Understanding and Preventing Overflow (I Had Too Much to Add Last Night)

Jason Sachs●December 4, 2013 2 Tweet

- Software Development
- Tutorials

Happy Thanksgiving! Maybe the memory of eating too much turkey is fresh in your mind. If so, this would be a good time to talk about overflow.

In the world of floating-point arithmetic, overflow is possible but not particularly common. You can get it when numbers become too large; IEEE double-precision floating-point numbers support a range of just under 2^{1024}, and if you go beyond that you have problems:

```
for k in [10, 100, 1000, 1020, 1023, 1023.9, 1023.9999, 1024]:
    try:
        print "2^%.4f = %g" % (k, 2.0 ** k)
    except OverflowError, e:
        print "2^%.4f ---> %s" % (k,e)
```

```
2^10.0000 = 1024
2^100.0000 = 1.26765e+30
2^1000.0000 = 1.07151e+301
2^1020.0000 = 1.12356e+307
2^1023.0000 = 8.98847e+307
2^1023.9000 = 1.67731e+308
2^1023.9999 = 1.79757e+308
2^1024.0000 ---> (34, 'Result too large')
```

2^{1024} is a really, *really* big number. You probably won't ever need numbers like 2^{1024} unless you're working with combinatorics. Just for comparison, here are some ranges of numbers found in physical quantities:

This article is available in PDF format for easy printing

- the number of protons in the universe is estimated at $10^{80} \approx 2^{266}$
- the ratio between the Planck length and the size of the universe is 46 × 10^9 light-years / 1.62 × 10^{-35} m ≈ 2^{204}
- the ratio between the Planck time and the age of the universe is 13.8 billion years / 5.39 × 10^{-44} seconds ≈ 2^{202}

So double-precision numbers should be pretty safe.

Single-precision floating-point numbers go up to just below 2^{128} and are "vulnerable" to overflow in very large physical calculations. It's more likely you wouldn't use them because of precision, rather than range.

Those of us in the embedded systems world usually don't have the luxury of working with floating-point calculations. It costs extra, either in CPU execution time, or in $$$. We're resigned to using integer arithmetic. (Some of us actually like it!)

So the rest of this article is about overflow in integer arithmetic. What follows may surprise you.

Sources of overflow

The usual suspects: addition and multiplication

What can cause overflow in integer math? The usual suspects are addition and multiplication:

```
import numpy as np

a = np.int16(32767)
b = np.int16(2)
print "32767 + 2 = %s" % (a+b)
print "32767 * 2 = %s" % (a*b)
```

```
32767 + 2 = -32767
32767 * 2 = -2

-c:3: RuntimeWarning: overflow encountered in short_scalars
-c:4: RuntimeWarning: overflow encountered in short_scalars
```

Each integer format has a representable range:

- signed 16-bit integers support the range [-32768,32767]
- unsigned 16-bit integers support the range [0,65535]
- signed 32-bit integers support the range [-2147483648,2147483647]
- unsigned 32-bit integers support the range [0,4294967295]

And if you go outside this range, even temporarily, you need to be very careful. Most environments handle overflow gracefully, and give you "wraparound" or "modulo" behavior (32767 + 1 = -32768 in a signed 16-bit environment) where carry bits outside the range just disappear and you're left with the low-order bits corresponding to the exact result.

If you're programming in C with a compiler that meets the C99 standard, you may be surprised to learn that unsigned integer math is guaranteed to have modulo behavior under overflow conditions, but the behavior of overflow with signed integer math is undefined.

To would-be language lawyers, the relevant sections in C99 are:

Section 6.5 item 5:

> If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

Section 6.2.5 item 9:

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

(In practice, modern compilers such as gcc and clang tend to use modulo behavior anyway for basic arithmetic calculations, but be careful, as sometimes compiler optimizations will treat comparisons or conditional expressions incorrectly even though the basic arithmetic is "correct". Symptoms of this behavior can be really tricky to recognize; the LLVM website has some explanation of this, along with other bugbears of undefined behavior. If you want to be safe, use the `-fwrapv` compiler flag, which both gcc and clang support; this guarantees modulo behavior with all aspects of signed arithmetic.)

