Part I

# Managing Multiple Data Segments Under *Microsoft Windows*

*Segment tables can help you manage multiple data segments under MS Windows*

Tim Paterson and Steve Flenniken

If you've ever done any serious program development work for Microsoft Windows, you're probably aware of the freewheeling attitude Windows takes with memory management. In short, nothing stays put!

In this two-part article, we'll present a technique that helps you cope with this "memory movement" phenomenon by using a little-known Windows feature, the segment table. In this month's installment, we'll begin by defining the problem and identifying the solution, including a short library of macros and functions that will make the technique easy to use. Next month, we'll have a working sample application that demonstrates both the use of the segment table and our library, and provides a window (figuratively and literally) into the operation of the segment table while Windows is running.

The memory movement activity described above is particularly obvious in a Windows debugging session under Symdeb. Especially when Symdeb has been asked for full memory movement reporting (*/W3*), the memory movement/allocation/discard messages just spew out on the debugging screen. The purpose of all this activity is to utilize global memory as efficiently as possible. Each item being manipulated is a segment that was defined by the programmer either at compile and link time, or at run time. The offset of a particular item within one of these segments is not affected by movement of the segment. (Offsets within the local heap in the default data segment can change, but this is independent of global memory management.)

While these segments are being moved around, Windows keeps the values in *DS* and *SS* updated. *SS* is always the default data segment (*DGROUP*), and *DS* usually is, too. But *DS* could be any segment known to Windows, and if Windows moves that segment, *DS* will reflect that change. However, some segments (particularly code segments) are discardable; if *DS* is set to a segment that is later discarded, there will be no updating and no warning. Note that *ES* is never updated for segment movement.

Because *SS* and *DS* are kept current by Windows, near pointers into the stack or to static data are not a problem. Far pointers, on the other hand, seem to become invalid at every opportunity available to Windows. Fortunately, Windows does not interrupt your code and move things; any block of instructions that doesn't give control to Windows does not have to worry about memory movement for its duration. Windows gets control from a program (and possibly moves memory) in three ways. First, and most obvious, the program gives Windows control whenever it calls a Windows function. Second, almost any far call the program makes could also give control to Windows. Third, almost any far return might do it as well. These second and third cases occur because code segments are normally marked as discardable, and Windows has positioned itself to grab control and load any code segment that's not in memory when it gets called or returned to. This cannot happen when calling/returning into a fixed segment, or when calling/returning into the currently executing segment, since in each case the target segment is already present.

At face value, it would appear that you could never pass a far pointer as a parameter nor store it in a pointer variable,

*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. They can be reached c/o* DDJ.

because it wouldn't be valid by the time it got used. (One exception is far pointers to data in *DGROUP*, which will not be invalidated when passed to Windows functions.) The standard solution for seasoned Windows programmers is to lock the segments of the data down, so that they won't move. Locked segments, however, are blockages that choke up Windows' memory management and hurt performance. A well-behaved Windows application will lock no more than a few segments at a time, and then only for a short period. Seasoned Windows programmers know about all this stuff because they've all read Chapter 8 of Charles Petzold's *Programming Windows*.

There is another approach that is useful only for passing parameters between assembly language routines because it violates the C calling convention. A single far pointer can be passed by using *DS* to carry the segment, passing the offset in another register or on the stack. As mentioned already, Windows will keep *DS* updated while it moves memory around, as long as it doesn't point to a segment that gets discarded. There is, however, a restriction on the value in *DS* at the time of a far call from (not to) a discardable code segment. *DS* must contain a segment that is subject to Windows' memory management, that is, one that has a Windows handle associated with it. Segments outside of Windows' memory management would include low memory (such as *40H*, where BIOS data are stored) or expanded memory (say, at segment *E000H*) that is managed directly by the application. This restriction is needed should the caller's code segment be discarded. When this happens, the return address on the stack is patched so that it will transfer to code in Windows that reloads the caller. As part of this patching, the saved value of *DS* in the stack gets replaced by its handle. A handle does not require all 16 bits, and the extra bits made available by this replacement are used to store some of the segment reload and return information.

