UNIVERSITY of WASHINGTON | BOTHELL

PSoC 5 LP Random Number Generators

Matthew A. Hicks

Abstract Random number generators (RNGs) have many practical applications including gaming, computer simulations, communication systems, and cryptography to name a few. Many of those applications are carried out within embedded systems. The following paper documents a project that evaluates several well-known and several experimental methods of generating random numbers in Cypress Semiconductor's Programmable System on Chip (PSoC) 5 as a standalone, embedded product. In addition, an analysis of these methods is conducted to test and evaluate randomness.

***Index Terms*—PSoC, Random Number Generator, White Noise, Linear Feedback Shift Register, Programmable Gain Amplifier, Analog to Digital Converter, Waveform Generator, Linear Congruential Generator, C99, Pseudo-Random Sequence**

I. INTRODUCTION

RANDOM number generators (RNGs) have many applications ranging from gambling and gaming, to communication systems and cryptography, all of which with their own constraints and objectives. When selecting an RNG for an application, there are two main categories to choose from, pseudo-random number generators (PRNGs) and true random number generators (TRNGs). The difference between the two is how the number or sequence of numbers is generated. Pseudo is defined as being "not genuine", while true, is defined as being "in accordance with fact or reality." So, PRNGs are not genuinely random, but rather appear to be random, while TRNGs are based upon some source that people have deemed to be *truly* random. One common benchmark used to differentiate PRNGs from TRNGs is that PRNGs are said to produce periodic, predictable sequences of numbers, where TRNGs do not [1].

Utilizing Cypress Semiconductor's Programmable System on Chip (PSoC), the following project seeks to implement common and experimental PRNGs and TRNGs. The ARM Cortex-M3 processor accompanied by analog and digital programmable coprocessors make the PSoC an excellent platform for experimentation.

II. PRNGs

The key word in PRNG, *pseudo*, comes from the way in which the numbers are generated. Often times a number or sequence of numbers from a PRNG is generated from a mathematical formula or calculated from a table. Because these formulas and tables are derived by people, the PRNGs associated with them have a known performance. Meaning, if given a starting point or initial condition of a PRNG, using the formula or table, one could calculate and predict numbers that would be produced by the generator. One common software implementation of a PRNG is a linear congruential generator. Many of the major languages use this method or random number generation including Borland C/C++, C++11, Java, POSIX, and Pascal [2].

A hardware version of a PRNG is a linear feedback shift register (LFSR). Depending on the structure and "taps" on the LFSR, a seemingly random number can be generated. However, an LFSR will generate a periodic sequence of numbers if sampled at every step, thus categorizing it as a PRNG.

III. TRNGs

TRNGs, unlike PRNGs, are not based on manmade formulas or devices, are not periodic, and rely on some external phenomenon accepted as being *truly* random. There are many such

phenomena including white noise, thermal noise, atmospheric noise, radioactive decay, and entropy. Other methods have been developed that use semiconductor devices (PN junction avalanche break-down) and lava lamps.

As mentioned previously, the primary difference between TRNGs and PRNGs is that the former are not periodic and produce indeterminate numbers or sequences that cannot be calculated or predicted.

IV. PSoC

The PSoC 5 is a 32-bit ARM Cortex-M3 microcontroller with programmable analog and digital blocks. The analog blocks include operational amplifiers, programmable gain amplifiers, multiple types of analog to digital converters, comparators, and a waveform generator. The digital blocks form a programmable logic device (PLD). Between the microcontroller centric architecture, and programmable analog and digital blocks, the PSoC 5 makes a great platform to test complete system on chip RNG methods.

Additionally, the PSoC has on chip USB capabilities that make user interfacing to extract the randomly generated numbers easy. No additional hardware other than a micro-USB cable and a computer is needed. For this project commands have been generated for each random function and each method accepts a single parameter value indicating the desired quantity random numbers to be retrieved.

V. Commands & RNGs

The following are the functions passed over USB Serial emulation to get a certain quantity of random numbers:

1. getRandC99(int quantity)
2. getRandPRS(int quantity)
3. getRandADC(int quantity)
4. getRandPGA(int quantity)
5. getRandCus(int quantity)

A brief overview of the functions are provided and expanded on in subsequent sections. Function 1 uses the compiler's built in *rand()* function. Function 2 uses a built in digital block constructed by Cypress Semiconductor called a Pseudo-Random Sequence (PRS) block. The PRS is based on an LFSR. Function 3 is a theoretical TRNG based on assigning differential inputs on an ADC to floating pins on the PSoC. Function 4 is similar to the function 3 but instead uses a programmable gain amplifier, with a waveform DAC generator, and an ADC. Finally, function 5 is another custom PRNG using a LFSR as a base methodology to generate a number between 0-255, and using a real time clock (RTC) and system timer for a seed.

VI. C99

The PSoC is programmed using a customized C99 compiler. Within the standard library (stdlib.h) there is a function, *rand()*, that returns a pseudo random double value between 0 and 1. The PRNG used with the *rand()* function is generated from what is called a linear congruential generator (LCG). The LCG is a common method of generating random numbers in many major languages including: Java, C++11, Borland C/C++, and POSIX [2]. This function was included in the project as somewhat of a benchmark for randomness.

VII. Psuedo-Random Sequence

Within PSoC Creator, the development environment for PSoC, is a pre-programmed digital logic block abstracting an LFSR. Cypress calsl it a PRS or pseudo-random sequence generator.

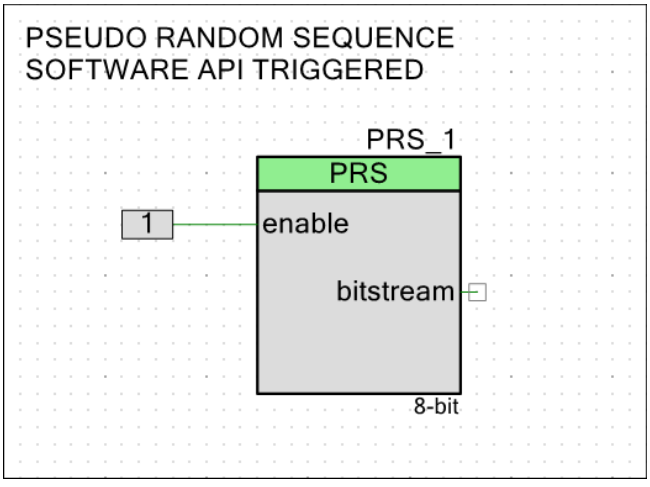

Figure 1: Image of Block Diagram for PRS

By dragging the block onto the top level design diagram (see Figure 1), one can begin to configure and utilize the PRS. First, the enable pin is set high by wiring a logic high constant. Then, the PRS is configured by double-clicking on the GUI icon for the device opening the configuration window. Once the configuration window opens (see Figure 2) and the PRS is configured.

