Chapter 8

Speeding Up C with Assembly Language

Chapter 8

Jumping Languages When You Know It'll Help

When I was a senior in high school, a pop song called "Seasons in the Sun," sung by one Terry Jacks, soared up the pop charts and spent, as best I can recall, two straight weeks atop *Kasey Kasem's American Top 40.* "Seasons in the Sun" wasn't a particularly good song, primarily because the lyrics were silly. I've never understood why the song was a hit, but, as so often happens with undistinguished but popular music by forgotten one- or two-shot groups ("Don't Pull Your Love Out on Me Baby," "Billy Don't Be a Hero," *et al.*), I heard it everywhere for a month or so, then gave it not another thought for 15 years.

Recently, though, I came across a review of a Rhino Records collection of obscure 1970s pop hits. Knowing that Jeff Duntemann is an aficionado of such esoterica (who do *you* know who owns an album by The Peppermint Trolley Company?), I sent the review to him. He was amused by it and, as we kicked the names of old songs around, "Seasons in the Sun" came up. I expressed my wonderment that a song that really wasn't very good was such a big hit.

"Well," said Jeff, "I think it suffered in the translation from the French."

Ah-ha! Mystery solved. Apparently everyone but me knew that it was translated from French, and that novelty undoubtedly made the song a big hit. The translation was also surely responsible for the sappy lyrics; dollars to donuts that the original French lyrics were stronger.

Which brings us without missing a beat to this chapter's theme, speeding up C with assembly language. When you seek to speed up a C program by converting selected parts of it (generally no more than a few functions) to assembly language, make sure you end up with high-performance assembly language code, not fine-tuned C code. Compilers like Microsoft C/C++ and Watcom C are by now pretty good at fine-tuning C code, and you're not likely to do much better by taking the compiler's assembly language output and tweaking it.

To make the process of translating C code to assembly language worth the trouble, you must ignore what the compiler does and design your assembly language code from a pure assembly language perspective. With a merely adequate translation, you risk laboring mightily for little or no reward.

Apropos of which, when was the last time you heard of Terry Jacks?

Billy, Don't Be a Compiler

The key to optimizing C programs with assembly language is, as always, writing good assembly language code, but with an added twist. Rule 1 when converting C code to assembly is this: *Don't think like a compiler.* That's more easily said than done, especially when the C code you're converting is readily available as a model and the assembly code that the compiler generates is available as well. Nevertheless, the principle of not thinking like a compiler is essential, and is, in one form or another, the basis for all that I'll discuss below.

Before I discuss Rule 1 further, let me mention rule number 0: *Only optimize where it matters.* The bulk of execution time in any program is spent in a very small portion of the code, and most code beyond that small portion doesn't have any perceptible impact on performance. Unless you're supremely concerned with code size (an area in which assembly-only programs can excel), I'd suggest that you write most of your code in C and reserve assembly for the truly critical sections of your code; that's the formula that I find gives the most bang for the buck.

This is not to say that complete programs shouldn't be *designed* with optimized assembly language in mind. As you'll see shortly, orienting your data structures towards assembly language can be a salubrious endeavor indeed, even if most of your code is in C. When it comes to actually optimizing code and/or converting it to assembly, though, do it only where it matters. Get a profiler—and use it!

Also make it a point to concentrate on refining your program design and algorithmic approach at the conceptual and/or C levels before doing any assembly language optimization.

Assembly language optimization is the final and far from the only step in the optimization chain, and as such should be performed last; converting to assembly too soon can lock in your code before the design is optimal. At the very least, conversion to assembly tends to make future changes and debugging more difficult, slowing you down and limiting your options.

Don't Call Your Functions on Me, Baby

In order to think differently from a compiler, you must understand both what compilers and C programmers tend to do and how that differs from what assembly language does well. In this pursuit, it can be useful to examine the code your compiler generates, either by viewing the code in a debugger or by having the compiler generate an assembly language output file. (The latter is done with /Fa or /Fc in Microsoft C/C++ and -S in Borland C++.)

C programmers tend to modularize their code with lots of function calls. That's good for readable, reliable, reusable code, and it allows the compiler to optimize better because it can deal with fewer variables and statements in each optimization arena—but it's not so good when viewed from the assembly language level. Calls and returns are slow, especially in the large code model, and the pushes required to put parameters on the stack are expensive as well.

What this means is that when you want to speed up a portion of a C program, you should identify the entire critical portion and move *all* of that critical portion into an assembly language function. You don't want to move a part of the inner loop into assembly language and then call it from C every time through the loop; the function call and return overhead would be unacceptable. Carve out the critical code *en masse* and move it into assembly, and try to avoid calls and returns even in your assembly code. True, in assembly you can pass parameters in registers, but the calls and returns themselves are still slow; if the extra cycles they take don't affect performance, then the code they're in probably isn't critical, and perhaps you've chosen to convert too much code to assembly, eh?

