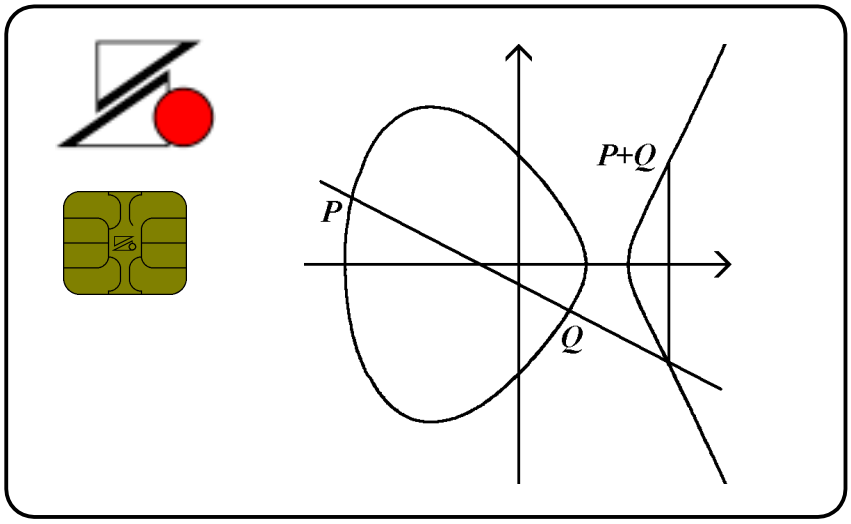
Elliptic Curves in the BasicCard

Tony Guilfoyle
ZeitControl cardsystems GmbH
e-mail: TonyGuil@aol.com
14th February 2000



Abstract: This paper describes the implementation of Elliptic Curve Cryptography in ZeitControl's Enhanced BasicCard, an inexpensive programmable smart card. The full BasicCard documentation and development software is available, free of charge, from our web site at www.BasicCard.com.

Note: This paper assumes familiarity with IEEE proposed standard P1363.

ZeitControl cardsystems GmbH

Siedlerweg 39
D-32429 Minden
Germany

Tel: +49 (0) 571-50522-0
Fax: +49 (0) 571-50522-99
Web sites:
http://www.zeitcontrol.de
http://www.basiccard.com

1 Introduction

ZeitControl's BasicCards were developed with the aim of making the programming of smart card applications as simple and inexpensive as possible. There are two types of BasicCard: the Compact BasicCard, with 1K of EEPROM, and the Enhanced BasicCard, with up to 16K of EEPROM. Elliptic Curve Cryptography requires the Enhanced BasicCard; the entry level version, the Enhanced BasicCard ZC3.3 with 8K of EEPROM, is available for $3.95 in small quantities. Enhanced BasicCard versions ZC3.5 and ZC3.6 contain the Elliptic Curve Fast Signature Algorithm (**EC-FSA**), which enables them to perform an Elliptic Curve digital signature in 1.2 seconds.

A BasicCard application is programmed in ZeitControl's ZC-Basic language. This language is similar to Microsoft® Visual Basic, with special features to facilitate the implementation of ISO-compatible commands. An application will typically consist of two programs: the Terminal program, running in a PC and issuing commands; and the BasicCard program, running in a BasicCard and executing those commands. The commands are compatible with ISO standard 7816-4, so it is not necessary to program both ends in ZC-Basic. For instance, you can write a ZC-Basic Terminal program to talk to a German Geldkarte; or you can program a BasicCard in ZC-Basic to respond to ISO commands from a card reader in a bank. (We have done both these things; fortunately the highly-secure Geldkarte system offered us no temptations that we had to resist.)

So ZC-Basic is an amphibious language, running equally happily in a PC or in a BasicCard. With a few exceptions, the Elliptic Curve library presents the same user interface on both platforms.

2 Elliptic Curves in *Basic?*

Basic is a much-denigrated language. It has been estimated (by Microsoft, admittedly) that 70% of all programs are written in Basic; but I have never met anybody who claims to be a Basic programmer. Perhaps I move in the wrong circles. In any case, when ZeitControl came to me and asked if I could make them a Basic card, my immediate response was, "Surely you mean Java card?" "No, no", they said, "we want to build a Volkswagen, not a Ferrari! Look out of your window – how many Ferraris do you see in the street?" And they were right. Basic has proven to be perfectly adequate for smart card applications, and free of all the implementation headaches associated with Java cards.

