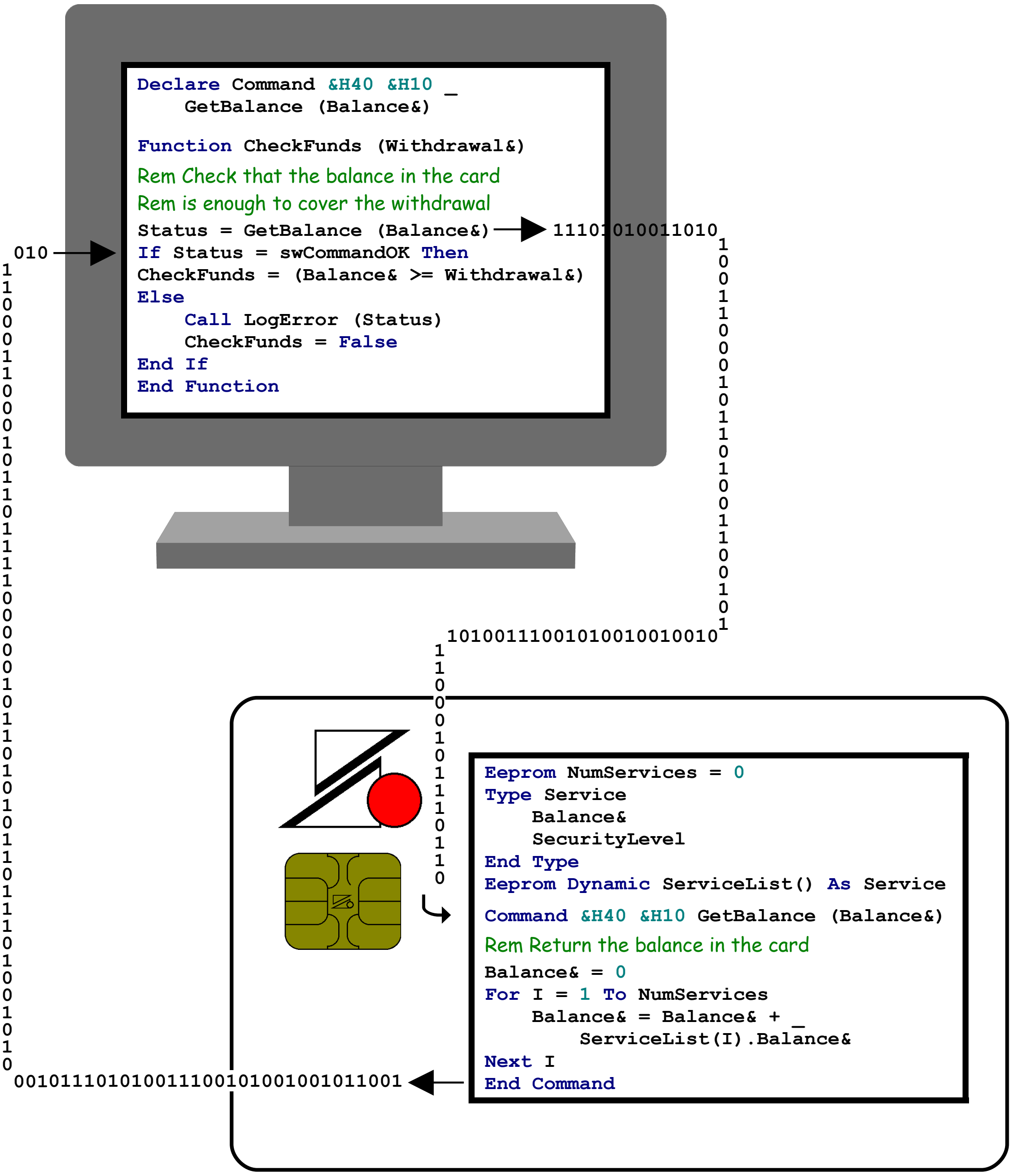
BasicCard



The ZeitControl BasicCard Family

ZeitControl
cardsystems GmbH

The ZeitControl BasicCard Family

Enhanced BasicCard
Professional BasicCard
MultiApplication BasicCard

Document version 8.15

13th July 2012

Author: Tony Guilfoyle

e-mail: development@ZeitControl.de

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

Overview

Like most computer hardware, the price of smart cards is steadily decreasing, while performance and capacity are improving all the time. You can now buy a fully-functional computer, the size of your thumb-nail, for less than a euro. And with ZeitControl's BasicCard, you can program your own smart card in an afternoon, with no previous experience required. If you can program in Basic, you can design and implement a custom smart card application.

The BasicCard family

ZeitControl's BasicCard family consists of the Enhanced BasicCard, the Professional BasicCard, and the MultiApplication BasicCard. (The Compact BasicCard is no longer supported.) A BasicCard contains 256-4800 bytes of RAM, and 2-72 kilobytes of user-programmable EEPROM. The EEPROM contains the user's Basic code, compiled into a virtual machine language known as P-Code (the Java programming language uses the same technology). The user's permanent data is also stored in EEPROM, either as Basic variables, or in the BasicCard's directory-based file system. The RAM contains run-time data and the P-Code stack.

The ZC-Basic language.

All the BasicCards are programmed in a dialect of Basic which we call ZC-Basic. This resembles many well-known Basic dialects, such as QBasic, but contains some constructs specific to the smart card environment. Most importantly, it contains no surprises for a Basic programmer.

