North American Response Centers

HP 3000 APPLICATION NOTE

PASCAL/3000 Using Dynamic Variables To Increase Stack Efficiency

HEWLETT PACKARD

August 15, 1986
Document P/N 5958-5824/2633

PASCAL/3000

Preventing Stack Overflows

With Dynamic Variables

Is the Bit Baron in your HP3000 complaining about your Pascal program's stack size? Does he say,

```
"ABORT  :program.group.acct.%seg_num.%code_offset

 PROGRAM ERROR #20 :STACK OVERFLOW",
```

after you say...

```
":RUN program.group.acct;MAXDATA=31232;NOCB"?
```

If so, then it is probably time to examine some means of reducing the stack memory required by your program.

One option, potentially very effective and available only to Pascal programmers, is the use of Dynamic Variables. Dynamic Variables offer a means of dynamically allocating, deallocating, and reallocating memory for GLOBALLY accessible data. Dynamic Variables are allocated in the logical "Heap", and in comparison to permanent global data, offer an enormous potential for stack memory savings. However, in order to insure that a given implementation of Dynamic Variables yields this savings, the PASCAL/3000 programmer needs to to take into account some of the specifics of Heap memory management. If this is not done, poor usage of Dynamic Variables can result in additional processing overhead with no memory savings at all.

The Pascal/3000 compiler implements the Heap by allocating memory for Dynamic Variables in the DL Area (DL to DB) of the user stack. The Heap begins with no memory allocation. As Dynamic Variables are initially allocated, Heap growth occurs away from DB and towards DL in descending memory locations.

PASCAL PROCESS STACK

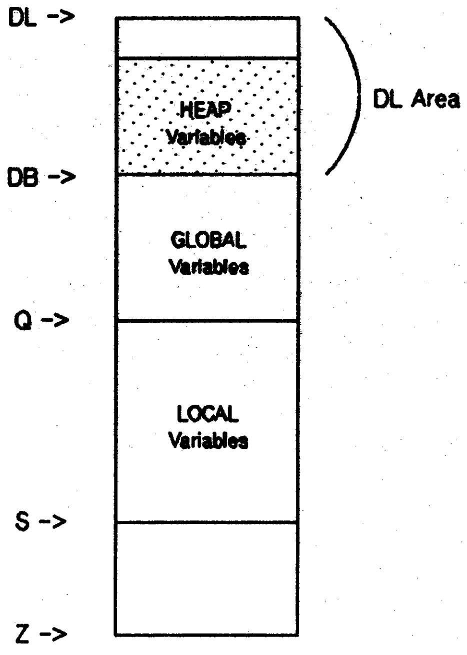

Fig. 1

When Heap growth causes the Heap limit to descend to DL, the DL Area is expanded to accommodate the larger Heap. However, the inverse is not always true. Contraction of the Heap will not always result in contraction of the DL Area. When the Heap is contracted and the DL Area is not, the memory released from the Heap is still counted in the DL Area.

This is very significant since the entire DL Area, not just the Heap within it, is included in the stack size that is counted against a given MAXDATA. In light of the fact that "Heap Compaction" efforts (to use the Manual's terminology) do not assure stack memory savings in and of themselves, the key to deriving memory savings from Dynamic Variables becomes one of programming for minimum DL Area size.

Pascal/3000 programmers can indirectly control the size of the DL Area through the Heap procedures they use for acquisition and disposition of Dynamic Variables. The two basic alternatives are the use of NEW & DISPOSE calls; or the use of NEW calls in conjunction with MARK and RELEASE calls. Additionally, the $HEAP_DISPOSE$ and $HEAP_COMPACT$ compiler options can alter the affect these procedures have on Heap memory.

The use of these procedures and compiler options will control two important aspects of Heap memory management. First, they will determine whether Heap memory previously returned through the return of variables is ever reused, and second, they will determine if the DL Area is ever contracted after expansion due to Heap growth. It is worth noting at this point that PASCAL/3000 never performs true "garbage collection" on the Heap (dynamic concatenation of allocated variables). Heap memory management is limited to the two operations just mentioned.

NEW

Calls to the NEW procedure will allocate enough memory for the variable defined by the pointer type plus an additional two bytes used to record the variable's length. With the exception of one case, the variable will be allocated at the Heap limit (the end of the Heap closest to DL). This pushes the Heap limit towards DL and often requires expansion of the DL Area.

When the $HEAP_DISPOSE ON$ compiler option is in effect, the NEW procedure will first search the Heap for an available "hole" left by the prior return of another Dynamic Variable. If a hole of sufficient size is found, the new variable will be allocated there instead of at the end of the Heap. This option is normally desirable if Dynamic Variables are being returned to the Heap through calls to the DISPOSE procedure.

