The OCELOT Stream Ciphering Method (Version 2.1.0) - Design and Security Considerations, and Comments to C++ Reference Code

by George Anescu, December 2010
george.anescu@sc-gen.com, www.sc-gen.com

1. Introduction

In cryptography a stream cipher is a symmetric key cipher (i.e. the same key data is used for both encryption and decryption) where the plaintext bits are combined, typically by an XOR (exclusive or) operation, with a bit stream (or keystream) generated by a Pseudo Random Number Generator (PRNG). Due to the symmetric properties of the XOR operation, the decryption is performed identically to the encryption. If the keystream changes independently of the plaintext or ciphertext messages, the stream cipher is classified as a synchronous stream cipher, and it is basically reduced to a PRNG. The PRNGs used in cryptography need good cryptographic properties from the security perspective. It means that an attacker should not be able to easily predict the future values of the keystream based on a known current keystream history of any length.

My stream ciphering method OCELOT (with two variants, OCELOT1 and OCELOT2) is an attempt to propose good cryptographic PRNGs with improved security and additional useful features.

1.1 Design Goals and Characteristics

Simplicity and Mathematical elegance;

Scalability - the capability to work with an internal state of any size using the same code implementation. The size granularity can be at byte level or at word level.

Extensibility - the simple XOR and arithmetic modulo operations at byte or word level can be naturally extended for increased security to latin squares (also known as quasigroups) operations having the dimensions 256×256;

Key data flexibility - the capability to work with key data of any size (the larger is the key data size the better is the security);

Good Control over PRNG's period in order to generate sequences of random data as large as desired;

Efficiency - Staring with Version 2.0.0 OCELOT works at word level with the computing speed almost doubled compared to byte level versions.

Resistence - to clasical and more modern security attacks;

1.2 Limitations

Security Holes - although I designed everything having security in mind, the methods are new, never before published and not extensively studied;

2. Design Considerations

The OCELOT stream ciphering method is inspired by RC4 stream ciphering method (also known as ARC4 or ARCFOUR meaning Alleged RC4). RC4 was designed by Ron Rivest of RSA Security in 1987 and initially was a trade secret, but in September 1994 a description of it was anonymously posted to the Cypherpunks mailing list. It was soon posted on the *sci.crypt* newsgroup, and from there to many sites on the Internet. The leaked code was confirmed to be genuine as its output was found to match that of proprietary software using licensed RC4. Because the algorithm is known, it is no longer a trade secret. The name "RC4" is trademarked, so RC4 is often referred to as "ARCFOUR" or "ARC4" (meaning Alleged RC4) to avoid trademark problems. Presently RC4 is considered unsecure due to attacks developed mainly against the Key-Scheduling Algorithm (KSA). Unlike the modern stream ciphers, RC4 does not take a separate nonce (number used once) alongside the key. This means that if a single long-term key is to be used to securely encrypt multiple streams, the cryptosystem must specify how to combine the nonce and the long-term key to generate the stream key for RC4. One approach to addressing this is to generate a "fresh" RC4 key by hashing a long-term key with a nonce. However, many applications that use RC4 simply concatenate the key and the nonce. RC4's weak Key-Scheduling Algorithm then gives rise to a variety of serious problems.

The main idea taken by OCELOT from RC4 is to use in the internal state of the stream cipher a dynamic permutation of the of the numbers from 0 to 255. In OCELOT the internal state is extended and the dynamic permutation is also used in order to isolate the other parts of the internal state from the keystream output, which is considered public. Improvements of OCELOT over RC4 are:

- Improved KSA to eliminate the problems associated with RC4;
- Extended internal state for improved security;
- Control on the PRNG's period based on the size of a counter;

2.1 Extended Internal State

The size of the PRNG's internal state is controlled by the Ocelot1 (Ocelot2) constructor's parameter $size4$ which determines the instance variables $\_size4$ (size in bytes) and $\_size$ (size in words). It consists in the instance arrays $\_data[64]$, $\_ss[256]$, the instance variable $\_val$, and additionally the Counter's state from instance array $\_data[64]$ in class Counter. It has an internal state of size $(2 \times \_size4 + 1) + 256$ bytes. $\_val$, $\_data[64]$ and the Counter form the fast varying part of the internal state, compensating for the slow variation of the dynamic permutation $\_ss[256]$. The Counter has also the purpose to control the period of the PRNG which we expect to be not less than Couter's period (i.e. $2^{8 \times \_size4}$) multiplied by $\_size4$ (the period's length is expressed in bytes).

2.2 Help Functions

Some help functions are used in the PRNG function and also in the initialization process:

```
UINT SS(UINT const& val);
void Swap(UINT const& val1, UINT const& val2);
UINT F1(UINT const& val);
UINT F2(UINT const& val);
UINT G1(UINT const& val);
UINT G2(UINT const& val);
```

The function $SS()$ is applying the $\_ss$ permutation to the bytes of the word argument;

The function $Swap()$ is swapping the positions of the $\_ss$ permutation based on the two word arguments;

$F1()$ and $F2()$ are two simple functions for flipping half of the bit positions in a word;

$G1()$ and $G2()$ are two simple shift with rotation functions. $G1()$ shifts a word left 5 positions with rotation. $G2()$ shifts a word right 5 positions with rotation;

There are also some other help methods in the OCELOT1 and OCELOT2 classes, like the methods for transforming bytes and words data in an endianess specific manner (littel endian or big endian). All of them can be easily identified in the source code.

2.3 Initialization

The initialization process, also known as the Key-Scheduling Algorithm (KSA), has the purpose to generate the initial internal state in a pseudo-random manner based on key data of any size (we are not concerned here about how the key data is obtained). The state array *_data*[64] in class Ocelot1 (Ocelot2) and the arrays *_init*[64], *_incr*[64] in class Counter are generated by using an expansion algorithm applied to the key data. The increment's bytes have values between 64 and 191 (see method *MakeGoodIncrement*() in class Counter). The initialization process in simplemented in method *Initialize*(), while the expansion algorithm is implemented in method *Expansion*(). The permutation array *_ss*[256] is starting with a predefined state and is initialized according to the known Durstenfeld shuffle algorithm, which ensure that the permutation array *_ss* is significantly changed by the initialization process. If the Durstenfeld algorithm is not completed during the expansion processes it is continued in method *Initialize*() until the data in all the 256 array positions is modified at least once.