The smallest BasicCard, the Enhanced BasicCard ZC3.12, contains 2 kilobytes of EEPROM. How much Basic code can you squeeze into this card? While no exact figure can be given, our experience suggests a ratio of about 10-20 bytes of P-Code to every statement of Basic code. Assuming on average one statement every two lines (for comments and blank lines), this works out at 200-400 lines of source code. Some BasicCards contain 36 times as much EEPROM. The latest MultiApplication BasicCard,with up to 72 kilobytes of EEPROM, allows several sizeable Applications in a single card.

To create P-Code and download it to the BasicCard, you need ZeitControl's BasicCard support software. This software is free of charge, and can be downloaded at any time from ZeitControl's BasicCard page on the Internet (www.BasicCard.com). The support software runs under Microsoft® Windows® XP or later. With this support package, you can test your software even if you don't have a card reader, by simulating the BasicCard in the PC. The package contains fully-functional ZC-Basic Multiple Debuggers, that can run multiple Terminal and BasicCard programs simultaneously. So you can try out your idea for a smart card application without it costing you a cent.

The Smart Card Environment

Obviously, programming a smart card is not the same as programming a desktop computer. It has no keyboard or screen, for a start. So how does a smart card receive its input and communicate its output? It talks to the outside world through its bi-directional I/O contact. Communication takes place at 9600 baud or more, according to the T=0 and T=1 protocols defined in ISO/IEC standards 7816-3 and 7816-4. (The latest cards also implement the contactless ISO14443 Type A protocol, and the Mifare™ protocol.) But this is invisible to the Basic programmer – all you have to do is define a command in the card, and program it like an ordinary Basic procedure. Then you can call this command from a ZC-Basic program running on the PC. Again, the command is called as if it was an ordinary procedure.

The BasicCard operating system takes care of all the communications for you. It will even encrypt and decrypt the commands and responses if you ask it to. All you have to do is specify a different two-byte ID for each command that you define. (If you are familiar with **ISO/IEC 7816-4**: *Interindustry commands for interchange*, you will know these two bytes as **CLA** and **INS**, for Class and Instruction.)

Here is a simple example. Suppose you run a discount warehouse, and you are issuing the BasicCard to members to store pre-paid credits. You will want a command that returns the number of credits left in the card. So you might define the command **GetCustomerCredits**, and give it an ID of **&H20 &H02** (**&H** is the hexadecimal prefix):

```
Eeprom CustomerCredits ' Declare a permanent Integer variable
Command &H20 &H02 GetCustomerCredits (Credits)
    Credits = CustomerCredits
End Command
```

You can call this command from the PC with the following code:

```
Const swCommandOK = &H9000
Declare Command &H20 &H02 GetCustomerCredits (Credits)
Status = GetCustomerCredits (Credits)
If Status <> swCommandOK Then GoTo CancelTransaction
```

The value &H9000 is defined in **ISO/IEC 7816-4** as the status code for a successful command. This value is automatically returned to the caller unless the ZC-Basic code specifies otherwise.

It's as simple as that. Of course, there is a lot more going on below the surface, but you don't have to know about it to write a BasicCard application.

Technical Summary

There are three BasicCard families: the Enhanced, Professional, and MultiApplication BasicCards. They contain some or all of the following:

Communication Protocols

- **T=0** byte-level communication protocol defined in **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*
- **T=1** block-level communication protocol defined in **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*
- **T=CL Type A** contactless protocol, as defined in **ISO/IEC 14443**: *Proximity Cards*
- **Mifare™** contactless protocol from NXP Semiconductors

Cryptographic Algorithms

- **RSA** public-key algorithm with keys up to 4096 bits long
- **EC-p** Prime Field Elliptic Curve public-key algorithm with field size up to 544 bits
- **EC-167** and **EC-211** Binary Elliptic Curve public-key algorithm with field size up to 211 bits
- **DES** Data Encryption Standard, with 8-, 16-, and 24-byte keys
- **AES** Advanced Encryption Standard, with 16-, 24-, and 32-byte keys
- **EAX** algorithm for Authenticated Encryption
- **OMAC** algorithm for Message Authentication
- Secure Hash Algorithms **SHA-1**, **SHA-224**, **SHA-256**, **SHA-384,** and **SHA-512**

Command Handling

- a command dispatcher built around the structures defined in **ISO/IEC 7816-4**: *Interindustry commands for interchange* (**CLA INS P1 P2** [**Lc IDATA**] [**Le**])
- support for extended **Lc**/**Le**, allowing commands and responses up to 2048 bytes long
- configurable Secure Messaging according to **ISO/IEC 7816-4**
- built-in commands for loading EEPROM, enabling encryption, etc.
- code for the automatic encryption and decryption of commands and responses, using the **AES** or **DES** symmetric-key algorithm

Further Features

- a Virtual Machine for the execution of ZeitControl's P-Code
- a directory-based, PC-like file system
- IEEE-compatible floating-point arithmetic

The data sheet on the next two pages contains details of available BasicCard versions, and the features that they support.

