Timer Performance

Early measurements[1] of WaitUs in release 1 showed a fixed overhead of a little over 10 μS and a bug where very short values would cause very long wait periods. The second fault was due to the request wait being so short that it had passed before the timer match register could be properly set. This meant that the match would not happen until the timer had overflowed and reached the match point again (some 420+ seconds later). The 10+ μS overhead was simply the accumulated setup and housekeeping associated with the WaitUs function.

These issues were the main reason for developing release 2.

Both of these problems can be dealt with by measuring key areas of the performance of WaitUs during startup.

The most serious problem is the failure for short time waits. It also ends up being the more difficult of the two to fix. The basic approach used was to duplicate the operations of the short branch of code that sets up the timer match registers and does the first test for a match. By timing this short section of code we can determine a lower bound of time the routine can wait for without running into a problem missing matches. A test is then added to WaitUs so that for any waits that are lower than this bound the routine simply exits rather than try to match. Measurements[2] taken while developing this protection indicate that this lower bound is approximately 1.3 μS.

The fixed overhead for a WaitUs is easier to compensate for, the startup simply measures the time it takes to make a call. It then subtracts off the requested wait to get the overhead. On subsequent calls WaitUs simply subtracts this measured overhead.

Measurement of this approach showed promise, but at about the 26 μS mark a discrete jump in the error was seen. Further testing showed a series of jumps in the error as the requested wait period was increased. This is shown in Figure 1. Note that the time scale is logarithmic. Further investigation showed that this error was growing larger because the scaling operation was taking longer with larger values, specifically the division in the scaling operation was taking longer. Since this is a division by a constant it is possible to replace it with a series of multiplies, shifts and adds. The result of doing this is also shown in Figure 1. The multiply, shift and add routine is as fast as the divide at low values and more importantly takes the same amount of time regardless of the value used. The downside is that it takes more room to add this.

---

[1] Thanks to Curt Powell for his measurements and bring this to my attention.

[2] Unless otherwise indicated, measurements mentioned in this note are taken with a 10 MHz oscillator PLLed to 60 MHz with MAM fully on and flash access set to 3 cycles.