2.4 PRNG Functions

There are two variants of the PRNG function implemented by method *GetNextWord*() in class *Ocelot*1 and respectively class *Ocelot*2, the differenece between the two being that the method in class *Ocelot*2 is somewhat more complex. After *_size* words of pseudo-random data are generated the Counter is incremented. For each Counter value the extended internal state is combined with the Couter's state in a different way (path) determined by the index *_ix*1, which is calculated pseudo-randomly from the current value of *_val* at Counter incrementation. The index *_ix*1 is forced an odd number which is prime with the size *_size* (usually a power of 2), in this way ensuring the full spanning property of index *_ix*1. As mentioned above, the dynamic permutation *_ss*[256] is isolating the other parts of the internal state from the generated keystream.

3. Security Considerations

Being inspired from RC4, I expect OCELOT to provide security characteristics for its generator similar to the ones of RC4. As it is stated on RSA's website (RSA Security Response to Weaknesses in Key-Scheduling Algorithm of RC4 - "http://www.rsa.com/rsalabs/node.asp?id=2009"), "all the recent attacks to RC4 relate only to the Key-Scheduling Algorithm, not to the generator. There are at present no known practical attacks against the RC4's generator when initialized with a randomly-chosen initial state." As possible solutions to the KSA problems the RSA site recommends: "The initial key scheduling component of RC4 should for now be routinely amended for new applications to include hashing and/or discarding the first 256 bytes of pseudo-random output. (This has in any case been RSA's routine recommendation.)". The OCELOT method can be considered as applying the hashing idea because the used expansion algorithm can be viewed as an ad-hoc inner hashing. Due to this inner hashing OCELOT allows the technique of concatenating the long-term key with a nonce without suffering from the problems plaguing RC4.

OCELOT being a new stream ciphering method, some further security research is needed.

4.0 Apendix

4.1 Test Samples for Ocelot1

1) statesize=16, key="XXXXXXX":
hexresult="85815117 AC8FDE76 B322240B 5FD687F5 9CE2702B ED5D1777 3257698A
832B4BE7 A0811520 2E0D7F29 A675853B 563655E6 5063716D 05FBA749
3A37BD59 CB6C3415 754C64FE 5C77352F B3DC188B B4172E84 46709D74
671A8C0A A4DDB042 FE797E42 095BCE1A 28616AFE 8AF1AAFF 904C0AEB

CC2CA0B9 6604DC8E 1696CFB2 4A0D84AC E3344F80 B6EACB1B 5821BDA0
73BF9945 6F2D8C12 73A5A787 AD205816 D0B7B83C 82D3684B 53973504
BD2469AF 6B7B53F3 4B5A8CAF 56AFB44A”’...

2) statesize=16, key=”YXXXXXX”:
hexresult=”BB53781D 00E53372 C30DA673 13033AB2 0234D468 9A010A9B B1C0A0BE
0B40BF0C BF6584CA 2F26079E 5EA11379 2B259F47 56AEFD90 B7243AEF
FBFC242A A3187B68 957062A2 0EFF2AC9 E23C2988 F95A7D9D 6E50DDBE
B3F5E4BD B5767CE3 C8467B96 A4891FCA ”...

3) statesize=16, key=”\0\0\0\0” :
hexresult=”93912BB7 3814C2D9 6B8D4CC4 CE160707 286C80FE AEA22044 021261BD
37A7310E 99CDD9E5 4780651C 5441AA6D DCCB8235 8B9B1184 7F489A0C
03AC5612 92849891 195078AE F0DBC3BD EE50179D 1804EC33 9D210FE0
17A81801 A0C8C1E7 ”...

