Appendix to: "A Review of TESTU01"
reviewed by B. D. McCullough[1]

1 Introduction

The purpose of this Appendix is to present the reader with instructions for installing TESTU01 and getting it running, using TESTU01 to apply both specific tests and batteries of tests to a pre-programmed RNG, and to test an RNG that is not pre-programmed. By the time this review appears, TESTU01 will have been revised a couple more times, so the specific commands given here might not work. Nonetheless, simply reading this appendix will give the reader a good idea of how to use TESTU01, and will make it easier to understand the TESTU01 documentation. More sophisticated uses of TESTU01 can be learned by consulting the User Guide.

All code used to produce this appendix is contained in two zip files: ccodeforRrngs.zip and codelistings.zip (some programs appear in both files). The former contains the code necessary to test all the RNGs given in Table 1, and the latter contains ASCII versions of the code listings given at the end of this appendix.

2 Installing TESTU01

The following worked on Red Hat 9.0 with TestU01-03. The procedure might change slightly for subsequent versions of TestU01, so be sure to consult the README file that comes with the software.

From www.iro.umontreal.ca/~simardr/ obtain the software as a zip file, *e.g.*, Testu01-03.zip and place the it file in the preferred directory, *e.g.*, /home/myname, and unzip it. This will create the testu01-02 subdirectory which, in turn, will have a README file and 7 subdirectories: doc, examples, include, lib, mylib, param, probdist, and testu01.

The README instructs the user to edit two files. In the file /mylib/gdef.tex change nothing. In the subdirectory /lib depending on whether one uses the c-shell or the bash shell, in about the sixth line of the file Crc.csh or Crc.sh (respectively) change the default directory from /u/simardr/paquets/alcatel/testu01-03 to, *e.g.*, /home/myname/testu0-03. From within the /lib subdirectory, to invoke the former from a c-shell, issue the command `source Crc.csh` and to invoke the latter from bash the command ./Crc.sh must be invoked.

[1]Thanks to Akram Muhammad and Thomas Flottemesch, who test-drove the intstallation instructions, and Merrill Liechty, who test-drove the examples. Thanks are also due to Pierre L'Ecuyer and Richard Simard for comments, with additional thanks to Simard for writing the C programs used in this Appendix.

smultin (based on multinomial distribution)
  independent cells
  overlapping cells
  independent cells (bit test version)
  overlapping cells (bit test version)
sentrop (entropy-based tests)
  independent blocks
  overlapping blocks; number of bits $n \leq 31$
  overlapping blocks; number of bits $n > 31$
  Dudewicz/van der Meulen test
  Dudewicz/van der Meulen circular version
snpair (based on distance between points)
  distance measured by L_p norm
  a binary measure of distance
  closest pairs test of Bickel and Breiman
sknuth (Knuth's classic tests)
  serial test
  serial sparse test
  permutation test
  gap test
  simplified poker test
  coupon collector test
  runs of bits test
  independent runs test
  'maximum of t' test
  collision test
  collision-permutation test
smarsa (tests due to Marsaglia)
  overlapping t-tuple test
  overlapping pairs sparse occupancy test
  monkey test
  monkey test (bit test version)
  birthday spacings test
  rank of a binary random matrix
  savir2 test
  all spacings two test
svaria (various tests)
  sample mean test
  sample correlation test
  sample product test
  sum of the logarithms test
  test of Matsumoto and Kurita
  weighted distribution test
  arg max collision test
  sum of uniforms test
  appearance spacings test
swalk (tests based on the random walk)
  random walk one
  random walk one-a
  number of steps necessary to exceed μ
  number of steps still less than $1 - \mu$
scomp (tests based on linear complexity)
  jumps test
  jumps size test
  compressibility test
sspectral (tests based on spectral methods)
  Fourier one
  Fourier two
  Fourier three
sstring (tests applied to strings of random bits)
  periods in strings
  longest run of ones
  Hamming weights test
  Hamming correlation test
  Hamming independence test
  runs test
  autocorrelation test
sspacings (tests based on the sorted uniforms)
  sum of the logs test
  sum of the squares test
  scans test
  all spacings test

Table 1: Groups (italics) of tests (indented) in TESTU01

Among other things, the files Crc.csh and Crc.sh set environment variables, etc. and define some useful commands, *e.g.*, ccl (that's a lowercase ell, not a one). Executing `ccl fn.c` submits fn.c to gcc with all the TESTU01 libraries, etc., as part of the call.