Stack Frames Slow So Much

C compilers work within the stack frame model, whereby variables reside in a block of stack memory and are accessed via offsets from BP. Compilers may store a couple of variables in registers and may briefly keep other variables in registers when they're used repeatedly, but the stack frame is the underlying architecture. It's a nice architecture; it's flexible, convenient, easy to program, and makes for fairly compact code. However, stack frames have a few drawbacks. They must be constructed and destroyed, which takes both time and code. They are so easy to use that they tend to bias the assembly language programmer in favor of accessing memory variables more often than might be necessary. Finally, you cannot use BP as a general-purpose register if

you intend to access a stack frame, and having that seventh register available is sometimes useful indeed.

That doesn't mean you shouldn't use stack frames, which are useful and often necessary. Just don't fall victim to their undeniable charms.

Torn Between Two Segments

C compilers are not terrific at handling segments. Some compilers can efficiently handle a single far pointer used in a loop by leaving ES set for the duration of the loop. But two far pointers used in the same loop confuse every compiler I've seen, causing the full segment:offset address to be reloaded each time either pointer is used.

This particularly affects performance in 286 protected mode (under OS/2 1.X or the Rational DOS Extender, for example) because segment loads in protected mode take a minimum of 17 cycles, versus a mere 2 cycles in real mode.

In assembly language you have full control over segments. Use it, and, if necessary, reorganize your code to minimize segment loading.

Why Speeding Up Is Hard to Do

You might think that the most obvious advantage assembly language has over C is that it allows the use of all forms of instructions and all registers in all ways, whereas C compilers tend to use a subset of registers and instructions in a limited number of ways. Yes and no. It's true that C compilers typically don't generate instructions such as **XLAT**, rotates, or the string instructions. On the other hand, **XLAT** and rotates are useful in a limited set of circumstances, and string instructions *are* used in the C library functions. In fact, C library code is likely to be carefully optimized by experts, and may be much better than equivalent code you'd produce yourself.

Am I saying that C compilers produce better code than you do? No, I'm saying that they *can*, unless you use assembly language properly. Writing code in assembly language rather than C guarantees nothing.

You can write good assembly, bad assembly, or assembly that is virtually indistinguishable from compiled code; you are more likely than not to write the latter if you think that optimization consists of tweaking compiled C code.

Sure, you can probably use the registers more efficiently and take advantage of an instruction or two that the compiler missed, but the code isn't going to get a whole lot faster that way.

True optimization requires rethinking your code to take advantage of assembly language. A C loop that searches through an integer array for matches might compile

A. What the compiler outputs:

```
LoopTop:
      mov   ax,[bp-8]   ;Get the searched-for value
      cmp   [di],ax     ;Is this a match?
      jz    Match       ;Yes
      add   di,2        ;No, advance the pointer
      dec   si          ;Decrement the loop counter
      jnz   LoopTop     ;Continue if there are more data points
```

B. Removing stack frame access:

```
LoopTop:
      lodsw             ;Get the next array value
      cmp   ax,bx       ;Does it match the searched-for value?
      jz    Match       ;Yes
      loop  LoopTop     ;No, continue if there are more data points
```

Tweaked compiler output for a loop.
Figure 8.1

to something like Figure 8.1A. You might look at that and tweak it to the code shown in Figure 8.1B.

Congratulations! You've successfully eliminated all stack frame access, you've used **LOOP** (although **DEC SI/JNZ** is actually faster on 386 and later machines, as I explained in the last chapter), and you've used a string instruction. Unfortunately, the new code isn't going to run very much faster. Maybe 25 percent faster, maybe a little more. Big deal. You've eliminated the trappings of the compiler—the stack frame and the restricted register usage—but you're still *thinking* like the compiler. Try this:

```
repnz scasw
jz    Match
```

It's a simple example—but, I hope, a convincing one. Stretch your brain when you optimize.

Taking It to the Limit

The ultimate in assembly language optimization comes when you change the rules; that is, when you reorganize the entire program to allow the use of better assembly language code in the small section of code that most affects overall performance. For example, consider that the data searched in the last example is stored in an array of structures, with each structure in the array containing other information as well. In this situation, **REP SCASW** couldn't be used because the data searched through wouldn't be contiguous.

However, if the need for performance in searching the array is urgent enough, there's no reason why you can't reorganize the data. This might mean removing the array elements from the structures and storing them in their own array so that **REP SCASW** *could* be used.

