Microprocessor Implementation of a Time Variant Floating Mean Counting Algorithm

Russell Kevin Huffman
Senior Engineer
Westinghouse Savannah River Company

1. Introduction

Rate estimation of nuclear pulses emitted from nuclear detectors has been well documented in papers written as early as 1965 [1] to as recently as 1990 [2]. It is well known that pulses emitted from a nuclear detector can vary with time [2] and an accurate estimate of the count rate must be based on a sufficient number of pulse counts within a sample period as well as the recent history of pulse counts acquired in previous sample windows to accurately estimate the current rate.

This paper will review the attributes of three popular counting methods and show the implementation of one of these methods, the floating mean algorithm, on an embedded controller system. The software discussion will look at how to apply the chosen algorithm on two popular platforms: the Motorola 68HC11 and the Intel 805X series embedded controllers.

2. Mathematical Algorithms

2.1 Quasi-exponential algorithm

The quasi-exponential algorithm is a digital adaptation of the traditional analog RC type rate meter [2]. The RC type rate meter was based on a detector output charge being applied to the capacitor and then dissipating off the capacitor in a time period (*T*) determined by the resistor value employed. The resultant output is a most recent value weighted mathematical expression in which a residual "history" of previous pulses is summed with the most recent value. Mathematically, the output (rate) can be expressed as:

$$(1)\quad Rn = a/T \sum_{i=1}^{n} (1-a)^{n-i} N_I$$

where *a* is effectively the weighting factor applied to each measurement (analogous to the time constant of an RC circuit), *T* is the sample interval for a single measurement, N_i is the number of pulses in the ith measurement, *n* is the total number of measurement sets being considered, and R_n is the resultant rate. For sufficiently small values of *a*, it can be shown that the expression can be reduced to a recursive algorithm in which the current rate is a weighted function of the previous rate and the current measurement [2].

2.2 Floating mean algorithm

The floating mean algorithm does not lend itself well to analog implementation. The algorithm predicts the current rate as an equally weighted mean of the last *m* measurements. Mathematically, the floating mean may be expressed as:

$$(2)\quad R_n = 1/(mT) \sum_{i=q}^{n} N_i$$

where R_n is the current rate estimate, *m* is the number of measurements included in the mean, *T* is the time interval for a single measurement, N_i is the number of pulses measured in the *ith* measurement, *n* is the total number of measurements acquired (thus the *nth* measurement is the most recent measurement), and *q* is equal to 1 when $n < m$ and *q* is equal to $n-m+1$ when $m \geq n$ [2].

For the purposes of microprocessor implementation, the pulses may be stored in a circular buffer with the $m + 1$ sample discarded as new measurements are acquired.

2.3 Weighted moving average algorithm

The weighted moving average algorithm is a hybrid between the quasi-exponential algorithm and the floating mean algorithm. In simplest terms, the weighted moving average algorithm is a weighted value version of the floating mean in which preference is shown toward more recent measurements. The weighted moving algorithm assumes that the more recent measurements are closer approximations to the true rate while being bound by a finite amount of historical measurements. The weighted moving average algorithm can be expressed as:

$$(3)\quad R_n = 2/((k(k+1)T)\sum_{i=q}^{n}(n-k+1)N_i$$

where R_n is equal to the current rate estimate, k is the number of measurement sets being considered in the equation, n is the total number of measurements that have been acquired (thus the *nth* measurement is the most recent), N_i is the number of pulses measured in the most recent sample period, T is the sample collection period for a single measurement, and q is equal to 1 when $n<k$ or q is equal to $n-k+1$ when $n \geq k$[2]. As in the floating mean algorithm, the microprocessor implementation of this algorithm involves a circular buffer of storage size k. The $k+1$ sample is discarded as a new sample is acquired.

2.4 Comparison of Counting Methods

Because incoming pulses conform to a Gaussian distribution, the calculated count rate actually represents the probabilistic count rate with variations from the mean determined by the number of points acquired [4] An evenly weighted mean calculation of multiple measurements tends to contain the estimated count rate and limit deviations from the mean. A time dependent weighting scheme, such as those employed in the quasi-exponential (1) and weighted mean algorithms (3), produce predicted count rates that deviate more from the mean than an evenly weighted mean calculation but less than the deviations expected from single measurement predictions. The exact deviations expected from weighted mean algorithms are determined by the weighting factors employed. Use of the floating mean algorithm (2) sacrifices response time to transients due to prior measurements that must be factored into the mean. For large transients and a measurement set of size k, it will require k measurement sets to be taken before the reported value is within the predicted standard deviation of mean.

As shown in Table 1, the steady state weighted mean and floating mean values closely approximate each other for predicted count rates. The method chosen to test the algorithms involved taking two times the standard deviation for each mean, producing two numbers (mean + 2 standard deviations, mean - 2 standard deviations) and alternately using the high and low numbers as the measured data set. A summation of ten data sets for each mean was taken. The time interval was assumed to be 1 second and the weighting factor used in the quasi-exponential algorithm was 0.2.

	Data Range		Ca lculated Results		
Mean Count Rate	Min Expected Rate at 2Sigma	Max Expected Rate at 2Sigma	Quasi-exp	Floating Mean	Weighted mean
10.0	9.2	10.8	9.0	10.0	10.1
20.0	18.8	21.2	18.0	20.0	20.1
30.0	28.6	31.4	26.9	30.0	30.1
40.0	38.4	41.6	35.9	40.0	40.1
50.0	48.1	51.8	44.8	50.0	50.2
60.0	58.0	62.0	53.8	60.0	60.2
70.0	67.8	72.2	63.0	70.0	70.2
80.0	77.7	82.3	71.6	80.0	80.2
100.0	97.4	102.6	89.5	100.0	100.2
200.0	196.3	203.7	178.9	200.0	200.3
300.0	295.5	304.5	268.2	300.0	300.4
400.0	394.8	405.2	357.6	400.0	400.5
500.0	494.2	505.8	446.9	500.0	500.5
600.0	593.7	606.3	536.2	600.0	600.6
700.0	693.2	706.8	625.5	700.0	700.6
800.0	792.7	807.3	714.8	800.0	800.7
900.0	892.3	907.7	804.1	900.0	900.7
1000.0	991.8	1008.2	893.4	1000.0	1000.7

TABLE 1 COMPARISON OF MATHEMATICAL ALGORITHMS

3. Pulse Collection Methodology

It is most often assumed that the above algorithms are implemented by acquiring an unknown number of pulses in a fixed period of time. Acquiring measurements at a fixed time interval produces rate meter estimates at a known frequency, however it is possible to acquire an insufficient number of pulses in a sample window to produce accurate rate estimates. The statistical errors produced in fixed time sampling can be minimized by narrowly bounding the operating range and adjusting the sample period accordingly. This approach works well if the operating range is sufficiently limited and the measured source is between 25% and 75 % of the operating range. For values outside of these bounds, the system may have insufficient information to predict the correct rate (at the low end of the scale) or may unnecessarily sacrifice response time for extra data points not required (at the upper end of the range).

