reduce the amount of global data (the space which remains during the entire compilation) and to keep down the size of individual procedures. Several approaches to the global problem will be presented first.

1. GLOBAL/EXTERNAL Compiler Options. In a normal program (neither \$GLOBAL\$ nor \$EXTERNAL\$ specified) the compiler assumes that all global variables are present and therefore assigns addresses to them in Primary DB. For all other compilation units which comprise the program, the same assumption is made. Obviously, if the compiler is assigning addresses for the global variables, they must all be declared in each compilation unit and in the same order to ensure that the addresses used are the same for each compilation.

This requirement is relaxed with the use of the \$GLOBAL\$ and \$EXTERNAL\$ compiler options. As before, the compilation unit which contains the outer block must have all of the program's global variables declared within it. The compiler option \$GLOBAL\$ is used for this compilation unit to tell the Pascal/3000 compiler that 1) all of the program's global variables are present, 2) the compiler should assign addresses to the global variables and 3) special information should be put into the USL file regarding these variables. This information consists of the name and address of every global variable. In the other compilation units of the program, marked with the \$EXTERNAL\$ option, the compiler does not assign addresses to the global variables. When it generates code which references a global variable, it doesn't use an address in the instruction; instead, it leaves the address part of the instruction blank and puts that instruction on a list with any other instructions which accessed that variable. For each global variable, the compiler puts into the USL file a special entry with the variable's name and a link to the list of its references. When the USL file is prepared into a program file, the MPE Segmenter gets the actual address of the variable from the information generated by the outer block compilation and uses it to fix up all of the instructions which actually reference that variable in the other compilation units. In this way, the binding between a variable's name (the ways it is referenced at the user level) and its address (the way it is referenced by the computer) is postponed until PREP time, allowing the compiler to relax the need to see all of the global variables in all of the compilation units.

Not only does this approach yield significant savings, it can also be implemented as needed on individual compilation units. All of the compilation units that comprise the Pascal/3000 compiler use $EXTERNAL$, but only those that have experienced heap or stack overflow problems include subsets of the global variables. Compilation units which have never had problems are still compiled using all of the global variables. This method was only used on the units that needed it as they needed it.

To implement this approach, each compilation unit should have a separate file for the global variables that it uses. Just include the global types and a file containing only the global variables used in the compilation unit (Fig. 3). This method requires that global variable names should be unique to 15 characters, a limit imposed by the MPE Segmenter.

```
        $EXTERNAL$
        PROGRAM CmpUnit1(Input,Output);

           {The global constants and types}
        $INCLUDE 'GlobTyps.Types.Account'$

           {Just the global variables that this compilation unit uses}
        $INCLUDE 'CmpUnit1.Vars.Account'$

           {Include the procedures for this compilation unit here}
         .
         .
         .

        BEGIN   {Empty outer block}
        END.    {Empty outer block}

    *** Now the outer block compile ***

        $GLOBAL$
        PROGRAM MainProg;

           {The global constants and types}
        $INCLUDE 'GlobTyps.Types.Account'$

           {All of the global variables}
        $INCLUDE 'GlobVars.Vars.Account'$

        BEGIN  {Actual outer block}

           {Code for the outer block}

        END.   {Actual outer block}
```

Fig. 3. Use of $GLOBAL$ and $EXTERNAL$ Compiler Options

2. External Procedure Declarations. Perhaps the greatest savings in stack space can be achieved by declaring external procedures only on the level used. Although the most natural place to declare external procedures may be the outer block, there is no requirement that they be so declared. (Even though not declared in the outer block, the compiler knows that they are external to the program.) External procedures may be declared on any level and the suggestion here is to declare them only at the level used. The source code for the Pascal/3000 compiler consists of about fifty separately compiled programs. During one stage of development, each of these compilation units had from 20 to 100 external procedures declared globally. An individual procedure rarely referenced more than five of these external procedures, yet space used to describe the name and parameters for all of procedures was

globally allocated since they were declared in the outer block. By moving these external procedure declarations inside the scope in which they were referenced, the Pascal/3000 team was able to reclaim a significant amount of space.

