

LESSONS FROM THE TRENCHES

George Martin

Number Crunching with Embedded Processors

This month, George walks us through binary numbers, embedded processors, and how different numbering systems work together with CPU instructions and the C language. He takes us through the steps and then encourages us to walk on our own with different examples. And remember, practice makes perfect!



This month, let's talk numbers—binary numbers, embedded processors, and how different numbering systems interact with CPU instructions and the C programming language.

NUMBER TYPES IN C

The C language has some built-in number types—*char*, *int*, *long*, and *float*, along with modifiers, signed, unsigned, short, and double. These number types have different meanings, depending on the compiler and CPU. After I define the basic number types, I'll focus on signed and unsigned.

It's good practice to use *#typedefs*, so you're sure about the amount of data you are referencing. Let's assume you are writing code for an 8-bit CPU, and your compiler defines *char* as 8 bits, *int* as 16 bits, and *long* as 32 bits. Table 1 shows how the *typedefs* would look.

BINARY BASICS

Let's see how the CPU works with binary numbers. The experts may wish to skip ahead, but let's assume you have an 8-bit computer with a basic 8-bit register. An 8-bit register can hold the binary patterns from 0000 0000 to 1111 1111. In a straightforward notation, 0000 0000 represents 0 decimal and 1111 1111 represents 255 decimals. In C, this is called the unsigned modifier.

THE UNSIGNED MODIFIER IN ACTION

The hardware rules for binary addition follow logical or mathematical principles: $0 + 0 = 0$, $1 + 0 = 1$, $0 + 1 = 1$, and $1 + 1 = 10$, which is referred to as a half adder in hardware. A full adder differs in that it has a carry input. You can modify the rules for the half adder to create a full adder (see Table 2).

The addition logic in a processor is built from full adder logic. For example, if you add the values 10 and 25 to it, you should get 35 as an answer.

```
0000 1010 (10) + 0001 1001 (25)
= 0010 0011 (35)
```

But, what if you add numbers that overflow the 8-bit values? Let's add 200 to 200, which equals:

```
1100 0100 (200) + 1100 0100 (200)
= 1 1000 1000 (400)
```

Because you only have an 8-bit result, the carryover gets lost. Al-

<i>#typedef</i>	CHAR8	char
<i>#typedef</i>	UCHAR8	unsigned char
<i>#typedef</i>	INT16	int
<i>#typedef</i>	UINT16	unsigned int
<i>#typedef</i>	INT32	long
<i>#typedef</i>	UINT32	unsigned long

Table 1—Depending on what CPU and compiler you use, the *typedefs* are defined differently.

though the carry flag is set and could be checked for a problem, that's not usually done for every addition in a program.

THE SIGNED NOTATION

If you want to work with negative numbers, you need another notation called two's complement notation. Table 3 illustrates the notation needed for an 8-bit register.

This notation is called signed in C, and it's the default mode if you don't use a modifier. And, because the hardware doesn't know the difference, the binary addition still takes place. The notation is just that, a notation. It gives something the programmer can use.

Original	Decimal	Invert	Invert +1
0000 0010	2	1111 1101	1111 1110
0000 0001	1	1111 1110	1111 1111
0000 0000	0	1111 1111	0000 0000
1111 1111	-1	0000 0000	0000 0001
1111 1110	-2	0000 0001	0000 0010

Table 4—With two's complement notation, you get the negative by inverting the bits and adding one.

Note that this two's complement notation is linear. With addition, each larger value is a result of adding one to the smaller value, and for subtraction, the converse is true. In fact, the computer's hardware uses two's complement notation when performing subtraction. The subtrahend is converted to two's complement and then added to the minuend.

To convert a number to its negative, invert all the bits and add one (see Table 4). Look back at Table 3 and do the two's complement conversions. Isn't zero interesting?

Two's complement notation forms the basis of signed and unsigned notation. Simply tell the compiler which representation the variable will take so the compiler can generate the proper assembly language instructions, but the CPU does not change its operation.

A LESSON IN SUBTRACTION

Take 35 and subtract 10. Using two's complement notation, invert each bit in the value 10, add one, and then add the resulting numbers. In-

A _{IN}	B _{IN}	C _{IN}	Sum	C _{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2—Turning a half adder into a full adder takes a bit of rule tweaking.

verting is easy for the hardware and the adding of one is accomplished by using the carry input to the full adder for the least significant bit. Essentially, what you're asking it to do is:

$$0010\ 0011\ (35) - 0000\ 1010\ (10) = 0001\ 1001\ (25)$$

And, here's how the hardware does it:

$$0010\ 0011\ (35) + 1111\ 0101\ (10\ \text{inverted}) + 1 = 1\ 0001\ 1001\ (25)$$

What happens if you add numbers that cause an overflow of your two's complement 8-bit values? Look at what happened by adding 100 to 100:

$$0110\ 0010\ (100) + 0110\ 0100\ (100) = 1100\ 1000\ (200)$$

Two hundred is the proper response if you're using unsigned notation (0 to 255), but the same bit pattern represents the -56 in signed notation, which means you had an overflow. It's your job to look after these types of problems. The compiler

0111 1111	127
0111 1110	126
0111 1101	125
.....	
0000 0010	2
0000 0001	1
0000 0000	0
1111 1111	-1
1111 1110	-2
.....	
1000 0011	-125
1000 0010	-126
1000 0001	-127
1000 0000	-128

Table 3—With two's complement notation, it's tricky characterizing some numbers in an 8-bit register.

doesn't know the values of the numbers you're adding unless they are constants. A good compiler will warn you if the constants are getting out of range, but only at compile time, not run time.

What happens if you subtract a large number? Let's subtract 100 from the number -100. You should end up with -200. However, if you do it wrong, this is what you get:

$$1001\ 1100\ (-100) - 0110\ 0100\ (100) = 1\ 0011\ 1000\ (56)$$

Or, you could try:

$$1001\ 1100\ (-100) - (1001\ 1011\ (100\ \text{inverted}) + 1) = 1\ 0011\ 1000\ (56)$$

As you see, the signed notation number system overflowed.

PRACTICE, PRACTICE, PRACTICE

Try some examples, so you're comfortable using the signed and unsigned systems and understand how they behave. Look at overflow and underflow in both addition and subtraction.

Next month, I'll discuss how to use these rules for fun and profit. ☛

George started his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and cofounded a design and manufacturing firm. Typical systems that George designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill Gates' new house. You can reach him at george.martin@worldnet.att.net.

RESOURCES

- G. Novacek, Testing 1, 2, *Circuit Cellar Online*, www.chipcenter.com/circuit/ July–October, 1999.
- J. DiBartolomeo, MS: EMI Gone Technical, *Circuit Cellar*, 91–94.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.