CompactECC Plus – Elliptic Curve Cryptography

C++ class template library suitable for embedded systems

Optimized implementation for signature creation on the secp192r1 curve, including optimized assembler implementations for the ARM7TDMI and ARM Cortex-M3 cores.

Reference Manual

Revision 1.2

18th July 2010

ubisys®

1 Overview

The stream-line CompactECC library is based on a flexible, variable-sized big integer representation, the CBigUnsigned class. This provides for the potential of further speed optimizations at the expense of reduced flexibility.

Some functions were implemented in optimized assembler for the 32-bit ARM platform, namely for the AT91SAM7S/X and ATSAM3S microcontroller series, while focussing on the ECDSA signature generation using the secp192r1 curve.

Support for the 8-bit AVR architecture is also included. Currently, no assembler optimized functions are available for AVR, though.

This document does only describe the relevant differences to the generic CompactECC implementation. For a full documentation, refer to the CompactECC Referenece Manual [1].

2 Architecture-specific issues

2.1 ARM

The ARM architecture is a 32-bit architecture and has a flat memory model, covering the full address range of 4 GB. The memory type to be accesses will be determined by decoding the addresses, i.e. flash memory and data memory have dedicated memory areas. This type of memory model is best suited for C/C++ compilers and does not require any special treatment.

2.2 AVR

The AVR 8-bit architecture is a modified Harvard architecture and uses separate memories for non-volatile program and volatile data storage. Access to the program storage is possible to allow loading constants from non-volatile memory. Separate machine opcodes exist to access data and program memories.

In C/C++, the compiler needs to know which memory a pointer (or reference in C++) refers to, to generate the correct opcodes. C/C++ compilers for the AVR architecture handle this situation by non-standard language extensions.

The IAR C/C++ compiler for AVR introduces extended keywords to specify the memory type as well as the pointer size (tiny, small, far, huge pointers) and placement of objects into different memories.

Keyword	Memory

`__tiny`
`__near`
`__far`
`__huge`
	data memory space (RAM)

`__tinyflash`
`__flash`
`__farflash`
`__hugeflash`
	code memory space (Flash)

Further keywords exist for EEPROM and IO space. If no keyword is specified, the object will be located in data memory space or the pointer will point to data memory space. The pointer size is determined by the selected memory model (i.e. `__near` for a small memory model).

Another modifier is available to support pointers to any type of object: `__generic`. The drawback of this approach is that the type of object is encoded into the pointer. The compiler will generate code which will be evaluated at runtime on every access, resulting in a speed penalty.

3 Architecture-specific implementation

It was attempted to unify the implementation for the ARM and AVR architecture and separate architecture-dependent issues from the main code base. Architecture-dependent definitions are contained in the header `CompactECCFArch.h`.

Base types for the digit storage as well as for the multiplication result and for generic integers are customizable per architecture.

CompactECC is a C++ class-based implementation and thus uses the implicit '`this`' pointer, which faces the issues described in section 2.2. A class can be defined to reside in near memory, and thus the '`this`' pointer will be automatically a pointer with the attribute `__near`. An instance of this class can only reside in near memory. CompactECC makes use of predefined instance data, which is best stored in flash memory. In this case, the class must be defined with the `__flash` attribute. To avoid declaring each class more than once, e.g. for flash and near memory, a solution was implemented which allows specifying the type of the 'this' pointer via a template parameter. The solution is detailed in section 0.

3.1 Architecture-specific data types

Architecture-dependent type definitions are defined in a per-architecture type class. ArchARM is defined for the ARM architecture and ArchAVR8 is defined for the AVR 8-bit architecture. The typedef `T` defines the base type for the digit storage. This should be the natural data width of the processor. The typedef `T2` defines the result of a multiplication of two variables of type `T`, i.e. has twice the bit width as `T`. The `uint` and `sint` typedefs define the type to be used for general-purpose integers inside the code. These types are mainly used for loop counters and array indices. The enum `digitBits` defines the bit width of T.

The typedef defaultMem defines the default memory type to use if not explicitly specified (c.f. the following section 0).

- for ARM:

```
class ArchARM
{
    // Typedefs
    public:
        // the digit type (T) and multiplication result type (T2)
        typedef unsigned int T;
        typedef unsigned long long T2;

        // Define the type of general-purpose integers,
        // mainly loop counters
        // preferably 32 bits on ARM
        typedef unsigned int uint;
        typedef int sint;

        // the default memory/pointer type
        typedef memGen defaultMem;

    // Enums
    public:
        enum {
            // bits in T
            digitBits = 32
        };
};
```

- for AVR:

```
class ArchAVR8
{
    public:
        // the digit type (T) and multiplication result type (T2)
        typedef unsigned char T;
        typedef unsigned short T2;

        // Define the type of general-purpose integers
        // mainly loop counters
        typedef unsigned char uint;
        typedef signed char sint;

        // the default memory/pointer type:
        // 'memGen' without any specifiers
        // (e.g. near pointers when using the small memory model)
        typedef memGen defaultMem;

    // Enums
    public:
        enum {
            // bits in T
            digitBits = 8
        };
};
```

3.2 Architecture-specific memory type attributes

Support for architecture-specific memory types was included for the AVR architecture, but is adaptable to other architectures with a non-flat memory model, under the following conditions:

- the compiler allows to specify the memory attributes by using a typedef
- the compiler inherits memory attributes for the '`this`' pointer from a base class

For ARM, only the generic memory type is defined by the class `memGen`.

For AVR, the classes `memNear` and `memFlash` are defined, explicitly allowing to specify the `__near` and `__flash` attributes.

A mapping from these classes to attributed memory types is achieved by using a specialized class template `MemAttrMapping`:

```
template<typename Type, typename Memory> class MemAttrMapping;
```

Full template specialization allows mapping the memory classes memGen, memNear and memFlash to attributed types:

```
template<typename Type> class MemAttrMapping<Type, memGen>
{
    public:
        typedef Type T;
};

// For AVR only:

template<typename Type> class MemAttrMapping<Type, memNear>
{
    public:
        typedef __near Type T;
};

template<typename Type> class MemAttrMapping<Type, memFlash>
{
    public:
        typedef __flash Type T;
};
```

A pointer to an unsigned int residing in flash memory can be defined in the following way:

```
MemAttrMapping<memFlash, unsigned int>::T *pPointer;
```

The full flexibility of this approach is achieved by replacing memFlash with a template parameter to a class or function template, allowing the user of the class or function to specify which memory type to use, without defining the function or class more than once.

