I. Introduction

C++11 introduced a comprehensive mechanism to manage 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 P1932R0 paper [1]. Current paper is focused on the family of the counter-based Philox engines.

II. Motivation

See P1932R0 [1] for motivation.

III. General Description

Philox engine is one of the counter-based engines which were introduced in 2011 in [2] for the first time. All counter-based engines have a small state (e.g. Philox4x32-10 has 6 x 32-bits elements in state) and long period (e.g. period of Philox4x32-10 is 2^130). This family effectively supports parallel simulations via block-splitting techniques and enable a broad HW spectrum including CPU/GPU/FPGA/etc.

Philox engine was chosen as an extension of the list of C++ random number engines based on the following (criteria proposed in P1932R0 [1]):

- **Statistical properties.** Authors of the counter-based engines took crypto-algorithm as the reference for Philox and claimed that Philox family passes rigorous statistical tests including TestU01’s BigCrush [2]. This statement was independently verified by the different authors, e.g.: TestU01 batteries for Philox4x32-10 and Philox4x32-7 were tested in [4], DieHard testing results for Philox4x32-10 were published as part of Intel® Math Kernel Library (Intel® MKL) documentation in [5].
- **Usage scenarios.** Philox is broadly used in Monte-Carlo simulations which require massively parallel random number generation (e.g. Philox in financial simulations [6], high-quality pseudo-random behavior simulation [7], etc.).
- **HW friend-ness.** Philox engine can be easily vectorized and parallelized on CPU, for example Intel® MKL provides highly vectorized version of Philox4x32-10. Philox is proven to work on GPU – it’s implemented in the GPU-optimized Nvidia and AMD libraries: cuRand and rocRand.

IV. Algorithm Details

Detailed description of the Philox engine can be found in [2].

Philox (Philox-\(n \times w \cdot r\) engine relies on substitution-permutation network (SP-network). SP-network consists of S-boxes and P-boxes responsible for producing highly diffusive bijection and permutations respectively. A state of the Philox contains \(n\) words of size \(w\) and \(n/2\) keys which are used to produce round-keys for each of the \(r\)-rounds (see Figure 1 for 1-round illustration).
Each S-box has 2 elements as input (see Figure 2) and performs next computation:

**Equation 1.**

\[
\begin{align*}
L'_k &= \text{mullo}(R_k, M_k) \\
R'_k &= \text{mulhi}(R_k, M_k) \oplus \text{key}_k^i \oplus L_k
\end{align*}
\]

Round-keys \(\text{key}_k^i\) are generated by using:

**Equation 2.**

\[\text{key}_{k+1}^i = \text{key}_k^i + C_k\]

where:

- \(i\) – index of round
- \(k\) – index of S-box
- \(L_i/L'_i\) – the first input/output value
- \(R_i/R'_i\) – the second input/output value
- \(\text{key}_k^i\) – round key, specific for S-box and round
- \(\text{key}_0^i\) – initial key from the engine state
- \(M_k\) – multiplier, specific S-box constant
- \(C_k\) – round constant, specific for S-box
- mullo - the low half of the product \((a \ast b) \ mod \ 2^w\)
- mulhi – the high half of the product \(\lfloor (a \ast b)/2^w \rfloor \)
- \(\oplus\) - bitwise XOR operator

For \(n = 2\), the Philox-2 \(\times w\)-\(r\) performs \(r\) rounds of the Philox S-box on a pair of \(w\)-bit inputs. For larger \(n\), the inputs are permuted using the Threefish \(n\)-word P-box before being fed, two-at-a-time, into \(n/2\) Philox S-boxes [2]. P-box of Threefish [9] is represented in Table 1:
Table 1. P-box of Threefish algorithm. Indexes of output words

| n | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 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  | 1  | 8  | 1  |

Authors of Philox engine recommend next algorithm’s parameters ([2], [8]):

- \( n \) equals \( 2; 4; 8; 16 \)
- \( w \) equals to 32 or 64
- \( M \) satisfies “avalanche criterion” (any single-bit change in the input should result (on average) in a 0.5 probability change in each output bit)
- \( C \) is selected based on crush-resistance testing
- \( r \) is greater than or equal to 8

We propose API with broader algorithm parameters to support possible modifications of Philox engine.

V. Proposed API

We propose to add Philox to the C++ standard as the philox_engine engines’ family with several instantiations: philox4x32x10, philox4x64x10.

Class template philox_engine

philox_engine is a counter-based random number engine described in [2]. It produces high quality unsigned integer random numbers of type UIntType in the closed interval \([0, 2^w-1]\). The state of philox_engine object is of size \((n+n/2)\) contains \(n\) words and \(n/2\) keys of size \(w\) both.

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

    public:
        // types
        typedef UIntType result_type;

        // 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;

        // constructors and seeding functions
        ...