In particular, the ZC-Basic language copes admirably with the data elements used in Elliptic Curve Cryptography. The algorithms themselves are written in other languages (C++ in the PC, and 8051 Assembly Language in the BasicCard), but session keys and curve parameters and such like can be handled quite naturally in Basic. A variable of type **String** is stored as a length byte *n* followed by *n* bytes of data; so it can hold binary data, including nulls. A fixed-length **String*** *n* type is also available. And ZC-Basic supports user-defined types, or structures. So, for example, we can define a data type that contains all the curve parameters:

```
Type EC161DomainParams
  a as Byte
  b As String*21
  r As String*21
  k As Byte
  G As String*22
End Type
```

This definition comes from the file EC–161.DEF, supplied with the BasicCard development kit.

The Elliptic Curve library is not contained in ROM; it is loaded into EEPROM with the ZC-Basic statement:

```
#Include EC–161.DEF
```

The Elliptic Curve library (together with the Secure Hash Algorithm library, which it uses) occupies about 5 kilobytes of EEPROM. Even in the 8-kilobyte Enhanced BasicCard, this leaves plenty of room for a simple application; and more complex applications are possible in the 16-kilobyte version.

3 EC–161: An Overview

The **EC–161** library implements 161-bit Elliptic Curve Cryptography, for Terminal programs and Enhanced BasicCard programs. The following operations are supported:

- private/public key pair generation;
- shared secret derivation;
- session key generation;
- digital signature generation;
- digital signature verification (Terminal program only).

Here is a list of the procedures in the **EC–161** 161-bit Elliptic Curve library. The definitions (in **bold** type) are presented in standard Basic. For those unfamiliar with the Basic language, there are two ways to specify the data type of a function or parameter:

- with the final character of the function or parameter name (here only '$' occurs, specifying a **String** type);
- with '**As** *type*' following the declaration.

Initialisation Procedures

Sub EC161SetCurve (*Params* **As EC161DomainParams)**

Specify the domain parameters for the Elliptic Curve. (These will usually come from a file generated by ZeitControl software, but this is not required.) See previous section for the definition of type **ECDomainParams**.

Note: This procedure is required only in Terminal programs, not BasicCard programs.

Sub EC161GenerateKeyPair (*Seed$***)**

Generate a private-key/public-key pair, saving the results in the public **String** variables **EC161PrivateKey** and **EC161PublicKey**.

Sub EC161SetPrivateKey (*Key$***)**

Set a specific private key, and generate the corresponding public key, saving the results in the public **String** variables **EC161PrivateKey** and **EC161PublicKey**.

Cryptographic Schemes

Function EC161SessionKey (*KDP$***,** *SharedSecret$***) As String**

Generate a 20-byte session key from the Key Derivation Parameters *KDP$* and the given *SharedSecret$*. This is the P1363 scheme **ECKAS-DH1**: Elliptic Curve Key Agreement Scheme, Diffie-Hellman version, where each party contributes one key pair.

Sub EC161HashAndSign (*Signature$***,** *Message$***) As String**

Compute the SHA-1 hash of the given message, and sign it with **EC161Sign**. This is the Signature Generation Operation of P1363 scheme **ECSSA**: Elliptic Curve Signature Scheme with Appendix.

Function EC161HashAndVerify (*Signature$***,** *Message$***,** *PublicKey$***)**

Compute the SHA-1 hash of the given message, and verify the signature with **EC161Verify**. This is the Signature Verification Operation of P1363 scheme **ECSSA**: Elliptic Curve Signature Scheme with Appendix.

Cryptographic Primitives

Function EC161SharedSecret (*PublicKey$***) As String**

> Generate the shared secret corresponding to the given *PublicKey$*. This is the P1363 primitive **ECSVDP-DH**: Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version.

Sub EC161Sign (*Signature$***,** *Hash$***) As String**

> Compute the signature for the given *Hash$* value. This is the P1363 primitive **ECSP-NR**: Elliptic Curve Signature Primitive, Nyberg-Rueppel version.