Working from bash, the instruction did not appear to work. To get around this, still in the /lib subdirectory, switch to the c-shell by issuing the command `exec /bin/csh`. Then issue the command `source Crc.csh`. There is one thing left to do. From the /lib subdirectory, issue the command `make`. If this runs without an error, TESTU01 has been installed. Otherwise, refer to the README file for further information.

After successful installation, every time an xterm window is opened for the purpose of using TESTU01, it is necessary to issue the pair of commands `exec /bin/csh` and `source Crc` from the /lib subdirectory.

3 A Preliminary Example

Now go to the subdirectory /examples and issue the command `ccl birth1.c` followed by `./a.out`. With the exception of some deleted blank lines, you should see the output below:

```
*********************************************************
MACHINE = localhost.localdomain
ulcg_CreateLCG:   m = 2147483647,   a = 397204094,   c = 0,   s = 12345

smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 1000,  r =  0,    d = 10000,    t = 2,    Order = 1

      Number of cells = dˆt =          100000000
      Lambda = Poisson mean =       2.5000
---------------------------------------------------
Total expected number = N*Lambda      :       2.50
Total observed number                 :       6
Significance level of test            :    0.04
-----------------------------------------------
CPU time used                    :  00:00:00.00

Generator state:
 s = 1858647048
```

```
***********************************************************
MACHINE = localhost.localdomain
ulcg_CreateLCG:   m = 2147483647,   a = 397204094,   c = 0,   s = 12345

smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 10000,  r =  0,    d = 1000000,    t = 2,    Order = 1

      Number of cells = dˆt =      1000000000000
      Lambda = Poisson mean =       0.2500
-----------------------------------------------------
Total expected number = N*Lambda      :       0.25
Total observed number                 :      44
Significance level of test            :   eps      *****
-----------------------------------------------
CPU time used                    :  00:00:00.01

Generator state:
 s = 731506484
```

The program birth1.c demonstrates the importance of being able to vary the parameters of a statistical test: the RNG being tested passes the "Birthday Spacings Test" when the number of birthdays $n = 1000$ and the number of intervals on the line $d = 10,000$. The $p-$value of 0.04 is suspect. As opposed to DIEHARD, where the test parameters are fixed, with TESTU01 the parameters can be varied to investigate the suspicious behavior of the RNG. When the parameters are changed to $n = 10,000$ and $d = 1,000,000$, the RNG fails catastrophically, with a significance level of "eps".

The usual hypothesis testing framework does not apply to testing random numbers. By "failure" we mean catastrophic failure, the requisite evidence of which is a p-value that is either zero or unity to several decimal places. For example, a p-value of 1E-10 is a catastrophic failure (L'Ecuyer and Simard, 2002, p. 79). This is necessary, but not sufficient. The additional necessary condition is that the failure be replicated for a different seed. For example, if an RNG fails the "Birthday Spacings" test with p-value=1E-10 for one seed, but fails only with p-value=1E-6 (or does not fail at all) with another seed, then one would not conclude that the RNG fails the Birthday Spacings test. On the other hand, if it failed both times with p-value=1E-10, then it would be correct to conclude that the RNG fails the Birthday Spacings test.

It is useful to examine the program birth1.c, given below:

```
#include "unif01.h"
```

```
#include "ulcg.h"
#include "smarsa.h"
#include <stddef.h>

int main (void)
{
   unif01_Gen *gen;
   gen = ulcg_CreateLCG (2147483647, 397204094, 0, 12345);
   smarsa_BirthdaySpacings (gen, NULL, 1, 1000, 0, 10000, 2, 1);
   smarsa_BirthdaySpacings (gen, NULL, 1, 10000, 0, 1000000, 2, 1);
   ulcg_DeleteGen (gen);
   return 0;
}
```

The `#include` statements specify the various modules of TESTU01 that will be needed. For example `#include "ulcg.h"` specifies the linear congruential generators module (the "u" at the beginning stands for "uniform"), and `#include "smarsa.h"` specifies the Marsaglia tests (the "s" at the beginning stands for "statistical tests"). The actual commands to be used, *e.g.*, `smarsa_BirthdaySpacings` are described in detail in the User's Guide.