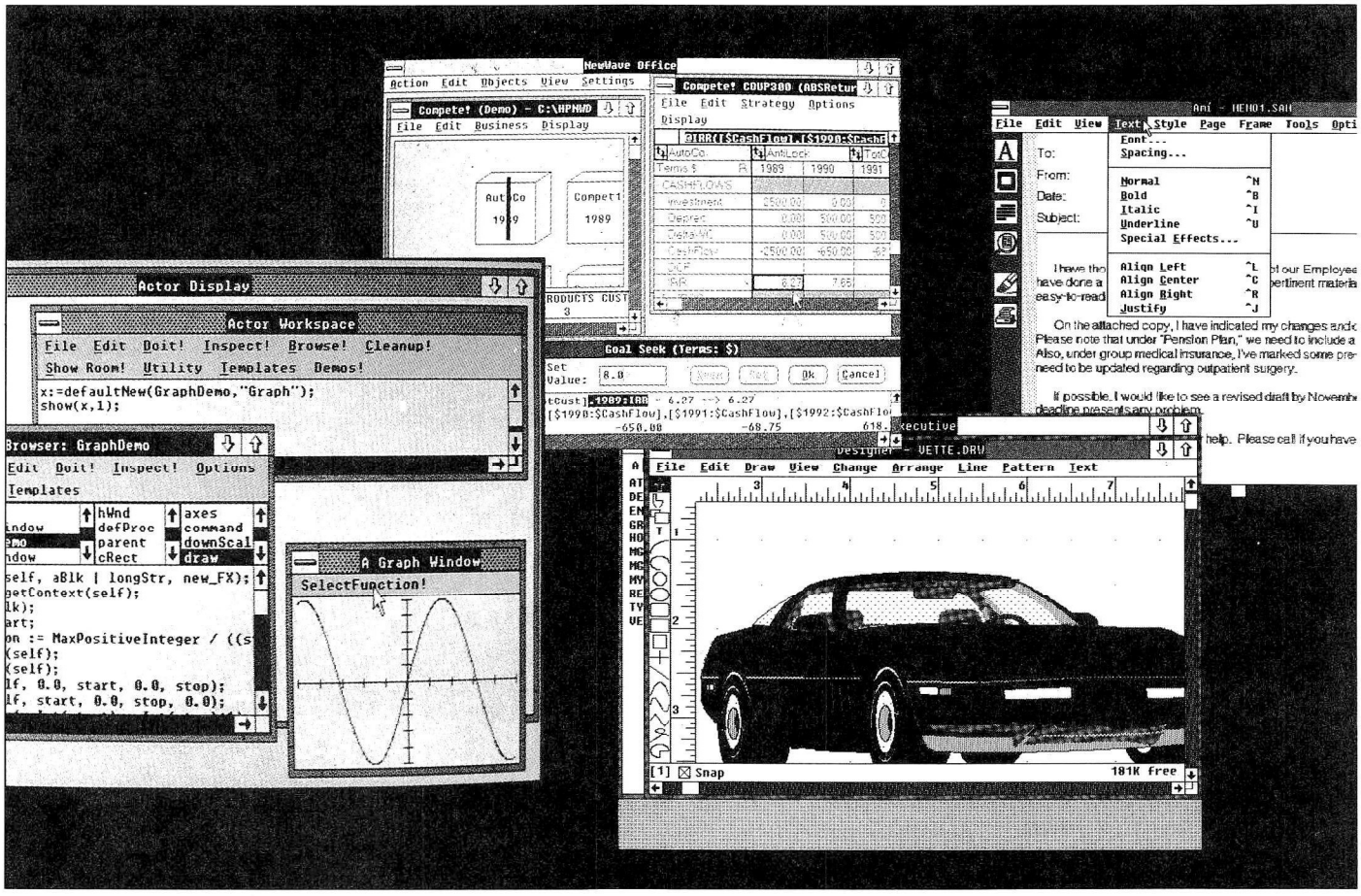
## Big Applications

Let's consider one of the big applications — spreadsheets, word processors, data base managers, and integrated programming environments — that might (someday) be run under Windows. Taking a spreadsheet program as an example, we can speculate on how its data segments might be organized.

