Paper Number:	P2075R3
Title:	Philox as an extension of the C++ RNG engines
Authors:	Ilya Burylov <burylov@gmail.com> (Nvidia) Ruslan Arutyunyan <ruslan.arutyunyan@intel.com> (Intel) Andrey Nikolaev <af.nikolaev@gmail.com> Alina Elizarova <alina.elizarova@intel.com> (Intel) Pavel Dyakov <pavel.dyakov@intel.com> (Intel)
Contributors:	John Salmon
Audience:	LEWG
Date:	2023-10-13

1. Introduction

C++11 introduced a comprehensive mechanism to manage the generation of random numbers in the <random> header file (including distributions, pseudo random and non-deterministic engines).

We proposed a set of engine candidates for the C++ standard extension in the P1932R0 paper [1]. This paper is focused on the family of the counter-based Philox engines.

We propose 2 possible API approaches and seek feedback from the committee on which path is preferable.

2. Revision History

Key changes compared with R2 (reviewed at telecon 2023-09-26 in LEWG):

- Dropped several aliases for simplification.
- changed API for std::counter to control number of values passed
- extended design consideration section to discuss set_counter and get_counter functionality and API
- added explicit mention of little_endian notation for interpretation of the words passed to set_counterin order to keep code portability
- extended pseudo_random_function concept with counter_size parameter to avoid ambiguous logic of calculation of this value, previously counter_size assumed to be equal to output_count, which is not necessarily the case in all cases

Key changes compared with R1 (reviewed at telecon 2022-05-22 in SG6):

- Wording for the Philox-focused API was simplified.
- Wording for the counter_based_engine based API was extended.
- Design considerations section was added.
- set_counter() member function was added to the engine.
- c template parameter was removed for the sake of ease of use.

Key changes compared with R0 (reviewed in Prague in SG6):

- Aligned wording for philox_engine with the C++ standard.
- Added an alternative API with a std::array template parameter. Removed alternative APIs with calculated constant values.

- Added an alternative approach with a generic counter_based_engine and a specific philox_prf pseudo-random function.

3. Motivation

Random number generators (engines) are at the heart of Monte Carlo simulations used in many applications such as physics simulations, finance, statistical sampling, cryptography, noise generation and others.

Each of the C++11 random number generators has own advantages and disadvantages, e.g. linear congruential generators, the simplest generators with 32-bit state, has a quite short generation period (2^32) and weak statistical properties, while Mersenne Twister 19937 generator has long generation period and strong statistical properties, but has a large vector state that affects efficiency of parallelism in Monte Carlo simulations.

Several new algorithms were introduced in the last decade, which can utilize modern hardware parallelism and provide solid statistical properties.

4. General Description

Philox is one of the counter-based engines introduced in 2011 in [2]. All counter-based engines have a small state (e.g., Philox4x32 has 10 x 32-bit elements in its state) and a long period (e.g., the period of Philox4x32 is 2^{130}). Counter-based engines effectively support parallel simulations via both block-splitting and independent-stream techniques and many of them (including Philox) are well-suited to a wide variety of hardware including CPU/GPU/FPGA/etc.

Philox is proposed as the first new engine since C++11 for standardization. It satisfies the following criteria, as discussed in P1932R0 [1]):

- **Statistical properties.** The original paper asserted that the Philox family passes rigorous statistical tests including hundreds of different invocations of TestU01's BigCrush [2]. This statement has been independently confirmed: the TestU01 batteries for Philox4x32-10 and Philox4x32-7 were tested in [4] and DieHard testing results for Philox4x32-10 were published in the Intel® Math Kernel Library (Intel® MKL) documentation [5].
- **Wide usage.** Philox is broadly used in Monte-Carlo simulations which require massively parallel random number generation, e.g., financial simulations [6], simulation of non-deterministic finite automata [7], etc.
- **HW friendliness.** Philox's distinguishing features are its small state and reliance on simple primitive operations. It is, therefore, easy to vectorize and parallelize.

The value of the Philox engine is widely recognized by various vendors, to name a few:

- Intel* provides implementation of Philox4x32-10 as part of Intel® oneAPI Math Kernel Library.
- Nvidia* provides implementation of Philox4x32-10 as part of cuRAND library.
- AMD* provides implementation of Philox4x32-10 as part of rocRAND project.
- MathWorks* provides GPU-optimized implementation of Philox4x32-10 as part of their product.
- Microsoft* is using Philox4x32-10 as part of DirectML project.
- numPy provides implementation of Philox4x64-10.
- cuPy provides implementation of Philox4x32-10.

5. High-level API Design

Two approaches to an API definition are considered:

1. A philox-focused API defines a self-contained engine class template analogous to the other random number engines in the standard. (This is an evolution of the R0 version of this paper).

2. A counter-based-engine API, which is more generic and allows the creation of engines based on other pseudo-random functions as well.

Current authors support the 1-st approach for its simplicity and consistency with existing engines. It helps avoid confusion for novice users by introducing fewer entities to deal with.

2-nd approach is targeted towards more advanced users and is especially useful for people with expertise to implement a custom algorithm for counter_based_engine missing from the standard for some reason.

New engine introduces a dedicated function .set_counter() to set the state to arbitrary position, which enables support of parallel simulations and is a trivial operation for all counter based engines. Its use cases are described in the Design considerations section.