The historical reason for this little quirk in the C standard, is that in the olden days some processors used ones-complement representation for signed numbers, in which case arithmetic for signed and unsigned numbers is slightly different. Nowadays twos-complement is the norm, and the implementations of addition, subtraction, and multiplication are the same for signed and unsigned when using base word lengths for the results. But one of the main principles of C, for better or for worse, is to allow machine-specific behavior to maintain efficient code, so the standard was written with no guarantees for the results of signed arithmetic overflow. Languages like Java, on the other hand, have machine-independent definitions of arithmetic. Any arithmetic operation in Java will give you the same answer no matter which processor you are running it on. The price of this guarantee is that on some processors, extra instructions will be necessary.

By the way, if you're using C for integer math, be sure to `#include <stdint.h>` and use the typedefs it includes, such as `int16_t`, `uint16_t`, `int32_t`, and `uint32_t`. These are portable, whereas the number of bits in a `short` or an `int` or a `long` may vary with processor architecture.

If you are using the fixed-point features of MATLAB, beware that the default behavior of integer overflow is to saturate at the limits of the integer range. This avoids some of the problems of overflow, but it may not give you the results you expect, if you're used to languages that provide wraparound semantics.

Preventing overflow

Just because we can use `-fwrapv` in gcc or clang, and guarantee wraparound behavior, doesn't make it the correct behavior for an application program.

If I am controlling a valve, and I want the output to increase, and I keep adding 1 to a process control variable, so that I get 32765, 32766, 32767, -32768, -32767, etc. this creates a jump discontinuity which is BAD. The only machine-independent method of preventing this is to stay out of overflow. One approach is to upcast to a larger type size, check for overflow, and saturate the results:

```
#include <stdint.h>

int16_t add_and_saturate(int16_t x, int16_t y)
{
    int32_t z = (int32_t)x + y;
    if (z > INT16_MAX)
    {
        z = INT16_MAX;
    }
    else if (z < INT16_MIN)
    {
        z = INT16_MIN;
    }
    return (int16_t)z;
}
```

You could also do something like this to stay within the range a 16-bit type, but it gets a little tricky, uses more operations, and I'm not 100% confident I've got my code correct:

```
#include <stdint.h>

int16_t add_and_saturate(int16_t x, int16_t y)
{
   int16_t z = x+y;
   if ((y > 0) && (x > (INT16_MAX - y)))
   {
       z = INT16_MAX;
   }
   else if ((y < 0) && (x < (INT16_MIN - y)))
   {
       z = INT16_MIN;
   }
   return z;
}
```

Handling multiplication is similar, and is really only practical using the type-widening approach.

Subtraction?

There's a bug in the C code below; can you see why?

```
int16_t calcPI(int16_t command, int16_t measurement)
{
   int16_t error = command - measurement;
   /* other stuff */
}
```

The problem is with the subtraction. If `command = 32767` and `measurement = -32768`, the "real" value of error is 65535. And if `command = -32768` and `measurement = 32767`, then the "real" value of the error is -65535. Just as with addition, in subtraction the result of operating on two k-bit numbers is a (k+1)-bit result. There are a couple of ways of dealing with this to avoid incorrect results.

First, we can use a 32-bit intermediate calculation:

```
int32_t error = (int32_t)command - measurement;
```

This has the drawback that the more operations we add, the larger the range of output values becomes. But as long as we don't run into overflow in intermediate calculations, we're fine.

Second, we can saturate the error to the range of an `int16_t`, so that the limits are -32768 to +32767. This has the drawback that there is no difference between cases at the edge of saturation and deep into saturation. (For example: `command = 32767` and `measurement = 0` gives `error = 32767`, but so does `measurement = -32768`.) In reality we may want to maintain linearity throughout the whole range.

Third, we can modify the operation we're doing so the results are within limits:

```
int16_t error = ((int32_t)command - measurement) / 2;
```

This reduces the net gain but avoids both overflow and saturation.

Division?

Division is the ugly beast among the arithmetic operations. Fortunately, the overflow cases aren't too complicated.

If you are working in C, where division means taking an N-bit number and dividing by another N-bit number, there's only one example that can cause overflow, which we'll get to a little bit later. (And it's not divide-by-zero. If you're dividing n / d without excluding the case of d=0 first, you deserve whatever you get.)

Some processors have built-in division facilities, and compilers support these with "builtin" or "intrinsic" functions that directly map into the processors' division feature. Here you can do things like divide a 32-bit integer by a 16-bit integer to yield a 16-bit result, but you'll need to make sure you avoid dividing by a number that would create a quotient that won't fit in a 16-bit result. For example: 180000 / 2 = 90000, and 90000 is out of bounds for 16-bit numbers, both signed or unsigned.