First, each spreadsheet needs at least one data segment for its cell values and formulas. Considering the huge size that modern spreadsheet programs can theoretically attain (16,384 rows × 256 columns > 4 million cells), many segments of 64K could be required. After all, each cell will take many bytes to represent a formula or constant.

But besides a segment for cell information, there are other kinds of data tied to each spreadsheet. For example, names can be defined to refer to cells within the sheet, and you need to keep track of which segments constitute the cell data. This stuff might all go into a "control" data segment for each spreadsheet.

The spreadsheet program would allow several sheets to be loaded at once, each in their own window. Each spreadsheet loaded will require at least two data segments. The program-wide data, such as the list of information (e.g., name and path) about each loaded sheet, should probably go into its own segment, rather than into *DGROUP*. The rule we're following is that data that grow at the whim of the user — like loading more sheets — should not be crammed into *DGROUP*, as it already contains plenty of stuff. It's embarrassing to have a program report "out of memory" because *DGROUP* is full, when there is really 200K of global



memory left.

The point is, a big application will need a lot of data segments. And the data are mostly structures which will get passed around by reference. In other words, far pointers will be necessary everywhere. And Windows will be happy to move memory around and invalidate these pointers before they ever get used. Or the segments can be locked, and the application will turn into a memory pig.

### The Segment Table

A little-known alternative to this dilemma has been built into Windows specifically to help with these problems of multiple data segments. Because a description does not appear in the Windows Software Development Kit, we will create our own terminology in order to explain it. With this method, segments are never locked, so Windows has the freedom to manage memory for optimum efficiency. Segment references are passed around and stored indirectly, but can be dereferenced into a hard segment number instantly, without calling a Windows function.

The heart of the mechanism is the segment table. This is an array of words in *DGROUP* that the application program registers with Windows by calling the Windows function *DefineHandleTable(pSegTable)*. The words in the segment table are the segment numbers of all the data segments used by the application. Windows will keep these segment numbers updated and valid as memory is moved around.

To use this technique, the application must first set aside a static array of words in *DGROUP*. Source code in Microsoft C might look like that in Example 1.

The first two words in the segment table have special meaning, while the rest can all contain segment numbers. The length of the array is set to *MAXPSEGS + 2* to account for this. The array declaration assumes that the Small or Medium model is being used, which is typical of Windows applications. If the Compact or Large model is used, *NEAR* should be added to assure allocation in *DGROUP*.

The first word of the segment table, *cwSegs*, contains the number of segment entries in the table. This is the number of words Windows scans, looking for the segment it just moved. If it finds it, Windows will replace the old value in the table with the new location of the segment. Windows will continue to scan the table in case the segment appears in it more than once.

In the sample code in Example 1, an initializer could have been used in the declaration of *SegmentTable* to set *SegmentTable[0]* (*cwSegs*) to *MAXPSEGS (30)*. However, it's more efficient to make *cwSegs* only as large as needed to include all active entries in the table. For example, if only the first five entries in the segment table actually have segments it them, there is no point in having Windows scan 30 entries every time it moves memory. The initial value for *cwSegs* could be zero (as shown here), or it could be the smallest number of segments the application could need.

The application program's initialization code will register the table with Windows. For example:

```
void FAR PASCAL DefineHandleTable(WORD NEAR *);
                                        /* prototype */
. . .
DefineHandleTable(SegmentTable);
```

When this call is made, Windows will zero out *cwClear* and the following *cwSegs* entries in the table. The application can then start putting segment numbers in the table, which Windows will keep updated. The first segment the application will want in the table is *DGROUP* itself. Although the correct value of *DGROUP* is always maintained in segment register *SS* (and usually *DS*), it can be helpful to have it in the table as well, for reasons we'll explain later. This would require code like that in Example 2.