Organizing a program's data so that the performance of the critical sections can be optimized is a key part of design, and one that's easily shortchanged unless, during the design stage, you thoroughly understand and work to bring together your data needs, the critical sections of your program, and potential assembly language optimizations.

More on this shortly.

To recap, here are some things to look for when striving to convert C code into optimized assembly language:

- Move the entire performance-critical section into a single assembly language function.
- Don't use calls or stack frame accesses inside the critical code, if possible, and avoid unnecessary memory accesses of any kind.
- Change segments as infrequently as possible.
- Optimize in terms of what assembly does well, *not* in terms of fine-tuning compiled C code.
- Change the rules to the benefit of assembly, if necessary; for example, reorganize data structures to allow efficient assembly language processing.

That said, let me show some of these precepts in action.

A C-to-Assembly Case Study

Listing 8.1 is the sample C application I'm going to use to examine optimization in action. Listing 8.1 isn't really complete—it doesn't handle the "no-matches" case well, and it assumes that the sum of all matches will fit into an **int**—but it will do just fine as an optimization example.

LISTING 8.1 L8-1.C

```
/* Program to search an array spanning a linked list of variable-
   sized blocks, for all entries with a specified ID number,
   and return the average of the values of all such entries. Each of
   the variable-sized blocks may contain any number of data entries,
   stored as an array of structures within the block. */

#include <stdio.h>
#ifdef __TURBOC__
#include <alloc.h>
#else
#include <malloc.h>
#endif
```

```
void main(void);
void exit(int);
unsigned int FindIDAverage(unsigned int, struct BlockHeader *);
/* Structure that starts each variable-sized block */
struct BlockHeader {
   struct BlockHeader *NextBlock; /* Pointer to next block, or NULL
                                     if this is the last block in the
                                     linked list */
   unsigned int BlockCount;       /* The number of DataElement entries
                                     in this variable-sized block */
};

/* Structure that contains one element of the array we'll search */
struct DataElement {
   unsigned int ID;     /* ID # for array entry */
   unsigned int Value;  /* Value of array entry */
};

void main(void) {
   int i,j;
   unsigned int IDToFind;
   struct BlockHeader *BaseArrayBlockPointer,*WorkingBlockPointer;
   struct DataElement *WorkingDataPointer;
   struct BlockHeader **LastBlockPointer;

   printf("ID # for which to find average: ");
   scanf("%d",&IDToFind);
   /* Build an array across 5 blocks, for testing */
   /* Anchor the linked list to BaseArrayBlockPointer */
   LastBlockPointer = &BaseArrayBlockPointer;
   /* Create 5 blocks of varying sizes */
   for (i = 1; i < 6; i++) {
      /* Try to get memory for the next block */
      if ((WorkingBlockPointer =
          (struct BlockHeader *) malloc(sizeof(struct BlockHeader) +
           sizeof(struct DataElement) * i * 10)) == NULL) {
         exit(1);
      }
      /* Set the # of data elements in this block */
      WorkingBlockPointer->BlockCount = i * 10;
      /* Link the new block into the chain */
      *LastBlockPointer = WorkingBlockPointer;
      /* Point to the first data field */
      WorkingDataPointer =
            (struct DataElement *) ((char *)WorkingBlockPointer +
            sizeof(struct BlockHeader));
      /* Fill the data fields with ID numbers and values */
      for (j = 0; j < (i * 10); j++, WorkingDataPointer++) {
         WorkingDataPointer->ID = j;
         WorkingDataPointer->Value = i * 1000 + j;
      }
      /* Remember where to set link from this block to the next */
      LastBlockPointer = &WorkingBlockPointer->NextBlock;
   }
   /* Set the last block's "next block" pointer to NULL to indicate
      that there are no more blocks */
   WorkingBlockPointer->NextBlock = NULL;
   printf("Average of all elements with ID %d: %u\n",
         IDToFind, FindIDAverage(IDToFind, BaseArrayBlockPointer));
```