6. Philox-Focused API and Wording

This section describes the 1st of two approaches.

This API specifies a single, new philox_engine class template.

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType ...consts>
class philox_engine;
```

The philox_engine is described in terms of the Philox function which acts as a keyed bijection on a domain of size 2^{W*N}. Consequently, the philoxNxW engines have a period of $N*2^{W*N}$.

Pre-defined aliases are provided for instantiations with constants and parameters that are known to produce high-quality random numbers.

The philoxNxW aliases have a pre-defined round-count, r=10, that is somewhat larger than the minimum required to pass known statistical tests, but is widely used in practice. In other words, they provide a statistical safety margin at a modest performance cost.

```
using philox4x32 = ...;
using philox4x64 = ...;
```

• Wording

The changes affect only section “26.6 Random number generation”.

• Changes in section 26.6 Random number generation

…

(5.3) – the operator mullo denotes the low half of the modular multiplication of a and b: $(a * b) mod\ 2^w$

(5.4) – the operator mulhi denotes the high half of the multiplication of a and b: ($\lfloor (a * b)/2^w \rfloor$)

• Changes in sub-section 26.6.1 Header <random> synopsis

…

// 26.6.3.4 *class template* philox_engine

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r,
         UIntType ...consts>
    class philox_engine;
```

…

// 26.6.5 engines and engine adaptors with predefined parameters.

…

```
using philox4x32 = see below;

using philox4x64 = see below;
```

…

- New sub-section “26.6.3.4 Class template philox_engine”

26.6.3.4 Class template philox_engine

1 A `philox_engine` random number engine produces unsigned integer random numbers in the closed interval $[0,\ 2^w - 1]$, where the template parameter *w* defines the range of the produced numbers. The state x_i of a `philox_engine` object is of size *(5n/2+1)* and consists of a sequence *X* of *n* result_types, a sequence *K* of *n/2* result_types, a sequence *Y* of n result_types, and a scalar, *I*, the index of the next value to be returned by the GA from *Y*.

2 The generation algorithm GA(x_i) returns Y_I, the value stored in the I^{th} element of *Y*, in state x_{i+1}, i.e., **after** applying the transition algorithm: $x_{i+1} = TA(x_i)$.

3 The state transition algorithm, TA, is performed as follows:

```
I=I+1
if(I == n){
    Y = Philox(K, X) // see below
    X = (X+1)
    I = 0
}
```

where X behaves as if it is a `n*w`-bit integer

4 The Philox function maps the n/2-length sequence K and the n-length sequence X into an n-length output sequence. Philox applies an *R*-round substitution-permutation network to the values in X. A single round of the generation algorithm performs the following steps:

(4.1) – The output sequence X’ of the previous round (*X* in case of the first round) is permuted to obtain the intermediate state V:

$$V_j = X'_{f(j)}$$

where $j = 0,\ \ldots, n - 1$ and $f(j)$ is defined in Table 1 below, as in [2, 9]:

Table 1. Values for the word permutation f(j)

		j=															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n =	2	0	1														
	4	0	3	2	1												
	8	2	1	4	7	6	5	0	3								
	16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

[*Note*: for *n=2* the sequence is not permuted]

(4.2) – The following computations are applied for the elements of the V sequence:

$$X'_{2*k} = mullo(V_{2*k+1},\ M_k)$$

$$X'_{2*k+1} = mulhi\left(V_{2*k+1},\ M_k\right)\ xor\ key_k^q\ xor\ V_{2*k}$$

where: k = 0 … n/2-1, q is the index of the round: q = $0 \ldots r-1$, key_k^q is the k[th] round key for round q, $key_k^q = \left(K_k + q * C_k\right) mod\ 2^w$, and M_k and C_k are constants (template parameters).

5 After r applications of the single-round function, Philox returns the value of X'.

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r,
         UIntType ...consts>
class philox_engine {
    // Exposition only
    static constexpr std::size_t array_size = n / 2;

public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr std::size_t word_size            = w;
    static constexpr std::size_t word_count           = n;
    static constexpr std::size_t round_count          = r;
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2ʷ - 1; }
    static constexpr result_type default_seed = 20111115u;

    // constructors and seeding functions
    philox_engine() : philox_engine(default_seed) {}
    explicit philox_engine(result_type value);
    template<class Sseq> explicit philox_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    void set_counter(const std::array<result_type, n>& counter);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

6 The template parameter `…consts` represents the M_k and C_k constants which are grouped as follows: $[M_0, C_0, M_1, C_1, M_2, C_2 \ldots M_{N/2-1}, C_{N/2-1}]$

7 The following relations shall hold: $(n == 2) \,||\, (n == 4) \,||\, (n == 8) \,||\, (n == 16)$, $0 < r, w <= numeric\_limits < UIntType >:: digits,$.

8 The textual representation of x_i consists of the values of $K_0, \ldots, K_{n/2-1}, X_0, \ldots, X_{n-1}, I$, in that order. Note that the stream extraction operator can reconstruct *Y* from *K* and *X*, as needed.

```
explicit philox_engine(result_type value);
```

9 *Effects:* Sets the K_0 element of sequence K to value. All elements of sequences X and K (except K_0) are set to 0. The value of *I* is set to n-1.

```
template<class Sseq> explicit philox_engine(Sseq& q);
```

10 *Effects*: With $W = [w/32]$ and *a* an array (or equivalent) of length $(n/2) * W$, invokes q.generate(a+0, a+n/2*W) and then iteratively for i=0, …, $n/2 - 1$, sets K_i to $\left(\sum_{j=0}^{W-1} a[i * W + j] * 2^{32*j}\right) mod 2^w$. All elements of sequence X are set to 0. The value of *I* is set to n-1.

```
void set_counter(const std::array<result_type, n>& counter);
```

11 *Effects:* Sets the X_i for i = 0, …, n elements of sequence X starting from words containing the most significant parts of the counter value in the descending order to values counter[i]. The value of *I* is set to n-1.