In practice, any or all of the above algorithms could be implemented by acquiring a fixed number of pulses over a variable time period. Using a fixed number of pulses has the advantage of ensuring a statistically significant number of pulses is acquired for the measurement set[4]. A fixed pulse algorithm has the disadvantages in that, for low count rates, the estimated rate update may be unacceptably slow while for very high count rates, the device used to display the estimated rate may vary too quickly to produce readable data.

A third approach is to use a bounded fixed pulse counting algorithm in which minimum and maximum time limits are imposed on the pulse acquisition algorithm. This approach attempts to acquire a significantly number of pulses, within a bounded operating range, while being

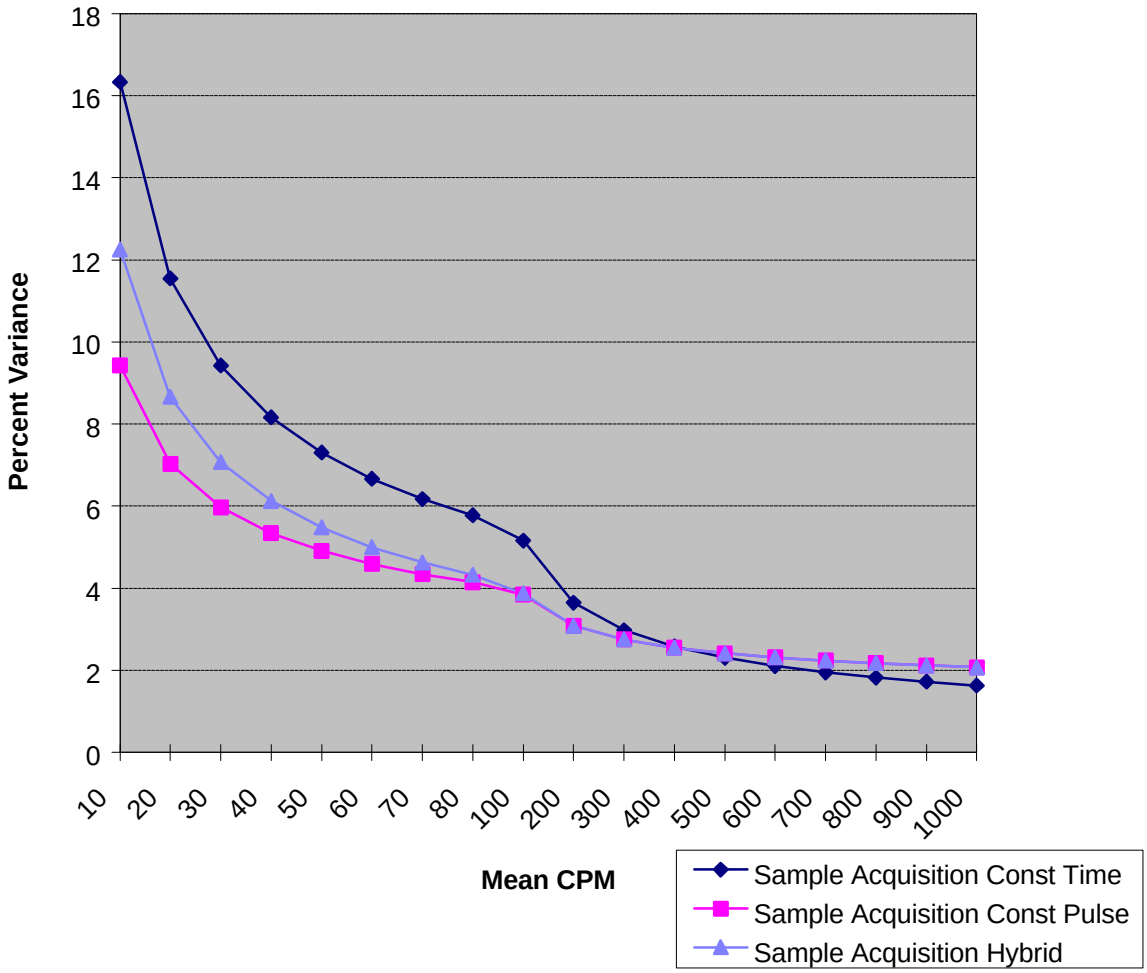
confined to minimum and maximum rate estimate updates. The bounded fixed pulse method produces rate estimates that closely approximate those obtained with a pure fixed pulse method while mitigating the disadvantages.

As shown in Figure 1, the constant pulse counting method produces the most statistically accurate count rate mean value. The figure was derived by considering the counting requirements over the range of 10 counts per minute to 1000 counts per minute. The collection time for the constant pulse and the count value for the constant time methods overlap at mid-scale (500 cpm). The percent variance is calculated as follows:

$$\%\ \text{Variance} = 100 \times \frac{(\text{Mean CPM} + 2\text{Sigma dev}) - (\text{Mean CPM} - 2\ \text{Sigma dev})}{\text{Mean CPM}}$$

The constant pulse method employed a fixed count acquisition of 7 pulses. The fixed time acquisition acquired pulses for 1 second. The hybrid method employed a fixed pulse method that was time constrained between 4 seconds and ¾ seconds. While the hybrid approach didn't achieve the percent variance of the constant pulse algorithm, the acquisition time (at 10 cpm) was reduced from 42 seconds (fixed pulse) to 4 seconds (hybrid).

Figure 1 Percent Variation from Mean Value for Different Counting Methods



A second advantage to the hybrid approach (as compared to a fixed time method) is that the predicted count rate is updated based on sufficient information to produce statistically accurate results. In a fixed time method, the response time of the system is fixed for a given range even in the event of a sudden increase in the level of activity. With the hybrid method, the response time of the system will sharply decrease (bounded by the minimum collection time) with increasing radiation fields. This attribute is very desirable for health protection monitoring and other applications where it is desirable to minimize transient response time.

4. Floating Mean Algorithm with Hybrid Pulse Counting

4.1 Attributes of Floating Mean with Hybrid Pulse Counting

The floating mean algorithm (2) is preferred for embedded applications because the simplicity of the algorithm minimizes the amount of floating point math involved, and has the best coefficient of variation (variation from the mean value). The number of floating point operations performed

is significant on an embedded controller because of the cumulative effect of round off errors and because of the significant amount of microprocessor time involved. The coefficient of variation for the floating mean is slightly better than that of the weighted mean and significantly better than that of the quasi-exponential algorithm. The coefficient of variation can best be thought of as "needle wiggle" or the amount of variance in the indicating device between successive updates. Use of the floating mean algorithm does sacrifice the transient response time somewhat as compared to the weighted floating mean algorithm (3).