Development Software

The **ZeitControl MultiDebugger** software support package consists of:

- **BCDevEnv**, the BasicCard Development Environment
- **ZCMDTerm** and **ZCMDCard**, debuggers for Terminal programs and BasicCard programs
- **ZCMBasic**, the compiler for the ZC-Basic language
- **ZCMSim**, for low-level simulation of Terminal and BasicCard programs
- **BCLoad**, for downloading P-Code to the BasicCard
- **KeyGen**, a program that generates random keys for use in encryption
- **BCKeys**, for downloading cryptographic keys to the Enhanced BasicCard

BasicCard Versions

Enhanced BasicCard

Version	*PK Algorithm*	*EEPROM*	*RAM*	*Protocol*	*Encryption*	*Hash*
ZC3.12, ZC3.13	**EC-161**	**2K**	**256** bytes	**T=1**	**DES, AES**	**SHA-1**
ZC3.32, ZC3.33	**EC-161**	**8K**	**256** bytes	**T=1**	**DES, AES**	**SHA-1**
ZC3.42, ZC3.43	**EC-161**	**16K**	**256** bytes	**T=1**	**DES, AES**	**SHA-1**

Professional BasicCard[1]

Version	*PK Algorithm*	*EEPROM*	*RAM*	*Protocol* [2]	*Encryption* [3]	*Hash*
ZC5.4	Binary **EC** [4]	**16K**	**2K**	**0, 1**	**D, A, E, O**	**SHA-256**
ZC5.5	Binary **EC** [4]	**32K**	**2K**	**0, 1**	**D, A, E, O**	**SHA-256**
ZC5.6	Binary **EC** [4]	**60K**	**2K**	**0, 1**	**D, A, E, O**	**SHA-256**
ZC7.4	All **PK** [5]	**16K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]
ZC7.5	All **PK** [5]	**32K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]
ZC7.6	All **PK** [5]	**72K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]

MultiApplication BasicCard[1]

Version	*PK Algorithm*	*EEPROM*	*RAM*	*Protocol* [2]	*Encryption* [3]	*Hash*
ZC6.5	Binary **EC** [4]	**31K**	**1.7K**	**0, 1**	**D, A, E, O**	**SHA-256**
ZC8.4	All **PK** [5]	**16K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]
ZC8.5	All **PK** [5]	**32K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]
ZC8.6	All **PK** [5]	**72K**	**4.3K**	**0, 1, CL, M**	**D, A, E, O, SM**	All **SHA** [6]

[1] See **Professional and MultiApplication BasicCard Datasheet** for more information

[2] **0**: **T=0**; **1**: **T=1**; **CL**: **T=CL**; **M**: **Mifare™**

[3] **D**: **DES**; **A**: **AES**; **E**: **EAX**; **O**: **OMAC**; **SM**: **ISO** Secure Messaging

[4] **EC-167**, **EC-211**

[5] **RSA** (4096 bits), **EC-167**, **EC-211**, **EC-p** (544 bits)

[6] **SHA-1**, **SHA-224**, **SHA-256**, **SHA-384**, **SHA-512**

Algorithms and Protocols

Public-Key Algorithms

Name	*Description*	*Key size*	*Reference*
RSA	Rivest-Shamir-Adleman algorithm	Up to 4096 bits	IEEE P1363: Standard Specifications for Public Key Cryptography
EC-p	Elliptic Curve Cryptography over the field GF(p)	Up to 544 bits	
EC-211	Elliptic Curve Cryptography over the field GF(2^{211})	211 bits	
EC-167	Elliptic Curve Cryptography over the field GF(2^{167})	167 bits	
EC-161	Elliptic Curve Cryptography over the field GF(2^{168})	161 bits	

Symmetric-Key Algorithms

Name	*Description*	*Key size*	*Reference*
EAX	Encryption with Authentication for Transfer (using **AES**)	128/192/ 256 bits	EAX: A Conventional Authenticated-Encryption Mode[1] M. Bellare, P. Rogaway, D. Wagner
OMAC	One-Key CBC-MAC (using **AES**)	128/192/ 256 bits	OMAC: One-Key CBC MAC[1] Tetsu Iwata and Kaoru Kurosawa Department of Computer and Information Sciences, Ibaraki University 4–12–1 Nakanarusawa, Hitachi, Ibaraki 316-8511, Japan
AES	Advanced Encryption Standard	128/192/ 256 bits	Federal Information Processing Standard FIPS 197
DES	Data Encryption Standard	56/112/168 bits	ANSI X3.92-1981: Data Encryption Algorithm

[1] These documents are available at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/

Data Hashing Algorithms

Name	*Description*	*Reference*
SHA-224, SHA-256, SHA-384, SHA-512	Secure Hash Standard	Federal Information Processing Standard FIPS 180-2
SHA-1	Secure Hash Algorithm, revision 1	

Communication Protocols

Name	*Description*	*Reference*
T=0	Byte-level transmission protocol	ISO/IEC 7816-3: Electronic signals and transmission protocols
T=1	Block-level transmission protocol	
T=CL	Contactless Type A transmission protocol	ISO/IEC 14443-4: Transmission protocol
Mifare™	Contactless fare card protocol	NXP Semiconductors

