Supreme Court of Florida

No. AOSC20-45

IN RE: JUROR SELECTION PLAN: JEFFERSON COUNTY

ADMINISTRATIVE ORDER

Section 40.225, Florida Statutes, provides for the selection of jurors to serve within the county by "an automated electronic system." Pursuant to that section, the chief judge of the circuit must review and consent to the juror selection process, and the clerk of the circuit court must submit to the Supreme Court of Florida a description of the method for selecting jurors. Section 40.225(3), Florida Statutes, charges the Chief Justice of the Supreme Court with the review and approval of the proposed juror selection process, hereinafter referred to as the "juror selection plan."

The use of technology in the selection of jurors has been customary within Florida for more than 20 years and the Supreme Court has developed standards necessary to ensure that juror selection plans satisfy statutory, methodological, and due process requirements. The Court has tasked the Office of the State Courts Administrator with evaluating proposed plans for compliance with those standards.

By letter dated January 24, 2020, the Clerk of the Court for Jefferson County submitted the Jefferson County Juror Pool Selection Plan for review and approval

in accordance with section 40.225(2), Florida Statutes. The proposed plan reflects changes to both hardware and software used for juror pool selection in Jefferson County.

The Office of the State Courts Administrator has completed an extensive review of the proposed Jefferson County Juror Selection Plan, including an evaluation of statutory, due process, statistical, and mathematical elements associated with selection of jury candidates. The plan meets established requirements for approval.

Accordingly, the attached Jefferson County Juror Pool Selection Plan, received on February 12, 2020, from The Honorable Kirk Reams, Clerk of Court for Jefferson County, and approved by The Honorable Jonathan Sjostrom, Chief Judge of the Second Circuit, is hereby approved for use.

DONE AND ORDERED at Tallahassee, Florida, on June 10, 2020.

AOSC20-45 6/10/2020

Chief Justice Charles T. Canady

AOSC20-45 6/10/2020

ATTEST:

AOSC20-45 6/10/2020

John A. Tomasino, Clerk of Court

AOSC20-45 6/10/2020

SUPREME COURT OF THE STATE OF FLORIDA
SAT CITO SI RECTE

KIRK REAMS
Jefferson County
Clerk of Court & CFO

1 Courthouse Circle
Monticello, FL 32344
(850) 342-0218
Fax (850) 342-0222

January 24, 2020

Court Services
Office of the State Courts Administrator
Supreme Court Building
500 South Duval Street
Tallahassee, FL 32399

Dear Court Services:

Enclosed for your review is Juror Pool Selection Plan 2020 for Jefferson County. Included is the proposed hardware, software, random number generator program and algorithms to be used in the jury selection process. We are seeking Office of the State Courts Administrator review and Supreme Court approval of this process.

Chief Judge Jonathan Sjostrom has reviewed the attached documentation. A signed statement indicating her/his review is attached.

Please let me know if there is any other information that I may provide to you. I look forward to the Court's approval of the proposed jury pool selection plan. Thank you.

Sincerely,

Kirk Reams, Clerk of Court
Jefferson County

Attachments

xc: The Honorable Jonathan Sjostrom Chief Judge
Trial Court Administrator

OFFICE OF

JONATHAN SJOSTROM
CHIEF JUDGE
SECOND JUDICIAL CIRCUIT

LORRAINE GAUSS
JUDICIAL ASSISTANT
PHONE: (850) 606-4321
FAX: (850) 606-4474

LEON COUNTY COURTHOUSE
301 SOUTH MONROE STREET
TALLAHASSEE, FLORIDA 32301

February 7, 2020

Court Services
Office of the State Courts Administrator
Supreme Court Building
500 South Duval Street
Tallahassee, FL 32399

Dear Court Services:

In accordance with section 40.225(1), Florida Statutes, I have reviewed the juror pool selection plan for Jefferson County and consent to its use within my circuit.