- Changes in sub-section 26.6.5 Engines and engine adaptors with predefined parameters

…

```
using philox4x32 = philox_engine<uint_fast32_t, 32, 4, 10, 0xD2511F53, 0x9E3779B9,
0xCD9E8D57, 0xBB67AE85>;
```

- *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox4x32` produces the value XXXXXXXXX

```
using philox4x64 = philox_engine<uint_fast64_t, 64, 4, 10, 0xD2E7470EE14C6C93,
0x9E3779B97F4A7C15, 0xCA5A826395121157, 0xBB67AE8584CAA73B>;
```

- *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox4x64` produces the value XXXXXXXXX

- Changes in [version.syn]

Add a predefined macro to [version.syn]:

```
#define __cpp_lib_philox_engine 202310L // also in <random>
```

7. Generic counter_based_engine API

An alternative specification divides the Philox engine into 2 entities:

- A `counter_based_engine` class template, which encapsulates the TA and GA described, but depends on a generic pseudo-random function template parameter to generate a random sequence.
- A pseudo-random function (prf), `philox_prf`, defined as a class template, which encapsulates the logic contained in the Philox function but not the transition algorithm (TA) or generation algorithm (GA).

Instantiations of counter_based_engine<philox_prf> result in engines with exactly the same properties as the philox_engines described in the previous section.

This approach requires slightly more standardized machinery, e.g., a `pseudo_random_function` concept to constrain the permissible values of the `counter_based_engine`'s template parameter, but it paves the way for a set of engines with desirable properties. For example, the Threefry engine mentioned in P1932R0 as a candidate for standardization and engines based on widely deployed pseudo-random functions such as SipHash [10] and Chacha [11] can be accommodated. This can be done either as part of extending the standard or programmers can implement new pseudo-random functions with desirable properties for specific purposes (perhaps trading quality or bit-width for speed or size), instantiate a `counter_based_engine` and gain access to the power of <random>.

1 Class template philox_prf

A pseudo-random function (PRF) is a stateless function-like class that returns an array of unsigned integer values when invoked with an array of unsigned integer values. The Philox function specified in the description of the TA in section 6 above is just such a function. For the counter-based API, it is hoisted out of the philox_engine and given an independent existence as a class template.

The `philox_prf` class template may be declared as follows:

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
         ...consts>
class philox_prf {
    // Exposition only
    static constexpr std::size_t key_count = n / 2;
public:
    // generic PRF characteristics: types, data and function members
    using input_value_type = UIntType;
    using output_value_type = UIntType;
    static constexpr std::size_t input_word_size = w;
    static constexpr std::size_t output_word_size = w;
    static constexpr std::size_t input_count  = 3 * key_count;
    static constexpr std::size_t output_count = n;
    static constexpr std::size_t counter_size = n;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }

    // Philox specific characteristics
    static constexpr std::size_t round_count  = r;
    static constexpr std::array<UIntType, key_count> multipliers;
    static constexpr std::array<UIntType, key_count> round_consts;

    // signature of generating function
    void operator()(std::span<input_value_type, input_count> input,
                    std::span<output_value_type, output_count> output);

};
```

The `philox_prf`'s member `operator()(std::span<input_value_type, input_count> input, std::span<output_value_type, output_count> output)` method acts as follows:

1. Copy exactly n/2 values from `input` into sequence K, as if by doing K_i =input[i]; for i in 0,…,n/2-1, in order.
2. Copy exactly n values from `input` into sequence X, as if by doing X_i = input[n/2 + i]; for i in 0,…,n-1, in order.
3. Perform the steps described above in Section VI for the Philox(K, X) function.
4. Copy exactly n values of the Philox function's final value of X' to `output`, as if by doing output[i] = X'_i; for i in 0,…,n-1, in order

The `philox_prf` has predefined aliases analogous to those of the Philox engine, above:

```
// PRF for 10-round Philox with output consisting of 4 32-bit words
using philox4x32_prf = philox_prf<uint_fast32_t, 32, 4, 10, 0xD2511F53, 0x9E3779B9,
0xCD9E8D57, 0xBB67AE85>;


