Compute √SQUARE ROOTS Fast

√ by Tim Paterson

Square roots have a number of possible applications in microcontroller systems ...

My particular application — as we discuss here — is in filtering input data used to calibrate the odometer in a road rally computer. After computing a best-fit line using a linear regression on a set of data points, square roots are needed to compute the distance of each data point from the line. If a data point is too far off the line, it is discarded on the assumption of human error and the line is recalculated.

Square Root on a PIC in the November '06 issue of *Nuts & Volts* (**www.nutsvolts.com**) demonstrated a simple algorithm for computing square roots. While the implementation was very compact, the algorithm has a significant performance issue — it gets extremely slow as the argument to the square root function gets large. This is because the algorithm sequentially computes each square starting at 1 until it finds the one closest to the argument.

Let's look at an alternative method of computing square roots. It has the advantage of taking nearly constant time, regardless of argument — about 150 microseconds for a 32-bit input on a 20 MHz PIC18. This makes it practical to compute square roots in a robot control loop running 100 times per second, for example. Code size is about 35% larger, but still well under 200 bytes.

√ Algorithm

This algorithm for computing square roots determines one result bit at a time, in a manner very similar to binary long division. In fact, it is so similar that a review of long division — decimal and binary — will be helpful to set the stage.

```
      6
     79
897)62518
   6279
  -5382
   8698
  -8073
    625
```

Figure 1. Decimal long division example.

Figure 1 shows a sample long division problem. The basic steps for each digit are:

- Make a guess of the digit.
- Multiply the trial digit by the divisor.
- Try to subtract that product from the current remainder.
  - If the product is larger than the remainder, make a new guess with a smaller digit.
  - If the subtraction produces a result larger than the divisor, make a new guess with a larger digit.

In the case of Figure 1, the guess for the first digit (estimated from 62÷8) initially turned out to be too large. The first trial digit and product are crossed out and a new set was tried.

The description for binary long division is the same, but the only two choices for the trial digit are 0 and 1. This makes it very easy to make the right guess, and also very easy to compute the trial * divisor product that is to be subtracted from the remainder. Figure 2 shows a sample binary division problem.

```
      1101
1001)1111001
   -1001
     1100
    -1001
      0110
     -0000
      1101
     -1001
       100
```

Figure 2. Binary long division example.

By definition of square root, if the divisor equals the quotient, then we have the square root of the dividend. The algorithm for square root takes the form of binary division where the divisor is continually changed to match the quotient.

The square root of an N-bit number has N/2 bits; in our example, we'll be taking the four-bit square root of an eight-bit number. From our binary division viewpoint, we'll be looking for the square root of the dividend. We start with a divisor at the midpoint of the possible result range: a 1 followed by N/2 – one zero; in our case, a 1 followed by three zeros.

```
          1
1xxx)01111001
     -1000
       111

          11
11xx)01111001
     -1000
       1110
      -1100
      -1000
```

Figure 3. First and second bits of square root are being calculated.

The first half of Figure 3 shows an example with the first bit calculated. Notice in the figure how we don't know the divisor yet, so I've substituted "xxx." The binary division procedure requires multiplying the guess by the divisor, so "0" is used for each "x" for now.

The second half of Figure 3 shows the partially-completed calculation of the second bit. We make a guess of 1, multiply the trial bit by the divisor (which is 11xx now to stay equal to our quotient), and subtract from the remainder. We must also account for what we should have subtracted when we computed the previous bit. Back then, we only subtracted 1000. Now we have a divisor of 11xx, so we should have subtracted 1100 before. I've put this make-up value to be subtracted on an additional line. Because we're shifted right one bit, it's written as 1000, but you can follow the "1" bit up the column and see it's in the right place.

So at this point, we have two values to subtract — 1100 and 1000 — whose binary total is 10100. This total amount to subtract is larger than our current remainder of 1110, so we must conclude the trial was too big — instead of 1, it must be 0.

The first half of Figure 4 shows the second bit set back to 0, and the calculation of the third bit. Everything is done the same: We subtract the product of the trial bit and the divisor, and also include a line to subtract the

```
          101
101x)01111001
     -1000
      1110
     -0000
      11100
     - 1010
     - 1000
       1010

          1011
1011)01111001
     -1000
      1110
     -0000
      11100
     - 1010
     - 1000
       10101
     - 1011
     - 1010
```

Figure 4. Third and fourth bits of square root being calculated.

value of the trial bit had it been set back when we computed the first bit. This time, the total of the two values to subtract (1010 and 1000) is 10010, while the current remainder is 11100. The remainder is big enough, so the trial bit is correct.

The second half of Figure 4 shows the calculation of the final bit. Note how that extra value we subtract (to correct for previously subtracting the wrong value) has more "1" bits in it. Each of the 1 bits corresponds to a subtraction for a previous quotient bit where we didn't subtract the bit we're working on now. So, for each 1 bit in the quotient (not counting the trial bit), there will be a corresponding 1 bit in the extra value to subtract to account for not subtracting it originally.