The resolution range of the PRS is 2 to 64 bits, meaning it can generate anywhere from 2 to 2^{64} numbers before having its sequence repeat. For this particular project, an 8-bit resolution is used, so 2^8 or 256 (0-255) random numbers are generated.

The LFSR field in Figure 2 shows where the XOR gate taps are placed in relation to the D flip-flops, and the seed value dictates where in the sequence the PRS will start. The run mode determines how the PRS steps through its sequence, whether it will be done using a system clock or manually using an API command. Figure 2 shows the exact configuration and setup for the PRS implementation that's used in this project.

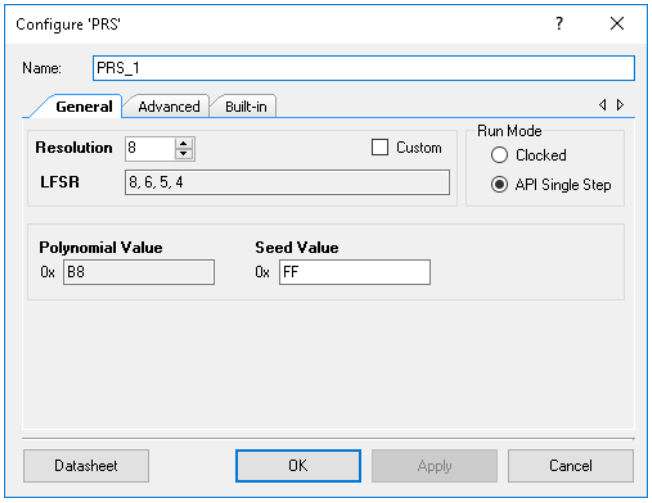

Figure 2: PRS Configuration Window

Lastly, the name field is important for API generation and reference. All API calls are prefixed by the name, or PRS_1. For example, to initialize the PRS, calling PRS_1_Start() is all that is needed. A sample of the API documentation provided by Cypress for the PRS is shown in Figure 3.

Function	Description
PRS_Start()	Initializes seed and polynomial registers provided from customizer. PRS computation starts on rising edge of input clock.
PRS_Stop()	Stops PRS computation.
PRS_Sleep()	Stops PRS computation and saves PRS configuration.
PRS_Wakeup()	Restores PRS configuration and starts PRS computation on rising edge of input clock.
PRS_Init()	Initializes seed and polynomial registers with initial values.

Figure 3: Sample PRS API Documentation

All components provided by Cypress have API documentation allowing for easy initialization and utilization of PSoC peripherals.

VIII. ADC TRNG

The first attempt at a TRNG is a simple implementation of an ADC on the PSoC and floating pins. The theory of operation is that a random number can be generated if the ADC is configured for a high resolution, and a variation in measurements on the ADC due to random noise on the pins exists at sampling.

Figure 4 shows in the top level design file how the TRNG is created. Essentially, using a differential input on the ADC, with a floating pin on the negative and positive inputs with the ADC configured to measure a differential of ±1.024V, the ADC would be able to measure a noise on the pins and be used as a TRNG.

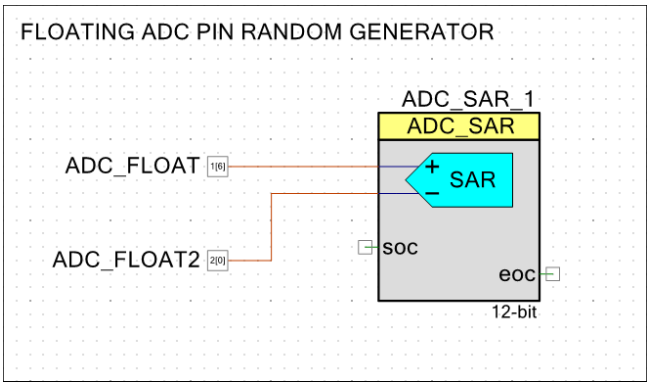

Figure 4: ADC TRNG Block Diagram

The configuration of the ADC is shown in Figure 5. The resolution of the ADC is configured for 12 bits in order to better detect changes in noise with the sample rate adjusted over several trials for best performance. Input range and reference settings are set up to measure two floating pins differentially, and lastly, the trigger mode is set to software trigger.

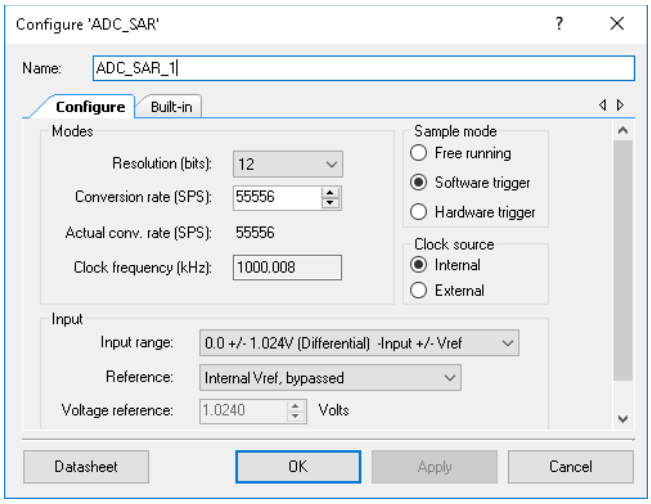

Figure 5: ADC TRNG Configuration

Much like the PRS, PSoC Creator generates an API based on the name of the block, ADC_SAR_1. So initializing the ADC is as easy as calling *ADC_SAR_1_Start()* and triggering an ADC measurement is as easy as calling *ADC_SAR_1_GetResult16()*, which returns a 16 bit value of the ADC result register.

IX. PGA TRNG

Like the ADC, a similar approach is taken for the PGA TRNG. However, rather than floating pins directly into a differential ADC, a waveform generator feeds into one input of the ADC, and a PGA feeds into the other input. A floating pin is then connected to the input of the PGA. The theory of operation is that the noise on the pin connected to the PGA is amplified, allowing for a greater differential on the ADC creating a larger variability for more random numbers. The waveform generator is continuously applying a periodic waveform on the other input of the ADC and creates a differential that, when sampled with a random noise, generates a random number. Figure 6 shows the block diagram as designed in the top level design file.

