Chapter 1

The Best Optimizer Is between Your Ears

Chapter 1

The Human Element of Code Optimization

This book is devoted to a topic near and dear to my heart: writing software that pushes PCs to the limit. Given run-of-the-mill software, PCs run like the 97-pound-weakling minicomputers they are. Give them the proper care, however, and those ugly boxes are capable of miracles. The key is this: Only on microcomputers do you have the run of the whole machine, without layers of operating systems, drivers, and the like getting in the way. You can do *anything* you want, and you can understand everything that's going on, if you so wish.

As we'll see shortly, you should indeed so wish.

Is performance still an issue in this era of cheap 486 computers and super-fast Pentium computers? You bet. How many programs that *you* use really run so fast that you wouldn't be happier if they ran faster? We're so used to slow software that when a compile-and-link sequence that took two minutes on a PC takes just ten seconds on a 486 computer, we're ecstatic—when in truth we should be settling for nothing less than instantaneous response.

Impossible, you say? Not with the proper design, including incremental compilation and linking, use of extended and/or expanded memory, and well-crafted code. PCs can do just about anything you can imagine (with a few obvious exceptions, such as applications involving super-computer-class number-crunching) if you believe that it can be done, if you understand the computer inside and out, and if you're willing to think past the obvious solution to unconventional but potentially more fruitful approaches.

My point is simply this: PCs can work wonders. It's not *easy* coaxing them into doing that, but it's rewarding—and it's sure as heck fun. In this book, we're going to work some of those wonders, starting...

...now.

Understanding High Performance

Before we can create high-performance code, we must understand what high performance is. The objective (not always attained) in creating high-performance software is to make the software able to carry out its appointed tasks so rapidly that it responds instantaneously, as far as the user is concerned. In other words, high-performance code should ideally run so fast that any further improvement in the code would be pointless.

Notice that the above definition most emphatically does *not* say anything about making the software as fast as possible. It also does not say anything about using assembly language, or an optimizing compiler, or, for that matter, a compiler at all. It also doesn't say anything about how the code was designed and written. What it does say is that high-performance code shouldn't get in the user's way—and that's *all*.

That's an important distinction, because all too many programmers think that assembly language, or the right compiler, or a particular high-level language, or a certain design approach is the answer to creating high-performance code. They're not, any more than choosing a certain set of tools is the key to building a house. You do indeed need tools to build a house, but any of many sets of tools will do. You also need a blueprint, an understanding of everything that goes into a house, and the ability to *use* the tools.

Likewise, high-performance programming requires a clear understanding of the purpose of the software being built, an overall program design, algorithms for implementing particular tasks, an understanding of what the computer can do and of what all relevant software is doing—*and* solid programming skills, preferably using an optimizing compiler or assembly language. The optimization at the end is just the finishing touch, however.

Without good design, good algorithms, and complete understanding of the program's operation, your carefully optimized code will amount to one of mankind's least fruitful creations—a fast slow program.

"What's a fast slow program?" you ask. That's a good question, and a brief (true) story is perhaps the best answer.

When Fast Isn't Fast

In the early 1970s, as the first hand-held calculators were hitting the market, I knew a fellow named Irwin. He was a good student, and was planning to be an engineer.

Being an engineer back then meant knowing how to use a slide rule, and Irwin could jockey a slipstick with the best of them. In fact, he was so good that he challenged a fellow with a calculator to a duel—and won, becoming a local legend in the process.

When you get right down to it, though, Irwin was spitting into the wind. In a few short years his hard-earned slipstick skills would be worthless, and the entire discipline would be essentially wiped from the face of the earth. What's more, anyone with half a brain could see that changeover coming. Irwin had basically wasted the considerable effort and time he had spent optimizing his soon-to-be-obsolete skills.

