# A PWM Algorithm with Optimal Averaging Properties



## Summary

The following software shows an algorithm for creating PWM waveforms on the QED Board's output lines.

## Description

The QED Board's 68HC11 has several output compare channels that can be used to produce pulse width modulated (PWM) digital waveforms. However, there are times when more PWM output channels are needed than are provided by the 68HC11. This application note presents a software algorithm for creating PWM waveforms on any of the QED Board's output lines. This method has many uses, for example:

- motor speed control, light dimming, or control of heaters;
- creating any number of PWM channels simultaneously, as might be needed, for example, by a multi-channel PID temperature controller;
- software digital-to-analog conversion using a digital output line and a low-pass filter, with optimal settling characteristics;
- producing PWM waveforms synchronized to an external signal, for instance the AC power line;
- continuously variable speed control for stepper motors (as used in the QED stepper motor driver software, see MI-AN-039); or,
- converting a low-resolution DAC to a greater resolution (see MI-AN-057).

In short, this application note presents a method for creating PWM signals for any digital output line in software. The method can be made computationally efficient so that many PWM channels can be updated by a single QED Board. Each PWM signal is produced by putting out a string of zeros and ones on the digital line. All digital lines can be updated together in response to a clock-driven interrupt, or synchronized to an external signal via an external interrupt.

To create a PWM signal we need to approximate a target duty cycle by putting out a string of ones and zeros. We can create the waveform by putting out either a zero or a one on each clock tick, adjusting the number of zeros and ones so that after some number of clock ticks their average best approximates the target duty cycle. For example, to create a duty cycle of 0.3 the most straightforward method would be to repetitively put out a sequence of ten bits, three ones followed by seven zeros. The period would be ten, the duty cycle would be precisely 0.3, and the sequence would look like this (where spaces are inserted between the periods for readability):

...1110000000 1110000000 1110000000 1110000000 1110000000... Eqn. 1

Sequences like this have poor averaging properties because most of the signal lies in the lowest frequency, the inverse of the period. A sequence that can be averaged much more easily than the above one would distribute the ones and zeros more uniformly in time. While keeping a total period of ten, the following sequence has most of its energy expressed in a smaller pseudo-period of roughly three:

...1001001000 1001001000 1001001000 1001001000 1001001000... Eqn. 2

In this sequence two sequences of periods of three are alternated with one of period of four, 100100 with 1000. The total period is still ten, but for averaging purposes we can think of the waveform as being characterized by an average pseudo-period of 3.3333 and a duty cycle that is the inverse of this pseudoperiod, or 0.3. We must still average downstream with an averaging window of the true period of ten in order to get the most precise approximation of the duty cycle, but this sequence also gives a good short range average. Any sequence of three or four bits gives a good approximation of the duty cycle, whereas for the first sequence most short run averages are very poor approximations of the target duty cycle.

This application note presents two algorithms for producing pulse trains like Eqn. 2 above. Neither one requires much computation and can be implemented easily. The first, which is a counting algorithm, is described only briefly. It's a good algorithm in itself and it sheds light on the relationship between the pseudo period and the total period of a sequence. The second algorithm is an averaging algorithm. It is described in more detail because it's the more computationally efficient of the two.

**First Algorithm**

The first algorithm is illustratedby way of an example. Suppose we want to create a duty cycle of 0.28. ( This duty cycle is less than 0.5. For duty cycles between 0.5 and 1.0 the algorithm will be the same but we'll replace the duty cycle, r, with 1-r, and we'll interchange all zeros and ones! ) We note that the number 0.28 lies between the inverses of two consecutive integers, that is, between 1/3 and 1/4. These are found by noting that its inverse, 3.571428.., lies between 3 and 4. We'll call these two integers p and q, defined as,

$p = \text{Floor}(1/r)$ and, $q = p + 1$ if $1/r > p$ Eqn. 3

$q = 0$ if $1/r = p$

where Floor(x) is the greatest integer less than x. The sequence we produce should alternate pseudoperiods of p and q (that is, three and four), with the number of each chosen to approximate 0.28. How many of each should we choose? It must depend on the total number of bits we allow in our sequence. From considering that these packets of p and q bits each must average to 0.28 we find that the number of p-bit packets, n, and the number of q-bit packets, m, must satisfy the following two equations:

$pn + qm = T$ and, $(n + m)/T = r$ Eqn. 4

where T is the total period and r is the duty cycle (r=0.28). These can be solved as,

$m = \text{Int}[\, T(1-r\ \text{Floor}(1/r))]$ and, $n = \text{Int}[rT] - m$ Eqn. 5

where Int is a function that returns the nearest integer of its argument and Floor is a function that returns the greatest integer less than its argument. Eqn. 5 provides an n and m that will create a duty cycle that best approximates the target duty cycle for a given total period, T. For example, if T = 20 we find that n = 3 and m = 3 and the actual duty cycle is given by,

$r = (n+m)/T$ Eqn. 6

or 0.3 instead of 0.28. The duty cycle produced is always the best approximation that can be produced with a total period of T. The error is always less than 1/2T. We can not create a period of 0.28 exactly unless the total period is a multiple of 25 because the rational expression of 0.28 with the smallest denominator is 7/25. That sequence is,

...1000 100 1000 100 1000 100 1000 1000 100 1000 100 1000 100 1000... Eqn. 7

How do we put out the sequences? We shouldn't put out all n p-bit packets followed by all m q-bit packets. We should mix them up! So here's how we do it: First we put out a packet of the smallest number, n or m. Say it is n (in the sequence Eqn. 7 above n=3 and m=4). We put out a p-bit packet, then we put out k q-bit packets where k is defined as,

$$k = \text{int}(m/n) \quad \text{Eqn. 8}$$

Each packet is a one followed by p-1 or q-1 zeros. Then we subtract one from n and k from m. Then again we put out one p-bit packet and k=int(m/n) q-bit packets (recomputing k using the new n and m). Then we decrement n and m again. We keep doing this until n and m are zero. Now we will have put out the total period's worth of bits. Then we just start over.

Computational Load and Errors

To implement this algorithm we must do the computations of Eqn. 5, involving multiplication and division, whenever we change the duty cycle, and we must do the computation of Eqn. 8, involving integer division, for each packet we put out, and we have to do some decrements.

The error depends on the total period chosen, T, and for some T the error is zero. The error after T bits are put out is bounded by 1/(2T).

**Second Algorithm**

The second algorithm accomplishes the same result as the first, that is, it puts out sequences that best approximate a target duty cycle in the shortest runs but it is simpler. I'll present two flavors of the algorithm, then some error analysis for it, and finally a very fast implementation for the QED Board.

The Algorithm

The algorithm maintains a running average of the duty cycle produced and compares it to the target (desired) duty cycle at each tick of a clock. The two flavors differ only in how the running average is computed. Sequentially this is what is done in the first flavor:

1. We'll maintain a running average defined by: Running.Average = #Hits / #Attempts, and we'll initialize things as Running.Average = #Hits = #Attempts = 0. We update the Target duty cycle here.
2. Wait for a tick of the clock and test the running average against the target duty cycle.
3. If Running.Average < Target then output a one and update #Hits and #Attempts as,

   #Hits = #Hits + 1, #Attempts = #Attempts + 1 Eqn. 9
4. Otherwise output a zero and update only #Attempts as,

   #Attempts = #Attempts + 1 Eqn. 10
5. Update the running average as,

   Running.Average = #Hits / #Attempts Eqn. 11
6. If #Attempts >= T then go to step 1 and reinitialize all variables, else go to step 2.

Using this algorithm, the target duty cycle should not be changed asynchronously with the procedure, but it should be changed only whenever a total period of T pulses has been put out, and #Attempts is reset to zero. This algorithm will produce the same sequences as the first but with less computation. There is still a division required to recompute the running.average at each iteration.

A second flavor of this algorithm at first appears more complex, but for T=256 the division becomes unnecessary and the algorithm becomes computationally very efficient. This second flavor maintains the running.average using an exponential smoother:

1. Initialize the running average by: Running.Average = Target Eqn. 12
2. On each tick of the clock test the running average against the target
3. If Running.Average < Target then output a one and update the running average as,
   Running.Average = Running.Average * (T-1)/T + 1.0 * (1/T) Eqn. 13
4. Otherwise output a zero and update the running average as,
   Running.Average = Running.Average * (T-1)/T + 0.0 * (1/T)
   or simply,
   Running.Average = Running.Average * (T-1)/T Eqn. 14
5. Wait for the next clock tick and go back to step 2.