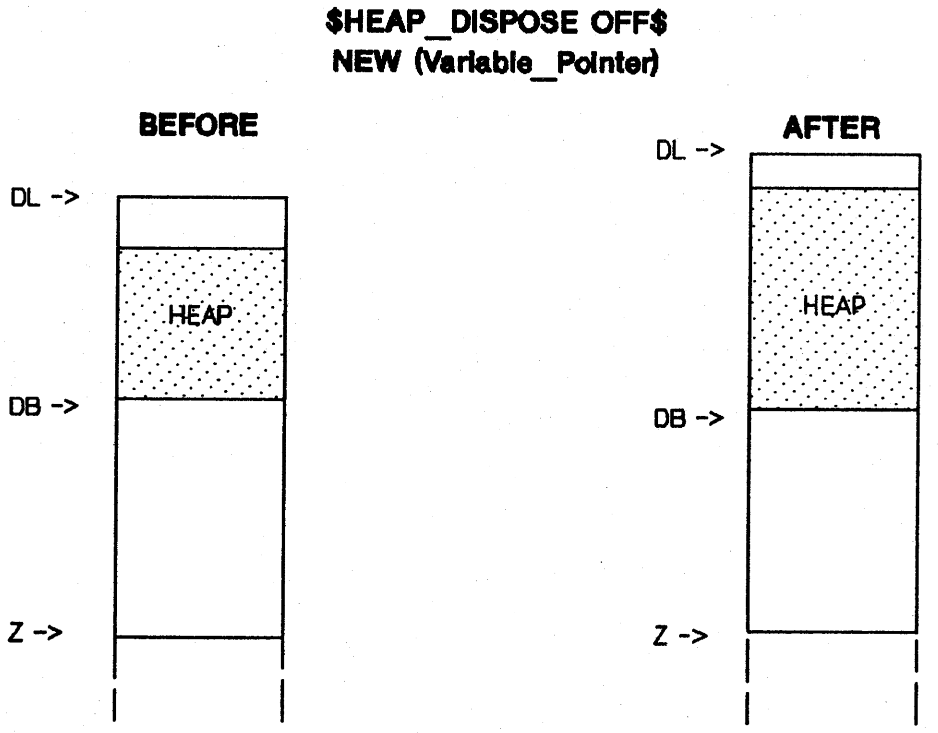

Fig. 2

DISPOSE

The DISPOSE procedure performs a logical return of the specified variable by resetting the pointer to it. This leaves the above mentioned "hole" in the heap of allocated variables.