Respectfully,

Jonathan Sjostrom
Chief Judge

KIRK REAMS
Clerk of Court & Chief Financial Officer
Jefferson County, Florida

January 24, 2020

Prepared by:

Kirk Reams

Clerk of Courts, Information Technology Department

Purpose and Scope

The purpose of this document is to describe the design and implementation of the Jury Selection Process to be used in a new Jury Management System for Jefferson County, Florida. The scope of this document includes the algorithms and methods used to:

- Create and Maintain a Master Candidate table
- Select a set of names from the Master Candidate table to create a Jury Pool

Also in scope is a description of the equipment, operating system and programming software, methods, and modes of operation to be used in the Jury Selection process. Sufficient detail will be provided to satisfy the statutory condition of selection "by lot and at random" and of due process as required by Chapter 40, Florida Statues.

I. Creation of the Initial Candidate Selection List

A. Source Lists

The initial candidate selection list is created from three (3) sources:

1. The quarterly list of names provided by the Department of Highway Safety and Motor Vehicles ("DHSMV") pursuant to 40.011 F.S. This data file is processed against the Master Candidate (master_jurors) table, adding new names and updating existing information, such as addresses.

2. The list of names of persons who have filed affidavits pursuant to 40.011 F.S. requesting to serve as a Juror. The JMS provides the capability for the Clerk to manually add individuals to the Master Candidate table as required.

3. An exclusion list containing those persons who are unable to serve, have already served, or whose service is otherwise excused or postponed pursuant to 40.013, 40.022, and 40.023 F.S. The JMS provides the capability for the Clerk to Exempt someone from service (i.e. Over 70), to Exclude some from serving on an individual Venire (i.e. a Caretaker), and to Postpone someone so that they will be included in a later Venire. For example, if an individual was postponed until September 30th 2012 due to a scheduling conflict, that postponement date is stored in their Master Candidate record, and the first Venire that is created on or after that date will automatically include that individual in addition to those that were selected at random.

4. The Clerk is the official custodian for the Master Candidate table (Source List) and is responsible for the security and integrity of the list and all processes.

5. The Master Candidate table will contain not less than 250 names, pursuant to 40.02 F.S.

B. Source Data Adjustments

1. Quarterly, DHSMV provides the Florida Court Clerks and Comptrollers ("FCCC") with a data file that contains all persons holding a valid Florida Driver License or Identification Card. FCCC acts as a distribution agent, grouping the data by County and dispersing it to the Clerk of Court. Jefferson County receives this data from FCCC quarterly via a dedicated network connection and the file is placed into a secured folder on a server for processing. The data received is an ASCII, fixed-length file, and is not modified in any way.

Once received, the DHSMV data is parsed and used to update the Master Candidate table in the Jury Management System where new persons are added and existing persons are updated (address, etc.). The Master Candidate table is a MSSQL relational database containing all eligible candidates gathered from the Initial Candidate Source Lists.

2. For those people who have filed affidavits pursuant to 40.011 F.S. requesting to serve as a Juror, their information is manually entered into the Candidates table by the Clerk of Court.

3. FCCC provides an ASCII data file to the Clerk of Court twice per month that contains persons who have deceased. This is an ASCII, fixed-length file similar to the Quarterly DHSMV file, and is also placed into a secured folder on a server for processing. It is not modified in any way. Once received, the data is parsed and used to update the Candidates table in the Jury Management System. Each deceased person is marked as such in the Master Candidate table and is no longer eligible for selection.

C. Time Frame

The DHSMV file is provided by FCCC quarterly, in January, April, July, and October. The Master Candidate table is updated with this information when it is received. Persons requesting to serve by signing an Affidavit are entered manually by the Clerk of Court as received. The Department of Health file containing deceased persons is provided by FCCC twice per month, on the 1st and the 15th. The Master Candidate table is updated with this information when it is received. Candidates that are excused pursuant to 40.013 F.S. or postponed to a later date are updated on-demand by the Clerk of Court.

D. Initial Jury Candidate List

Since this is a new application that is being created to replace an existing application, the Master Candidate table will be initially loaded by converting the current master Jury list file from an Informix database to a MSSQL database. No names will be excluded or modified during this conversion. This table contains all individuals possessing a Florida driver license or ID card as provided by the DHSMV for Jefferson County.

