The SD16_A as a thermal random number-generator

Phil Ekstrom and Ray Glaze, Northwest Marine Technology, Inc.

The SD16_A analog-to-digital converter module, implemented in some members of the Texas Instruments MSP430 microcontroller line, can function as a surprisingly good source of randomly generated bytes, producing them at the rate of four per millisecond. The randomness can be traced to a fundamentally thermal source, so this is a truly random generator, not a pseudo-random one.

What to do – for randomly generated bytes.
Configure the SD16_A for maximum gain, input 7 (which is an internal short circuit), and an oversampling ratio (OSR) of at least 256. (Smaller may do in some cases - see below). Set the input clock divider to make a 1MHz converter clock and set the LSBACC bit to give access to the low order part of the converter's filter register.

When running with a 16MHZ MCLK, C code to accomplish this would be (assuming an appropriate header file that defines the symbols the same way the User's guide does):

```
//1 MHz, (MCLK/16), turn reference generator on.
SD16CTL = SD16XDIV_2 + SD16DIV_0+ SD16SSEL_0 + SD16REFON;
// Oversampling ratio set to 256, enable LSB access, no interrupts,
SD16CCTL0 = SD16OSR_256 + SD16LSBACC;
// Gain = 32 (actually 28), channel is 7 (shorted)
SD16INCTL0 = SD16GAIN_32 + SD16INCH_7;
// Now Go
SD16CCTL0 |= SD16SC;
```

After each conversion, the SC16IFG flag will set in the SD16CCTL0 register. If you have the interrupt enabled (by setting SD16IE in that same register) the module will post an interrupt. When the flag sets, the lower byte of the SD16MEM0 result register contains your new randomly generated eight-bit value. Reading that byte will reset the flag bit. At this clock rate and OSR value, the flag bit will set again with a new randomly-generated byte every 256 μseconds. To access it in C, assign the value in SD16MEM0 to a variable of type unsigned char.

This recipe assumes that the SD16_A is used for no other purpose. In fact it can be shared with another use, and configured as a random generator only when needed or when it is free from other demands. Even when doing its intended job as an ADC for nonzero signals, it will also be generating a noisy byte in the bottom of its output register as a result of each conversion. If it is run with high gain and an OSR of at least 256, and if the signal being converted lies safely within its input range limits, one would expect that the low order byte would be much like it is during dedicated operation. We have not investigated the quality of that byte in such conditions, but you may find it usable.

As discussed below, you may be able to use an oversampling ratio as small as 128, doubling the rate of byte production, but the resulting entropy stream fails tests 8, 12, and 13 (see appendix) of the NIST test battery.

Finally, for each processor to be used in this service, you need to verify that the least significant bit is actually a 1 half the time, and 0 half the time. Of 14 MSP430F2013 processors we have tested, 10 behaved perfectly. Four of them, however, showed a marked preference, for generating even numbers. They would show a 0:1 ratio in the least significant bit as high as 5:3.

If all you want is a recipe for randomly generated bytes with a uniform distribution, there it is.

What to do - for randomly generated numbers over some other range

For some purposes you need numbers uniformly distributed over some range other than 0-255, perhaps over a range that is not any power of two.

One particular case of interest is the Fisher-Yates Shuffle algorithm. That algorithm requires one number randomly generated modulo 2, another modulo 3, another modulo 4, and so on up to a final number randomly generated modulo N for shuffling N objects. See the Wikipedia article on the Fisher-Yates shuffle for a description of the algorithm and of the difficulties that can arise when randomly generating the input numbers that are needed to specify an output order.

In any such case you can start out with numbers that have a Gaussian (Normal) distribution with a standard deviation at least 70% as large as the numerical range you need. You then take those Gaussian-distributed numbers modulo your range. The result is a uniform distribution over that range. That is, in fact, just what we did in the case above, taking Gaussian-distributed numbers from the SD16 modulo 2^8.

However when the desired range is not a power of two, we will need to divide the word read from the SD16 by the modulus using unsigned integer division with remainder; that means we must start with a positive number. The offset specifications of the SD16 are not tight enough to determine the sign of the low order word of the filter register at high gain and large oversampling ratio. At an oversampling ratio of 128 the offset , specified as 1.5% of full scale, amounts to 2^{14} LSB in the lower end of the filter register, and that will fit in a 16-bit field. Because of the way the result is scaled by the ADC hardware, each doubling of the OSR multiplies the apparent offset (as seen in the bottom half of the register) by a factor of 8, so the offset at OSR=256 is 2^{17} LSB, which will not fit. We will need to do something more to control the average value of the distribution so that all of our numbers are positive. The simplest option is to use the general scheme above, but add three more processing steps:

1) We first generate two Gaussian numbers. In this case we set up the SD16 as above, but instead of a byte we load an entire 16-bit word from the low end of the SD16.
2) We form the difference of those two words (which makes numbers with a zero mean, no matter what the offset voltage of the ADC is) and finally,
3) We add the desired offset, ordinarily2^{15}=32768. (The actual value does not matter, so long as it prevents both overflow and underflow. 2^{15} is as far from both 0 and 2^{16} as we can simultaneously get.)

That scheme is very robust and especially suited to a production version of some product. Another possibility that is twice as fast per sample generated is to do a preliminary experiment to measure the mean of each particular SD16 for the particular mode of operation we will use, and then subtract that mean from each sample before adding the desired offset. One should probably re-measure the average every so often, perhaps at every power-on.

A more surprising possibility that still produces an entropy item for every conversion is to take the difference of successive ADC results, but to re-use the later one for forming the next difference. That is, from ADC results a_i, a_{i+1}, a_{i+2}, etc. one generates $x_i = a_{i+1} - a_i$, $x_{i+1} = a_{i+2} - a_{i+1}$ etc. That it can be useful is surprising because even if the a_i are uncorrelated, one can show that the original differences x_i (before taking them modulo the desired range) are highly correlated at lag 1. The expected correlation coefficient is r(1) =-0.5. It will turn out below that successive ADC words are already positively correlated, so the observed value of r(1) is not exactly -½, but it is negative and substantially different from zero. However it will also turn out that taking the stream modulo a sufficiently small number m, as we must do to get a flat distribution, will also erase the correlations. We leave

the details for a later section, but the point to be made here is that there is a third way of forming differences with known mean, that all three can be considered, and one makes twice as many results from a given stream of ADC outputs.

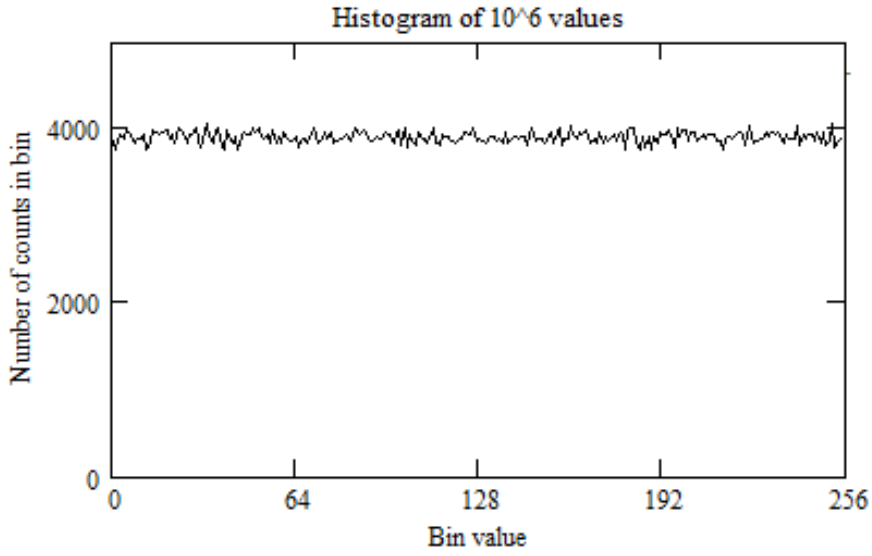
With our known-positive numbers drawn from a population with a Gaussian distribution, we divide by the desired range of each number we need, and each remainder is our randomly generated number over the range bounded by 0 and the divisor. We may divide each Gaussian number repeatedly until the product of all the divisors exceeds 1.4 times the standard deviation. For instance, we will see below that the difference of two Gaussian numbers generated with OSR = 256 has a standard deviation near 1250, and 1250*1.4 = 1750. If we are making integers for a Fisher-Yates shuffle, we can safely divide out remainders modulo the numbers 2 through 6, since 2x3x4x5x6 =720. We cannot safely divide again by our next modulus 7, since 720x7=5040 > 1750. Instead, we begin with a new Gaussian number from the SD16 and divide by 7 through 9, since 7x8x9=504 but cannot then also divide by 10 since 504x10=5040 > 1750.

If you know in advance how large N is, there is usually a way of grouping your divisors to make better use of the entropy you have generated than this example does, that is to make the product of your divisors nearer to the limit of 1750 than we did just above. We leave that as a matter of tactics for any given situation.

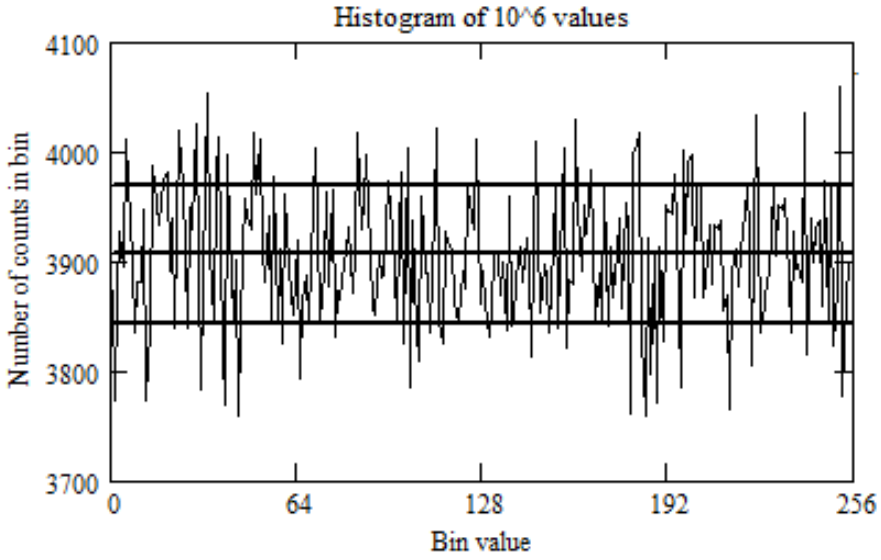
How well it works - bytes

If we are trying to make a number generator with a uniform distribution, one that produces all possible values in its output range with equal probability, we ought to check its output to see that all values really are produced about equally often. If we want the generator's past history to offer no clues about its future actions, we ought to check the autocorrelation function of the byte sequence to make sure that a large byte value is not, for instance, usually followed by a small one or vice versa. As will be argued in the appendix, these are necessary but not sufficient tests for randomness.

To run these tests, we need a file of output bytes from our generator. There is a program in the appendix which when loaded into the target board of an eZ430_2013 evaluation kit sends out bytes in start-stop serial form (for receipt by a UART) that have been randomly generated by the SD16_A in the manner described above. It uses the Timer_a module to simulate a serial communication port and runs at 115.2 KBaud with its output on P1.1. We removed the target board from the USB stick hardware of that evaluation kit, attached a serial-to-USB interface cable from FTDI, and read the byte stream into a PC for further processing.

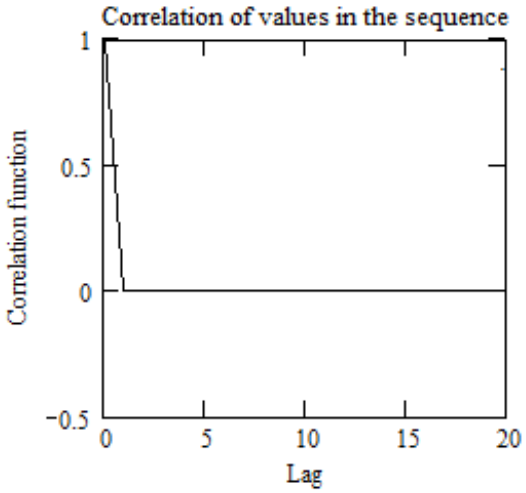


The results of these two tests on a sample of $N=10^6$ bytes generated in this manner are shown in the figures at the right.

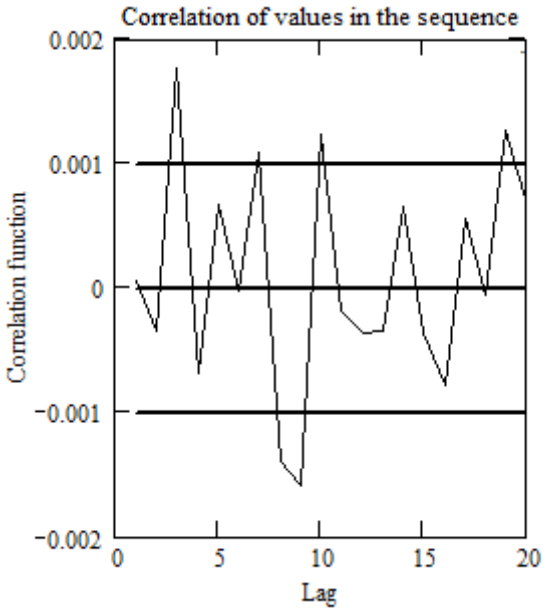


The first figure shows a histogram of values observed in the record, accumulated by setting up 256 counters (the "bins"), one for each possible byte value, scanning the record, and for each value observed incrementing the corresponding bin. The second figure is an expanded version with three heavy black lines indicating the average n and the average plus and minus the expected standard deviation $n \pm \sqrt{n}$. The average number in each bin is $n = N/256 = 10^6/256 =$ 3906.25. Theoretically we expect about 63% of the values to fall between the outer lines.