4.2 C++ Reference Code for Ocelot1

```
// Ocelot1.h
#ifndef __OCELOT1_H__
#define __OCELOT1_H__

#include "Counter.h"

#include <cstring>

using namespace std;

#define BYTE unsigned char

class Ocelot1
{
//*****************************************************************************
//
// The OCELOT1 stream ciphering method method, Version 2.1.0 (09 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
// This is the C++ implementation of a new stream ciphering method called OCELOT1. It is acceptin
// be set to any practical state size.
//
// COPYRIGHT PROTECTION: The OCELOT1 stream ciphering method is still under development and testi
// freely distributed only for TESTING AND RESEARCH PURPOSES. The author is reserving for himself
// code, but any ideas about improving the code are welcomed and will be recognized if implemente
//
// If you are interested in testing the code, in research collaborations for possible security ho
// other information please contact the author at <george.anescu@sc-gen.com>.
//
// Updates for Version 2.0.0 (30 September 2010):
// - Adaptation from arrays of bytes to arays of words for increased performance
//
// Updates for Version 2.1.0 (09 December 2010):
```

```
// - Changes for efficiency and security improvements
//
// For C++ portability testing please use the Test Samples. The test code is given in readme.txt.
//
//******************************************************************************
public:
    //Sizes in bytes
    enum OCELOTSize
    {
        OCELOTSize16 = 16, OCELOTSize32 = 32, OCELOTSize64 = 64, OCELOTSize128 = 128,
        OCELOTSize256 = 256
    };

    //Constructors
    Ocelot1()
    {
        if (Ocelot1::IsBigEndian())
        {
            Swap = &Ocelot1::SwapBE;
            Bytes2Word = Ocelot1::Bytes2WordBE;
            Word2Bytes = Ocelot1::Word2BytesBE;
        }
        else
        {
            Swap = &Ocelot1::SwapLE;
            Bytes2Word = Ocelot1::Bytes2WordLE;
            Word2Bytes = Ocelot1::Word2BytesLE;
        }
    }

    Ocelot1(BYTE const* key, int keysize, OCELOTSize size4=OCELOTSize16)
    {
        if (Ocelot1::IsBigEndian())
        {
            Swap = &Ocelot1::SwapBE;
            Bytes2Word = Ocelot1::Bytes2WordBE;
            Word2Bytes = Ocelot1::Word2BytesBE;
        }
        else
        {
            Swap = &Ocelot1::SwapLE;
            Bytes2Word = Ocelot1::Bytes2WordLE;
            Word2Bytes = Ocelot1::Word2BytesLE;
        }
        Initialize(size4, key, keysize);
  }

    void Initialize(OCELOTSize size4, BYTE const* key, int keysize);

    void Initialize(OCELOTSize size4, UINT const* data, BYTE const* ss);

    void Reset()
```

```
    {
        _bcnt = 0;
        memcpy(_data, _data0, _size4);
        _val = _data0[_size];
        memcpy(_ss, _ss0, 256);
        _cnt.Reset();
        _ix = _ix0;
        _ix1 = _ix1_0;
        _incr = _incr0;
    }

    void GetNextWord(UINT& rnd);

    void GetNextByte(BYTE& rnd);

    void GetWords(UINT* words, int n)
    {
        for (register int i = 0; i < n; i++)
        {
            GetNextWord(words[i]);
        }
    }

    void GetBytes(BYTE* bytes, int n)
    {
        for (register int i = 0; i < n; i++)
        {
            GetNextByte(bytes[i]);
        }
    }

private:
    void Expansion(UINT const* data, int size, UINT* res, int dim, short iter, bool cpl);

    UINT SS(UINT const& val)
    {
        return _ss[(BYTE)val] | _ss[(BYTE)(val>>8)]<<8 | _ss[(BYTE)(val>>16)]<<16 |
               _ss[(BYTE)(val>>24)]<<24;
    }

    void SwapLE(UINT const& val1, UINT const& val2)
    {
        register BYTE *p1, *p2;
        register BYTE temp, v1, v2;
        p1 = (BYTE*)(&val1) + 3;
        p2 = (BYTE*)(&val2) + 3;
        //1
        v1 = *p1; v2 = *p2;
        temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
        //2
        v1 = *(--p1); v2 = *(--p2);
        temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
```

```
    //3
    v1 = *(--p1); v2 = *(--p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //4
    v1 = *(--p1); v2 = *(--p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
}

void SwapBE(UINT const& val1, UINT const& val2)
{
    register BYTE *p1, *p2;
    register BYTE temp, v1, v2;
    p1 = (BYTE*)&val1;
    p2 = (BYTE*)&val2;
    //1
    v1 = *p1; v2 = *p2;
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //2
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //3
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //4
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
}

static bool IsBigEndian()
{
    static UINT ui = 1;
    //Executed only at first call
    static bool result(reinterpret_cast<BYTE*>(&ui)[0] == 0);
    return result;
}

static UINT Bytes2WordLE(BYTE const* bytes)
{
    return (UINT)(*(bytes+3)) | (UINT)(*(bytes+2)<<8) |
           (UINT)(*(bytes+1)<<16) | (UINT)(*bytes<<24);
}

static UINT Bytes2WordBE(BYTE const* bytes)
{
    return *(UINT*)bytes;
}

static void Word2BytesLE(UINT word, BYTE* bytes)
{
    bytes += 3;
    *bytes = (BYTE)word;
    *(--bytes) = (BYTE)(word>>8);
```

```
    *(--bytes) = (BYTE)(word>>16);
    *(--bytes) = (BYTE)(word>>24);
}

static void Word2BytesBE(UINT word, BYTE* bytes)
{
    *bytes = (BYTE)word;
    *(++bytes) = (BYTE)(word>>8);
    *(++bytes) = (BYTE)(word>>16);
    *(++bytes) = (BYTE)(word>>24);
}

//F1 function
static UINT F1(UINT const& val)
{
    return (val & 0x55AA55AA) | (~val & 0xAA55AA55);
}

//F2 function
static UINT F2(UINT const& val)
{
    return (val & 0xAA55AA55) | (~val & 0x55AA55AA);
}

//G1 function - shift left rotating
static UINT G1(UINT const& val)
{
    //take last 5 bits, shift left 5 positions and make last 5 bits first
    return (val << 5) | ((val & 0xF8000000) >> 27);
}

//G2 function - shift right rotating
static UINT G2(UINT const& val)
{
    //take first 5 bits, shift right 5 positions and make first 5 bits last
    return (val >> 5) | ((val & 0x000001F) << 27);
}

void Bytes2Words(UINT* ar1, UINT* ar2, int const& len)
{
    for(register int i=0; i<len; i++)
    {
        ar2[i] = Bytes2Word((BYTE*)&ar1[i]);
    }
}

void Words2Bytes(UINT* ar1, BYTE* ar2, int const& len)
{
    BYTE* pbytes = ar2;
    for(register int i=0; i<len; i++,pbytes+=4)
    {
        Word2Bytes(ar1[i], pbytes);
```

```
        }
    }

    void (Ocelot1::*Swap)(UINT const& val1, UINT const& val2);
    UINT (*Bytes2Word)(BYTE const* bytes);
    void (*Word2Bytes)(UINT word, BYTE* bytes);

    const static BYTE _sss[256];
    Counter _cnt;
    UINT _data0[65];
    UINT _data[64];
    BYTE _ss0[256];
    BYTE _ss[256];
    UINT _val;
    UINT _bcnt;
    int _ssix;
    int _ix;
    int _ix1;
    int _incr;
    int _ix0;
    int _ix1_0;
    int _incr0;
    int _size4; //size in bytes
    int _size; //size in words
    int _size1;
};

#endif // __OCELOT1_H__

// Ocelot1.cpp
#include "Ocelot1.h"

//*****************************************************************************
//
// The OCELOT1 stream ciphering method, Version 2.1.0 (09 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
//*****************************************************************************

const BYTE Ocelot1::_sss[256] = {
    246, 79, 28, 40, 39, 27, 4, 148, 153, 149, 22, 75, 31, 38,
    222, 233, 110, 147, 102, 189, 144, 143, 11, 215, 249, 70,