While the transient response time of the floating mean algorithm (2) is not as fast as that of the weighted mean (3), the response time is greatly enhanced through the use of the hybrid pulse counting method. The hybrid pulse counting method ensures the system responds as quickly as a statistically significant number of pulses is available. By bounding the counting method with lower and upper time values, an acceptable system time response is ensured.

4.2 Adaptation of the Floating Mean for Microprocessor Implementation

As described in the section above, the floating mean algorithm can be expressed as follows:

$$(2)\ R_n = 1/(mT) \sum_{i=q}^{n} N_i$$

The algorithm above is based on the assumption of a constant time collection value. In order to use this algorithm in a fixed or hybrid pulse counting method, the function must be modified to the following expression:

$$(2)\ R_n = \sum_{i=q}^{n} N_i \Big/ \sum_{i=q}^{n} T_i$$

This follows from the resultant produced in the first equation (2) from *mT* which produces the total sampling time. Microprocessor implementation of this expression in its current form causes a problem in that the summation of *T* is limited to either a 16 bit unsigned value (65535) or a 32 bit long unsigned value. Since *T* in this implementation is derived from the microprocessor clock, it can be expected to be quite large for lower count rates. Implementation of a 64 bit data type is possible, but would greatly increase the microprocessor cycle time for count rate calculations.

It was highly desirable to reduce the expression to produce manageable lengths of data. Even with moderately slow count rates and few summations (small values of *n*), it is probable that more than 32 bits of storage space could be required for the summation of *T*. For steady state count rates and a sufficient number of pulses acquired to predict the count rate within two standard deviations of the mean, the average of the count rates for individual data sets closely approximates the expected mean value (small variations in T_i between data sets). While this holds true, the (closely approximated) expected mean value can be expressed as follows:

$$(5)\ R_n = 1/n \sum_{i=q}^{n} (N_i/T_i)$$

This expression is not valid for transients. However, the errors introduced in the transient response of this method have been empirically proven to be less than the round off and scaling errors introduced from implementing an accurate expression for the floating mean.

The above assertion is paramount to the design proposed in this paper. As shown in Table 2, if estimated count rates are computed by alternately using the mean + 2 standard deviations and the mean - 2 standard deviations for a 10 set measurement, the resultant estimated count rate approximately equals the mean. The estimated rate produced is not as close as those produced by a true weighted mean (3) or floating mean calculation but is considerably better than that produced by a quasi-exponential estimate (1). The assertion of the above approach is that the resultant count rate produced from averaging computed count rates is a reasonable approximation to the mean during steady state operation (data sets are within 2 standard deviations of the mean).

Mean Count Rate	Min Expected Rate at 2Sigma	Max Expected Rate at 2Sigma	Average of CPM with 2 Sigmas	Weighted Mean with 2 Sigmas
10.0	9.9	10.8	10.3	10.1
20.0	19.7	21.2	20.5	20.1
30.0	29.6	31.4	30.5	30.1
40.0	39.5	41.6	40.6	40.1
50.0	49.4	51.8	50.6	50.2
60.0	59.2	62.0	60.6	60.2
70.0	69.1	72.2	70.6	70.2
80.0	79.0	82.3	80.7	80.2
100.0	98.7	102.6	100.7	100.2
200.0	197.5	203.7	200.6	200.3
300.0	296.2	304.5	300.3	300.4
400.0	395.0	405.2	400.1	400.5
500.0	493.7	505.8	499.7	500.5
600.0	592.4	606.3	599.4	600.6
700.0	691.2	706.8	699.0	700.6
800.0	789.9	807.3	798.6	800.7
900.0	888.7	907.7	898.2	900.7
1000.0	987.4	1008.2	997.8	1000.7

TABLE 2 - COMPARISON OF AVERAGE OF CPM AND WEIGHTED MEAN

As mentioned earlier, the count rate averaging scheme introduces errors during transients. Consider the effect on the calculated count rate when $T_{i+1} >> T_i$ for both the floating mean (2)and average of count rates (5). The floating mean algorithm, for constant pulse or hybrid counting methods can be expressed as follows:

$$(6)\quad R_n = \sum_{i=q}^{n} N_i \Big/ \sum_{i=q}^{n} T_i$$

As T_{i+1} approaches infinity, the resultant rate approaches zero with no restrictive bounds. In fact, the resultant rate can only be bounded if a limit is placed on the maximum collection time allowed. It follows that the floating mean algorithm responds with a minimum number of updates to decreasing count rates (increasing time values).

Next, consider the effect of $T_{i+1} >> T_i$ for the average of count rates method (5). The average of count rates can be expressed as follows:

(5) $R_n = 1/n\sum_{i=q}^{n}(N_n/T_n)$

As T_{i+1} approaches infinity, the resultant rate approaches:

(7) $R_n = 1/n\sum_{i=q}^{n-1}(N_n/T_n)$

Therefore the output is bounded to *n-1* summations divided by *n* in the case where the count rate is decreasing. This tends to push the transient reading higher than the true mean in cases where the count rate is decreasing. This bounding condition is present during the transient until all measurement sets are within 2 standard deviations of the steady state mean.

Next consider the case where T_{i+1} approaches zero (count rate increasing) for the floating mean algorithm. As T_{i+1} approaches zero, the small time value is insignificant compared to the other values. The bounding equation can be expressed as follows:

(8) $R_n = 1/n(\sum_{i=q}^{n}N_i / \sum_{i=q}^{n-1}T_i)$

For the average count rate algorithm (5), with T_i approaching zero (8), the single measurement count rate computed from T_i approaches infinity. As this algorithm is a true average, the resultant average count rate will also approach infinity unbounded.

For relatively slow transients in either the upward or downward direction, the average count rate reasonably approximates the floating mean. In cases where a sharp transient occurs for an increasing count rate, the errors will be significant but only for a short duration of time. As the count rate increases, so does the system update frequency (bounded in the hybrid algorithm by the minimum sample time) so that large upward transient errors tend to diminish very quickly. For quick transients in the downward direction (count rate decreasing) the average count rate algorithm produces an upper bound in which it is known that the true mean is less than the resultant output.

For most applications, the average count rate algorithm produces acceptable results. Usually it is most desirable to know the steady state count rate and to be able to track upwardly occurring transients quickly. In the case of upward transients, the average counting algorithm tends to weight the resultant output in a manner similar to the weighted mean algorithm. For downward transients, the predicted rate of the average counting algorithm tends to perform similar to a traditional analog RC type rate meters.