3.2.1 this-Pointer attributes

The IAR C/C++ Compiler for the AVR microcontroller allows specifying the memory for the class instance storage and the this-pointer by using the extended keywords:

```
class __ flash ClassFlash  // class instance stored in flash (read-only!)
{
};

class __near ClassNear  // class instance stored in near memory
{
};
```

When a class inherits from another class, the memory type is automatically inherited and needs to be specified only for the base class.

The following base classes are defined:

```
class memGenBaseClass {};

// For AVR only:
class __near memNearBaseClass {};
class __flash memFlashBaseClass {};
```

Besides being a key to be used with `MemAttrMapping`, the classes `memGen`, `memNear` and `memFlash` define the corresponding base class by using a typedef:

```
class memGen
{
    public:
          typedef memGenBaseClass baseClass;
};

class memFlash
{
    public:
          typedef memFlashBaseClass baseClass;

};

class memNear
{
    public:
          typedef memNearBaseClass baseClass;
};
```

This allows defining a generic class template, by specifying the memory class as a template parameter and privately inheriting from it in order to inherit the memory type:

```
template<typename MEM>
class GenericClass : private MEM::baseClass
{
    // ...
};
```

`GenericClass<memFlash>` would generate a class instance residing in Flash, whereas `GenericClass<memNear>` would generate a class instance residing in near memory. `GenericClass<memGen>` would result in the default memory type, e.g. near as well for the small memory model.

4 Optimizations

The `CBigUnsigned` class responsible for handling big unsigned integer numbers was simplified. The flexibility of handling different-sized integer numbers was removed in favour of a fixed-sized implementation. The fixed-size implementation has a fixed size at runtime, allowing for certain compiler optimizations, but its size can be configured through a numeric template parameter at compile time. Through this step, the in-memory layout of the simplified integer class is just an array of th e underlying base type[1]. The simple layout allows performing calculations with hand-coded assembler functions for maximum performance.

The “modulo-p” operation was optimized for the secp192r1 curve by using a fast reduction algorithm available for so-called pseudo Mersenne numbers, which avoids division and multiplication.

4.1 ARM-specific optimizations

Several mathematical functions were implemented in assembler for ARM architecture to obtain a further speed gain. The most time-consuming operations were identified with a profiler. The multiplication of 192 x 192 bits to a result of 384 bits was identified to be the most time-consuming operation and was thus implemented in assembler. The ARM instruction set was chosen, because the multiplication 32 x 32 bits to 64 bits is not available in the Thumb instruction set and the multiplication requires many registers for adequate performance, which are not available in the Thumb instruction set either.

The squaring operation of 192 bits to a result of 384 bits was also identified as time-consuming. For simplicity, it was implemented as a call to the optimized multiplication function.

The addition operation of 192 + 192 bits to a result of 192 bits was also identified as consuming a certain amount of time and was implemented in assembler as well. The addition uses only few registers and thus it is implemented in the Thumb instruction set.

All other mathematical operations are using the generic C++ implementation, adapted to the simplified `CBigUnsigned` class.

The AT91SAM7S/X microcontroller series suffer from a bottle-neck when executing ARM instructions from the internal flash memory at higher clock frequencies. The internal flash can operate at single-cycle access only up to 30 MHz and is connected to the processor core by using a 32 bit bus. A simple cache register allows executing consecutive 16 bit Thumb instructions at the maximum clock frequency of 55 MHz, but requires wait-states when executing 32 bit ARM instructions. Thus, to achieve optimum performance, the mentioned multiplication function, implemented in the ARM instruction set, is to be executed from the internal SRAM, which allows single-cycle access up to the maximum clock frequency, avoiding the wait-states.

[1] The type is defined by the architecture definition, c.f. section 3.1.

ARM Cortex-M3 optimizations use the same 32-bit ARM instructions as the ARM7TDMI implementation, making full use of the Thumb-2 instruction set.

5 Performance

5.1 Runtimes

On an AT91SAM7S operating at 48 MHz and executing the optimized assembler multiplication function from within the internal SRAM, the average execution time for creating the signature is ~130 ms and within a range of 120 to 140 ms.

On an ATSAM3S operating at 48 MHz, the ECDSA signature on the p192r1 curve takes less than 100ms, typically about 95ms. Notice that on this controller, execution times can be reduced to approximately 75 ms when running at full speed, i.e. 64MHz.

Runtimes vary slightly, because there are certain conditional paths in the code which cannot be eliminated. Even the ARM instruction set, namely the multiplication instruction, has a varying runtime due to an early termination condition.

Please note that there is a further inherent uncertainty regarding the execution times, as the ECDSA signature algorithm might require a partial or full restart of the computations, depending on the generated random number. Refer to section 1.5.3 in the CompactECC Reference Manual [1] for details.

5.1.1 ARM

Runtimes were measured on an AT91SAM7S operating at 48 MHz and executing the optimized assembler multiplication function from within the internal SRAM. The average execution time for creating of the signature is ~130 ms and within a range of 120 to 140 ms.

5.1.2 AVR

Runtimes were measured on an ATmega128 microcontroller, running at 8 MHz. Signature creation takes approximately 8.2 s.

5.2 Memory requirements

To determine memory requirements, a small test application containing only the required functions for creating signatures as well as the necessary hardware initialization code and runtime library parts was created. User variables were allocated on the stack. Program data memory is mainly used by the C runtime, mainly by the pseudo-random number generator used for the testing application.

5.2.1 ARM

	Memory	Space requirement
Program code and init values	Flash	~ 13 kB
Program code (executed from RAM)	SRAM	160 bytes
Program data	SRAM	241 bytes
Stack	SRAM	~ 2.2 kB

Notice: These figures are for ARM7TDMI devices. Cortex-M3 devices typically require slightly less code memory and no program code is executed from SRAM.