*Finding these empty entries is simple and involves no searching. Keeping* cwSegs *as small as possible is important for good performance, because Windows scans the whole table for every memory movement*

All data segments, besides the default data segment *DGROUP*, must be allocated with the Windows function *GlobalAlloc( )*. *GlobalAlloc( )* (Example 3) returns a handle, not a segment, and traditionally a call to *GlobalLock( )* is used to get the segment and lock it. We don't want it locked, however, so a call to *GlobalHandle( )* should be used instead. *GlobalHandle( )* is sort of a strange beast because its argument can be either a handle or a segment — it will figure out which. (This is actually pretty easy, because handles are always even numbers, and segments are always odd.) Given a handle or a segment, *GlobalHandle( )* returns both: The low word of the return value is the handle, and the high word is the segment.

### Using Fixed Table Entries

If we were building a big application such as a spreadsheet, we would probably always need some data segments in addition to *DGROUP*. Let's suppose we always have two additional data segments. One of them will store the full path name of each spreadsheet file that is currently loaded. The other segment will have some kind of "descriptor" for each loaded sheet, including the offset of its name in the first segment. Because these segments are always there, we'll assign them to the next two locations in *SegmentTable* after *DGROUP*.

```
#define segPathNames        SegmentTable[3]
#define segSheetDescr       SegmentTable[4]
```

```
/* WORD defined as unsigned short in windows.h */
#define MAXPSEGS 30  /* max # of table segments */
WORD SegmentTable[MAXPSEGS+2];
#define cwSegs  SegmentTable[0]
#define cwClear SegmentTable[1]
```

***Example 1:*** *Sample source code for setting aside the segment table*

```
#define segDgroup       SegmentTable[2]
segDgroup = HIWORD((void FAR *)&SegDgroup);
++cwSegs;  /* was zero, now 1 */
```

***Example 2:*** *Sample code for putting the group into the table*

```
if ( hMem = GlobalAlloc(flags, dwBytes) )
   SegmentTable[i] = HIWORD( GlobalHandle(hMem) );
else
   /* out of memory */
```

***Example 3:*** GlobalHandle( ) *converts a handle to a segment*

*(continued from page 18)*

Initialization procedures should call the Windows functions *GlobalAlloc( )* and *GlobalHandle( )* to assign values to these table entries. Note that the value in *cwSegs* (*SegmentTable[0]*) must be maintained at no less than the number of entries in the table being used. At the latest, *cwSegs* can be updated immediately after the segment number is stored in the table — before any function call or return is made. For example, after the first segment number of these two has been stored, *cwSegs* must be no less than 2; thus the first two entries in *SegmentTable* (*segDgroup* and *segPath Names*) will be kept updated as memory is moved during the function calls that allocate *segSheetDescr*. After *segSheet Descr* has been set, *cwSegs* must be no less than 3.

I've been saying "no less than" because it's OK if *cwSegs* is bigger than what's needed. In fact, the simplest approach for this example would be to initialize it to 3 in the declaration of *SegmentTable*. The effect of this is that Windows will scan three entries whenever it moves memory around to see if any of them are the segment it just moved. Because external data in C is initialized to zero, it will never find a match on the segment number, and nothing will happen. But it doesn't hurt even if garbage is present in the scanned part of *SegmentTable* — if a match is found, Windows will update that location with the new segment number. New garbage in place of the old — no problem, so long as *cwSegs* is still within the total size set aside for *SegmentTable*.

Some macros can be a big help in putting the segment table to use. The segment from the segment table must be combined with an offset into a long integer that is typecast to a far pointer as shown in Example 4. Another typecast is needed when the pointer is dereferenced to specify the type being pointed to. If there are many references to a particular type, a macro for that type can reduce some of the clutter as shown in Example 5.

This idea can be extended one step further. Some of the entries in the segment table have a dedicated meaning, which could be written into a macro. Suppose that there was a separate structure in the segment *segSheetDescr* for each spreadsheet that's been loaded. Because these structures can only exist in that one segment, the access macro can have that segment built in, as in Example 6.