Function EC161Verify (*Signature$***,** *Hash$***,** *PublicKey$***)**

> Verify a signature for the given *Hash$* value. This is the P1363 primitive **ECVP-NR**: Elliptic Curve Verification Primitive, Nyberg-Rueppel version.

In the descriptions of the individual libraries, error codes may be defined. These error codes are signalled via the ZC-Basic variable **LibError**. This variable contains the most recent error code signalled by a library procedure.

4 EC–161: The Elliptic Curve Library

This section describes the procedures in more detail. (For a full explanation of the ZC-Basic language, download the BasicCard documentation from `www.BasicCard.com`.) To load the Elliptic Curve library:

#Include EC–161.DEF

The file EC–161.DEF is supplied with the distribution kit, in the `BasicCrd\Lib` directory.

Setting the Elliptic Curve Parameters

An Elliptic Curve is defined by its EC Domain Parameters; three suitable Elliptic Curves are supplied in the directory `BasicCrd\Lib`. Choose one of these at random for your application. Files EC161-1.CRV through EC161-3.CRV contain curve definitions in ZC-Basic, for inclusion in a source program. File EC-161.BIN contains the binary data for all three curves, for run-time loading in a Terminal program.

To specify an Elliptic Curve in an Enhanced BasicCard program:

#Include EC161-*X*.DEF

where *X* is a number from 1 to 3. In a BasicCard program, the curve must be chosen at compile time; it can't be re-loaded at run-time.

In the Terminal program, an Elliptic Curve must be explicitly loaded using **EC161SetCurve**. There are three ways of doing this:

- If you know in advance which curve to use, you can include its definition file. For example:

  ```
  #Include EC161-3.DEF
  Call EC161SetCurve (EC161Params)
  ```

  But note that only one such definition file is allowed in a program.

- If the card has a suitable command, you can load the curve from the card. For example:

  ```
  Private Curve As EC161DomainParams
  Call GetCurve (Curve) : Call CheckSW1SW2()
  Call EC161SetCurve (Curve)
  ```

  See `BasicCrd\Examples\EC` for an example of this.

- You can read the curve from the binary file EC-161.BIN. For example:

```
Private Curve As EC161DomainParams
Open "EC-161.BIN" For Random As #1 Len=64
Get #1, 3, Curve ' Read Elliptic Curve #3
Close #1
Call CheckFileError()
Call EC161SetCurve (Curve)
```

If the EC domain parameters are invalid, procedure **EC161SetCurve** returns error code **EC161BadCurveParams** in variable **LibError**.

In the Terminal program, you must call **EC161SetCurve** before you call any other procedures from the **EC–161** library. If not, error code **EC161CurveNotInitialised** will be returned in variable **LibError**.

Key Generation

To generate a public/private key pair:

Call EC161GenerateKeyPair (*Seed$***)**

This procedure uses Secure Hash Algorithm library **SHA–1** to generate a cryptographically strong pseudo-random number from *Seed$*, for use as a private key. The 21-byte private key and its associated 22-byte public key are stored in **Eeprom** strings **EC161PrivateKey** and **EC161PublicKey**.

See the BasicCard documentation for more about pseudo-random numbers in **SHA–1**.

Setting an Explicit Private Key

To set an explicit value for a private key:

Call EC161SetPrivateKey (*Key$***)**

This procedure copies *Key$* (reduced modulo *r*) to the 21-byte **Eeprom** string **EC161PrivateKey**, and computes the associated 22-byte **Eeprom** string **EC161PublicKey**. (Here *r* is the order of point *G* – see **5 Binary Representation Formats**: *EC Domain Parameters*.)

If *Key$* is zero modulo *r*, error code **EC161BadProcParams** is returned in variable **LibError**.

Note: In Enhanced BasicCard versions ZC2.3 and ZC2.4, this procedure takes about 2 seconds to execute at a clock speed of 3.57 MHz. However, in the **EC-FSA** cards, it can take up to 7 seconds. But if you don't need to compute **EC161PublicKey**, you can simply copy *Key$* to the public variable **EC161PrivateKey**, and the Elliptic Curve routines will work correctly.