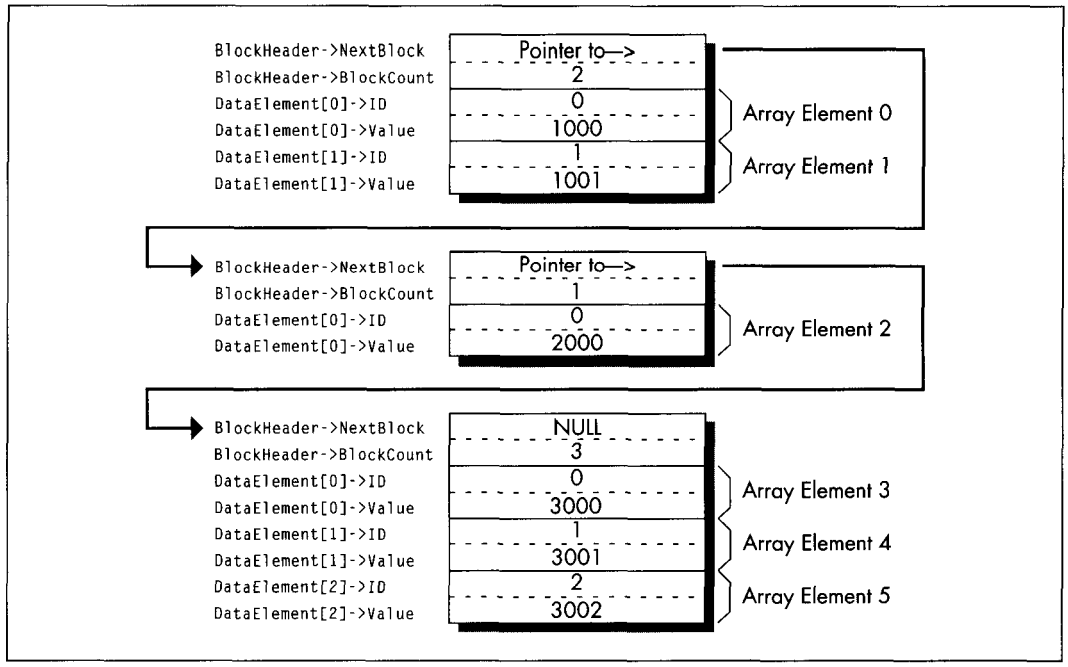
```
   exit(0);
}

/* Searches through the array of DataElement entries spanning the
   linked list of variable-sized blocks, starting with the block
   pointed to by BlockPointer, for all entries with IDs matching
   SearchedForID, and returns the average value of those entries. If
   no matches are found, zero is returned */

unsigned int FindIDAverage(unsigned int SearchedForID,
      struct BlockHeader *BlockPointer)
{
   struct DataElement *DataPointer;
   unsigned int IDMatchSum;
   unsigned int IDMatchCount;
   unsigned int WorkingBlockCount;

   IDMatchCount = IDMatchSum = 0;
   /* Search through all the linked blocks until the last block
      (marked with a NULL pointer to the next block) has been
      searched */
   do {
      /* Point to the first DataElement entry within this block */
      DataPointer =
            (struct DataElement *) ((char *)BlockPointer +
            sizeof(struct BlockHeader));
      /* Search all the DataElement entries within this block
         and accumulate data from all that match the desired ID */
      for (WorkingBlockCount=0;
            WorkingBlockCount<BlockPointer->BlockCount;
            WorkingBlockCount++, DataPointer++) {
         /* If the ID matches, add in the value and increment the
            match counter */
         if (DataPointer->ID == SearchedForID) {
            IDMatchCount++;
            IDMatchSum += DataPointer->Value;
         }
      }
      /* Point to the next block, and continue as long as that pointer
       isn't NULL */
   }  while ((BlockPointer = BlockPointer->NextBlock) != NULL);
   /* Calculate the average of all matches */
   if (IDMatchCount == 0)
      return(0);            /* Avoid division by 0 */
   else
      return(IDMatchSum / IDMatchCount);
}
```

The main body of Listing 8.1 constructs a linked list of memory blocks of various sizes and stores an array of structures across those blocks, as shown in Figure 8.2. The function **FindIDAverage** in Listing 8.1 searches through that array for all matches to a specified ID number and returns the average value of all such matches. **FindIDAverage** contains two nested loops, the outer one repeating once for each linked block and the inner one repeating once for each array element in each block. The inner loop—the critical one—is compact, containing only four statements, and should lend itself rather well to compiler optimization.



Linked array storage format (version 1).
Figure 8.2

As it happens, Microsoft C/C++ does optimize the inner loop of **FindIDAverage** nicely. Listing 8.2 shows the code Microsoft C/C++ generates for the inner loop, consisting of a mere seven assembly language instructions inside the loop. The compiler is smart enough to convert the loop index variable, which counts up but is used for nothing but counting loops, into a count-down variable so that the **LOOP** instruction can be used.