II. Name Selection

A. Equipment and Software

The Jury Management System is a graphical, browser-based application written in Microsoft's .NET environment, using the Visual Studio 2015 software development tool. It is a custom application designed and developed by Civitek.

The application will be housed on multiple servers with the following specifications:

- Model: virtual machine; physical HW model HPE proliant BL460 Gen9
- Processor: Physical CPUs = Intel Xeon E5-2630 v3 2.4 GHz. Dual socket; 8 cores per socket; 16 cores per host. Virtual CPUs = 2 virtual cpu's (per vm); total 4 cores (per vm).
- Memory: 8 GB memory per vm
- HDD:[SIZE] (OS Partition – C:\)[size] 80GB (Data Partition – E:\) 20 GB. Can be expanded.
- NIC: physical nic = Broadcom qlogic 57840 10 Gbps; virtual nic = vmxnet3 vmware network adapter.
- OS: Windows Server 2016 Standard Edition x64 version 1607 build 14393.2490

The database will be housed on four clustered database servers/SAN configuration model with the following specifications:

Server

- Model: ProLiant DL380 Gen9
- OS: Windows 2012 R2\ SQL server 2016 enterprise AlwaysOn
- Processor: Xeon e5-2667 v3, 2 sockets with 8 cores at 3.20GHz
- Memory: 768GB Ram Memory

Storage Area Network (SAN)

- Model: HP HP 3Par 7400
- Storage capacity: 54 Terabytes
- Storage allocated to the Jury database: 25 GB

The Server will store the SQL Server 2016 database software. The SAN will store the actual Jury database.

B. Security

The Clerk of Court is the official custodian and is responsible for the security and integrity of the name lists and of the juror pool selection process. All servers are located in the Civitek Data Center, which is secured by magnetic key-lock. Only authorized System Administrators have access to the Data Center. The Data Center is designed for 24x7 operations, and is supported by an Uninterruptible Power Supply and generator systems.

The Jury Management System is an Intranet application and as such, is not accessible to the public via the Internet. Only authorized users have access to the application. Only Civitek application developers and database administrators have access to the database. All information necessary for auditing purposes, including random number seed sets, is stored perpetually in MSSQL Server relational database tables as part of the application base.

C. Process Overview for Name Selection

The Clerk of Court and a Judge determine that a Jury Pool needs to be created and how many potential jurors the pool should contain. Information about this pool is entered into the Jury Management System ("JMS") such as the venire date, number of jurors needed, etc.

The JMS begins the selection process by determining the number of all eligible jury candidates (persons not permanently excused, have not served in the past year, etc.) from the Master Candidate table. For example, the Master Candidate table contains approximately 488,000 names in total, but approximately 280,000 names have either been Exempted (i.e. Over 70, Convicted Felon, Deceased, etc.) , or served or summoned within 1 year, or have been Postponed. The resulting set contains approximately 206,000 names. Each of these 206,000 names by example has a uniquely assigned Juror_ID.

The system seeds the CLCG4 Random Number Generator ("RNG") with a seed set and obtains a random number for each of the eligible jury candidates.

The next step of the Jury Selection process is to assign a Random Number to each of the 206,000 Juror_ID(s) and save to the Eligible Jurors temporary table. (Note: The Eligible Jurors temporary table is emptied at the start of this process.)

The Eligible Jurors temporary table is then sorted by the random number and read providing a randomly ordered list of jurors until the desired number of jurors for the pool is obtained. For example 15,000 requested jurors of the 206,000 Eligible Jurors would be randomly selected from the Eligible Jurors temporary table to create the Jury Pool.

D. Name Selection Algorithm

Random Number Generator

The Random Number Generator ("RNG") used in the Jury Management System is the C# implementation of the CLCG4 Random Number Generator by Pierre L'ecuyer and Terry H. Andres originally written in C and based on the Combination of Four LCGs(Linear Congruential Generators).