The next two figures show the results of an autocorrelation calculation based on the same data record for lags between 0 and 20. (See the appendix for details of the calculation.) The first figure includes for comparison the correlation at zero lag, which is by definition 1.



The second figure expands the scale for the remaining values and shows for comparison two horizontal lines at $\pm 1/\sqrt{N} = \pm 0.001$ indicating the expected standard deviation of the results for uniformly distributed randomly-generated values.



The TI user's manual text and figures describing the SD16 makes it clear that the value of the converted number does not settle in a single conversion cycle, and that for two conversion cycles after an input value changes there is a substantial error in the most significant bits of the ADC result. The new value remains correlated with the previous value. We would not expect this correlation to occur with constant input and in the noisy lower bits, and the results shown here confirm that indeed it does not. Again we expect about 63% of the values to lie between the outer lines, and we see that they do. Successive bytes in the sequence are effectively uncorrelated.

A stream of uncorrelated symbols, each occurring with probability p_i, has a Shannon entropy in bits per symbol defined by $S = -\sum_i p_i \cdot \log_2(p_i)$, where the summation index i runs over all symbols. In our case, $0 \le i \le 255$. For a theoretically perfect generator, all the probabilities would be $p_i = 1/256$, so $-\log_2(p_i) = 8$, and S=8.

We can approximate the probability of seeing a byte of value i by the relative frequency of occurrence of that value in our test sequence. With a finite sequence, the bin contents will of course not be exactly equal and not exactly equal to the underlying probability so we do not expect the resulting estimate to be exactly S=8, but with a large sample such as the one we have, we expect to be close if the generator is indeed random. To make that estimate we take $p_i \approx n_i/N$ where n_i is the number of counts in the i-th bin (as plotted in the first two figures of this section) and N is the total length of the sample, $N = \sum_{i=0}^{255} n_i$. Estimating S in this way gives S ≈ 7.99981 bits/symbol. We will not be far wrong to call that result 8 bits of entropy per generated byte.

The appendix contains a pointer to a suite of tests offered by NIST for candidate random number generators, and to the "Dieharder" test suite, a more convenient implementation of those tests along with several others.

It also contains some rude words about the ability of any test to actually confirm true randomness. Still, if we have a supposedly random generator of numbers, it ought to be able to pass those tests, as the appendix argues, most of the time. When we ran the STS suite on the same file of 10^6 randomly generated bytes tested above it passed all tests successfully. The result file is in the last appendix.

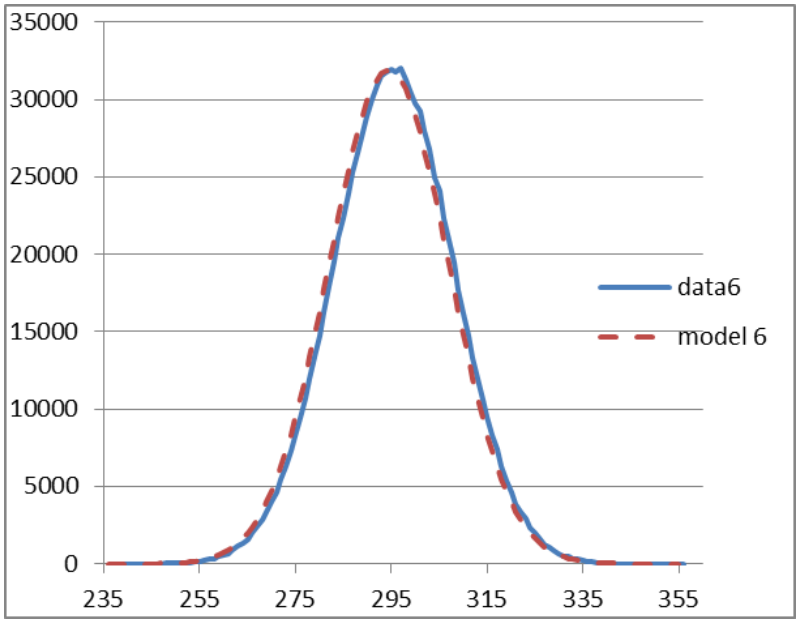
However when we condensed that experimental setup into a compact USB memory-key format and made ten copies for a project, we found that seven worked perfectly but three failed in an interesting and puzzling way. In those three units all odd byte values were equally likely to occur, and all even byte values equally likely, but the even bytes were more likely than the odd ones in a ratio of about 3:5! Something was wrong with the least significant bit! Unfortunately we had blown the security fuse of all ten processors before testing statistical quality, so were unable to perform a number of experiments which may have elucidated the matter. We replaced the processors in all three of the failed units and left the security fuse intact to allow investigations. One of the replaced processors misbehaved, and persisted in misbehaving when erased and re-programmed with the same image used originally, and also used in the successful units. Including the original processor from the EZ430 module, we have now tested 14 of these processors, and the SD16 units of 10 have worked perfectly for us. Four showed a marked preference for generating even bytes, and that preference seems clearly to be a property of the particular chip being tested.

As a result of this experience, we have included a health check in all of our subsequent experiments that will quickly verify that the least significant bit of the output is equal to zero as close to half of the time as one would expect. We will meet a related issue in the next section, and will discuss a possible quick health check there.

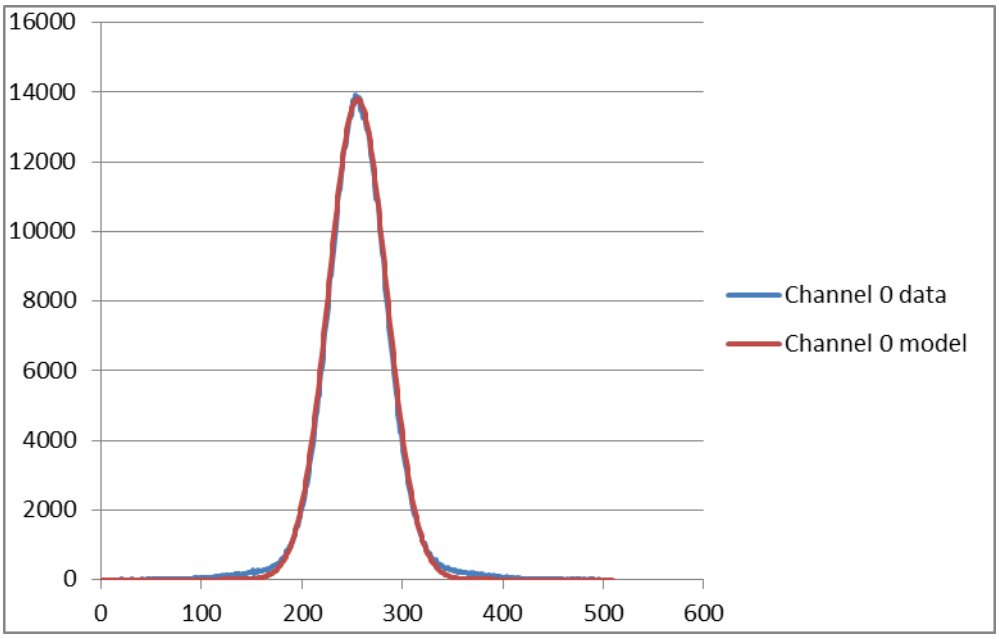
How well it works – Gaussian

We used two channels of the SD16_A in an MSP430F47197 processor to investigate the actual distribution of the nominally Gaussian ADC output. In each run we accumulated a 10^6-sample histogram for each channel in a 512-word array in that processor's larger but still limited RAM, using just over half of the available memory. Therefore the values had to be binned in classes of either 128 or (for cases where the standard deviation was smaller) 16 values per bin. We will see below that errors in going from Gaussian to uniform distributions occur in the form of a single-cycle approximate sinusoid. The binned data will be sufficient to exhibit that sort of problem to us whenever it is present.

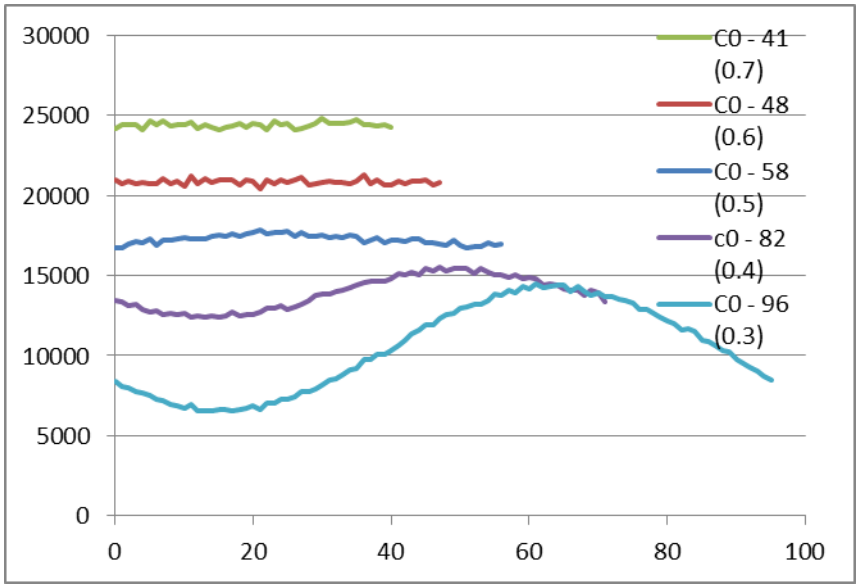
Well, how good are our Gaussian distributions? They are very good indeed. At the right is an example of single data values (not yet differences of successive pairs) generated with the oversampling ratio equal to 128. The horizontal axis is bin number (16 values per bin) and the vertical axis is the number of counts in the bin. The dotted red model trace is a theoretical Gaussian prediction of the bin contents for 10^6 numbers with the standard deviation best fit to the data. It has been deliberately shifted left by one bin to make both traces visible. When that is not done the data trace disappears behind the prediction. That best-fit model has a standard deviation of 12.5 bins, or 200 LSB.



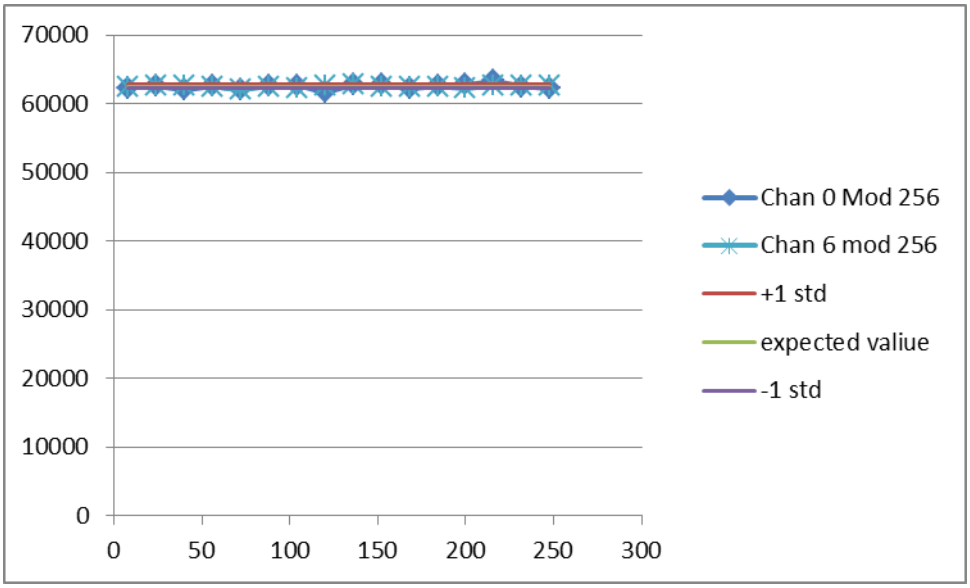
Histograms for OSR=128 and OSR=256 with two successive bytes subtracted look as good as the one shown just above. However with OSR=512, we see the kind of "fat tails" deviation from Gaussian behavior that is commonly seen in distributions of experimental data. Here the bins are 128 units wide and again 10^6 data values are represented. The data and best-fit model are here not offset, emphasizing the small region in the tails of the distribution where they deviate enough so that the data trace emerges from behind the model. The data can be modeled quite successfully as a sum of two Gaussians, one with a standard deviation of 3431 and the second with smaller amplitude and a standard deviation of 7770.



These “fat tails” that are so troublesome in some circumstances are in this context of no concern. Our rule will be that the distribution must be wide enough, and an extra contribution that is an even-wider second Gaussian does us no harm. So long as the central peak corresponds to a sufficiently large standard deviation, we get a uniform distribution in the end.

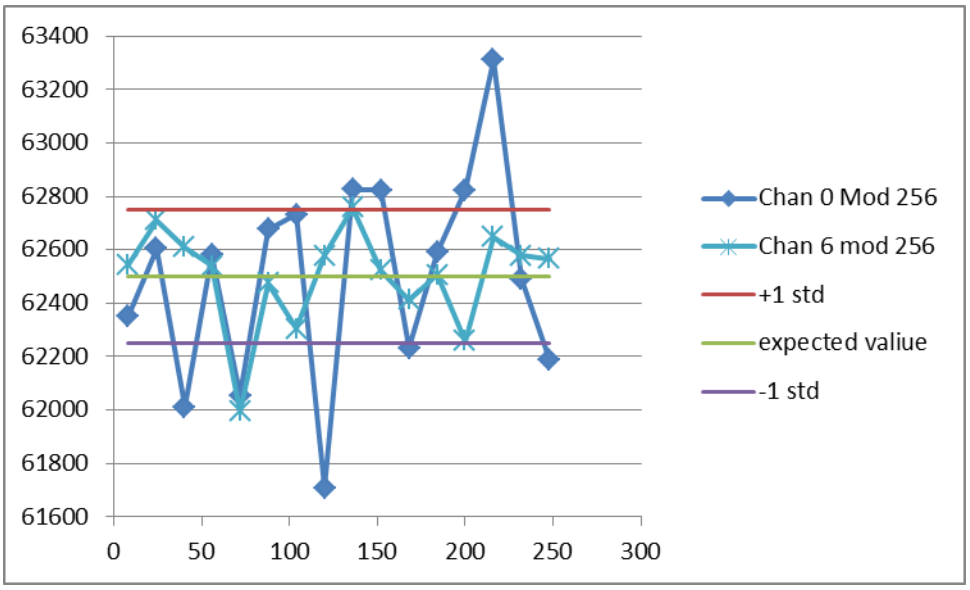


That rule will be justified theoretically later, but let us see some examples of it in action. At the right we see sums over the histogram above. Each individual case groups together those counts that would land in the same bin if the data were taken modulo some particular value that is an exact multiple of the bin size. If the standard deviation is to be as small as 0.3 times the modulus, then we sum the contents of bins that are 96 bins apart and obtain the wide variations shown in the bottom, light blue trace. If instead we follow the rule suggested earlier and require that the standard deviation be at least 0.7 times the modulus (modulus ≤1.4 times standard deviation) then we sum together data that are 41 bins apart and get the top green trace. Intermediate values of the modulus give intermediate results, with the total systematic variation decreasing rapidly as the modulus decreases. Towards the top any systematic variation is lost in the natural noise.