Contents

Part I: User's Guide

Part II: Technical Reference

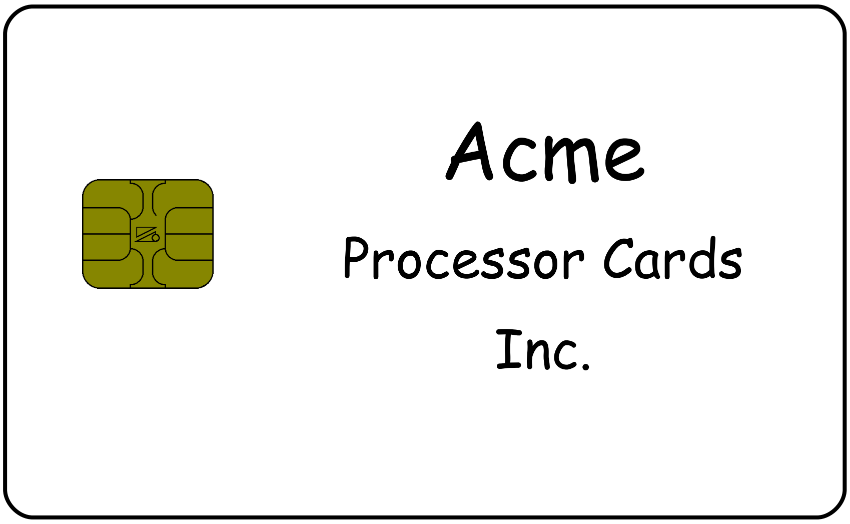
Part I

User’s Guide

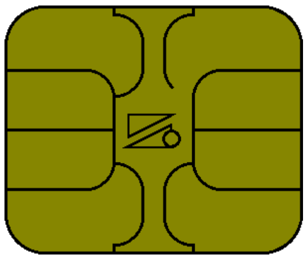
1. The BasicCard

1.1 Processor Cards

A processor card looks like this:



Most of this is just plastic. The important part is the metallic contact area:

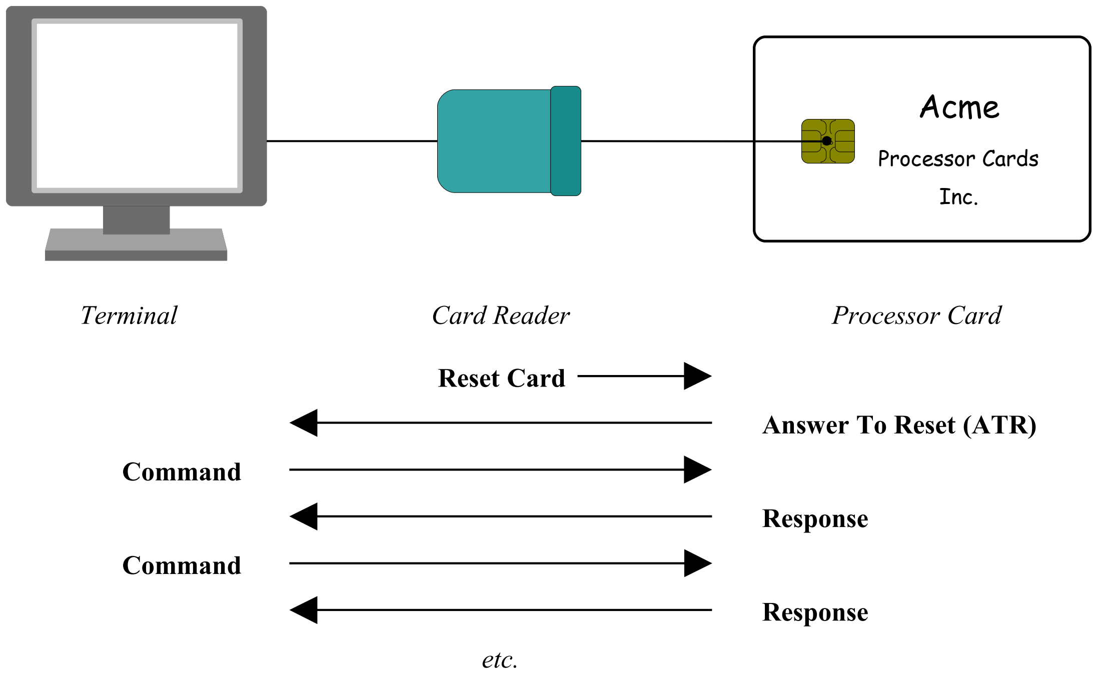


This area has the same layout as a standard telephone card. However, a telephone card contains only memory, while a processor card contains a CPU as well – in effect, a complete miniature computer. A typical processor card today might contain 32-256 kilobytes of ROM (Read-Only Memory) for the operating system machine code, 8-72 kilobytes of EEPROM (Electrically Erasable, Programmable Read-Only Memory) for the data in the card, and 256-4096 bytes of RAM (Random Access Memory). The EEPROM is the 'hard disk' of the card – data written to EEPROM retains its value when the card is powered down.

The single most important aspect of processor card design is security. That's what processor cards are for. If I want to make telephone calls for free, I can buy the equipment to make my own telephone cards – but the reward is not proportional to the effort required (not to mention the risk of detection). But if those telephone cards contained real money, instead of just telephone credits, there would be plenty of people working on making illegal copies.

So for cards that contain so-called *electronic cash* that can be spent like real money, a processor card is required. The processor protects access to the memory, using tamper-proof hardware design coupled with high-security software algorithms.

Communication with a processor card is by means of a *command-response* protocol. When a card is inserted in the reader, a command-response session is initiated:



The processor card is the passive partner in this exchange. After sending the Answer To Reset, it does nothing until it receives a command from the Terminal. Then after sending the response to this command, it waits passively for the next command, and so on. The command-response protocol used by most processor cards is defined in the **ISO** standard documents **ISO/IEC 7816-3**: *Electronic signals and transmission protocols* and **ISO/IEC 7816-4**: *Interindustry commands for interchange*. These documents are summarised in **Chapter 8: Communications**.