This is a portable package for uniform random number generation based on a backbone generator with period length near 2^121, which is a combination of four linear congruential generators. The package provides for multiple (virtual) generators evolving in parallel. Each generator also has many disjoint subsequences, and software tools are provided to reset the state of any generator to the beginning of its first, previous, or current subsequence. Such facilities are helpful to maintain synchronization for implementing variance reduction methods in simulation.

A sample of output showing the C# version of CLCG4 RNG produces identical values as the C version is provided in Appendix B: CLCG4 Random Number Generator.

For testing validation of the CLCG4 Random Number Generator, please refer to Appendix C: CLCG4 Random Number Generator – Validation Results.

Seed Sets

The RNG must be seeded before use to set a starting point for random number generation. The CLCG4 Random Number Generator accepts 4, 32-bit (128 bits) seed values and can produce astronomically more (upwards of 3x10^38) streams.

The JMS will create the 4 value seed set automatically in real time when a Jury Pool is being selected (see the Initialization Section (E) for details of how the seeds are created. After the random selection process is complete, the seed set that was used will be stored in a table for auditing purposes as well as an internal JMS step that verifies the same set of 4 Seed values have not been previously used. *This will 100% guarantee that the same 4 value seed set combination will never be used more than once.*

Name Selection Process

1. A record set is created with an SQL Select from the Master Candidate table. The SQL Select contains the necessary select/omit logic so as to include only persons actually eligible to serve at that point in time. It omits persons who have been excused or are otherwise not eligible to serve in this Jury Pool (i.e. Over 70, Convicted Felon, Deceased, other permanent excusals, or served or summoned within 1 year, and is not postponed to a future venire date.).

2. The RNG is initialized with the seed set (See Section (E) Initialization).

3. The system seeds the CLCG4 Random Number Generator ("RNG") with a seed set and obtains a random number for each of the eligible jury candidates.

4. The Eligible Jurors temporary table is emptied.

5. A Random Number is assigned to each of eligible jurors determined in step 1 and then saved to the Eligible Jurors temporary table. The Eligible Jurors temporary table consists of two columns the Juror_ID and the Random_Integer <random number>.

Juror_ID	**Random Integer**
10	4144534
13	9745733
15	7682723
18	1316912
19	6204797

Etc… (Up to All Eligible Jurors determined in step 1).

6. The Eligible Jurors temporary table is then sorted by random number and read providing a randomly ordered list of jurors.

Juror_ID	Random Integer
330698	2011963
459678	2012031
189324	2012063
137539	2012090
360849	2012095

Etc… (Will include All Eligible Jurors determined in step 1, just order by the Random Number).

7. All the jurors who have previously been granted a postponement for the current venire date are the first included into the selection list for jury service.

8. The top number of Eligible Jurors requested minus those previously postponement to this date as described in Step 7, are read from the randomly ordered list of jurors as defined in Step 6 and then inserted into the Pool_Jurors table as the Jury Pool members. During this process the record in the Master Candidate table is updated to reflect selection and thereby omitting them form the next Eligible Juror selection for 1 year.

9. The total number of requested jurors has been fulfilled and the Jury Pool has been selected.

E. Initialization

Seed Set Creation

The CLCG4 Random Number Generator requires four seed values upon initialization. These four seed values are four 32-bit integers, automatically system generated by the getSeedValue() class method which uses the Microsoft CryptGenRandom [advapi32.dll] function. The CryptGenRandom is a cryptographically secure pseudorandom number generator function and produces random bytes which are multiples of 8 bits. The CryptoAPI uses underlying system entropy to seed its internal crypto-graphic RNG. The entropy pool is hashed using one of the SHA algorithms so it applies approximately 128-160 bits of randomness (4-5 32-bit words).

The four seeds produced by the getSeedValue() class method are stored in a JMS SQL database table for auditing purposes as well as an internal JMS step that verifies the same set of four Seed values have not been previously used.

The review the C# code of the getSeedValue() class method and the usage of the Microsoft CryptGenRandom [advapi32.dll] function to produce the random four Seed Set values, please refer to Appendix A: CryptGenRandom Seed Value Generator.