In actual practice, you might want to copy a structure such as this into a fixed location in *DGROUP* to improve both the code size and the speed of repeated accesses. This particularly makes sense when there is a single "current" structure that is used heavily, which certainly applies to the spreadsheet case. Those structures that aren't current can still be accessed with the additional overhead.

```
/* HIWORD defined in windows.h */
#define FARPTR(off,seg) ((void FAR*)MAKELONG(off,seg))
*(WORD FAR *) FARPTR(0,segSheetDescr) = 0;
```

***Example 4:*** *Macro for creating a file pointer*

```
#define FARWORD(off,seg) (*(WORD FAR*)FARPTR(off,seg))
FARWORD(0,segSheetDescr) = 0;     /* no sheets loaded */
```

***Example 5:*** *Macro for referencing a particular type*

```
#define FARDESCR(off) (*(DESCR FAR*)FARPTR
                                (off,segSheetDescr))
typedef struct  /* descriptor for a spreadsheet */
{
          unsigned short              SheetId;
          char                        *SheetName;
          ...
} DESCR;

FARDESCR(pCurSheet).SheetId = NextSheetNum++;
pCurSheetName = FARDESCR(pCurSheet).SheetName;
```

***Example 6:*** *Building in a segment*

```
if ( hMem = GlobalAlloc(flags, dwBytes) )
{
   pSeg = &SegmentTable[++cwSegs + 1];
   *pSeg = HIWORD( GlobalHandle(hMem) );
}
else
   /* out of memory */
```

***Example 7:*** *Allocating a segment*

```
typedef unsigned long IFP;
#define MAKEIFP(off,pseg) ((IFP)MAKELONG(off,pseg))
#define IFP2SEG(ifp) ( *(WORD NEAR *)HIWORD(ifp))
#define IFP2PTR(ifp) FARPTR(LOWORD(ifp),IFP2SEG(ifp))
```

***Example 8:*** *Defining a new data type*

```
IFP memcpyifp(
  MAKEIFP(&CurSheetBuf,&segDgroup), /* destination */
  MAKEIFP(pCurSheet,&segSheetDescr), /* source */
  sizeof(DESCR) /* bytes to copy */
 );
```

***Example 9:*** *A call to a general-purpose block copy routine*

### Variable *pSegs*

So far, we have only talked about a few dedicated entries in the segment table. But a big application will need to grab and release new data segments on the fly. For example, earlier we proposed that each spreadsheet could need two segments, and any number of spreadsheets could be loaded at once. We need a method to allocate new entries in the segment table as spreadsheets are loaded.

There's no trick to this part. Windows doesn't care which entries in the segment table we use. One simple-minded approach would be to use *cwSegs* to keep track of the next free entry. For example, if *cwSegs* is 5, then *SegmentTable[2]* through *SegmentTable[6]* are in use. *SegmentTable[7]* would be the next free entry, and *cwSegs* would need to be incremented when it's allocated, as shown in Example 7.

The final result of the allocation is *pSeg*, which is a pointer to a segment. This replaces the Windows handle as the way to remember a segment. If you wanted to set the first word in a newly allocated segment to zero, you could use

```
FARWORD(0,*pSeg) = 0;
```

Now that we're using *pSegs*, we'll need a new data type to represent a *pSeg* combined with an offset. Let's call it an "Indirect Far Pointer," or IFP. IFP is a 32-bit quantity, with an offset in the low word and a near pointer to a segment in the high word, as in Example 8.

The IFP is the quantity that will be passed as a parameter to other functions in the program. Note, however, that the IFP cannot be passed to Windows nor to any standard C libraries. Example 9 shows an example of a call to a general-purpose block copy routine that uses IFPs to specify the source and destination. Except for using IFPs, this function has the same inputs, outputs, and purpose as the standard Microsoft C library routine *memcpy( )*. The first argument is the destination, in this case the static variable *CurSheetBuf* of type *DESCR*, which holds the whole descriptor for the "current" spreadsheet. Now you can see why an entry in the

*(continued from page 20)*
segment table is reserved for *DGROUP*, even though its current value can always be found in segment register *SS*. Because the copy routine is general purpose, it must always be passed IFPs, which are *offset/pSeg* pairs. In order to pass a *DGROUP* variable, such as *CurSheetBuf*, we must be able to pass a pointer to a place where *DGROUP* is stored.

```
IFP memcpyifp(IFP DestIfp, IFP SourceIfp, unsigned cb)
{
    unsigned i;
    for (i=0; i<cb; i++)
              ((char FAR *)IFP2PTR(DestIfp))[i] =
                ((char FAR *)IFP2PTR(SourceIfp))[i];
    return(DestIfp);
}
```