Here are a few comments on some of these steps:

Step 1. At the outset the running average is initialized to the target. If we wish to change the target duty cycle there are two ways of doing it. We can just change the target and leave the running average alone. The response of the system will be to overshoot or undershoot, that is, to put out a sequence of all zeros or ones for awhile, until the running average catches up. This is best if a downstream process is averaging the bit stream with the same exponential time as our running average. In that case, the output of *that* process will integrate the overshoot enabling it to react most quickly. On the other hand, if we want *our* bit sequence to adapt most rapidly to the changed target duty cycle and best approximate the new target quickly for short runs of bits, without overshooting or undershooting, then we should reinitialize the running.average to the target whenever we change the target.

Step 3. This kind of comparison, a simple less than, lets us go down all the way to zero duty cycle and up to a duty cycle of just less than 1.0. The maximum duty cycle will always be less than 1.0 though. (However, in the T=256 implementation provided below we can force a duty cycle of 1.0 also.)

Steps 3 and 4. This kind of averaging produces a total period of about T bits for most duty cycles, but as the target duty cycle approaches within 1/T of zero or one the period stretches out towards 2*T. (Maybe, for real number math, but maybe not, I'll have to think about this point some more.)

For a running average with period, T, of 256 we can do away with the division. That's why we focus on using the exponential smoother for the running average rather than by taking a straight average.

Error Bounds

We'll compute the error bounds as follows: Assume that the target duty cycle is less than 1/2. Suppose that the running average has just become less than the target. We'll put out a one and the new running average becomes,

Running.Average = Running.Average * (T-1)/T + 1/T Eqn. 15

which is greater than the target. We require m iterations of putting out zeros and reducing the running average as,

Running.Average = Running.Average * (T-1)/T Eqn. 16

before it is again less than the target. The total number of iterations, 1 for putting out the one and m for putting out zeros, m+1, is given by,

m+1 = - ln[ 1+ 1/(Target*(T-1))] / ln[ 1 - 1/T ] Eqn. 17

This is the pseudoperiod, or average separation between one pulses. If the target duty cycle is greater than 1/2 then the pseudoperiod is given by,

m+1 = - ln[ 1+ 1/ ( ( 1- Target)*(T-1) ) ] / ln[ 1 - 1/T ] Eqn. 18

and it is the average interval between zero pulses. The inverse of the pseudoperiod is the actual duty cycle produced. The average error is computed as,

Error = 1/(m+1) - Target Eqn. 19

(using first order approximations for the logarithms), and is approximately given by,

Error = (1 - Target ) / (2T) Eqn. 20

for targets of less than 1/2 and,

Error = Target / (2T) Eqn. 21

for targets of greater than 1/2. These equations are first order approximations. Of course, if the number of pulses averaged downstream are less than T then the averaging error also contributes to this error. In this case, if we call the number of pulses averaged S, the error bound is given by,

Error = Target / (2T) + 1 / (2S) Eqn. 22

These error bounds are for the maximum error expected of any arbitrary target duty cycle. For many duty cycles the error can be very small. For example, whenever the target duty cycle can be expressed as a rational number with denominator equal to the total period, T, then the duty cycle is produced exactly. So all of the duty cycles, 1/T, 2/T, 3/T, ... (T-1)/T, are produced exactly.

Low Pass Filtering

The pulse stream produced can be low passed filtered to produce a DC level equal to the PWM duty cycle. The ripple on the DC level depends on the time constant of the filter used. For a single pole filter (a single RC filter stage) the ripple is decreased by a factor of two for every factor of two increase in filter time constant. Suppose the interval between updates is •t, the filter time constant is •, and we designate the full scale output of the digital output as FSO. Then if we set •=•t the peak-to-peak output ripple is approximately FSO/2. The output ripple decreases in proportion to •. For every doubling of • the output ripple decreases a factor of two, so that with •=T*•t the output ripple is decreased to FSO/2T, and for T=256 the peak-to-peak output ripple is limited to FSO/512.