Once more: For each 1 bit in the quotient, there will be a 1 bit in the extra value to subtract. This is the same as saying the extra value to subtract equals the quotient (and divisor) before setting our trial bit. So, we subtract the divisor with the trial bit, and we subtract the divisor without the trial bit. Adding that up, we subtract 2 * divisor + trial.

That's it. Since the divisor and quotient are always equal, I'm going to start calling them the root. To calculate each bit of the root, try to subtract two times the root plus the trial bit from the current remainder. If it fits, the trial bit is part of the root; if not, that bit is zero.

√ Programming Square Roots

Listing 1 represents this algorithm as a function in C. You don't need to know C to understand it if you look in the sidebar for the explanation of the operators. This algorithm also includes the extra step of rounding the result based on whether the next bit would be 0 or 1. This function has been compiled and tested for both Windows and the AVR microcontroller.

LISTING 1. Fast square root algorithm in C.

```
typedef unsigned long UINT;
#define INT_BITS ((sizeof(UINT)) * 8)

UINT SquareRoot(UINT arg)
{
    UINT trial;
    UINT root;

    trial = 1 << (INT_BITS - 1);        // set up 100000... binary
    root = 0;                           // Inside loop, really root * 2

    do
    {
        trial >>= 1;                    // move trial to next position in root
        root |= trial;                  // combine trial bit into root

        if (arg < root)                 // does trial root bit fit?
            root ^= trial;              // no, remove trial bit
        else
        {
            arg -= root;                // root fits, remove it from arg
            root += trial;              // double the trial bit
        }

        root >>= 1;                     // move both root & trial to next
        trial >>= 1;                    //    position within arg

    } while (trial != 0);

    // Compute rounding
    // See if next bit computed would be 1; round up if so.
    if (arg > root)
        root++;

    return root;
}
```

LISTING 2. PIC version of fast square root, built on macros for 32-bit operations.

```
SquareRoot:
        clrf32  root            ; root = 0
        clrf16  trial           ; trial = 0x80000000
        clrf    trial+2         ;  "
        movlw   0x80            ;  "
        movwf   trial+3         ;  "
        bcf     STATUS,C; Clear C flag for following rotate
RootBit:
        rrf32   trial           ; trial >>= 1
        ior32   root,trial      ; root != trial
        cmp32   root,arg; if (arg < root) - C flag set if arg < root
        bnc     RootFits;  "
; Didn't fit, remove trial bit
        xor32   root,trial      ; root ^= trial
        bra     ShiftRight

RootFits:
        sub32   arg,root; arg -= root
        add32   root,trial      ; root += trial
ShiftRight:
        bcf     STATUS,C; Clear C flag for following rotate
        rrf32   root            ; root >>= 1 - Always leaves C flag clear
        rrf32   trial           ; trial >>= 1
        bnc     RootBit         ; if trial bit shifted out, we're done

#if     ROUND
; Compute rounding. Note that root always fits in 16 bits here,
; but it could round up to 17 bits.
        movf    arg+2,f         ; see if bits 17-23 are non-zero
        bnz     IncResult       ; if non-zero, arg is big, always > root
        cmp16   arg,root; if (arg > root) - C flag set if root < arg
        bnc     Done            ;  "
IncResult:
        inc24   root            ; root++
Done:
#endif  ;ROUND

        retlw   0
```

The function starts by initializing the variable trial which contains the current trial bit. The function is designed to be easily targeted for 16-bit or for 32-bit integers (or any other size), which explains the somewhat complicated-looking expression that initializes trial. I'll assume we're using 32-bit integers, in which case trial gets initialized to 0x80000000 — a 1 in the MSB and the rest 0. Note that the first thing that happens inside the loop is that trial is shifted right one bit, so the first trial bit value is 0x40000000.

In the main loop, the variable named root actually keeps the value of two times the current root. Combining trial into root gives us the amount to subtract from the current remainder (called arg) if it fits. If it doesn't fit, the exclusive-OR operation is used to clear the trial bit in root. If it does fit, we need to double the trial bit so that, as part of the new root, it will be root times two.

At the bottom of the loop, root and trial are each shifted right one bit. This moves them to the next bit position for which we'll be computing the root. Looking back at Figure 4, you can see how the subtraction for each bit is shifted right compared to the previous bit. Conveniently, when it's time to end the loop, this right shift of

Square Root Algorithms

I have never seen any description of this algorithm published before. I don't mean that it's original with me — in fact, I believe this algorithm is widely used in floating-point hardware. I became aware of the algorithm many years ago when reading about a particular computer architecture that included hardware square root.