5.2.2 AVR

	Memory	Space requirement
Program code and init values	Flash	~ 15 kB
Program data	SRAM	407 bytes
CStack	SRAM	844 bytes
RStack	SRAM	18 bytes

Please note that the exact amount of memory depends on the specific application. The size of the RStack (return address stack) depends on the compiler optimization settings as well, i.e. the number of inlined functions calls.

6 Implementation differences

In this section, differences of the optimized implementation compared to the generic CompactECC library are outlined.

6.1 Namespace

All classes are defined in the namespace “ceccf”.

6.2 Handling of big unsigned integer numbers and points

The `CBigUnsigned` class was modified to handle fixed-size integer numbers instead of providing variable storage for variable-spaced numbers, resulting in a different interface.

Points are stored in the class `CPoint`, which consists of two members of type modified type `CBigUnsigned`, representing the x and y coordinates.

Arithmetic operations on the class `CBigUnsigned` are not defined as operators. This is due to the fact that template argument deduction is not possible for return types in C++. Instead, they are implemented separately as static functions within the class `CBigUnsignedArithmetic`.

The implementation in a separate class was chosen because the class `CBigUnsigned` needs a template parameter specifying the width of the stored number. This parameter is not required for the static arithmetic member functions, but still needs to be specified when calling static member functions and will lead to multiple instantiations of the same member function for different (unused) parameters. This issue cannot be resolved by the linker and would lead to duplicate, identical code in the resulting application.

6.3 Finite fields

The optimized implementation does not make use of a separate finite field class. Instead, the relevant operations are embedded into the elliptic curve class.

6.4 Table multipliers

The original CompactECC implementation, as well as the first version of the optimized implementation, uses a sliding-window multiplication algorithm in conjunction with a table of pre-computed points to speed up multiplication on the elliptic curve by processing groups of 4 bits. This functionality was moved in separate classes to allow for further customization.

The class `CEllipticCurvePrecomputeMultiplier` will precompute points at runtime during instantiation and keep the table in the system RAM. This is the behaviour of the original implementation.

The class `CEllipticCurveTableMultiplier` will use the same table-based algorithm, but will not compute the table at runtime, but use a pre-computed table stored in non-volatile memory (flash).

The class `CEllipticCurveSimpleMultiplier` will use the straight, bitwise multiplication and is notably slower than the table-based algorithms. It is included for the sake of completeness.

6.5 Random numbers

The interface to the random number generator used by the Sign function is different, due to the different number representation.

6.6 Number conversion

As mentioned before, the generic CompactECC implementation [1] and the present optimized implementation use different storage classes for the big integer numbers, which require conversion between both formats if both implementations should be used in conjunction with each other.

Two simple functions are defined for this purpose:

- `ConvertBigUnsigned()` in the namespace `cecc`:
  Convert from a `ceccf::CBigUnsigned<N, MEM, ARCH>` into a `cecc::CBigUnsigned<T, T2>` with `T = ARCH::T` and `T2 = ARCH::T2`.

- `ConvertBigUnsigned()` in the namespace `ceccf`:
  Convert from a `cecc::CBigUnsigned<T, T2>` into a `ceccf::CBigUnsigned<N, MEM, ARCH>` with `T = ARCH::T` and `T2 = ARCH::T2`.

Refer to section 8.10 for details.

7 Usage

A short usage example for creating signatures is provided based on the secp192r1 curve.

7.1 Signature generation – ARM

- Instantiate the curve and a random number generator

```
// Instantiate the curve
CEllipticCurve192r1<> curve;

// Instantiate a random number generator (c.f. section 6.5 and 8.9)
CRandomNumberGenerator rng;
```

- Define the private key (pre-generated)

  Usually, the private key is stored somewhere in non-volatile memory and a `CBigUnsigned` instance must created:

```
// Create the private key from the unsigned int array anPrivateKey[]
// Digits (of unsigned int type) must be stored with the
// least-significant digit first

const unsigned int anPrivateKey[] = { 0x628f6ca5, 0x1e3a45db,
       0xf4b12d89, 0x594470af, 0x70791d12, 0xdeb8634e };

// Instantiate CBigUnsigned by using the given init values
CBigUnsigned<192> privateKey(anPrivateKey);

// As an alternative, a reference to storage in ROM can be created:
CBigUnsigned<192> &privateKeyRef =
       CBigUnsigned<192>::CreateReferenceToMemory(anPrivateKey);
```

- Generate the private key on-the-fly (testing only)

  Alternatively, for testing, the private key may be generated on the fly, by creating a random number and verifying that it meets the requirements (i.e. it must be in the range $[1, p-1]$)

```
CBigUnsigned<192> privateKey;

do
{
    rng(privateKey);
} while ((privateKey >= curve.m_p) || (privateKey == 0));
```

- Derive the public key from the private key (if required)

  The public key can be derived from the private key by multiplication on the curve:

```
CPoint<192> publicKey;
curve.Multiply(publicKey, privateKey);
```

- Calculate the hash of the message

The following fragment is only provided as an example for a message stored in the array abMessage. The user should calculate a cryptographic digest, e.g. SHA-1 or SHA-256. Note the SHA-1 is considered insecure.

```
CBigUnsigned<192> digest;

// Calculate the digest - to be be implemented by the user
CalculateDigest(abMessage, sizeof(abMessage), digest);
```

- Instantiate the ECDSA sign-only class
  Default parameters for the memory type can be used on ARM and do not need to be specified.