What does all this have to do with programming? Plenty. When you spend time optimizing poorly-designed assembly code, or when you count on an optimizing compiler to make your code fast, you're wasting the optimization, much as Irwin did. Particularly in assembly, you'll find that without proper up-front design and everything else that goes into high-performance design, you'll waste considerable effort and time on making an inherently slow program as fast as possible—which is still slow—when you could easily have improved performance a great deal more with just a little thought. As we'll see, handcrafted assembly language and optimizing compilers matter, but less than you might think, in the grand scheme of things—and they scarcely matter at all unless they're used in the context of a good design and a thorough understanding of both the task at hand and the PC.

Rules for Building High-Performance Code

We've got the following rules for creating high-performance software:

- Know where you're going (understand the objective of the software).
- Make a big map (have an overall program design firmly in mind, so the various parts of the program and the data structures work well together).
- Make lots of little maps (design an algorithm for each separate part of the overall design).
- Know the territory (understand exactly how the computer carries out each task).
- Know when it matters (identify the portions of your programs where performance matters, and don't waste your time optimizing the rest).
- Always consider the alternatives (don't get stuck on a single approach; odds are there's a better way, if you're clever and inventive enough).
- Know how to turn on the juice (optimize the code as best you know how when it *does* matter).

Making rules is easy; the hard part is figuring out how to apply them in the real world. For my money, examining some actual working code is always a good way to get a handle on programming concepts, so let's look at some of the performance rules in action.

Know Where You're Going

If we're going to create high-performance code, first we have to know what that code is going to do. As an example, let's write a program that generates a 16-bit checksum of the bytes in a file. In other words, the program will add each byte in a specified file in turn into a 16-bit value. This checksum value might be used to make sure that a file hasn't been corrupted, as might occur during transmission over a modem or if a Trojan horse virus rears its ugly head. We're not going to do anything with the checksum value other than print it out, however; right now we're only interested in generating that checksum value as rapidly as possible.

Make a Big Map

How are we going to generate a checksum value for a specified file? The logical approach is to get the file name, open the file, read the bytes out of the file, add them together, and print the result. Most of those actions are straightforward; the only tricky part lies in reading the bytes and adding them together.

Make Lots of Little Maps

Actually, we're only going to make one little map, because we only have one program section that requires much thought—the section that reads the bytes and adds them up. What's the best way to do this?

It would be convenient to load the entire file into memory and then sum the bytes in one loop. Unfortunately, there's no guarantee that any particular file will fit in the available memory; in fact, it's a sure thing that many files *won't* fit into memory, so that approach is out.

Well, if the whole file won't fit into memory, one byte surely will. If we read the file one byte at a time, adding each byte to the checksum value before reading the next byte, we'll minimize memory requirements and be able to handle any size file at all.

Sounds good, eh? Listing 1.1 shows an implementation of this approach. Listing 1.1 uses C's **read()** function to read a single byte, adds the byte into the checksum value, and loops back to handle the next byte until the end of the file is reached. The code is compact, easy to write, and functions perfectly—with one slight hitch: It's *slow.*

LISTING 1.1 L1-1.C

```
/*
* Program to calculate the 16-bit checksum of all bytes in the
* specified file. Obtains the bytes one at a time via read(),
* letting DOS perform all data buffering.
*/
#include <stdio.h>
#include <fcntl.h>

main(int argc, char *argv[]) {
```

```
    int Handle;
    unsigned char Byte;
    unsigned int Checksum;
    int ReadLength;

    if ( argc != 2 ) {
        printf("usage: checksum filename\n");
        exit(1);
    }
    if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
        printf("Can't open file: %s\n", argv[1]);
        exit(1);
    }

    /* Initialize the checksum accumulator */
    Checksum = 0;

    /* Add each byte in turn into the checksum accumulator */
    while ( (ReadLength = read(Handle, &Byte, sizeof(Byte))) > 0 ) {
        Checksum += (unsigned int) Byte;
    }
    if ( ReadLength == -1 ) {
        printf("Error reading file %s\n", argv[1]);
        exit(1);
    }

    /* Report the result */
    printf("The checksum is: %u\n", Checksum);
    exit(0);
}
```

Table 1.1 shows the time taken for Listing 1.1 to generate a checksum of the WordPerfect version 4.2 thesaurus file, TH.WP (362,293 bytes in size), on a 10 MHz AT machine of no special parentage. Execution times are given for Listing 1.1 compiled with Borland and Microsoft compilers, with optimization both on and off; all four times are pretty much the same, however, and all are much too slow to be acceptable. Listing 1.1 requires over two and one-half minutes to checksum *one* file!