It is very easy to crib from the examples in the /testu01-02/examples subdirectory to choose different combination of RNGs and tests. Suppose one wished to combine two LCGs and then apply the Small Crush battery of tests to this combined LCG. The program test.c (availabe at the journal archive) would be as follows:

```
#include "unif01.h"
#include "ulcg.h"
#include "bbattery.h"
#include <stddef.h>

int main (void)
{
   unif01_Gen *gen1, *gen2, *gen3;
   gen1 = ulcg_CreateLCG (2147483543, 10064, 0, 12345);
   gen2 = ulcg_CreateLCG (2147483563, 16493, 0, 12345);
   gen3 = unif01_CreateCombAdd2 (gen1, gen2,"My Combination of LCGs");
   bbattery_SmallCrush (gen3);
   unif01_DeleteCombGen (gen3);
   ulcg_DeleteGen (gen1);
```

```
   ulcg_DeleteGen (gen2);
   return 0;
}
```

To test this combined RNG, simply issue the command `ccl test.c` followed by the command `./a.out` and read the output. Note that instead of loading the Marsaglia tests, the library that contains the Small Crush battery of tests is loaded. Two very nice features of the Crush batteries is that, in addition to providing a listing of all the test results, at the end is a separate list that summarizes the failures, so there is no need to wade through several pages of output trying to decide if some particular result constitutes success or failure (persons who have applied DIEHARD will appreciate this feature). Moreover, it is possible (by loading the 'swrite.h' module) to vary the amount of output produced by each test.

4 A Practical Example

Of course, what we really want to do is test the RNGs in our statistical and econometric packages, and if these RNGs are not pre-programmed in TESTU01 then we have to implement them ourselves. This can be done in two ways: program the RNG in C, or call the package's RNG from C code (if the package permits this). Either way, the ability to program C is required. The User Guide gives an example of how to "roll your own" (Figure 2.4, p. 19), but persons not fluent in the C programming language may find it completely unsatisfactory.

A good choice for a guinea pig is the default uniform RNG in the package "R" (Ihaka and Gentleman, 1996). There are several reasons for this. First, no formal, independent testing has been done on the "R" RNGs. Second, the source code for the RNG is readily available at the "R" website (www.r-project.org) so there is no quibbling with some software developer (try getting the precise details of the RNG in your favorite statistical package and you'll see what I'm talking about). Third, the "R" RNG is superb: it offers a variety of uniform generators (as well as a variety of transforms to normality) including (and this is very important) the ability to call user-written RNGs that are compiled. This is very important because no user should have to depend solely on the RNG that the software developer happens to include.

The default uniform RNG in R v1.6.1 is Marsgalia's Multicarry. TESTU01 has this pre-programmed as `umarsa_CreateMWC97R`, but it is instructive to see how it might be programmed by a user. The relevant lines of the "R" source code (which is written in C) are as follows:

```
    case MARSAGLIA_MULTICARRY:/* 0177777(octal) == 65535(decimal)*/
I1= 36969*(I1 & 0177777) + (I1>>16);
I2= 18000*(I2 & 0177777) + (I2>>16);
```

```
return ((I1 << 16)^(I2 & 0177777)) * i2_32m1; /* in [0,1) */
```

The C program that implements the Marsaglia Multicarry for use in TESTU01, multicarry.c, is available at the journal archive.

Examining the “include” statements at the top of multicarry.c, the only one that is not already part of one or another of the libraries in TESTU01 is “multicarry.h” which has to be written by the user, and is available at the journal archive:

```
#include "unif01.h"
unif01_Gen *CreateMultiCarry (unsigned long I1, unsigned long I2);
void DeleteMultiCarry (unif01_Gen * gen);
```

The C program that calls multicarry.h and applies the Small Crush battery, to it, poil.c (available at the journal archive) follows:

```
/*
 * Generate 10 numbers in [0, 1), then apply SmallCrush on MultiCarry
 */
#include "bbattery.h"
#include "unif01.h"
#include "multicarry.h"
#include <stdio.h>

int main (void)
{
   int i;
   unif01_Gen *gen;

   gen = CreateMultiCarry (1012585244, -997230227);
   for (i = 0; i < 10; i++)
      printf ("%18.16f\n", unif01_StripD (gen, 0));

   bbattery_SmallCrush (gen);
   DeleteMultiCarry (gen);

   return 0;
}
```