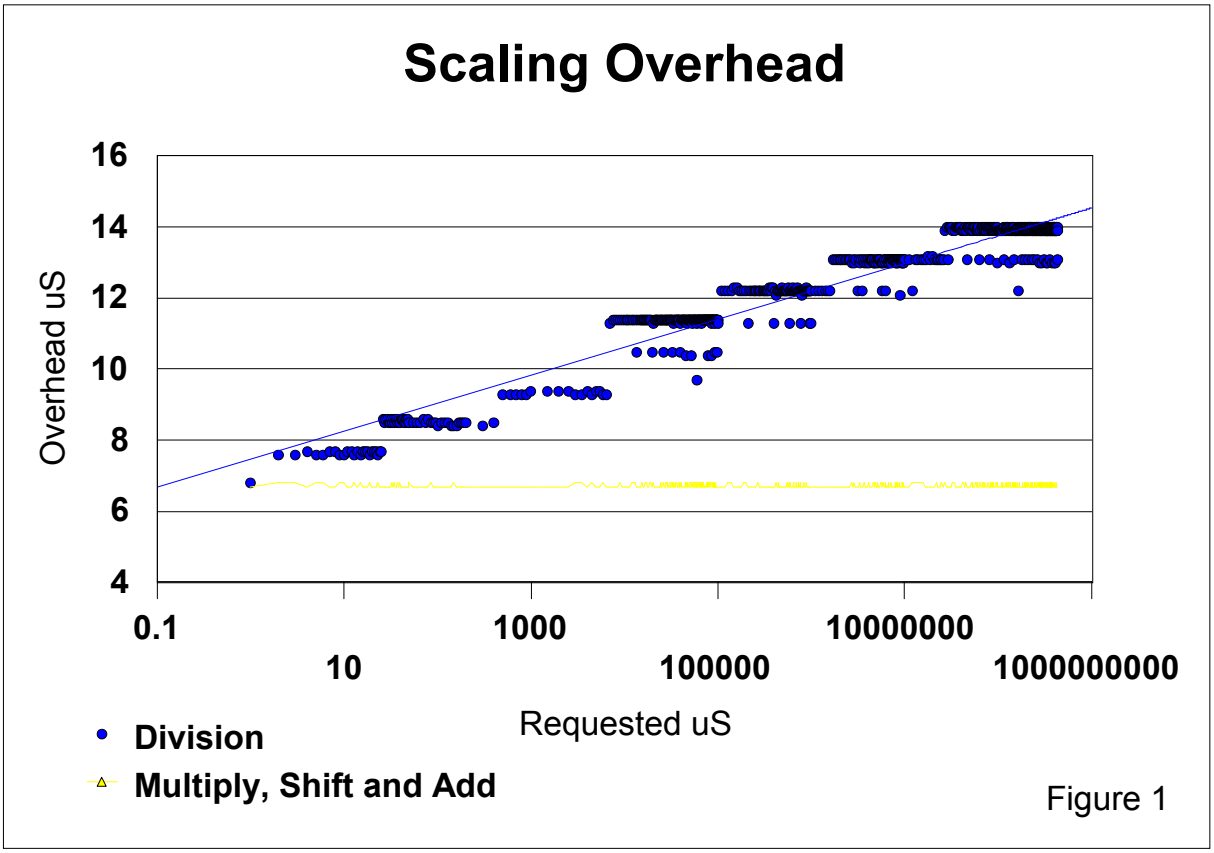

Figure 1

Two methods were used to verify the performance of the timer. The first was to set an I/O pin before entry to WaitUs and clear the pin after exit. The toggle was then observed and measured on an oscilloscope. This works for smaller values of wait and for a small number of samples, but becomes cumbersome for large waits and larger numbers of samples. Because of this a software test was developed to measure elapsed wait time and the pin toggling technique was used to establish confidence in the SW technique. Measurements of the elapsed time using both techniques for the first 31 possible values is shown in Table 1 and Figure 2.

Table 1[3]

Requested Wait	Actual Wait SW Measured	Actual Wait Pin Toggle Measure
0	1.3	1.67
1	9.2	9.62
2	9.3	9.61
3	9.2	9.62
4	9.2	9.62
5	9.2	9.62
6	9.3	9.61
7	9.2	9.62
8	9.2	9.62

[3] Note: Pin toggle measurements were taken on different test runs than the SW measurements.

9	9.3	9.63
10	9.2	9.61
11	9.2	9.62
12	11.9	12.36
13	12.9	13.35
14	13.9	14.35
15	14.9	15.35
16	15.9	16.35
17	16.9	17.35
18	17.9	18.35
19	18.9	19.35
20	20.2	20.35
21	20.9	21.35
22	22.2	22.35
23	22.9	23.35
24	24.2	24.35
25	25.2	25.35
26	25.9	26.35
27	27.2	27.35
28	28.2	28.35
29	28.9	29.4
30	30.2	30.35

Several features are readily apparent from this data. First, the measurement using the pin toggle is clearly longer. This difference appears to be due to the extra instruction and I/O overhead since

the toggle wraps around the timing code that measures the time internally. Taking this into account the correspondence between the two techniques appears to be close enough to be trusted.