// PRF for 10-round Philox with output consisting of 4 64-bit words
using philox4x64_prf = philox_prf<uint_fast64_t, 64, 4, 10, 0xD2E7470EE14C6C93,
0x9E3779B97F4A7C15, 0xCA5A826395121157, 0xBB67AE8584CAA73B>;
```

Pseudo-random functions are stateless, pure functions. So it makes no sense to state the value of the 10000th invocation. Instead, the standard will state the values returned by a specific invocation, e.g.,

With Z={0x243f6a8885a308d3, 0x13198a2e03707344, 0xa4093822299f31d0, 0x082efa98ec4e6c89, 0x452821e638d01377, 0xbe5466cf34e90c6c} , philox4x64_prf(Z) shall return an array containing:

{0xa528f45403e61d95, 0x38c72dbd566e9788, 0xa5a1610e72fd18b5, 0x57bd43b5e52b7fe6}

With Z={0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0}, philox4x32_prf(Z) shall return an array containing:

{0xd16cfe09, 0x94fdcceb, 0x5001e420, 0x24126ea1}

N.B. these values are from the known-answer-test “kat_vectors” in the reference implementation of Philox [8].

2 The pseudo_random_function concept

The `philox_prf` class template has a number of public constexpr values (input_count, output_count, input_word_size, output_word_size, counter_size), dependent class types (result_type) and static public member functions (min(), max()). These members are required of any class that is intended for use as a pseudo-random function by counter_based_engine and are formalized as a `pseudo_random_function` concept.

3 Class template counter_based_engine

Instantiations of the class template `counter_based_engine` satisfy the requirements of a *random number engine*. The result_type, the word_size, and min() and max() functions are obtained from the template parameter, `prf`, which is constrained to satisfy the requirements of a `pseudo_random_function`. The period of the resulting engine is thus prf::output_count * $2^{n*prf::word\_size}$.

The specifications here are very similar to those in the “philox-focused” API above, with only minor differences arising because various sequence lengths and constants are obtained from the prf template parameter.

4 Wording

The changes affect only section “26.6 Random number generation”.

- Changes in section 26.6 Random number generation

…

(5.3) – the operator mullo denotes the low half of the modular multiplication of a and b: $(a * b) mod\ 2^w$

(5.4) – the operator mulhi denotes the high half of the multiplication of a and b: ($\lfloor (a * b)/2^w \rfloor$)

- Changes in sub-section 26.6.1 Header <random> synopsis

…

```
// 26.6.3.x. pseudo_random_function concept
template <class Prf>
    concept pseudo_random_functon = see below;
```

…

```
// 26.6.x class template philox_prf
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r,
         UIntType ...consts>
    class philox_prf;
```

…

```
// 26.6.x pseudo random function with predefined parameters.
using philox4x32_prf = see below;
using philox4x64_prf = see below;
```

```
// 26.6.4 class template counter_based_engine
```

…

```
template<pseudo_random_function prf>
```

```
class counter_based_engine;
```
…

// 26.6.5 engines and engine adaptors with predefined parameters.

…

```
// Philox engine with 10 rounds
using philox4x32 = counter_based_engine<philox4x32_prf>;

// Philox engine with 10 rounds
using philox4x64 = counter_based_engine<philox4x64_prf>;
```
…

- New sub-section “26.6.x pseudo_random_function concept”

26.6.3.x Pseudo Random function requirements

1. A *pseudo random function* **prf** of type **Prf** is a functional object which contains generic PRF characteristics used in counter_based_engine.

```
template <class Prf>
concept pseudo_random_function =
    requires(Prf prf,
             span<typename Prf::input_value_type, Prf::input_count> in,
             span<typename Prf::output_value_type, Prf::output_count> out) {
    typename Prf::input_value_type;
    typename Prf::output_value_type;
    Prf::input_word_size;
    Prf::output_word_size;
    Prf::input_count;
    Prf::output_count;
    Prf::counter_size;
    { Prf::min() }
    ->same_as<typename Prf::output_value_type>;
    { Prf::max() }
    ->same_as<typename Prf::output_value_type>;
    prf(in, out);
};
```

- New sub-section “26.6.x Class template philox_prf”

26.6.x Class template philox_prf

1. A philox_prf is a stateless class describing an algorithm of Philox random number generator satisfying pseudo_random_function concept. It produces unsigned integer values of type result_type in the closed interval $[0,\ 2^{w} - 1]$ from a given engine’s state of input_count of input_value_type.
2. The template parameter *w* defines the range of the produced numbers. The *span<input_value_type, input_count> input* of *operator()* consists of a sequence X of n input_value_type and sequence K of n/2.
3. The generation algorithm GA(x_i) fills *span<ouput_value_type, output_count> output* with Y_i:

```
operator()(input(K, X), Y) // see below
```

4. The Philox function maps the n/2-length sequence K and the n-length sequence X into an n-length output sequence. Philox applies an *R*-round substitution-permutation network to the values in X. A single round of the generation algorithm performs the following steps:

(4.1) – The output sequence X' of the previous round (*X* in case of the first round) is permuted to obtain the intermediate state V:

$$V_j = X'_{f(j)}$$

where $j = 0, \ldots, n-1$ and $f(j)$ is defined in Table 1 below, as in [2, 9]:

Table 1. Values for the word permutation f(j)

		j=															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n =	2	0	1														
	4	0	3	2	1												
	8	2	1	4	7	6	5	0	3								
	16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

[*Note*: for *n=2* the sequence is not permuted]

(4.2) – The following computations are applied for the elements of the V sequence:

$$X'_{2*k} = mullo(V_{2*k+1}, M_k)$$

$$X'_{2*k+1} = mulhi\left(V_{2*k+1}, M_k\right) xor\ key_k^q\ xor\ V_{2*k}$$

where: k = 0 … n/2-1, q is the index of the round: q = $0 \ldots r-1$, key_k^q is the k[th] round key for round q, $key_k^q = \left(K_k + q * C_k\right) mod\ 2^w$, and M_k and C_k are constants (template parameters).

5. After r applications of the single-round function, Philox stores the value of X' in Y.