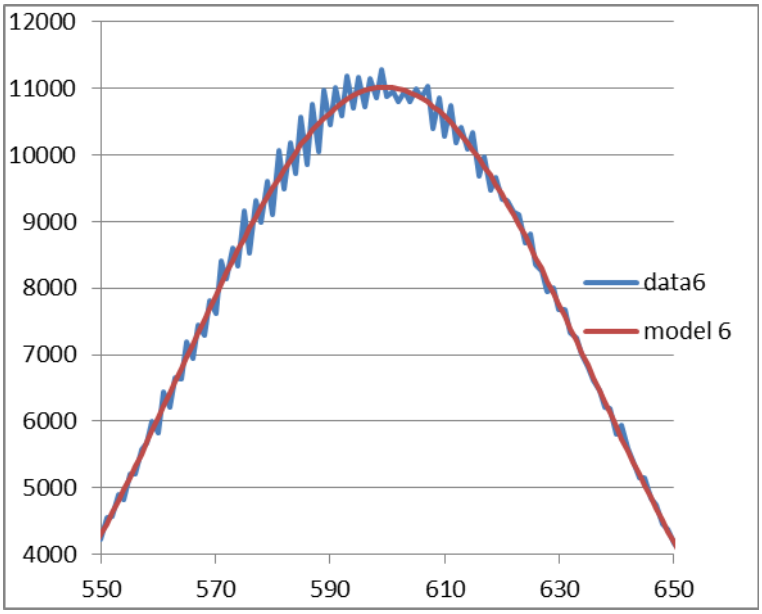


The rule illustrated above, that one can use a modulus as large as 1.4 times the standard deviation, suggests that one could take the output of even the data first shown that was generated at OSR=128 modulo a value as large as 200*1.4 = 280; one should be able to extract a byte from each. We can simulate doing that by again summing up for each bin n the counts in bin n, bin n+16, n+32, n+64 etc. The result is shown in the figure at the right, a satisfactorily flat distribution. The horizontal scale here is in byte values rather than bin numbers. Values on the vertical scale are the contents of the virtual bins summed as above. They represent the bin contents that would have been obtained if the values were first taken modulo 256, and then binned.

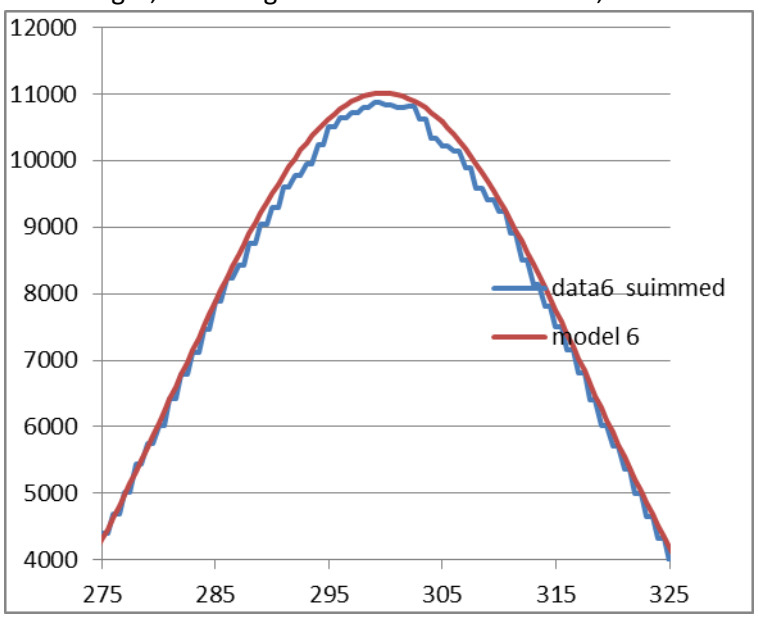
To allow a better look at the variations, the next figure suppresses the zero of the vertical scale. We expect that 63% of the points will be within the lines marked +-std. That is, we expect about 0.63*16 ≈ 10 points on each trace to be inside. The channel 0 trace has 7 inside, the channel 6 trace has 14. For both taken together we expect 20 and observe 21. With this small numbers of points all of that is satisfactory agreement.



This is the basis for suggesting that one can make bytes sufficiently randomly with the oversampling ratio as small as 128, which generates bytes twice as fast as a ratio of 256 does – about 8 per millisecond per ADC channel. A test file of 10^6 bytes generated at OSR=256 passed all of the NIST battery of tests for random number-generators. A comparable file generated as OSR=128 passed most but not all of those tests, so we use the larger oversampling ratio unless the higher generation rate is needed.
At smaller values of oversampling ratio the standard deviation of the noise is not sufficient to generate bytes (that is to use a modulus as large as 256) but they can be useful for generating values with respect to a smaller modulus. However histograms of data taken with OSR = 64 show a slight preference for even numbers, as shown in the magnified plot at the right. The resulting standard deviation is 36, which will allow use of a modulus as large as 50 if the preference, which we can estimate by eye to be about 10%, is acceptable.

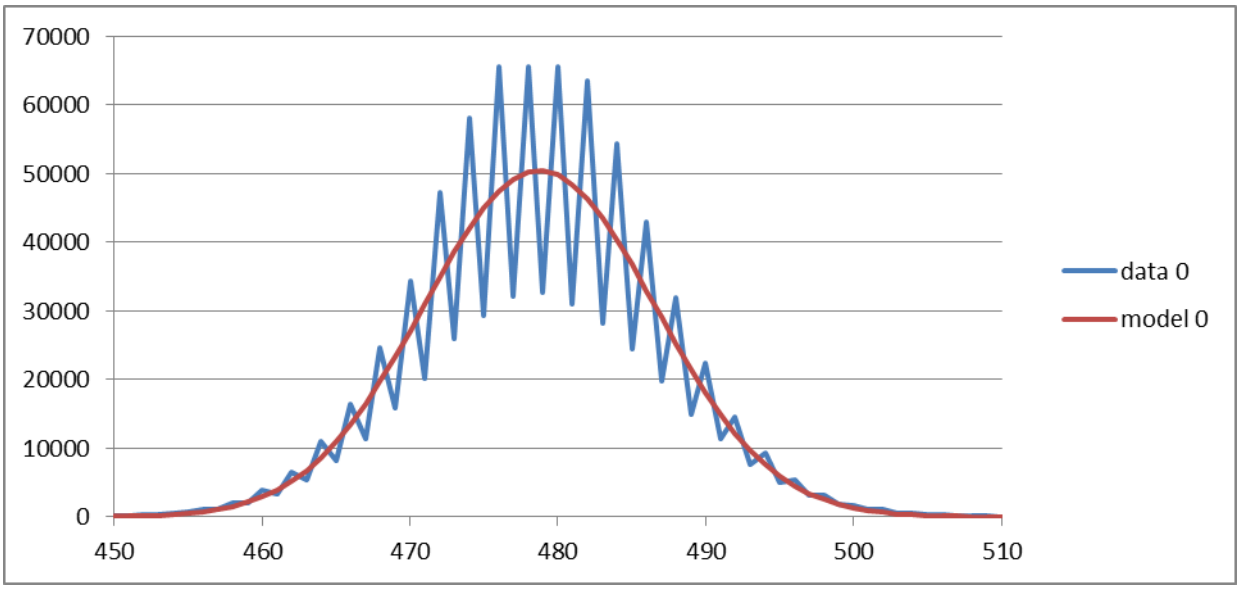


Alternately, the result can be shifted one place to the right, obtaining a standard deviation of 18, a maximum modulus of 25. The next histogram at the right was generated for this case by adding together each even numbered bin and the following odd-numbered one. To plot this on the same scale as the model curve above, the sums were then divided by two, and were plotted against a scale which is the original one divided by two. To get around Excel limitations without artificially smoothing the record, each summed data value was plotted twice, leading to the stair-step effect that can be seen in the data trace. The crucial point here is that with the LSB removed, there is no visible systematic preference for, for instance, either state of the next higher bit.



This result, obtained with MSP430f47127 chip 1 is a bit equivocal, because when the same options were set up in chip 2, both channels behaved well with no sign of a preference for even numbers, and gave a standard deviation of 18.5. So at least here it looks like the kind of failure we saw earlier with the byte generator where a few units had the disease, most did not, but the disease affects only some of the possible ways to set up the SD16.

However at OSR = 32 (the smallest available) even numbers become much more likely than adjacent odd numbers in the data, as illustrated in the next figure, and the disease afflicts both units comparably. One could again try shifting the data right one place. The regular progression of the peaks seen below lead one to expect that higher bits are indeed not involved and that one would meet success when doing that. However the result would have a standard deviation near 4 and a maximum modulus near 6 – of little practical use. Therefore we do not pursue this possibility and do not suggest operating at such a small value of oversampling ratio. Note that the central three peaks are clipped at 2^{16}=65536 in this histogram. These three bins became full and saturated; about 7400 counts were lost out of the 10^6 total.



This preference for even numbers is presumably related to the similar preference observed in the MSP430F2013 byte generator and mentioned in a previous section, but there are significant differences. First, the effect seen here was seen in both channels tested, in both devices tested, repeatedly at any sufficiently small OSR, but never at an OSR as large as 256. The previous instance was seen in 4 out of 14 separate processor chips operating at OSR=256. In this case the malady affected both channels of one chip at OSR=64, neither channel of the other chip.

However in all cases when the effect occurred at all, it seemed to occur stably. Modeling the process as a binomial distribution with p= ½ (so q=1-p=1/2), we expect that in a sample of N values the average number of zeros will be μ=Np$\rightarrow$N/2, and the standard deviation of that number will be σ=$\sqrt{}$(Npq) $\rightarrow$(1/2)$\sqrt{}$N. For a large sample, the number of zeros is approximately normal so we expect that number to be within 2.58σ of the predicted mean 99% of the time. That is, the observed number of zeros of an actually good unit will lie outside of the limits $\left(N/2-1.29\sqrt{N}\right)< s <\left(N/2-1.29\sqrt{N}\right)$ only 1% of the time. For a set of N=2^{16} generated values, the test limits would be $2^{15} \pm 331$. If a processor fails the test, it could be discarded wasting only 1% of actually good processors. One that failed initially can be re-tested, and accepted if it satisfies the condition the second time. In fact, all failed processors we have seen violate that limit by a huge margin and a much wider acceptance band could be used to catch all failures we have seen. Note that this test can be applied at either of two stages of the generation process, either to the full word when first generating the Gaussian samples, or to the uniformly distributed results of taking these modulo any even modulus.

Early experience applying this test to the Gaussian output of eight MSP430F47127 processors, testing ADC channels 0 and 6 of each processor so in all 16 channels were tested, showed that the results separated into two clearly defined groups, with one outlier. Two of the channels failed solidly, with their results never lying within the test limits. Thirteen of the channels behaved as predicted with the worst showing a failure rate of 1.6%. The outlier was an extreme 14th member of the well-behaved group, showing a failure rate of 3.1%. This appears to be a simple statistical fluctuation, since on re-testing this channel's results lay well within the others of the "well behaved" group.

A later examination was performed with N = 2^14 (that is, the LSBs from 16384 data words were summed, so the expected value was 8192). Each channel was tested at least 256 times, and one running overnight was tested 13,345 times. The largest results produced by the two bad channels in these trials were never larger than 7179 (12.4%low nominal). The smallest value produced by any of the 14 "well behaved" channels was 7937 (3.1% below nominal). Tentatively, since we have looked closely at only two misbehaving channels and four others in the byte tests only qualitatively, it looks like when operating at OSR=256 there is a clear separation between good and bad SD16 modules, and that a single test with N =2^14 (which takes only four seconds) will determine which of the two kinds we have.

Any similar test would have immediately flagged the earlier problem units. We suggest that all applications include a similar health check, both as a screening measure before installing the processor, and also in the application for ongoing monitoring.