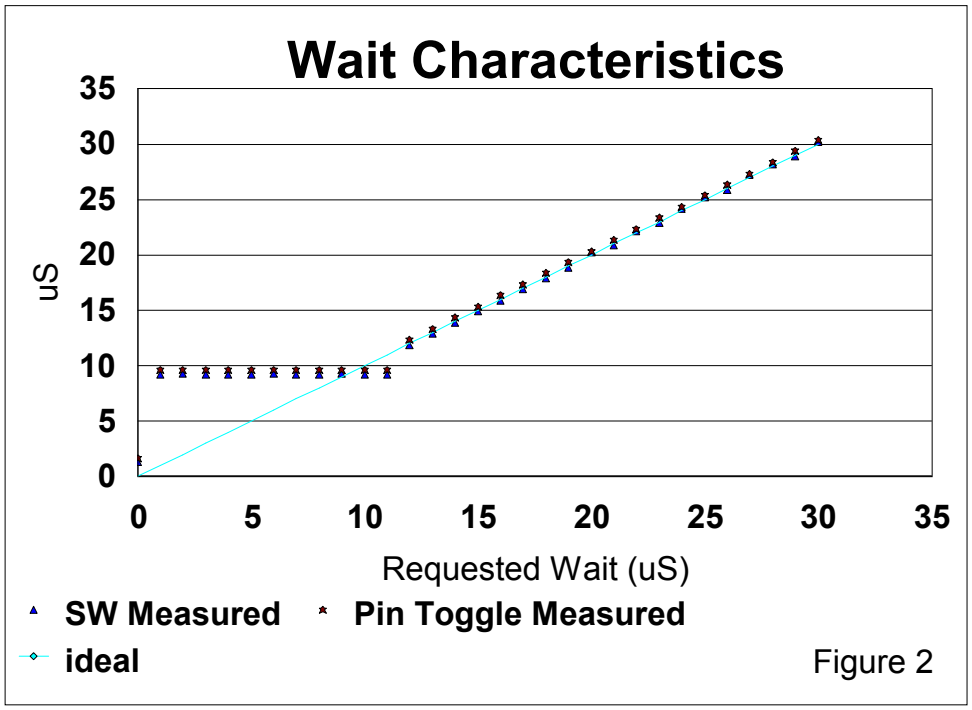

Figure 2

The next obvious feature is related to a short time protection code. When a wait of 0 μS is requested the routine returns immediately rather than even attempt to measure a wait time. The measurement at 0 μS then is 1.3 μS the shortest amount of time this routine will take.

Next the timing shows a constant wait time regardless of the wait requested. In the first part of this region the requested wait is less than the overhead for WaitUs. The routine shortcuts as before but at a later point. The final point of this constant response region (at 11 μS) is actually longer than the overhead. However once the overhead is subtracted the remaining time is still less than the minimum time that the WaitUs can wait for (as measured during startup) so for one last value the routine exits early.

The final region is from 12 μS on. In this region the actual time measured via software tracks the requested wait to within +/- 0.2 μS.

