Programming in Tight Spaces

In the realm of the ultrasmall, inexpensive memory and fast processors can't compensate for inelegant code

RICK GREHAN

In the brave new world of OOP (object-oriented programming), where huge development systems fit only on CD-ROMs and cross-platform API libraries take up most of your disk space, the solutions to many programming problems lie in hardware: more memory, a faster processor, and a disk cache. Today's computer science students are taught that the future lies in writing structured, reusable code. However, there are places where none of this works.

The programming techniques of the large-scale microcomputer world collapse if you apply them in the world of the ultrasmall microcontroller or IBP (itty-bitty processor). In this fascinating realm, there are no Pentium processors with 16 MB of RAM. Instead, you find designs with 4 KB of RAM paired with 8-bit Zilog Z8s, Motorola HC05s, and Microchip Technology PICs. (For details about these processors, see "Processors Proliferate," September BYTE.)

The Arena

Programmers working with IBPs create embedded components for automobiles, cordless and cellular communications equipment, medical instrumentation, and entertainment products. Programming battlegrounds exist on a few fronts. First, there's chip count and size. Because a number of microcontroller-based products are hand-held devices, toys, and consumer items, smaller chip real estate (or fewer chips) translates directly to reduced cost. This is especially important for consumer electronics because a manufacturer's production run may be in the hundreds of thousands, and the full-run price difference between adding an additional 20-cent chip or not can be significant. For the programmer, this means he or she has to work with reduced resources, such as less memory, or use software instead of hardware to solve some problems.

The second battleground is performance. Programs on microcontrollers are always managing interfaces to the real world, an activity that is governed by precise timing constraints. For example, the microcontroller in a little Santa Claus doll has to make the eyes blink rhythmically *and* play Christmas-carol tones at the proper pitch. The processor must do this using only 8 bits, and you're lucky if you can run it at anything over 10 MHz. It's a classic conflict of space versus speed. And when you're working with routines that must fit in tens of bytes, you don't have the luxury of clicking on a compiler's dialog box to pick "optimize for speed" over "optimize for space." You have to optimize for both (I'll present an example of how you do this in a moment).

The last battleground is the sheer claustrophobia programmers face. An IBP is usually a one-chip device, with CPU, RAM, and ROM all in one package. Registers and RAM are typically the same thing, and there's often no more than 32 bytes of RAM. As I mentioned before, programmers may get 4 KB of RAM if they're lucky, which means the code may have room for only 512 instructions.

Case in Point

Because of its size, a microspace program cannot be a collection of independent, separate routines. Instead, routines are interdependent and interlocking. One programmer of IBPs described his work as being similar to assembling a jigsaw puzzle. Programmers working in these tight spaces are more like artists than they are like factory workers cranking out objects for some software foundry.

Chip Gracey, a software engineer with Parallax, wrote the on-chip code for the company's BASIC Stamp, an elegantly designed PIC 16C56 development system that consists of two ICs, a 4-MHz oscillator circuit, a voltage regulator, and some passive electronics that are all powered by a 9-V transistor battery (for more details, see the text box "The Taming Power of the Small," September BYTE, page 68). You can program the board in BASIC by downloading code from a PC-compatible machine.

Sine Wave in Six Steps

Super-simple Microchip Technology PIC 16C5x code for generating a sine wave. Note that the heart of the routine is composed of only six instructions. On a 20-MHz PIC microcontroller, those instructions execute in only 1 µ.

```
sine = 08h              ;Declare sine reg.
velo = 09h              ;Declare velocity reg.

init    mov sine,#32    ;Set init. value
        clr velo        ;Reset velocity
loop    mov w,-velo     ;Get decr'd val. of velocity
        snb sine,7      ;Skip next instr. if sine>=0
        mov w,++velo    ;Get incr'd val. of velocity
        add sine,w      ;Add velocity to sine
        mov velo,w      ;Store new velocity
        code to use sine goes here...
        jmp loop
```

continued

Building the Stamp meant working on a processor with 1 KB of code space—about one-thousandth the memory space of most personal computers.

Gracey took four days to locate and fix the last bug on the BASIC Stamp. The time was needed not because the bug was hard to find, but because its correction required adding an instruction to the code. Unfortunately, there was no room, because Gracey had stuffed so much functionality into the Stamp the on-chip ROM was full. So he spent those four days figuring out how to rewrite code to free up the one word of instruction space he needed to make the fix. This was not just an editing job; he had to figure out how to make one routine one instruction shorter, and so much looping and code interdependence was at play that changing any given instruction sequence might have affected the performance of several routines. He had no choice but to write spaghetti code.