LISTING 8.2 L8-2.COD

```
; Code generated by Microsoft C for inner loop of FindIDAverage.
;|*** for (WorkingBlockCount=0;
;|***       WorkingBlockCount<BlockPointer->BlockCount;
;|***       WorkingBlockCount++, DataPointer++) {
          mov     WORD PTR [bp-6],0         ;WorkingBlockCount
          mov     bx,WORD PTR [bp+6]        ;BlockPointer
          cmp     WORD PTR [bx+2],0
          je      $FB264
          mov     cx,WORD PTR [bx+2]
          add     WORD PTR [bp-6],cx        ;WorkingBlockCount
          mov     di,WORD PTR [bp-2]        ;IDMatchSum
          mov     dx,WORD PTR [bp-4]        ;IDMatchCount
$L20004:
;|*** if (DataPointer->ID == SearchedForID) {
          mov     ax,WORD PTR [si]
          cmp     WORD PTR [bp+4],ax        ;SearchedForID
          jne     $I265
```

```
;|***              IDMatchCount++;
          inc     dx
;|***             IDMatchSum += DataPointer->Value;
          add     di,WORD PTR [si+2]
;|***           }
;|***         }
$I265:
          add     si,4
          loop    $L20004
          mov     WORD PTR [bp-2],di        ;IDMatchSum
          mov     WORD PTR [bp-4],dx        ;IDMatchCount
$FB264:
```

It's hard to squeeze much more performance from this code by tweaking it, as exemplified by Listing 8.3, a fine-tuned assembly version of **FindIDAverage** that was produced by looking at the assembly output of MS C/C++ and tightening it. Listing 8.3 eliminates all stack frame access in the inner loop, but that's about all the tightening there is to do. The result, as shown in Table 8.1, is that Listing 8.3 runs a modest 11 percent faster than Listing 8.1 on a 386. The results could vary considerably, depending on the nature of the data set searched through (average block size and frequency of matches). But, then, understanding the typical and worst case conditions is part of optimization, isn't it?

LISTING 8.3 L8-3.ASM

```
; Typically optimized assembly language version of FindIDAverage.
SearchedForID   equ     4     ;Passed parameter offsets in the
BlockPointer    equ     6     ; stack frame (skip over pushed BP
                              ; and the return address)
NextBlock       equ     0     ;Field offsets in struct BlockHeader
BlockCount      equ     2
BLOCK_HEADER_SIZE equ   4     ;Number of bytes in struct BlockHeader
ID              equ     0     ;struct DataElement field offsets
Value           equ     2
DATA_ELEMENT_SIZE equ   4     ;Number of bytes in struct DataElement
        .model  small
        .code
       public  _FindIDAverage
```

	On 20 MHz 386	On 10 MHz 286
Listing 8.1 (MSC with maximum optimization)	294 microseconds	768 microseconds
Listing 8.3 (Assembly)	265	644
Listing 8.4 (Optimized assembly)	212	486
Listing 8.6 (Optimized assembly with reorganized data)	100	207

Table 8.1 Execution Times of FindIDAverage.

```
_FindIDAverage  proc    near
        push    bp              ;Save caller's stack frame
        mov     bp,sp           ;Point to our stack frame
        push    di              ;Preserve C register variables
        push    si
        sub     dx,dx           ;IDMatchSum = 0
        mov     bx,dx           ;IDMatchCount = 0
        mov     si,[bp+BlockPointer]    ;Pointer to first block
        mov     ax,[bp+SearchedForID]   ;ID we're looking for
; Search through all the linked blocks until the last block
; (marked with a NULL pointer to the next block) has been searched.
BlockLoop:
; Point to the first DataElement entry within this block.
        lea     di,[si+BLOCK_HEADER_SIZE]
; Search through all the DataElement entries within this block
; and accumulate data from all that match the desired ID.
        mov     cx,[si+BlockCount]
        jcxz    DoNextBlock     ;No data in this block
IntraBlockLoop:
        cmp     [di+ID],ax      ;Do we have an ID match?
        jnz     NoMatch         ;No match
        inc     bx              ;We have a match; IDMatchCount++;
        add     dx,[di+Value]   ;IDMatchSum += DataPointer->Value;
NoMatch:
        add     di,DATA_ELEMENT_SIZE ;point to the next element
        loop    IntraBlockLoop
; Point to the next block and continue if that pointer isn't NULL.
DoNextBlock:
        mov     si,[si+NextBlock] ;Get pointer to the next block
        and     si,si           ;Is it a NULL pointer?
        jnz     BlockLoop       ;No, continue
; Calculate the average of all matches.
        sub     ax,ax           ;Assume we found no matches
        and     bx,bx
        jz      Done            ;We didn't find any matches, return 0
        xchg    ax,dx           ;Prepare for division
        div     bx              ;Return IDMatchSum / IDMatchCount
Done:   pop     si              ;Restore C register variables
        pop     di
        pop     bp              ;Restore caller's stack frame
        ret
_FindIDAverage  ENDP
        end
```

Listing 8.4 tosses some sophisticated optimization techniques into the mix. The loop is unrolled eight times, eliminating a good deal of branching, and **SCASW** is used instead of **CMP [DI],AX**. (Note, however, that **SCASW** is in fact slower than **CMP [DI],AX** on the 386 and 486, and is sometimes faster on the 286 and 8088 only because it's shorter and therefore may prefetch faster.) This advanced tweaking produces a 39 percent improvement over the original C code—substantial, but not a tremendous return for the optimization effort invested.