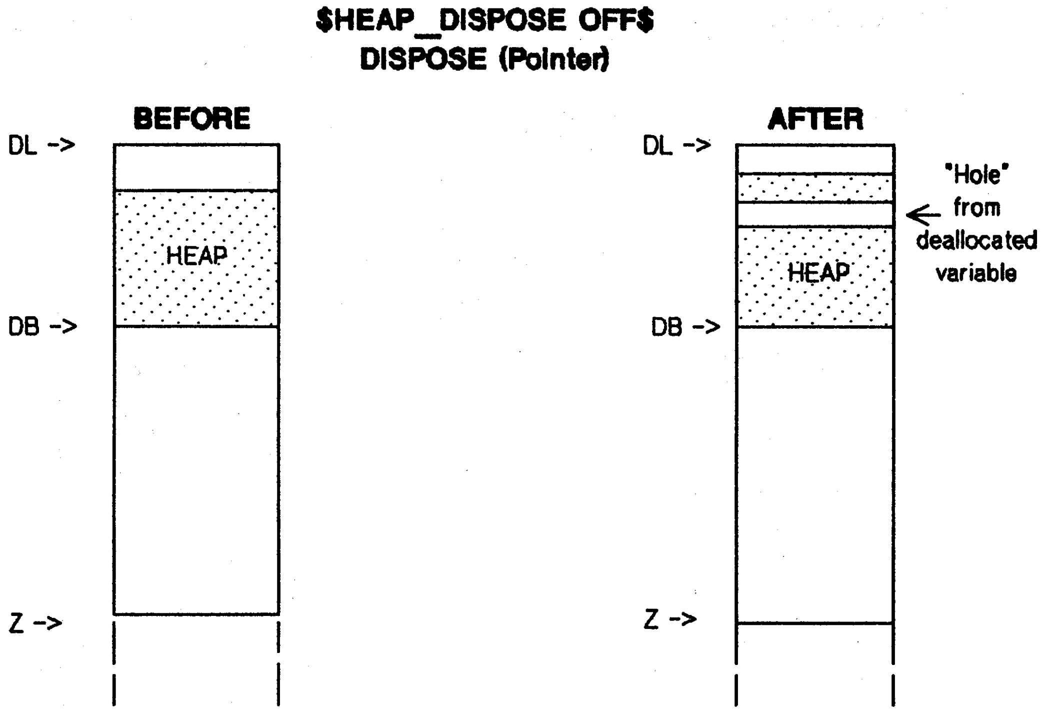

Fig. 3

If the $HEAP_COMPACT$ compiler option is in effect, and theDISPOSEd variable is at the Heap limit, then the Heap itself, but not DL, will eventually be contracted. This lopping off of the variable from the Heap can reduce the time necessary for "hole" searches by the NEW procedure, but does not free any memory from the DL Area.

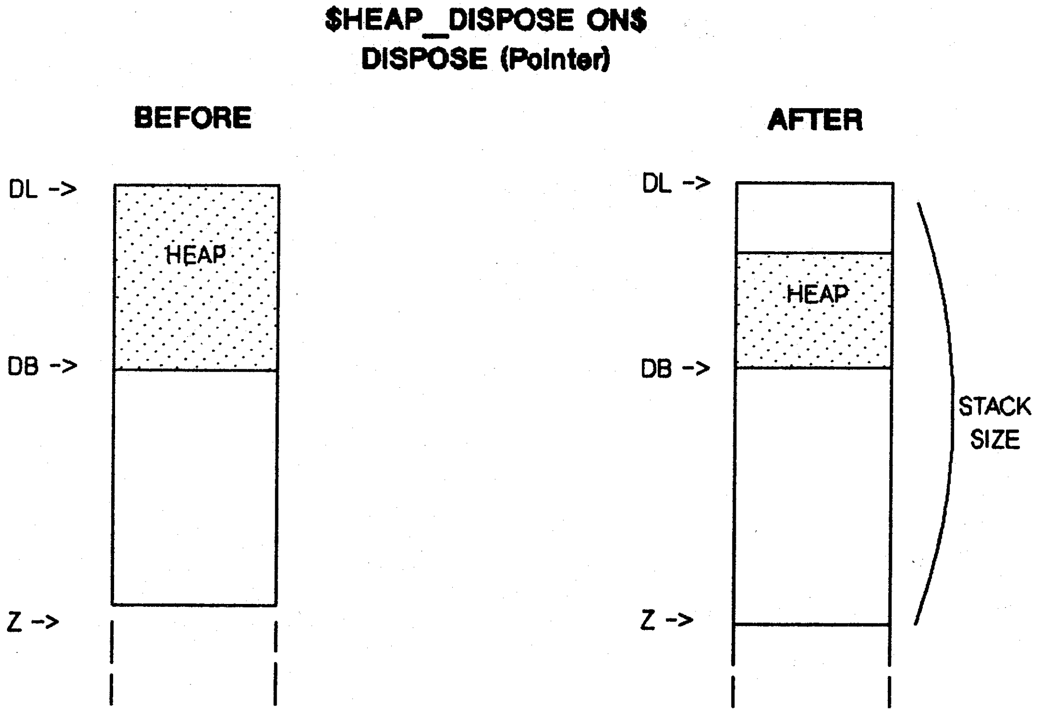

Fig. 4

RELEASE

As an alternative to the DISPOSE procedure, variables may be logically returned to the Heap through the use of MARK and RELEASE calls. When the RELEASE procedure is used to dispose of variables, both the HEAP and the DL Area will be contracted, or cut back, to the limit of the Heap at the time of the previous corresponding MARK call. This occurs regardless of the compiler options in effect. It is the ONLY Pascal procedure which will cause the DL Area to be contracted, and as such, has the greatest potential for stack memory savings.