    112, 207, 195, 192, 35, 124, 133, 66, 127, 188, 62, 104, 180,
    211, 19, 213, 68, 128, 82, 6, 203, 95, 156, 204, 119, 239,
    220, 43, 247, 221, 109, 238, 7, 118, 9, 15, 163, 101, 52, 94,
    64, 0, 197, 138, 85, 235, 176, 65, 25, 45, 24, 241, 21, 2,
    51, 255, 125, 140, 10, 13, 61, 228, 33, 14, 161, 115, 202,
    114, 191, 205, 83, 30, 67, 54, 186, 5, 169, 226, 165, 132, 69,
    23, 200, 20, 146, 183, 193, 48, 253, 56, 72, 126, 59, 209, 16,
    103, 81, 113, 60, 47, 73, 229, 208, 80, 106, 34, 50, 243,
    3, 178, 107, 108, 242, 100, 162, 217, 181, 129, 87, 250, 219,
```

```
    150, 167, 78, 29, 1, 168, 199, 12, 201, 155, 231, 91, 46,
    177, 53, 214, 92, 121, 136, 71, 17, 42, 36, 49, 158, 175, 254,
    137, 170, 173, 26, 252, 120, 88, 174, 139, 122, 58, 18,
    141, 184, 84, 37, 166, 230, 32, 160, 152, 117, 159, 240, 164,
    44, 245, 99, 232, 74, 135, 223, 55, 96, 63, 131, 134, 182,
    218, 130, 77, 142, 111, 244, 187, 248, 76, 57, 157, 97, 145,
    172, 171, 86, 41, 236, 151, 206, 198, 194, 227, 105, 8, 116,
    225, 210, 93, 212, 89, 154, 234, 237, 123, 196, 251, 224, 90,
    216, 190, 185, 98,179,
};

void Ocelot1::Initialize(OCELOTSize size4, BYTE const* key, int keysize)
{
    _bcnt = 0;
    _size4 = size4; //size in bytes
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    _ssix = 0;
    //Initialization, 256 cycles
    memcpy(_ss, _sss, 256);
    UINT kwords[64];
    int keysizew = keysize >> 2;
    if (keysize & 3)
    {
        kwords[keysizew] = 0;
        keysizew++;
    }
    memcpy(kwords, key, keysize);
    Bytes2Words(kwords, kwords, keysizew);
    Expansion(kwords, keysizew, _data0, _size+1, 3, false);
    UINT words[128];
    Expansion(kwords, keysizew, words, _size << 1, 3, true);
    _cnt.Initialize(words, _size);
    memcpy(_data, _data0, _size4);
    _val = _data0[_size];
    BYTE temp[4], by;
    UINT rnd, temp1;
    _incr = (_ss[_val & 0xFF] & _size1) | 1;
    _ix = -1;
    _ix1 = _incr;
    while (_ssix < 256)
    {
        _ix++;
        if (_ix == _size)
        {
            _ix = 0;
            _cnt.Increment();
            _ix1 += _incr;
            _ix1 &= _size1;
            _incr = (_ss[_val & 0xFF] & _size1) | 1;
        }
        _ix1 += _incr;
```

```
        _ix1 &= _size1;
        by = _data[_ix];
        _val ^= Ocelot1::F1(by);
        _val += _cnt[_ix];
        _data[_ix1] ^= SS(_val);
        rnd = SS((_cnt[_ix1] + by) ^ _val);
        temp[0] = (BYTE)_ssix++;
        temp[1] = (BYTE)_ssix++;
        temp[2] = (BYTE)_ssix++;
        temp[3] = (BYTE)_ssix++;
        temp1 = Bytes2Word(temp);
        (this->*Swap)(temp1, rnd);
    }
    //Create initial state
    memcpy(_ss0, _ss, 256);
    memcpy(_data0, _data, _size4);
    _data0[_size] = _val;
    _cnt.SaveState();
    _ix0 = _ix;
    _ix1_0 = _ix1;
    _incr0 = _incr;
}

//Initializaation from precalculated data (_size + 1) + 2*_size
void Ocelot1::Initialize(OCELOTSize size4, UINT const* data, BYTE const* ss)
{
    _bcnt = 0;
    _size4 = size4; //size in bytes
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    memcpy(_ss, ss, 256);
    memcpy(_data0, data, _size4+4);
    Bytes2Words(_data0, _data0, _size+1);
    memcpy(_data, _data0, _size4);
    _val = _data0[_size];
    UINT words[128];
    memcpy(words, data+_size+1, _size4 << 1);
    Bytes2Words(words, words, _size << 1);
    _cnt.Initialize(words, _size);
    _incr = (_ss[_val & 0xFF] & _size1) | 1;
    _ix = -1;
    _ix1 = _incr;
    //Create initial state
    memcpy(_ss0, _ss, 256);
    _cnt.SaveState();
    _ix0 = _ix;
    _ix1_0 = _ix1;
    _incr0 = _incr;
}

void Ocelot1::Expansion(UINT const* data, int size, UINT* res, int dim, short iter, bool cpl)
{
```

```
UINT words[64];
memcpy(words, data, size << 2);
int lend2 = size >> 1;
//Combining with dim
if (cpl)
{
    words[0] += SS(dim);
}
else
{
    words[0] ^= SS(dim);
}
//Propagate differences
BYTE temp[4];
temp[0] = _ss[0]; temp[1] = _ss[64];
temp[2] = _ss[128]; temp[3] = _ss[192];
register UINT val1 = Bytes2Word(temp);
temp[0] = _ss[32]; temp[1] = _ss[96];
temp[2] = _ss[160]; temp[3] = _ss[224];
register UINT val2 = Bytes2Word(temp);
UINT temp1, temp2;
register int k, i, ix = lend2;
for (k = 0; k < iter; k++)
{
    for (i = 0; i < size; i++, ix++)
    {
        if (ix == size) ix = 0;
        temp1 = words[i];
        temp2 = words[ix];
        if (cpl)
        {
            val1 += SS(temp1);
            val1 ^= temp2;
            val2 += Ocelot1::F1(val1);
            val2 ^= SS(temp2);
            val2 += temp1;
            words[i] = Ocelot1::G1(temp1) + SS(val1);
            words[ix] = Ocelot1::F2(temp2) ^ SS(val2);
        }
        else
        {
            val1 ^= SS(temp1);
            val1 += temp2;
            val2 ^= Ocelot1::F1(val1);
            val2 += SS(temp2);
            val2 ^= temp1;
            words[i] = Ocelot1::G1(temp1) ^ SS(val1);
            words[ix] = Ocelot1::F2(temp2) + SS(val2);
        }
        temp[0] = (BYTE)_ssix; temp[1] = (BYTE)(_ssix+1);
        temp[2] = (BYTE)(_ssix+2); temp[3] = (BYTE)(_ssix+3);
        temp1 = Bytes2Word(temp);
```

```
        (this->*Swap)(temp1, val2);
        _ssix+=4;
    }
}
//Expanding
i = 0;
ix = lend2;
int max = dim - lend2;
int j = 0;
for (k = 0; k < dim; k++, i++, ix++)
{
    if (i == size) i = 0;
    if (ix == size) ix = 0;
    temp1 = words[i];
    temp2 = words[ix];
    if (cpl)
    {
        val1 ^= SS(temp1);
        val1 += temp2;
        val2 ^= Ocelot1::G1(val1);
        res[k] = Ocelot1::F2(val2) + SS(data[j]);
        val2 += SS(temp2);
        val2 ^= temp1;
        if (k < max)
        {
            words[i] = Ocelot1::G2(temp1) ^ SS(val1);
            words[ix] = Ocelot1::F1(temp2) + SS(val2);
        }
    }
    else
    {
        val1 += SS(temp1);
        val1 ^= temp2;
        val2 += Ocelot1::G1(val1);
        res[k] = Ocelot1::F2(val2) ^ SS(data[j]);
        val2 ^= SS(temp2);
        val2 += temp1;
        if (k < max)
        {
            words[i] = Ocelot1::G2(temp1) + SS(val1);
            words[ix] = Ocelot1::F1(temp2) ^ SS(val2);
        }
    }
    temp[0] = (BYTE)_ssix; temp[1] = (BYTE)(_ssix+1);
    temp[2] = (BYTE)(_ssix+2); temp[3] = (BYTE)(_ssix+3);
    temp1 = Bytes2Word(temp);
    (this->*Swap)(temp1, val2);
    _ssix+=4;
    j++;
    if (j >= size)
    {
        j = 0;
```

```
        }
    }
}

void Ocelot1::GetNextWord(UINT& rnd)
{
    static UINT temp;
    _ix++;
    if (_ix == _size)
    {
        _ix = 0;
        _cnt.Increment();
        _ix1 += _incr;
        _ix1 &= _size1;
        _incr = (_ss[_val & 0xFF] & _size1) | 1;
    }
    _ix1 += _incr;
    _ix1 &= _size1;
    temp = _data[_ix];
    _val ^= Ocelot1::F1(temp);
    _val += _cnt[_ix];
    _data[_ix1] ^= SS(_val);
    rnd = SS((_cnt[_ix1] + temp) ^ _val);
    (this->*Swap)(_val, temp);
}

void Ocelot1::GetNextByte(BYTE& rnd)
{
    static UINT word = 0;
    switch (_bcnt)
    {
        case 0:
            GetNextWord(word);
            rnd = (BYTE)word;
            break;

        case 1:
            rnd = (BYTE)(word >> 8);
            break;

        case 2:
            rnd = (BYTE)(word >> 16);
            break;

        case 3:
            rnd = (BYTE)(word >> 24);
            break;
    }
    _bcnt = (_bcnt + 1) & 3;
}
```