Appendix A: CryptGenRandom Seed Value Generator (C# Language Version)

```
public static int getSeedValue()
{
 //The getSeedValue()class method produces four, 32-bit integers using the Microsoft CryptGenRandom
 //[advapi32.dll] function. The CryptGenRandom is a cryptographically secure pseudorandom number generator function
 //that produces random bytes which are multiples of 8 bits.
 //The CryptoAPI uses underlying system entropy to seed its internal crypto-graphic RNG.
 //The entropy pool is hashed using one of the SHA algorithms so it applies approximately 128-160 bits
 //of randomness (4-5 32-bit words).

    //--------------------------------------------------------------------------------------
    // Declare and initialize variables.

    IntPtr hProv = new IntPtr();

    uint dwLength = 32;
    Byte[] pbBuffer;
    pbBuffer = new byte[dwLength];

    //--------------------------------------------------------------------

    uint[] seed;
    seed = new uint[4];

    //--------------------------------------------------------------------

    try
    {
        bool res = Crypt32.CryptAcquireContext(out hProv, "", null, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT);

        if (res == true)
        {

            bool res2 = Crypt32.CryptGenRandom(hProv, dwLength, pbBuffer);

            if (res2 == true)
            {
                Buffer.BlockCopy(pbBuffer, 0, seed, 0, 16);

                gSeed1 = seed[0];
                gSeed2 = seed[1];
                gSeed3 = seed[2];
                gSeed4 = seed[3];

            }
            else
            {
                throw new ArgumentException("Error: Unable to instantiate Crypt32.CryptGenRandom.");
            }


            bool res3 = Crypt32.CryptReleaseContext(hProv, 0);

            return 1;
        }
        else
        {
            throw new ArgumentException("Error: Unable to instantiate Crypt32.CryptAcquireContext.");
        }


    }
    catch (Exception ex)
    {
        throw new ArgumentException("Error = " + ex.Message);
    }

}
```

Appendix B: CLCG4 Random Number Generator (C# Language Version)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RandomGenerator
{
    public class RandomGenerator
    {
        public static long Maxgen = 100;
        public static long i;
        public static long j;

        public static long[] aw = new long[4];
        public static long[] avw = new long[4];

        public static long[] a = new long[4] { 45991, 207707, 138556, 49689 };
        public static long[] m = new long[4] { 2147483647, 2147483543, 2147483423, 2147483323 };

        public static long[,] Ig = new long[4, 101];
        public static long[,] Lg = new long[4, 101];
        public static long[,] Cg = new long[4, 101];

        public static long H = 32768; // = 2^15 : use in MultModM.

        public enum SeedType
        {
            InitialSeed,
            LastSeed,
            NewSeed,
        }

        public static int gSeed1;
        public static int gSeed2;
        public static int gSeed3;
        public static int gSeed4;

        /*************************************************************************

        This is the Random Number Generator Based on the Combination of Four LCGs
        (Linear Congruential Generators) by Pierre L'ecuyer and Terry H. Andres.

         *
         * This is a portable package for uniform random number generation based on a
         * backbone generator with period length near 2^121, which is a combination
         * of four linear congruential generators.  The package provides for multiple
         * (virtual) generators evolving in parallel.  Each generator also has many
         * disjoint subsequences, and software tools are provided to reset the state
         * of any generator to the beginning of its first, previous, or current
         * subsequence.  Such facilities are helpful to maintain synchronization for
         * implementing variance reduction methods in simulation.
         *
         * For those who are interested in the details or the theories behind the
         * implementation, you can refer to the following papers by Prof. L'Ecuyer:
         *
         * 1. A Random Number Generator Based on the Combination of Four LCGs.
         *     (This can be downloaded from the "papers" page of Prof. L'Ecuyer's
         *     web page: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clcg4.ps).
         * 2. Implementing a Random Number Package with Splitting Facilities
         *    (ACM Transactions on Math. Software, Vol. 17, No. 1, March 91,
         *     p. 98-111)
         * 3. Efficient and portable combined random number generators.
         *    (Communications of the ACM, 31(6):742-749 and 774, 1988.)
         *
```

```
         * and the papers referred to therein.
         *
        *************************************************************************/

        public static float[] getBlockOfRandomNumbers(int seed1, int seed2, int seed3, int seed4,
int iCount, int iTestSwitch)
        {

            /**************************************************************************
             *
             * The Seeds MUST satisfy the following constraints:
                 1<= s[0] <=2147483646
                 1<= s[1] <=2147483542
                 1<= s[2] <=2147483422
                 1<= s[3] <=2147483322.