LISTING 8.4 L8-4.ASM

```
; Heavily optimized assembly language version of FindIDAverage.
; Features an unrolled loop and more efficient pointer use.
SearchedForID   equ     4       ;Passed parameter offsets in the
BlockPointer    equ     6       ; stack frame (skip over pushed BP
                                ; and the return address)
NextBlock       equ     0       ;Field offsets in struct BlockHeader
BlockCount      equ     2
BLOCK_HEADER_SIZE equ   4       ;Number of bytes in struct BlockHeader
ID              equ     0       ;struct DataElement field offsets
Value           equ     2
DATA_ELEMENT_SIZE equ   4       ;Number of bytes in struct DataElement
        .model  small
        .code
        public  _FindIDAverage
_FindIDAverage  proc    near
        push    bp              ;Save caller's stack frame
        mov     bp,sp           ;Point to our stack frame
        push    di              ;Preserve C register variables
        push    si
        mov     di,ds           ;Prepare for SCASW
        mov     es,di
        cld
        sub     dx,dx           ;IDMatchSum = 0
        mov     bx,dx           ;IDMatchCount = 0
        mov     si,[bp+BlockPointer]    ;Pointer to first block
        mov     ax,[bp+SearchedForID]   ;ID we're looking for
; Search through all of the linked blocks until the last block
; (marked with a NULL pointer to the next block) has been searched.
BlockLoop:
; Point to the first DataElement entry within this block.
        lea     di,[si+BLOCK_HEADER_SIZE]
; Search through all the DataElement entries within this block
; and accumulate data from all that match the desired ID.
        mov     cx,[si+BlockCount] ;Number of elements in this block
        jcxz    DoNextBlock  ;Skip this block if it's empty
        mov     bp,cx  ;***stack frame no longer available***
        add     cx,7
        shr     cx,1   ;Number of repetitions of the unrolled
        shr     cx,1   ; loop = (BlockCount + 7) / 8
        shr     cx,1
        and     bp,7   ;Generate the entry point for the
        shl     bp,1   ; first, possibly partial pass through
        jmp     cs:[LoopEntryTable+bp] ; the unrolled loop and
      ; vector to that entry point
        align   2
LoopEntryTable  label   word
        dw      LoopEntry8,LoopEntry1,LoopEntry2,LoopEntry3
        dw      LoopEntry4,LoopEntry5,LoopEntry6,LoopEntry7
M_IBL   macro   P1
        local   NoMatch
LoopEntry&P1&:
        scasw    ;Do we have an ID match?
        jnz     NoMatch      ;No match
      ;We have a match
        inc     bx     ;IDMatchCount++;
        add     dx,[di]      ;IDMatchSum += DataPointer->Value;
NoMatch:
        add     di,DATA_ELEMENT_SIZE-2 ;point to the next element
      ; (SCASW advanced 2 bytes already)
```

```
        endm
        align   2
IntraBlockLoop:
        M_IBL   8
        M_IBL   7
        M_IBL   6
        M_IBL   5
        M_IBL   4
        M_IBL   3
        M_IBL   2
        M_IBL   1
        loop    IntraBlockLoop
; Point to the next block and continue if that pointer isn't NULL.
DoNextBlock:
        mov     si,[si+NextBlock]       ;Get pointer to the next block
        and     si,si                   ;Is it a NULL pointer?
        jnz     BlockLoop               ;No, continue
; Calculate the average of all matches.
        sub     ax,ax       ;Assume we found no matches
        and     bx,bx
        jz      Done        ;We didn't find any matches, return 0
        xchg    ax,dx       ;Prepare for division
        div     bx          ;Return IDMatchSum / IDMatchCount
Done:   pop     si          ;Restore C register variables
        pop     di
        pop     bp          ;Restore caller's stack frame
        ret
_FindIDAverage  ENDP
        end
```