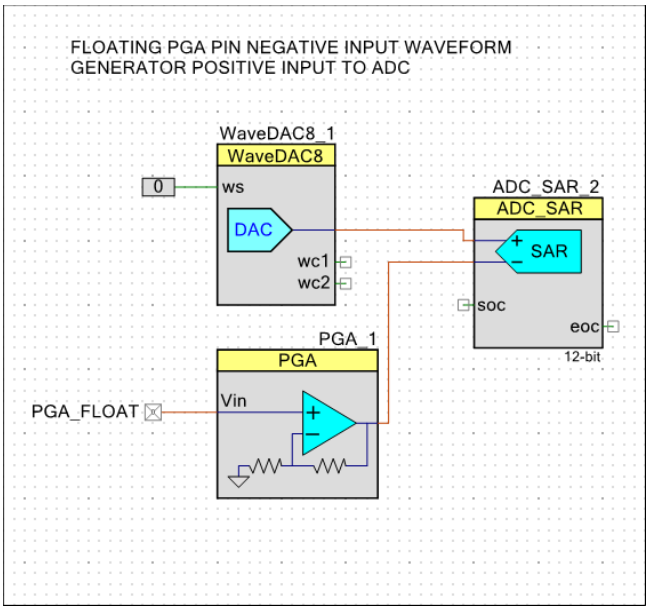

Figure 6: PGA TRNG Block Diagram

The WaveDAC8 is an interesting analog component in PSoC Creator that can generate various waveforms. For this TRNG application an "arbitrary draw" waveform is used as shown in the configuration window for the WaveDAC8 in Figure 7. Waveform 1 is used, but the generator can be configured for two different waveforms.

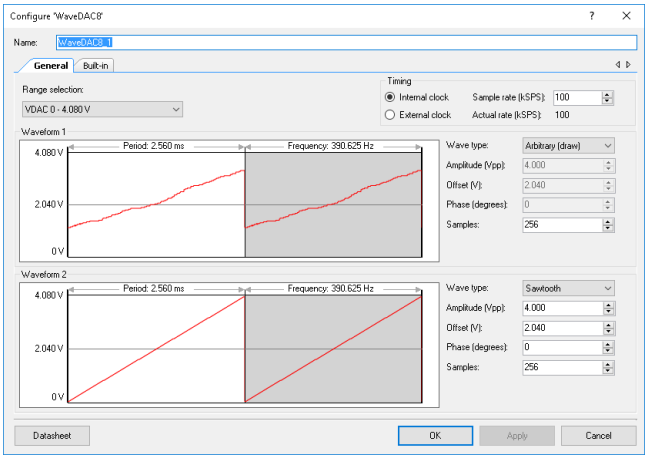

Figure 7: Configuration of WaveDAC8

The PGA portion is configured as shown in Figure 8, where it is set for high power with a gain of 2. The high power mode allows the amplifier to pass higher frequency noises. The gain was adjusted using trial and error for the best fit of random numbers generated.

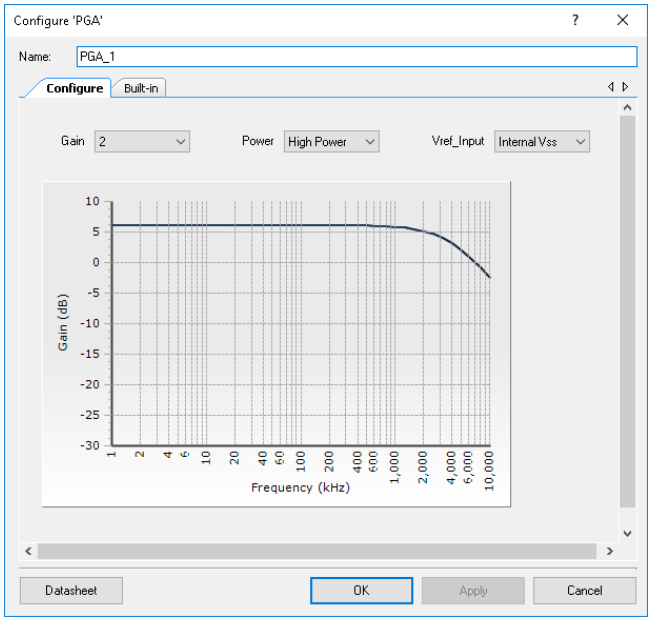

Figure 8: PGA Configuration

Lastly, the ADC is set up similarly to the single ADC TRNG. With differential inputs, the ADC is setup to measure the negative input ± Vdda. Figure 9 shows the setup for the ADC in this TRNG.

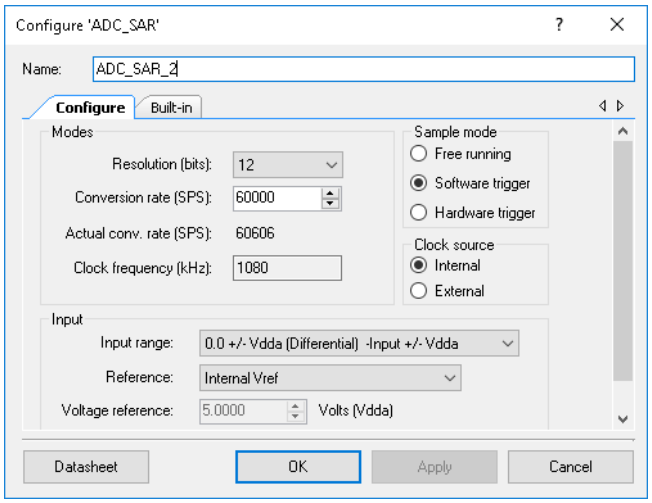

Figure 9: PGA TRNG Configuration

X. LFSR RTC Seed PRNG

The last and final PRNG is a custom LFSR implementation using the PSoC's digital logic blocks, and the on chip real time clock and a system timer for a seed. An overall system diagram is shown in Figure 10.

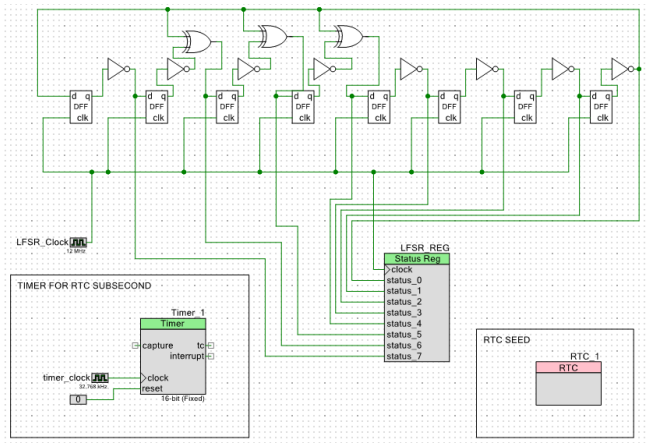

Figure 10: LFSR RTC Seed Block Diagram

The theory of operation for this PRNG is that an 8-bit LFSR is free-running in the background cycling through its periodic random number sequence using an on chip 12MHz clock (LFSR_Clock in Figure 10). The LFSR Status Register (LFSR_REG in Figure 10) is what is used to get the value from the LFSR. The RTC is used to get the system time, and the timer (Timer 1 in Figure 10) is set up to return microsecond resolution of the system time.