5. Embedded Controller Requirements

Both the Motorola 68HC11 and Intel 805X families of embedded controllers contain external interrupt lines and internal real time clocks (clock speed based on the microprocessor crystal). The algorithms demonstrated in this paper utilize an external interrupt, the real time clock counter, timer overflow interrupt, and near ANSI compliant “C” language. While special purpose input capture and counters are available on many Intel and Motorola embedded controllers, it was decided to approach the design as generically as possible in order to facilitate porting to other embedded controller families. This paper does not address pulse discrimination algorithms and assumes any hardware pulse shaping has occurred prior to introduction to the microprocessor. Dependent upon software requirements not addressed in this paper, it should be expected that a minimum of 8K code storage space will be required and at least 256 bytes of RAM. It should also be noted that the proposed algorithms are dependent upon the microprocessor free-running timer speed. Adjustments to the algorithms will be required if the

specific embedded controller timer operates at a speed different from those listed. Finally, it should be noted that data compression is employed on resultant values stored in volatile memory. If sufficient data space is available and the "C" compiler being used supports long (32 bit integer) data types, the data compression may be eliminated.

6. Software Implementation

6.1 Pulse Counting and Timer Functions

For the Motorola 68HC11 based embedded controllers, the free running 16 bit timer was utilized to acquire time values. There is no provision for clearing the free running timer, therefore it was necessary to retain previous values and calculate the elapsed time. The Intel 8051 series timers are cleared at startup and therefore elapsed time calculations are not required. For portability between the two platforms, the only code discussed will utilize 16 bit elapsed time calculations. Dependent upon the count rate range of the application and desired accuracy, it may be necessary to capture timer overflows and append them to the elapsed time value acquired (effectively creating a 32 bit time counter). The interrupt code segment written to capture timer overflow values for the Motorola can be expressed below:

```
#pragma interrupt_handler TOFIntr
void TOFIntr(void)
{
        TFLG2 = 0x80;                   // reset TOF interrupt flag
        tof_counts++;                   // increment TOF counts
}
```

For the Intel 805X series embedded controller, the code segment would be as follows:

```
#pragma interrupt_handler TOFIntr
void TOFIntr(void)
{
        tof_counts++;                   // increment TOF counts
        TCON = TCON | 0x20;             // 16 bit timer 0 does not automatically restart, must
                                        // clear and restart timer
}
```

The Motorola timer always runs and automatically restarts after overflow occurs. However, it is necessary to clear the interrupt flag as this is a software function. The Intel timer does not automatically restart after overflow occurs. However, it is not necessary to clear the interrupt flag as the hardware resets the interrupt register upon vectoring to the interrupt routine. For the Intel, Timer 0 is used in this application. Dependent upon the specific 805X microcontroller used, there may be as many as two other timers available that could be substituted for Timer 0. Note the software may vary slightly dependent upon the specific compiler used. The ImageCraft HC11 "C" compiler for the Motorola 68HC11 was used in this design.

Pulse counting on the Motorola is accomplished through the Port A input capture 2 function. There are at least two other interrupt capture pins (0 & 1) and two external interrupt lines that may be substituted for the one chosen in this application. The pulse capture function can be expressed as follows:

```
#pragma interrupt_handler PulseIntr     // digital input interrupt handler (counts incoming
void PulseIntr(void)                    // rad pulses)
{
```

```
        TFLG1 = 0x02;                   //clear interrupt flag
        p_counts++;                             // increment pulses received counter
}
```

Using the Intel 805X series embedded controller, the code segment would be as follows:

```
#pragma interrupt_handler PulseIntr     // digital input interrupt handler (counts incoming
void PulseIntr(void)                    // rad pulses)
{
        EXF2 = 0x40;                    //clear interrupt flag, not reset by hardware
        p_counts++;                             // increment pulses received counter
}
```

Note that the Timer 2 is only present on the 8052 and higher derivative Intel controllers. These software segments do not address initialization routines. See the attached source code listing for complete setup and initialization routines (Motorola only).

The timer and pulse capture interrupts run until the desired number of pulses (or time constraints) have been satisfied. Once the constraints have been satisfied, the time, timer overflow, and pulse counts can be saved to respective buffers. It is then possible to set a flag to alert the main routine of available data. With the Motorola, this can be shown as follows:

```
#pragma interrupt_handler PulseIntr             // digital input interrupt handler (counts
void PulseIntr(void)                    // incoming pulses)
{
        TFLG1 = 0x02;                   //clear interrupt flag
        p_counts++;                             // increment pulses received
        if((p_counts  > mincounts) && mintime)  // if minimum # of pulses received and
                {                               // and minimum sample collection time elapsed
                buf_num++;                      // increment data buffer pointer
                buf_num = buf_num % 10;         // max buffer length of length 10
                INTR_OFF();                             // turn off interrupts
                tof_vals[buf_num] = tof_counts;  // store timer overflow values
                time_vals[buf_num] = TCNT;              // store timer values
                count_numbs[buf_num] = p_counts;        // save pulse counts
                tof_counts = 0;                 // clear counters and start next counting
                p_counts = mintime =0;          // sequence
                cnt_flag = 1;                           // let main know data ready to
                INTR_ON();                              // process, restart system interrupts
                }
}

#pragma interrupt_handler TOFIntr               // interrupt handler responsible for counting
TOFs
void TOFIntr(void)
{
        TFLG2 = 0x80;                           // reset interrupt flag
        tof_counts++;                           // increment TOF counts
        if(tof_counts > 0x7a)           //minimum update every 4 seconds, get whatever
                {                               // data is available if # of pulses not received in 4 sec
                INTR_OFF();             // Also good place to implement low count rate alarm
                buf_num++;                      // save counts, ticks, and TOFS to buffers
                buf_num = buf_num % 10;
```

```
                tof_vals[buf_num]  = tof_counts;
                time_vals[buf_num] = TCNT;
                count_numbs[buf_num] = p_counts;
                p_counts = tof_counts = 0;                      // reset counters
                cnt_flag = 1;                   // flag main program that data needs processing
                INTR_ON();                      // turn on system interrupts
                }
        if(tof_counts > 0x16)       // Don't want to update any faster than ¾ seconds, acquire
                        {           // extra pulses if necessary
                        INTR_OFF();
                        mintime = 0xff;             // set min time expired flag for pulse intr
                        INTR_ON();
                }
        else                    // else minimum time not satisfied, set flag to zero
                mintime = 0;
}
```

By substituting in the Intel specific interrupt code segments listed earlier, the software segment above is also valid for the Intel 805X series embedded controllers. The program segment above ensures a minimum sampling time of ¾ seconds and a maximum sampling time of 4 seconds. The minimum time requirement ensures the output device update rate is sufficiently slow to be useable while the maximum sampling rate ensures the output device is updated at least every 4 seconds. Optimum count rate prediction is achieved when the number of desired pulses is counted within the time constraints.