```
template<typename UIntType, std::size_t w, std::size_t n, std::size_t r, UIntType
         ...consts>
class philox_prf {
    // Exposition only
    static constexpr std::size_t key_count = n / 2;
public:
    // generic PRF characteristics: types, data and function members
    using input_value_type = UIntType;
    using output_value_type = UIntType;
    static constexpr std::size_t input_word_size = w;
    static constexpr std::size_t output_word_size = w;
    static constexpr std::size_t input_count  = 3 * key_count;
    static constexpr std::size_t output_count = n;
    static constexpr std::size_t counter_size = n;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }

    // Philox specific characteristics
    static constexpr std::size_t round_count  = r;
    static constexpr std::array<UIntType, key_count> multipliers;
    static constexpr std::array<UIntType, key_count> round_consts;

    // generic signature of generating function
    void operator()(std::span<input_value_type, input_count> input,
                    std::span<output_value_type, output_count> output);

};
```

6. The template parameter `…consts` represents the M_k and C_k constants which are grouped as follows: $[M_0, C_0, M_1, C_1, M_2, C_2 \ldots M_{N/2-1}, C_{N/2-1}]$

7. The following relations shall hold: $(n == 2) \,||\, (n == 4) \,||\, (n == 8) \,||\, (n == 16)$, $0 < r, w <= numeric\_limits < UIntType >:: digits,.$

```
void operator()(std::span<input_value_type, input_count> input,
                std::span<output_value_type, output_count> output);
```

8. Effects: fills output according to Philox algorithm (4) with state (K, X) given as an input.

- New sub-section 26.6.x.x Pseudo random functions with predefined parameters

```
using philox4x32_prf = philox_prf<std::uint_fast32_t, 32, 4, 10, 0xD2511F53,
0x9E3779B9, 0xCD9E8D57, 0xBB67AE85>;

using philox4x64_prf = philox_prf<std::uint_fast64_t, 64, 4, 10, 0xD2E7470EE14C6C93,
0x9E3779B97F4A7C15,0xCA5A826395121157, 0xBB67AE8584CAA73B>;
```

- New sub-section "26.6.3.4 Class template counter_based_engine"

26.6.3.4 Class template counter_based_engine

1 A `counter_based_engine` is a random number engine producing unsigned integer values of type result_type = prf::result_type in the closed interval $\left[0,\ 2^{prf::output\_word\_size} - 1\right]$. The state x_i of a `counter_based_engine` object is of size *(*prf::input_count + prf::output_count*+1)* and consists of a sequence *Z* of `prf::input_count prf::input_value_type`, a sequence *Y* of `prf::output_count result_types` and an index *I*, of the next value to be returned by the GA from *Y*. The sequence *Z* is treated as the concatenation of a sequence *K* of N_K=(`prf::input_count-prf::counter_size`) result_types, and a sequence *X* of N_X=(`prf::counter_size`) result_types. I.e.,

$Z = [K_0\ K_1 \ldots K_{Nk-1}\ X_0\ X_1 \ldots X_{Nx-1}]$.

In the descriptions that follow, assignments to elements of *X* and *K* are understood as assignments to the corresponding elements of *Z*.

2 The generation algorithm GA(x_i) returns Y_I, the value stored in the I^{th} element of *Y*, in state x_{i+1}, i.e., after applying the transition algorithm: x_{i+1} = TA(x_i).

3 The TA is:

```
I=I+1
If(I == prf::output_count){
    prf{}(Z, Y)// Z span is an input and Y is an span output
    X = (X+1)
    I = 0
}
```

where X behaves as if it is a `prf::output_count` * `prf::output_word_size`-bit integer.

4 The textual representation of x_i consists of the values of $Z_0, \ldots, Z_{prf::input\_count-1}$, and *I*, in that order. Note that the stream extraction operator can reconstruct *Y* from *Z*, as needed.

```
template<pseudo_random_function Prf>
class counter_based_engine {
    // Exposition only
public:
    // types
```

```
    using result_type = typename prf::output_value_type;

    // engine characteristics
    static constexpr std::size_t state_count = prf::input_count;
    static constexpr result_type min() { return prf::min(); }
    static constexpr result_type max() { return prf::max(); }
    static constexpr prf::input_value_type default_seed = 20111115u;

    // constructors and seeding functions
    counter_based_engine() : counter_based_engine(default_seed) {}
    explicit counter_based_engine(prf::input_value_type value);
    template<class Sseq> explicit counter_based_engine(Sseq& q);
    void seed(prf::input_value_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    void set_counter(const std::array<prf::input_value_type, prf::counter_size>&
counter);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

5 The template parameter `prf` represents class, which satisfies the *pseudo_random_function* concept

```
explicit counter_based_engine(prf::input_value_type value);
```

6 *Effects:* Sets the K_0 element of sequence K to value. All elements of sequences X and K (except K_0) are set to 0. The value of *I* is set to (prf::output_count-1).

```
template<class Sseq> explicit counter_based_engine(Sseq& q);
```

7 Effects: With $W = [w/32]$ and a an array (or equivalent) of length $N_K * W$, invokes q.generate(a+0, a+N_K*W) and then iteratively for i=0, ..., $N_K - 1$, sets K_i to $\left(\sum_{j=0}^{W-1} a[W * i + j] * 2^{32*j}\right) mod 2^w$. All elements of sequence X are set to 0. The value of *I* is set to (prf::output_count-1).

```
void set_counter(const std::array<prf::input_value_type, prf::counter_size>&
counter);
```

8 *Effects:* With m=prf::counter_size, sets the X_i for i = 0, …, m elements of sequence X starting from words containing the most significant parts of the counter value in the descending order to values counter[i]. The value of *I* is set to (prf::output_count-1).