In the routine shown in Figure 11, a delay is made between status register calls to the LFSR based on the RTC hours, minutes, and seconds and the timer microseconds value. The delays are dependent on one another and are based on the modulus of the microsecond timer and the sum of all the RTC values (hours, minutes, and seconds).

```
void Get_Random_Cus(const int Random_Count){
    int i;
    for(i = 0; i < Random_Count; i++){
        int aInt = LFSR_REG_Read();
        uint16 mic = Timer_1_ReadCapture()%256;
        uint16 hour = (uint16)RTC_1_ReadHour();
        uint16 min = (uint16)RTC_1_ReadMinute();
        uint16 second = (uint16)RTC_1_ReadSecond();
        CyDelay(mic%256);
        CyDelay(hour+min+second);
        char str[10];
        sprintf(str, "%d", aInt);
        USBUART_1_PutString(str);
        while(!USBUART_1_CDCIsReady()){} /* Wait for Tx to finish */
        USBUART_1_PutCRLF();
        while(!USBUART_1_CDCIsReady()){} /* Wait for Tx to finish */
    }
}
```

Figure 11: Routine for Custom RNG

Stepping through the routine in Figure 11, as with all of the RNG routines, a parameter is passed into the function requesting a count of random numbers, and then a for loop iterates passing each randomly generated number through USB.

Timer_1_ReadCapture() is a Cypress generated API function associated with *Timer_1*. The timer is configured as shown in Figure 12. The purpose of the timer is to obtain sub-second resolution since the RTC only allows for second resolution.

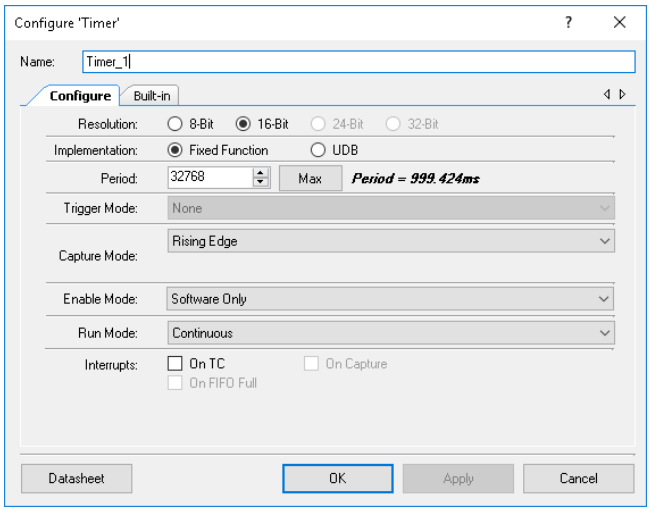

Figure 12: Sub-second Timer Configuration

The configuration in Figure 12 shows that the period of the timer is close to 1 second (999.424 ms) but not exact, and that the timer period is 32768 bits. Therefore, the resolution of the timer is:

$$\frac{0.999424\ s}{32768\ bits} = 30.5\frac{\mu s}{bit}$$

An interrupt is triggered every second for the RTC and the timer is reset to ensure it starts at zero every second.

XI. PSoC Communication

Communication between the PSoC and a user occurs through USB serial emulation, a USB protocol that emulates traditional serial ports. By dragging the USB block in PSoC Creator onto the top level design schematic, an API is created allowing for the use of the peripheral. After calling the initial startup functions for the USB API, a USB serial port now enumerates when plugging the PSoC board into a computer. In the main loop, shown in Figure 13, on the PSoC board the function *USBUART_1_GetCount()* retrieves the number of characters that are waiting in the USB buffer.

```
for(;;){
    Count = USBUART_1_GetCount();
    if(Count != 0) /* Check for input data from PC */
    {
        LED_Write(!LED_Read());
        USBUART_1_GetData(Buffer, Count);
        Process_Command(Buffer, Count); /* Echo data back to PC */
        while(!USBUART_1_CDCIsReady()){} /* Wait for Tx to finish */
    }
}
```

Figure 13: PSoC Program Main Loop

If one or more characters are waiting in the buffer, the function *USBUART_1_GetData(char[] receivedData, int receivedDataCount)* is called placing the data into a character array. The function *Process_Command(char[] receivedData, int receivedDataCount)* then handles the command from the PC. The functions that follow to parse the commands are lengthy, and consequently omitted, but the commands themselves to retrieve random numbers from the PSoC board are described in section **V. Commands and RNGs**.

A serial session in TeraTerm is one of the easiest ways to evaluate the commands and demonstrate functionality of the RNG board at the most fundamental level. As seen in Figure 14, the serial port enumerates as COM5, and setup just like any other serial device. For USB serial emulation the BAUD rate setting is irrelevant. Once the communication port is set up and the connection is established, commands can now be passed to the board.

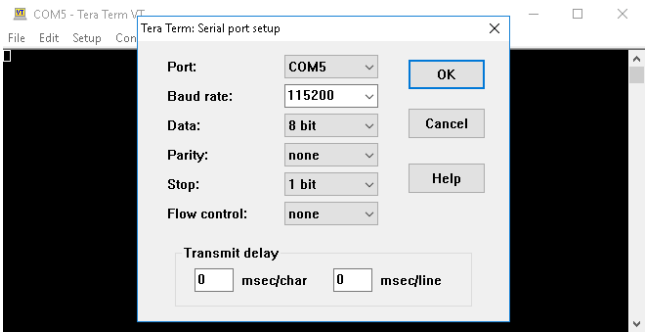

Figure 14: TeraTerm Serial Session with PSoC

In Figure 15, a command to retrieve ten random numbers generated with the C99 function is executed.

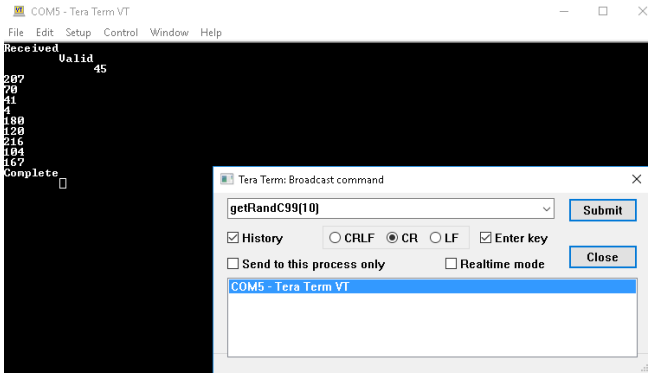

Figure 15: Command Verification

TeraTerm, and other serial monitoring software can be used to get random numbers from the RNG board, but a user interface has been created to more easily retrieve, evaluate, and test the random numbers that were generated.

XII. User Interface

The user interface was developed using the Java programming language, JavaFX GUI framework, and the Java Simple Serial Connector (jSSC) API. The purpose of the UI is to abstract the commands to retrieve random numbers, making the retrieval of random numbers more user friendly, eliminating the need to setup a serial session, and accurately, and tediously type commands. Figure 16 shows the user interface.