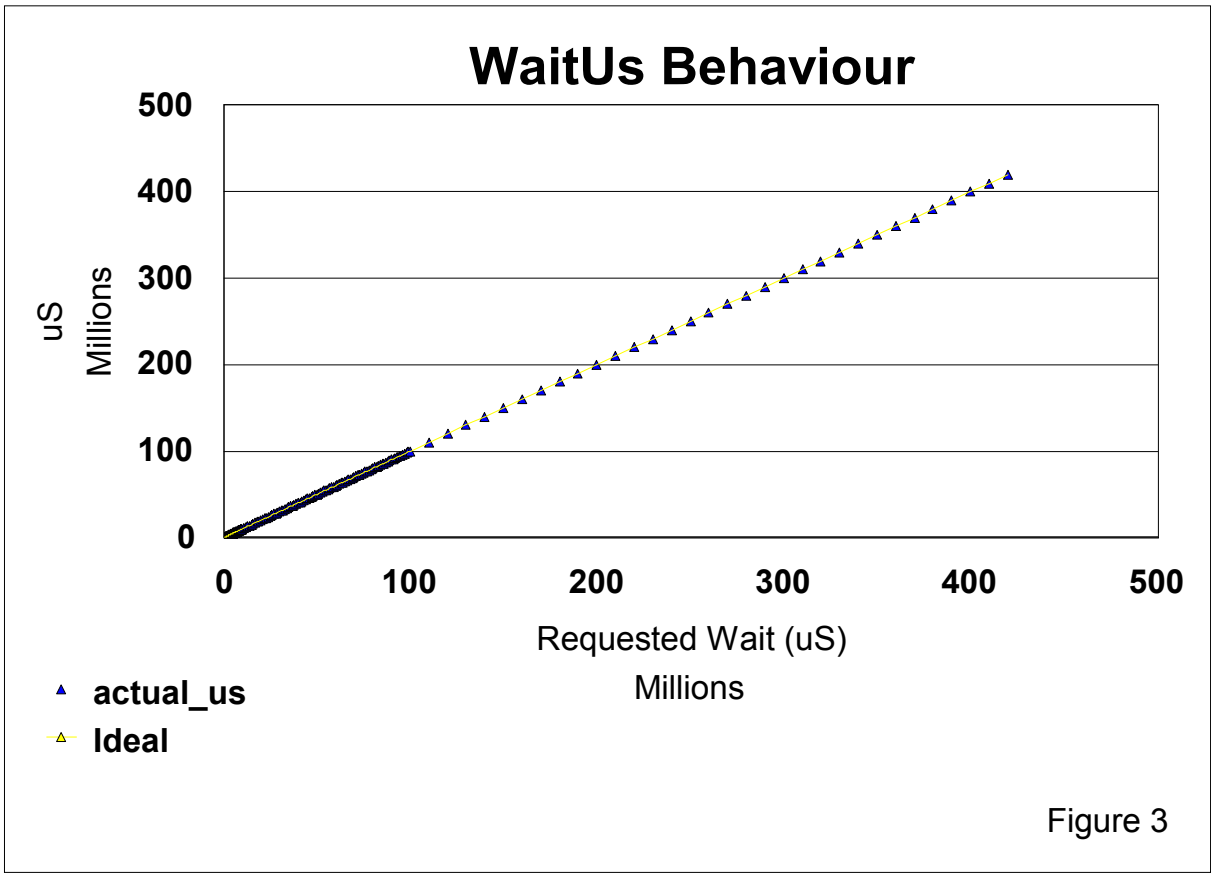

Figure 3

The automatic test was extended up to 420 seconds with the result shown in Figure 3. The difference between the requested wait and the actual wait is shown in Figure 4 and in greater detail in Figure 5.