1.2 Programmable Processor Cards

Before the BasicCard became available, programming a processor card was a major undertaking. The following skills were involved:

- Assembly language programming. Although ‘C’ compilers were available for some processor cards, it was not possible to write the whole operating system in ‘C’.
- Byte-level communication protocols, such as the **T=0** or **T=1** protocol.
- Block-level communication protocols at the command-response level.
- Programming at the hardware level for writing to EEPROM.
- Security algorithms. You had to write your own.

You would also need a complex (and expensive) development environment. And on top of everything, after submitting your program to the chip manufacturer, you would have to wait for two or three months, while it was burned into ROM in several thousand chips, before you could test it in a real card.

However, the situation has improved. Programmable processor cards are now available. The heart of a programmable processor card is its P-Code interpreter. You write a program for the card, in a high-level language like Java or Basic. This is compiled into so-called P-Code, which is a machine-independent language that looks like machine code. The P-Code is downloaded to the card, where it is executed by the interpreter. And if your code doesn’t work first time, you can download a new version into the same card. So the development cycle is closer to what most programmers are used to.

1.3 BasicCard Features

The BasicCard is a programmable processor card, with a P-Code interpreter optimised for executing programs written in Basic. It was designed with four criteria in mind at all times. It had to be:

Inexpensive The development software is free of charge – you can download the latest version from our web site at any time at www.BasicCard.com. And most versions of the BasicCard are less than half the price of other currently-available programmable processor cards.

Easy to program Everybody can program in Basic – or if they can't, they can pick it up in an afternoon. That's all you need to program the BasicCard. A command from the Terminal to the BasicCard is defined and called just like a Basic function. The file system in the BasicCard looks just like a regular disk. Encryption has been made as simple as possible to implement – you just turn it on or off. And EEPROM data is read and written just like RAM data.

Secure State-of-the-art cryptographic algorithms are available for all BasicCard types:

Professional and MultiApplication BasicCards

- public-key cryptography: **RSA** (up to 4096 bits) or **EC** (up to 544 bits)
- **AES** Advanced Encryption Standard and **DES** Data Encryption Standard
- Secure Hash Algorithms from **SHA-1** to **SHA-512**
- Provably secure modes of operation **EAX** for Authenticated Encryption and **OMAC** for Message Authentication

Enhanced BasicCard

- **DES** Data Encryption Standard
- Plug-In Libraries: **AES**, **SHA-1**, and **EC** over GF(2^{168})

The security of the BasicCard implementation is enhanced by our cryptographic key generation program – see **6.9.4 The Key Generator KeyGen.**.

ISO-compliant In the ZC-Basic programming language, defining your own **ISO**-compliant command is as easy as declaring a function. Just as importantly, **ISO**-defined commands, such as **SELECT FILE** and **READ RECORD**, can be programmed in ZC-Basic. So you can implement your own **ISO** card, or call an existing **ISO** card from a ZC-Basic Terminal program. And the latest BasicCards support Secure Messaging according to ISO 7816-4. See **8.6 Commands and Responses** and **7.6.8 Secure Messaging Procedures** for more information.

The operating systems in all BasicCards contain the following features:

- A full implementation of the **T=1** communications protocol defined in **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*, including chaining, retries, and WTX requests. In addition, the Professional and MultiApplication BasicCards contain the **T=0** protocol; and the **ZC7-**series BasicCards contain the contactless Type A **T=CL** protocol.

  These protocols define the structure and duration of the bits and bytes that constitute the messages in a command-response session. For more information, see **8.3 The T=0 Protocol**, **8.4 The T=1 Protocol**, and **8.5 The T=CL Contactless Protocol**.

- Pre-defined commands for downloading programs and data to the BasicCard, enabling automatic encryption, etc.

  These commands are described in **8.9 Pre-Defined Commands**.

- A Virtual Machine for the execution of ZeitControl's P-Code.

  The compiler **ZCMBasic** compiles ZC-Basic source code into P-Code, an intermediate language that can be thought of as the machine code for a Virtual Machine. (The Java programming language uses the same technology, although the P-Code instruction set is not the same.) The P-Code is downloaded to the card using the **BCLoad** Card Loader program, or the **ZCMDCard** debugger. Then the Virtual Machine in the BasicCard executes the P-Code instructions at run-time.

The latest BasicCards (**ZC7-** and **ZC8-**series from **REV D**) can be configured to double as **Mifare™** cards. The card acts as a Mifare™ card or a BasicCard, according to the type of card reader; and when the card is acting as a BasicCard, the contents of the Mifare™ data blocks can be read and written from within the ZC-Basic program. For more information, see **7.13 The Mifare™ Library**.

1.4 BasicCard Programs

1.4.1 Applications

BasicCard programs are written in ZC-Basic, which is a procedure-oriented language similar to QBasic, but with special features for the processor card environment. It is described in **Chapter Error: Reference source not found: Error: Reference source not found**.

A BasicCard program is specified in a single source file (which may, however, include other source files). This file will typically have a **.BAS** extension. It consists of a set of Commands, with associated files and data.

Single-application BasicCards (Enhanced and Professional) can contain only a single Application; all Commands in the Application's Command set have Read and Write access to all the associated files and data.

A MultiApplication BasicCard can contain up to 128 different Applications, each with its own Command set and associated data. Associated data is accessible only by its own Application. Files, however, can be accessed for Reading or Writing by any Application that has the necessary permission.