In the main software module, the *cnt_flag* variable is continuously checked to see if data is available. Since the data is stored in ring buffers, data will not be lost during the next pulse / time acquisition cycle. The main software function can be expressed as follows:

```
int cnt_flag

main()
{
while(1)                        // do forever
 {
  if(cnt_flag)                   // new pulse / time information ready to process
     {
     cnt_flag = 0;              // reset data ready flag
     if(buf_num)                // if current buffer greater than zero
          {                     // get elapsed time from timer (ticks
          if(time_vals[buf_num] > time_vals[buf_num - 1])     //if new greater than old
              cpm_low[buf_num] = time_vals[buf_num] - time_vals[buf_num - 1];          // save
          else                          // new ticks less than old ticks, have to  "borrow" from TOF
             {
              cpm_low[buf_num] = (0xffff - time_vals[buf_num-1]) +
                    time_vals[buf_num];                         // get elapsed ticks
              if(tof_vals[buf_num])
                    --tof_vals[buf_num];                        // "borrow" one from TOF
              }
          }
    else                // current buffer equals zero, special case
          {
          if(time_vals[0] > time_vals[11])                      // same as above except must get
```

```
            cpm_low[buf_num] = time_vals[0] - time_vals[11];      // previous time from buf #10
        else
            {
            cpm_low[buf_num] = (0xffff - time_vals[11]) + time_vals[0];
            if(tof_vals[0])
                --tof_vals[0];
             }                                          // end time_vals if
        }                                               // end buf_num else
    cpm_high[buf_num] = tof_vals[buf_num];              // get timer overflow counts
    advalue = compute_display();                        // get D/A output value
    output_display(advalue);                            // send value to D/A
    }                                                   // end cnd_flg if

  }                                                     // end while(1)

}                                                       // end main
```

As mentioned earlier, the timer value obtained from the Motorola 68HC11 16 bit free running timer must be converted from an absolute time to a relative time (elapsed time between measurement sets). This is accomplished by taking the difference between the most recent timer value and the previous timer value. In certain instances, the new timer value will be smaller than the old value due to overflows. In these instances, the timer and TOF can be viewed as a combined 32 bit time value (TOF representing the upper 16 bits). Therefore, “borrowing” hex value 0x10000 (code actually “borrows” 0xffff) for the timer from the corresponding TOF count reduces the TOF count by 1 (remember to think of the TOF as the upper 16 bits of a 32 bit timer). In addition to timer overflow considerations, the programmer must also consider the case where the previous data is stored in the last buffer location and the new data is stored in the first buffer location. In this instance, it is required to calculate elapsed time using the highest buffer location instead of the buffer location below the current one. If these routines are implemented with the Intel family, the elapsed time computations need not be performed if the programmer restarts (and clears) the timer after each measurement acquisition.

The pulse counting and timer functions discussed are valid for any of the mathematical algorithms discussed in this paper. In fact, with minor modifications to the code segments above, the routines could be used for constant time or constant pulse counting. It is important to remember that the timer values recorded are tied to the microprocessor cycle time and not to a unit of time. In order to convert the values to time units, it is necessary to use the appropriate conversion factor. For example on the Motorola 68HC11A8 with a 2.0 MHz clock (8 MHz crystal) and divide by 1 prescaler, each timer tick represents 500 nanoseconds. On an Intel 8052 with a 16 MHz crystal (1.33 MHz clock), each timer tick represents 751 nanoseconds.

6.2 Modified Floating Mean Algorithm with Hybrid Pulse Counting

The modified floating mean algorithm (average of count rates) (5) was chosen as the preferred counting algorithm. The average of count rates directly lends itself to 8 bit microprocessor implementation due to the relatively small size of the numbers and the number of calculations involved. As the 32 bit timer cannot be readily held in a 16 bit integer and it was not desirous to implement 32 bit long integers (not supported by the compiler used), the timer values are scaled to 16 bit integers.

The timer scaling is based on the maximum timer value that can be expected in a given operating range. For example if the range being considered is 100 to 10,000 counts per minute, the largest timer value expected is at 100 counts per minute. With a fixed pulse count rate for this range of 60 counts, the largest timer value to be expected is 36 seconds. With a 500 ns microprocessor timer speed (cycle time), the value stored in the timer overflow will be hex 44A while the timer will contain hex A64A. In order to scale the timer value to a 16 bit integer, it is necessary to retain the 16 most significant bits (11 from the timer overflow counter and 5 bits for the timer). With the timer value scaled, it is then necessary to either scale the timer conversion factor (to convert from clock ticks to minutes), to scale the counts, or to scale the output conversion factor.

In order to limit the number of mathematical operations that must be performed, the timer scaling factor is combined with the output conversion factor. This particular design operated a 14 bit D to A converter. Therefore, at the highest count rate for a given operating range, the resultant output should be hex 3fff (decimal 16383). The final output can be expressed as follows for a count rate within the given operating range:

D to A output = (measured count / measured time) (maximum time/ maximum counts) (D to A maximum value)

The first two numbers generated (measured count rate and inverse maximum count rate) produce a percentage of maximum output. When this percentage is multiplied by the maximum D to A output value, the actual D to A output is obtained. Since each measured output is one element in the summation of the count average algorithm the true expression becomes:

D to A output = $(1/n\sum(\text{measured count / measured time}))$ (maximum time/ maximum counts) (D to A maximum value)

where n represents the number of measurement sets in the summation. This can further be reduced to:

D to A output = $\sum(1/n(\text{measured count / measured time})$ (maximum time/ maximum counts) (D to A maximum value))

where all floating point operations are performed within the summation. In fact, only the measured time and counts are variables within an operating range. The maximum values can be expressed as a constant. Doing so limits the floating point operations to just two per update cycle. For a 100 to 10,000 count rate scale, the maximum time value would be hex AFC86, the maximum counts would be 60, the maximum D to A value would be hex 3fff. The timer conversion factor (to convert from clock ticks to minutes) must be applied to both the maximum time and measured time. Since this value is a constant, it drops from the equations (the conversion appears in both the numerator of the maximum count rate and the denominator of the measured count rate). The only other item to consider is the timer scaling.

To offset the scaling of the measured timer value, it is possible to either shift the measured counts or the maximum timer value. As the maximum timer value is a constant, it is the logical item to shift. For the 100 to 10,000 count rate example, the 32 bit measured timer value was shifted 11 bits to the right. Therefore, the maximum timer value must be shifted in the same manner to maintain the proper ratios. The maximum timer value for 10,000 counts is hex AFC86. Shifting this value to the right eleven places produces the 16 bit shifted maximum timer value of hex 15f (351 decimal). Combining this with the other constants produces hex value 17660. With the case that *n* is equal to 10 (10 summations of measured sets), the constant value can be divided by 10 to produce the final conversion factor of 2570 or 9584 decimal.