An example serves to illustrate the potential savings. Assume the following for a particular compilation unit:

- 100 external declarations

- 3 parameters per external declaration (average)

- 5 external procedures referenced by each level one procedure in the compilation unit (average)

- each procedure identifier requires 14 words of space

- each parameter identifier requires 8 words of space

The sizes for procedure and parameter identifiers are given in HP3000 16-bit words. The parameter specification in an average procedure declaration would then be

```
          3          *          8          =       24

        # of              space for             total
     parameters          a parameter          space used
                                            for parameters
```

Therefore, the amount of space consumed by a single external declaration is

```
            14         +      24           =       38

        space for         space for            total
     procedure id.       parameters          space used
```

If all of the external procedure declarations are declared in the outer block, the total space consumed is

```
       100       *       38             =       3800

       # of            space for               total
     externals       external decl.          space used
```

Now if externals are only declared at the level in which they are referenced, space consumption is

```
        5        *       38             =       190

       # of            space for               total
     externals       external decl.          space used
   per procedure
```

In this example, over 3600 words of stack space were saved by declaring the external procedures locally.

The value of this approach depends on the structure of one's program. If there exists a procedure in the compilation unit which references all of the externals used in the entire compilation unit (or if the entire compilation unit is a single procedure), this method will be of no benefit. Modularized programs with procedures oriented toward single functions will realize the most gain by declaring external procedures at the level in which they are used.

Another advantage of this approach towards saving space is that it can be applied to individual compilation units as needed. Removing all external declarations from the outer blocks of all the compilation units for the Pascal/3000 compiler would have been a large effort. Instead we only used this approach for those compilation units which were experiencing stack overflows during compilation.

Rather than imbed in-line declarations of external procedures into your code, it is suggested that you create one file for each external procedure header and place these files in a special group. (We called this group "EXT".) If you need to use an external procedure, then include the file which has the external procedure declaration in the procedure which calls it (Fig. 4). The viability of putting each external into its own file depends on your naming conventions. It was a natural manner in which to organize external declarations for the Pascal/3000 project, since each procedure in the compiler already resided in its own file in a special group for procedures (called "PROCS"). However it is done, localizing external procedure declarations can yield significant savings in stack space.

```
PROCEDURE Proc1 (Parm: Integer);

   VAR
      I : Integer;

   $INCLUDE 'InsertNu.Ext.Account'$
   $INCLUDE 'DeleteNu.Ext.Account'$

BEGIN  {Proc1}
   .
   .
   .
InsertNumber (I);   {an external procedure call}
   .
   .
   .
DeleteNumber (I);   {an external procedure call}
   .
   .
   .
END;   {Proc1}
```

Fig. 4. Local Declaration of External Procedures

3. Odds and Ends. Here a few general tips which the Pascal/3000 team learned during development. If your procedure or function spans more than 2 pages (without comments) you may try to break part of it up into one or more nested procedures. Procedures that are too long tend to be hard to read, are difficult to debug, take a lot of stack space to compile, and generally fail to follow structured programming guidelines. If a case element statement is long, make it a nested procedure to save space. If there is a large number of cases in the case statement, break it up into smaller ones. Put the more frequent case elements in the first part of the case statement; in the OTHERWISE part of the case statement, have a procedure call to a nested procedure which has the rest of the case elements.

That brings to a close the methods actually used in the development of the Pascal/3000 compiler. There are, however, several more ideas that will be presented in brief.

4. If you are compiling with the option $TABLES ON$ and are at the end of the procedure and either the code offsets or code listing has been generated and the identifier table has not yet been generated, you most likely have a stack overflow in the routines that generate the tables. As a short term solution, you can insert the compiler option $TABLES OFF$ before the procedure to proceed with the compilation. If you really want an identifier map, try one of the other solutions.

5. Divide! Are you including a large number of constants that you don't use? For example, if you have one file for all error numbers, you may want to break them up into smaller files based on their functionality.

6. Divide! Try separating global types into files which reflect their functional independence. Include only the files needed to compile an individual compilation unit. You may also have to divide up the source into smaller units to take advantage of this. Also, if a source has many level-1 procedures, try dividing the source into separate compilation units, since then there would be fewer procedure headers (procedure names and parameter lists) declared.

7. Run the compiler specifying "NOCB" in the RUN command; this moves part of the system information stored in the PCBX to an area outside the program's stack.

```
:RUN PASCAL.PUB.SYS;PARM=7;NOCB
```

This will save you a small amount of space immediately, but it does not really solve the problem. Try one of the other solutions for more permanent results.