        // generation functions
        ...
};
```

The following relations shall hold: \((n == 2) \lor (n == 4) \lor (n == 8) \lor (n == 16), 0 < r, w = numeric_limits<UIntType>::digits, n == size_of...(consts)\).
The following type aliases define the random number engine with two commonly used parameters sets:

**Table 2. Proposed philox_engine instantiations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>philox4x32x10</td>
<td>using philox4x32x10 = philox_engine&lt;uint32_t, 4, 10, 0x2511F53, 0x9B37F53, 0x9E3779B9, 0xCDA8D57, 0xBB57AEB85&gt;; 4 32-bits words algorithm with 10 rounds</td>
</tr>
<tr>
<td>philox4x64x10</td>
<td>using philox4x64x10 = philox_engine&lt;uint64_t, 4, 10, 0x2DE7470EE14C6C93, 0x9E3779B97F4A7C15, 0xCA5F826395121157, 0xBB57AEB8584CAA73B&gt;; 4 64-bits words algorithm with 10 rounds</td>
</tr>
</tbody>
</table>

Other possible options:

**Table 3. Other possible philox_engine instantiations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>philox2x32x10</td>
<td>using philox2x32x10 = philox_engine&lt;uint32_t, 2, 10, 0x256d193, 0x9B37F53&gt;; 2 32-bits words algorithm with 10 rounds</td>
</tr>
<tr>
<td>philox2x64x10</td>
<td>using philox2x64x10 = philox_engine&lt;uint64_t, 2, 10, 0x2DE7470EE14C6C93, 0x9E3779B97F4A7C15&gt;; 2 64-bits words algorithm with 10 rounds</td>
</tr>
</tbody>
</table>

philox2x32x10 and philox2x64x10 do not appear to be broadly-used but still show good statistical properties and performance [8].

**philox_engine template parameters and members description are represented below:**

**Table 4. philox_engine template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIntType</td>
<td>One of types: unsigned short, unsigned int, unsigned long, or unsigned long.</td>
</tr>
<tr>
<td>n</td>
<td>The number of words in the internal engine state, equals to the number of values produced by the one generation loop</td>
</tr>
<tr>
<td>w</td>
<td>The word size</td>
</tr>
<tr>
<td>r</td>
<td>The number of rounds in the one generation loop</td>
</tr>
<tr>
<td>...consts</td>
<td>Constants that are used in the algorithm (see Equation 1 and 2). The constants are grouped per S-box (M, C) where M is a multiplier constant, C is a round constant. The constants are set for each S-box one after another: [M₀, C₀, M₁, C₁, M₂, C₂ ... Mₙ₋₁/₂, Cₙ₋₁/₂]</td>
</tr>
</tbody>
</table>

**Table 5. philox_engine members description**

<table>
<thead>
<tr>
<th>Type</th>
<th>Member object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static constexpr std::size_t</td>
<td>word_size</td>
<td>The template parameter w, determines the range of values generated by the engine</td>
</tr>
<tr>
<td>static constexpr std::size_t</td>
<td>word_count</td>
<td>The template parameter n, determines the number of words in the engine state</td>
</tr>
<tr>
<td>static constexpr std::size_t</td>
<td>round_count</td>
<td>The template parameter r, determines the number of rounds in the Philox algorithm</td>
</tr>
<tr>
<td>static constexpr std::array</td>
<td>multipliers</td>
<td>Contains the M, elements of the template parameter ...consts</td>
</tr>
<tr>
<td>std::array&lt;UIntType, array_size&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>static constexpr std::array</td>
<td>round_consts</td>
<td>Contains the C, elements of the template parameter ...consts</td>
</tr>
<tr>
<td>std::array&lt;UIntType, array_size&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VI. Possible Alternative APIs

Template parameter \( w \) from the API described in Section V can be deduced from \( \text{UIntType} \) however this approach is inconsistent with the other existing C++ engines.

```cpp
// **************************************************
// Alternative API I: \( w \) template parameter is deduced
// **************************************************

template<typename \( \text{UIntType} \), std::size_t \( n \), std::size_t \( r \), \( \text{UIntType} \) ...\( \text{consts} \)>
class philox_engine {
    static constexpr std::size_t array_size = \( n \) / 2; // Exposition only

public:
    // types
    typedef \( \text{UIntType} \) result_type;

    // engine characteristics
    static constexpr std::size_t word_size = numeric_limits<\( \text{UIntType} \)>::digits;
    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;

    // constructors and seeding functions
    ...

    // generation functions
    ...
};
```

Template parameter \( n \) can also be deduced from the size of the variadic template \( ...\text{consts} \) but it makes the API less clear for the users.

```cpp
// **************************************************
// Alternative API II: \( w \) and \( n \) template parameters are deduced
// **************************************************

template<typename \( \text{UIntType} \), std::size_t \( r \), \( \text{UIntType} \) ...\( \text{consts} \)>
class philox_engine {
    static constexpr std::size_t array_size = sizeof...(\( \text{consts} \)) / 2;

public:
    // types
    typedef \( \text{UIntType} \) result_type;

    // engine characteristics
    static constexpr std::size_t word_size = numeric_limits<\( \text{UIntType} \)>::digits;
    static constexpr std::size_t word_count = sizeof...(\( \text{consts} \));
    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;

    // constructors and seeding functions
    ...

    // generation functions
    ...
};
```

VII. Impact on the Standard

This is a library-only extension. It adds new engine class template and commonly used instantiations.

VIII. References