  - To use the pre-computed in-flash table for multiplication:

    ```
    CEllipticCurveDSASignOnly<CEllipticCurve192r1<>,
         CEllipticCurveTableMultiplier<CEllipticCurve192r1<> > >
         ecdsa(ec);
    ```

  - To use the pre-computed table in RAM (calculated during instantiation):

    ```
    CEllipticCurveDSASignOnly<CEllipticCurve192r1<>,
         CEllipticCurvePrecomputeMultiplier<CEllipticCurve192r1<> > >
         ecdsa(ec);
    ```

  - To use bitwise multiplication (slow):

    ```
    CEllipticCurveDSASignOnly<CEllipticCurve192r1<>,
         CEllipticCurvePrecomputeMultiplier<CEllipticCurve192r1<> > >
         ecdsa(ec);
    ```

- Sign the hash of the message

```
CBigUnsigned<192> r, s;

// Pass r, s to receive the signature
// the private key d, the message digest and the random number
// generator to generate k
ecdsa.Sign(r, s, d, digest, rng);
```

- (r, s) contain the signature of the message

7.2 Signature generation – AVR

- Instantiate the curve and a random number generator
  The memory type of the coefficient storage must be specified (memFlash).

```
// Instantiate the curve
CEllipticCurve192r1<memFlash> curve;

// Instantiate a random number generator (c.f. section 6.5 and 8.9)
CRandomNumberGenerator rng;
```

- Define the private key (pre-generated)

  Usually, the private key is stored somewhere in non-volatile memory and a `CBigUnsigned` instance must created:

```
// Create the private key from the unsigned char array abPrivateKey[]
// Digits (of unsigned char type) must be stored with the
// least-significant digit first

const unsigned char __flash abPrivateKey[] =
{
    0xa5, 0x6c, 0x8f, 0x62, 0xdb, 0x45, 0x1e, 0x3a,
    0x89, 0x2d, 0xf4, 0xb1, 0xaf, 0x70, 0x59, 0x44,
    0x12, 0x1d, 0x70, 0x79, 0x4e, 0x63, 0xde, 0xb8
};

// Instantiate CBigUnsigned by using the given init values
// The Create method has to be used and the memory type (in which
// the init values are stored), needs to be specified explicitly
CBigUnsigned<192> privateKey =
   CBigUnsigned<192>::Create<memFlash>(abPrivateKey);

// As an alternative, a reference to storage in flash can be created.
// The storage memory type memFlash needs to be specified explicitly.
const CBigUnsigned<192, memFlash> & privateKeyRef =
   CBigUnsigned<192, memFlash>::CreateReferenceToMemory(abPrivateKey);
```

- Generate the private key on-the-fly (testing only)

  Alternatively, for testing, the private key may be generated on the fly, by creating a random number and verifying that it meets the requirements (i.e. it must be in the range $[1, p-1]$)

```
CBigUnsigned<192> privateKey;

do
{
    rng(privateKey);
} while ((privateKey >= curve.m_p) || (privateKey == 0));
```

- Derive the public key from the private key (if required)

  The public key can be derived from the private key by multiplication on the curve:

```
CPoint<192> publicKey;
curve.Multiply(publicKey, privateKey);
```

- Calculate the hash of the message

  The following fragment is only provided as an example for a message stored in the array abMessage. The user should calculate a cryptographic digest, e.g. SHA-1 or SHA-256. Note the SHA-1 is considered insecure.

```
CBigUnsigned<192> digest;

// Calculate the digest - to be be implemented by the user
CalculateDigest(abMessage, sizeof(abMessage), digest);
```

- Instantiate the ECDSA sign-only class
  - To use the pre-computed in-flash table for multiplication:

```
CEllipticCurveDSASignOnly<
    CEllipticCurve192r1<memFlash>,
    CEllipticCurveTableMultiplier<
        CEllipticCurve192r1<memFlash>, memFlash>
> ecdsa(ec);
```

  - To use the pre-computed table in RAM (calculated during instantiation):

```
CEllipticCurveDSASignOnly<CEllipticCurve192r1<memFlash>,
    CEllipticCurvePrecomputeMultiplier<
        CEllipticCurve192r1<memFlash> >
> ecdsa(ec);
```

  - To use bitwise multiplication (slow):

```
CEllipticCurveDSASignOnly<CEllipticCurve192r1<memFlash>,
    CEllipticCurvePrecomputeMultiplier<
        CEllipticCurve192r1<memFlash> >
> ecdsa(ec);
```

- Sign the hash of the message

```
CBigUnsigned<192> r, s;

// Pass r, s to receive the signature
// the private key d, the message digest and the random number
// generator to generate k
ecdsa.Sign(r, s, d, digest, rng);
```

- (r, s) contain the signature of the message

7.3 Signature verification

Signature verification is not yet implemented in the optimized CompactECC implementation.

7.4 Class instantiation

Please note that the class `CEllipticCurvePrecomputeMultiplier` will pre-compute base points during its instantiation, which will consume a certain amount of time. Instantiation is only necessary once, not every time a signature is created.

8 Class Reference

8.1 CBigUnsigned

8.1.1 Declaration

```
template<unsigned int N,
         typename MEM = defaultMem,
         typename ARCH = defaultArch>
class CBigUnsigned : private MEM::baseClass;
```

Template parameters to the class template are the bit width of the number to be stored, the memory type where the class instance is stored in and the architecture definition for which the class is generated.

The architecture is predefined, depending on the compilation environment, and does not need to be specified, except for testing purposes.

The memory type has a default value as well. For ARM, there is no other memory type and this parameter never needs to be specified. For AVR, the default is system RAM and the pointer size depends on the chosen memory model. It can be overridden to generate a class instance e.g. in flash.

CBigUnsigned privately inherits from the per-memory defined base class to inherit the specified memory attributes, depending on the parameter MEM.

8.1.2 Typedefs

```
// declare ARCH types locally (short-hand)
typedef typename ARCH::T T;
typedef typename ARCH::T2 T2;
typedef typename ARCH::uint uint;
typedef typename ARCH::sint sint;

// typedef for the base type T, qualified with any memory attributes,
// e.g. __flash. Used to pass pointers or references to the base types
typedef typename MemAttrMapping<typename ARCH::T, MEM>::T qT;

// typedef for this class, including any memory attributes.
// Used when defining references to this class, which requires
// explicit specification of the memory attributes
typedef
typename MemAttrMapping<CBigUnsigned<N, MEM, ARCH>, MEM>::T classType;
```

8.1.3 Enums

```
enum {
     // the width in bits (specified by template parameter N)
     width = N,

     // the number of digits of type T in the underlying storage
     digits = (N + ARCH::digitBits - 1) / ARCH::digitBits,

     // short-hand for the maximum digit value
     maxDigit = static_cast<T>(-1)
};
```