To change from Small Crush to Big Crush it is necessary only to change `bbattery_SmallCrush (gen)` to `bbattery_BigCrush (gen)`. Two things are of particular note in the poil.c program. First, the seeds for the Multicarry are given as 1012585244 and -997230227 . Second, a 'for' loop prints out the first ten random numbers, so that I can compare them to the random numbers actually produced by "R" when I give the same seeds to its default generator. The relevant output from "R" is given below, where a single integer, 123457, is used as a seed from which the two seeds necessary to define the Multicarry RNG are created. To find these two seeds the command '.Random.seed' is issued.

```
R : Copyright 2002, The R Development Core Team
Version 1.6.1  (2002-11-01)

>  set.seed(123457,kind="Marsaglia-Multicarry")
>  .Random.seed
[1]          1 1012585244 -997230227
> runif(10)
 [1] 0.5735857858261059 0.0658577107977722 0.9885551664020293
 [4] 0.3964184309347575 0.3679865683354405 0.1116248930598667
 [7] 0.6149889928323655 0.0728358628863552 0.8065828624662436
 [10] 0.4819400479276524
>
```

Now all that remains is to put poil.c, multicarry.c and multicarry.h in the same directory and issue the command `ccl poil.c multicarry.c`. This creates the 'a.out' executable file, partial output from which follows:

```
[bmccullo@localhost examples]\$ ./a.out
0.5735857858261059
0.0658577107977722
0.9885551664020293
0.3964184309347575
0.3679865683354405
0.1116248930598667
0.6149889928323655
0.0728358628863552
0.8065828624662436
0.4819400479276524
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                  Starting Crush
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

************************************************************
Test smarsa_SerialOver calling smultin_MultinomialOver

************************************************************
MACHINE = localhost@localdomain

CreateMultiCarry:   I1 = 1012585244,   I2 = 3297737069

.... several hundred lines deleted ....

------------------------------------------------
CPU time used                    :  00:01:10.83

Generator state:
 I1 = 1414800700,   I2 = 413436365

============== Summary results of Crush ==============

 Generator:        CreateMultiCarry
 Number of tests:  60
 Total CPU time:   01:44:34.14
 The following tests gave p-values outside [0.01, 0.99]:
 (eps means a value < 1.0e-15)

       Test                          p-value
 ----------------------------------------------
  1  SerialOver (t = 2)             1 - eps
  2  SerialOver (t = 4)               eps
  4  CollisionOver (t = 2)          1 - eps
  8  MultinomialBitsOver            1 -  3.1e-7
 10  BirthdaySpacings (t = 2)        1.0e-4
 11  BirthdaySpacings (t = 4)         eps
 12  BirthdaySpacings (t = 13)        eps
 13  BirthdaySpacings (t = 13)        eps
```

```
 14  ClosePairs (t = 2)              eps
 15  ClosePairs (t = 4)              eps
 16  ClosePairsBitMatch (t = 2)    1 - eps
 23  Gap                            8.8e-5
 24  Gap                             eps
 25  Gap                             eps
 27  Permutation                   1 - 6.6e-14
 29  MaxOft                          eps
 30  MaxOft                          eps
 37  WeightDistrib                  8.1e-6
 48  Fourier3                       2.6e-3
 52  HammingCorr (L = 30)          2.1e-14
 55  HammingIndep (L = 30)          1.6e-4
 ----------------------------------------------
 All other tests were passed
```

After each of the tests is applied, TESTU01 prints the amount of CPU time that the test required (this is not shown in the abbreviated output above). In the summary output, which can be seen above, is also displayed the "Total CPU time:"; be sure not to confuse an individual test time with the total test time.

As can be seen from the comparison of the "R" output and the TESTU01 output, both are using the same generator since they produce the same string of numbers. Thus we can be confident that the results produced by TESTU01 shed light on the efficacy of the default RNG in "R".[2] The program prints a convenient summary: the names of all tests that produce a p-value outside the [0.1, 0.99] interval (these limits can be adjusted by using the variable `gofw_suspectp` in module `probdist/gofw`, see page 27 of the User Guide). Actually, the summary is more than convenient, it is a necessity, since the output for Small Crush, Crush, and Big Crush is on the order of 500, 2000, and 1500 lines, respectively.