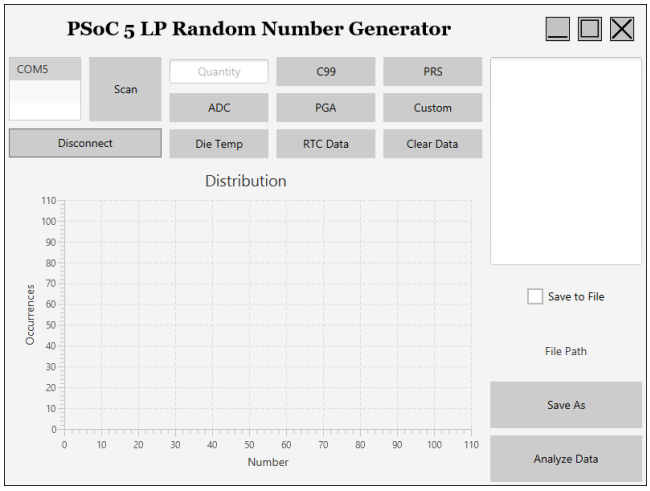

Figure 16: User Interface

The first aspect of the UI is serial connection. In the top left is a ListView that populates with available COM ports when the *Scan* button is pressed. If communication ports are found, they are enumerated in the ListView. From there, whichever COM port is selected within the ListView when the *Connect* or *Disconnect* button is pressed, the corresponded COM port that is selected is connected or disconnected.

Once connected, a variety of commands, functions, and abilities are enabled. The buttons correspond to random number generator command functions: *C99, PRS, ADC, PGA,* and *Custom* are abstracted. Additional features including getting the real time clock data, and die temp are included.

When a user fills in a value in the *Quantity* text field and clicks one of the five RNG buttons, values are streamed into the program and plotted in the user interface showing a distribution of the random numbers. This feature was added to quickly, and visually assess the series of random numbers created by each RNG for a uniform distribution.

For further analysis, an option to stream the data into a text file was added in addition to an *Analyze Data* feature to run tests on the data sets saved to a file. Currently, the only testing that was conducted to assess randomness was Chi-Squared testing (Goodness of Fit) and a couple tests from the Dieharder test suite.

XIII. Results & Analysis

In order to conserve space, all distributions and chi-squared testing screen shots from which the following results and analysis has been derived have been placed in appendices A & B.

From the law of large numbers what is expected from a very large sample of randomly generated numbers is a uniform distribution. For example, if one was to roll a six-sided die 6000 times, one might expect, given the probability of rolling a 1, 2, 3, etc., that 1000 one's, 1000 two's, 1000 three's and so on are witnessed. The same principle has been applied to the five RNGs under scrutiny – a uniform distribution is expected for large sets of data.

All RNGs generate a random number between 0 and 255 inclusive. For the distributions in Appendix A, ten thousand samples have been taken from each RNG. So, if each value has an equal probability of occurring, i.e. $1/256^{th}$, then from ten thousand samples, one would expect each number between 0 and 255 to occur about 39 times.

$$\frac{10{,}000\ samples}{256\ numbers} \approx 39\frac{samples}{number}$$

The uniform distribution won't be as pronounced since the law of large numbers usually requires millions, or billions of samples to play out, but still, a general idea is conveyed visually.

From the distribution drawings it can be seen that most every RNG, except the ADC, method generates a fairly uniform distribution. The ADC method appears to be the only RNG with issues. The built in *rand()* C99 standard library function does a good job, and so does the Cypress PRS digital block. Of the three RNGs that were created for this experiment, the PGA method looks good and so does the LFSR with RTC seed. The ADC method has an unusually large production of zeroes in the distribution triggering a red flag (see Appendix A for distributions).

Visual analysis of the distribution is a subjective method, and a method of analysis that is more objective, quantitative, and scientific has also been used to assess uniformity in the distribution. For this reason, the Pearson's Chi-Squared test exists. The chi-squared test is a statistical test applied to sets of categorical data to evaluate how likely it is that any observed difference between the sets arose by chance. The test consists of five steps:

1. Calculate X^2
2. Determine Degrees of freedom
3. Select desired confidence level
4. Compare X^2 to critical value from chi-squared distribution
5. Accept or reject null hypothesis

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} = N\sum_{i=1}^{n} \frac{(O_i/N - p_i)^2}{p_i}$$

where

χ^2 = Pearson's cumulative test statistic, which asymptotically approaches a χ^2 distribution.

O_i = the number of observations of type i.

N = total number of observations

$E_i = Np_i$ = the expected (theoretical) frequency of type i, asserted by the null hypothesis that the fraction of type i in the population is p_i

n = the number of cells in the table.

Figure 17: Formula for X^2 [3]

Figure 17 shows the formula for how to calculate X^2, while the degrees of freedom (DoF) are calculated by the possible outcomes *n*. In this analysis the possible outcomes are all the numbers between 0 and 255 inclusive, therefore, $n = 256$.

$$DoF = n - 1 \rightarrow DoF = 255$$

Step 3, selecting the desired confidence level, or confidence value, is related to the null hypothesis. A confidence value less than or equal to 0.05 is commonly selected as failing the null hypothesis. What the confidence value means, is that if a chi-squared test results in a confidence value of 0.05 or less, that there is a 5% chance or less that the data set resembles a chi-squared distribution, and the "goodness of fit" for the data set has not been proven. Thus, the data set should not be considered random.

For step 4, the confidence value is calculated from the chi-squared cumulative distribution function (CDF) with the following formula, with *k* being the DoF, and *x* being the X^2 value:

$$F(x;k) = \frac{\gamma(\frac{k}{2},\frac{x}{2})}{\Gamma(\frac{k}{2})}$$

Where the numerator of the function is the lower incomplete gamma function equal to:

$$\gamma(s,x) = \int_0^x t^{s-1}e^{-t}dt$$

And the denominator is the gamma function equal to:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$$

The total function F(x; k) can fortunately be simplified for easy implementation into software with the following simplification:

$$\frac{\gamma(\frac{n}{2},\frac{x}{2})}{\Gamma(\frac{n}{2})} = e^{-x}e_{n-1}(x)$$

Where $e_{n-1}(x)$ is the exponential function equal to:

$$e_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$$

The implementation for this math has been done in Java and is included in Appendix C. From the chi-squared testing, the following confidence values were calculated:

RNG	Confidence Value
C99	1.0
PRS	1.0
ADC	0.0
PGA	1.0
LFSR (Custom)	1.0

Table 1: RNG Confidence Levels

XIV. DIEHARDER TESTS

The Dieharder test suite is essentially a software wrapper for the GNU Scientific Library (GSL) random number tests and RNGs [4]. The Dieharder program has a couple dozen tests and subtests, but for many of the tests, a large quantity of samples is required in order to execute properly. These quantities can be in the millions and billions (See Table 2).

The PSoC RNG board developed for this experiment has several limitations preventing the entire Dieharder test from executing properly without a significant time commitment. The first, is that the UI program over USB takes some time to get random numbers from the board. The fastest generation method, the C99 function, can only supply 100,000 samples in about 15 seconds, and the slowest function, the LFSR with RTC seed can take several minutes for the same amount of samples. The slow speeds make it difficult to collect data and run tests requiring large quantities of samples.