8.1.4 Constructors

```
// Default constructor. Initializes the stored number to 0.
CBigUnsigned();

// Copy constructor
template<typename MEM2>
CBigUnsigned(const CBigUnsigned<N, MEM2, ARCH> &info);

// Copy constructor (non-equal number of digits)
// Allows assigning different-sized CBigUnsigned classes to each other
template<unsigned int NB, typename MEMB>
explicit CBigUnsigned(const CBigUnsigned<NB, MEMB, ARCH> &info);

// Initialize from array of unsigned int (N elements)
explicit CBigUnsigned(const T *info);

// Initialize from array of type T(N elements) of the given memory type
template<class MEM2>
static const classType
Create(const typename MemAttrMapping<T, MEM2>::T *info);

// Initialize from array of unsigned int (n elements, possibly n != N)
// if (n < N): most-significant digits (not contained in info)
// will be zeroed
// if (n > N): most-significant digits (contained in info, but not
// fitting into this instance) will be discarded
explicit CBigUnsigned(const T *info, const uint n);

// Initialize from array of type T (n elements, n !=N )
// from the given memory type
template<class MEM2>
static const classType
Create(const typename MemAttrMapping<T, MEM2>::T *info, const uint n);

// Special "constructor-like" function: Create a const reference to a
// pre-initialized raw in-memory instance, e.g. stored in ROM.
// (N unsigned int digits, least significant digit first)
static const classType & CreateReferenceToMemory(const qT *info);
```

8.1.5 Attributes

```
// The digit storage
// digit #0 is the least-significant digit
T m_nDigits[digits];
```

8.1.6 Operators

8.1.6.1 Assignment operators

```
template<typename MEM2>
CBigUnsigned<N, MEM, ARCH> &
operator= (const CBigUnsigned<N, MEM2, ARCH> &info);

// Assign CBigUnsigned of different size
// Most-significant digits not fitting in this class will be
// silently discarded.
template<unsigned int NB, typename MEM2>
CBigUnsigned<N, MEM, ARCH> &
operator= (const CBigUnsigned<NB, MEM2, ARCH> &info);

// Assign a single digit number to the least significant digit.
// Clears the other digits.
CBigUnsigned<N, MEM, ARCH> & operator= (const T n);
```

8.1.6.2 Comparison operators

```
// Comparison operators: equality/inequality
// Compare with  a single digit
bool operator==(const T digit) const;
bool operator!=(const T digit) const;

// Compare with another CBigUnsigned, possibly of different size
template<unsigned int NB, typename MEMB>
bool operator==(const CBigUnsigned<NB, MEMB, ARCH> &b) const;

template<unsigned int NB, typename MEMB>
bool operator!=(const CBigUnsigned<NB, MEMB, ARCH> &b) const;

// Comparion operators: "less than/greater than" and
// "less than/greater than or equal"
template<unsigned int NB, typename MEMB>
bool operator>=(const CBigUnsigned<NB, MEMB, ARCH> &b) const;

template<unsigned int NB, typename MEMB>
bool operator<=(const CBigUnsigned<NB, MEMB, ARCH> &b) const;

template<unsigned int NB, typename MEMB>
bool operator>(const CBigUnsigned<NB, MEMB, ARCH> &b) const;

template<unsigned int NB, typename MEMB>
bool operator<(const CBigUnsigned<NB, MEMB, ARCH> &b) const;
```

8.1.6.3 Other operators

```
// Index operator []: return by value (read-only)
// Bounds-checking will be done on the index value.
// 0 will be returned for digits which do not exist in this instantiated
// class, i.e. index >= N.
// This is done to ease operations on differently-sized numbers.
T operator[](const sint nIndex) const;

// Index operator []: return reference (read-write)
// It is only allowed to write to digits which actually exist.
qT & operator[](const sint nIndex);

// Compare this instance (A) and B
// returns -1 if A < B
//          0 if A == B
//         +1 if A > B
template<unsigned int NB, typename MEMB>
sint Compare(const CBigUnsigned<NB, MEMB, ARCH> &b) const;
```

Arithmetic operators (Add, Subtract etc.) are defined in the class `CBigUnsignedArithmetic`.