The code sample that implements this for the 100 to 10,000 count rate range is shown below:

```
case 10:            // 10K count rate maximum
        // push timer overflow to top of sixteen bit integer and add in upper 5 bits of timer
        cpm_high[buf_num] = cpm_high[buf_num] <<5;      // timer overflow
        cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>11);// timer
scaling
        tlow = cpm_high[buf_num];          // save scaled timer to a float
        tlow = (float)( 9584/tlow); // constant conversion factor - specific to scale
        tlow = tlow*count_numbs[buf_num];          // multiply by measured counts
        cpm_high[buf_num] = tlow;          // Save results as integer value
        outval2 = 0;                    // sum up previous count rates
        for(i = 0; i < 10; i++)
                outval2 = cpm_high[i] + outval2;
        break;              // outval2 contains the d to a output value
```

In the above software segment, *cpm_high* initially holds the timer overflow value and at the end of the segment contains the scaled count rate for each measurement set; *cpm_low* contains the elapsed timer information; *tlow* is a floating point variable used to store floating numbers during the mathematical operations; *count_numbs* contains the pulse counts for each measurement set, and *outval2* contains the resultant output value written to the D to A converter. The timer overflow value is pushed 5 places to the left. This places the least 11 significant timer overflow bits as the most significant digits in the scaled (16 bit) timer value. Next the lower 5 bits of the scaled timer value are filled in with the upper 5 bits of the elapsed timer value. The resultant number is converted to a floating point and divided into the constant conversion factor. It is important to perform the division before the multiplication (by the number of counts) so as not to introduce overflow. The next set multiplies the measured counts by the result obtained from the division. This number represents 1/10 of the D to A output value. When this number is summed with the 9 previously converted measurements, the actual output value to the D to A converter is produced.

For compilers that implement 32 bit long data types, scaling of the timer values will most likely not need to be performed. It is still recommended that the number of floating point operations be minimized to reduce round-off errors. A complete listing of the source code for the Motorola 68HC11 embedded controller is included in Appendix 1.

The software was implemented for an alpha rate counter with 1K, 3K, 10K, 30K, and 100K scales. The system was tested against ANSI standard N42.17B-1989 sections 5.2, 5.3, and 5.4. The software implementation strictly addressed count rates independent of background and alarm conditions. The background and alarm software were outside of the scope of this design although they will be implemented in the final product. Of the tests performed: stability, response time, and coefficient of variation, the software algorithm passed all ANSI acceptance criteria with no deviations noted.

The hybrid pulse - average count rate algorithm presented in this paper produces very accurate and stable readings between 10% and 100% scale for most applications. The response time of the system can easily be tuned to optimize the tradeoff between stability, accuracy, and time response. Unlike time based systems, the hybrid pulse - average count rate algorithm varies the speed of the counting system as needed to acquire statistically accurate counting information. Additionally, the algorithm responds well to rapid increases in pulse counts. Because of the simplicity of the algorithm, it was possible to optimize the design for an 8 bit microcontroller while not substantially sacrificing counting accuracy or performance. Due to all of its strengths, the hybrid-pulse average count rate algorithm is a strong contender for other count rate applications utilizing microcontroller based designs.