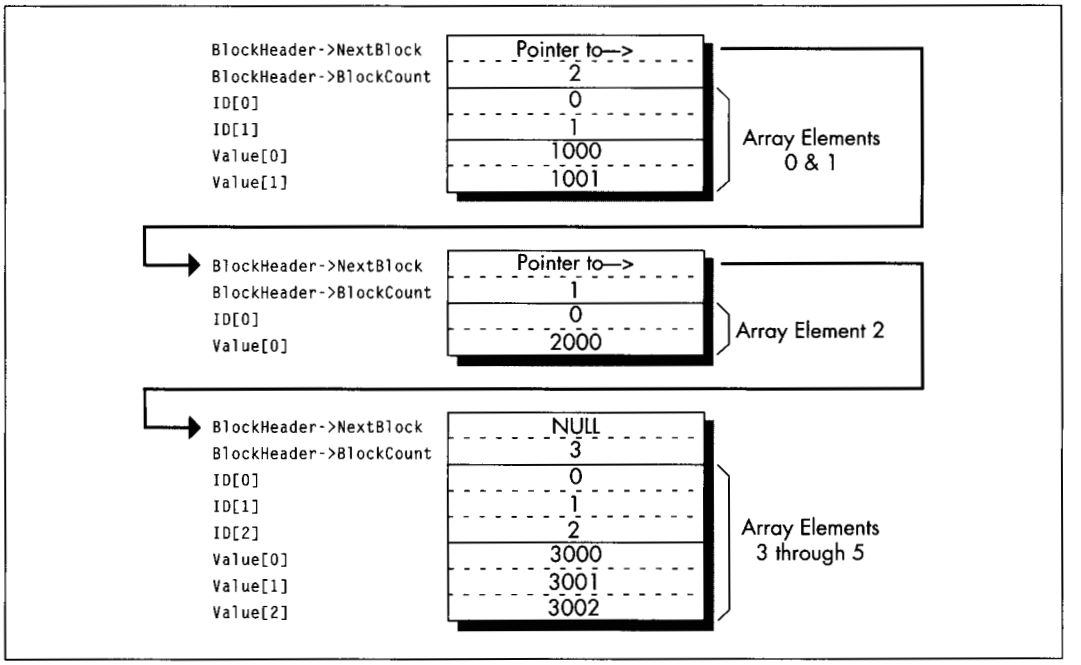
Listings 8.5 and 8.6 together go the final step and change the rules in favor of assembly language. Listing 8.5 creates the same list of linked blocks as Listing 8.1. However, instead of storing an array of structures within each block, it stores *two* arrays in each block, one consisting of ID numbers and the other consisting of the corresponding values, as shown in Figure 8.3. No information is lost; the data is merely rearranged.

LISTING 8.5 L8-5.C

```
/* Program to search an array spanning a linked list of variable-
   sized blocks, for all entries with a specified ID number,
   and return the average of the values of all such entries. Each of
   the variable-sized blocks may contain any number of data entries,
   stored in the form of two separate arrays, one for ID numbers and
   one for values. */

#include <stdio.h>
#ifdef __TURBOC__
#include <alloc.h>
#else
#include <malloc.h>
#endif

void main(void);
void exit(int);
extern unsigned int FindIDAverage2(unsigned int,
                                   struct BlockHeader *);
```



Linked array storage format (version 2).
Figure 8.3

```
/* Structure that starts each variable-sized block */
struct BlockHeader {
   struct BlockHeader *NextBlock; /* Pointer to next block, or NULL
                                     if this is the last block in the
                                     linked list */
   unsigned int BlockCount;       /* The number of DataElement entries
                                     in this variable-sized block */
};

void main(void) {
   int i,j;
   unsigned int IDToFind;
   struct BlockHeader *BaseArrayBlockPointer,*WorkingBlockPointer;
   int *WorkingDataPointer;
   struct BlockHeader **LastBlockPointer;

   printf("ID # for which to find average: ");
   scanf("%d",&IDToFind);

   /* Build an array across 5 blocks, for testing */
   /* Anchor the linked list to BaseArrayBlockPointer */
   LastBlockPointer = &BaseArrayBlockPointer;
   /* Create 5 blocks of varying sizes */
   for (i = 1; i < 6; i++) {
      /* Try to get memory for the next block */
```

```
      if ((WorkingBlockPointer =
          (struct BlockHeader *) malloc(sizeof(struct BlockHeader) +
           sizeof(int) * 2 * i * 10)) == NULL) {
         exit(1);
      }
      /* Set the number of data elements in this block */
      WorkingBlockPointer->BlockCount = i * 10;
      /* Link the new block into the chain */
      *LastBlockPointer = WorkingBlockPointer;
      /* Point to the first data field */
      WorkingDataPointer = (int *) ((char *)WorkingBlockPointer +
            sizeof(struct BlockHeader));
      /* Fill the data fields with ID numbers and values */
      for (j = 0; j < (i * 10); j++, WorkingDataPointer++) {
         *WorkingDataPointer = j;
         *(WorkingDataPointer + i * 10) = i * 1000 + j;
      }
      /* Remember where to set link from this block to the next */
      LastBlockPointer = &WorkingBlockPointer->NextBlock;
   }
   /* Set the last block's "next block" pointer to NULL to indicate
      that there are no more blocks */
   WorkingBlockPointer->NextBlock = NULL;
   printf("Average of all elements with ID %d: %u\n",
         IDToFind, FindIDAverage2(IDToFind, BaseArrayBlockPointer));
   exit(0);
}
```