8.2 CBigUnsignedArithmetic

8.2.1 Declaration

```
template<typename ARCH>
class CBigUnsignedArithmetic;
```

8.2.2 Methods

```
// Calculates A = B + C and returns carry
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
static T Add(CBigUnsigned<NA, MEMA, ARCH> &A,
             const CBigUnsigned<NB, MEMB, ARCH> &B,
             const CBigUnsigned<NC, MEMC, ARCH> &C);

// Calculates A = B - C and returns borrow
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
static T Subtract(CBigUnsigned<NA, MEMA, ARCH> &A,
                  const CBigUnsigned<NB, MEMB, ARCH> &B,
                  const CBigUnsigned<NC, MEMC, ARCH> &C);


// Calculates A = B * C
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
static void Multiply(CBigUnsigned<NA, MEMA, ARCH> &A,
                     const CBigUnsigned<NB, MEMB, ARCH> &B,
                     const CBigUnsigned<NC, MEMC, ARCH> &C);

// Calculates a = b^2
template<unsigned int NA, unsigned int NB, typename MEMA, typename MEMB>
static void Square(CBigUnsigned<NA, MEMA, ARCH> &A,
                      const CBigUnsigned<NB, MEMB, ARCH> &B);

// Calculates a = c / d and b = c % d
template<unsigned int N, unsigned int NC, unsigned int ND,
         typename MEM, typename MEMC, typename MEMD>
static void DivideEx(CBigUnsigned<N, MEM, ARCH> &a,
                     CBigUnsigned<N, MEM, ARCH> &b,
                               const CBigUnsigned<NC, MEMC, ARCH> &c,
                               const CBigUnsigned<ND, MEMD, ARCH> &d);

// Calculates b = c % d
template<unsigned int NB, unsigned int NC, unsigned int ND,
         typename MEMB, typename MEMC, typename MEMD>
static void Modulo(CBigUnsigned<NB, MEMB, ARCH> &b,
                   const CBigUnsigned<NC, MEMC, ARCH> &c,
                           const CBigUnsigned<ND, MEMD, ARCH> &d);


// Calculates a = (b + c) mod d
template<unsigned int NA, unsigned int NB, unsigned int NC, unsigned int ND,
         typename MEMA, typename MEMB, typename MEMC, typename MEMD>
static void AddModulo(CBigUnsigned<NA, MEMA, ARCH> &A,
                      const CBigUnsigned<NB, MEMB, ARCH> &B,
                      const CBigUnsigned<NC, MEMC, ARCH> &C,
                      const CBigUnsigned<ND, MEMD, ARCH> &D);

// Calculates a = (b - c) mod d
template<unsigned int NA, unsigned int NB, unsigned int NC, unsigned int ND,
         typename MEMA, typename MEMB, typename MEMC, typename MEMD>
static void SubtractModulo(CBigUnsigned<NA, MEMA, ARCH> &A,
                           const CBigUnsigned<NB, MEMB, ARCH> &B,
                           const CBigUnsigned<NC, MEMC, ARCH> &C,
                           const CBigUnsigned<ND, MEMD, ARCH> &D);
```

```
// Calculates a = b - c * d (where c is a digit) and returns borrow
template<unsigned int NA, unsigned int NB, unsigned int ND,
         typename MEMA, typename MEMB, typename MEMD>
static T SubtractMultiply(CBigUnsigned<NA, MEMA, ARCH> &a,
                          const CBigUnsigned<NB, MEMB, ARCH> &b,
                          const T c,
                          const CBigUnsigned<ND, MEMD, ARCH> &d);

// Calculates a = 1/b mod c;
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
static void InverseModulo(CBigUnsigned<NA, MEMA, ARCH> &a,
                          const CBigUnsigned<NB, MEMB, ARCH> &b,
                          const CBigUnsigned<NC, MEMC, ARCH> &c);

// Calculates a = b * 2^c and returns carry
template<unsigned int NA, unsigned int NB, typename MEMA, typename MEMB>
static T ShiftLeft(CBigUnsigned<NA, MEMA, ARCH> &a,
                   const CBigUnsigned<NB, MEMB, ARCH> &b,
                   const T c);

// Calculates a = b * 2^(-c) and returns carry
template<unsigned int NA, unsigned int NB, typename MEMA, typename MEMB>
static T ShiftRight(CBigUnsigned<NA, MEMA, ARCH> &a,
                    const CBigUnsigned<NB, MEMB, ARCH> &b,
                    const T c);
```

8.2.3 Optimized assembler implementations for ARM

The following methods are available as optimized ARM assembler implementations:

- `CBigUnsignedArithmetic<ArchARM>::Add()`
  for NA=192, NB=192, NC=192
  using the Thumb instruction set

- `CBigUnsignedArithmetic<ArchARM>::Multiply()`
  for NA=384, NB=192, NC=192
  using the ARM instruction set

- `CBigUnsignedArithmetic<ArchARM>::Square()`
  for NA=384 NB=192
  is mapped to `CBigUnsignedArithmetic<ArchARM>::Multiply()`

All Assembler functions are implemented in the syntax of the IAR ARM Assembler, Version 5.x. For optimal performance, the `CBigUnsignedArithmetic::Multiply()` function should be executed from internal SRAM on AT91SAM7S/X microcontrollers for processor clocks higher than 30 MHz. By default, the code is placed into the section '.textrw' with the attributes 'code read/write'.

The linker configuration file should include an instruction like

```
place in RAM_region { section .textrw };
initialize by copy { readwrite };
```

This will result in a correct placement of the object code into SRAM, with an initialization copy in flash memory. The IAR DLIB initialization code will copy the code into SRAM during start-up without user intervention.

8.3 CPoint

8.3.1 Declaration

```
template<unsigned int N,
         typename MEM = defaultMem,
         typename ARCH = defaultArch>
class CPoint : private MEM::baseClass;
```

8.3.2 Typedefs

```
// The digit type
typedef typename ARCH::T T;

// The digit type, fully qualified with memory attributes
typedef typename MemAttrMapping<typename ARCH::T, MEM>::T qT;

// The class type of this class, fully qualified with memory attributes
typedef typename MemAttrMapping<CPoint<N, MEM, ARCH>, MEM>::T classType;
```

8.3.3 Constructors

```
// Default constructor
CPoint();

// Copy constructor
template<typename MEM2>
CPoint(const CPoint<N, MEM2, ARCH> &info);

// Initialize with given X, Y coordinates
template<typename MEM2>
CPoint(const CBigUnsigned<N, MEM2, ARCH> &x,
       const CBigUnsigned<N, MEM2, ARCH> &y);

// Initialize with raw digit data for X, Y coordinates (N digits each)
CPoint(const unsigned int *px, const unsigned int *py);

// Special "constructor-like" function: Create a reference to a
// pre-initialized raw in-memory instance, e.g. stored in ROM.
static const classType & CreateReferenceToMemory(const qT *info);
```

8.3.4 Attributes

```
// X and Y coordinates
CBigUnsigned<N, MEM, ARCH> m_x;
CBigUnsigned<N, MEM, ARCH> m_y;
```

8.3.5 Operations

```
// Compare two points for equality
template<typename MEM2>
bool operator==(const CPoint<N, MEM2, ARCH> &info) const;
```

8.4 CEllipticCurve192r1

8.4.1 Declaration

```
template<typename MEM = defaultMem, typename ARCH = defaultArch>
class CEllipticCurve192r1;
```

The template parameter `MEM` defines the storage memory type for the curve's coefficients. The parameter `ARCH` defines the architecture. The default value is automatically set depending on the compilation environment.