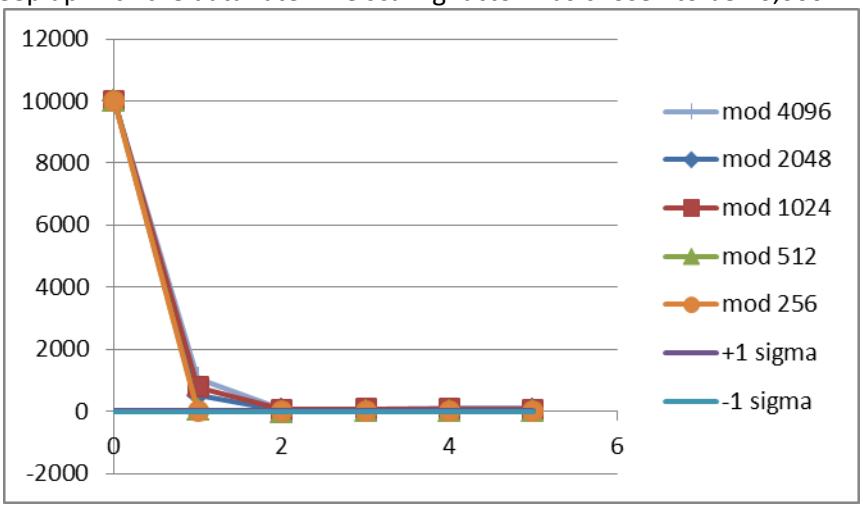
Summarizing our other observations of performance, various setups have yielded best-fit standard deviations as given in the table shown here. Several other items not yet discussed are also shown in that table, and will be referred to in the discussions that follow. The sample standard deviation (not displayed here) is in most cases slightly larger than the best-fit model standard deviation σ, as a result of a few outlier points in the tails of the

distribution. Avoiding undue optimism, we use the standard deviation of the model best-fit to the central peak. In particular, in the case of OSR=512, the smaller of the two standard deviations is quoted.

	Single Value					σ ratio	Difference (data not shared)			Difference (data shared)
OSR	even bias	offset LSB	σ	$\sigma \rightarrow$ max m	$r \rightarrow$ max m		σ	$\sigma \rightarrow$ max m	$r \rightarrow$ max m	$r \rightarrow$ max m
32	large	2^8	7.8	11	Data rate too high to compute correlations in the setup used.					
64	small	2^{11}	36/18	50/25						
128	none	2^{14}	200	280	256<max <512	1.10	220	308	512<max <256	512<max<256
256	none	2^{17}	1152	1613	2048<max <4096	1.09	1250	1750	2048<max <4096	2048<max <4096
512	none	2^{20}	3257x2	9120	8192<max <16384	1.05	3430 x2	9604	8192<max <16384	8192<max <16384

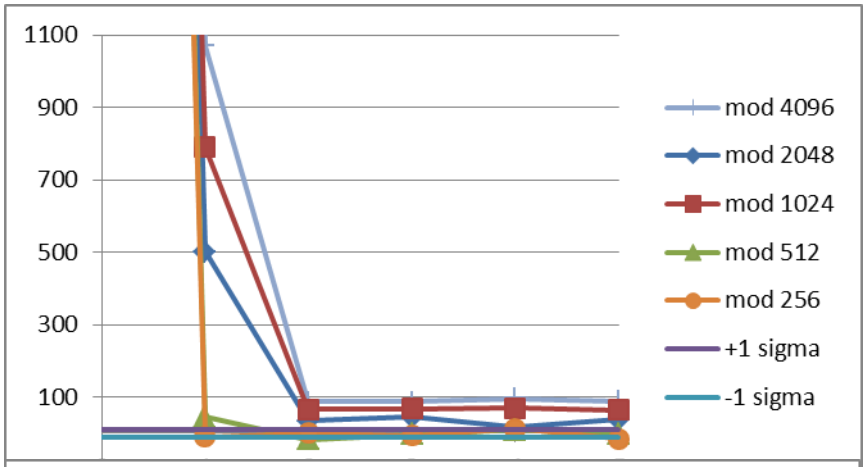
We have seen in the previous section that when we use only the low order 8 bits of data generated at OSR=256, successive bytes are uncorrelated. When we subtract two uncorrelated numbers from the same distribution, as we have done in some cases just above, we expect that the standard deviation of the result will be $\sqrt{2}$ times as large as that of either one, about 41% larger. What we see above is a standard deviation only 5% to 10% larger. This is our first warning that when we start to include higher order bits, successive ADC results may be no longer uncorrelated. That is, there another limit on the maximum modulus if we demand that the results be uncorrelated (which we do). The SD16 documentation does warn that if we change its setup the output may take up to three conversions to settle, but there is reason to suspect that, as with the single-byte data, it will not take as long for the lower bits to become uncorrelated with those of the previous sample.

To explore this issue, we calculated the autocorrelation function in the MSP430F47197 for some of the cases above. The calculation was implemented using integer arithmetic for speed, as described in the appendix, and the words read from the SD16 were taken modulo various powers of two, the only moduli for which division could be performed rapidly enough to keep up with the data rate. The scaling factor was chosen to be 10,000 in order to preserve four decimal places in the result. That is, a correlation of 1 will correspond to 10,000 units on the vertical scale of the graphs here. The horizontal scale will be the lag L.

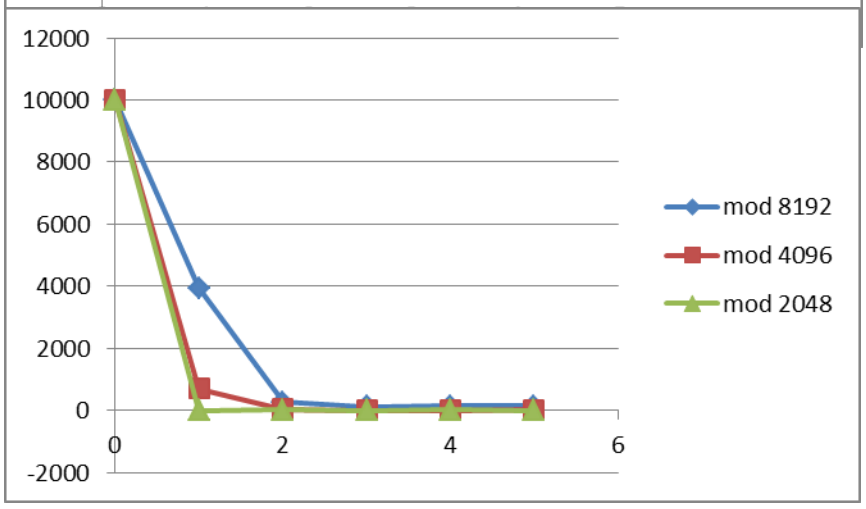


For OSR=128, the results are as shown at the right. Indeed there is a visible correlation at one unit lag which decreases rapidly with lag and with oversampling ratio. The next graph expands the vertical scale to show more detail.

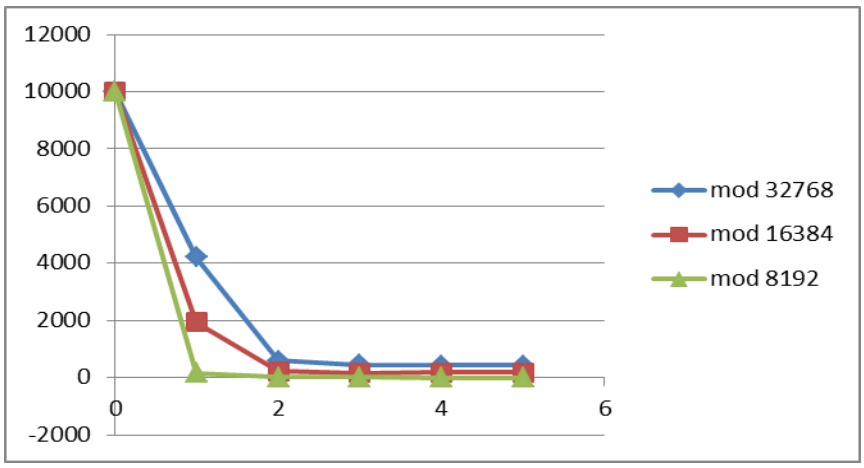
As the modulus decreases, the correlations also decrease. When the modulus is as small as 512, the correlation at one unit lag is only 0.0045, and at modulus 256 the largest correlation magnitude (at lag 5) is -0.0015, or 1.5x the expected standard deviation for these points. Yes, there is a separate limit on the maximum modulus that is set by the need for uncorrelated samples, but one that is not more stringent than the limit m < 280 set by the requirement for a flat distribution.



The correlations are larger for larger OSR, as we might expect from the fact that differencing successive samples offered an even smaller increase in standard deviation. The data are shown here for OSR= 256. A modulus of 4096 brings the correlation at one lag down to 0.07, and for 2048 it actually goes negative, -0.001. Again, the earlier limit of m < 1316 is more stringent than the one set by correlations.



For OSR = 512, the story is much the same. In this case the data were divided by 2 before processing, so the modulus 8192, which here is seen to reduce the correlation at one lag to 0.016, is equivalent to 16384 applied to the raw data. An actual 8192 is shown here as 4096, and reduces that correlation to 0.0018. The limit set by the flatness requirement is here m < 9120, again more stringent.



These results have been added to the table on the previous page that kicked off this investigation, and the conclusion is clear: if we obey the proposed limit that $m \leq 1.4\sigma$, we lose nothing significant to serial correlations. While several conversions must elapse for the entire word to be uncorrelated with a first conversion result, the numbers we divide out of that result will be insignificantly correlated with the ones divided out of the previous converison result.

The fact that the nonlinear modulo operation removed the correlations in this case suggested that it might also remove the correlation that would arise if we were to difference our ADC result in overlapping pairs, as mentioned in an earlier section. Without much theory to guide us here, we just repeated the same experiments done above for the case where we did overlapping differences. The results were much the same. Although the correlations seen were different, always negative when they were large, they came down to the expected noise level within the same modulus range. These results have also been added to the table above, under the heading “data shared” differences.

Again we conclude that if we obey the limit on modulus m set by the flatness criterion, correlations from either of the two sources discussed will be eliminated.

Note that the standard deviation increases by a factor near 6 when doubling the oversampling ratio, offering between two and three more bits of entropy. However halving the oversampling ratio generates twice as many data words per unit time, offering twice as many bits. Operating at low OSR maximizes the rate of entropy generation.

Why it works at all

Every time a capacitor is connected to a resistive source, allowed to settle, and then disconnected it acquires a thermally generated noise voltage with standard deviation $V_{RMS} = \sqrt{kT/C}$ in addition to whatever voltage the source is intentionally providing. This random addition is called "contact noise", but it is actually the resistor that is noisy, not the contact, as explained in the appendix. Here k is Boltzmann's constant and T is the absolute temperature, so at room temperature the result becomes $V_{RMS} = 64\mu V/\sqrt{C}$ for C in picofarads.

Every microsecond when the input sampler of the SD16_A contacts the external circuit and the voltage on its 20pF sampling capacitor settles to a new measurement of the nominally zero input value, that value is zero plus or minus "contact noise" that has a standard deviation of 64/√20 = 14 microvolts. That voltage is amplified by a factor of 28, so will have a standard deviation of 0.4mV, and applied to the sigma-delta modulator. In the course of the conversion, 256 of these values (for OSR=256) will be more-or-less averaged by the digital filter, to yield a random contribution no smaller than 400/√256≈25μV. This truly-random contribution could possibly approach that theoretically smallest value if the filter simply averaged. In fact it does something more complicated to minimize the shaped quantization noise of the second order delta modulator, so we expect that there will be more of the truly random noise than this. Also there are other noise sources in the modulator, so for that reason also the noise we have just estimated is a minimum.

For OSR = 256, the most significant bit of the output register is bit 23 (see figure 26-5 of the MSP430F2xxx User's guide, SLAU144J), and that bit is worth 600 mV. The least significant bit of the register is therefore worth 2^{-23} *600 mV≈72nV. That means the 25μV noise contribution is worth at least 25μV/72nV = 350 LSB, and we expect it to be Gaussian noise, distributed along the familiar bell-shaped curve of probability density. Again this is a minimum.

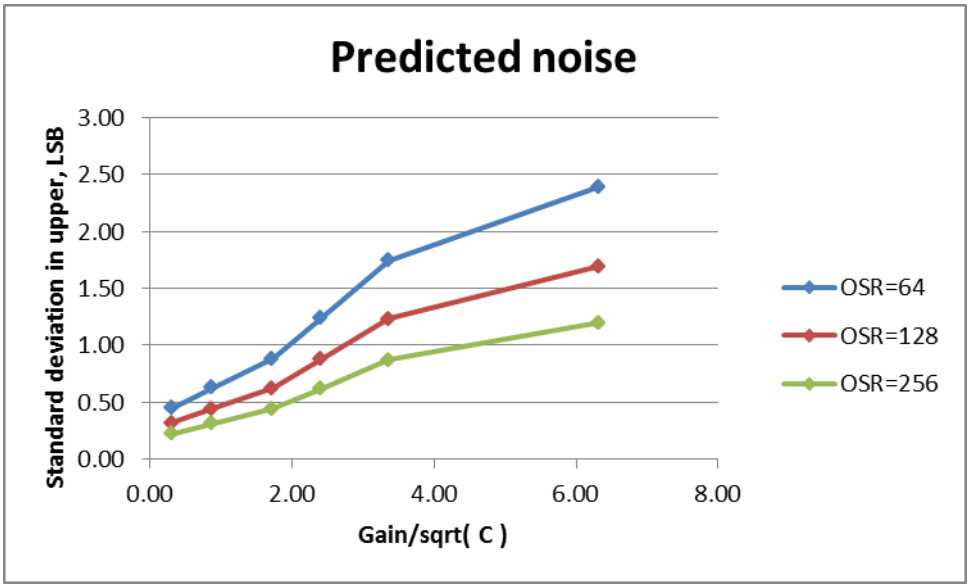
The simple treatment in the paragraph above requires some assumptions about just how the input amplifier and delta modulator are constructed and operated. Based on the gain and capacitance specifications of the SD16 and the fact that the module still offers gain when the active amplifiers are omitted, I have assumed that each of the two Cs capacitors (10pF at gain 32) in figure 26-2 of theF2xx users guide (SLAU144J) is made of eight 1.25pF capacitors (like the one that is used at gain 1) that are charged in parallel and then connected in series when presented to the ADC core. A final factor-of-four gain is achieved some other way, perhaps in the delta modulator, or perhaps by actually splitting each of the 1.25pF capacitors four ways. There are other ways that the input amplifier could be operating, but most of them lead to the same noise estimate. None we have thought of leads to a smaller one.