LISTING 8.6 L8-6.ASM

```
; Alternative optimized assembly language version of FindIDAverage
; requires data organized as two arrays within each block rather
; than as an array of two-value element structures. This allows the
; use of REP SCASW for ID searching.

SearchedForID          equ   4     ;Passed parameter offsets in the
BlockPointer           equ   6     ; stack frame (skip over pushed BP
                                   ; and the return address)
NextBlock              equ   0     ;Field offsets in struct BlockHeader
BlockCount             equ   2
BLOCK_HEADER_SIZEequ   4     ;Number of bytes in struct BlockHeader

        .model  small
        .code
        public  _FindIDAverage2
_FindIDAverage2 proc    near
        push    bp              ;Save caller's stack frame
        mov     bp,sp           ;Point to our stack frame
        push    di              ;Preserve C register variables
        push    si
        mov     di,ds           ;Prepare for SCASW
        mov     es,di
        cld
        mov     si,[bp+BlockPointer]    ;Pointer to first block
        mov     ax,[bp+SearchedForID]   ;ID we're looking for
        sub     dx,dx                   ;IDMatchSum = 0
        mov     bp,dx                   ;IDMatchCount = 0
                                        ;***stack frame no longer available***
; Search through all the linked blocks until the last block
; (marked with a NULL pointer to the next block) has been searched.
```

```
BlockLoop:
; Search through all the DataElement entries within this block
; and accumulate data from all that match the desired ID.
        mov     cx,[si+BlockCount]
        jcxz    DoNextBlock  ;Skip this block if there's no data
                             ; to search through
        mov     bx,cx        ;We'll use BX to point to the
        shl     bx,1         ; corresponding value entry in the
                             ; case of an ID match (BX is the
                             ; length in bytes of the ID array)
; Point to the first DataElement entry within this block.
        lea     di,[si+BLOCK_HEADER_SIZE]
IntraBlockLoop:
        repnz   scasw        ;Search for the ID
        jnz     DoNextBlock  ;No match, the block is done
        inc     bp           ;We have a match; IDMatchCount++;
        add     dx,[di+bx-2] ;IDMatchSum += DataPointer->Value;
                             ; (SCASW has advanced DI 2 bytes)
        and     cx,cx        ;Is there more data to search through?
        jnz     IntraBlockLoop  ;yes
; Point to the next block and continue if that pointer isn't NULL.
DoNextBlock:
        mov     si,[si+NextBlock] ;Get pointer to the next block
        and     si,si             ;Is it a NULL pointer?
        jnz     BlockLoop         ;No, continue
; Calculate the average of all matches.
        sub     ax,ax       ;Assume we found no matches
        and     bp,bp
        jz      Done        ;We didn't find any matches, return 0
        xchg    ax,dx       ;Prepare for division
        div     bp          ;Return IDMatchSum / IDMatchCount
Done:   pop     si          ;Restore C register variables
        pop     di
        pop     bp          ;Restore caller's stack frame
        ret
_FindIDAverage2 ENDP
        end
```

The whole point of this rearrangement is to allow us to use **REP SCASW** to search through each block, and that's exactly what **FindIDAverage2** in Listing 8.6 does. The result: Listing 8.6 calculates the average about *three times* as fast as the original C implementation and more than twice as fast as Listing 8.4, heavily optimized as the latter code is.

I trust you get the picture. The sort of instruction-by-instruction optimization that so many of us love to do as a kind of puzzle is fun, but compilers can do it nearly as well as you can, and in the future will surely do it better. What a compiler *can't* do is tie together the needs of the program specification on the high end and the processor on the low end, resulting in critical code that runs just about as fast as the hardware permits. The only software that can do that is located north of your sternum and slightly aft of your nose. Dust it off and put it to work—and your code will never again be confused with anything by Hamilton, Joe, Frank, and Reynolds or Bo Donaldson and the Heywoods.