8.4.2 Typedefs

```
typedef ARCH ECARCH;
typedef MEM ECMEM;
```

8.4.3 Enums

```
enum {
     R = 192,                   // order of the curve
     Rd = R / ARCH::digitBits  // corrosponding number of digits of type T
};
```

8.4.4 Constructors

```
CEllipticCurve192r1();
```

8.4.5 Attributes

```
// prime modulus
const typename CBigUnsigned<R, MEM, ARCH>::classType &m_p;

// curve's coefficients, a
const typename CBigUnsigned<R, MEM, ARCH>::classType &m_a;

// curve's coefficients, b
const typename CBigUnsigned<R, MEM, ARCH>::classType &m_b;

// base point, a point on e of order r
const typename CPoint<R, MEM, ARCH>::classType &m_g;

// a positive, prime integer dividing the number of points on e
const typename CBigUnsigned<R+1, MEM, ARCH>::classType &m_r;

static const bool m_bAIsZero = false;
static const bool m_bAIsMinus3 = true

// attributes of coefficient A
// enables certain optimizations if A is either 0 or -3 (modulo m_p)
static const bool m_bAIsMinus3 = true;
static const bool m_bAIsZero = false;
```

8.4.6 Operations

```
// P0 = P1 + P2
template<typename MEM0, typename MEM1, typename MEM2>
void Add(CPoint<R, MEM0, ARCH> &p0,
         const CPoint<R, MEM1, ARCH> &p1,
         const CPoint<R, MEM2, ARCH> &p2) const;

// P0 = n * P1 (scalar point multiplication)
template<typename MEMP0, typename MEMN, typename MEMP1>
void Multiply(CPoint<R, MEMP0, ARCH> &p0,
              const CBigUnsigned<R, MEMN, ARCH> &n,
              const CPoint<R, MEMP1, ARCH> &p1) const;
```

```
// P0 = n * P1 (scalar point multiplication), with P1 = G (m_g)
template<typename M0, typename MEMN>
void Multiply(CPoint<R, M0, ARCH> &p0,
              const CBigUnsigned<R, MEMN, ARCH> &n) const;

// (P0,Z0) = (P1,Z1) + (P2,Z2) in Jacobian projective coordinate space
template<typename MEMP0, typename MEMZ0, typename MEMP1,
         typename MEMZ1, typename MEMP2, typename MEMZ2>
void AddProjective(CPoint<R, MEMP0, ARCH> &p0,
                   CBigUnsigned<R, MEMZ0, ARCH> &z0,
                   const CPoint<R, MEMP1, ARCH> &p1,
                   const CBigUnsigned<R, MEMZ1, ARCH> &z1,
                   const CPoint<R, MEMP2, ARCH> &p2,
                   const CBigUnsigned<R, MEMZ2, ARCH> &z2) const;

// (P0,Z0) = 2*(P1,Z1)
template<typename MEMP0, typename MEMZ0, typename MEMP1, typename MEMZ1>
void DoubleProjective(CPoint<R, MEMP0, ARCH> &p0,
                      CBigUnsigned<R, MEMZ0, ARCH> &z0,
                      const CPoint<R, MEMP1, ARCH> &p1,
                      const CBigUnsigned<R, MEMZ1, ARCH> &z1) const;

// Converts back from projective to affine coordinates
template<typename MP, typename MZ>
void MakeAffine(CPoint<R, MP, ARCH> &p, CBigUnsigned<R, MZ, ARCH> &z) const;

// Calculates a = (b + c) mod m_p
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
void AddModulo(CBigUnsigned<NA, MEMA, ARCH> &A,
               const CBigUnsigned<NB, MEMB, ARCH> &B,
               const CBigUnsigned<NC, MEMC, ARCH> &C) const;

// Calculates a = (b - c) mod m_p
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
void SubtractModulo(CBigUnsigned<NA, MEMA, ARCH> &A,
                    const CBigUnsigned<NB, MEMB, ARCH> &B,
                    const CBigUnsigned<NC, MEMC, ARCH> &C) const;

// Calculates a = b * c mod m_p
template<unsigned int NA, unsigned int NB, unsigned int NC,
         typename MEMA, typename MEMB, typename MEMC>
void MultiplyModulo(CBigUnsigned<NA, MEMA, ARCH> &a,
                    const CBigUnsigned<NB, MEMB, ARCH> &b,
                    const CBigUnsigned<NC, MEMC, ARCH> &c) const;

// Calculates a = b^2 mod m_p
template<unsigned int NA, unsigned int NB, typename MEMA, typename MEMB>
void SquareModulo(CBigUnsigned<NA, MEMA, ARCH> &a,
                  const CBigUnsigned<NB, MEMB, ARCH> &b) const;

// Calculates a = 1/b mod m_p
template<unsigned int NA, unsigned int NB, typename MEMA, typename MEMB>
void InverseModulo(CBigUnsigned<NA, MEMA, ARCH> &a,
                   const CBigUnsigned<NB, MEMB, ARCH> &b) const;
```

8.5 CEllipticCurveDSASignOnly

8.5.1 Declaration

```
template<typename EC, typename MUL>
class CEllipticCurveDSASignOnly;
```

`EC` defines the type of elliptic curve to use. Only `CEllipticCurve192r1` is available at the moment.

`MUL` defines the multiplier to use. Available multiplier and their characteristics are described in section 6.4.

8.5.2 Constructor

```
// Construct CEllipticCurveDSASignOnly using the given curve
CEllipticCurveDSASignOnly(EC &ec);
```

8.5.3 Attributes

```
// The elliptic curve of type EC
EC &m_ec;
```

8.5.4 Operations

```
// Sign the message.
// (r, s) is the ECC signature, d is the private key,
// digest is an appropriate hash value (MD5, SHA),
// rng is a function object providing a cryptographically
// strong random number.
template<typename MEMR, typename MEMS, typename MEMD,
         typename MEMDIG, typename RNG>
void Sign(CBigUnsigned<EC::R, MEMR, ARCH> &r,
          CBigUnsigned<EC::R, MEMS, ARCH> &s,
          const CBigUnsigned<EC::R, MEMD, ARCH> &d,
          const CBigUnsigned<EC::R, MEMDIG, ARCH> &digest,
          RNG &rng) const;
```

8.6 CEllipticCurvePrecomputeMultiplier

8.6.1 Declaration

```
template<typename EC>
class CEllipticCurvePrecomputeMultiplier;
```

`EC` defines the elliptic curve to use. Currently, only `CEllipticCurve192r1` is available.

8.6.2 Enums

```
enum {
    R = EC::R,
    Rd = EC::Rd,
    window = 4,  // size of the sliding window
    points = (1 << window)-1,
    windowsPerDigit = ARCH::digitBits / window
};
```

8.6.3 Constructor

```
template<typename MEM>
CEllipticCurvePrecomputeMultiplier(const EC &ec,
                                   const CPoint<R, MEM, ARCH> &p);
```

`ec` is the instance of the elliptic curve. `p` defines the base point on the elliptic curve (curve parameter G). A set of base points will be computed during instantiation.