Other Considerations

What about reusability, a cornerstone of object-oriented technology? Is there any place for it in the environment of microcontrollers? Not much, according to Dave Hampton, an independent consultant who has been writing microcontroller programs for nearly a decade.

He admits to keeping a toolbox of routines that he uses often, but there's danger in starting a project by simply grabbing an off-the-shelf routine, he says. A generic routine might not make use of memory-saving or performance-boosting features of the processor, so blindly using such a routine as the starting point might take him down a wrong path. Consequently, Hampton finds that most of his projects consist of about 25 percent reused code and 75 percent custom code.

When programming for IBPs, intimate knowledge of the hardware is unquestionably a key requirement for a successful design. The upshot is that while many of us are clamoring for portability, IBP programmers thrive on the precise opposite.

A Taste

Programmers in the world of the small must become algorithm innovators. The famous sourcebooks of computer algorithms, such as *The Art of Computer Programming* series (Addison-Wesley) by Donald Knuth and *Numerical Recipes* editions (Cambridge University Press) by William Press, et al., have a place on the IBP programmer's shelf, but when you're counting every byte of code space, you've got to do some trailblazing. You can't simply go after an existing algorithm with a mental paring knife and shave away layers until you've reduced the code to its essentials. Instead, you often have to devise a completely new routine for solving your specific programming problem.

Gracey offers the following example: The routine shown in the listing "Sine Wave in Six Steps" on page 217 acts as a sine function generator: Each time you call the routine, it produces the next "step" in the waveform. You could send the output of the routine to a DAC (D/A converter) and have a minimum-overhead function generator.

The listing shows the algorithm as it's implemented in PIC 16C5x instructions. As you can see, the entire sine algorithm consumes six instructions. If this seems like unnecessary masochism, consider the code's destination processor. The Microchip PIC 16C5x 8-bit processors are complete 1-chip microcontrollers ideal for super-small applications (some members fit in 18-pin packages). Each 16C5x-family member features an on-chip ROM that can hold from 512- to 2-KB of instructions. However, there are no provisions for external memory, so code space is at a premium.

You'll notice that the algorithm performs its magic without series approximation or table-lookup; either would be impractical in most IBP applications. A series approximation would have taken precious space for both the routine and the coefficients, and it would have involved time-consuming multiplication instructions. A table-lookup approach would have meant using up storage to hold the table.

The listing "QuickBasic Sine Wave" is a short QuickBasic-compatible program that graphically demonstrates how well the sine-wave function in the listing on page 217 performs. To use it, type in this listing and alter the initial values of `sine` and `velocity` that appear near the beginning of the program. Watch the resulting waveform as you do.

QuickBasic Sine Wave

QuickBasic source code demonstrating how the algorithm in "Sine Wave in Six Steps" on page 217 generates a sine wave. Note: You can type in and run the source code to see the waveform. Alter initial values of `sine` and `velocity` that appear near the beginning of the program to change the waveform.

```
REM Simple method for sequentially computing a
REM sine wave.
SCREEN 9: x=639: y=175
REM Initialize sine and velocity
sine=128: velocity=0
REM Draw axes and initial offsets
COLOR 12
LINE (0,y)-(x,y)
LINE (0,y-sine)-(x,y-sine)
LINE (0,y+sine)-(x,y+sine)
COLOR 15
REM Draw pattern
FOR i=0 to x
IF sine>=0 THEN velocity=velocity-1
 ELSE velocity=velocity+1
sine=sine+velocity
PSET (i,y+sine)
NEXT i
```

Get Small

As memory cost-per-megabyte continues to descend, processor performance-per-dollar continues to ascend, and system needs continue to expand to consume system resources, we'll see bigger and bigger programs. I can't help but believe this will create fertile ground for sloppy programming. The day draws near when it will no longer be an embarrassment to use a bubble sort instead of a heap sort: Who's going to care if the list gets alphabetized in one-tenth of a second instead of one-hundredth of a second?

Nevertheless, it's good to know that one sharp edge of the programming community is still alive and well. IBP programmers are still scrounging to free up that 1 byte, and they are still counting cycles. In the end, it's reassuring to know there are still assembly language programmers. ■

Rick Grehan is technical director of the BYTE Lab. He has a B.S. in physics and applied mathematics and an M.S. in mathematics/computer science. He can be reached on the Internet or BIX at rick_g@bix.com.