Additionally, the Dieharder test requires 32-bit random numbers to be generated, so every four random numbers generated by the PSoC are multiplied to get 32-bits in the UI program. Therefore, it takes four times as long to generate numbers. Table 2 shows the minimum number of samples for each test to run properly. Tests 10, the Diehard Parking Lot test, and test 11, the Diehard Minimum Distance are the only two tests used to verify the RNGs. The parking lot test is described as follows:

This tests the distribution of attempts to randomly park a square car of length 1 on a 100x100 parking lot without crashing. We plot n (number of attempts) versus k (number of attempts that didn't "crash" because the car squares overlapped and compare to the expected result from a perfectly random set of parking coordinates. This is, alas, not really known on theoretical grounds so instead we compare to n=12,000 where k should average 3523 with sigma 21.9 and is very close to normally distributed. Thus (k-3523)/21.9 is a standard # normal variable, which converted to a uniform p-value, provides input to a KS test with a default 100 samples[5].

With the minimum distance test described as follows:

It does this 100 times:: choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the (n^2-n)/2 pairs of points. If the points are truly independent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995 . Thus 1-exp(-d^2/.995) should be uniform on [0,1) and a KSTEST on the resulting 100 values serves as a test of uniformity for random points in the square. Test numbers=0 mod 5 are printed but the KSTEST is based on the full set of 100 random choices of 8000 points in the 10000x10000 square[5].

Test results for the Dieharder tests can be found in Appendix D.

Test Number	Test Name	Data needed by default parameters (bytes)	Test Reliability
0	Diehard Birthdays Test	5,120,000	Good
1	Diehard OPERM5 Test	100,000,000	Good
2	Diehard 32 x 32 Binary Rank Test	512,000,000,000	Good
3	Diehard 6x8 Binary Rank Test	240,000,000	Good
4	Diehard Bitstream Test	209,715,200	Good
5	Diehard OPSO	209,715,200	Suspect
6	Diehard OQSO Test	209,715,200	Suspect
7	Diehard DNA Test	209,715,200	Suspect
8	Diehard Count the 1s (stream) Test	25,600,000	Good
9	Diehard Count the 1s Test (byte)	102,400,000	Good
10	Diehard Parking Lot Test	2,400,000	Good
11	Diehard Minimum Distance (2d Circle) Test	3,200,000	Good
12	Diehard 3d Sphere (Minimum Distance) Test	48,000,000	Good
13	Diehard Squeeze Test	40,000,000 <=n <= 1,920,000,000	Good
14	Diehard Sums Test	100,000	Do not use
15	Diehard Runs Test	40,000,000	Good
16	Diehard Craps Test	80,000,000 <= n <= 1,680,000,000	Good
17	Marsaglia and Tsang GCD Test	4,000,000,000	Good
100	STS Monobit Test	40,000,000	Good
101	STS Runs Test	40,000,000	Good
102	STS Serial Test (Generalized)	1,200,000,000	Good
200	RGB Bit Distribution Test	480,000,000	Good
201	RGB Generalized Minimum Distance Test	560,000,000	Good
202	RGB Permutations Test	560,000,000	Good
203	RGB Lagged Sum Test	400,000,000	Good
204	RGB Kolmogorov-Smirnov Test Test	40,000,000	Good
205	Byte Distribution	51,200,000	Good
206	DAB DCT	51,200,000	Good
207	DAB Fill Tree Test	640,000,000	Good
208	DAB Fill Tree 2 Test		Good
209	DAB Monobit 2 Test		Good

Table 2: Dieharder Tests with Samples [5]

XV. CONCLUSION

In conclusion, the null hypothesis is accepted for the following RNGs:

- C99
- PRS
- PGA
- LFSR RTC Seed

and rejected for:

- ADC

Thus, two of the predefined PRNGs, the built in C99 *rand()* function and the Cypress provided PRS block were proven to be random, and two of the three experimental RNGs, the PGA (TRNG) and the LFSR RTC seed (PRNG) were proven to be random. Only the ADC TRNG was evaluated and proven to be incapable of producing random numbers and/or sequences.