Generating a Digital Signature

A private key is used to generate digital signatures. To sign a message consisting of a **String** expression:

Call EC161HashAndSign (*Signature$***,** *Message$***)**

This subroutine returns a 42-byte string in the *Signature$* parameter.

To sign a longer message, first compute the hash function for the message (see the Secure Hash Algorithm library **SHA–1** in the BasicCard documentation), and then

Call EC161Sign (*Signature$***,** *Hash$***)**

If no private key has been set, these functions return error code **EC161KeyNotInitialised** in variable **LibError**.

In Enhanced BasicCard versions ZC2.3 and ZC2.4, digital signature generation takes about 2.5 seconds at a clock speed of 3.57 MHz. In the **EC-FSA** cards, it takes about 1.2 seconds.

Verifying a Digital Signature

Note: Verification of Digital Signatures is only possiible in a Terminal program. It is not supported in the Enhanced BasicCard.

To verify a digital signature, you need the signer's public key. To verify the signature of a message consisting of a **String** expression:

Status **= EC161HashAndVerify (***Signature$***,** *Message$***,** *PublicKey$***)**

Signature$ The 42-byte signature to be verified

Message$ The message that was signed

PublicKey$ The signer's 22-byte public key

This function returns **True** of **False** according to whether the signature is valid or not.

To verify a longer message, first compute the hash function for the message (see the Secure Hash Algorithm library **SHA–1** in the BasicCard documentation), and then verify its signature with the function:

Status **= EC161Verify (***Signature$***,** *Hash$***,** *PublicKey$***)**

If *Signature$* is not 42 bytes, or *PublicKey$* is not 22 bytes, error code **EC161BadProcParams** is returned in variable **LibError**.

Session Key Generation

If two parties know each other's public keys, they can use them to agree on a secret 21-byte value. This value is called the *shared secret* for the two parties; to compute it, you need to know the private key of one party (either one will do) and the public key of the other party. To compute the shared secret:

*SharedSecret$* = **EC161SharedSecret (***PublicKey$***)**

PublicKey$ The other party's 22-byte public key

SharedSecret$ The 21-byte shared secret

If *PublicKey$* is not 22 bytes long, or it is not a point on the curve, error **EC161BadProcParams** is returned in variable **LibError**.

This shared secret can then be used to generate 20-byte session keys for encrypting messages between the two parties; unlike the shared secret, a session key can be different on different occasions.

To generate a session key, the parties must agree on a *Key Derivation Parameter*, which can be any sequence of bytes, and need not be kept secret. For maximum security, it should be different each time a session key is generated. For example, it might be a standard header followed by the date and time. To generate the session key:

*SessionKey$* = **EC161SessionKey (***KDP$***,** *SharedSecret$***)**

KDP$ Key Derivation Parameter, a string of any length

SharedSecret$ The shared secret value, returned by **EC161SharedSecret**

SessionKey$ The 20-byte session key

Note: In the BasicCard, generating a shared secret takes about 7 seconds at a clock speed of 3.57 MHz. But once a shared secret has been generated for a given public key, session key generation takes less than half a second at the same clock speed, provided **Len(***KDP$***)** <= 42. (Typically, a smart card application will only need to generate session keys for a single public key, for which the shared secret is computed just once in the card's lifetime.)

5 Binary Representation Formats

This section specifies the binary representations of the data objects that are used in the library: integers, field elements, elliptic curves, points on the curve, and signatures.

Integers

Integers in this implementation have a length of either 1 byte or 21 bytes. The first (or leftmost) byte is the most significant – in a 21-byte integer, it contains bits 167-160. The last (or rightmost) byte contains bits 7-0.

Field Elements

The library **EC–161** implements operations on Elliptic Curves over the field **GF**(2^m), with m = 168. An element of **GF**(2^m) is represented by 168 bits stored in 21 bytes. The field representation is non-standard (i.e. it does not use a Polynomial Basis or a Normal Basis); for this reason we provide source code, in C and ZC-Basic, for converting between ZeitControl's EC-161 representation and a standard Polynomial Basis representation. This Polynomial Basis representation uses irreducible field polynomial

$$p(t) = t^{168} + t^{15} + t^3 + t^2 + 1$$

The source code is in directory `BasicCrd\Source\FldConv` in the BasicCard development kit.