Listings 1.2 and 1.3 form the C/assembly equivalent to Listing 1.1, and Listings 1.6 and 1.7 form the C/assembly equivalent to Listing 1.5.

These results make it clear that it's folly to rely on your compiler's optimization to make your programs fast. Listing 1.1 is simply poorly designed, and no amount of compiler optimization will compensate for that failing. To drive home the point, consider Listings 1.2 and 1.3, which together are equivalent to Listing 1.1 except that the entire checksum loop is written in tight assembly code. The assembly language implementation is indeed faster than any of the C versions, as shown in Table 1.1, but it's less than 10 percent faster, and it's still unacceptably slow.

Listing	Borland (no opt)	Microsoft (no opt)	Borland (opt)	Microsoft (opt)	Assembly	Optimization Ratio
1	166.9	166.8	167.0	165.8	155.1	1.08
4	13.5	13.6	13.5	13.5	...	1.01
5	4.7	5.5	3.8	3.4	2.7	2.04
Ratio best designed to worst designed	35.51	30.33	43.95	48.76	57.44	

Note: The execution times (in seconds) for this chapter's listings were timed when the compiled listings were run on the WordPerfect 4.2 thesaurus file TH.WP (362,293 bytes in size), as compiled in the small model with Borland and Microsoft compilers with optimization on (*opt*) and off (*no opt*). All times were measured with Paradigm Systems' TIMER program on a 10 MHz 1-wait-state AT clone with a 28-ms hard disk, with disk caching turned off.

Table 1.1 Execution Times for WordPerfect Checksum.

LISTING 1.2 L1-2.C

```
/*
* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file. Obtains the bytes one at a time in
* assembler, via direct calls to DOS.
*/

#include <stdio.h>
#include <fcntl.h>

main(int argc, char *argv[]) {
      int Handle;
      unsigned char Byte;
      unsigned int Checksum;
      int ReadLength;

      if ( argc != 2 ) {
            printf("usage: checksum filename\n");
            exit(1);
      }
      if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
            printf("Can't open file: %s\n", argv[1]);
            exit(1);
      }
      if ( !ChecksumFile(Handle, &Checksum) ) {
            printf("Error reading file %s\n", argv[1]);
            exit(1);
      }

      /* Report the result */
      printf("The checksum is: %u\n", Checksum);
      exit(0);
}
```

LISTING 1.3 L1-3.ASM

```
; Assembler subroutine to perform a 16-bit checksum on the file
; opened on the passed-in handle. Stores the result in the
; passed-in checksum variable. Returns 1 for success, 0 for error.
;
; Call as:
;           int ChecksumFile(unsigned int Handle, unsigned int *Checksum);
;
; where:
;           Handle = handle # under which file to checksum is open
;           Checksum = pointer to unsigned int variable checksum is
;           to be stored in
;
; Parameter structure:
;
Parms       struc
                dw          ?     ;pushed BP
                dw          ?     ;return address
Handle          dw          ?
Checksum        dw          ?
Parms       ends
;
                .model small
                .data
TempWord label  word
TempByte        db          ?     ;each byte read by DOS will be stored here
                db          0     ;high byte of TempWord is always 0
                                  ; for 16-bit adds
;
                .code
                public      _ChecksumFile
_ChecksumFile   proc  near
                push        bp
                mov         bp,sp
                push        si                    ;save C's register variable
;
                mov         bx,[bp+Handle]        ;get file handle
                sub         si,si                 ;zero the checksum
                                                  ;accumulator
                mov         cx,1                  ;request one byte on each
                                                  ;read
                mov         dx,offset TempByte    ;point DX to the byte in
                                                  ;which DOS should store
                                                  ;each byte read
ChecksumLoop:
                mov         ah,3fh          ;DOS read file function #
                int         21h             ;read the byte
                jc          ErrorEnd        ;an error occurred
                and         ax,ax           ;any bytes read?
                jz          Success         ;no-end of file reached-we're done
                add         si,[TempWord]   ;add the byte into the
                                            ;checksum total
                jmp         ChecksumLoop
ErrorEnd:
                sub         ax,ax           ;error
                jmp         short Done
Success:
                mov         bx,[bp+Checksum] ;point to the checksum variable
                mov         [bx],si         ;save the new checksum
                mov         ax,1            ;success
```

```
;
Done:
                pop         si              ;restore C's register variable
                pop         bp
                ret
_ChecksumFile   endp
                end
```

