pbRand, an Innovative Pseudo-Random Generator

pbRand comes in four forms: 10 and 20 32-bit words and 10 and 20 64-bit words. These words form the seed used to generate successive pseudo-random numbers. In this document please assume that all numbers are unsigned.

pbRand1032: 10 32-bit words or 40 bytes

pbRand2032: 20 32-bit words or 80 bytes

pbRand1064: 10 64-bit words or 80 bytes

pbRand2064: 20 64-bit words or 160 bytes

All four forms are implemented in C++ and two assembly languages, x64 (tested using VS 2019) and 32-bit ARM (tested as M4 using IAR tools).

Each form has the same entry points for seeding and returning random values. (The assembly language versions do the C++ polymorphism using the buffer address. The C++ version uses the “ptr->func()” syntax.

All PRNGs have two basic functions: seed=f1(seed) and return f2(seed). And many have an often hidden seeding function as well. In pbRand several useful seeding functions are provided.

pbRand has an inner algorithm, mix(), used by both seeding and returning random numbers. That inner algorithm is described first. For simplicity, the 1032 form is described followed by the minor differences used by the other forms.

**THE MIX**

Let’s start with almost-the-mix()

seed[3] += seed[0];

seed[4] += seed[1];

seed[5] += seed[2];

seed[6] += seed[3];

seed[7] += seed[4];

seed[8] += seed[5];

seed[9] += seed[6];

seed[0] += seed[7];

seed[1] += seed[8];

seed[2] += seed[9];

As the above code runs many times the low-order bits of each word follow a 1023 cycle sequence. Think linear feedback shift register length 10 with feedback at 7. (See the 7 when seed[0] is changed.)

The designer was unhappy with this almost-the-mix because none of the higher-order bits contribute to the low-order bits. His first thought was using the add-with-carry instruction after adding to the first seed word. As easy as that is in assembly (his favorite language) the algorithm had to be useable in C/C++. mix() actually uses rol(int,1), a rotate left one bit. This allows the low-order bits to be changed by high-order bits.

seed[3] += seed[0]; seed[3] ^= rol(seed[2],1);

seed[4] += seed[1]; seed[4] ^= rol(seed[3],1);

seed[5] += seed[2]; seed[5] ^= rol(seed[4],1);

seed[6] += seed[3]; seed[6] ^= rol(seed[5],1);

seed[7] += seed[4]; seed[7] ^= rol(seed[6],1);

seed[8] += seed[5]; seed[8] ^= rol(seed[7],1);

seed[9] += seed[6]; seed[9] ^= rol(seed[8],1);

seed[0] += seed[7]; seed[0] ^= rol(seed[9],1);

seed[1] += seed[8]; seed[1] ^= rol(seed[0],1);

seed[2] += seed[9]; seed[2] ^= rol(seed[1],1);

The fact that mix() is invertible is a feature, not a bug. Whatever entropy already exists in the seed is retained no matter how many mix() operations are performed.

**SEEDING**

Whatever the source, seeding always converts its input into a 32 or 64-bit int. (Same int type as the seed table.) Seeding 64-bit values into 32-bit wide seeds performs the low half as a 32-bit int first then seeds the high half as another 32-bit int.

Call the possibly converted seed item sVal.

seed[0] += rol(sVal,1);

seed[1] += rol(sVal,12);

seed[2] += rol(sVal,21);

mix();

The different rotates are designed to smear byte-oriented consistencies in seeding data.

Ignoring the rol in mix() and its wraparound, mix() contains

seed[3]+=seed[0]; seed[6]+=seed[3]; seed[9]+=seed[6]; // 0,3,6,9

seed[4]+=seed[1]; seed[7]+=seed[4]; // 1,4,7

seed[5]+=seed[2]; seed[8]+=seed[5]; // 2,5,8

Thus at least for a short time in mix() there are three separate phases. The sVal additions are performed in each of the phases. The rotates spread the sVal bits around while at the same time helping to smear out any byte-oriented consistencies in the seeding data.

Seeding 8 or 16-bit values performs sVal+=sVal<<5 twice before modifying the first 3 seed words. On some processors, sVal\*=33\*33 may produce shorter or faster code.

**RETURNING RANDOM VALUES**

mix();

int indx = seed[9]&7;

int t0 = seed[indx+0];

int t1 = seed[indx+1];

int t2 = seed[indx+2];

t0 = rol(t0,1);

t1 = rol(t1,12);

t2^= t2>>11; t2^= t2<<17; t2^= t2>>13; //see notes at end

return (t0+t1)^t2);

**NON-1032 NOTES**

With 64-bit seed words, the (1, 12, 21) rotates become (1, 21, 42) in both seeding and returning values.

With 20 word seeds, mix() is extended in the obvious way ending with:

seed[19] += seed[16]; seed[19] ^= rol(seed[18],1);

seed[ 0] += seed[17]; seed[ 0] ^= rol(seed[19],1);

seed[ 1] += seed[18]; seed[ 1] ^= rol(seed[ 0],1);

seed[ 2] += seed[19]; seed[ 2] ^= rol(seed[ 1],1);

With 20-word seed words, the “&7” in returning random values becomes “&15”. In both 10 and 20 word versions indx is effectively random 0 to 7 or 0 to 15. The indx value is always derived from the last word in the seed table.