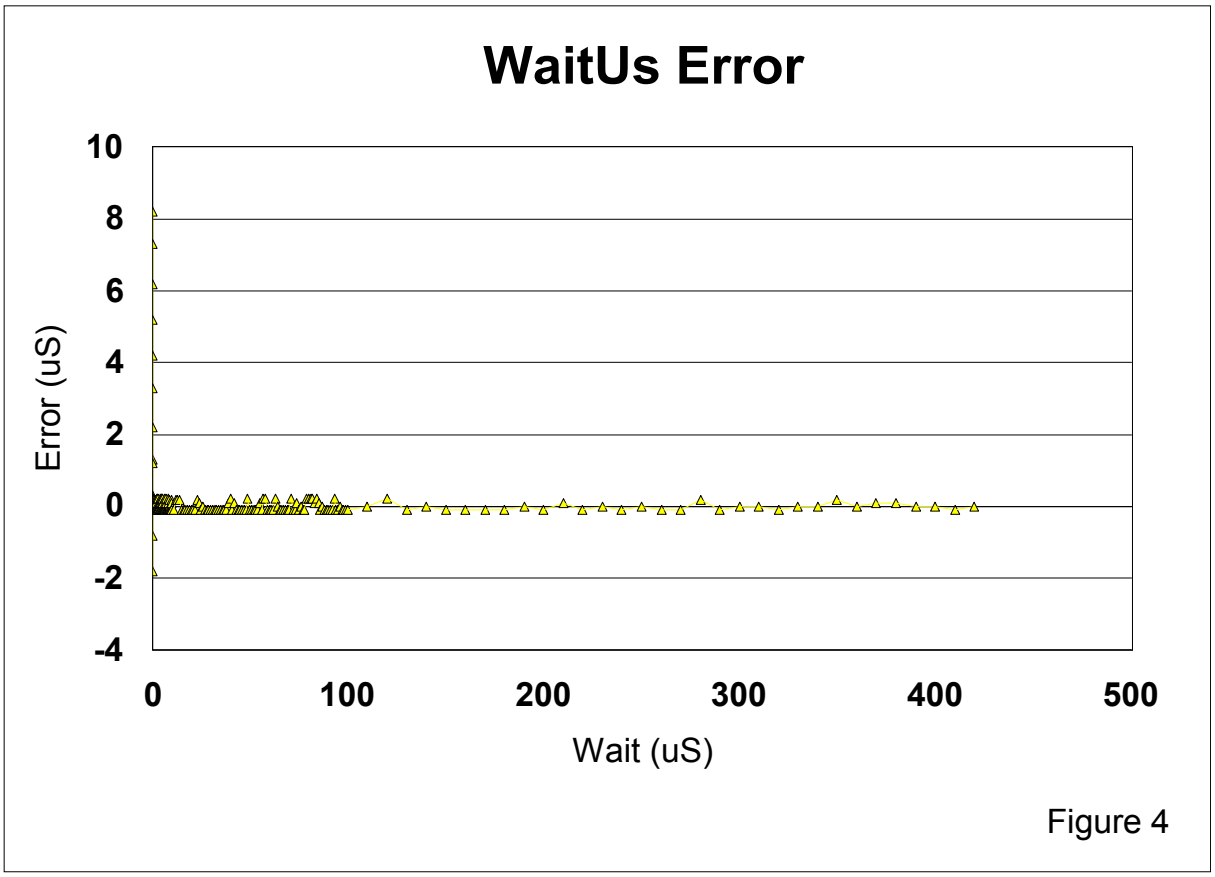

Figure 4

A final note about accuracy. An oscillator has limited accuracy, for example the SG636PCE from Epson has a stability of +/- 100 ppm. Over a period of 1 second that is the equivalent of +/- 100 μS. In other words the precision of the timer exceeds the available accuracy of the input clock. An important point to keep in mind when attempting high precision timing.