Examining the summary, many catastrophic failures can be seen (tests 1, 2, 4, 11, 12, 13, 14, 15, 16, 24, 25, 27, 29, 30, 52). Near catastrophic failures occurs with tests 8 and 37. However, the birth1.c example in Section Two suggests that some of these near failures might be failures, if only the parameters of the test were varied, especially the "27 Permutation" and "52 HammingCorr (L=30)" tests. As the Marsaglia Multicarry passes all the DIEHARD tests, this is an example of a test that passes all the DIEHARD yet fails TESTU01.

[2]Technically, since only the first ten numbers are checked, and later numbers may vary, the only way to be absolutely sure is to feed the numbers directly from the package into TESTU01 (thanks to Duncan Murdoch for pointing this out).

5 A Second Example

There exists a pre-programmed Wichmann-Hill generator `ulcg_CreateCombWH3` (Wichmann-Hill 1) but it is also possible to construct one by combining three LCGs (Wichmann-Hill 2). The two methods are algebraically equivalent, but not numerically equivalent. The pre-programmed version is slightly more efficient, because the three components are updated by the same function call, whereas the constructed version makes three separate function calls. Consequently, there can be a large difference in the amount of time each takes, as shown in Table 1.

```
Wichmann-Hill-1
   unif01_Gen *gen1;
   gen1 = ulcg_CreateCombWH3(30269, 30307,30323,171,172,
          170,0,0,0,26656,2092,3794);


Wichmann-Hill-2
   unif01_Gen *gen1, *gen2, *gen3, *gen4;
   gen1 = ulcg_CreateLCG (30269,171, 0, 26656);
   gen2 = ulcg_CreateLCG (30307,172, 0, 2092);
   gen3 = ulcg_CreateLCG (30323,170, 0, 3794);
   gen4 = unif01_CreateCombAdd3 (gen1, gen2, gen3, "Wichmann-Hill");
```

```
 (eps means a value < 1.0e-15)
       Test                          p-value
 ----------------------------------------------
 10  BirthdaySpacings (t = 2)          eps
 11  BirthdaySpacings (t = 4)          eps
 13  BirthdaySpacings (t = 13)        2.4e-3
 14  ClosePairs (t = 2)                eps
 15  ClosePairs (t = 4)               2.1e-14
 44  RandomWalk1 (L = 10000)           0.9966
 ----------------------------------------------
 All other tests were passed
```

Table 1 presents results for all the uniform generators in R. The code for running these tests is at this journal's archive. There is a slight problem in the table. In R there seems to be a problem setting the seed for the Mersenne Twister. This researcher was unable to use the pre-programmed Mersenne Twister in TESTU01 to duplicate the first ten random numbers produced

	catastrophic failures (CPU time)		
RNG	Small Crush	Crush	Big Crush
Marsaglia Multicarry	1 (00:00:46)	13 (01:44:34)	19 (15:06:40)
Super-Duper	1 (00:00:48)	9 (01:47:21)	12 (15:21:20)
Wichmann-Hill-1	1 (00:01:03)	3 (02:18:39)	4 (19:57:47)
Wichmann-Hill-2	1 (00:01:21)	3 (02:56:37)	4 (25:39:25)
Mersenne Twister	0 (00:00:46)	0 (01:52.22)	0 (15:58:25)
Knuth TAOCP	0 (00:00:44)	0 (01:38:29)	0 (14:20:26)
Knuth TAOCP 2002	0 (00:00:44)	0 (01:39:46)	0 (14:29:36)

Table 2: Crush Tests Applied to Several RNGs from "R"
time in hours:minutes:seconds

by the Mersenne Twister in R. Nonetheless, the Mersenne Twister results in Table 2 are indicative of the performance of the, the same-named RNG in R.

Examining Table 1, of the six uniform RNGs in R, only the Mersenne-Twister and the pair of Knuth-TAOCP generators appear to be bullet-proof, consequently only they can be recommended. We know from experience that non-uniform uniform RNGs act in unpredictable ways. For example, the Marsaglia-Zaman subtract-with-borrow generator gives perfectly sensible results in many situations, but when combined with the Box-Muller method it produces normal random deviates that are not normal. The non-normality is slight enough to avoid detection by the standard methods, but still substantial enough to noticeably change Monte Carlo results. Of course, one will only observe that the Monte Carlo results are wrong if one runs the same experiment using two different generators. Many packages do not afford users this "luxury", so the unfortunate users of such deficiently designed packages could not run their Monte Carlo with two different generators, even if they were so inclined.