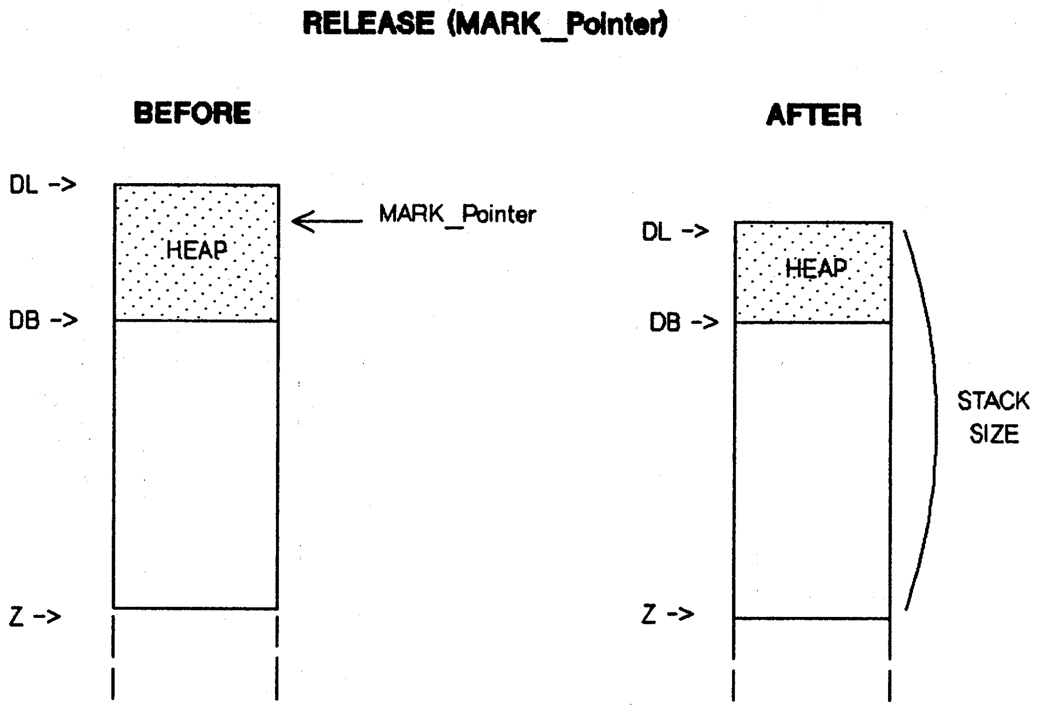

Fig. 5

As a result of its operation though, the RELEASE procedure is only appropriate for returning the most recently acquired Dynamic Variable(s). Additionally, there are three important warnings associated with the use of this procedure.

> First, any dynamic variables allocated between matching MARK and RELEASE calls are no longer accessible. If referenced after the RELEASE call, these variables can cause a program abort with PROGRAM ERROR #24: BOUNDS VIOLATION.
>
> The second warning involves the use of GETHEAP calls. Seldom used directly by user applications, GETHEAP is called by VPLUS/3000 whenever a VOPENFORMF is done. Care must be taken to insure that GETHEAP (or VOPENFORMF) calls are not bracketed by MARK/RELEASE calls unless corresponding RETURNHEAP (or VCLOSEFORMF) calls are included. Failure to do so will result in a program abort when the RELEASE call

attempts to contract DL over an allocated Getheap Area; (PASCERR 607) RELEASE PARAMETER ENCLOSES GETHEAP AREA.

And finally, the DLSIZE intrinsic should not be called if Dynamic Variables are allocated. This intrinsic can contract DL over allocated variables which presents the danger of program aborts with bounds violation.

There is also one important performance consideration associated with the use of Pascal/3000 Dynamic Variables. Expansion or contraction of the DL Area requires that the stack be swapped out to virtual memory and brought back in. This is a very high overhead operation. Expansions of DL can be minimized if an estimate of the eventual heap size can be calculated prior to program execution. This size can then be included in the DL= parm of the :PREP or :RUN for the program. Since contractions of the DL Area only result from RELEASE calls, RELEASE calls should be used as sparingly as possible.

IN REVIEW

To help clarify this "heap" of information, lets examine a few typical cases where effective use of Dynamic Variables can save stack memory.

Tables of information with capacities that need to vary between runs (i.e. product codes, personnel numbers, etc.) lend themselves to the use of the NEW procedure with the default compiler options. By allocating a variable for each table entry only as it is needed, the programmer can assure that tables never over allocate memory for the data present during any given run. The default compiler options ($HEAP_DISPOSE OFF;HEAP_COMPACT OFF$) eliminate the overhead associated with NEW searches for available "holes" in the Heap.

Data that need only be present during one phase of a process' execution (i.e. program initialization) can be allocated and then returned in mass through the use of the MARK and RELEASE procedures.

And last, but not least, is dynamic, random, data. A good example of this was seen in a monitor program that ran continuously and tracked information on active sessions. A very large record was needed for each session so long as it was active, yet it was not possible to predict how many records would be needed at any given time nor how long any given record would need to exist. This application successfully utilized the NEW and DISPOSE procedures with $HEAP_DISPOSE ON;HEAP_COMPACT ON$ to solve a reoccurring problem with stack overflows.