Part II

Managing Multiple Data Segments Under *Microsoft Windows*

The segment table provides a little-known way of managing multiple data segments

Tim Paterson and Steve Flenniken

In last month's installment, we presented a method for managing multiple data segments under MS Windows using a little-known Windows feature, the segment table, along with a library of macros and functions to assist in applying the technique. For this month's installment, we've prepared a sample Windows program called "segments" that demonstrates the *segtable* library. In its "random action" phase, it randomly allocates, reallocates, and frees global memory. A window displays statistics about each memory block, including its *pSeg* (the address of its *SegmentTable* entry), the current segment number, the previous segment number, and the number of times it has moved since it was allocated. A timer function is used to keep the window continuously updated, even when another application has the input focus.

The sample application in Listing One (page 106) uses one segment as the place to keep track of all the other segments that it fiddles with and displays in the window. That segment contains an array of structures, one for each additional segment. Because it is referenced so often, the macro *FARDATAP* is defined to return the far pointer to the first structure in this segment. Listings Two through Five (beginning on page 108) provide the rest of the files required by the application.

The menu bar is used to start and stop the random action mode. When on, the timer function picks one of the structures. If the structure does not yet have a *pSeg*, it allocates one with a random amount of memory. If it already has a *pSeg*, it will do one of three things: Reallocate the *pSeg* with a different memory size; free the data, but keep the *pSeg*; or free the *pSeg* altogether. Whenever a segment is allocated or reallocated, a text string containing the last action ("A" for allocate or "R" for reallocate) and the size of the segment (for example, "1484 bytes") is copied into the segment as its data. Whether the random action is on or off, the function checks to see if any of the segment numbers in the segment table have changed, and updates the display window if they have.

This sample is a useful demonstration in two ways. First, it has examples on how to code with the segment table. It includes many references to its far array of memory descriptor structures, and shows how IFP (indirect far pointer) parameters are passed to the functions *strcpyifp( )* and *strlenifp( )*. Second, it makes the segment table visible through a window so that its activity can be observed. As other applications are run (with random action stopped), you can see the effects as Windows keeps rearranging memory. Unless, of course, you are using LIM 4.0 EMS, which lets Windows just swap the data out without physically moving it.

Read-Only Data

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel, which is written in C and compiled into pcode, not native 8086 code. The pcode is

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980-82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold to Phoenix Technologies, the ROM BIOS maker. Steve formerly worked at Seattle Computer Products, Rosesoft (makers of ProKey), and is now with Microrim, working with OS/2 and Presentation Manager. Both can be reached c/o DDJ.

(continued from page 58)
data, not code, because it is never actually executed. Other applications could simply have large amounts of data in the form of tables or other structures.

Like code, read-only data should be marked as discardable in the linker definition file. This allows Windows to throw it away to make room, but reload it from disk later when needed. Another good practice is to keep segment size to less than 16K, the size of the LIM 3.2 expanded memory page frame. Windows can then choose to use space in EMS for those segments that fit, entirely transparent to the application.

Code and read-only data don't sound any different so far, but there is an important distinction. Windows keeps track of how often each code segment is used, in order to help it make a good decision on discarding one when it needs to free some memory. It does this with the reload thunk. Every far call to a discardable segment actually calls a thunk specific to that entry point. If the segment being called is present in memory, the thunk will contain a jump to the entry point. If the segment is not loaded, the thunk will cause Windows to load it. Either way, the thunk also notes the fact that a call to that segment was made. Windows uses a least-recently used (LRU) algorithm for determining the best segment to discard when memory is needed. The thunks are the source of its information.

The easiest way to deal with discardable read-only data segments is to put a little code in them. These lines of assembly language belong in each segment (but with a unique entry point name for each):

```
Load_This_Segment:
    mov     ax,cs
    retf
```

To ensure that a segment is loaded, and to find out where, call this entry point. The return value in *ax* is the segment of the data. The call to this entry point is, of course, actually a call to a thunk that ensures the segment is loaded.

The segment number returned by this call can be stuffed into an empty entry in *SegmentTable* so that it will stay updated in case of movement. But recall that this segment can also be discarded. In that case, Windows will update the segment table with the (even-numbered) handle for that segment. This complicates things a bit. Now we could make a reference to the segment table and find an even number, indicating that the segment we want has been discarded. Calling the entry point (the reload thunk) is the easiest way to bring the segment back.

Once the segment has been loaded, we can use it as much as we want, as long as Windows doesn't discard it. But if we never call the segment's entry point again, Windows will think we've stopped using it — after all, it's the calls through the thunk that keep track of usage. Without periodic calls to the entry point, this segment will be one of the first to be discarded, no matter how much we've actually been using it.