Recall that in the experiments reported above we saw a standard deviation of 1250 LSB for OSR=256; the output is actually noisier than this estimate would lead us to expect, but that's OK even though it is not quite clear just where that extra noise is all coming from. We can guess that it is partly additional contact noise arising in the integrators of the delta modulator, which are likely implemented using switched-capacitor techniques. Even though we are less confident that we can count on all of this additional noise despite unit-to unit variations, we do note that it is remarkably Gaussian, and are glad to have it.

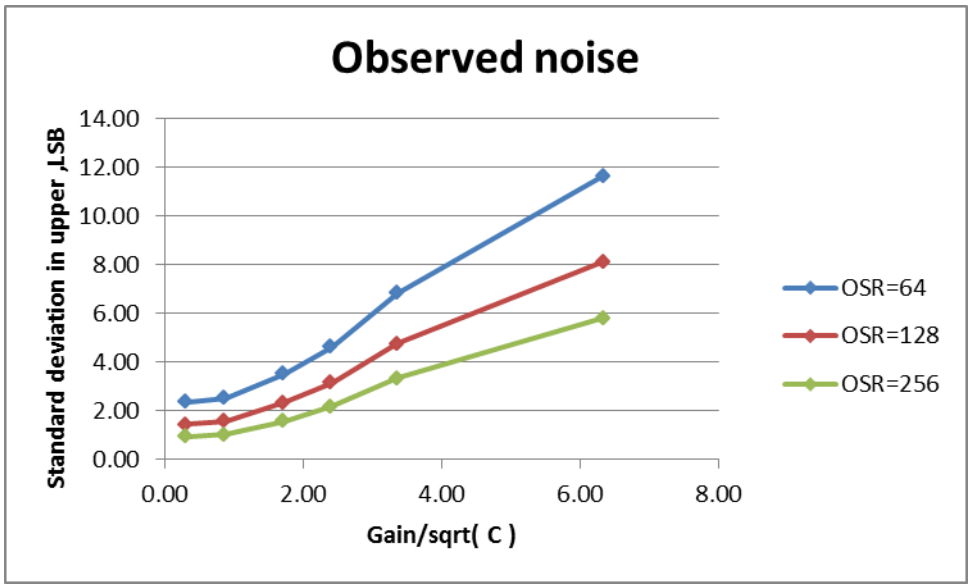
To check on this noise estimate, we wrote a program for the eZ430F2013 that accumulated statistics on the SD16_A output in the upper (normally used) section of the SD16 output register. For comparison, we need a prediction based on our noise model above for what we should expect. The MSB of the upper output register is worth 600mV at Gain=1, so its LSB is worth 600mV*2^{-15} = 18μV. At any other gain it is worth 18μV/G. Our simple model of the filter effect is that it will simply average the noise samples, so we expect them to be

attenuated by a factor of $1/\sqrt{OSR}$ Thus we predict a noise in f the upper register equal to $\frac{64\mu V}{\sqrt{C}}\frac{G}{18\mu V}\frac{1}{\sqrt{OSR}} = 3.6\frac{G}{\sqrt{OSR \bullet C}}$ LSB.

Plotting this out vs. $G/\sqrt{C}$ for the various available gain settings (tick marks on the traces) and interesting values of OSR, we obtain the figure on the right.



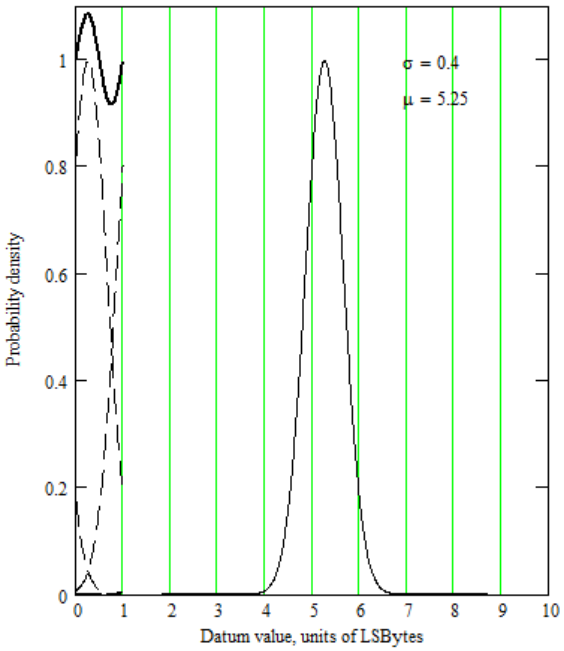
Running a program in an eZ430F2013 to calculate the mean and standard deviation of 10^6 conversions for each of those cases gives instead the next figure. It has approximately the same shape, but the measured values are about four times larger than the predicted ones. If we knew where all that came from and how securely it was tied to a thermal source, perhaps we could confidently use a smaller OSR and generate bytes more rapidly.



The experiments reported above in “How well it works” (done later than these and based on best-fit Gaussian standard deviation instead of sample standard deviation) have in this revision increased our confidence in the unaccounted-for extra noise.

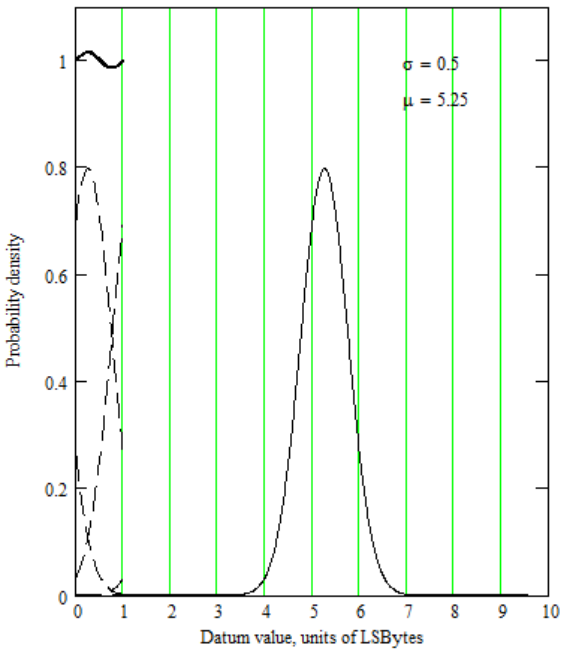
Why it works so well

When you have a set of data with a Gaussian distribution and you take those numbers modulo something comparable with or smaller than their standard deviation, the result always comes out to be nearly uniformly distributed. That is the real key; if you have enough Gaussian noise to fill your byte, you don't care about much else. The rest of this section will show how that works.



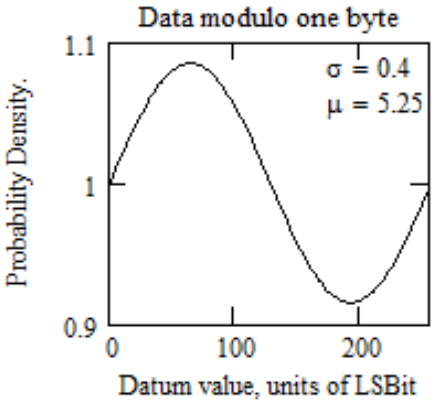
In the example at the right, we see some graphs illustrating how that begins to take effect, where in these graphs we are taking things modulo 1. In our present application we can think of that unit 1 as being one byte; in our hardware the numbers are taken modulo 256, which is one byte. In the first figure the standard deviation of the Gaussian is less than half a byte – still a bit small – but when you take the ordinate of the curve modulo 1, that is when you slice it along each of the vertical green lines, superimpose those slices, and add them up as has been done with the dashed lines on the left side of the graph, you get a sum (heavy black line) that is not yet flat, but already has sharply limited variation. Throughout this section the mean of the original distribution has been chosen ¼ unit off-center to cause the maximum possible variation in the black trace.

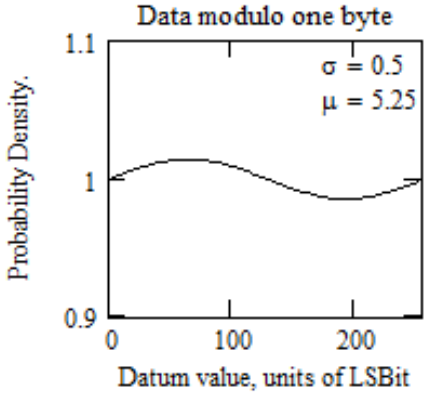
Well, how about adding a little more noise? When the Gaussian is only 20% wider, the variations in the heavy black trace shrink markedly. Let's plot that black trace separately, and follow its behavior as we increase σ (and thereby increase the width of the Gaussian) by a few more steps.



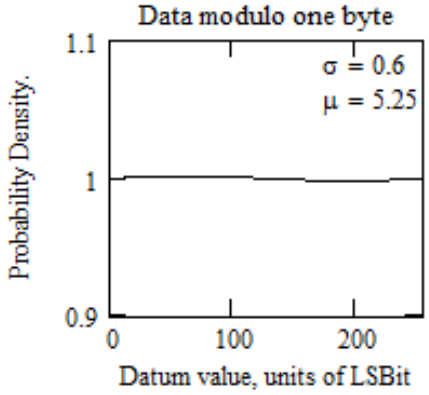
In these graphs below we have shown the X-axis scale in LSB, 0 to 255, replacing the scale of 0 to 1 bytes, now that we do not have to also show the Gaussian peak. The values of μ and σ are still shown in bytes for compactness and would have to be multiplied by 256 to get the corresponding values in LSB.

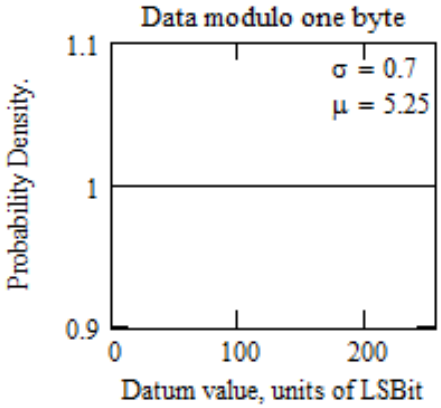
The first two graphs represent the two cases shown in full above.



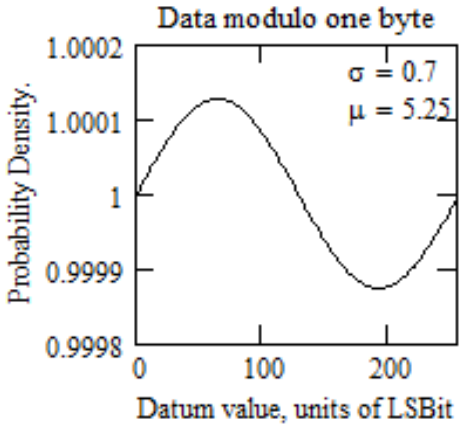


As the standard deviation increases, the probability density rapidly flattens out until its variation becomes completely invisible when plotted at the original scale.

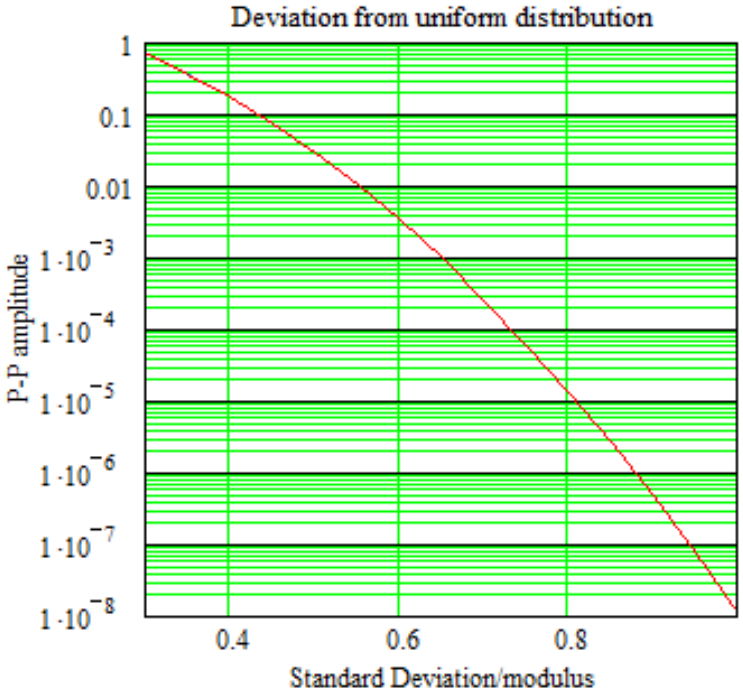




Now plotting that last case again but allowing the vertical scale to adjust we see that the shape of the variation has not changed, but that its scale has become so small as to be undetectable in practice. To get that degree of flatness takes a standard deviation of 0.7 times the modulus value. For data taken modulo one byte, that requires 180 LSB of noise standard deviation. More noise is better, so long as it all stays within the input range of the ADC; our OSB=256 case, with a contact noise contribution of 350LSB, is safe by a wide margin.