Using Period = 256

For integer implementations T can be any power of two, up to the square root of the size of the integers used. For 16-bit arithmetic it is convenient to use T=256, or 0100 hex, and to accept a resolution on the duty cycle of 1 part in 256, with duty cycles of 0, 1/256, 2/256, 3/256, ... up to 255/256 (and even 256/256

as shown below). The target duty cycle can be considered an 8-bit number converted to a 16-bit unsigned integer by padding a zero byte to its right side. That is, we represent the duty cycles as a 16-bit number with only the top 8 bits being significant; 0000 corresponds to 0.0 and FF00 corresponds to just less than 1.0. Then the computation of Eqns. 13 and 14 is easy: The factor,

Running.Average * (T-1)/T Eqn. 23

is given by

Running.Average - Running.Average/256 Eqn. 24

with the divide by 256 (0100 hex) done just by manipulating bytes. The addition of the increment, 1.0 * (1/T), is performed by adding 255 ( 00FF hex ). (255 gives better rounding properties in the addition than 256 does because we're using floored, i.e. truncated, arithmetic throughout). Because our resolution is 1 part in 256 and our total period is 256 the algorithm produces the duty cycles exactly. We can measure these exact duty cycles with a boxcar averager so long as our averaging period is an even multiple of 256 pulses. If we try to get better resolution by providing more significant bits in the target value (by including a nonzero lower byte) then the errors in the actual duty cycles produced are still less than 1 part in 512 as explained earlier. The pseudoperiod is always optimal; it is as small as possible. The pseudo period is given by,

m+1 = 1 / r for r <= 0.5 and, Eqn. 25

m+1 = 1 / (1-r ) for r >= 0.5

The following is a snippet of MacFORTH code that implements the algorithm. The routine's numbers are in hexadecimal. MacFORTH used 32 bit arithmetic of which we use only the lower 16 bits. Division is floored so the divide by 0100 hex is equivalent to a byte shift. On a QED Board we would do a byte shift (probably by just fetching with a C@ instead of with a @). The variable TARGET is in the range { 0000, 0100, 0200, 0300, ... FD00, FE00, FF00 }. AVERAGING.LENGTH is the total number of pulses created and averaged.

```
TARGET @ AVERAGE  !
0 #HITS !
AVERAGING.LENGTH @ 0
DO
     AVERAGE @ TARGET @ <
     IF
          ." 1 "  1 #HITS +!
          AVERAGE @ DUP 0100 / - 00FF +   AVERAGE !
     ELSE
          ." 0 "
          AVERAGE @ DUP 0100 / -          AVERAGE !
        ENDIF
LOOP
#HITS @ FLOT AVERAGING.LENGTH @ FLOT F/ 10000 FLOT F* FIXX
CR ." Actual.Duty.Cycle = " .
```

In addition of putting out exact duty cycles of 0/256, 1/256, 2/256, ... 254/256 and 255/256 corresponding to the 8-bit unsigned integers 0, 1, 2, ... 254, and 255 we can also put out 100% duty cycle (256/256) by choosing a target in the above algorithm of FFFF hex (or any other number from FF01 to FFFF).