Shifts?

Don't forget your shift operators `<<` and `>>`. These won't cause overflow, but in C, don't forget the pesky undefined and implementation-specific behaviors. Here the relevant section from the C99 standard is section 6.5.7 paragraphs 3-5:

> 3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.
>
> 4 The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 × 2^{E2}, reduced modulo one more than the maximum value representable in the result type. If E1 has a signed type and nonnegative value, and E1 × 2^{E2} is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.
>
> 5 The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2^{E2}. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

That's right! If you have a signed integer E1 containing a negative number, and you shift right, the results are implementation defined. The "sensible" thing to do is an arithmetic shift right, putting `1` bits into the most significant bits to handle sign-extension. But the C standard allows the compiler to produce a logical shift right, putting `0` bits into the most significant bits, and that's probably not what you wanted.

More recent languages like Java and Javascript give you *two* shift right operators: the regular `>>` operator for computing arithmetic shift right, and the `>>>` operator for computing logical shift right to produce an unsigned binary integer with zeros shifted into the most significant bits.

Is that it? Whew!

No, that's not it! What else is there? Well, there are the increment and decrement operators ++ and `--`, but with `-fwrapv` they should work with wraparound semantics as expected.

The rest of the overflow items fall into what might be called Pathological Cases. These all involve asymmetry between `INT_MIN` and `INT_MAX` that causes unwanted aliasing.

We talked about divide earlier, and the one integer divide overflow is when you take -32768 and divide by -1.

```
# Let's divide -32768 / -1
a=np.int16(-32768)
b=np.int16(-1)
a/b


-32768
```

D'OH! We should have gotten +32768 but that just won't fit into a 16-bit signed integer, so it aliases to the `int16_t` bit-equivalent value -32768.

The same thing happens with unary minus:

```
# Let's negate -32768
-a
```

```
-32768
```

Then there are the fixed-point multiplication flaws along the same line. Let's say you're doing Q15 math, where integers represent numbers of the form $\frac{k}{2^{15}}$. C code looks like this:

```
int16_t a = ...;
int16_t b = ...;
int16_t c = ((int32_t)a * b) >> 15;
```

Does this work properly without overflow? Well, it does, except for one single case. $-32768 \times -32768 = 2^{30}$, and if we shift that right by 15 we get $2^{15} = 32768$. But in a signed 16-bit integer, 32768 aliases to -32768. Ouch!

How do we deal with this?

Q15 multiplication

Well, the Q15 multiplication is a tough one. If you are *absolutely certain* you won't ever evaluate -32768 × -32768, go ahead and use the usual C code as is. One example is PI control loops where the gains are always nonnegative. Or if you know the range of one of the numbers doesn't include -32768 then you're fine.

Alternatively, if you're willing to have an extra shift right, you can do this:

```
int16_t a = ...;
int16_t b = ...;
int16_t c = ((int32_t)a * b) >> 16;
```

The shift-right-by-16 operation is usually faster in embedded systems than a shift by 15 by one instruction cycle, since it often maps to a "grab the high word" operation in assembly, which you usually get for free when storing memory. This operation represents $a \times b \times \frac{1}{2}$ if a and b are Q15 fixed-point integers, or $a \times b$ if one of the values is Q15 and the other is Q16.

Other than that, you need some way of saturating the intermediate product so it is safe to shift right:

```
int16_t a = ...;
int16_t b = ...;
int16_t c = sat30((int32_t)a * b) >> 15;
```

where `sat30(x)` limits the results so that they are within the range $-2^{30} + k_1$ to $2^{30} - 1 - k_2$, where k_1 and k_2 are any numbers between 0 and 32767. (Why this ambiguity? Because the least significant 15 bits don't matter, and certain processors may have quirks that allow you to execute code faster by choosing different values of k. For example, the literal value of $2^{30} - 1$ may not be possible to load in one instruction, whereas the literal value of $2^{30} - 32768$ `= 32767 << 15` may be possible to load in one instruction.)

Unary minus

The unary minus case is somewhat similar. We have a few alternatives:

If we are absolutely sure that the input cannot be -32768, we are fine leaving it as is.

Otherwise, we have to test for -32768 and convert to +32767.