1.4.2 Image Files

The compiler can create a ZeitControl Image File (with **.IMG** extension) from your BasicCard program source file. This image file can then be downloaded to a BasicCard; or it can be run in the **ZCMSim** P-Code interpreter together with a Terminal Program – see **6.9.2 The P-Code Interpreter ZCM** for details.

1.4.3 Debug Files

If the BasicCard Application is to be run in the **ZCMDCard** BasicCard debugger, the compiler must create a ZeitControl Debug File (with **.DBG** extension). This is a ZeitControl Image File with symbolic debugging information included. Image files and debug files are described in **Chapter 11: Output File Formats**.

1.4.4 Card Program Files

The **ZCMDCard** BasicCard debugger works with simulated BasicCards. A simulated card is described by a Card Program File, with extension **.ZCC**. This file contains the simulated EEPROM, which retains its contents between program runs, and various other data, such as source filename of each Application, the BasicCard version, and compiler options. A single source file may be the basis for several Card Program files, each running the same program, but with different data stored in simulated EEPROM.

1.5 BasicCard Program Layout

A BasicCard program consists of *initialisation code*, *procedure definitions*, and *file definition sections*.

1.5.1 Initialisation Code

The first block of code that is not contained inside a procedure definition is *initialisation code*. In a single-application BasicCard, this initialisation code gets executed when the first user-defined command is called from the Terminal. In the MultiApplication BasicCard, an Application's initialisation code is executed whenever the Application is selected.

Initialisation code is not required, but it can be useful for certain things; for instance, checking that the card has not been cancelled by the issuer, or that the expected files and directories are present.

1.5.2 Procedure Definitions

ZC-Basic has three types of procedure: subroutines, functions, and commands. Each procedure is self-contained – nested procedure definitions are not allowed, and **GoTo** and **GoSub** statements can only transfer control to labels within the current procedure. Subroutines and functions are familiar to Basic programmers – a subroutine is a block of code that can be called from other procedures, and a function is a subroutine that returns a value. The command, however, is special to ZC-Basic; it is the mechanism by which the Terminal program communicates with the BasicCard program.

According to the **ISO** standard document **ISO/IEC 7816-4**: *Interindustry commands for interchange*, each command is assigned a unique two-byte ID. This is all the ZC-Basic programmer needs to know about ISO standards. For the curious, these two bytes are known as **CLA** and **INS** (for Class and Instruction); the full command-response protocol defined in the standard is described in **8.6 Commands and Responses**. The two-byte ID must be supplied between the **Command** keyword and the name of the command. Here is an example (**&H** is the hexadecimal prefix):

```
Command &H80 &H10 GetCustomerName (Name$)
    Name$ = CustomerName$
End Command
```

Then whenever the BasicCard receives a command from the Terminal with **CLA = &H80** and **INS = &H10**, the card's operating system automatically executes the **GetCustomerName** command.

A command behaves like a cross between a function and a subroutine: it is defined like a subroutine (as above), but called like a function (see **2.2 Terminal Program Layout**). The BasicCard operating system fills in the return value that gets passed back to the Terminal program. This return value consists of the two status bytes **SW1** and **SW2** defined in **ISO/IEC 7816-4**. The return value of a command should always be checked; for instance, the card may have been removed from the reader, or the reader may have lost power for some reason. If **SW1 = &H90** and **SW2 = &H00**, or if **SW1 = &H61**, then the command completed successfully. Otherwise a problem has occurred that prevented successful execution of the command.

These two status bytes are available as pre-defined variables in the BasicCard, so you can define your own error codes. The two-byte **Integer** variable **SW1SW2** is also defined. For instance:

```
Eeprom Balance As Long : Rem Declare permanent (Eeprom) variable
Const InsufficientCredit = &H6F00
Command &H80 &H20 DebitAccount (Amount As Long)
    If Balance < Amount Then
        SW1SW2 = InsufficientCredit
    Else
        Balance = Balance - Amount
    End If
End Command
```

Notes:

- You don't need to specify **SW1** and **SW2** if the command completes successfully. They are set to **&H90** and **&H00** before the command is called.
- If you specify values for **SW1** and **SW2** other than the two indicators of successful completion (**SW1SW2 = &H9000** or **SW1 = &H61**), the operating system throws away the response data and just returns the two status bytes to the Terminal program. (This is in accordance with **ISO/IEC 7816-4**.) In the Professional and MultiApplication BasicCards, you can override this behaviour – see **3.3.13 The #Pragma Directive** and **7.15.5 Communications** for details.
- Your own **SW1-SW2** error codes can take any values. However, for **ISO** compliance, or if you are programming a Professional BasicCard that uses the **T=0** protocol, the high nibble of **SW1** must be **6**, i.e. **SW1 = &H6***X*. You should also avoid assigning new meanings to ZC-Basic's own error codes. ZC-Basic's error codes are listed in **8.8 Status Bytes SW1 and SW2**; you can avoid any clashes if you use **SW1 = &H6B** or **&H6F** (except **SW1-SW2=&H6F00**).

1.5.3 File Definition Sections

All BasicCards contain a standard directory-based file system, with directories organised in a tree structure. There are several ways to access BasicCard files and directories.