***Example 10:*** *One way to code* memcpyifp( )

```
          char FAR *Dest;
          char FAR *Source;

          Dest = IFP2PTR(DestIfp);
          Source = IFP2PTR(SourceIfp);
          for (i=0; i<cb; i++)
                        Dest[i] = Source[i];
```

***Example 11:*** *Storing pointers in local variables*

```
IFP memcpyifp(IFP DestIfp, IFP SourceIfp, unsigned cb)
{
   movedata( IFP2SEG(SourceIfp), /* source seg */
   LOWORD(SourceIfp),     /* source offset */
   IFP2SEG(DestIfp),      /* destination seg */
   LOWORD(DestIfp),       /* destination offset */
   cb                     /* count of bytes */
  );
 return(DestIfp);
}
```

***Example 12:*** *Writing* memcpyifp( ) *using a standard library function*

In this example, the source of the copy is the descriptor that is being made current. *MAKEIFP* builds the reference for the source by using the offset held in the static variable *pCurSheet*, and the dedicated *pSeg segSheetDescr*. Note how the "address of" operator "&" applied to *segDgroup* or *segSheetDescr* returns a *pSeg*; leaving it off would fetch the segment value itself from *SegmentTable*.

One way to code *memcpyifp( )* is shown in Example 10. Unfortunately, the Microsoft C compiler is not as good at eliminating loop invariants as we might like. Even at maximum optimization (*/Ox*), the generated code completely rebuilds both far pointers — and fetches the segments from the segment table — each time through the loop. A slight variation, then, would be to bring the building of the far pointers out of the loop and store them in a couple of local variables, such as those shown in Example 11.

The code generated for this sequence is pretty good, and ends with the block move instruction *REP MOVSW* to actually perform the copying. It is critical, however, to remember the limitations of applying this technique to other cases. Once an IFP has been dereferenced, there is no longer any protection from global heap movement. Almost any far call can be assumed to invalidate far pointer variables. An exception to this rule is when far calls are made to functions in the same segment; because the segment is already loaded, no memory movement will occur. This allows us to write "helper" functions to access standard C library routines. The helper must reside in the same segment as the run time, and converts the IFPs passed to it into far pointers for the library

*(continued from page 22)*
function. As shown in Example 12, *memcpyifp( )* could be written using the standard Microsoft C library function *movedata( )*, which takes explicit segment and offset arguments.

For the best performance and smallest size, key functions can be coded in assembly language. Listing One, page 89, shows an assembly-language version of *memcpyifp( )*, demonstrating the dereferencing of IFPs.

## The *segtable* Library

Listing Two, page 89, shows the *segtable* library, a collection of five functions that manage memory through a segment table. The header file segtable.h (Listing Three, page 90) includes all the function prototypes, type definitions, and macro definitions needed to use the library. The first few lines of segtable.h are constants that should be tailored to the application: MAXPSEGS is the maximum number of segments that the segment table can hold. You want this large enough so you never run out, but you don't want to eat up too much of *DGROUP* with the table. Microsoft Excel, for example, apparently has room for more than 2000 segments. *MINPSEGS* is the initial value for *cwSegs*. It must be equal to the number of fixed table entries — at least 1, for *segDgroup*.

Immediately following these constants are the definitions of the fixed table entries. *segDgroup* is always there as *SegmentTable[2]*, and any additional entries must follow with consecutive indices.

The header file also defines a pair of validation macros for debugging purposes. *VALIDPSEG(pseg)* verifies that its *pSeg* argument points into the segment table and is word-aligned (even). *VALID_VARIABLE_PSEG(pseg)* adds the additional requirement that the *pSeg* point above the fixed table entries, into the variable part of the table. The library routines make these checks whenever possible if the *DEBUG* switch is turned on. Also, whenever a segment table entry contains a valid segment number, there are checks to verify that the entry is odd (as are all Windows segment numbers). If any of these debugging tests fails, the library will call the function *SegmentError( )*.