APPENDIX I

Source Code for Implementation of Hybrid-Pulse Average Count Rate Algorithm on a Motorola 68HC11 Platform with ImageCraft HC11 “C” Compiler

```
/*****************************************************************************
* Sample Implementation of hybrid-pulse average count rate algorithm for the Motorola 68HC11   *
* embedded controller.  Code drives an analog display meter through a D/A converter (maximum value  *
* hex 3fff).  Code Developed on a New Micros                                   *
* NMIX-0020 embedded controller with opto-isolated input, opto-isolated output                *
* Software used - ImageCraft HC11 "C" compiler, version 3                     *
*               Russell Kevin Huffman, Senior Engineer                        *
*               Westinghouse Savannah River Company                           *
*                       4/1/98                                                *
*****************************************************************************/

#include <hc11.h>      // 68HC11 specific stuff
#include <float.h>     // floating point routines
#include "digio.h"                  // digital input and output stuff


unsigned int p_counts, time_vals[20], low_count, tof_vals[20];      // global counter values & flags
unsigned int cpm_low[20], cpm_high[20];                             // global TOF & tick buffers
unsigned int test_high[20], test_high2[20], test_high3[20];                 // test buffers for debugging
extern unsigned int tof_counts, buf_num, intr_num;                  //more global counters

int compute_display(void);                          // compute value to drive D/A
void output_display(int advalue);           // output computed value to D/A
int get_range(void);                                // determine position of range selector

char cnt_flag;

unsigned int count_numbs[20];

unsigned int hflag;                 //maximum counting time allowed
/*****************************************************************************/
/*                      Main Routine for counting algorithm                  */
/*                      Kevin Huffman 3/98                                   */
/*                      Westinghouse Savannah River Company                  */
/*****************************************************************************/



main()
{
int i;
int advalue;
do_ints();   /* Setup interrupts */
   init_tof();                      // enable timer overflow interrupt (blink interrupt) used for TOF counting
            hflag = 19;             // set minimum time and count durations for initial count
            INTR_OFF();             // turn off system interrupts until initialization is complete

            for(i = 0; i < 20; i++)                 // initialize buffers for ticks, TOFs, and counts
                        {
                        time_vals[i] = tof_vals[i] = count_numbs[i] = 0;
                        }
            p_counts = low_count = cnt_flag = buf_num = 0;          // initialize flags
 INTR_ON();                         // turn on system interrupts

 while(1)                           // do forever
            {
            if(cnt_flag)            // time to update display, new buffer of information ready to process
                        {
                        cnt_flag = 0;                   // reset buffer flag
                        test_high[buf_num] = tof_vals[buf_num];             // get TOF values
                        if(buf_num)                                         // if current buffer greater than zero
```

```
                                {                                       // get elapsed time from timer (ticks), do if
                                if(time_vals[buf_num] > time_vals[buf_num - 1])     //new ticks greater than old
                                        cpm_low[buf_num] = time_vals[buf_num] - time_vals[buf_num - 1];
                                else            // new ticks less than old ticks, have to "borrow" one from TOF
                                        {
                                        cpm_low[buf_num] = (0xffff - time_vals[buf_num-1]) +
                                                time_vals[buf_num]; // get elapsed ticks
                                        if(tof_vals[buf_num])
                                                --tof_vals[buf_num];                    // "borrow" one from TOF
                                        }
                                }
                        else                    // current buffer equals zero, special case
                                {               // same as above except must get previous time from buf #9
                                if(time_vals[0] > time_vals[11])
                                        cpm_low[buf_num] = time_vals[0] - time_vals[11];
                                else
                                        {
                                        cpm_low[buf_num] = (0xffff - time_vals[11]) + time_vals[0];
                                        if(tof_vals[0])
                                                --tof_vals[0];
                                        }               // end time_vals if
                                }               // end buf_num else
                        cpm_high[buf_num] = tof_vals[buf_num];
                        advalue = compute_display();                    // get D/A output value
                        output_display(advalue);                                        // send value to D/A
                        }               // end cnd_flg if

                }                       // end while(1)


}                       // end main


/*********************************************************************/
/*                              main interrupt setup function                   */
/*                              Kevin Huffman                                   */
/*                              11/12/97                                        */
/*                              Westinghouse Savannah River Company             */
/*                              This function calls most of the interrupt       */
/*                              Setup Functions                                 */
/*********************************************************************/
void do_ints(void)
{
 INTR_OFF();                    // turn off system interrupts
 InitDigInt ();                 /* Turn on digital input capture interrupt */
 INTR_ON();                     // turn on system interrupts

}

/*****************************************************************************/
/*                                      Compute_Display function                        */
/*                                      Kevin Huffman                                   */
/*                                      3/98                                            */
/*                                      Equipment Engineering Section - WSRC            */
/*                      This function takes the raw time and counts produced from       */
/*                              interrupts and converts it to the D/A output value for display  */
/*                                      updates.  The output is an average of 10 points         */
/*                                 All scales are 10 point averages                     */
/*****************************************************************************/

int compute_display(void)
{
unsigned int range, i, outval, outval2, h_val, l_val;
float tlow;
int ohvalue, l_count, h_count;

        range = get_range();            // get range selector switch position
        switch(range)
                {
                case 100:       // on 100k scale, throw away 8 most significant bits of TOF
```

```
                cpm_high[buf_num] = cpm_high[buf_num] <<8;           //throw away bottom 8 bits of
                cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>8);          // tick
                test_high2[buf_num] = cpm_high[buf_num]; // debug only
                tlow = cpm_high[buf_num];                       // convert time to float
                tlow = (float)(468/tlow);           // get ratio of measured CPM to max CPM for range
                tlow = tlow*count_numbs[buf_num];
                 cpm_high[buf_num] = (float)(tlow*10.0);       // scale ratio by 10 and save to integer
                outval2 = 0;
                for(i = 0; i < 10; i++)             // sum 10 most recent ratios
                        {
                        test_high3[i] = cpm_high[i];
                        outval2 = cpm_high[i] + outval2;
                        }
                tlow = outval2;                     // convert to a float
                tlow = tlow*1.638;                  // multiply ratio by max D/A value
                outval = tlow;                      // save D/A value as int
                break;

        case 30:
        // same as 100k scale, except 30k scale
                cpm_high[buf_num]= cpm_high[buf_num]<<7;  // shifts are different for TOF and ticks
                cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>9);
                test_high2[buf_num] = cpm_high[buf_num];
                tlow = cpm_high[buf_num];
                tlow = (float)(780/tlow);
                tlow = (float)(tlow*count_numbs[buf_num]);
                 cpm_high[buf_num] = (float)(tlow*10.0);
                outval2 = 0;
                for(i = 0; i < 10; i++)
                        {
                        test_high3[i] = cpm_high[i];
                        outval2 = cpm_high[i] + outval2;
                        }
                tlow = outval2;
                tlow = tlow*1.638;
                outval = tlow;
        break;

        case 10:
// same as 100K scale, except 10K scale
                cpm_high[buf_num] = cpm_high[buf_num] <<5;   // shifts are different for TOF & ticks
                cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>11);
                test_high2[buf_num] = cpm_high[buf_num];
                tlow = cpm_high[buf_num];
                tlow = (float)(60/tlow);
                tlow = tlow*count_numbs[buf_num];
                cpm_high[buf_num] = (float)(tlow*100.0);
                outval2 = 0;
                        for(i = 0; i < 10; i++)
                        {
                        test_high3[i] = cpm_high[i];
                        outval2 = cpm_high[i] + outval2;
                        }
                tlow = outval2;
                tlow = tlow*1.638;
                outval = tlow;
        break;
```

```
                        case 3:     // same as 100K scale, except 3K scale

                                    cpm_high[buf_num] = cpm_high[buf_num] <<5;
                                    cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>11);
                                    test_high2[buf_num] = cpm_high[buf_num];
                                    tlow = cpm_high[buf_num];