8. Use integer constants instead of enumerated types and then only include the specific constant values that are used in any particular procedure. To then declare a variable of that "type", it is only necessary to include the constants that denote the starting and ending values of the range (Fig. 5).

```
PROGRAM Prog;

CONST
   FirstError = 0;
   LastError = 100;
TYPE
   ErrorNumber = FirstError..LastError;
.
.
.
PROCEDURE Proc1(Pointer : PtrType);
   CONST
      $INCLUDE 'ErrFile1.Const.Account'$
   VAR
       Error : ErrorNumber;
   BEGIN
   IF Pointer = NIL
   THEN
      Error := BadPointer;  {BadPointer is in ErrFile1}
   END;

BEGIN   {outer block}
.
.
.
END.    {outer block}
```

Fig. 5. Replacing Enumerated Types with Constants

This solution is recommended only as a last resort, as it asks the programmer to sacrifice the use of enumerated types, a very nice feature of Pascal. One of the other approaches should be tried first, since most of them maintain or improve the program's style and organization.

9. If one procedure is too big to compile and there is a good reason for not breaking it up into smaller procedures, you may try moving the procedure so that it is compiled earlier in the compilation unit. This will help a little since the declarations of the other procedures in the unit will be processed after the offending procedure.

Recommendations

Not all solutions cited here are appropriate for every project and every circumstance and stage of development. Below is a quick guide for finding a method to suit your particular situation.

If you are just starting development of a large system, consider using the following solutions from the outset: 1, 2 and 3.

If you are having a problem with just one procedure and want a quick solution to get it compiled, try one or more of the following: 3, 4, 7 or 9.

If you are interested in saving space on all of your compiles and are willing to spend some time restructuring your programs, try one or more of the following: 1, 2, 5, 6 or 8.

Summary

This paper has presented a number of ways that the stack size limitation can be avoided when using the Pascal/3000 compiler. It is the hope of the author that the approaches outlined here may help the Pascal programmer to more easily develop large systems on the HP3000.

Acknowledgments

I would like to thank Ron Smith for his help in compiling the data for this paper, Sue Kimura for her daily encouragement while it was being written and the Pascal/3000 project team, both past and present, for the opportunity to work and learn with them on the project.

References

MPE IV System Manager/System Supervisor Reference Manual, HP3000 Computer Systems, Hewlett-Packard Company, 1981, Part. No. 30000-90014.

Pascal/3000 Reference Manual, HP3000 Computer Systems, Hewlett-Packard Company, 1981, Part No. 32106-90001.

System Reference Manual, HP3000 Computer Systems, Hewlett-Packard, 1978, Part. No. 30000-90020.