The lesson is clear: Optimization makes code faster, but without proper design, optimization just creates fast slow code.

Well, then, how are we going to improve our design? Before we can do that, we have to understand what's wrong with the current design.

Know the Territory

Just why is Listing 1.1 so slow? In a word: overhead. The C library implements the **read()** function by calling DOS to read the desired number of bytes. (I figured this out by watching the code execute with a debugger, but you can buy library source code from both Microsoft and Borland.) That means that Listing 1.1 (and Listing 1.3 as well) executes one DOS function per byte processed—and DOS functions, especially this one, come with a lot of overhead.

For starters, DOS functions are invoked with interrupts, and interrupts are among the slowest instructions of the x86 family CPUs. Then, DOS has to set up internally and branch to the desired function, expending more cycles in the process. Finally, DOS has to search its own buffers to see if the desired byte has already been read, read it from the disk if not, store the byte in the specified location, and return. All of that takes a *long* time—far, far longer than the rest of the main loop in Listing 1.1. In short, Listing 1.1 spends virtually all of its time executing **read()**, and most of that time is spent somewhere down in DOS.

You can verify this for yourself by watching the code with a debugger or using a code profiler, but take my word for it: There's a great deal of overhead to DOS calls, and that's what's draining the life out of Listing 1.1.

How can we speed up Listing 1.1? It should be clear that we must somehow avoid invoking DOS for every byte in the file, and that means reading more than one byte at a time, then buffering the data and parceling it out for examination one byte at a time. By gosh, that's a description of C's stream I/O feature, whereby C reads files in chunks and buffers the bytes internally, doling them out to the application as needed by reading them from memory rather than calling DOS. Let's try using stream I/O and see what happens.

Listing 1.4 is similar to Listing 1.1, but uses **fopen()** and **getc()** (rather than **open()** and **read()**) to access the file being checksummed. The results confirm our theories splendidly, and validate our new design. As shown in Table 1.1, Listing 1.4 runs more than an order of magnitude faster than even the assembly version of Listing 1.1, *even though Listing 1.1 and Listing 1.4 look almost the same.* To the casual observer, **read()**

and **getc()** would seem slightly different but pretty much interchangeable, and yet in this application the performance difference between the two is about the same as that between a 4.77 MHz PC and a 16 MHz 386.

Make sure you understand what really goes on when you insert a seemingly-innocuous function call into the time-critical portions of your code.

In this case that means knowing how DOS and the C/C++ file-access libraries do their work. In other words, *know the territory*!

LISTING 1.4 L1-4.C

```
/*
* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file. Obtains the bytes one at a time via
* getc(), allowing C to perform data buffering.
*/
#include <stdio.h>

main(int argc, char *argv[]) {
      FILE *CheckFile;
      int Byte;
      unsigned int Checksum;

      if ( argc != 2 ) {
            printf("usage: checksum filename\n");
            exit(1);
      }
      if ( (CheckFile = fopen(argv[1], "rb")) == NULL ) {
            printf("Can't open file: %s\n", argv[1]);
            exit(1);
      }

      /* Initialize the checksum accumulator */
      Checksum = 0;

      /* Add each byte in turn into the checksum accumulator */
      while ( (Byte = getc(CheckFile)) != EOF ) {
            Checksum += (unsigned int) Byte;
      }

      /* Report the result */
      printf("The checksum is: %u\n", Checksum);
      exit(0);
}
```