            **************************************************************************/

            long[] s = new long[4];

            if (iTestSwitch == 1)
            {
                // TEST Random Number Generator
                //1) Initialize the CLCG4 generator using the defaults.
                //   You will have access to 100 sub-generators in an array indexed from 0 to 99.
                //2) Advance the generator 20,000 numbers for generators 0, 1, 2, 3
                //3) The next five numbers from each generator segment and they should match the
                     numbers below.

                //   Skipping 20,000 numbers for generator 0 - 3 ...
                //       Gen 0      Gen 1           Gen 2           Gen 3
                //     ---------  ---------       ---------       ---------
                //1 - 0.5466955  0.8833518       0.8046209       0.7498480
                //2 - 0.3769046  0.9406571       0.7508963       0.0273824
                //3 - 0.8909396  0.5905970       0.6122372       0.0797647
                //4 - 0.8216189  0.1320690       0.2008421       0.4063177
                //5 - 0.0613126  0.9316329       0.2755311       0.1423134

                gSeed1 = 11111111; //<-- Initialize using the defaults of the TEST Requirements.
                gSeed2 = 22222222; //<-- Initialize using the defaults of the TEST Requirements.
                gSeed3 = 33333333; //<-- Initialize using the defaults of the TEST Requirements.
                gSeed4 = 44444444; //<-- Initialize using the defaults of the TEST Requirements.
                iCount = 5;        //<-- This required by the Rules of the TEST Requirements

                float[] fReturnBlock2 = new float[20]; //<-- 20 to return 4 Blocks of 5 from the
                                                       //    4 Generator Segments for TESTING.

                InitDefault();

                // Advance the generator 20,000 numbers for generators 0, 1, 2, 3
                    for (i = 1; i <= 20000; i++) GenVal(0);
                    for (i = 1; i <= 20000; i++) GenVal(1);
                    for (i = 1; i <= 20000; i++) GenVal(2);
                    for (i = 1; i <= 20000; i++) GenVal(3);

                //IF want to see the next 5 Test RAW Result Values for each of the 4 Generator Segments
                //output in a Console Applicaton in which you have reference the RandomGenertor.dll
                //then un-remark the 11 lines below.

                    //Console.WriteLine("Gen 0");
                    //for (i = 1; i <= iCount; i++) Console.WriteLine("{0:F2}", i + ": " + GenVal(0));
                    //Console.WriteLine("----------------");
                    //Console.WriteLine("Gen 1");
                    //for (i = 1; i <= iCount; i++) Console.WriteLine("{0:F2}", i + ": " + GenVal(1));
                    //Console.WriteLine("----------------");
                    //Console.WriteLine("Gen 2");
                    //for (i = 1; i <= iCount; i++) Console.WriteLine("{0:F2}", i + ": " + GenVal(2));
                    //Console.WriteLine("----------------");
                    //Console.WriteLine("Gen 3");
                    //for (i = 1; i <= iCount; i++) Console.WriteLine("{0:F2}", i + ": " + GenVal(3));

                //Return the next 5 Test Result Values for each of the 4 Generator Segments
                for (i = 0; i < iCount; i++)
                {
                    fReturnBlock2[i] = (float)GenVal(0);
```

```
            }

            for (i = 0; i < iCount; i++)
            {
                fReturnBlock2[i + 5] = (float)GenVal(1);
            }

            for (i = 0; i < iCount; i++)
            {
                fReturnBlock2[i + 10] = (float)GenVal(2);
            }

            for (i = 0; i < iCount; i++)
            {
                fReturnBlock2[i + 15] = (float)GenVal(3);
            }


            return fReturnBlock2;