The five library functions are described in the following section.

***void FAR PASCAL SegmentInit(void)*** — This function registers the segment table with Windows and initializes the *segDgroup* entry. All other fixed table entries will have a zero value, which is appropriate for allocating with *SegmentRealloc( )*.

***PSEG FAR PASCAL SegmentAlloc(DWORD size)*** — First, this function must find an unused entry in the segment table. It does not use the simple method described earlier; instead, it keeps a free list that links together all entries that were once used but have been freed. A static variable points to the head of the list. The segment table is located at an even address, so all the links in the free list will also be even. This means that when Windows scans the table for a segment it is moving, it will never match with a link because the segments are always odd. If the free list is empty, then *cwSegs* is increased and the next entry at the end of the table is taken.

This approach to finding unused entries ensures that *cwSegs* is never increased if there is already a free slot within the in-use range. Finding these empty entries is very simple and involves no searching. Keeping *cwSegs* as small as

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

**EDITORIAL**
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rosalie Cooke, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

**ART/PRODUCTION**
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TECHNICAL ILLUSTRATOR *Linda Ann Clark*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

**CIRCULATION**
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

**ADMINISTRATION**
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

**MARKETING/ADVERTISING**
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

**M&T PUBLISHING INC.**
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

***DR. DOBB'S JOURNAL*** (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

**ARTICLE SUBMISSIONS:** Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

**SUBSCRIPTION:** $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 for surface mail to all addresses out of the U.S.; add $33 for airmail to Canada and Mexico; or $32 for airlift to all other countries.

**POSTMASTER:** Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

**CUSTOMER SERVICE:** For subscription orders and changes of address call toll-free 800-456-1215 or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For subscription questions outside the U.S. call 1-303-447-9330. For book/software orders call 800-533-4372 (in California 800-356-2002).

**FOREIGN NEWSSTAND DISTRIBUTOR:** Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.





*(continued from page 24)*

possible is important for good performance, because Windows scans the whole table for every memory movement, which happens quite frequently. An additional step could be to detect when an entry at the end of the table is being freed, so *cwSegs* could be reduced, but that is probably more work than it's worth.

Once an empty entry is found, Windows is called to allocate the amount of memory requested. The *GMEM_MOVEABLE* flag is always used. If the application requires it, the function could be modified to accept the memory flags as an argument. If all goes well, the function returns a *pSeg*, a pointer to the entry in the table; zero returned indicates a failure.

***void FAR PASCAL SegmentFree(PSEG pseg)*** — This function frees the data associated with the *pSeg* argument, and frees the table entry by placing it on the free list. This function should not be called on a fixed table entry, because it will make that entry available for general use.

***void FAR PASCAL DataFree(PSEG pseg)*** — This function frees the data associated with the *pSeg* argument, but the *pSeg* itself is still reserved. Its table entry will contain zero. *SegmentRealloc* may be used later to allocate some memory to the *pSeg*.

***BOOL FAR PASCAL SegmentRealloc(PSEG pseg, DWORD size)*** — This function changes the amount of memory allocated to a *pSeg*. It can be used on any valid *pSeg*, whether or not any data are currently allocated to it. This function is used for the initial allocation of fixed table entries (except *segDgroup*).

***void FAR PASCAL SegmentError (void)*** — This function is present only when the *DEBUG* switch is on. It is the central exit point should any of the debugging checks fail. As written, it does nothing but *FatalExit(-1)*. The easiest way to use it is to simply put a breakpoint there, so you can do a stack backtrace as the first step to figuring out what went wrong. But it could also be spruced up to report an error without requiring a debugger to be present.

## Next Month

By now, you should have a good understanding of the problems of managing multiple data segments under Windows, not to mention how a segment table can help you come to terms with these problems. Next month, we'll introduce a sample program, called SEGMENTS, that demonstrates the use of the *segtable* library and lets you watch as Windows updates the segment table before your eyes.

## 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 89)**