EC Domain Parameters

An Elliptic Curve E over **GF**(2^m) is defined by an equation of the form

$$y^2 + xy = x^3 + ax^2 + b$$

where a and b are elements of **GF**(2^m) with $b \neq 0$. The curve E consists of all points (x, y) with $x, y \in$ **GF**(2^m) that satisfy this equation, together with a *Point at Infinity*, denoted $\mathcal{O}$. The order $\#E$ of the curve is the number of points in E. For cryptographic purposes, this order must have a large prime divisor, i.e. $\#E = kr$ for some (large) prime r. As well as a, b, r, and k, a point $G \in E$ must be specified, of order r (that is, r is the smallest positive integer such that $rG = \mathcal{O}$.) Field elements a and $b \in$ **GF**(2^m), integers r and k, and point $G \in E$ constitute the *EC domain parameters*. (k is redundant, as it can be calculated from a, b, and r; it is included for convenience.)

The library **EC–161** accepts any set of EC domain parameters (a, b, r, k, G) that satisfies the following conditions:

- a is zero in all bit positions except for bits 86-80 ;
- r is exactly 161 bits long, i.e. $2^{160} < r < 2^{161}$;
- k is a single byte, equal to 2 modulo 4.

The user-defined type **EC161DomainParams**, defined in file `BasicCrd\Lib\EC-161.DEF`, contains curve parameters a (1 byte), b (21 bytes), r (21 bytes), k (1 byte), and G (22 bytes), for a total of 66 bytes.

Points on the Curve

Points on the curve play two roles in library **EC–161**:

- EC domain parameter G is a point on the curve;
- every public key is a point on the curve. (For a private key s, the corresponding public key is sG.)

If P is on the curve and $x_P \neq 0$, then $y^2 + x_P y = x_P^3 + a x_P^2 + b$ has two solutions, y_0 and y_1. Moreover, the two expressions y_0 / x_P and y_1 / x_P differ only in bit 7 (in the representation used here); so if we know x_P and bit 7 of y_P / x_P , we can recover point P in full. This bit is called the *compressed y-coordinate* of

the point *P*, denoted $y)_P$. A point *P* on the curve is represented by 22 bytes, with x_P in the leftmost 21 bytes (i.e. bits 175-8), and the compressed y-coordinate $y)_P$ in bit 0.

Signatures

A signature consists of two 21-byte integers (c, d). See **IEEE P1363** for the definitions of *c* and *d*.

6 Conformance Specification

This implementation follows the proposed standard **IEEE P1363 / D9 (Draft Version 9): Standard Specifications for Public Key Cryptography**. In the terminology of this standard, the following schemes, primitives, and additional techniques are implemented:

Scheme	*Description*	*Terminal*	*Enhanced BasicCard*
ECKAS-DH1	Elliptic Curve Key Agreement Scheme, Diffie-Hellman version, where each party contributes one key pair. This scheme uses primitive **ECSVDP-DH**, with additional technique **KDF1.**	✓	✓
ECSSA	Elliptic Curve Signature Scheme with Appendix. This scheme uses primitives **ECSP-NR** (in the Terminal and the BasicCard) and **ECSV-NR** (in the Terminal only), and additional technique **EMSA1**.	✓	✓

Primitive	*Description*	*Terminal*	*Enhanced BasicCard*
ECSVDP-DH	Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version.	✓	✓
ECSP-NR	Elliptic Curve Signature Primitive, Nyberg-Rueppel version.	✓	✓
ECVP-NR	Elliptic Curve Verification Primitive, Nyberg-Rueppel version.	✓	

Additional Technique	*Description*	*Terminal*	*Enhanced BasicCard*
KDF1	Key Derivation Function. The hash function is **SHA–1: Secure Hash Algorithm, revision 1.**	✓	✓
EMSA1	Encoding Method for Signatures with Appendix. The hash function is **SHA–1: Secure Hash Algorithm, revision 1.**	✓	✓