Given the same seeding and mix() operations the four forms return different random values.

**ENTRY POINTS** in the assembly code

Note: the assembly code is x64 assembled with ml64. (Sorry, no x86.) But code for many 32-bit ARM Cortex processors is available.

unsigned int buf1032[12]; init1032(buf1032);

unsigned int buf2032[22]; init2032(buf2032);

unsigned long long buf1064[11]; init1064(buf1064);

unsigned long long buf2064[21]; init2064(buf2064);

The various initxxyy(void\* b) functions actually address the space for a structure which the init functions create. That space consists of a pointer to the function list “fList” which is followed by seed[10] or seed[20]. On the x64, the function list (which provides polymorphic behavior) is a 64-bit value. For init1032() or init2032() on x64, two 32-bit words are needed for the fList address. Thus unsigned int size must be at least 12 or 22. Although the need for the extra word does not apply to ARM 32-bit code the extra single 32-bit word is wasted to be consistent.

Always the first parameter is the seed table. In these notes “b” is used for the address of the seed table.

reInit(b) *;setup mix tbl again*

*; mix tbl is zeroed then seeded with 0x4321*

mix(b, n=10) *;do a mix n times – default value is only for C++*

*;*

*; RETURN RANDOM VALUES*

random32(b) *;returns random 32-bit value*

random64(b) *;returns random 64-bit value*

*; RETURN RANDOM VALUES into RAM...*

randomList32(b, &dest, count) *;returns random into 32-bit words*

randomList64(b, &dest, count) *;returns random into 64-bit words*

randomList (b, &dest, byteCount)

*; The above stores random<seed type> then bytes as needed*

*;*

*; the seedings...*

seedSelfvv(b) *;seed tbl with internally generated Random item*

seedItem64(b, item) *;seed tbl with 64-bit value*

seedItem32(b, item) *;seed tbl with 32-bit value*

seedItem16(b, item) *;seed tbl with 16-bit value*

seedItem8 (b, item) *;seed tbl with 8-bit value*

seedByte (b, item) *;exactly the same as above*

seedChar (b, item) *;exactly the same as above*

*;*

*; seeding from memory...*

*; (contrary to some early docx there is no extra mix() after seeding from memory)*

seedCstring(&s, &item)*;seed char’s with null terminated C-string*

seedList64 (&s, &item, count) *;seed with array of 64-bit items*

seedList32 (&s, &item, count) *;seed with array of 32-bit items*

seedList16 (&s, &item, count) *;seed with array of 16-bit items*

seedList8 (&s, &item, count) *;seed with array of 8-bit items*

seedList (&s, &item, count) *;seed byte count w/ best method*

*; when wrapped as a C++ object the last entry above*

*; is called like obj->SeedList(&item, count);*

**MISCELLANEOUS RAMBLINGS**

Other that the necessary memory access (assume briefly that mix() is performed entirely in registers) mix() could be implemented in many ARM processors in only 20 instructions.

This PRNG is not a speed demon. In the designer’s hand-coded assembly the 10-word mix() requires 13 memory fetches and 10 memory stores. It would be the rare optimizer that could do as well. Using the “randomList” forms many CPUs would assess the seed words almost entirely from cache RAM.

The algorithm designer speculates that pbRand1032 is as good as SHA256 and that pbRand2032 is likely better than SHA384. (Or possibly even SHA512.) When seeding with int arrays the pbRand forms are likely somewhat faster. The pbRand forms have the charm of being able to extract additional related randomness following the “SHA...” usage. And for memory-constrained systems, doing so using somewhat less RAM.

It might be almost as good to replace…

seed[3] += seed[0]; seed[3] ^= rol(seed[2],1);

…with…

seed[3] += rol(seed[0],1);

…used with the seed indexes modulo 10 as in the almost-the-mix shown earlier in this document. However, the same number of memory accesses would be necessary even with hand-optimized code.

There is a compile/assembly option, USE\_XOR\_SHIFT, which if used adds a little extra processor time to random number generation. In the C++ code it replaces a single rotate with **t2^= t2>>11; t2^= t2<<17; t2^= t2>>13;** This is one of 648 different xorshifts given in Dr. George Marsaglia’s “Xorshift RNGs”. All given xorshift parameters map one unsigned 32-bit value into another value with good randomness (except zero which remains unchanged). It is used here as an option to heap additional frustration upon attempts to derive the seed table from multiple sequential returned values. For 64-bit seed table words replace (11, 17, 13) with (15, 13, 28)

In many ARM processors, the xorshifts replace (the original design of) one rotate with three instructions. On Intel x64 three of (copy, shift, xor) is needed. For 64-bit seed words over 2000 xorshift PRNG mappings of 64-bit values exist.