Having seen all of these particular examples, let us write out in general that the probability density of a Gaussian sample with standard deviation σ and mean μ taken modulo a number m is $g(x+\mu,\sigma)=(1/n)\sum_{i=-\infty}^{\infty} N(x+n\cdot i,\sigma)$ over the interval [0,n), where N(x,σ) is the Gaussian density function with mean 0 and standard deviation σ. The maximum of this function is at x+μ =0, the minimum is at x+μ = n/2. Plotting the fractional difference of these two vs σ/n gives the following:

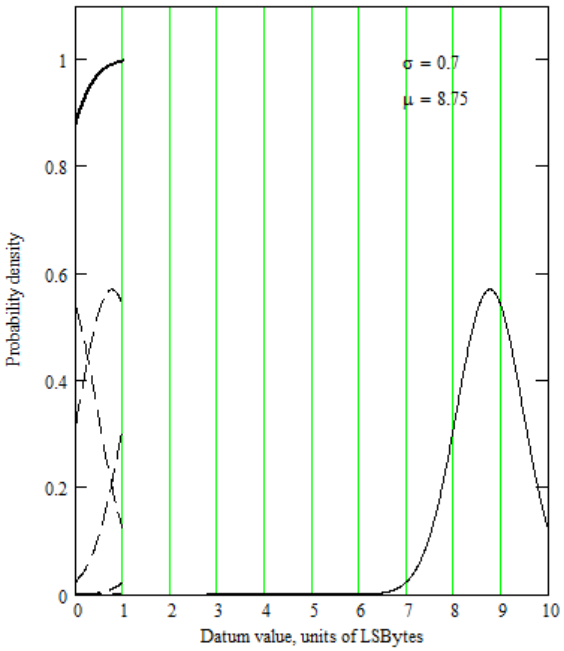


Depending on the needs of a given application, one may choose to work at a variety of positions along this curve, but for sample sizes no larger than 10^6, the criteria $m \leq 1.4\sigma$ or $\sigma \geq 0.7m$ would seem to beadequate, in that any systematic deviations from uniformity would be lost in the expected statistical variations in observed frequency. Those criteria are obviously a bit soft and while not quite equivalent they have been used interchangeably in this document.

There is additional value to an extremely flat distribution made by folding up a Gaussian as we have done above; you can't easily disturb it by adding something else to it. Adding something to the input of the ADC just moves the mean of the input Gaussian distribution. But when the resulting output distribution is flat, then the location of the original Gaussian's peak no longer matters. All that moving the peak could ever do was to move the location and perhaps reduce the height of that wavy trace we have been watching. If the wave has a small enough amplitude to be undetectable, then the effect of a change in the mean of the noise is also undetectable.

That is the starting point for a useful way to think about other contributions to our noise generator's result. All that any additive (interfering) signal can do is to move that mean. If it moves the mean and leaves it in a new location, then quite clearly it has no effect on the performance of our generator. We won't see the same result of a conversion that we would have seen without the addition, but we will see another value drawn from the same nearly-flat distribution. It will be just as unpredictable, will be drawn from the same population of values and will have the same distribution. If the added signal moves the mean back and forth between conversions, we still won't have any way to tell that it has done so or any reason to wish that it hadn't. If it changes back and forth during a conversion, we expect that all it can do is to broaden the noise distribution, and as we can see that helps us out. Once we have enough Gaussian noise, it looks like we can relax about interference. So long as the SD16_A is well enough shielded to do its normal ADC function, it should be able to make high quality noise bytes.

It is worth noting that we do need the SD16_A to be working well as an ADC, and as already mentioned we need the entire input noise voltage range to fit within its linear input range. If one tail of the input noise gets outside the linear range of the ADC function, we can no longer guarantee a flat noise spectrum in the result, as illustrated in the plot at the right. The output value distribution in the case illustrated here is no longer flat. It has a shape that is just the negative of the missing tail of the original distribution.



When the SD16_A is operating with its input range symmetric around zero, and with the input shorted as we do here, that requirement is easily met and the situation diagrammed here is easily avoided.

Conclusion

The SD16_A sigma-delta ADC can be operated as an excellent random generator of values, where the source of randomness can be traced to thermal noise. However a minority of SD16 modules exhibit a preference for generating even numbers rather than odd ones. The possibly-biased LSB of each result can be removed by shifting the result right one place, but that decreases the range of numbers that can be generated. If the LSB is retained, any processor used in this service should be screened to identify and discard those units that show this preference for even numbers.

Appendices

About Contact Noise

It is not actually the contact that is noisy.

Whenever you connect a warm resistor across a capacitor, the resistor generates Johnson (thermal) noise and applies it to the capacitor. Meanwhile the same resistor is discharging that capacitor. As the net result of those two actions, there is a random voltage appearing across the capacitor, which fluctuates as long as the resistor is connected. If the RC product is small, the range of frequencies involved may be very large and mostly outside the passband of a typical sensitive amplifier that may be looking at the capacitor. The amplifier may see nothing happening. When the resistor is disconnected from the capacitor, as happens every microsecond in the SD16_A, the voltage suddenly freezes at a definite value and can be seen by narrow-band circuitry. Since a new frozen value appears any time the capacitor is contacted long enough for a new equilibrium to be established, then disconnected, the effect has acquired the name contact noise even though the switch contact is not actually the source of the noise. The electrical resistance of the switch and circuit is the actual source, and what it contributes is thermal noise.

An electrical engineering text which analyzes this effect might proceed by integrating the Johnson noise spectrum over the bandwidth defined by the resistor and capacitor, considered as a single-section low-pass filter, and thereby could derive an expression for how much noise to expect. While it gives a nice picture of what is going on to anyone who already knows about Johnson noise, that approach is complicated; we won't do it that way.

A physicist looking at the same problem would more likely think about the Equipartition Theorem from Statistical Mechanics. It states (roughly) that any quadratic energy term in a system will have an average value equal to kT/2 when the system is in thermal equilibrium. In our case, we notice that the voltage across a capacitor C gives rise to an energy term ½ CV^2 which is quadratic in V, its terminal voltage. We can set that term equal to ½ kT, then solve for the mean-squared terminal voltage $< V^2 >= kT/C$.

At room temperature this becomes $V_{rms} = \sqrt{< V^2 >} = \sqrt{kT/C} = 64\mu V \Big/ \sqrt{C}$ for C in picofarads, the result quoted in the body of the note. The 20pF input capacitor of the SD16_A will have a root-mean-square thermal noise voltage due to this effect of $64\mu V \Big/ \sqrt{20} = 14\mu V$.

About Random Numbers

There aren't any. However there <u>are</u> randomly generated numbers.

Once you have a number, however generated, it has a perfectly definite value and there is nothing random about it. The only thing that can be random about the situation is the way the number was generated. Said more compactly, a number cannot be random, but its value can be chosen randomly and therefore can be unexpected. For most purposes it is fair to call the <u>process</u> that generated it truly random if there is no way to predict (better than chance) the number that the generator is going to make next – even if you have full knowledge of the generator's history and internal state.

So it is fair to talk about random <u>number-generators</u>, but not <u>random-number</u> generators. Mostly, people are not careful about that distinction, but this note will try to be.

About Testing for Randomness
There is no good way to do it.

In the output from a perfectly random generator of numbers, at least one with a uniform distribution like we are trying to make here, all possible bit sequences are equally likely. A string of all ones or all zeros looks to us wildly non-random, but it is as likely as any other particular sequence in a random generator's output. In a single-byte result from the generator proposed here, you will see a solid zero about 15 times a second. You will see a pair of two zero bytes together more-or-less every 16 seconds, a trio of three in a row more or less every 72 minutes, four in a row about every twelve days, and so on. No particular sequence of the same length that you could name would be either more or less likely than one with all bits zero. You can't call any of them right or wrong, random or non-random in themselves.

What you can do is to test for properties that a large majority of randomly generated sequences is likely to have, realizing that any such test will sometimes call foul on a sequence that came out of a perfectly random generator. A truly random generator must eventually make all possible sequences – including all of those that fail your test. Also, there may also be chaotic sequences generated by a perfectly predictable (pseudo-random) process that always pass such a test, which a genuinely random generator could not always do.

In short, a generator that fails a sensible test consistently is with high probability not random. One that fails rarely may or may not be truly random. One that always passes is with high probability not truly random, though it may take an impractically long time to adequately explore "always". So we can test for failed generators, but not for good ones.

We talked about two tests in some detail in the body of this note, based on the ideas that: 1) all possible byte values should occur about equally often and 2) pairs of bytes in sequence should be uncorrelated. These are relatively simple tests that are suggested by the physical process that we expect is going on in our generator. They characterize that process, which is believed to be fundamentally unpredictable, and they give these physicist writers a good feeling. They are also likely to catch a programing error should we make one. But they are not tests for randomness *per se*. Assurance of that has to come from knowledge of the underlying physics, and of the electronic and computational structure built atop that physics.

That said, there are some widely accepted tests that are thought to have relevance to cryptographic strength. A suite of them can be found at http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html. Any random generator of numbers that we want to take seriously ought to be able to pass them, at least most of the time (and that may be every time we have the patience to try). Again, as should be obvious from the fact that pseudo-random generators can also pass them, they are not tests for genuine randomness.

A more convenient implementation of some of these and other tests, ready to run on a Windows PC, can be found at http://www.phy.duke.edu/~rgb/General/dieharder.php

We have run the same 10^6-byte sequence from the MSP430F2013 that discussed above through the NIST test suite using the supplied default values of the test parameters. The sequence passed all tests. (We did not successfully explore "always".) The printout from those tests is included as the last appendix.

About Autocorrelation

Given two finite sequences x_i and y_i, for i=1...N, the means of the two sequences are $\mu_x =< x_i >= (1/N)\sum_{i=1}^{N} x_i$ and $\mu_y =< y_i >= (1/N)\sum_{i=1}^{N} y_i$. Here and below we denote an average over all values of i with the pointed brackets < >.

The covariance of the two sequences is defined as

$$C \equiv < (x_i - \mu_x)\cdot(y_i - \mu_y) >= < x_i \cdot y_i > - < x_i \cdot \mu_y > - < y_i \cdot \mu_x > + < \mu_x \cdot \mu_y >$$

Since the means are constant they may be factored out of the averages to yield a simpler result.

$$C =< x_i \cdot y_i > - < x_i > \cdot \mu_y - < y_i > \cdot \mu_x + \mu_x \cdot \mu_y =< x_i \cdot y_i > - \mu_x \cdot \mu_y - \mu_y \cdot \mu_x + \mu_x \cdot \mu_y$$

$$=< x_i \cdot y_i > - \mu_x \cdot \mu_y$$

The autocovariance of a sequence of values x_i at lag L is the covariance of that sequence with a copy of itself shifted by L places. That is, $C(L) =< x_i \cdot y_i > - \mu_x \cdot \mu_y$ where $y_i = x_{i-L}$ and we assume that the sequence is now of length at least N+L so that there are still N product terms to average over, and where the sums are still over the range i=1...N. C(0) is equal to the ordinary variance of the sequence.

The autocorrelation function is $r(L) = C(L)/C(0)$. The autocorrelation at zero lag, r(0), is always 1; any sequence is perfectly correlated with itself at zero lag. We are concerned that our sequences not be correlated with themselves at any lag larger than zero. That is, we want successive values to be independent.

When calculating the autocorrelation in a small processor "on the fly" (processing each value as it is generated without storing the entire sequence) we need to keep the sums $S(L) = \sum_{i=1}^{N} x_{i-L}$ and $SS(0,L) = \sum_{i=1}^{N} x_i \cdot x_{i-L}$ for L=0 and for all other values of L that are of interest. This is conveniently done by first populating a history array h_i that is as long as the largest lag to be calculated, then as each new value comes in

1) Copy each element of the history array into the next higher location; $h_{j+1} = h_j$ with values of j decreasing.{for(j=0;j<LMAX; j++) h[j+1]=h[j+1] }
2) Copy the new sequence value into $h_{LMAX;}$ {h[LMAX]=x}
3) Maintain the sums, reading the value of x_{i+L} from $h_{L..}$ { SS[L] +=h[0]*h[L]}.

When the entire record has been processed in this way, we calculate

$$C(L) = (SS(0,L) - (S(L)/\sqrt{N}) * (S(0)/\sqrt{N}))/N$$

$$r(L) = Scale * C(L)/C(0)$$

The sum of products can be large when processing a large file. A random million-byte file will generate a sum of squares with a value approximately $128^2 \times 10^6 \approx 2^{34}$ that will overflow a four-byte field. The sum cannot be accumulated to adequate precision either as a 32-bit integer or in single precision floating point.

A million-<u>word</u> sample will result in sums as large as $2^{16} * 10^6 \le 2^{16} * 2^{20} = 2^{36}$, and a sum of squares as large as $2^{32} * 10^6 \le 2^{32} * 2^{20} = 2^{52}$. The resulting covariance can be as large as 2^{32}. Autocorrelation calculations for 16-bit full-word data calculated on the fly as suggested above must do their sums in 64-bit **long long** variables in MSP430 C. Even with 64-bit variables, care is necessary (for instance breaking up one factor of N in the denominator for C(L) above into two factors of $\sqrt{N}$) in order to both maintain precision and to avoid overflow when doing what is effectively a fixed-point calculation with integers.

Handling these64-bit variables slows the calculation significantly, and one must check that a calculation done "on the fly" can keep up with the rate of data production. The requirement that data overrun not occur, and that no data is lost, is one limit to the number of lags that can be computed on one data set to something like six when running at OSR=256. One assumes that a floating point calculation would be much worse.

Program for the eZ430-F2013

The program below loaded into an eZ430-F2013 will send randomly generated bytes in start-stop serial format at 115.2 KBaud out through P1.1, which is pin 3 of the MSP430F2013 processor included in the eZ430-F2013. The oversampling ratio may be changed by altering a statement just below the comment line // SD_16. However notice the warning message just above that line.

The program works as described in the text only for OSR $\geq$ 128. For smaller oversampling ratios, characters are produced faster than they can be sent. The program will still run, but successive characters in the serial communication output will no longer represent successive characters in the generator output.

