EZ Square Root

30 July 2003

A simple method for finding the square root of an integer without using multiplication, division, or floating point arithmetic. An example program for the PIC series of processors is included.

This document:
www.bcpl.net/~jpower/ezsqrt.pdf

Web site:
www.bcpl.net /~jpower/powertech.html

POWER TECHNOLOGIES

Introduction. In the early days of mechanical computing, the complexity of the machines used was limited by their reliance on mechanisms with moving parts. This in turn placed a premium on simple algorithms which could extract results from numbers essentially presented as integers. Even the early electronic computers such as the ENIAC had to contend with the lack of floating point capability and with little storage capacity. One such algorithm for finding the square root of an integer is presented here. It uses only integer addition, subtraction, and comparision, and could be suitable for use in microcontroller firmware when a precision result is not needed. First the basic theorems on which the theory is based will be presented. A flowchart of the basic process will then be presented, followed by some tips on streamlining the program.

The algorithm for finding the approximate square root of an integer is based on Theorem 2 below. Subtracting successive odd integers, beginning with 1, from the given number N, determines the maximum number of terms in the summation, which is the square root of that number if the remainder becomes zero, and the greatest lower bound on the square root otherwise. In the latter case, an additional test can be applied to determine whether the square root is greater than the number of subtractions plus one half.

****

Theorem 1: The sum of the first N positive integers is N(N+1) / 2.

$$S1 \equiv \sum_{k=1}^{N} k = N(N+1)/2 \tag{1}$$

Proof:

Construct an array with successive columns holding k elements, k=1,2,...,N. For N=6, e.g.,

```
.....x
....xx
...xxx
..xxxx
.xxxxx
xxxxxx
```

There is a total of N squared positions in the array, and N elements on the diagonal. That leaves

$$(N^2 - N)/2$$

positions above the diagonal and an equal number below it. The total number of occupied elements in the array is the sum of the number on the diagonal and the number below it, giving

$$S1 = (N^2 - N)/2 + N$$

$$S1 = N^2/2 + N/2 = N(N+1)/2 \quad \textbf{QED} \tag{2}$$

****

Theorem 2: The sum of the first N odd positive integers is N squared.

$$S2 \equiv \sum_{k=1}^{N} (2k-1) = N^2 \tag{3}$$

Proof:

$$S2 = 2\sum_{k=1}^{N} k - N$$

From Theorem 1:

$$S2 = 2 \times [N(N+1)/2] - N$$

$$S2 = N^2 + N - N = N^2 \qquad \textbf{QED}$$

****

Alternate proof:

The interval between two consecutive squares is

$$\Delta_k = (k+1)^2 - k^2 = 2k+1$$

This is the $(k+1)$'st odd integer. Since summing consecutive intervals gives the final value,

$$\sum_{k=0}^{N-1} \Delta_k = N^2$$

but this is the same as summing the first *N* odd integers, which will also generate *N* ^ 2. Therefore the sum of the first *N* odd integers equals *N* ^ 2. **QED**

****

Theorem 3: The integer part of the square of the mid-value between *k* and *k*+1 is the sum of *k* ^ 2 and *k*. If *k* is the square root of *N*, this is *N* + *k*.

Proof:

$$(k+\frac{1}{2})^2 = k^2 + 2k(\frac{1}{2}) + \frac{1}{4} = k^2 + k + \frac{1}{4} \qquad \textbf{QED} \qquad (4)$$

****

Since this expression is the sum of two integers and the fraction 1/4, its value is not an integer. This means that if an integer value is given for *N*, the square root cannot be [*k* + 1/2] for integer *k*.

If consecutive odd integers are subtracted from *N* until *remainder* becomes zero on the *k*'th subtraction, the square root of *N* is *k* because this is the number of odd integers which sum to *N*.