Know When It Matters

The last section contained a particularly interesting phrase: *the time-critical portions of your code.* Time-critical portions of your code are those portions in which the speed of the code makes a significant difference in the overall performance of your program—and by "significant," I don't mean that it makes the code 100 percent faster, or 200 percent, or any particular amount at all, but rather that it makes the program more responsive and/or usable *from the user's perspective.*

Don't waste time optimizing non-time-critical code: set-up code, initialization code, and the like. Spend your time improving the performance of the code inside heavily-used loops and in the portions of your programs that directly affect response time. Notice, for example, that I haven't bothered to implement a version of the checksum program entirely in assembly; Listings 1.2 and 1.6 call assembly subroutines that handle the time-critical operations, but C is still used for checking command-line parameters, opening files, printing, and the like.

If you were to implement any of the listings in this chapter entirely in hand-optimized assembly, I suppose you might get a performance improvement of a few percent—but I rather doubt you'd get even that much, and you'd sure as heck spend an awful lot of time for whatever meager improvement does result. Let C do what it does well, and use assembly only when it makes a perceptible difference.

Besides, we don't want to optimize until the design is refined to our satisfaction, and that won't be the case until we've thought about other approaches.

Always Consider the Alternatives

Listing 1.4 is good, but let's see if there are other—perhaps less obvious—ways to get the same results faster. Let's start by considering why Listing 1.4 is so much better than Listing 1.1. Like **read()**, **getc()** calls DOS to read from the file; the speed improvement of Listing 1.4 over Listing 1.1 occurs because **getc()** reads many bytes at once via DOS, then manages those bytes for us. That's faster than reading them one at a time using **read()**—but there's no reason to think that it's faster than having our program read and manage blocks itself. Easier, yes, but not faster.

Consider this: Every invocation of **getc()** involves pushing a parameter, executing a call to the C library function, getting the parameter (in the C library code), looking up information about the desired stream, unbuffering the next byte from the stream, and returning to the calling code. That takes a considerable amount of time, especially by contrast with simply maintaining a pointer to a buffer and whizzing through the data in the buffer inside a single loop.

There are four reasons that many programmers would give for not trying to improve on Listing 1.4:

1. The code is already fast enough.
2. The code works, and some people are content with code that works, even when it's slow enough to be annoying.
3. The C library is written in optimized assembly, and it's likely to be faster than any code that the average programmer could write to perform essentially the same function.
4. The C library conveniently handles the buffering of file data, and it would be a nuisance to have to implement that capability.

I'll ignore the first reason, both because performance is no longer an issue if the code is fast enough and because the current application does *not* run fast enough—13 seconds is a long time. (Stop and wait for 13 seconds while you're doing something intense, and you'll see just how long it is.)

The second reason is the hallmark of the mediocre programmer. Know when optimization matters—and then optimize when it does!

The third reason is often fallacious. C library functions are not always written in assembly, nor are they always particularly well-optimized. (In fact, they're often written for *portability*, which has nothing to do with optimization.) What's more, they're general-purpose functions, and often can be outperformed by well-but-not-brilliantly-written code that is well-matched to a specific task. As an example, consider Listing 1.5, which uses internal buffering to handle blocks of bytes at a time. Table 1.1 shows that Listing 1.5 is 2.5 to 4 times faster than Listing 1.4 (and as much as 49 times faster than Listing 1.1!), even though it uses no assembly at all.

Clearly, you can do well by using special-purpose C code in place of a C library function—if you have a thorough understanding of how the C library function operates and exactly what your application needs done. Otherwise, you'll end up rewriting C library functions in C, which makes no sense at all.