- Changes in sub-section 26.6.5 Engines and engine adaptors with predefined parameters

…

```
// Philox engine with 10 rounds
using philox4x32 = counter_based_engine<philox4x32_prf>;

// Philox engine with 10 rounds
using philox4x64 = counter_based_engine<philox4x64_prf>;
```

- Changes in [version.syn]

Add a predefined macro to [version.syn]:

```
#define __cpp_lib_counter_based_engine 202310L // also in <random>
#define __cpp_lib_philox_prf           202310L // also in <random>
```

8 Design considerations

1 Compare approaches

Consistency with existing approaches:

1. A philox-focused API introduced a new engine in the same way as existing C++11 engines.
2. A counter-based-engine API approach introduced an additional new concept of a stateless pseudo-random function and defining communication protocol between two entities, which brings in new machinery.

The main reason for existence of the second approach is extendibility. A counter-based-engine approach allows the extension with a variety of counter-based generators which can be supported via the same API, such as Threefry, Siphash or other user-defined prf.

It goes for the price of extra complexity in describing the generic protocol between the counter_based_engine and prf. Because of this complexity different types of data, which is being by prf is abstracted in a single input_value sequence, which should be reinterpreted by prf implementation to split it into:

1. counter part (an entity which is monotonically increasing in time),
2. constant state part (which is filled with the seed sequence).

See paragraph 2 in *philox_prf* wording.

It should be additionally noted that some counter based engines have modifications in the algorithm of counting, e.g. SHISHUA algorithm [16] has non-unit step for the counter. Such algorithms were considered too exotic to generic facilities during the discussion in SG6.

Our prototype showed that the implementation of philox_engine has 268 LOC.
Implementation of the second approach took 317 LOC for counter_based_engine and pseudo_random_function concept + 130 LOC for philox_prf.
See [15] for both prototypes.

2 Span vs. range

Operator() of prf in the counter-based-engine API approach is the communication protocol between *counter_based_engine* class object and *pseudo_random_function* class object. In order to provide early diagnostic we introduce pseudo_random_function concept which checks for the valid operator() of prf.

We considered input range and output iterator as a more generic approach to provide more flexibility for Prf implementations and reuse scenarios. But it pollutes Prf concept with additional template parameters of the concept and affected *counter_base_engine* itself:

```
template<class Prf, class InputRange, class OutputIterator>
concept pseudo_random_function = std::input_range<InputRange> &&
std::sized_range<InputRange> && std::output_iterator<OutputIterator> &
   requires(Prf prf, InputRange range, OutputIterator o) {
    ...
    { prf(range, o) } -> std::output_iterator;
};

template< std::sized_range InputRange, std::output_iterator OutputIterator,
pseudo_random_function<InputRange, OutputIterator>, std::size_t c>
class counter_based_engine;
```

It complicates the usage of counter_based_engine to the level, which we did not consider adequate for the purpose of verifying the existence of proper operator() overload in Prf function.

With that we refactored protocol to use std::span, which is sufficient for use with counter_based_engine, but might not be too generic to use Prf for other hypothetical purposes:

```
void operator()(std::span<input_value_type, input_count> input,
std::span<output_value_type, output_count> output);
```

3 set_counter use case

The following example shows the typical flow for a Monte Carlo simulation of a large number of "atoms" for a large number of timesteps:

```
uint32_t global_seed = 999;
  for(uint32_t timestep  = 0; timestep < Ntimesteps; ++timestep){
      for(uint32_t atomid = 0; atomid < Natoms; ++atomid){
          philox4x32 eng(global_seed);
          eng.set_counter({atomid, timestep, 0, 0});
          normal_distribution nd;
          auto n1 = nd(eng);
          auto n2 = nd(eng);
          // ...
      }
  }
```

Using set_counter() allows creation of the engine on the fly without storing `Natoms` of states. In addition it does not prevent parallelisation of either of the loops.

On the down side, one should control the number of random numbers consumed per timestep per atom. If the number consumed numbers overcome $4*2^{32*2}$, then sequences in different atoms may overlap, which brings in undesired cross correlation. The following section discussed the way to avoid that.

Under certain limitations a similar effect can be achieved via using .discard() function, but it differs in several aspects. The most critical one:

- .discard() shifts are limited to *unsigned long long*, which on many systems is 64-bits integer, while `philox4x64` has a period of 4*2^(64*4), thus splitting this sequence in 2 parts would require 4*2^(64*3 - 64) calls of discard(), while .set_counter() can do the same in one call.

There are other differences:

1. .discard() shifts the counter only forward relative to its current position. This API exists because some (but not all) engines have efficient algorithms to move their state forward.
2. .set_counter() sets the absolute value for the counter. It is a unique property of counter-based engines - it is trivial to set their absolute state.

4 set_counter API

P2075R2 defined API as `void set_counter(std::initializer_list<result_type> counter)`, which allowed passing more than n (template argument of philox_engine class) required values, with excessive values silently ignored.