Fortunately, Windows provides a mechanism to remind us to call the entry point periodically. On a regular basis (typically every fourth timer tick, or 4.5 times per second), Windows performs an "LRU sweep." One of the things Windows will do during the LRU sweep is to fill part of our segment table with zeros. The number of words set to zero is specified in *SegmentTable[1]*; the zero fill starts at *SegmentTable[2]*. In addition, *SegmentTable[1]* itself is also set to zero, which means nothing will be zero-filled again until it is reset to some value. This use of *SegmentTable[1]* suggests using a macro to give it the name *cwClear*.

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel

The idea is to set aside the first portion of the segment table for read-only data. At every LRU sweep, Windows will zero fill the segment numbers that were stored in there. When we try to access a segment number that has been zeroed, we will see an even number and conclude it was discarded. Then we call the segment's entry point to reload it, and the thunk will record the activity. Hopefully, this will prevent the segment from being discarded while it is still needed. The overhead of zero filling the table and calling the entry point is quite small compared with the time to reload a segment from disk.

Note that the *segtable* library, as written, is not set up for this type of use. The non discardable data segments, such as *segDgroup*, must be moved above the zero-fill area in *SegmentTable*. Because there are a fixed number of read-only data segments, they would probably each have their own fixed

(continued from page 60)
segment table entry. New access macros would be required that could deal with a segment that was not present.

Debugging Considerations

Microsoft considers the ideal environment for running Windows to be a 80386 computer with extended memory running 386MAX by Qualitas of Bethesda, Maryland. 386MAX puts the computer into Virtual 8086 Mode and manages memory by using the 386's paging mechanism. It provides three important benefits for Windows. First, it fully emulates LIM 4.0 expanded memory (EMS). Second, it performs the same function as the Windows program HIMEM.SYS, making available the first 64K of extended memory for use by Windows. Third, it allows TSR programs such as mouse and network drivers to be loaded out of the way of conventional memory — the base 640K memory space.

When Windows finds itself loaded into a computer with LIM 4.0 EMS, and there's a fair amount (like 256K) of conventional memory left, it will use "large frame" EMS. This means that the base 640K memory space becomes part of the EMS page frame. Windows can then swap different logical memory pages into the base 640K.

While this is a great way to run Windows, especially when running several large applications, it's not so good for debugging with Symdeb. Symdeb seems to get confused by the EMS swapping, and we've gotten some very strange results. Now we always disable 386MAX whenever we will be debugging a Windows program with Symdeb. On the other hand, CodeView for Windows is apparently so large that Windows doesn't use large frame EMS. CodeView is so big that it requires EMS to run, and it works fine with 386MAX.

While my comments about EMS apply generally to Windows debugging, there is a booby trap specific to working with a segment table. (Naturally we're telling you this because it happened to us.) Recall that, during Windows' LRU sweep, *cwClear* (*SegmentTable[1]*) is used as a count of words in the segment table to zero fill. Should this word get accidently set through a programming error, unbelievably strange results can occur. A random value stored in *cwClear* will zero out a random amount of *DGROUP*, possibly including your stack. What makes this bug so nasty is that the LRU sweep is driven by the timer tick interrupt, so the data gets wiped out without you ever seeing how. Even a 386 hardware breakpoint will not necessarily catch it. (In our experience, the hardware breakpoint caught this bug when debugging with a serial terminal, but not when using a monochrome monitor.)

Extensions

As written, the *segtable* library and associated macros assume that the segments in the table are always present in memory. This is guaranteed by the fact that none of the segments in the table are marked as discardable. Except for *DGROUP*, they are all allocated by *SegmentAlloc( )*, which does not set the *GMEM_DISCARDABLE* flag.

If the use of the segment table was expanded to include read-only segments as discussed above, then there would be discardable segments in the table. An even value in a table entry would signify that that segment had been discarded. More complicated access macros would be needed to account for this possibility and to provide the mechanism to reload the segment. The macros could take one of two approaches. The first method would be to always call a near function for each segment reference, and that function would test for an even entry and perform the reload if needed. The alternative is to make the test for an even entry in line, and call a function only when reloading is necessary. In fact, having both of these forms available might be handy so that the speed/size tradeoff can be made on a case-by-case basis. It is likely that read-only segments would be used only in special ways, so that many segment table references could still assume the segment was always present and use the original, more efficient macros.

We have been describing the whole idea of the segment table as being suitable for large applications with multiple segments of data. There is, however, a limit on how much data a Windows program can have. Being nondiscardable, the data must be present in memory at all times. This usually limits an application to not more than 300K under the best conditions. Large frame EMS does not increase this limit, but it does allow each of several applications running simultaneously to have about as much data space as if they were running alone.

The problem is the 640K limit on conventional memory, and one possible answer is EMS. Windows will allow individual applications to control the small (LIM 3.2-style) EMS frame, which provides four 16K portholes into the EMS space. It is completely up to the application to manage its expanded memory, using interrupt *67H* to access EMS functions.