6 Conclusions

The careful reader should be able to perform rudimentary analyses of RNGs using TESTU01. The reader who is interested in further uses of TESTU01 is referred to the User Guide.

REFERENCES

Ihaka, Ross and Robert Gentleman (1996), "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics* **5**, 299-314

L'Ecuyer, Pierre and Richard Simard (2002), "TESTU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators, User's Guide, Detailed Version" (comes with the software TESTU01)

L'Ecuyer, Pierre and Richard Simard (2002a), "MYLIB-C: A Small Library of Basic Utilities in ANSI C" (comes with the software TESTU01)

L'Ecuyer, Pierre and Richard Simard (2002b), "PROBDIST: A Software Library of Probability Distributions and Goodness-of-Fit Statistics in ANSI C" (comes with the software TESTU01)

Code Listing A: multicarry.c

```
#include "multicarry.h"
#include "unif01.h"
#include "util.h"
#include "addstr.h"
#include <string.h>

typedef struct{
   unsigned long I1, I2;
} MultiCarry_state;

static unsigned long MultiCarry_Bits (void *par, void *sta)
{
   MultiCarry_state *state = sta;
   state->I1 = 36969 * (state->I1 & 0177777) + ((state->I1 >> 16) & 0177777
   state->I2 = 18000 * (state->I2 & 0177777) + ((state->I2 >> 16) & 0177777
   return (state->I1 << 16) ^ (state->I2 & 0177777);
}

static double MultiCarry_U01 (void *par, void *sta)
{
   return MultiCarry_Bits (par, sta) * 2.328306437080797e-10;
}

static void WrMultiCarry (void *sta)
{
   MultiCarry_state *state = sta;
   printf (" I1 = %lu,   I2 = %lu\n", state->I1, state->I2);
}

unif01_Gen *CreateMultiCarry (unsigned long I1, unsigned long I2)
{
   unif01_Gen *gen;
   MultiCarry_state *state;
   size_t leng;
   char name[60];
```

```
   gen = util_Malloc (sizeof (unif01_Gen));
   gen->state = state = util_Malloc (sizeof (MultiCarry_state));
   state->I1 = I1;
   state->I2 = I2;
   gen->param = NULL;
   gen->Write = WrMultiCarry;
   gen->GetU01 = MultiCarry_U01;
   gen->GetBits = MultiCarry_Bits;

   strcpy (name, "CreateMultiCarry:");
   addstr_Ulong (name, "   I1 = ", I1);
   addstr_Ulong (name, ",   I2 = ", I2);
   leng = strlen (name);
   gen->name = util_Calloc (leng + 1, sizeof (char));
   strncpy (gen->name, name, leng);
   return gen;
}

void DeleteMultiCarry (unif01_Gen * gen)
{
   gen->state = util_Free (gen->state);
   gen->name = util_Free (gen->name);
   util_Free (gen);
}
```

Code Listing B: multicarry.h

```
#include "unif01.h"
unif01_Gen *CreateMultiCarry (unsigned long I1, unsigned long I2);
void DeleteMultiCarry (unif01_Gen * gen);
```

Code Listing C: poil.c

```
/*
 * Generate 10 numbers in [0, 1), then apply SmallCrush on MultiCarry
 */
#include "bbattery.h"
#include "unif01.h"
```

```
#include "multicarry.h"
#include <stdio.h>

int main (void)
{
   int i;
   unif01_Gen *gen;

   gen = CreateMultiCarry (1012585244, -997230227);
   for (i = 0; i < 10; i++)
      printf ("%18.16f\n", unif01_StripD (gen, 0));

   bbattery_SmallCrush (gen);
   DeleteMultiCarry (gen);

   return 0;
}
```

Code Listing D: test.c

```
#include "unif01.h"
#include "ulcg.h"
#include "bbattery.h"
#include <stddef.h>

int main (void)
{
   unif01_Gen *gen1, *gen2, *gen3;
   gen1 = ulcg_CreateLCG (2147483543, 10064, 0, 12345);
   gen2 = ulcg_CreateLCG (2147483563, 16493, 0, 12345);
   gen3 = unif01_CreateCombAdd2 (gen1, gen2,"My Combination of LCGs");
   bbattery_SmallCrush (gen3);
   unif01_DeleteCombGen (gen3);
   ulcg_DeleteGen (gen1);
   ulcg_DeleteGen (gen2);
   return 0;
}
```