If *remainder* is less than the [*k* + 1]'st odd integer, the next subtraction will yield a negative remainder. This places sqrt(*N*) between *k* and [*k* + 1]. The value can be further refined to either the upper or lower half of that interval by inspecting *remainder*. After *k* subtractions, we have "removed" *k* ^ 2 from *N*, leaving *remainder* = (*N* - *k* ^ 2) or *N* = *remainder* + *k* ^ 2. The maximum point of the lower half of the interval is at *y* = *k* ^ 2 + *k*. If *N* <= *y* or equivalently, *remainder* <= *k* then sqrt(*N*) is in the lower half of the interval with inequalities at each end of the region:†

$$k < \sqrt{N} < [k+1/2] \qquad \text{if } remainder <= k \qquad (5)$$

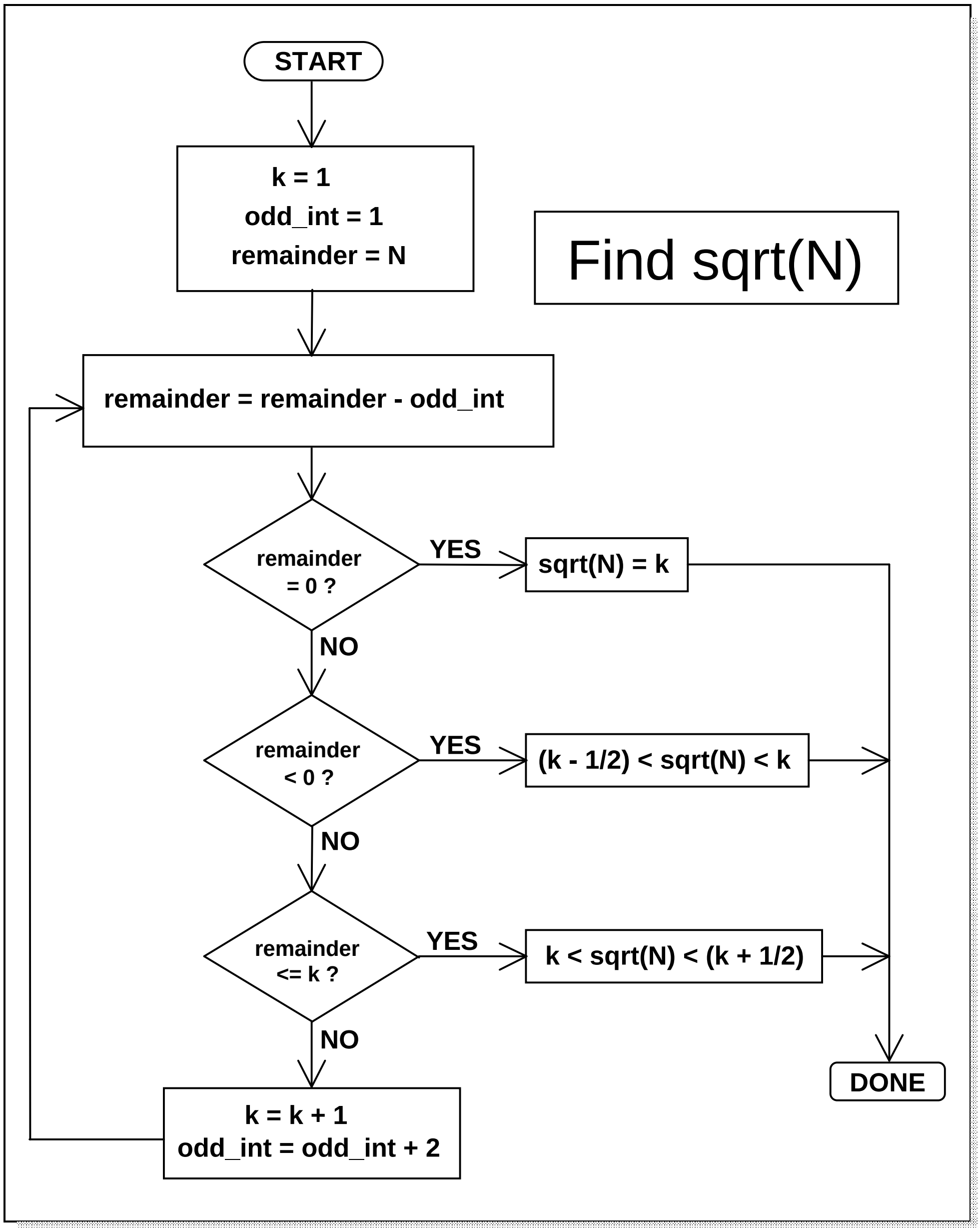
If *remainder* > *k*, then sqrt(*N*) will be in the upper half of the region.