The following code implements the above algorithm on the QED Board. The first word is written in high level FORTH and the following word is written in assembly code.

```
ANEW BIT.PWM

DECIMAL

\ The following two variables must be defined consequitively; the routine
\ assumes that the address of Average.PWM immediately follows that of Target.PWM

VARIABLE Target.PWM\ Holds the target PWM as an 8-bit number in the high order
                   \ byte.  The contents of the low order byte are irrelevant.
                   \ Fetch or store to this variable using C@ and C!.

VARIABLE Average.PWM \ Used internally by the algorithm; holds a running average PWM.
                   \ To update the PWM immediately set both Target.PWM and Average.PWM
                   \ to the new value.  To update the 256-bit long integral of the
                   \ output most quickly do not modify Average.PWM when Target.PWM
                   \ is reset.  The Average.PWM is set by setting its high order byte
                   \ to the desired PWM (0-255) and setting its low order byte
                   \ to 127.

: Update.PWM  ( xaddr -- Flag )
        \ xaddr is the address of Target.PWM and xaddr+2 is the address of
        \ Average.PWM, both as 16-bit unsigned integers.
        \ Flag is the bit to be outputted.
        \ Each time Update.PWM is called Flag is set to either true or false
        \ to maintain the proper average value for the PWM output.
        [ BASE @ HEX ]          \ compile this word in hexadecimal
        XDUP 2 XN+
        Locals{ x&Average.addr x&Target.addr }
        x&Target.addr @ FF00 AND x&Average.addr @ U>
        IF
                x&Average.addr @ x&Average.addr C@ - 00FF +         x&Average.addr !
                TRUE
        ELSE
                x&Average.addr @ x&Average.addr C@ -                x&Average.addr !
                FALSE
        ENDIF
        [ BASE ! ]        \ restore prior number base
        ;

CODE *Update.PWM   ( xaddr -- Flag )
        \ xaddr is the address of Target.PWM and xaddr+2 is the address of
        \ Average.PWM, both as 16-bit unsigned integers.
        \ Flag is the bit to be outputted.
        \ Each time Update.PWM is called Flag is set to either true or false
        \ to maintain the proper average value for the PWM output.
        BASE @ HEX                          \ assemble in hexadecimal
        02 IND,Y LDD                        \ Get the Target.PWM address
        02 IMM ADDD                         \ and increment by 2 and push it on the stack
        DEY DEY 00 IND,Y STD                        \ as the Average.PWM address.
        02 IND,Y LDD                        \ Fetch,
        DEY DEY 00 IND,Y STD                        \ and push the page too.
```

```
02 IND,Y LDD                          \ Then XDUP the Average.PWM xaddress
        DEY DEY 00 IND,Y STD
        02 IND,Y LDD
        DEY DEY 00 IND,Y STD
CALL @                                      \ @ Average.PWM and push it
00 IND,Y LDAB CLRA                    \ get Average.PWM/256
DEY DEY 00 IND,Y STD                        \ push Average.PWM/256
02 IND,Y LDD 00 IND,Y SUBD  \ replace Average.PWM/256 on tos with
00 IND,Y STD                          \ Average.PWM - Average.PWM/256
0A IND,Y LDD DEY DEY 00 IND,Y STD \ put Target address on tos
0A IND,Y LDD DEY DEY 00 IND,Y STD \ put Target page on tos
CALL @                                      \ @ Target.PWM and push it
00 IND,Y LDD                          \ get target from stack
CLRB                                  \ zero out the low order byte
04 IND,Y CPD                          \ Target.PWM - Average.PWM
HI IF,                                \ if Target.PWM > Average.PWM
        TRUE IMM LDD 0C IND,Y STD \ set output to true and store it in place
                                      \ of target.addr on stack
        CLRA 02 IND,Y ADDD            \ add 255 to Average.PWM - Average.PWM/256
        0A IND,Y STD                  \ and store it in place of the target page on
                                      \ the stack
ELSE,
        FALSE IMM LDD 0C IND,Y STD          \ else just set output false
        02 IND,Y LDD 0A IND,Y STD     \ and store Average.PWM - Average.PWM/256
                                      \ in place of the target page on the stack
ENDIF,
06 IMM LDAB ABY                       \ drop top three stack cells
                                      \ we now have ( flag\new.avg\avg.xaddr )
CALL !                                \ store to Average.PWM
RTS                                   \ return from subroutine
BASE !                                \ restore prior number base
END.CODE
```

Michael Dorman's comments after using the algorithm to code pulse rates for stepper motor drivers:

1. "If we choose a target between FF01 and FFFF in order to put out 100% duty cycle, to avoid skipping the first pulse we must initialize the average to FF00, not FFFF."

2. "For code cleanliness, target duty cycles in the range of 0 <TD <= 00FF should be clamped to 0000; otherwise you have a nonzero target that is never updated."

3. "For a synchronous system like our stepper motor controller where we switch to a new speed just after a step (a 'hit'), you should initialize the average to the Target + 00FF. Otherwise you get a step on the second tick regardless of the target duty cycle !"



Mosaic Industries

5437 Central Ave Suite 1, Newark, CA 94560 Telephone: (510) 790-8222 Fax: (510) 790-0925