

## LESSONS FROM THE TRENCHES

George Martin

# Making The Numbers Work For You

????



Last month, I gave you a foundation in signed and unsigned binary numbers and how the compilers, assembly language, and CPUs support them. This month, I'd like to talk about how you can use these signed and unsigned numbers to your advantage.

### WORTHY OF REPEATING

Remember, it's good practice to use *#typedefs* so you're sure of the size of data being referenced. Let's assume you are writing code for a 16-bit CPU and your compiler defines a *char* as 8 bits, *int* as 16 bits, and *long* as 32 bits. Table 1 shows the *typedefs* that would be set up.

### STATING THE CASE

For the first example of unsigned number use, let's work with the following assumptions. You have a 12-bit A/D converter that is measuring a real-world parameter that is always

positive, such as oil pressure in an engine. The signal conditioning electronics produce a voltage ranging from 0 to 5 V for a given input range of oil pressure, say 0 to 100 psi. Inputting the 12-bit A/D into a 16-bit CPU is straightforward. The least significant bit of the A/D would be placed in the least significant bit on the CPU data bus, like this:

```
CPU Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A/D Bit x x x x 11 10 9 8 7 6 5 4 3 2 1 0
```

Bits 12–15 would be read in as all zeros. So, 0-psi pressure would be 0 counts, and 100-psi pressure would be 4095 counts. Any manipulation of the data, such as filtering, would be a pure mathematical process on the binary readings.

### WORKING THE WIRING

Now, let's assume the A/D converter is measuring acceleration. The readings have a range of 3 to –3 g. Let's also assume that the signal conditioning produces an electrical signal of –5 V for –3 g, 0 V for 0 g, and 5 V for 3 g. You could wire the 12-bit A/D a couple different ways. Here's one example:

```
CPU Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A/D bit s s s s 11 10 9 8 7 6 5 4 3 2 1 0
```

Here the bits marked *s* are wired to Bit 11, the sign bit. Table 2 is the signed number system values.

<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**—Here you can see the *typedefs* that would be set up for a 16-bit CPU with *char* as 8 bits, *int* as 16 bits, and *long* as 32 bits.

Input	Signal	Hex	Decimal
3 g	5 V	0x07FF	2047
0 g	0 V	0x0000	0
-3 g	-5 V	0xF801	-2047
-3 g	-5 V	0xF800	-2048

**Table 2**—Here you can see the signed number system values.

In the two's compliment number system, there are 2048 positive numbers and 2048 negative numbers. I've wrestled with how to scale the bits. Is it 3 g per 2047 or 3 g per 2048? I'll leave that up to you to decide.

## WORKING THE WIRING WITH OVERFLOW

If you add readings together, you could stand 16 full-scale (either positive or negative) readings before you overflow the register in the CPU. So, averaging eight readings could be done without concern for overflow. If you've got a system that needs some filtering, consider this input mapping:

```
CPU Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A/D Bit 11 10 9 8 7 6 5 4 3 2 1 0 x x x x
```

The bits marked x are wired as zero. The signed number system values can be seen in Table 3.

Each bit on the A/D converter has a binary weight of 16 in the CPU. In the two's compliment number system there are 2048 positive numbers and 2048 negative numbers.

In this scaling, you cannot start adding numbers together. You probably would overflow (or underflow) immediately. But if the first step was a scaling that reduced the magnitude, you're set up for it.

And, of course, you can wire the A/D inputs to any bit starting with 15 through 11. You would then have a scaling between the two examples I've described.

## A CASE STUDY IN TIMER INTERRUPTS

Now, I'll show you a second example of how unsigned numbers can be useful. Most microprocessors have a timer, and most timers can generate some sort of interrupt. Typically you

Input	Signal	Hex	Decimal
3 g	5 V	0x7FF0	32752
0 g	0 V	0x0000	0
-3g	-5 V	0x8010	-32752
-3g	-5 V	0x8000	-32768

**Table 3**—Here you can see more signed number system values for this example.

set up the timer and timer interrupt to give interrupts, say every 10 ms. In that interrupt routine, you either increment counters or do some processing. If you are incrementing counters and then testing these counters in other parts of the code to measure elapsed time, consider another approach.

Initialize the timer so it's free running and counting at a reasonable rate. The timer is a 16-bit number; if you used unsigned notation, it starts at 0x0000, counts up to 0xFFFF, and then rolls over to 0x0000. In your code, define timer variables as 32-bit unsigned integers and set them to 0. You will need two variables defined as unsigned integers named *NowTime* and *LastTime*. Set these both to 0 at startup.

A routine called *UpdateTimers* might look like Listing 1.

At the start of your code, zero out a long unsigned number called *Timer1*. Whenever you need to know how much time has passed, call *UpdateTimers*, and the value *Timer1* will be the number of *HW\_TIMER* ticks since you last zeroed out the number.

NowTime	LastTime	DiffTime
0x0010	0x0000	0x0010
0xFF10	0xFF00	0x0010
0x0008	0xFFFF	0x0010

**Table 4**—As long as you call the *UpdateTimers* routing before the hardware timer can roll over twice, math will keep returning the proper value.

## THE LOGIC BEHIND IT ALL

How does this work? As the hardware timer rolls over the unsigned integer, math keeps returning the proper value as long as you call the *UpdateTimers()* routing before the hardware timer can roll over twice (see Table 4).

So, *DiffTime* is accumulating timer ticks since the last update, and the *Timer1* is then totaling these accumulated ticks.

The upshot is that you can now measure time without interrupts. Just be sure to keep calling *UpdateTimers* often enough. Code that is not interrupt-driven is easier to design, test, debug, and support.

## NOW IT'S YOUR TURN

I've barely scratched the surface of what can be done with signed and unsigned numbers and different numbering systems. Don't be afraid to experiment. If you have interesting results, let me know, and I'll pass them along. Also, don't forget to look in the *Circuit Cellar* newsgroups. 📧

**Listing 1**—Here you can see what the *UpdateTimers* might look like. *UINT16 LastTime;*

```
UINT16 NowTime;
UINT16 DiffTime;
UINT32 Timer1;
void interrupt UpdateTimers (void){
    NowTime = (UINT16) HW_TIMER; // Read the hardware timer
    register
    DiffTime = NowTime - LastTime;
    LastTime = NowTime;
    Timer1 = Timer1 + (UINT32) DiffTime;}

```

*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](mailto:george.martin@worldnet.att.net).*

## RESOURCES

- G. Novacek, Testing 1, 2, *Circuit Cellar Online*, [www.chipcenter.com/circuit/](http://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](mailto:subscribe@circuitcellar.com) or  
[www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).