        }
        else
        {
            // Random Number Generator
            gSeed1 = seed1;
            gSeed2 = seed2;
            gSeed3 = seed3;
            gSeed4 = seed4;

            float[] fReturnBlock2 = new float[iCount];

            InitDefault();

            for (i = 0; i < iCount; i++)
            {
                fReturnBlock2[i] = (float)GenVal(0);
            }

            return fReturnBlock2;
        }

    }

    static long MultModM(long s, long t, long M)
    /* Returns (s*t) MOD M.  Assumes that -M < s < M and -M < t < M.     */
    /* See L'Ecuyer and Cote (1991).                                     */
    {
        long R, S0, S1, q, qh, rh, k;

        if (s < 0) s += M;
        if (t < 0) t += M;
        if (s < H) { S0 = s; R = 0; }
        else
        {
            S1 = s / H; S0 = s - H * S1;
            qh = M / H; rh = M - H * qh;
            if (S1 >= H)
            {
                S1 -= H; k = t / qh; R = H * (t - k * qh) - k * rh;
                while (R < 0) R += M;
            }
            else R = 0;
            if (S1 != 0)
            {
                q = M / S1; k = t / q; R -= k * (M - S1 * q);
                if (R > 0) R -= M;
                R += S1 * (t - k * q);
                while (R < 0) R += M;
            }
            k = R / qh; R = H * (R - k * qh) - k * rh;
            while (R < 0) R += M;
```

```
        }
        if (S0 != 0)
        {
            q = M / S0; k = t / q; R -= k * (M - S0 * q);
            if (R > 0) R -= M;
            R += S0 * (t - k * q);
            while (R < 0) R += M;
        }
        return R;
    }


    //*---------------------------------------------------------------------*/

    public static void SetSeed(short g, long[] s)
    {
        if (g > Maxgen) Console.WriteLine("ERROR: SetSeed with g > Maxgen \n");
        for (j = 0; j < 4; j++) Ig[j, g] = s[j];
        InitGenerator(g, SeedType.InitialSeed);
    }

    public static void WriteState(short g)
    {
        Console.WriteLine("\n State of generator g = %u :", g);
        for (j = 0; j < 4; j++)
        {
            Console.WriteLine("\n   Cg[%u] = %lu", j, Cg[j, g]);
        }
        Console.WriteLine("\n");
    }

    public static void GetState(short g, long[] s)
    {
        for (j = 0; j < 4; j++) s[j] = Cg[j, g];
    }

    public static void InitGenerator(long g, SeedType Where)
    {
        if (g > Maxgen) Console.WriteLine("ERROR: InitGenerator with g > Maxgen \n");
        for (j = 0; j < 4; j++)
        {
            switch (Where)
            {
                case SeedType.InitialSeed:
                    Lg[j, g] = Ig[j, g]; break;
                case SeedType.NewSeed:
                    Lg[j, g] = MultModM(aw[j], Lg[j, g], m[j]); break;
                case SeedType.LastSeed:
                    break;
            }
            Cg[j, g] = Lg[j, g];
        }
    }

    public static void SetInitialSeed(long[] s)
    {
        long g;
        for (j = 0; j < 4; j++) Ig[j, 0] = s[j];
        InitGenerator(0, SeedType.InitialSeed);
        for (g = 1; g <= Maxgen; g++)
        {
            for (j = 0; j < 4; j++)
                Ig[j, g] = MultModM(avw[j], Ig[j, (g - 1)], m[j]);
            InitGenerator(g, SeedType.InitialSeed);
        }

    }

    public static void Init(long v, long w)
    {
        long[] sd = new long[4] { gSeed1, gSeed2, gSeed3, gSeed4 };
```