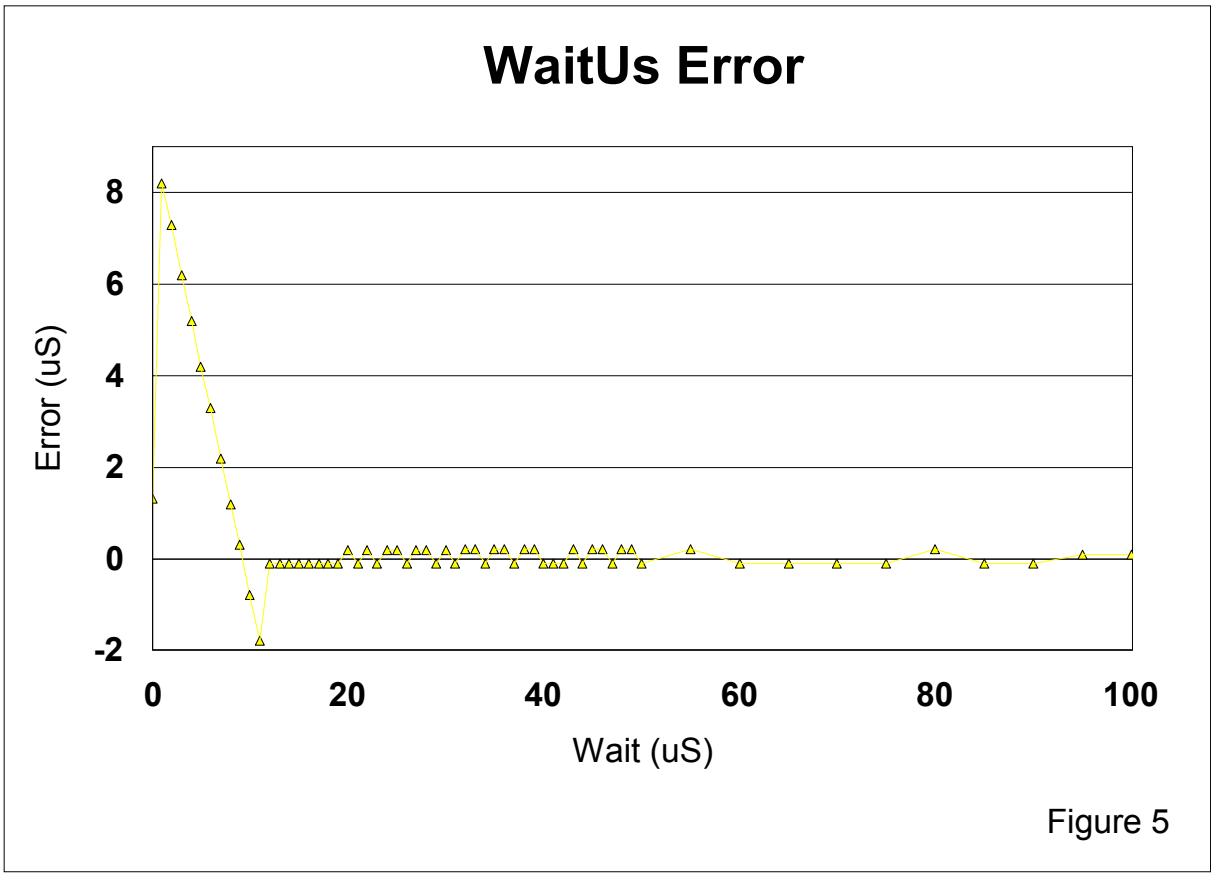

Figure 5

For time periods longer than about 430 S(for default parameters and standard conditions) the underlying HW timer will experience on overflow. Each overflow will introduce an additional timing error as the routine performs multiple matches to gain the requested time period. Those additional matches will have the same source of timing error near the boundary as well since the portion left after the split is small leading to periodic short flat spots in the response. Table 2 shows data from a measurement taken across one of those boundaries. While the data is not taken finely enough to show a 'flat spot' it clearly shows the jump in the accumulated error.

Table 1

Requested Wait	Actual Wait	Error
389,496,729	389,496,728.9	0.1
399,496,729	399,496,729	0
409,496,729	409,496,729.1	-0.1
419,496,729	419,496,729	0
429,496,729	429,496,729.1	-0.1
439,496,729	439,496,737.3	-8.3
449,496,729	449,496,737.3	-8.3
459,496,729	459,496,737.3	-8.3

A finer scale test was made around the HW timer over flow to give the result in Figure 7 showing the flat response area clearly. Note that at this point the error due to a +/- 100ppm oscillator would be over +/- 400 μS.