From the mathematical point counter is a single big integer described in wording sections as *X*. There is no existing facility in the standard, which can represent this entity in its mathematical sense. There is the paper P1889R1, which introduces `std::wide_integer<Bits,S>`. std::wide_integer may potentially be useful in the future in different aspects of random numbers facilities, but it is targeted to Numerics TS and is not specific to set_counter(), thus we decided to not rely on this facility.

We considered 4 options to fix this API:

1. `void set_counter(std::span<const UIntType, n> counter);`
    a. `obj.set_counter({{atomid, timestep, 0, 0}});`

The most common use case for set_counter functionality is a temporary object created on the fly and not a real container. The main use case requires double braces, which might be not syntactically friendly for average users. This API enforces exactly n values passed, which addresses the main concern of the API from revision of the paper.

Double braces will not go away with P2447R4, because the newly introduced constructor for std::span is explicit for cases with non dynamic_extent.

2. `void set_counter(const std::array<UIntType, n>& counter);`

    a. `obj.set_counter({atomid, timestep, 0, 0});`

This approach is slightly less generic - it fixes std::array as the only acceptable container. The corner case of this approach - it allows passing less variables in curly braces than required with assumed zero-tail. Passing less value can be considered reasonable, because the sequence is being split for parallelization, the list significant bits are left for consumption within the block of computations and can be zeroed.

    b. `obj.set_counter({atomid, timestep});`

Existing Philox aliases allow engines with n=4, which translates into 4 elements in the input array. But general facilities allow instantiation of the template with n up to 16. In this case, the way to avoid typing all of the 16 elements would be useful. That said, authors are not aware of n>4 usage in real user codes.

Thus in ninja use cases that approach is less verbose but is slightly more error prone.

3. `void set_counter(see below);`

    a. `obj.set_counter(atomid, timestep, 0, 0);`

It is assumed, that description would define the signature, which requires exactly n UIntType values to be passed to the function and that solves the main problem of APIs from previous revision of the paper. At the same time this approach is harder to understand from its specification and it loses perception that the counter is actually a single entity because of several arguments of the function

4. `void set_counter(const counter_t& counter);`

    a. `obj.set_counter({atomid, timestep, 0, 0});`

If counter_t covers representation of the counter entity in its aspects needed for set_counter, then it becomes a thin wrapper over std::array. If it is extended to cover additional manipulations with the counter it represents, then it needs additional arithmetic operations and becomes a thin wrapper around non-existing std::wide_integer. In both cases we see no strong reason to introduce a new type.

Within the provided set of cases, we decided to stick to:

```
void set_counter(const std::array<UIntType, n> counter);
```

It is less generic than std::span, but avoids verbosity of additional braces. When/if std::wide_integer becomes the standard, additional overload for set_counter can potentially be introduced if considered necessary.

5 get_counter member function

get_counter() function is a natural counterpart for the introduced set_counter() function. But it is important to note, that there are several gotchas, which makes it less convenient, than one may consider:

1. Counter is not fully represent the state of the engine. Looking into Wording section, especially the description of the transition algorithm TA, one can see that the counter is being ticked only every n-th invocation of operator() of the engine. This happens because the Philox algorithm

generates a batch of n values each time, thus we have a buffer *Y*, which stores n-1 values to return before the next run of the Philox algorithm. In order to fully restore the state of the Philox, one should restore the counter, buffer *Y* and the position *I* inside this buffer.

2. set_counter is most useful, when we statically divide the whole period of the Philox generator into several sub-sequences and further do the whole computation within these subsequences. That approach does not require checking the current state of the engine.
3. Let us consider dynamic parallelism, when we get the engine in some unknown state and plan to parallelise some amount of work. This use case is covered by existing .discard() functionality, where engine state is being shifted in relative manner. If one wants to reproduce that scenario with set/get_counter(), additional arithmetics should be introduced for counter type, which would require efforts comparable with introduction of std::wide_integer<> discussed in section about set_counter API.

Within the provided set of considerations, we decided to not introduce get_counter(). One can return to this question when additional use cases are found and/or std::wide_integer<> be accepted for the standard.

6 Splitting sequence in sub sequences

P2075R1 revision of this paper had *c* template argument for *counter_based_engine*:

```
template<pseudo_random_function prf, size_t c>
class counter_based_engine.
```

The main purpose of this parameter was to split counter X into lower *c* words X1 and higher n-c words X2. X1 behaves as a normal counter and wraps when depleted. X2 is predefined by the user and is considered constant by the algorithm.

The intention of this parameter was to add a simple way to split a full sequence of random numbers into independent (n-c)*word_size subsequences, which can be used for parallelisation and easy creation of such subsequences on the flight.

Further analysis revealed that this concept can be applicable for a wider set of engines, which makes *c* parameter on the level of the engine not generic enough.

As a further design consideration for this methodology we propose to consider a dedicated additional adapter, such as:

```
template<template Engine, size_t c>
class subsequence_engine;
```

This adaptor can be customized for a subset of engines where dedicated optimizations are possible.

Further investigations for this adaptor can be done in a separate paper. Authors removed the *c* parameter from this revision.