                                    tlow = (float)(200/tlow);
                                    tlow = tlow*count_numbs[buf_num];
                                     cpm_high[buf_num] = (float)(tlow*100.0);
                                    outval2 = 0;
                                    for(i = 0; i < 10; i++)
                                                {
                                                test_high3[i] = cpm_high[i];
                                                outval2 = cpm_high[i] + outval2;
                                                }
                                    tlow = outval2;
                                    tlow = tlow*1.638;
                                    outval = tlow;
                        break;


                        case 1:     // Same as 100K scale with different scaling factors
                                    cpm_high[buf_num] = cpm_high[buf_num] <<4;
                                    cpm_high[buf_num] = cpm_high[buf_num] + (cpm_low[buf_num]>>12);
                                    test_high2[buf_num] = cpm_high[buf_num];
                                    tlow = (float)cpm_high[buf_num];
                                     tlow = (292/tlow);
                                     tlow = tlow*count_numbs[buf_num];
                                     tlow = tlow*100;
                                     cpm_high[buf_num] = (int)tlow;
                                    outval2 = 0;
                                    h_val = 0;                          // initialize high and low values
                                    l_val = 0xffff;
                                    for(i = 0; i < 10; i++)
                                                outval2 = cpm_high[i] + outval2;
                                    tlow = outval2;
                                    tlow = tlow*1.638;
                                    outval = tlow;
                        break;
                        }
if(outval > 0x3fff)  // if value greater than max D/A, set to max val
                outval = 0x3fff;
return(outval);
}


/*****************************************************************************/
/*                      Output_Display function                              */
/*                      Kevin Huffman                                        */
/*                      3/98                                                 */
/*                      Equipment Engineering Section - WSRC                 */
/*                      This function outputs the D/A value to the D/A circuit.      */
/*****************************************************************************/


void output_display(int advalue)
{
static int ohvale;
int totval;

                        OTBD1 = (advalue&0xff);                 // write lower 8 bits to output board
                        OTBD2 = (advalue>>8)&0x3f;    // write upper 6 bits to output board

}


/*****************************************************************************/
/*                                      Get_range function                   */
/*                                      Kevin Huffman                        */
/*                                      3/98                                 */
```

```
/*                                      Equipment Engineering Section - WSRC            */
/*                                      This function gets the range switch position setting    */
/*****************************************************************************/

int get_range(void)
{
        char range;

        range = INBD1;                  // get switch value from input board
        switch(range)
                {
                case 0x0e:              // set minimum count and time values for interrupts based on range
                        hflag = 19;
                        return(1);                      // return range value
                case 0x0d:
                        hflag = 19;
                        return(3);
                case 0x0c:
                        hflag = 59;
                        return(10);
                case 0x0b:
                        hflag = 59;
                        return(30);
                case 0x0a:
                        hflag = 99;
                        return(100);
                default:
                        hflag = 99;
                        return(100);
                }
}
```

```
/*  This module processes all of the interrupt based functions (timer overflow and pulse capture)                  *
*           & digital input & output routines                                                                       *
*           Hardware:  NMIX-0020 embedded board (New Micros), opto-input card, opto-output card     *
*           Kevin Huffman - Westinghouse Savannah River Company                                                     *
*           3/1998                                                                                                  *
*******************************************************************************************/

#include <hc11.h>      // 68HC11 specific stuff, port addresses, registers, etc
#include "digio.h"     // digital I/O stuff


unsigned int tof_counts, buf_num, buf_num;   //global counters

extern unsigned int p_counts, time_vals[20], low_count, tof_vals[20];          // global counters & buffers
extern unsigned int cpm_low[20], cpm_high[20];
extern unsigned int count_numbs[20];
extern char cnt_flag;
extern unsigned int hflag, min_num;
unsigned int mintime, max_time;

/*  This routine is the interrupt driver that captures incoming pulses RKH 1998 */
#pragma interrupt_handler DigInIntr                    // digital input interrupt handler (counts incoming pulses)
void DigInIntr(void)
{

        TFLG1 = 0x06;                                  //clear interrupt flag
        p_counts++;                                    // increment pulses received
        if(p_counts  > hflag)    // if minimum # of pulses received
                {
                if(mintime > 1)                        // if minimum counting time filled
                {
                buf_num++;                             // increment buffer
                buf_num = buf_num % 12;        // circular of length 12
                INTR_OFF();                            // turn off interrupts
                tof_vals[buf_num] = tof_counts;   // get timer overflow value
                time_vals[buf_num] = TCNT;             // timer counter register          (ticks)
                count_numbs[buf_num] = p_counts;       // save pulse counts
                tof_counts = 0;                        // clear counters and start next counting sequence
                p_counts = mintime = max_time =0;
                cnt_flag = 1;                          // let main know info ready to process
                INTR_ON();
                }
        }
}

/* Interrupt handler for timer overflow counter RKH 1998 */
#pragma interrupt_handler TOFIntr                      // interrupt handler responsible for counting TOFs
void TOFIntr(void)
{
        TFLG2 = 0x80;                                  // reset interrupt flag
        tof_counts++;                                  // increment TOF counts
        if(tof_counts > 0x7a)          // never run longer than 32.77ms X hex 7a seconds (I think the value is 4 seconds)
                {
                max_time = 0xff;               // set max time flag (I don't think this is used)
                        INTR_OFF();
                buf_num++;                             // save counts, ticks, and TOFS to buffers
                buf_num = buf_num % 12;
                tof_vals[buf_num]  = tof_counts;
                time_vals[buf_num] = TCNT;
                count_numbs[buf_num] = p_counts;
                p_counts = tof_counts = 0;             // reset counters
                cnt_flag = 1;                          // flag main program that data needs processing
                INTR_ON();
                }
        if(tof_counts > 0x16)                          // if minimum time has expired (I believe 3/4 seconds)
                        {
                        INTR_OFF();
                        mintime = 0xff;                        // set min time expired flag
                        INTR_ON();
                }
        else
```

```
                        mintime = 0;
}



void enable_dig_int(void)           // enables digital input interrupt, received through Port A capture interrupt
{
            TCTL2 = TCTL2 | 0x28;           // Activate PA bit 1, 2 for input captures
            TMSK1 = TMSK1 | 0x06;           // activate PA bit 1,2 for input captures
}

void disable_dig_int(void)          // disables digital input interrupt
{
            TCTL2 = TCTL2 & ~0x14;          // turn off digital interrupts
            TMSK1 = TMSK1 & ~0x06;
}



void InitDigInt (void)              // setup routine for TOF interrupt and digital input interrupt
{

            INTR_OFF ();
/*  Port A bit 1 Input Capture (IC2) used for incoming pulse captures*/
  *(unsigned char *)0x00e5 = 0x7e;  /* ASM Jump cmd  */
  *(unsigned char *)0x00e6 = (((int)DigInIntr & 0xFF00)>>8);
  *(unsigned char *)0x00e7 = ((int)DigInIntr & 0x00FF);
/* TOF used for capture upper 16 bits of timer indicators */
            *(unsigned char *)0x00d0 = 0x7e;
            *(unsigned char *)0x00d1 = (((int)TOFIntr & 0xFF00)>>8);
            *(unsigned char *)0x00d2 = ((int) TOFIntr & 0x00ff);

            enable_dig_int();       // turns on interrupt enable bits
            INTR_ON ();
}

void init_tof(void)     // turns on TOF interrupt
{
            INTR_OFF();
            TMSK2 = TMSK2 | 0x80;           // turn on TOF enable interrupts
            TFLG2 = 0x80;                   // clear any pending interrupts
            INTR_ON();
}
```

```
/*  Digital Input / Output Header File */

#ifndef __DIG_H
#define __DIG_H


#define INBD2 *(unsigned char *)0x8020     // reversed order of boards
#define INBD1 *(unsigned char *)0x8021     // (bd2 & 1 on i/o to
#define OTBD2 *(unsigned char *)0x8010     // match swapped i/o cables
#define OTBD1 *(unsigned char *)0x8011     // rkh 10/20/97

void enable_dig_int(void);                 // turn on digital interrupts
void disable_dig_int(void);                // turn off digital interrupts
void InitDigInt (void);            // set interrupt vectors
void DigInIntr(void);              // digital interrupt handler
void TOFIntr(void);                        // blinking indicator interrupt handler
void dig_out(unsigned int bit, unsigned char val);                 // output digital value
int dig_in(int board);                                             // digital input handler (alarm / process status panel)
#endif
```

References

[1] G.F. Knoll, Radiation Detection and Measurement, Second Edition (Wiley, New York, 1989).

[2] G. White, Nuclear Instruments and Methods 125 (1975) p. 313