$$[k+1/2] < \sqrt{N} < [k+1] \qquad \text{if } k < remainder < [k+1] \qquad (6)$$

In the following flow chart, the latter test is performed as a test for *remainder* < 0, knowing that on the previous iteration *remainder* > [*k* - 1]. The logic is as follows: on this iteration, the remainder is negative, so the remainder on the previous pass was less than *k*, but on the other hand, that remainder was also > [*k* - 1], otherwise the algorithm would have terminated with a solution. That places the previous remainder in the upper half of the previous region. As a result, sqrt(*N*) lies between [*k* - 1] + 1/2 and *k*, or between [*k* - 1/2] and *k*, using the current value of *k*. Using the previous value of *k* gives the result shown in (6). Since it is convenient to present the result as an integer with positive or zero offsets, the code

† Sqrt(*N*) cannot be [*k* + 1/2], and if it were an integer, it would have been detected in the test for zero.

presented later decrements *k* if *remainder* is less than zero. This makes the previous value of *k* the integer part of the answer.



Discussion. This method works only for integer values of *N* and can require a large number of subtractions. These limitations can be worked around. The integer *N* can be shifted left by *S* bits; after finding the square root of *N* * (2 ^ *S)*, the binary point can then be moved *S*/2 positions to the left from its initial position at the right of the low order bit. The number of subtractions can be reduced by anticipating the approximate range of the result and using appropriate starting values for *remainder*, *k*, and *odd_int*. Both of these methods are utilized in the PIC code example which follows.

Example. Statement of the problem:
Find the square root of an unsigned 8 bit integer *N* and put the result in register *result*. Since the integer

part of *result* needs no more than 4 bits, use the remaining 4 bits in *result* to hold a fractional part. If feasible, obtain additional bits of resolution.

Statement of the method to be used:

The result will be considered a fixed point fraction with the binary point between bits r4 and r3. This allows values from zero to **1111.1111** . The initial value *N* will be shifted 8 bits to the left, multiplying it by 256; this will shift *result* by 8/2 = 4 bits left, accomodating the format described above. The supplementary information supplied by equations (5) and (6) will provide two additional bits. Here's how: for purposes of the algorithm, *result* is an integer. The additional information about which half of the interval (upper or lower) holds the answer places the result either between *result.0* and *result.1* or between *result.1* and *result+1* (note the binary point). On average, the result will lie at the center of the appropriate interval, so if in the lower half, state the result as *result.01*; if in the upper half, use *result.11* as the best estimate. These expressions will have an error no larger than ± B'.01'; after taking into account the real position of the binary point in *result*, this becomes ±B'.000001' which is ± 1/64. After finding *result*, append as auxiliary bits either 01 or 11, based on (5) or (6).

If *N* = 0 or 1, sqrt(*N*) = *N*, so these values do not need to be computed; they can be loaded directly into *result* (and then shifted left 4 places). The minimum value of *N* is now 2, making the smallest value of *remainder* 0x0200 or 512. The largest number of iterations that can be guaranteed to be completed without exhausting *remainder* is now 22, since sqrt(512) = 22.6. Since the preloaded value of *result* is the next value to be used, take the starting value of *result* to be 23. The initial value of *remainder* is the result of the previous subtraction, consequently since 22 ^ 2 = 484, the normal starting value of *remainder* must now be decreased by 484. Observe that D'484' = H'01E4', and that the original shifted value of *remainder*, **| N | | 0 |**, always has a low byte equal to zero. Decreasing *remainder* by 484 reduces the high byte of *remainder* by 2 for the following reason. Subtracting the E4 part from zero always creates a borrow from the high byte, leaving D'28' in the low byte. Subtracting the 1 from the high byte reduces the high byte once more. As a result, *remainder* is initially loaded with **| N - 2 | | 28 |** and *result* is loaded with 23. *Odd_int* must now be preloaded as the 23nd odd integer, which is 2 * 23 - 1 = 45.