4.3 Test Samples for Ocelot2

1) statesize=16, key=”XXXXXXX”:
hexresult=”0B75CA30 81F74147 D0A063FD C267AC5E 5F55E524 B1776B36 1B4A95FC AD5BBD67 3D311A37 F52B4FFA 0DF369EE 606BF010 02CD6F5E EC2EAEF3 C1755375 CD6ABE42 F5E479E5 9A0DC29D F4157CB0 2BFC2B05 45722A93 588D2967 3C2F49C5 CF060F7B DA6EAA68 64318269 2DE23173 7A6EFFF9 E30D8C0E F9318214 63D2AF90 3646C1CA AF1B793D 50BA7B45 3BB0E500 62AB5AC2 D9BC91B7 69B11BF6 F8E9C0AC CC5BCCC7 EB4F7C4E 0429EC98 F53DD3EB 981745B2 F324B17F 13C36ADC ”...

2) statesize=16, key=”YXXXXXX”:
hexresult=”CCF0DCEC 508D2868 124AC94E 7980FB6F 04693241 82F690DF 8592046D 1DFCF3FC 6028E415 2E130B06 9B22DA6D 6D002D96 441739D8 E8F7970E 111D3D83 AB306AF8 893CACBE 7C54D9E0 4CBDBC80 2B0B2B4D F1FC1C81 295ACFF6 8677A96E 860A0C90 AFA6627C ”...

3) statesize=16, key=”\0\0\0\0” :
hexresult=”65E58C8D 4C08FADB A5E313B1 183A2393 BB192EAF DDE83455 0964B3F1 EBE6DFA9 5C5EBF7F B35732F5 8005FB10 4A11FEB2 4530AFB5 A77BF14E 9B8E3767 8F2FC326 3DFC0D0B C2A37726 55ACE2D9 24686C66 9C650713 E94E1D68 E41102AD ”...