LISTING 1.5 L1-5.C

```
/*
* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file.  Buffers the bytes internally, rather
* than letting C or DOS do the work.
*/
#include <stdio.h>
#include <fcntl.h>
#include <alloc.h>   /* alloc.h for Borland,
                        malloc.h for Microsoft  */

#define BUFFER_SIZE  0x8000   /* 32Kb data buffer */

main(int argc, char *argv[]) {
      int Handle;
      unsigned int Checksum;
      unsigned char *WorkingBuffer, *WorkingPtr;
      int WorkingLength, LengthCount;

      if ( argc != 2 ) {
            printf("usage: checksum filename\n");
            exit(1);
      }
      if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
            printf("Can't open file: %s\n", argv[1]);
            exit(1);
      }

      /* Get memory in which to buffer the data */
      if ( (WorkingBuffer = malloc(BUFFER_SIZE)) == NULL ) {
            printf("Can't get enough memory\n");
```

```
            exit(1);
        }

        /* Initialize the checksum accumulator */
        Checksum = 0;

        /* Process the file in BUFFER_SIZE chunks */
        do {
            if ( (WorkingLength = read(Handle, WorkingBuffer,
                BUFFER_SIZE)) == -1 ) {
                printf("Error reading file %s\n", argv[1]);
                exit(1);
            }
            /* Checksum this chunk */
            WorkingPtr = WorkingBuffer;
            LengthCount = WorkingLength;
            while ( LengthCount-- ) {
            /* Add each byte in turn into the checksum accumulator */
                Checksum += (unsigned int) *WorkingPtr++;
            }
        } while ( WorkingLength );

        /* Report the result */
        printf("The checksum is: %u\n", Checksum);
        exit(0);
}
```

That brings us to the fourth reason: avoiding an internal-buffered implementation like Listing 1.5 because of the difficulty of coding such an approach. True, it is easier to let a C library function do the work, but it's not all that hard to do the buffering internally. The key is the concept of handling data in *restartable blocks;* that is, reading a chunk of data, operating on the data until it runs out, suspending the operation while more data is read in, and then continuing as though nothing had happened.

In Listing 1.5 the restartable block implementation is pretty simple because checksumming works with one byte at a time, forgetting about each byte immediately after adding it into the total. Listing 1.5 reads in a block of bytes from the file, checksums the bytes in the block, and gets another block, repeating the process until the entire file has been processed. In Chapter 5, we'll see a more complex restartable block implementation, involving searching for text strings.

At any rate, Listing 1.5 isn't much more complicated than Listing 1.4—and it's a *lot* faster. Always consider the alternatives; a bit of clever thinking and program redesign can go a long way.

Know How to Turn On the Juice

I have said time and again that optimization is pointless until the design is settled. When that time comes, however, optimization can indeed make a significant difference. Table 1.1 indicates that the optimized version of Listing 1.5 produced by Microsoft C outperforms an unoptimized version of the same code by more than 60 percent. What's more, a mostly-assembly version of Listing 1.5, shown in Listings 1.6

and 1.7, outperforms even the best-optimized C version of Listing 1.5 by 26 percent. These are considerable improvements, well worth pursuing—once the design has been maxed out.

LISTING 1.6 L1-6.C

```
/*
* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file.  Buffers the bytes internally, rather
* than letting C or DOS do the work, with the time-critical
* portion of the code written in optimized assembler.
*/
#include <stdio.h>
#include <fcntl.h>
#include <alloc.h>   /* alloc.h for Borland,
                        malloc.h for Microsoft  */

#define BUFFER_SIZE  0x8000   /* 32K data buffer */

main(int argc, char *argv[]) {
      int Handle;
      unsigned int Checksum;
      unsigned char *WorkingBuffer;
      int WorkingLength;

      if ( argc != 2 ) {
            printf("usage: checksum filename\n");
            exit(1);
      }
      if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
            printf("Can't open file: %s\n", argv[1]);
            exit(1);
      }

      /* Get memory in which to buffer the data */
      if ( (WorkingBuffer = malloc(BUFFER_SIZE)) == NULL ) {
            printf("Can't get enough memory\n");
            exit(1);
      }

      /* Initialize the checksum accumulator */
      Checksum = 0;

      /* Process the file in 32K chunks */
      do {
            if ( (WorkingLength = read(Handle, WorkingBuffer,
            BUFFER_SIZE)) == -1 ) {
                  printf("Error reading file %s\n", argv[1]);
                  exit(1);
            }
            /* Checksum this chunk if there's anything in it */
            if ( WorkingLength )
                  ChecksumChunk(WorkingBuffer, WorkingLength, &Checksum);
      } while ( WorkingLength );

      /* Report the result */
      printf("The checksum is: %u\n", Checksum);
      exit(0);
}
```