APPENDIX B: DISTRIBUTIONS

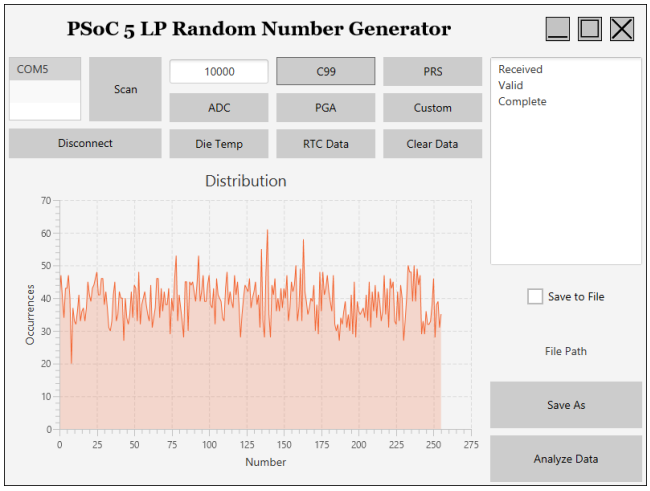

Figure A.1: C99 Distribution

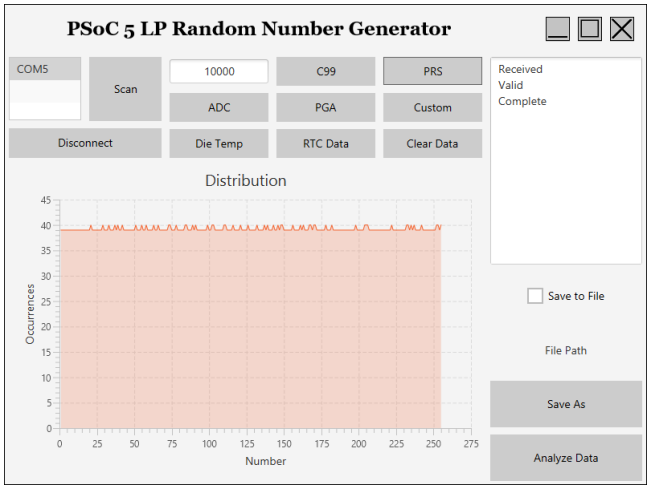

Figure A.2: PRS Distribution

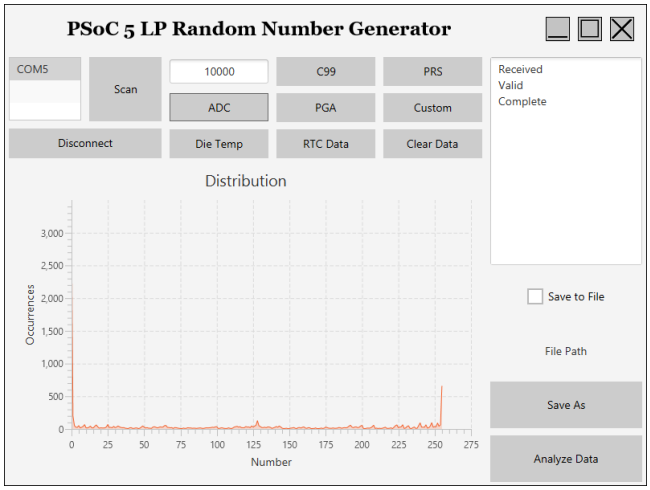

Figure A.3: ADC Distribution

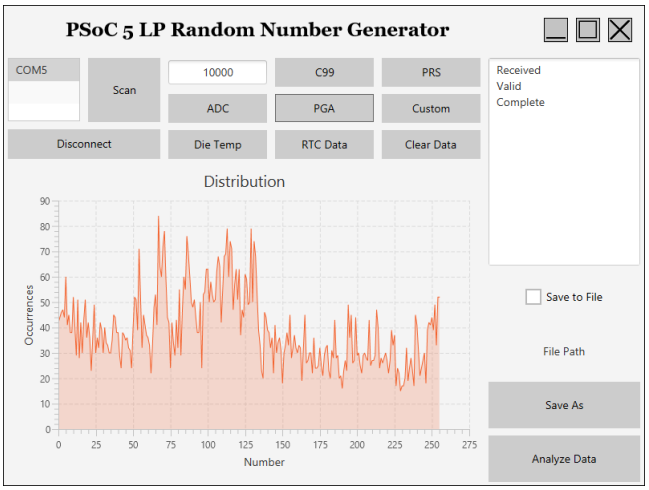

Figure A.4: PGA Distribution

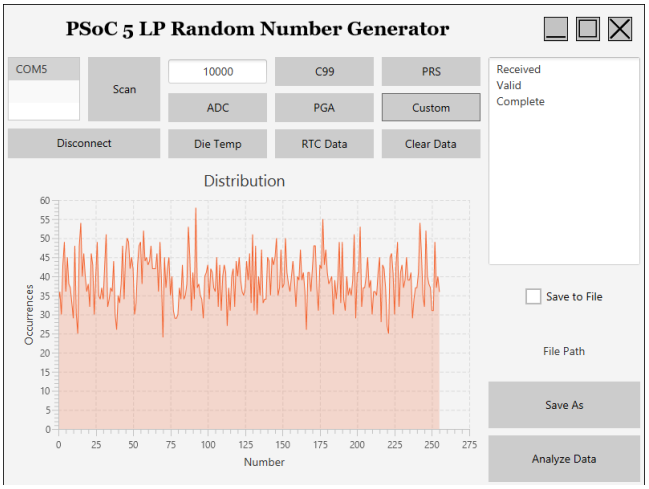

Figure A.5: LFSR RTC Seed Distribution

APPENDIX B: CHI-SQUARED TESTS

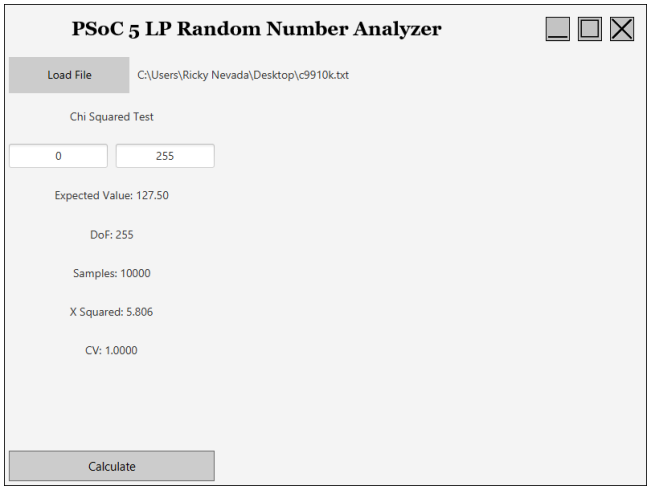

Figure B.1: C99 Results, P=1.0

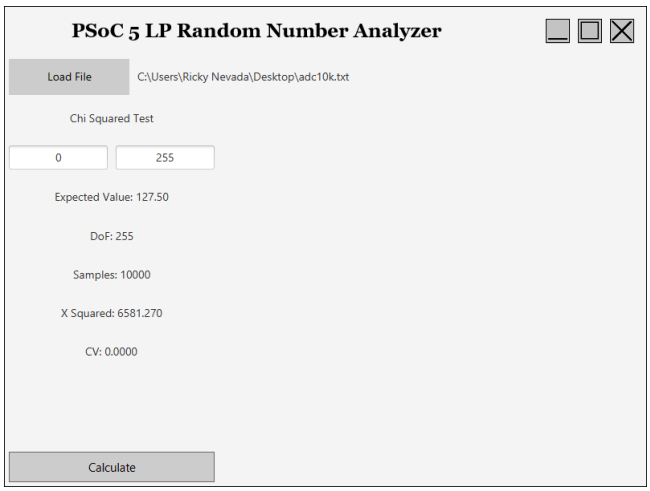

Figure B.2: ADC Results, P=0.0

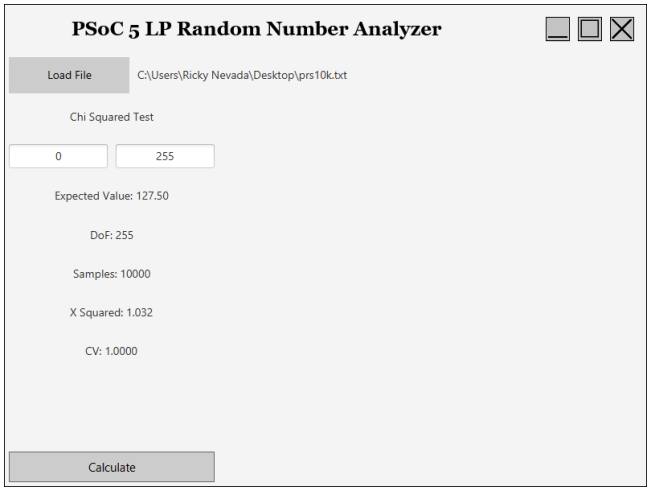

Figure B.3: PRS Results, P=1.0

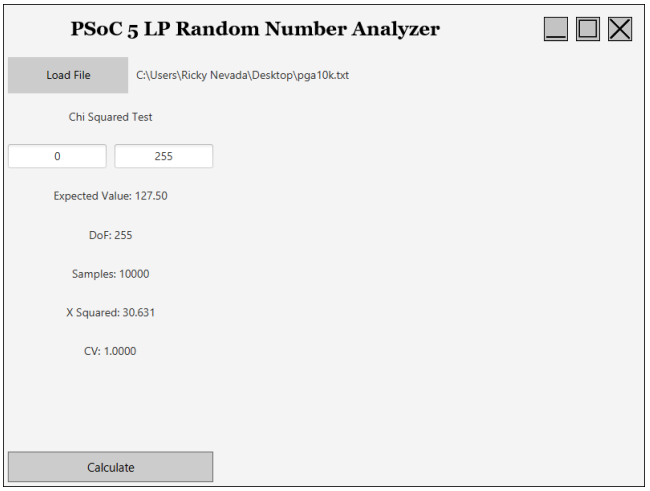

Figure B.4: PGA Results, P=1.0

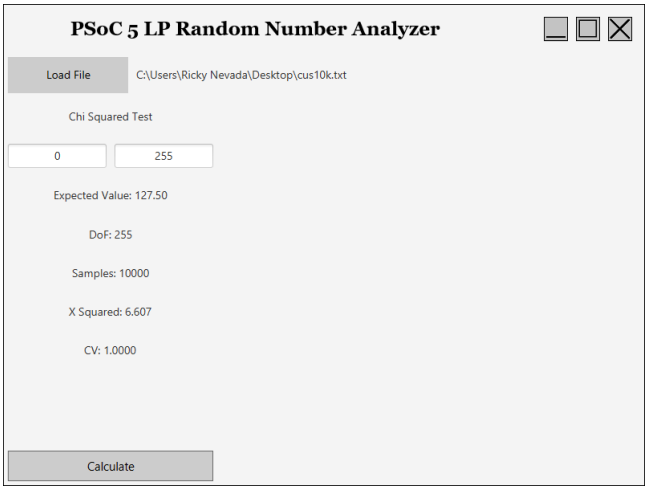

Figure B.5: LFSR RTC Seed Results, P=1.0

APPENDIX C: VARIOUS CODE

```
import java.math.BigInteger;

public class ChiSquared {

    public static double getConfidenceLevel(int n, double x){
        int newN = n/2;
        double newX = (double)x/2;
        double firstComponent = Math.exp(-1*newX);
        if(firstComponent == 0){
            return 0;
        }
        double secondComponent = exponential(newN-1,newX);
        return firstComponent*secondComponent;

    }

    private static double exponential(int n, double x){
        double summation = 0;
        for(int i = 0; i <= n; i++){
            double numerator = Math.pow(x,i);
            BigInteger denominator = factorial(i);
            summation += numerator/denominator.doubleValue();
        }
        return summation;
    }


    private static BigInteger factorial(int n){
        if(n <= 1){
            return new BigInteger(String.valueOf(1));
        }
       BigInteger bigInt = new BigInteger(String.valueOf(n));
       for(int i = n-1; i > 0; i--){
           bigInt = bigInt.multiply(new BigInteger(String.valueOf(i)));
       }
       return bigInt;
   }

}
```

Figure C.1: Confidence Value Implementation

```
HashMap<Integer, Integer> distribution = new HashMap<Integer, Integer>();
double chiSquared = 0;

public void processData(){
    chiSquared = 0;

    double expectedCount = (double)samples/range;
    for(int i = lowVal; i <= highVal; i++){
        if(distribution.containsKey(i)){
            chiSquared += Math.pow((double)(distribution.get(i)/expectedCount-1), 2);
        }else{
            chiSquared += 1;

        }
    }

    final double toPass = chiSquared;
    Platform.runLater(new Runnable(){

        @Override
        public void run() {
            DecimalFormat df = new DecimalFormat("#.000");
            xSquaredLbl.setText("X Squared: " + df.format(toPass));
        }

    });
}
```

Figure C.2: X^2 calculation w/ GUI Post, where the distribution HashMap is a collection of all generated numbers with their occurrences.

APPENDIX D: DIEHARDER TEST RESULTS

```
ricky@ricky-VirtualBox:~/Documents$ dieharder -f c99final.txt -g 202 -d 10
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    c99final.txt|  2.05e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 2 times
 diehard_parking_lot|   0|     12000|     100|0.12242429|  PASSED
ricky@ricky-VirtualBox:~/Documents$ dieharder -f c99final.txt -g 202 -d 14
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    c99final.txt|  2.01e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 2 times