In the following code, the order of the tests for *remainder* = 0 and *remainder* < 0 are reversed for convenience. It is only important that the test for *remainder* <= *k* come last, since both of the other tests can also satisfy this criterion.

The auxiliary bits are returned in *W*, with a value of 0 indicating that the result of the algorithm is an exact integer. This does not mean that the square root of *N* is an integer, but rather that the fraction *result* is exact with no more binary digits.

The code: (register addresses depend on the processor used)

```
parm        equ   0x20        ; unsigned 8 bit value of N
result      equ   0x21        ; the answer goes here ( = k)
oddintlo    equ   0x22        ; max result = 255 => max odd int =
oddinthi    equ   0x23        ;   2*255 -1 = 509 = 2 bytes
remhi       equ   0x24        ; high byte of remainder
remlo       equ   0x25        ; low byte of remainder

sqrt:       movlw 2           ; test N for 0 or 1
            subwf parm, W     ; N - 2: if borrow, N = 0 or 1
            btfsc STATUS, C   ; C set = no borrow, must calculate it
            goto  the_long_way
            movf  parm, W     ; sqrt(N) = N
            movwf result
            swapf result, F   ; shift the binary point 4 bits left
            retlw 0           ; 0 => result is a perfect integer
the_long_way:
            movwf remhi       ; W still has N - 2
            movlw D'28'
            movwf remlo       ; preload remainder
            movlw D'23'
            movwf result      ; preload result
            movlw D'45'
            movwf oddintlo    ; preload current odd integer
```

```
            clrf  oddinthi
loop:       movf  oddintlo, W ; subtract the next odd integer
            subwf remlo, F
            btfsc STATUS, C
            goto  auxcy       ; if no borrow from low byte, continue
            movlw 1           ; must decrement high byte, since low
            subwf remhi, F    ;   byte borrowed - if this decrement also
            btfss STATUS, C   ;   borrows, then whole subtraction
            goto  carry       ;   borrowed  - we're done
auxcy:      movf  oddinthi, W ; low byte didn't borrow, so check high
            subwf remhi, F
            btfsc STATUS, C   ; if clear, there is a carry
            goto  no_carry    ; 1st two tests are reversed - see text
carry:      decf  result, F   ; we went past it, so back up
            retlw B'11000000' ; it's in the upper half - exit
no_carry:   movf  remlo, W    ; test remainder for zero
            iorwf remhi, W    ; we still need remainder
            btfsc STATUS, Z   ;   so don't change it
            retlw 0           ; its an integer! - exit
above_k?:   movf  remhi, W    ; test remainder for > result
            btfss STATUS, Z   ; if remhi != 0, then rem is  > result
            goto  next
            movf  remlo, W    ; (result-remlo): if C, then rem > result
            subwf result, W
            btfsc STATUS, C
            retlw B'01000000' ; it's in the lower half - exit
next:       incf  result, F
            movlw 2
            addwf oddintlo, F
            btfsc STATUS, C
            incf  oddinthi, F
            goto  loop
```

The maximum number of iterations will be 255 **-** 22 = 233, which is a minimum reduction of 8.6% or an average reduction of twice that. This can be improved if *N* is tested for midpoint by checking both high order bits (p7 and p8) for zero. Note that for *result* equal to 1/2 of maximum value, *N* achieves only 1/4 of its maximum. Since the goal is to halve execution time, the break occurs at *N* = 255/4 = 63 which is indicated by both high order bits being zero. If either bit is set, a new set of preloads is used in place of those found above.

The program can be modified easily to handle 16 bit values of *N* giving 8 bit integer values of *result*. All that is required is an additional register for the low byte of *N;* high and low bytes of *N* would then be loaded into *remainder* in place of **| N - 2| | 28 |.** The standard preload numbers: 1 for *result*, and 1 for *odd_int* would then be used since there is no advantage otherwise.