- From within the BasicCard itself, files can be created, read, and written with exactly the same statements that you would use in a Basic program running as a Console application under Windows®. There are also some special statements for setting access conditions on files and directories, to restrict access from Terminal programs and from other Applications. These access conditions can depend on cryptographic keys, user passwords, etc.
- From a Terminal program, the BasicCard looks just like a disk drive, with the special drive name "`@:`". If the access conditions permit it, you can create, read, and write files and directories in the BasicCard as if it was a disk.
- You can initialise directory structures and files in a BasicCard program with File Definition Sections – see **4.11 File Definition Sections**. In a MultiApplication BasicCard program, a File Definition Section can also contain Component definitions and Application Loader commands. See **5.5 Application Loader Definition Section** for more information.

1.5.4 Permanent Data

Most BasicCard applications will contain permanent data, that retains its value while the BasicCard is powered down. Permanent data is stored in EEPROM (Electrically Erasable, Programmable Read-Only Memory). In most BasicCards, you can store permanent data in files; but it is often simpler to store permanent data in **Eeprom** variables, particularly if the length of the data is fixed. An example of an **Eeprom** variable was given in the previous section:

```
Eeprom Balance As Long : Rem Declare permanent (Eeprom) variable
```

The variable **Balance** declared here can be read or written just like a regular variable. **Eeprom** strings and arrays can also be declared. This can be a very convenient way of storing permanent data, in all types of BasicCard. Note, however, that in the MultiApplication BasicCard, **Eeprom** data can only be accessed by the Application that declares it; data to be shared between Applications must be file-based.

Writing to EEPROM can take a few milliseconds, so the possibility is always present that the card will lose power in the middle of the write operation. So all EEPROM write operations are automatically logged, to enable them to be completed in the event of power loss. In **ZC7**- and **ZC8-**series BasicCards from **REV C**, a Transaction Manager is available, to let the programmer execute a sequence of EEPROM writes as a single indivisible unit – see **7.5 The TMLib Transaction Manager Library**.

1.6 The Compact BasicCard

The Compact BasicCard was ZeitControl's first BasicCard. It is no longer available, and is not described further in this document.

1.7 The Enhanced BasicCard

The original Enhanced BasicCard – the **ZC2**-series Enhanced BasicCard – is no longer supported. The current Enhanced BasicCard is the **ZC3**-series Enhanced BasicCard:

BasicCard ZC3.12	Contains 2K of user-programmable EEPROM. Available since November 2008.
BasicCard ZC3.2	Contains 4K of user-programmable EEPROM. Available in large quantities only – contact ZeitControl for details.
BasicCard ZC3.32	Contains 8K of user-programmable EEPROM. Available since November 2008.
BasicCard ZC3.42	Contains 16K of user-programmable EEPROM. Available since November 2008.

1.8 The Professional BasicCard

All Professional BasicCards contain built-in public-key cryptography algorithms:

- **ZC5-**series cards support the **EC-167** and **EC-211** algorithms (Elliptic Curve cryptography over the finite fields $GF(2^{167})$ and $GF(2^{211})$);
- **ZC7**-series cards support the **EC-167** and **EC-211** algorithms, as above, and **RSA** and **EC-p** (Elliptic Curve cryptography over the finite field GF(p), where p is a prime number up to 544 bits long).

ZC7-series cards also implement configurable Secure Messaging according to **ISO 7816-4**; and they contain a programmable Transaction Manager for writing multiple EEPROM data items as an indivisible unit.

Currently available Professional BasicCards:

Version	*User EEPROM*	T=0	T=1	T=CL	EAX	OMAC	AES	DES	RSA	EC	SHA
ZC5.4	16K	✓	✓		✓	✓	✓	✓		**EC-211**	**SHA-256**
ZC5.5	32K	✓	✓		✓	✓	✓	✓		**EC-211**	**SHA-256**
ZC5.6	60.5K[1]	✓	✓		✓	✓	✓	✓		**EC-211**	**SHA-256**
ZC7.4	16K	✓	✓	✓	✓	✓	✓	✓	**4096**	**EC-p**	**SHA-512**
ZC7.5	32K	✓	✓	✓	✓	✓	✓	✓	**4096**	**All**[2]	**SHA-512**
ZC7.6	72K	✓	✓	✓	✓	✓	✓	✓	**4096**	**All**[2]	**SHA-512**

[1] The **ZC5.6** BasicCard contains 48K EEPROM, and 12.5K FLASH memory. The FLASH memory is available for the user program, and for data declared **ReadOnly**; it cannot be used for general-purpose EEPROM data.

[2] **EC-p**, **EC-211**, and **EC-167**

From time to time, new versions of the Professional BasicCard will appear, and new features will be added to existing cards. See the **Professional and MultiApplication BasicCard Datasheet** on ZeitControl's BasicCard web site www.BasicCard.com for the most up-to-date information.

The version number of the card, along with its software revision number, is returned by the card as an ASCII string in the response to the **GET STATE** command (see **8.9.3 The GET STATE Command**).

1.9 The MultiApplication BasicCard

Four MultiApplication BasicCards are currently available:

Version	*User EEPROM*	T=0	T=1	T=CL	EAX	OMAC	AES	DES	RSA	EC	SHA
ZC6.5	31K	✓	✓		✓	✓	✓	✓		**EC-211**	**SHA-256**
ZC8.4	16K	✓	✓	✓	✓	✓	✓	✓	**4096**	**All**[2]	**SHA-512**
ZC8.5	32K	✓	✓	✓	✓	✓	✓	✓	**4096**	**All**[2]	**SHA-512**
ZC8.6	72K	✓	✓	✓	✓	✓	✓	✓	**4096**	**All**[2]	**SHA-512**

[2] **EC-p**, **EC-211**, and **EC-167**

See **Chapter 5: The MultiApplication BasicCard** for more information.