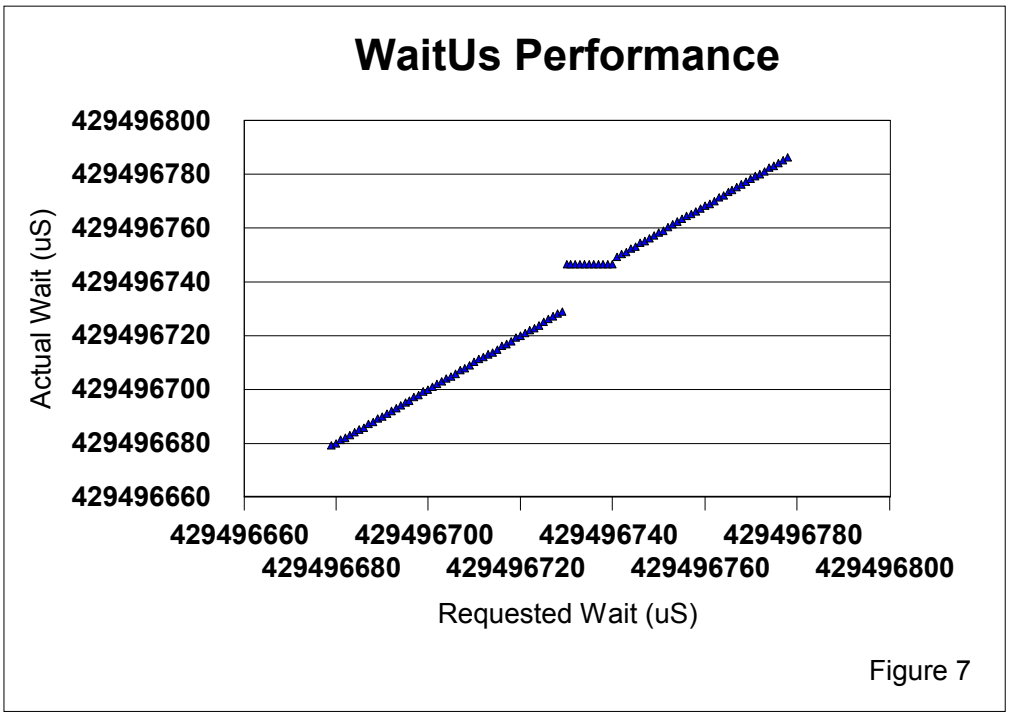

Figure 7

Finally a test was made using a 10 MHz oscillator with no PLLing to higher clock rates, MAM, VPB divider, and flash access left at default settings. The result is shown in Figure 6.

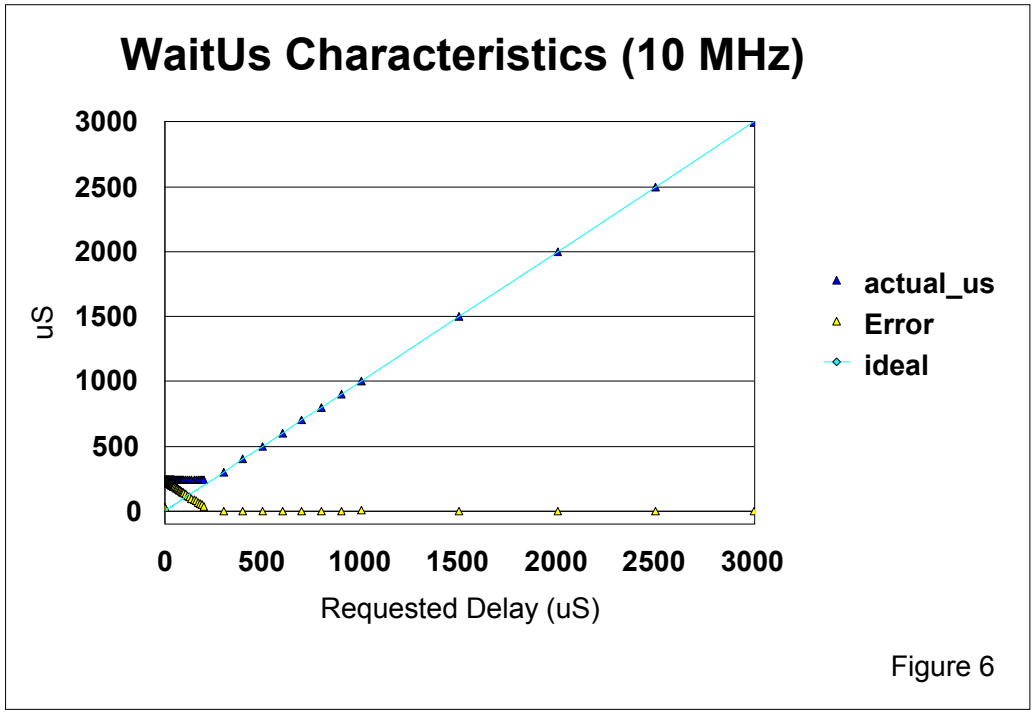

Figure 6