4.4 C++ Reference Code for Ocelot2

```
// Ocelot2.h
#ifndef __OCELOT2_H__
#define __OCELOT2_H__

#include "Counter.h"

#include <cstring>

using namespace std;

#define BYTE unsigned char

class Ocelot2
{
//*****************************************************************************
//
// The OCELOT2 stream ciphering method, Version 2.1.0 (09 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.scgen.com
// All right reserved.
//
// This is the C++ implementation of a new stream ciphering method called OCELOT2. It is accepting
// be set to any practical state size.
//
// COPYRIGHT PROTECTION: The OCELOT2 stream ciphering method is still under development and testin
// freely distributed only for TESTING AND RESEARCH PURPOSES. The author is reserving for himself
// code, but any ideas about improving the code are welcomed and will be recognized if implemente
//
// If you are interested in testing the code, in research collaborations for possible security ho
```

```
// other information please contact the author at <george.anescu@scgen.com>.
//
// Updates for Version 2.0.0 (31 July 2010):
// - Adaptation from arrays of bytes to arays of words for increased performance
//
// Updates for Version 2.1.0 (09 December 2010):
// - Changes for efficiency and security improvements
//
// For C++ portability testing please use the Test Samples. The test code is given in readme.txt.
//
//*****************************************************************************
public:
    //Sizes
    enum OCELOTSize
    {
        OCELOTSize16 = 16, OCELOTSize32 = 32, OCELOTSize64 = 64, OCELOTSize128 = 128,
        OCELOTSize256 = 256
    };

    //Constructors
    Ocelot2()
    {
        if (Ocelot2::IsBigEndian())
        {
            Swap = &Ocelot2::SwapBE;
            Bytes2Word = Ocelot2::Bytes2WordBE;
            Word2Bytes = Ocelot2::Word2BytesBE;
        }
        else
        {
            Swap = &Ocelot2::SwapLE;
            Bytes2Word = Ocelot2::Bytes2WordLE;
            Word2Bytes = Ocelot2::Word2BytesLE;
        }
    }

    Ocelot2(BYTE const* key, int keysize, OCELOTSize size4=OCELOTSize16)
    {
        if (Ocelot2::IsBigEndian())
        {
            Swap = &Ocelot2::SwapBE;
            Bytes2Word = Ocelot2::Bytes2WordBE;
            Word2Bytes = Ocelot2::Word2BytesBE;
        }
        else
        {
            Swap = &Ocelot2::SwapLE;
            Bytes2Word = Ocelot2::Bytes2WordLE;
            Word2Bytes = Ocelot2::Word2BytesLE;
        }
        Initialize(size4, key, keysize);
    }
```

```
    void Initialize(OCELOTSize size4, BYTE const* key, int keysize);

    void Initialize(OCELOTSize size4, UINT const* data, BYTE const* ss);

    void Reset()
    {
        _bcnt = 0;
        memcpy(_data, _data0, _size4);
        _val = _data0[_size];
        memcpy(_ss, _ss0, 256);
        _cnt.Reset();
        _ix = _ix0;
        _ix1 = _ix1_0;
        _incr = _incr0;
    }

    void GetNextWord(UINT& rnd);

    void GetNextByte(BYTE& rnd);

    void GetWords(UINT* words, int n)
    {
        for (register int i = 0; i < n; i++)
        {
            GetNextWord(words[i]);
        }
    }

    void GetBytes(BYTE* bytes, int n)
    {
        for (register int i = 0; i < n; i++)
        {
            GetNextByte(bytes[i]);
        }
    }

private:
    void Expansion(UINT const* data, int size, UINT* res, int dim, short iter, bool cpl);

    UINT SS(UINT const& val)
    {
        return _ss[(BYTE)val] | _ss[(BYTE)(val>>8)]<<8 | _ss[(BYTE)(val>>16)]<<16 |
               _ss[(BYTE)(val>>24)]<<24;
    }

    void SwapLE(UINT const& val1, UINT const& val2)
    {
        register BYTE *p1, *p2;
        register BYTE temp, v1, v2;
        p1 = (BYTE*)(&val1) + 3;
        p2 = (BYTE*)(&val2) + 3;
```

```
    //1
    v1 = *p1; v2 = *p2;
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //2
    v1 = *(--p1); v2 = *(--p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //3
    v1 = *(--p1); v2 = *(--p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //4
    v1 = *(--p1); v2 = *(--p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
}

void SwapBE(UINT const& val1, UINT const& val2)
{
    register BYTE *p1, *p2;
    register BYTE temp, v1, v2;
    p1 = (BYTE*)&val1;
    p2 = (BYTE*)&val2;
    //1
    v1 = *p1; v2 = *p2;
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //2
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //3
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
    //4
    v1 = *(++p1); v2 = *(++p2);
    temp = _ss[v1]; _ss[v1] = _ss[v2]; _ss[v2] = temp;
}

static bool IsBigEndian()
{
    static UINT ui = 1;
    //Executed only at first call
    static bool result(reinterpret_cast<BYTE*>(&ui)[0] == 0);
    return result;
}

static UINT Bytes2WordLE(BYTE const* bytes)
{
    return (UINT)(*(bytes+3)) | (UINT)(*(bytes+2)<<8) |
           (UINT)(*(bytes+1)<<16) | (UINT)(*bytes<<24);
}

static UINT Bytes2WordBE(BYTE const* bytes)
{
    return *(UINT*)bytes;
}
```

```
static void Word2BytesLE(UINT word, BYTE* bytes)
{
    bytes += 3;
    *bytes = (BYTE)word;
    *(--bytes) = (BYTE)(word>>8);
    *(--bytes) = (BYTE)(word>>16);
    *(--bytes) = (BYTE)(word>>24);
}

static void Word2BytesBE(UINT word, BYTE* bytes)
{
    *bytes = (BYTE)word;
    *(++bytes) = (BYTE)(word>>8);
    *(++bytes) = (BYTE)(word>>16);
    *(++bytes) = (BYTE)(word>>24);
}

//F1 function
static UINT F1(UINT const& val)
{
    return (val & 0x55AA55AA) | (~val & 0xAA55AA55);
}

//F2 function
static UINT F2(UINT const& val)
{
    return (val & 0xAA55AA55) | (~val & 0x55AA55AA);
}

//G1 function - shift left rotating
static UINT G1(UINT const& val)
{
    //take last 5 bits, shift left 5 positions and make last 5 bits first
    return (val << 5) | ((val & 0xF8000000) >> 27);
}

//G2 function - shift right rotating
static UINT G2(UINT const& val)
{
    //take first 5 bits, shift right 5 positions and make first 5 bits last
    return (val >> 5) | ((val & 0x000001F) << 27);
}

void Bytes2Words(UINT* ar1, UINT* ar2, int const& len)
{
    for(register int i=0; i<len; i++)
    {
        ar2[i] = Bytes2Word((BYTE*)&ar1[i]);
    }
}
```

```
    void Words2Bytes(UINT* ar1, BYTE* ar2, int const& len)
    {
        BYTE* pbytes = ar2;
        for(register int i=0; i<len; i++,pbytes+=4)
        {
            Word2Bytes(ar1[i], pbytes);
        }
    }

    void (Ocelot2::*Swap)(UINT const& val1, UINT const& val2);
    UINT (*Bytes2Word)(BYTE const* bytes);
    void (*Word2Bytes)(UINT word, BYTE* bytes);

    const static BYTE _sss[256];
    Counter _cnt;
    UINT _data0[65];
    UINT _data[64];
    BYTE _ss0[256];
    BYTE _ss[256];
    UINT _val;
    UINT _bcnt;
    int _ssix;
    int _ix;
    int _ix1;
    int _incr;
    int _ix0;
    int _ix1_0;
    int _incr0;
    int _size4; //size in bytes
    int _size; //size in words
    int _size1;
};

#endif // __OCELOT2_H__

// Ocelot2.cpp
#include "Ocelot2.h"

//*****************************************************************************
//
// The OCELOT2 stream ciphering method, Version 2.1.0 (09 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.scgen.com
// All right reserved.
//
//*****************************************************************************

const BYTE Ocelot2::_sss[256] = {
    246, 79, 28, 40, 39, 27, 4, 148, 153, 149, 22, 75, 31, 38,
  a  222, 233, 110, 147, 102, 189, 144, 143, 11, 215, 249, 70,
    112, 207, 195, 192, 35, 124, 133, 66, 127, 188, 62, 104,
    180, 211, 19, 213, 68, 128, 82, 6, 203, 95, 156, 204, 119,
    239, 220, 43, 247, 221, 109, 238, 7, 118, 9, 15, 163, 101,
```

```
    52, 94, 64, 0, 197, 138, 85, 235, 176, 65, 25, 45, 24, 241,