8.6.4 Attributes

```
// pre-computed masks for the sliding window
unsigned int m_masks[windowsPerDigit];

// pre-computed base points
CPoint<R, memGen, ARCH> m_basePoints[points];
```

8.6.5 Operations

```
// P0 = n * basepoint
// (scalar point multiplication, optimized sliding window algorithm)
// use the precomputed points m_basePoints
template<typename MEMP, typename MEMN>
void Multiply(const EC &ec, CPoint<R, MEMP, ARCH> &p0,
              const CBigUnsigned<R, MEMN, ARCH> &n) const;
```

8.7 CEllipticCurveTableMultiplier

8.7.1 Declaration

```
template<typename EC, typename MEM = typename EC::ECMEM>
class CEllipticCurveTableMultiplier;
```

`EC` defines the elliptic curve to use. Currently, only `CEllipticCurve192r1` is available.

`MEM` defines the memory storage type for the pre-computed table and defaults to the storage type of the elliptic curve parameters.

8.7.2 Enums

```
enum {
    R = EC::R,
    Rd = EC::Rd,
    window = 4,  // size of the sliding window
    points = (1 << window)-1,
    windowsPerDigit = ARCH::digitBits / window
};
```

8.7.3 Constructor

```
CEllipticCurveTableMultiplier(const EC &ec,
                              const CPoint<R, ECMEM, ARCH> &p);
```

`ec` is the instance of the elliptic curve. `p` defines the base point on the elliptic curve (curve parameter G). It is defined for compatibility with the other multiplier classes (same constructor signature), but is not used, because the table is pre-computed and stored in non-volatile memory.

8.7.4 Attributes

```
// the sliding window masks - computed in the contructor
unsigned int m_masks[windowsPerDigit];

// pointer to base points (CPoint)
const CPoint<R, MEM, ARCH> *m_pBasePoints;

// the backed storage for m_pBasePoints (init values - type ARCH::T)
static const typename MemAttrMapping<typename ARCH::T, MEM>::T basePoints[];
```

8.7.5 Operations

```
// P0 = n * basepoint
// (scalar point multiplication, optimized sliding window algorithm)
// use the precomputed points m_basePoints
template<typename MEMP0, typename MEMN>
void Multiply(const EC &ec, CPoint<R, MEMP0, ARCH> &p0,
              const CBigUnsigned<R, MEMN, ARCH> &n) const;
```

8.8 CEllipticCurveSimpleMultiplier

8.8.1 Declaration

```
template<typename EC>
class CEllipticCurveSimpleMultiplier;
```

`EC` defines the elliptic curve to use. Currently, only `CEllipticCurve192r1` is available.

8.8.2 Enums

```
enum {
     R = EC::R,
     Rd = EC::Rd
};
```

8.8.3 Constructor

```
CEllipticCurveSimpleMultiplier(const EC &ec,
                               const CPoint<R, ECMEM, ARCH> &p);
```

`ec` is the instance of the elliptic curve, `p` defines the base point on the elliptic curve (curve parameter G).

8.8.4 Attributes

```
// Stores the point G on the curve
const CPoint<R, ECMEM, ARCH> &m_p;
```

8.8.5 Operations

```
template<typename MEMP, typename MEMN>
void Multiply(const EC &ec, CPoint<R, MEMP, ARCH> &p0,
              const CBigUnsigned<R, MEMN, ARCH> &n) const;
```

8.9 Random number generator

The random number generator provided to the Sign method of `CEllipticCurveDSASignOnly` must be of class type and provide the “function call operator” operator() to generate a random number:

```
class CRandomNumberGenerator
{
    // Operations
    public:
        template<unsigned int N>
        void operator()(CBigUnsigned<N> &random);
};
```

8.10 Number conversion

Two conversion functions are defined to allow for easy conversion between the two implementations. The conversion functions are simple wrappers around special constructors in the two implementations of the `CBigUnsigned` class.

```
// Conversion functions: convert B into A
namespace cecc
{
    // Convert ceccf::CBigUnsigned<N> B into
    // cecc::CBigUnsigned<T, T2> A
    template<unsigned int N, typename MEM, typename ARCH>
    void ConvertBigUnsigned(
                CBigUnsigned<typename ARCH::T, typename ARCH::T2> &A,
                const ceccf::CBigUnsigned<N, MEM, ARCH> &B)
    {
         A = CBigUnsigned<typename ARCH::T, typename ARCH::T2>
              (B.m_nDigits, N / (sizeof(typename ARCH::T) * 8));
    }
}

namespace ceccf
{
    // Convert cecc::CBigUnsigned<T, T2> B into
    // ceccf::CBigUnsigned<N> A
    template<unsigned int N, typename MEM, typename ARCH>
    void ConvertBigUnsigned(
        CBigUnsigned<N, MEM, ARCH> &A,
        const cecc::CBigUnsigned<typename ARCH::T, typename ARCH::T2> &B)
    {
        A = CBigUnsigned<N>(B.GetData(), B.GetCount());
    }
}
```

9 File list

- Header files

  BigUnsignedF.h
  CompactECCF.h
  CompactECDSAF.h
  CompactECCFArch.h
  ConvertBigUnsigned.h

- C++ source files

  CompactECCF.cpp

- Assembler source files

  Add_6_6_6.s
  Multiply_12_6_6.s

10 References

[1] CompactECC Reference Manual, Revision 1.4, ubisys technologies GmbH

[2] AVR® IAR C/C++ Compiler Reference Guide, IAR Systems

11 Revision History

Revision	Date	Changes
1.0	28th August 2009	Initial Version
1.1	28th October 2009	Included architecture and memory type abstraction to support the AVR 8-bit architecture. Adapted document title.
1.2	18th July 2010	Updated for Cortex-M3, ATSAM3S

12 License

13 Contact

UBISYS TECHNOLOGIES GMBH
HARDWARE AND SOFTWARE DESIGN
ENGINEERING AND CONSULTING
AM WEHRHAHN 45
40211 DÜSSELDORF
GERMANY
T: +49. 211. 54 21 55 00
F: +49. 211. 54 21 55 99
www.ubisys.de
info@ubisys.de