I have an old Pentium manual that lists the time to perform a floating-point divide as 39 clock cycles, and the time to perform square root as 70 clock cycles – less than twice as long. The only way that could be happening is if the floating-point unit implements this algorithm in hardware.

For a processor that has hardware divide but no hardware square root, the most common way to compute square root is with Newton-Raphson iteration. This algorithm is very simple:

1) Make a guess at the root.
2) Divide the guess into the argument.
3) Average the quotient with the guess to make the next guess.

This method converges very fast; every iteration doubles the number of digits (or bits) of accuracy. The trick is to come up with a good initial guess. This is difficult for integer or fixed-point arithmetic, where it may take one iteration to get one bit correct. Then the subsequent iterations give you 2, 4, 8, and 16 bits of accuracy — typically five total iterations for a 16-bit root of a 32-bit number.

When using floating point arithmetic, it's easy to get an accurate initial guess because of the "normalized" format of the numbers. With an eight-bit multiply and addition, you can come up with a guess accurate to almost five bits. Then only three iterations are needed to achieve single-precision accuracy.

I used this technique when I wrote the x87 emulator for Microsoft in 1991. This program provided floating-point arithmetic back in the days before it was built into every processor; it hasn't been needed since the 486SX. It is still widely distributed, included as WIN87EM.DLL with every 32-bit operating system Microsoft has shipped.

While researching for this article, I found another square root algorithm on the Microchip website posted in Application Note TB040 for PICs that have hardware multiply. The general idea is to set a trial bit in the root, square the root, and see if it was too big. The algorithm calculates the result bit by bit, with one square (multiply) operation per bit. According to the app note, with a 32-bit input this method would take 200 microseconds at 20 MHz, and the code size appeared to be over 200 bytes. So it is both bigger and slower than the algorithm I present.

root is effectively the divide-by-two needed to convert the root * 2 that has actually been in root to the true root value.

I hand-compiled this function into PIC assembly language, partially shown in Listing 2. Following the lead of the previous article on square roots, I used macros extensively to encapsulate the 32-bit operations. For example, the ior32 macro actually requires eight instructions to perform a logical-OR on two 32-bit operands. This approach allows you to see the close relationship between the C and assembler versions of the program without getting caught up in the details of 32-bit arithmetic. The complete program listing with the macro definitions can be downloaded from the *SERVO* website (**www.servomagazine.com**).

C Operators

Here is an explanation of the operators used in Listing 1:

1 << (INT_BITS - 1)	With INT_BITS set to 32, this shifts a "1" bit left 31 bits, producing the value 0x80000000.
trial >>= 1	Shift trial right one bit.
root |= trial	root = root OR trial (logical OR).
root ^= trial	root = root XOR trial (exclusive OR). Reverses the effect of root |= trial, clearing the trial bit from root.
arg -= root	arg = arg - root
root += trial	root = root + trial
root++	root = root + 1

There is one optimization worth noting that is possible in assembly language but not in C. In C, the loop exits when the bit in trial has been shifted out and trial is now zero. In assembler, we can use the fact that the trial bit was shifted out into the carry flag. This saves us from testing the four bytes of trial to see if they're all zero.

In the PIC version, I have put the rounding code in a conditional block marked with #if ROUND. I wanted to make clear the optional nature of this section so it could be omitted if not appropriate for a given application. It adds very little execution time because it is out of the main loop, but it does add almost 20% more code. It also has the effect of allowing the result to exceed 16 bits, rounding it to 0x10000 if the input is big enough, so this must be taken into consideration if a purely 16-bit result is expected.

Another option you'll find in the full source code listing is to choose to assemble the program for the PIC16 or for the PIC18. This doesn't show up in Listing 2 because it only affects the macros. The PIC18 code is typically around 20% faster.

√ Performance Results

I compared the performance of this square root routine with the original one described in the November '06 issue. In both cases, I used PIC18-optimized code. I switched off the rounding code in my routine since the original didn't round. I used the Microchip MPLAB simulator with its stopwatch function to compute execution time, assuming a 20 MHz clock.

The execution time of my program ranges from 136 to 159 microseconds, for an average of about 150 microseconds. The variation is due to the different code paths inside the main loop on whether the trial bit fits or not. Code size is 134 bytes (22 bytes more when rounding is included).

The performance of the original square root program is totally dependent on the size of the argument. With an input of 500, it takes 155 microseconds; about the same as my program. For smaller arguments, it would be faster — as little as 12 microseconds — for an argument of 1. At the other end, it can take as long as 445 milliseconds (close to a 1/2 second!). Code size is 100 bytes.

I also compiled the C version of my routine for the AVR using the free WinAVR compiler package. Assuming a 16 MHz clock, execution ranged from 34 to 37 microseconds. Code size was 108 bytes.

In conclusion, this algorithm makes computing square roots entirely practical in microcontroller-based robotic projects, even in moderately fast service loops. No other approach comes close to the performance and code size of this algorithm. **SV**