One way to go about this is to integrate EMS management with the mem-

ory management functions of the *segtable* library. Any data segment of less than 16K is a candidate for allocation in EMS instead of using *GlobalAlloc( ). SegmentAlloc( )* could be modified to do this, putting the EMS segment into the segment table and returning a *pSeg*. In this way, the use of EMS becomes completely transparent to the rest of the application.

There is, however, a serious drawback. Because there is space for only four EMS pages in the page frame, we can't allocate more than four pages before we run out of places to put them. Of course, the whole point of EMS is that we can have many megabytes of data, but we only need to use a few pages at any one time. Some of the EMS pages we allocate for data will have to be mapped out of the page frame — becoming momentarily inaccessible — so that others can be mapped in when we need them.

Fortunately, the segment table mechanism provides a handy way to do this. *pSeg*s are the handle by which the application can refer to any chunk of memory, whether conventional, accessible EMS, or inaccessible EMS. If the *pSeg* points to an odd-numbered value in the segment table, then that segment is present; if it points to an even-numbered value, then it is not present. This is exactly the same rule that is used for read-only data segments.

To take this approach, the application's EMS manager must ensure that EMS segments are odd. Whenever it must change the EMS map, it will have to update the segment table. When a page is mapped out, its segment number in the table must be found and replaced with an even-numbered marker. This marker must represent sufficient information to make the page accessible again. For example, 1 byte of the marker could represent an index into a table that includes the EMS handle, while the other byte is the logical page number. Remember that only 15 bits are available, because the least significant bit must be zero.

The access macros must understand how to deal with segments that aren't present, using the same general techniques as they would for read-only segments. However, the segment is "reloaded" by calling the EMS manager, instead of by calling a Windows reload thunk. The application's memory manager will need to have some reasonable way to decide which logical page to map out when a different one must be mapped in. One approach would be to approximate the LRU algorithm by discarding the least-recently mapped-in page. Then when two different segments, say A and B, are needed at the same time, this can be ensured by the sequence access-A, access-B, access-A. The second access-A is required because the access-B might have caused A to get mapped out. This could happen only if A was already present at the start, so that the first access-A did nothing.

Microsoft's own Windows applications use all of the techniques discussed here

To support cases when more than two segments were needed at once, a locking mechanism could be used. This would be similar to Windows' *GlobalLock( )* and *GlobalUnlock( )*, except that it would be handled by the application's memory manager. A streamlined alternative to making function calls for locking would be to set aside one or more special locations in the segment table. The presence of the segment in a special location would tell the memory manager not to map it out.

If the computer has no (or not enough) EMS, we can still do something to handle large amounts of data. By using the segment table and some additional help from Windows, we can set up a virtual memory system — that is, disk swapping. The key is to allocate memory with the Windows function *GlobalAlloc( )* by using the flags *GMEM_DISCARDABLE* and *GMEM_NOTIFY*. This tells Windows that it can discard the memory if it needs to, but to ask permission first. When Windows notifies the application that it would like to discard a segment, we can write that segment to disk first, then stick a marker for that segment in the segment table. As with EMS, the marker will represent the information needed to reload the segment the next time it is accessed.

The function that Windows will call to ask permission to discard a segment is set by using *GlobalNotify( )*. This function is documented in the Windows 2.0 SDK update booklet, with additional information in the Windows 2.1 SDK update. The function we register with Windows in this manner could be declared as:

```
BOOL FAR PASCAL
        NotifyProc(HANDLE hmem);
```

(continued from page 63)
The argument is supposed to be the handle of the segment being discarded. However, the Windows 2.1 SDK update says that in Version 2.03, it was actually the segment number, not the handle. This can be straightened out for both versions by calling *GlobalHandle( )*, which can take either the handle or segment number as its argument, and will return them both, as mentioned earlier.

NotifyProc( ) is a function in the application, but it must be in a fixed code segment. It will be called for each segment Windows would like to discard. If the application wants the segment locked, the function can return a false (zero) value and Windows will not discard it. The locking protocols could be the same as we suggested for EMS: Adding lock and unlock functions, and/or reserving special locations in the segment table. If the segment isn't locked, *NotifyProc( )* can write it to a disk file that has already been created for that purpose. Then it returns true and Windows will reclaim the space.

Any of these extensions — read-only data, EMS, disk swapping — may be combined. Using any one of them requires handling the case of segments that are not currently accessible. Once this jump has been made, the others can be added with little or no additional change to the main body of the application. Microsoft's own Windows applications use all of the techniques discussed here (a great deal of time was spent using Symdeb on Excel in preparing this article). While we haven't covered all of the procedures in detail, these ideas can be used to build Windows applications with virtually unlimited data capacity.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 106.)