```
            for (j = 0; j < 4; j++)
            {
                aw[j] = a[j];
                for (i = 1; i <= w; i++)
                    aw[j] = MultModM(aw[j], aw[j], m[j]);
                avw[j] = aw[j];
                for (i = 1; i <= v; i++)
                    avw[j] = MultModM(avw[j], avw[j], m[j]);
            }
            SetInitialSeed(sd);
        }

        public static double GenVal(short g)
        {
            long k, s;
            double u;
            u = 0.0;
            if (g > Maxgen) Console.WriteLine("ERROR: Genval with g > Maxgen \n");

            s = Cg[0, g]; k = s / 46693;
            s = 45991 * (s - k * 46693) - k * 25884;
            if (s < 0) s = s + 2147483647; Cg[0, g] = s;
            u = u + 4.65661287524579692e-10 * s;

            s = Cg[1, g]; k = s / 10339;
            s = 207707 * (s - k * 10339) - k * 870;
            if (s < 0) s = s + 2147483543; Cg[1, g] = s;
            u = u - 4.65661310075985993e-10 * s;
            if (u < 0) u = u + 1.0;

            s = Cg[2, g]; k = s / 15499;
            s = 138556 * (s - k * 15499) - k * 3979;
            if (s < 0.0) s = s + 2147483423; Cg[2, g] = s;
            u = u + 4.65661336096842131e-10 * s;
            if (u >= 1.0) u = u - 1.0;

            s = Cg[3, g]; k = s / 43218;
            s = 49689 * (s - k * 43218) - k * 24121;
            if (s < 0) s = s + 2147483323; Cg[3, g] = s;
            u = u - 4.65661357780891134e-10 * s;
            if (u < 0) u = u + 1.0;

            return (u);
        }

        public static void InitDefault()
        {
            Init(31, 41);
        }

    }
}
```

Appendix C: CLCG4 Random Number Generator – Validation Results

The validation of the C# implementation of the CLCG4 Random Number Generator can be tested by following these steps.

1) Initialize the CLCG4 generator using the defaults defined

Seed1 = 11111111

Seed2 = 22222222

Seed3 = 33333333

Seed4 = 44444444

(Note: You will have access to 100 sub-generators in an array indexed from 0 to 99.)

2) Advance the generator 20,000 numbers for generators 0, 1, 2, 3

3) The next five numbers from each generator segment and they should match the numbers

```
Skipping 20,000 numbers for generator 0 - 3 ...

        Gen 0      Gen 1           Gen 2           Gen 3

     ---------     ---------       ---------       ---------

1 - 0.5466955     0.8833518       0.8046209       0.7498480

2 - 0.3769046     0.9406571       0.7508963       0.0273824

3 - 0.8909396     0.5905970       0.6122372       0.0797647

4 - 0.8216189     0.1320690       0.2008421       0.4063177

5 - 0.0613126     0.9316329       0.2755311       0.1423134
```

This can also be tested by via C# .Net by passing **iTestSwitch** = 1 to the RandomGenerator.getBlockOfRandomNumbers(iSeed1, iSeed2, iSeed3, iSeed4, iCount, **iTestSwitch**) class method.

Appendix D: Acknowledgements and References

Acknowledgments

The CLCG4 Random Number Generator upon which this implementation was based was created by Pierre L'ecuyer and Terry H. Andres.

A Random Number Generator Based on the Combination of Four LCGs.

(This can be downloaded from the "papers" page of Prof. L'Ecuyer's web page: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clcg4.ps).

A PDF version of this document is attached.

Additional Associated References

1) P.Bratley, B. L. Fox, and L. E. Schrage. A Guide to Simulation Springer-Verlag, New York, second edition, 1987.

2) D. E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., second edition, 1981

3) A. M. Law and W. D. Kelton. Simulation Modeling and Analysis. McGraw-Hill, New York, second edition, 1991.

4) P.L'Ecuyer. Effcient and portable combined random number generators. Communications of the ACM, 31(6):742-749 and 774, 1988.
See also the correspondence in the same journal, 32, 8 (1989) 1019-1024

5) P.L'Ecuyer. Random numbers for simulation. Communications of the ACM, 33(10):85-97, 1990.