There are two separate files below, Main.c and Timer_A.c. They were compiled using IAR Embedded Workbench.

```
//File Main.c
// Random Number generator test, target is MSP430F2013,
#include <msp430f2013.h>
int send_byte(char);
unsigned char Datum;
volatile unsigned int v = 0;

int main( void )
{
// Disable watchdog timer
WDTCTL =  WDTPW + WDTHOLD;

// Set DCO to 16MHz, which is OK at 3.3V Vcc.
BCSCTL1 = CALBC1_16MHZ;
DCOCTL  = CALDCO_16MHZ;

// Configure the ports
//P1
P1OUT = 0;      // Outputs are low
P1DIR = 0xFF;   // All pins are output
P1SEL = BIT1;   // And Bit1 is timer output.
//P2
P2OUT = 0;
P2DIR = 0;
P2REN = 0xFF;

// Timer_A
// P1.1 = data output Pin 3 of DIP package. (LED is P1.0, pin 2)
TACCR0 = 139;        // = 16 MHz/115.2 KBaud
```

```
TACCTL0 = OUTMOD_1 + CCIE;   // Set on event, enable interrupt;
TACTL = TASSEL_2 + ID_0 + MC_1;

// This SHIFT definition is the SD_16 register position from which the
// LSBit of the data byte is taken. SHIFT <= 8. Ordinarily 0 or 8.
#define SHIFT 0
// It and the oversampling ratio set below define the case.
// OSR >= 128 assumed so bytes are sent faster than they are produced.

// SD_16
//1 MHz, (MCLK/16), turn reference generator on.
SD16CTL = SD16XDIV_2 + SD16DIV_0+ SD16SSEL_0 + SD16REFON;
// Oversampling ratio set, enable LSB access, no interrupts,
SD16CCTL0 = SD16OSR_256 + SD16LSBACC;
// Gain = 32, channel is 7 (shorted)
SD16INCTL0 = SD16GAIN_32 + SD16INCH_7;
// Now Go
SD16CCTL0 |= SD16SC;

//*************************************
v = SD16MEM0; // Clear any leftover data.
//*********************************************************************
__enable_interrupt();

// Event loop starts here
while(1)
{
// Wait for next value
while((SD16IFG & SD16CCTL0) == 0) v++;
// Shift during this assignment to move up the filter register.
 Datum = SD16MEM0>>SHIFT;
send_byte(Datum);
} // End the event loop.

} // end main
```

```
// File Timer_A.c
#include <msp430f2013.h>
// Set TACCTL0 to one of these to either set or clear P1.1 on next event.
// Temporary reversal for test
#define SET   OUTMOD_1 + CCIE
#define RESET OUTMOD_5 + CCIE
enum {idle = 0, start, bit0, bit1, bit2, bit3, bit4, bit5, bit6, bit7, stop};
unsigned int serial_out_state = idle;
unsigned char Out;

// This routine called from main to initiate character output
// Return 0 for character accepted, 1 for refused (overrun).
int send_byte(char Outchar)
{
  if(serial_out_state == idle)
  {
    Out = Outchar;
    serial_out_state = start;
    return(0);
  }
  return(1); //Overrun error return
}

#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0( void )
{
switch(serial_out_state)
{
case idle:
  {
    TACCTL0 = SET; // Set Idle line and wait for send_byte to be called.
    break;
  }

case start:
  {
    TACCTL0 = RESET; // Make start bit
    serial_out_state = bit0;
    break;
  }
// Send next bit of character.
case bit0:
case bit1:
case bit2:
case bit3:
case bit4:
case bit5:
case bit6:
```

```
case bit7:
  {
    if((Out&0x01) == 0) TACCTL0 = SET;   // if bit is 0, output is 1
    else              TACCTL0 = RESET; // and vice versa.
    Out=Out>>1;                        // shift to next bit
    serial_out_state++;                  // increment to next state.
    break;
  }
case stop:
  {
    TACCTL0 = SET; // Set Idle line and
    serial_out_state = idle;    // go to idle state.
    break;
  }
default:
  {
    serial_out_state = idle;
    break;
  }
} //End switch(serial_out_state)
} //End __interrupt void Timer_A0( void )
```

Results of The NIST STS suite

These tests were performed running the STS test suite with the default parameters supplied by NIST.

```
STS Run
06 December 2013
RKG
------------------------------------------------------------------------------
Notes:
1) The data file, ekstrom.raw, is a file of 10^6 random bytes captured from a CPU
configured to stream randomly generated numbers generated by the SD_16A in the
processor of the EZ430F2013, using the program given in the previous section.

2) The data file contains the bits in binary format.  Each byte of data in the file
contains 8 bits of randomly generated data
------------------------------------------------------------------------------
Tests:
01    Frequency Test: Monobit
02    Frequency Test: Block
03    Cumulative Sums Test
04    Runs Test
05    Test for the Longest Runs of Ones in a Block
06    Binary Matrix Rank Test
07    Discrete Fourier Transform (Spectral Test)
08    Non-Overlapping Template Matching Test
09    Overlapping Template Matching Test
10    Maurer's Universal Statistical Test
11    Approximate Entropy Test
12    Random Excursions Test
13    Random Excursions Variant Test
14    Serial Test
15    Linear Complexity Test
==============================================================================

                            FREQUENCY TEST
            ---------------------------------------------
            COMPUTATIONAL INFORMATION:
            ---------------------------------------------
            (a) The nth partial sum = -898
            (b) S_n/n               = -0.000898
            ---------------------------------------------
SUCCESS           p_value = 0.369186
==============================================================================

                             BLOCK FREQUENCY TEST
            ---------------------------------------------
            COMPUTATIONAL INFORMATION:
            ---------------------------------------------
            (a) Chi^2           = 7967.437500
            (b) # of substrings = 7812
            (c) block length    = 128
            (d) Note: 64 bits were discarded.
            ---------------------------------------------
SUCCESS           p_value = 0.107356
==============================================================================
```

```
                CUMULATIVE SUMS (FORWARD) TEST
              ----------------------------------------
              COMPUTATIONAL INFORMATION:
              ----------------------------------------
              (a) The maximum partial sum = 918
              ----------------------------------------
SUCCESS             p_value = 0.705472

                CUMULATIVE SUMS (REVERSE) TEST
              ----------------------------------------
              COMPUTATIONAL INFORMATION:
              ----------------------------------------
              (a) The maximum partial sum = 1494
              ----------------------------------------
SUCCESS             p_value = 0.270336
================================================================================

                       RUNS TEST
              ----------------------------------------
              COMPUTATIONAL INFORMATION:
              ----------------------------------------
              (a) Pi                          = 0.499551
              (b) V_n_obs (Total # of runs)   = 500177
              (c) V_n_obs - 2 n pi (1-pi)
                  ---------------------       = 0.250886
                    2 sqrt(2n) pi (1-pi)
              ----------------------------------------
SUCCESS             p_value = 0.722734
================================================================================

                  LONGEST RUNS OF ONES TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) N (# of substrings)  = 100
              (b) M (Substring Length) = 10000
              (c) Chi^2                = 3.894568
              ---------------------------------------------
                    F R E Q U E N C Y
              ---------------------------------------------
              <=10  11  12  13  14  15 >=16 P-value  Assignment
                 7  28  22  16  13   6    8
SUCCESS             p_value = 0.690942
================================================================================

                       RANK TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) Probability P_32 = 0.288788
              (b)             P_31 = 0.577576
              (c)             P_30 = 0.133636
              (d) Frequency   F_32 = 278
              (e)             F_31 = 593
              (f)             F_30 = 105
              (g) # of matrices    = 976
```

```
            (h) Chi^2            = 6.531766
            (i) NOTE: 576 BITS WERE DISCARDED.
            ------------------------------------------
SUCCESS           p_value = 0.038163
================================================================================

                     FFT TEST
            ------------------------------------------
            COMPUTATIONAL INFORMATION:
            ------------------------------------------
            (a) Percentile = 94.965800
            (b) N_l        = 474829.000000
            (c) N_o        = 475000.000000
            (d) d          = -1.569204
            ------------------------------------------
SUCCESS           p_value = 0.116601
================================================================================

              NONPERIODIC TEMPLATES TEST
--------------------------------------------------------------------------------
              COMPUTATIONAL INFORMATION
--------------------------------------------------------------------------------
      LAMBDA = 244.125000     M = 125000  N = 8 m = 9 n = 1000000
--------------------------------------------------------------------------------
            F R E Q U E N C Y
Template   W_1  W_2  W_3  W_4  W_5  W_6  W_7  W_8    Chi^2   P_value Assignment
Index
--------------------------------------------------------------------------------
000000001  233  237  232  250  240  239  236  267  4.188479 0.839730 SUCCESS   0
000000011  228  247  235  216  245  241  219  263  9.069123 0.336498 SUCCESS   1
000000101  241  238  271  234  222  221  265  273 13.412770 0.098415 SUCCESS   2
000000111  249  248  255  202  245  235  233  254  9.476903 0.303674 SUCCESS   3
000001001  248  242  253  239  223  243  249  280  7.977121 0.435708 SUCCESS   4
000001011  235  231  270  227  249  237  257  244  6.179714 0.627109 SUCCESS   5
000001101  252  232  227  255  233  255  248  221  5.983768 0.649050 SUCCESS   6
000001111  249  250  235  211  246  229  233  227  7.999364 0.433532 SUCCESS   7
000010001  225  225  262  254  265  250  251  244  7.058823 0.530300 SUCCESS   8
000010011  257  221  238  250  235  279  248  247  8.877414 0.352743 SUCCESS   9
000010101  229  253  229  253  240  256  256  256  4.470217 0.812405 SUCCESS  10
000010111  229  236  264  249  255  256  241  236  4.442679 0.815140 SUCCESS  11
000011001  259  266  253  256  246  243  238  279  9.227998 0.323429 SUCCESS  12
000011011  248  253  214  258  240  233  254  226  7.459188 0.487994 SUCCESS  13
000011101  267  238  218  229  227  233  232  244  8.626391 0.374786 SUCCESS  14
000011111  239  235  225  222  243  208  234  209 15.283260 0.053866 SUCCESS  15
000100011  241  238  245  246  235  236  259  245  1.791582 0.986748 SUCCESS  16
000100101  251  264  258  280  244  221  257  252 11.372813 0.181456 SUCCESS  17
000100111  252  225  231  221  253  272  251  277 13.212587 0.104741 SUCCESS  18
000101001  249  233  279  248  235  228  223  254  9.599766 0.294248 SUCCESS  19
000101011  241  276  236  230  249  237  259  246  6.738954 0.565042 SUCCESS  20
000101101  223  266  274  247  262  239  255  253 10.034025 0.262646 SUCCESS  21
000101111  249  236  237  220  249  253  237  240  3.782817 0.876168 SUCCESS  22
000110011  245  269  247  262  233  246  236  243  4.837748 0.774769 SUCCESS  23
000110101  234  250  250  273  256  266  256  243  7.486727 0.485140 SUCCESS  24
000110111  228  229  226  245  250  243  267  229  6.803564 0.557968 SUCCESS  25
000111001  257  219  262  259  227  238  265  235  9.268247 0.320175 SUCCESS  26
000111011  268  236  231  240  240  250  229  235  5.036872 0.753628 SUCCESS  27
000111101  247  243  275  224  247  247  272  269 11.778474 0.161365 SUCCESS  28
```

000111111	239	256	233	241	234	208	237	228	8.554368	0.381270	SUCCESS	29
001000011	266	274	238	218	243	236	253	245	9.481139	0.303345	SUCCESS	30
001000101	219	227	241	251	258	279	232	242	10.769087	0.215129	SUCCESS	31
001000111	238	235	271	245	222	232	251	256	7.069415	0.529162	SUCCESS	32
001001011	265	249	243	280	235	237	249	248	8.137056	0.420198	SUCCESS	33
001001101	224	220	241	245	262	258	220	237	9.076537	0.335881	SUCCESS	34
001001111	243	206	236	254	255	259	258	271	12.170366	0.143763	SUCCESS	35
001010011	249	242	264	253	240	206	228	249	9.559518	0.297312	SUCCESS	36
001010101	243	224	220	243	217	224	231	260	10.823105	0.211925	SUCCESS	37
001010111	243	259	232	225	249	259	239	225	5.814301	0.668024	SUCCESS	38
001011011	233	281	233	239	240	237	249	239	7.419999	0.492068	SUCCESS	39
001011101	230	260	254	226	240	239	258	238	4.875878	0.770756	SUCCESS	40
001011111	248	226	247	240	247	247	257	247	2.369888	0.967522	SUCCESS	41
001100101	268	251	247	229	234	259	230	277	10.415325	0.237076	SUCCESS	42
001100111	238	246	250	251	236	247	219	239	3.620765	0.889618	SUCCESS	43
001101011	237	258	239	264	261	243	246	255	4.543300	0.805086	SUCCESS	44
001101101	252	238	236	256	234	222	249	221	6.173359	0.627820	SUCCESS	45
001101111	234	251	220	233	249	253	241	242	4.119633	0.846172	SUCCESS	46
001110101	234	251	227	226	253	218	240	252	6.828984	0.555192	SUCCESS	47
001110111	255	233	234	241	242	237	223	207	9.465252	0.304579	SUCCESS	48
001111011	268	251	273	207	236	252	242	269	15.169929	0.055924	SUCCESS	49
001111101	240	241	233	252	227	253	235	236	3.109187	0.927320	SUCCESS	50
001111111	263	245	215	245	228	212	235	247	10.971388	0.203326	SUCCESS	51
010000011	248	243	224	258	219	251	248	222	7.612768	0.472180	SUCCESS	52
010000111	244	238	238	215	246	227	244	216	8.520474	0.384346	SUCCESS	53
010001011	231	240	249	236	250	262	252	247	2.979968	0.935608	SUCCESS	54
010001111	214	248	263	259	245	223	259	261	10.393083	0.238513	SUCCESS	55
010010011	250	199	259	242	254	220	222	251	14.882895	0.061463	SUCCESS	56
010010111	249	262	223	266	216	242	239	267	11.070950	0.197711	SUCCESS	57
010011011	233	247	238	261	263	237	235	243	4.007361	0.856459	SUCCESS	58
010011111	227	245	231	264	250	248	251	277	8.638042	0.373744	SUCCESS	59
010100011	253	224	246	245	258	242	267	254	5.532562	0.699430	SUCCESS	60
010100111	235	264	264	258	270	219	238	262	11.539102	0.172983	SUCCESS	61
010101011	235	241	229	251	226	229	245	254	4.340999	0.825122	SUCCESS	62
010101111	227	249	245	241	242	253	251	231	2.670691	0.953296	SUCCESS	63
010110011	250	261	222	245	251	243	244	261	4.841985	0.774324	SUCCESS	64
010110111	216	253	259	221	246	234	271	237	10.612331	0.224647	SUCCESS	65
010111011	252	274	246	231	257	255	250	261	7.344798	0.499929	SUCCESS	66
010111111	226	233	240	255	258	263	255	237	5.530444	0.699665	SUCCESS	67
011000111	252	259	230	237	230	244	256	226	5.095126	0.747362	SUCCESS	68
011001111	250	244	240	243	215	226	210	222	12.216970	0.141783	SUCCESS	69
011010111	257	233	243	232	263	272	279	220	14.274932	0.074875	SUCCESS	70
011011111	247	263	218	259	260	228	271	238	10.761673	0.215572	SUCCESS	71
011101111	271	253	240	242	225	247	247	226	6.496405	0.591806	SUCCESS	72
011111111	245	250	206	243	259	244	242	218	10.161125	0.253899	SUCCESS	73
100000000	233	237	232	250	240	239	236	267	4.188479	0.839730	SUCCESS	74
100010000	258	247	248	242	251	236	256	262	3.364446	0.909452	SUCCESS	75
100100000	237	241	252	250	226	253	251	246	2.606082	0.956600	SUCCESS	76
100101000	243	248	246	224	244	238	265	269	6.426500	0.599571	SUCCESS	77
100110000	218	245	242	230	241	236	253	255	4.915068	0.766614	SUCCESS	78
100111000	236	233	268	254	246	236	250	254	4.486105	0.810822	SUCCESS	79
101000000	232	238	241	213	235	241	230	239	6.278216	0.616099	SUCCESS	80
101000100	272	239	227	267	228	259	231	261	10.837933	0.211052	SUCCESS	81
101001000	249	240	259	239	240	259	252	231	3.223577	0.919553	SUCCESS	82
101001100	240	261	258	222	227	249	233	242	6.054732	0.641101	SUCCESS	83
101010000	257	230	221	246	219	247	247	230	7.417881	0.492289	SUCCESS	84
101010100	277	231	210	239	241	261	202	250	19.265730	0.013501	SUCCESS	85