7 Using std::array in template arguments

The template parameter `consts` as a std::array was considered.

```
// ********************************************************************
// Alternative API: consts template parameter represented as std::array
// ********************************************************************

template<typename UIntType, std::size_t w, std::size_t n, std::size_t r,
std::array<UIntType, n> consts>
class philox_engine {
    static constexpr std::size_t array_size = n / 2; // Exposition only

public:
…
```

```
    static constexpr std::array<result_type, array_size> multipliers;
    static constexpr std::array<result_type, array_size> round_consts;
…
```

philox_engine template class is not expected to be frequently used by users - predefined aliases are the main way to use this engine. Having that in mind, we decided to not introduce a new API technique into standard for a minor simplification.

8 additional aliases philox2x32 and philox2x64

Original paper contained additional aliases:

```
template<size_t r>
using philox2x32_r = philox_engine<uint_fast32_t, 32, 2, r, 0xD2511F53, 0x9E3779B9>;
```

1 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox2x32_r<10>` produces the value XXXXXXXXX

```
template<size_t r>
using philox2x64_r = philox_engine<uint_fast64_t, 64, 2, r, 0xD2B74407B1CE6E93,
0x9E3779B97F4A7C15>;
```

2 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `philox2x64_r<10>` produces the value XXXXXXXXX

```
using philox2x32 = philox2x32_r<10>;

using philox2x64 = philox2x64_r<10>;
```

`philox4x32` and `philox4x64` define the most broadly used Philox parameter sets (supported in Intel® MKL, rocRAND, cuRAND, MATLAB, etc.).

`philox2x32` and `philox2x64` show good statistical properties and performance as well [8], but they are not broadly used across libraries.

Having two sets of aliases defined in the standard will complicate the choice and we decided to stick with the current consensus across the libraries by removing `philox2x32` and `philox2x64`.

P2075R2 paper revision contained additional aliases, which allowed setting the number of rounds of the algorithm:

```
template<std::size_t r>
using philox4x32_r<r> = see below;

template<std::size_t r>
using philox4x64_r<r> = see below;
```

Theoretically the philoxNxW_r<r> permits the program to trade speed for safety by specifying a number of rounds of mixing. Philox generators with r=7 have no known statistical flaws [2].

But in practice only rounds equal to 10 are widely used and implemented in a variety of the libraries. Taking that into consideration we simplified the user's choice by removing these additional aliases. The experts, who seek for finer control over statistical properties, can define such aliases in their code.

9 Counter X and its sequence representation

Counter X behaves as a big integer but is represented in some parts of the algorithm description and in set_counter() function as a sequence of values X_i. Users need to understand which words in this sequence represent the higher order parts of the mathematical X value in order to use the set_counter() function in a meaningful way (see section set_counter use case). That controls, whether they should write:

```
eng.set_counter({atomid, timestep, 0, 0});
```

or

```
eng.set_counter({0, 0, timestep, atomid});
```

for the Philox period to be evenly split between all computational blocks of the program.

In order to define the API but leave implementation details in hands of implementers, we specy the order of this only in the description of set_counter() function behavior.

9 Impact on the Standard

This is a library-only extension. It adds one or two new class templates, zero or one new concepts, and a small number of pre-defined template aliases.

10 References

1 P1932R0 “Extension of the C++ random number generators”: http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1932r0.pdf.

2 John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’11, pages 16:1–16:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0

3 L’Ecuyer, Pierre & Simard, Richard. (2007). A Software Library in ANSI C for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software - TOMS.

4 Manssen, Markus & Weigel, Martin & Hartmann, Alexander. (2012). Random number generators for massively parallel simulations on GPU. The European Physical Journal Special Topics. 210. 10.1140/epjst/e2012-01637-8.

5 Notes for Intel® Math Kernel Library (Intel® MKL) Vector Statistics : https://software.intel.com/en-us/mkl-vsnotes-philox4x32-10

6 Xu, Linlin & Ökten, Giray. (2014). High Performance Financial Simulation Using Randomized Quasi-Monte Carlo Methods. Quantitative Finance. 15. 10.1080/14697688.2015.1032549.

7 Wadden, Jack & Brunelle, Nathan & Wang, Ke & El-Hadedy, Mohamed & Robins, G. & Stan, Mircea & Skadron, Kevin. (2016). Generating efficient and high-quality pseudo-random behavior on Automata Processors. 622-629. 10.1109/ICCD.2016.7753349.

8 Random123 D. E. Shaw Research ("DESRES"): http://www.deshawresearch.com/resources_random123.html

9 N. Ferguson, S. Lucks, B. Schneier, B. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. http://www.schneier.com/skein.pdf, 2010.

10 J-P Aumasson and D. J. Bernstein. (2012). “SipHash: a fast short-input PRF”, https://131002.net/siphash/

11 Y. Nir and A. Langley. (2018). “ChaCha20 and Poly1305 for IETF Protocols”, https://tools.ietf.org/html/rfc8439

12 P1068R2 “Vector API for random number generation”: http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1068r2.pdf.

13 P1932R3 “Vector API for random number generation”: http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1068r3.pdf.

14 John Salmon’s github: https://github.com/johnsalmon/cpp-counter-based-engine

15 Alina Elizarova’s github: https://github.com/aelizaro/cpp-counter-based-engine/tree/alignment_with_proposal

16 SHISHUA: The Fastest Pseudo-Random Generator In the World https://espadrine.github.io/blog/posts/shishua-the-fastest-prng-in-the-world.html