    21, 2, 51, 255, 125, 140, 10, 13, 61, 228, 33, 14, 161, 115,
    202, 114, 191, 205, 83, 30, 67, 54, 186, 5, 169, 226, 165,
    132, 69, 23, 200, 20, 146, 183, 193, 48, 253, 56, 72, 126,
    59, 209, 16, 103, 81, 113, 60, 47, 73, 229, 208, 80, 106,
    34, 50, 243, 3, 178, 107, 108, 242, 100, 162, 217, 181, 129,
    87, 250, 219, 150, 167, 78, 29, 1, 168, 199, 12, 201, 155,
    231, 91, 46, 177, 53, 214, 92, 121, 136, 71, 17, 42, 36, 49,
    158, 175, 254, 137, 170, 173, 26, 252, 120, 88, 174, 139, 122,
    58, 18, 141, 184, 84, 37, 166, 230, 32, 160, 152, 117, 159,
    240, 164, 44, 245, 99, 232, 74, 135, 223, 55, 96, 63, 131,
    134, 182, 218, 130, 77, 142, 111, 244, 187, 248, 76, 57, 157,
    97, 145, 172, 171, 86, 41, 236, 151, 206, 198, 194, 227, 105,
    8, 116, 225, 210, 93, 212, 89, 154, 234, 237, 123, 196, 251,
    224, 90, 216, 190, 185, 98,179,
};

void Ocelot2::Initialize(OCELOTSize size4, BYTE const* key, int keysize)
{
    _bcnt = 0;
    _size4 = size4; //size in bytes
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    _ssix = 0;
    //Initialization, 64 cycles
    memcpy(_ss, _sss, 256);
    UINT kwords[64];
    int keysizew = keysize >> 2;
    if (keysize & 3)
    {
        kwords[keysizew] = 0;
        keysizew++;
    }
    memcpy(kwords, key, keysize);
    Bytes2Words(kwords, kwords, keysizew);
    Expansion(kwords, keysizew, _data0, _size+1, 3, false);
    UINT words[128];
    Expansion(kwords, keysizew, words, _size<<1, 3, true);
    _cnt.Initialize(words, _size);
    memcpy(_data, _data0, _size4);
    _val = _data0[_size];
    BYTE temp[4], by;
    UINT rnd, temp1;
    _incr = (_ss[_val & 0xFF] & _size1) | 1;
    _ix = -1;
    _ix1 = _incr;
    while (_ssix < 256)
    {
        _ix++;
        if (_ix == _size)
        {
            _ix = 0;
```

```
            _cnt.Increment();
            _ix1 += _incr;
            _ix1 &= _size1;
            _incr = (_ss[_val & 0xFF] & _size1) | 1;
        }
        _ix1 += _incr;
        _ix1 &= _size1;
        _val ^= Ocelot2::F1(_data[_ix]);
        _val += _cnt[_ix];
        by = _val;
        _val ^= _data[_ix1];
        _data[_ix1] ^= SS(_val);
        if (_ix != _ix1)
        {
            _data[_ix] += by;
        }
        rnd = SS((_cnt[_ix1] + by) ^ _val);
        temp[0] = (BYTE)_ssix++;
        temp[1] = (BYTE)_ssix++;
        temp[2] = (BYTE)_ssix++;
        temp[3] = (BYTE)_ssix++;
        temp1 = Bytes2Word(temp);
        (this->*Swap)(temp1, rnd);
    }
    //Create initial state
    memcpy(_ss0, _ss, 256);
    memcpy(_data0, _data, size4);
    _data0[_size] = _val;
    _cnt.SaveState();
    _ix0 = _ix;
    _ix1_0 = _ix1;
    _incr0 = _incr;
}

//Initializaation from precalculated data (_size + 1) + 2*_size
void Ocelot2::Initialize(OCELOTSize size4, UINT const* data, BYTE const* ss)
{
    _bcnt = 0;
    _size4 = size4; //size in bytes
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    memcpy(_ss, ss, 256);
    memcpy(_data0, data, _size4+4);
    Bytes2Words(_data0, _data0, _size+1);
    memcpy(_data, _data0, _size4);
    _val = _data0[_size];
    UINT words[128];
    memcpy(words, data+_size+1, _size4<<1);
    Bytes2Words(words, words, _size<<1);
    _cnt.Initialize(words, _size);
    _incr = (_ss[_val & 0xFF] & _size1) | 1;
    _ix = -1;
```

```
    _ix1 = _incr;
    //Create initial state
    memcpy(_ss0, _ss, 256);
    _cnt.SaveState();
    _ix0 = _ix;
    _ix1_0 = _ix1;
    _incr0 = _incr;
}

void Ocelot2::Expansion(UINT const* data, int size, UINT* res, int dim, short iter, bool cpl)
{
    UINT words[64];
    memcpy(words, data, size << 2);
    int lend2 = size >> 1;
    //Combining with dim
    if (cpl)
    {
        words[0] += SS(dim);
    }
    else
    {
        words[0] ^= SS(dim);
    }
    //Propagate differences
    BYTE temp[4];
    temp[0] = _ss[0]; temp[1] = _ss[64];
    temp[2] = _ss[128]; temp[3] = _ss[192];
    register UINT val1 = Bytes2Word(temp);
    temp[0] = _ss[32]; temp[1] = _ss[96];
    temp[2] = _ss[160]; temp[3] = _ss[224];
    register UINT val2 = Bytes2Word(temp);
    UINT temp1, temp2;
    register int k, i, ix = lend2;
    for (k = 0; k < iter; k++)
    {
        for (i = 0; i < size; i++, ix++)
        {
            if (ix == size) ix = 0;
            temp1 = words[i];
            temp2 = words[ix];
            if (cpl)
            {
                val1 += SS(temp1);
                val1 ^= temp2;
                val2 += Ocelot2::F1(val1);
                val2 ^= SS(temp2);
                val2 += temp1;
                words[i] = Ocelot2::G1(temp1) + SS(val1);
                words[ix] = Ocelot2::F2(temp2) ^ SS(val2);
            }
            else
            {
```

```
                val1 ^= SS(temp1);
                val1 += temp2;
                val2 ^= Ocelot2::F1(val1);
                val2 += SS(temp2);
                val2 ^= temp1;
                words[i] = Ocelot2::G1(temp1) ^ SS(val1);
                words[ix] = Ocelot2::F2(temp2) + SS(val2);
            }
            temp[0] = (BYTE)_ssix; temp[1] = (BYTE)(_ssix+1);
            temp[2] = (BYTE)(_ssix+2); temp[3] = (BYTE)(_ssix+3);
            temp1 = Bytes2Word(temp);
            (this->*Swap)(temp1, val2);
            _ssix+=4;
        }
    }
    //Expanding
    i = 0;
    ix = lend2;
    int max = dim - lend2;
    int j = 0;
    for (k = 0; k < dim; k++, i++, ix++)
    {
        if (i == size) i = 0;
        if (ix == size) ix = 0;
        temp1 = words[i];
        temp2 = words[ix];
        if (cpl)
        {
            val1 ^= SS(temp1);
            val1 += temp2;
            val2 ^= Ocelot2::G1(val1);
            res[k] = Ocelot2::F2(val2) + SS(data[j]);
            val2 += SS(temp2);
            val2 ^= temp1;
            if (k < max)
            {
                words[i] = Ocelot2::G2(temp1) ^ SS(val1);
                words[ix] = Ocelot2::F1(temp2) + SS(val2);
            }
        }
        else
        {
            val1 += SS(temp1);
            val1 ^= temp2;
            val2 += Ocelot2::G1(val1);
            res[k] = Ocelot2::F2(val2) ^ SS(data[j]);
            val2 ^= SS(temp2);
            val2 += temp1;
            if (k < max)
            {
                words[i] = Ocelot2::G2(temp1) + SS(val1);
                words[ix] = Ocelot2::F1(temp2) ^ SS(val2);
```

```
            }
        }
        temp[0] = (BYTE)_ssix; temp[1] = (BYTE)(_ssix+1);
        temp[2] = (BYTE)(_ssix+2); temp[3] = (BYTE)(_ssix+3);
        temp1 = Bytes2Word(temp);
        (this->*Swap)(temp1, val2);
        _ssix+=4;
        j++;
        if (j >= size)
        {
            j = 0;
        }
    }
}

void Ocelot2::GetNextWord(UINT& rnd)
{
    static UINT temp;
    _ix++;
    if (_ix == _size)
    {
        _ix = 0;
        _cnt.Increment();
        _ix1 += _incr;
        _ix1 &= _size1;
        _incr = (_ss[_val & 0xFF] & _size1) | 1;
    }
    _ix1 += _incr;
    _ix1 &= _size1;
    _val ^= Ocelot2::F1(_data[_ix]);
    _val += _cnt[_ix];
    temp = _val;
    _val ^= _data[_ix1];
    _data[_ix1] ^= SS(_val);
    if (_ix != _ix1)
    {
        _data[_ix] += temp;
    }
    rnd = SS((_cnt[_ix1] + temp) ^ _val);
    (this->*Swap)(_val, temp);
}

void Ocelot2::GetNextByte(BYTE& rnd)
{
    static UINT word = 0;
    switch (_bcnt)
    {
        case 0:
            GetNextWord(word);
            rnd = (BYTE)word;
            break;
```

```
        case 1:
            rnd = (BYTE)(word >> 8);
            break;

        case 2:
            rnd = (BYTE)(word >> 16);
            break;

        case 3:
            rnd = (BYTE)(word >> 24);
            break;
    }
    _bcnt = (_bcnt + 1) & 3;
}
```