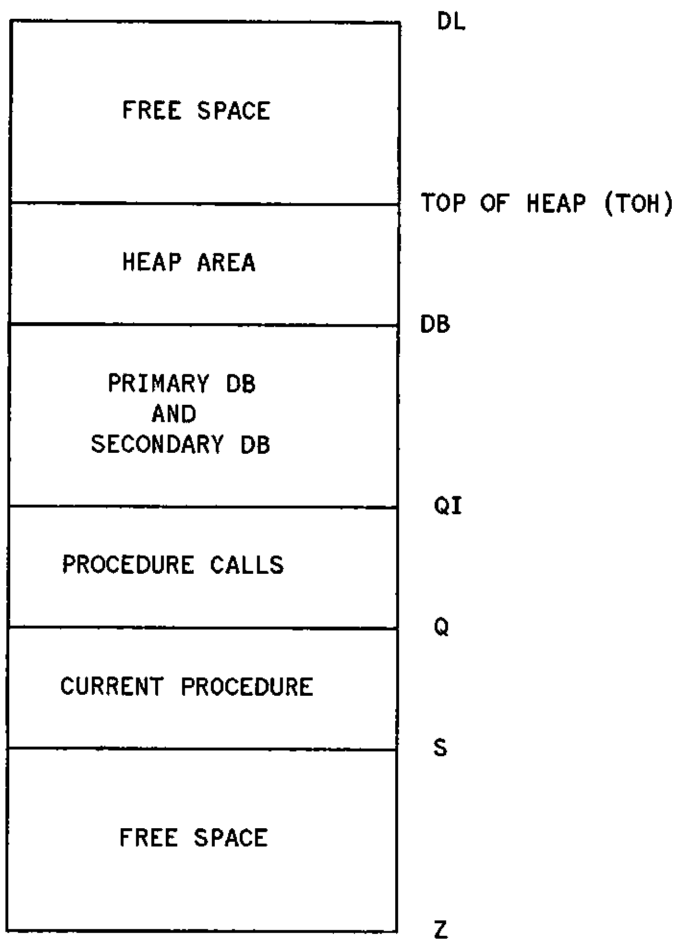
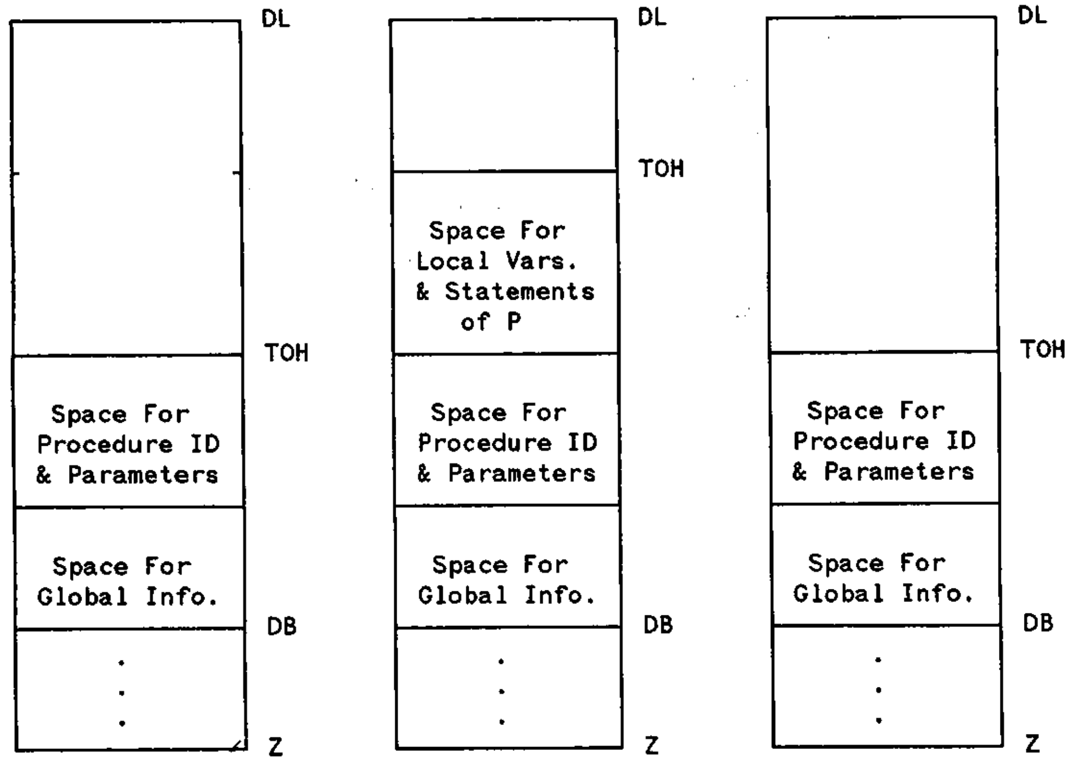


Fig. 1. Pascal/3000 Compiler Runtime Stack


DL
TOH
Space For Procedure ID & Parameters
Space For Global Info.
DB
Z
DL
TOH
Space For Local Vars. & Statements of P
Space For Procedure ID & Parameters
Space For Global Info.
DB
Z
DL
TOH
Space For Procedure ID & Parameters
Space For Global Info.
DB
Z

a. MARK saves Top_Of_Heap

b. After P processed

c. RELEASE after code generated

```
            PROGRAM Outerblock;

            VAR
               X, Y : Integer;

            PROCEDURE P ( Parm1, Parm2, Parm3 : Integer);
a. ->          VAR
                  Local1, Local2 : Integer;
               BEGIN  {Body of P}
               .
               .
               .
b. ->          END;   {Body of P}

c. ->       PROCEDURE Q;
               .
               .
               .
```

Fig. 2. Heap Activity of Pascal/3000 During Compilation

Biographical Sketch

Name : Christopher P. Maitz

Title : Member of Technical Staff

Employer : Hewlett-Packard Computer Languages Lab

Job Responsibilities : Involved in the development and enhancement of the Pascal/3000 compiler since 1979.

Education: BS, Mathematics, Carnegie-Mellon University

Marital Status : Married