101011000	210	232	255	264	246	246	211	242	12.428803	0.133073	SUCCESS	86
101011100	272	261	242	219	255	243	238	235	8.210138	0.413217	SUCCESS	87
101100000	254	264	271	257	247	232	248	264	8.244032	0.410003	SUCCESS	88
101100100	239	228	249	250	233	254	246	246	2.427083	0.965044	SUCCESS	89
101101000	250	261	253	250	258	238	237	216	6.373541	0.605466	SUCCESS	90
101101100	241	265	235	207	231	247	242	250	9.009810	0.341469	SUCCESS	91
101110000	235	230	241	249	237	253	249	244	1.989646	0.981328	SUCCESS	92
101110100	215	240	244	217	236	225	248	211	13.324859	0.101150	SUCCESS	93
101111000	235	259	235	234	228	241	243	259	4.163059	0.842120	SUCCESS	94
101111100	262	239	257	238	226	212	284	258	15.642318	0.047795	SUCCESS	95
110000000	239	240	252	232	249	236	231	284	8.915544	0.349471	SUCCESS	96
110000010	229	223	250	226	252	276	257	233	10.191841	0.251819	SUCCESS	97
110000100	221	229	248	275	223	270	259	228	14.103347	0.079111	SUCCESS	98
110001000	241	239	233	239	235	216	262	249	5.946697	0.653203	SUCCESS	99
110001010	261	238	254	237	244	240	240	245	2.141107	0.976377	SUCCESS	100
110010000	258	267	253	245	237	232	256	254	5.217990	0.734043	SUCCESS	101
110010010	259	221	241	243	236	260	244	253	4.930955	0.764930	SUCCESS	102
110010100	240	254	235	239	240	223	254	282	9.402761	0.309466	SUCCESS	103
110011000	231	231	246	229	262	258	246	244	4.628033	0.796491	SUCCESS	104
110011010	256	227	218	252	245	252	230	261	7.311964	0.503378	SUCCESS	105
110100000	237	241	234	233	225	246	241	213	6.925368	0.544706	SUCCESS	106
110100010	250	247	232	263	240	228	221	263	7.262183	0.508627	SUCCESS	107
110100100	264	226	230	239	252	225	264	234	7.942169	0.439140	SUCCESS	108
110101000	263	248	234	239	234	249	252	243	2.921714	0.939174	SUCCESS	109
110101010	269	248	219	247	251	266	215	244	11.216056	0.189755	SUCCESS	110
110101100	202	237	259	267	216	222	221	244	18.578331	0.017285	SUCCESS	111
110110000	244	256	261	249	232	247	253	235	3.248997	0.917774	SUCCESS	112
110110010	234	269	224	241	218	268	243	255	10.626100	0.223798	SUCCESS	113
110110100	259	258	243	261	241	214	214	227	11.938408	0.153973	SUCCESS	114
110111000	218	231	252	236	266	252	235	234	7.241000	0.510868	SUCCESS	115
110111010	214	236	249	220	252	229	236	207	14.041915	0.080679	SUCCESS	116
110111100	233	235	232	223	227	247	253	237	5.216930	0.734159	SUCCESS	117
111000000	259	235	250	219	250	235	235	274	8.743959	0.364354	SUCCESS	118
111000010	218	271	253	274	230	229	280	233	17.858097	0.022316	SUCCESS	119
111000100	248	228	245	220	245	215	265	238	9.236472	0.322742	SUCCESS	120
111000110	243	221	247	225	226	272	245	243	8.548013	0.381846	SUCCESS	121
111001000	250	276	235	218	236	254	242	265	10.253273	0.247698	SUCCESS	122
111001010	249	223	223	218	258	228	247	260	9.793594	0.279813	SUCCESS	123
111001100	215	225	239	225	264	258	249	248	9.457838	0.305156	SUCCESS	124
111010000	261	222	240	220	235	226	261	231	9.499145	0.301952	SUCCESS	125
111010010	229	220	241	222	233	223	253	222	10.372959	0.239818	SUCCESS	126
111010100	232	244	238	253	266	222	287	263	14.514304	0.069307	SUCCESS	127
111010110	214	245	243	248	226	255	228	220	9.377341	0.311470	SUCCESS	128
111011000	260	247	257	245	245	260	248	255	3.443884	0.903501	SUCCESS	129
111011010	258	271	226	244	219	249	213	235	12.499767	0.130259	SUCCESS	130
111011100	234	257	247	240	258	269	249	231	5.511379	0.701779	SUCCESS	131
111100000	261	250	225	209	264	229	256	267	13.586473	0.093201	SUCCESS	132
111100010	246	242	247	223	247	219	258	243	5.490196	0.704126	SUCCESS	133
111100100	251	246	254	219	218	246	221	260	9.542571	0.298609	SUCCESS	134
111100110	253	233	215	253	242	238	248	257	5.729568	0.677495	SUCCESS	135
111101000	246	227	231	264	241	235	236	240	4.406667	0.818697	SUCCESS	136
111101010	240	241	243	249	275	248	265	264	7.841548	0.449099	SUCCESS	137
111101100	245	240	235	261	241	277	242	253	6.607618	0.579497	SUCCESS	138
111101110	245	248	256	230	240	262	254	211	7.997246	0.433739	SUCCESS	139
111110000	270	250	239	280	251	225	270	241	13.174457	0.105986	SUCCESS	140
111110010	255	241	232	217	230	239	239	250	5.496551	0.703422	SUCCESS	141
111110100	240	232	220	250	234	224	255	227	7.200751	0.515136	SUCCESS	142

```
111110110  230  250  230  257  232  254  248  244  3.638771 0.888157 SUCCESS 143
111111000  251  260  229  244  240  211  241  214 10.844288 0.210679 SUCCESS 144
111111010  256  233  220  259  254  238  254  223  7.400934 0.494056 SUCCESS 145
111111100  272  244  228  243  260  227  224  229  9.394288 0.310133 SUCCESS 146
111111110  245  250  206  243  259  244  242  218 10.161125 0.253899 SUCCESS 147
================================================================================

            OVERLAPPING TEMPLATE OF ALL ONES TEST
           ---------------------------------------------
           COMPUTATIONAL INFORMATION:
           ---------------------------------------------
           (a) n (sequence_length)      = 1000000
           (b) m (block length of 1s)   = 9
           (c) M (length of substring)  = 1032
           (d) N (number of substrings) = 968
           (e) lambda [(M-m+1)/2^m]     = 2.000000
           (f) eta                      = 1.000000
           ---------------------------------------------
              F R E Q U E N C Y
             0   1   2   3   4 >=5   Chi^2   P-value  Assignment
           ---------------------------------------------
           359 194 137  97  60 121  4.106682 0.534161 SUCCESS
================================================================================


                     UNIVERSAL STATISTICAL TEST
           ---------------------------------------------
           COMPUTATIONAL INFORMATION:
           ---------------------------------------------
           (a) L         = 7
           (b) Q         = 1280
           (c) K         = 141577
           (d) sum       = 877347.519730
           (e) sigma     = 0.002768
           (f) variance  = 3.125000
           (g) exp_value = 6.196251
           (h) phi       = 6.196964
           (i) WARNING:  1 bits were discarded.
           ---------------------------------------------
SUCCESS          p_value = 0.796777
================================================================================

                 APPROXIMATE ENTROPY TEST
           ---------------------------------------------
           COMPUTATIONAL INFORMATION:
           ---------------------------------------------
           (a) m (block length)    = 10
           (b) n (sequence length) = 1000000
           (c) Chi^2               = 1083.704253
           (d) Phi(m)         = -6.930968
           (e) Phi(m+1)             = -7.623573
           (f) ApEn                = 0.692605
           (g) Log(2)              = 0.693147
           ---------------------------------------------
SUCCESS          p_value = 0.095242
================================================================================
```

```
                RANDOM EXCURSIONS TEST
            -----------------------------------------
            COMPUTATIONAL INFORMATION:
            -----------------------------------------
            (a) Number Of Cycles (J) = 1730
            (b) Sequence Length (n)  = 1000000
            (c) Rejection Constraint = 500.000000
            -----------------------------------------
SUCCESS           x = -4 chi^2 =  2.121795 p_value = 0.832049
SUCCESS           x = -3 chi^2 =  4.219629 p_value = 0.518247
SUCCESS           x = -2 chi^2 =  7.092899 p_value = 0.213822
SUCCESS           x = -1 chi^2 = 11.611561 p_value = 0.040516
SUCCESS           x =  1 chi^2 =  8.154913 p_value = 0.147902
SUCCESS           x =  2 chi^2 =  0.794205 p_value = 0.977401
SUCCESS           x =  3 chi^2 =  2.269508 p_value = 0.810735
SUCCESS           x =  4 chi^2 =  2.720236 p_value = 0.743022
================================================================================

                RANDOM EXCURSIONS VARIANT TEST
            -----------------------------------------
            COMPUTATIONAL INFORMATION:
            -----------------------------------------
            (a) Number Of Cycles (J) = 1730
            (b) Sequence Length (n)  = 1000000
            -----------------------------------------
SUCCESS           (x = -9) Total visits = 1823; p-value = 0.701378
SUCCESS           (x = -8) Total visits = 1750; p-value = 0.930043
SUCCESS           (x = -7) Total visits = 1755; p-value = 0.906165
SUCCESS           (x = -6) Total visits = 1789; p-value = 0.762328
SUCCESS           (x = -5) Total visits = 1759; p-value = 0.869465
SUCCESS           (x = -4) Total visits = 1688; p-value = 0.787257
SUCCESS           (x = -3) Total visits = 1722; p-value = 0.951500
SUCCESS           (x = -2) Total visits = 1873; p-value = 0.160444
SUCCESS           (x = -1) Total visits = 1833; p-value = 0.079937
SUCCESS           (x =  1) Total visits = 1839; p-value = 0.063874
SUCCESS           (x =  2) Total visits = 1854; p-value = 0.223570
SUCCESS           (x =  3) Total visits = 1760; p-value = 0.819580
SUCCESS           (x =  4) Total visits = 1749; p-value = 0.902831
SUCCESS           (x =  5) Total visits = 1733; p-value = 0.986436
SUCCESS           (x =  6) Total visits = 1737; p-value = 0.971377
SUCCESS           (x =  7) Total visits = 1726; p-value = 0.984952
SUCCESS           (x =  8) Total visits = 1798; p-value = 0.765332
SUCCESS           (x =  9) Total visits = 1967; p-value = 0.328467
================================================================================

                   SERIAL TEST
            -----------------------------------------
             COMPUTATIONAL INFORMATION:
            -----------------------------------------
            (a) Block length    (m) = 16
            (b) Sequence length (n) = 1000000
            (c) Psi_m               = 65563.586560
            (d) Psi_m-1             = 32968.732672
            (e) Psi_m-2             = 16480.137216
            (f) Del_1               = 32594.853888
            (g) Del_2               = 16106.258432
            -----------------------------------------
```

```
SUCCESS           p_value1 = 0.750144
SUCCESS           p_value2 = 0.938145
================================================================================

-----------------------------------------------------
      L I N E A R  C O M P L E X I T Y
-----------------------------------------------------
      M (substring length)     = 500
      N (number of substrings) = 2000
-----------------------------------------------------
        F R E Q U E N C Y
-----------------------------------------------------
  C0   C1   C2   C3   C4   C5   C6    CHI2    P-value
-----------------------------------------------------
      Note: 0 bits were discarded!
  21   62  281  980  505  114   37  5.788698 0.447272
```