Alternatively, in C we can use the bitwise complement operator `~`, because `~x = -x-1` for signed integers: `~-32768 = 32767`, `~0 = -1`, `~-1 = 0`, `~32767 = -32768`. This adjustment by 1 for all inputs may not be acceptable in some applications, but in others it is fine, representing a small offset, and the complement operation is often a fast single-instruction operation.

The same idea applies to absolute value; instead of

```
#define ABS(x) ((x) < 0 ? (-x) : (x))
```

we can use the bitwise complement operator:

```
#define ABS_Q15(x) ((x) < 0 ? (~x) : (x))
```

Alternatively, we'd have to special-case the test for -32768.

Just a reminder: `#define` macros are invisible to the compiler (because they happen in the preprocessor) and are also unsafe to use with arguments that have side effects e.g. `ABS(++x)` which would get incremented three times.

But wait, there's more!

Almost forgot one more thing! Everything we've discussed so far has been a single operation, and has included cautionary tales of woe.

With compound calculations that consist of multiple operations, there's one thing we can use in our favor:

If you have a compound calculation in fixed bit-width integer arithmetic (whether signed or unsigned) with modulo (wraparound) semantics, consisting only of these operations

- addition
- subtraction
- multiplication
- left shift
- bitwise AND
- bitwise OR
- bitwise XOR
- bitwise complement

(and *not* sign-extension, right shifts, or division), and the following conditions are true:

- intermediate computations overflow
- intermediate computations are not used for any other purpose
- the same compound calculation is performed using integer arithmetic without a limitation on bit width and produces a final result that does not overflow

then an amazing thing happens, which is that the final result in fixed bit-width integer arithmetic will still be correct. In fact, if we *do* provide saturation arithmetic on intermediate results, then whenever the saturation activates, we're likely to get a final result which is incorrect.

This kind of situation sounds kind of contrived and improbable, but it's really not, and shows up a lot in polynomial evaluation. And it results from the congruence relation we get from modular arithmetic. **If an arithmetic operation on k-bit numbers, from the list above, produces a k-bit result that may overflow with wraparound (modulo) semantics, it creates a result that is congruent to the true result, modulo 2^k.**

Let's go over a more concrete situation. Suppose we are computing $y = ax^2 + bx + c$ where a, b, c, and x are signed 4.12 fixed-point numbers, and a, b, and c are constants. This is equivalent to writing $Y = \frac{AX^2}{2^{24}} + \frac{BX}{2^{12}} + C = \frac{AX^2 + 2^{12}BX + 2^{24}C}{2^{24}}$ where A, B, and C, and X are 16-bit numbers. Suppose also that the values of a, b, and c are such that for any value of x within the valid 4.12 range (i.e. $-8.0 \le x < 8.0$), the result y is also within that range. Then it doesn't matter whether the intermediate calculations overflow, you'll still get the final result correct.

Okay, that's still a bit abstract. Let's suppose we're computing $y = f(x) = \frac{3}{16}x^2 - \frac{7}{16}x - \frac{121}{16}$.

For $|x| \le 8.0$, $f(x)$ has a minimum value of $-8 + \frac{35}{192}$ (near $x = \frac{4779}{4096}$) and a maximum value of $7.9375 = 8 - \frac{1}{16} = 127/16$ (at $x = -8.0$), so $f(x)$ won't produce a final result that overflows.

Now let's compute f(x) for $x = -8.0 = \frac{-32768}{4096}$ using two different ALUs. ALU1 is rather narrow, with 16-bit registers and a 32-bit product register, and ALU2 uses 32-bit registers and a 64-bit product register:

step	operation	ALU1		ALU2	
	x	-32768	-8.0Q12	-32768	-8.0Q12
1.	`t1p = x * x`	1073741824	64.0Q24	1073741824	64.0Q24
2.	`t2 = truncate(t1p,16)`	16384	64.0Q8	16384	64.0Q8
3.	`t3 = 3 * t2`	-16384	-64.0Q8 = -4.0Q12 [overflow!]	49152	192.0Q8 = 12.0Q12
4.	`t4p = -7 * x`	229376	56.0Q12 = 3.5Q16	229376	56.0Q12 = 3.5Q16
5.	`t5 = truncate(t4p,4)`	14336	3.5Q12	14336	3.5Q12
6.	`t6 = t3 + t5`	-2048	-0.5Q12	63488	15.5Q12
7.	`y = t6 - 30976`	32512	7.9375Q12	32512	7.9375Q12

A few comments here:

- Temporary results named with a "p" suffix are product register results; the remaining results are of regular length.
- The mysterious number -30976 is just the Q12 representation of $c = \frac{-30976}{4096} = \frac{-121}{16}$.
- `truncate(x,m)`, for an ALU with k-bit registers and a p-bit product register, is what we get when we select the k bits starting with bit #m: in other words, drop the m least significant bits of x, and then take the k least significant bits of the result, where k is the bit length of the ALU register. For an ALU with 16-bit registers, `truncate(x,m) = (int16_t)(x >> m)` and for an ALU with 32-bit registers, `truncate(x,m) = (int32_t)(x >> m)`. This operation does not introduce any sign-extension as long as k + m < p.
- Note that all the calculations are identical for both ALUs except in steps 3 and 6: ALU1 overflows, but ALU2 does not. The two results converge in the end.

Notice earlier I excluded right-shift from the list of "safe" operations, and I didn't mention anything about truncation, or about using two k-bit numbers to produce a 2k-bit product. Here I'll have to handwave a bit.

The part about right-shift that is unsafe has to do with sign-extension: we assume that we know the sign of a number based on the most significant bit that is stored. If overflow can be present, then we can't make any assumption about the sign of a number. So if you don't sign extend, or don't do anything with bits of sign extension, then you maintain congruence modulo 2^k.

If we multiply two k-bit numbers to produce a 2k-bit product, on the other hand, this is not an inherently "safe" operation with respect to using it in an intermediate calculation. Let's use k=16, with a = 1 + 65536 * m and b = 1 + 65536 * n, where m and n are unknown integers that we don't really care about, because all we need to keep for intermediate results is the bottom 16 bits, right? If we multiply the real values of a and b, we get (1 + 65536 * (m+n) + 65536 * 65536 * m * n). The bottom 32 bits of this result is 1 + 65536 * (m+n). Uh oh. If we want a 32-bit result, we really do have to know more bits. If we only care about the bottom 16 bits of the product, then we get 1, and we're ok.

So in this example we have to be careful; the "we don't care if intermediate results overflow" rule really only applies to steps 3, 6 and 7 (`y = (3*t2) + t5 - 30976`) and the rest of the steps need to be analyzed more rigorously to ensure there are no overflows with the range of given inputs. For the additions/subtractions/multiplications that use fixed bit width, we can just do them without worrying about overflow, as long as we know the final result is going to end up within valid limits.

I wish there were an easier way to explain this to beginners. I wish there were an easier way for experienced engineers to analyze this stuff, because it's kind of hard to be rigorous and make sure you get it right. Maybe someday someone will invent a kind of "overflow calculus" that is easy to use, where we can just turn the crank and not think, and it will tell us where we can have problems and how to fix them. Until then, stay watchful.

Further Reading

John Regehr at the University of Utah has a pretty interesting blog about embedded systems and static analysis and other whatnots. He has a few articles about undefined behavior in C and C++.

He's also one of the authors of an interesting article called "Understanding Integer Overflow in C/C++" which includes a sobering list of software programs (including PHP, SQLite, PostgreSQL, OpenSSL, Firefox, GCC, LLVM, and Python) for which potential integer overflows were discovered.

Summary

- Be aware of overflow!
- Know the range of inputs to arithmetic operations in your program
- Use compiler flags to ensure wraparound semantics (`-fwrapv` in clang and gcc)
- Use explicit saturation where appropriate
- Beware of the pathological cases involving `INT_MIN`
- Compound integer arithmetic calculations can tolerate overflow in intermediate results, as long as the final result is within range, and the intermediate calculations are part of a "safe list" (see detailed discussion above) and apply to operands and results that are fixed bit length.

Hope you all had a happy Thanksgiving, and remember, only *you* can prevent overflow!

Previous post by Jason Sachs:
How to Estimate Encoder Velocity Without Making Stupid Mistakes: Part II (Tracking Loops and PLLs)
Next post by Jason Sachs:
Efficiency Through the Looking-Glass

Comments:

- Comments
- Write a Comment
  Select to add a comment

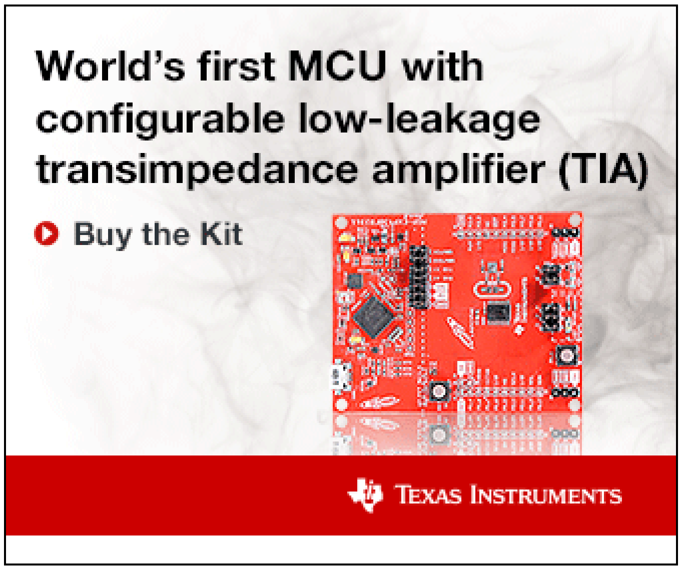
To post reply to a comment, click on the 'reply' button attached to each comment. To post a new comment (not a reply to a comment) check out the 'Write a Comment' tab at the top of the comments.

Sign in

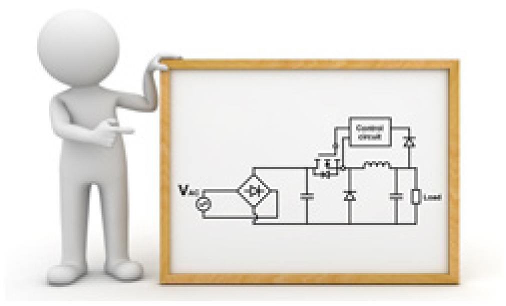
Username | Password | Sign in

[✓] Remember me

Forgot username or password? | Create account


World's first MCU with configurable low-leakage transimpedance amplifier (TIA)
Buy the Kit
Texas Instruments

You might also like...



TI has written the book on managing power in electronic systems

Subscribe to this Blog

Receive a notification when **Jason Sachs** publishes a new article:

Your Email | Subscribe

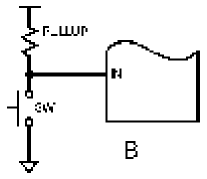
About Jason Sachs

Jason has 19 years of experience in signal conditioning (both analog + digital) in motion control + medical applications. He likes making things spin.

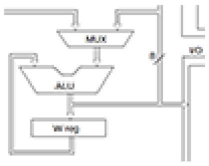
Popular Posts by Jason Sachs

- Musings on Publication — and Zero Sequence Modulation
- The Dilemma of Unwritten Requirements
- Python Code from My Articles Now Online in IPython Notebooks
- Organizational Reliability
- Someday We'll Find It, The Kelvin Connection

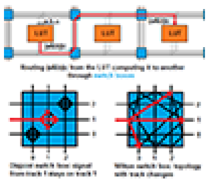
Blogs - Hall of Fame



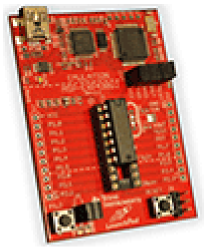
Introduction to Microcontrollers
Mike Silva



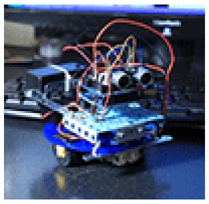
Important Programming Concepts (Even on Embedded Systems)
Jason Sachs



How FPGAs Work and Why You'll Buy One
Yossi Krenin

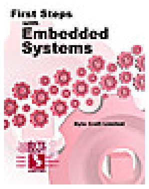


MSP430 Launchpad Tutorial
Enrico Garante

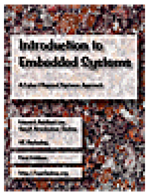


Arduino Robotics
Lonnie Honeycutt

Free PDF Downloads



First Steps with Embedded Systems



Introduction to Embedded Systems - A Cyber-Physical Systems Approach

PDF Embedded Systems – Theory and Design Methodology

All FREE PDF Downloads

Quick Links

- Home
- Blogs
- Tweets
- Forums
- Jobs
- #IoT
- Tutorials
- Books
- Free PDFs
- Vendors
- Code Snippets
- comp.arch.embedded

About EmbeddedRelated.com

- Advertise
- Contact

Social Networks

EmbeddedRelated.com DSPRelated.com
Electronics-Related.com FPGARelated.com