LISTING 1.7 L1-7.ASM

```
; Assembler subroutine to perform a 16-bit checksum on a block of
; bytes 1 to 64K in size. Adds checksum for block into passed-in
; checksum.
;
; Call as:
;     void ChecksumChunk(unsigned char *Buffer,
;     unsigned int BufferLength, unsigned int *Checksum);
;
; where:
;     Buffer = pointer to start of block of bytes to checksum
;     BufferLength = # of bytes to checksum (0 means 64K, not 0)
;     Checksum = pointer to unsigned int variable checksum is
;     stored in
;
; Parameter structure:
;
Parms struc
                dw    ?     ;pushed BP
                dw    ?     ;return address
Buffer          dw    ?
BufferLength    dw    ?
Checksum        dw    ?
Parms ends
;
      .model small
      .code
      public _ChecksumChunk
_ChecksumChunk   proc  near
      push  bp
      mov   bp,sp
      push  si                      ;save C's register variable
;
      cld                           ;make LODSB increment SI
      mov   si,[bp+Buffer]          ;point to buffer
      mov   cx,[bp+BufferLength]    ;get buffer length
      mov   bx,[bp+Checksum]        ;point to checksum variable
      mov   dx,[bx]                 ;get the current checksum
      sub   ah,ah                   ;so AX will be a 16-bit value after LODSB
ChecksumLoop:
      lodsb                 ;get the next byte
      add   dx,ax           ;add it into the checksum total
      loop  ChecksumLoop    ;continue for all bytes in block
      mov   [bx],dx         ;save the new checksum
;
      pop   si              ;restore C's register variable
      pop   bp
      ret
_ChecksumChunk   endp
      end
```

Note that in Table 1.1, optimization makes little difference except in the case of Listing 1.5, where the design has been refined considerably. Execution time in the other cases is dominated by time spent in DOS and/or the C library, so optimization of the code you write is pretty much irrelevant. What's more, while the approximately two-times improvement we got by optimizing is not to be sneezed at, it pales against the up-to-50-times improvement we got by redesigning.

By the way, the execution times even of Listings 1.6 and 1.7 are dominated by DOS disk access times. If a disk cache is enabled and the file to be checksummed is already in the cache, the assembly version is three times as fast as the C version. In other words, the inherent nature of this application limits the performance improvement that can be obtained via assembly. In applications that are more CPU-intensive and less disk-bound, particularly those applications in which string instructions and/or unrolled loops can be used effectively, assembly tends to be considerably faster relative to C than it is in this very specific case.

Don't get hung up on optimizing compilers or assembly language—the best optimizer is between your ears.

All this is basically a way of saying: Know where you're going, know the territory, and know when it matters.

Where We've Been, What We've Seen

What have we learned? Don't let other people's code—even DOS—do the work for you when speed matters, at least not without knowing what that code does and how well it performs.

Optimization only matters after you've done your part on the program design end. Consider the ratios on the vertical axis of Table 1.1, which show that optimization is almost totally wasted in the checksumming application without an efficient design. Optimization is no panacea. Table 1.1 shows a two-times improvement from optimization—and a 50-times-plus improvement from redesign. The longstanding debate about which C compiler optimizes code best doesn't matter quite so much in light of Table 1.1, does it? Your organic optimizer matters much more than your compiler's optimizer, and there's always assembly for those usually small sections of code where performance really matters.

Where We're Going

This chapter has presented a quick step-by-step overview of the design process. I'm not claiming that this is the only way to create high-performance code; it's just an approach that works for me. Create code however you want, but never forget that design matters more than detailed optimization. Never stop looking for inventive ways to boost performance—and never waste time speeding up code that doesn't need to be sped up.

I'm going to focus on specific ways to create high-performance code from now on. In Chapter 5, we'll continue to look at restartable blocks and internal buffering, in the form of a program that searches files for text strings.