4.5 C++ Reference Code for Counter

```
// Counter.h
#ifndef __COUNTER_H__
#define __COUNTER_H__

#include <cstring>

using namespace std;

#define BYTE unsigned char
#define UINT unsigned int

class Counter
{
public:
    //Constructors
    Counter() {}

    Counter(UINT const* init, int size)
    {
        Initialize(init, size);
    }

    void Initialize(UINT const* init, int size)
    {
        _size = size;
        _size4 = size << 2;
        memcpy(_init, init, _size4);
        memcpy(_data, init, _size4);
        memcpy(_incr, init+_size, _size4);
        MakeGoodIncrement();
    }

    UINT& operator[](int i) { return _data[i]; }

    void Reset()
```

```
    {
        memcpy(_data, _init, _size4);
    }

    void SaveState()
    {
        memcpy(_init, _data, _size4);
    }

    //Increment _data with _incr
    void Increment();

private:
    void MakeGoodIncrement();

    int _size;
    int _size4;
    UINT _init[64];
    UINT _data[64];
    UINT _incr[64];
};

#endif // __COUNTER_H__

// Counter.cpp
#include "Counter.h"

//Increment _data with _incr
void Counter::Increment()
{
    static UINT carry = 0;
    static long long res;
    carry = 0;
    for (register int i = 0; i < _size; i++)
    {
        res = _data[i] + _incr[i] + carry;
        _data[i] = (UINT)res;
        carry = (UINT)(res >> 32);
    }
}

void Counter::MakeGoodIncrement()
{
    //Set first bit to ensure that the increment is an odd number
    //(guarantees maximal period of the counter, the same as for incrementing with 1)
    _incr[0] |= 0x01;
    //A good increment should provide fast variation.
    //Ensure that each byte is between 64 and 191.
    BYTE* pbytes = (BYTE*)&_incr[0];
    for (register int i = 0; i < _size4; i++,pbytes++)
    {
        if ((*pbytes & 0x80) != 0)
```

```
        {
            //reset bit 64
            *pbytes &= 0xBF;
        }
        else
        {
            //set bit 64
            *pbytes |= 0x40;
        }
    }
}
```