        diehard_sums|   0|       100|     100|0.00026118|   WEAK
```

Figure D.1: C99 Results

```
ricky@ricky-VirtualBox:~/Documents$ dieharder -f prsfinal.txt -g 202 -d 10
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                   prsfinal.txt|  2.20e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 2 times
 diehard_parking_lot|   0|     12000|     100|0.12242429|  PASSED
ricky@ricky-VirtualBox:~/Documents$ dieharder -f prsfinal.txt -g 202 -d 14
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                   prsfinal.txt|  1.71e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 2 times
        diehard_sums|   0|       100|     100|0.00026118|   WEAK
```

Figure D.2: PRS Results

```
ricky@ricky-VirtualBox:~/Documents$ dieharder -f adcfinal.txt -g 202 -d 10
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                   adcfinal.txt|  2.80e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 6 times
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
ricky@ricky-VirtualBox:~/Documents$ dieharder -f adcfinal.txt -g 202 -d 14
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                   adcfinal.txt|  2.86e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 5 times
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
```

Figure D.3: ADC Results

```
ricky@ricky-VirtualBox:~/Documents$ dieharder -f cusfinal.txt -g 202 -d 10
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    cusfinal.txt|  2.32e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 6 times
 diehard_parking_lot|   0|     12000|     100|0.38232396|  PASSED
ricky@ricky-VirtualBox:~/Documents$ dieharder -f cusfinal.txt -g 202 -d 14
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    cusfinal.txt|  2.18e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 5 times
        diehard_sums|   0|       100|     100|0.06021147|  PASSED
```

Figure D.4: LFSR RTC Seed Results

```
ricky@ricky-VirtualBox:~/Documents$ dieharder -f pgafinal.txt -g 202 -d 10
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    pgafinal.txt|  2.16e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 6 times
 diehard_parking_lot|   0|     12000|     100|0.38232396|  PASSED
ricky@ricky-VirtualBox:~/Documents$ dieharder -f pgafinal.txt -g 202 -d 14
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                    pgafinal.txt|  2.35e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
# The file file_input was rewound 5 times
        diehard_sums|   0|       100|     100|0.06021147|  PASSED
```

Figure D.5: PGA Results

REFERENCES

[1] "Introduction to Randomness and Random Numbers." *Random.org*, 4 Feb. 2017, https://www.random.org/randomness/.

[2] Wikipedia contributors, "Linear congruential generator," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Linear_congruential_generator&oldid=768213826 (accessed March 4, 2017).

[3] Wikipedia contributors, "Pearson's chi-squared test," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Pearson%27s_chi-squared_test&oldid=766599635 (accessed March 5, 2017).

[4] "DieHarder: A Random Number Test Suite." *Robert G. Brown's General Tools Page*. 6 Mar. 2017, http://www.phy.duke.edu/~rgb/General/dieharder.php

[5] "The Tests." *A Study of Entropy*. 6 Mar. 2017, https://sites.google.com/site/astudyofentropy/background-information/the-tests