2. The Terminal

2.1 The Terminal Program

The ZC-Basic language was designed with the BasicCard in mind. But it can also run in a PC, with or without a card reader attached to the serial port. You can write a stand-alone ZC-Basic program to do your monthly accounts, or to help you solve crosswords, or whatever you like.

A ZC-Basic program that runs on a PC is referred to in this documentation as the **Terminal** program. Usually it will communicate with one or more ZC-Basic programs running in (real or simulated) BasicCards – the **BasicCard** programs.

The compiler can create executable files, image files, and debug files from a Terminal program source file – see **6.9.1 The ZC-Basic Compiler ZCMBasic.** for details.

2.1.1 Executable Files

The compiler can create standard executable files (files with **.exe** extension), that will run as Console applications under Windows®. Such programs can communicate with a real or simulated BasicCard. Such programs are not self-modifying, so they can't execute **Write Eeprom** statements (see **2.2.4 Permanent Data** below).

Command-line parameters passed to the executable file can be accessed from ZC-Basic in the pre-defined string array **Param$ (1 To nParams)** – see **3.22.9 Pre-Defined Variables**.

2.1.2 Image Files

For more flexibility during program development, the compiler can also create a ZeitControl Image File (with **.IMG** extension) from your Terminal program source file. The **ZCMSim** P-Code interpreter can then run this Terminal program together with a BasicCard program running in a real or simulated BasicCard – see **6.9.2 The P-Code Interpreter ZCMSim.exe** for details.

2.1.3 Debug Files

The compiler can also produce Debug Files (with **.DBG** extension), which are ZeitControl Image Files with debugging information included. These files are used by the **ZCMDTerm** Terminal Program debugger. Image files and debug files are described in **Chapter 11: Output File Formats**.

2.1.4 Terminal Program Files

The **ZCMDTerm** Terminal Program debugger saves the data for a given Terminal Program in a Terminal Program file, with **.ZCT** extension. This file contains the source filename, the compiler options, and various other data.

2.2 Terminal Program Layout

A Terminal program consists of the *main procedure* and *procedure definitions*. BasicCard commands are declared in *command declarations*, after which they can be called just like functions.

The Terminal program is executed by ZeitControl's P-Code interpreter, in one of three ways:

- as a stand-alone executable file (**.exe**) created by the compiler;
- by the **ZCMSim** P-Code interpreter, from an Image File (**.IMG**);
- by the **ZCMDTerm** Terminal Program debugger, from a Debug File (**.DBG**).

2. The Terminal

The P-Code interpreter can run BasicCard programs simultaneously in the PC in simulated BasicCards, or it can communicate with genuine BasicCards via a card reader – a ZeitControl Chip-X® or CyberMouse® card reader connected to a serial port or a USB port, or any other PC/SC-compatible card reader.

2.2.1 The Main Procedure

The *main procedure* starts at the first statement that is not contained inside a procedure definition, and ends at the start of the next procedure definition (or the end of the source file). The Terminal program begins execution at the first statement in the main procedure, and continues until it reaches the end of the main procedure, or until an **Exit** statement is executed.

2.2.2 Procedure Definitions

Procedure definitions in the Terminal program consist of functions and subroutines, exactly like a regular Basic program. Each procedure is self-contained – nested procedure definitions are not allowed, and **GoTo** and **GoSub** statements can only transfer control to labels within the current procedure.

2.2.3 Command Declarations

Before you can call a BasicCard command, you must declare it, so that the ZC-Basic compiler knows the two ID bytes of the command, and the types of the command parameters. Apart from the two ID bytes, a command declaration looks like a subroutine declaration. Here are declarations of the three example commands from **1.5 BasicCard Program Layout**:

```
Declare Command &H80 &H10 GetCustomerName (Name$)
Declare Command &H80 &H20 DebitAccount (Amount As Long)
Declare Command &H80 &H30 ChangeBalance (NewBalance As Long)
```

Calling these commands is just like calling a function:

```
Status = GetCustomerName (Name$)
If Status <> &H9000 And (Status And &HFF00) <> &H6100 Then
    Print "GetCustomerName: Status = &H"; Hex$ (Status)
    GoTo Retry
End If
```

You should always check the return value, even if the command itself has no error conditions, in case a communication problem has occurred (such as the card being removed from the reader). If you prefer, you can use the pre-defined variables **SW1**, **SW2**, and **SW1SW2**, which contain the status bytes from the most recently called command:

```
Call GetCustomerName (Name$)
If SW1SW2 <> &H9000 And SW1 <> &H61 Then
    Print "GetCustomerName: Status = &H"; Hex$ (SW1SW2)
    GoTo Retry
End If
```

See **8.8 Status Bytes SW1 and SW2** for a list of ZC-Basic status codes. The file `BasicCardV8\Inc\Commands.Def` defines these status codes in **Const** statements, so you can refer to &H9000 and &H61 as **swCommandOK** and **sw1LeWarning** respectively if you include this file in your program – see **3.3.1 Source File Inclusion**. Alternatively, you can call the subroutine **CheckSW1SW2()**, which is defined in the file CommErr.def. If a communication error has occurred, this subroutine prints a suitable error message and exits.

2.2.4 Permanent Data

ZC-Basic contains a very convenient mechanism for the reading and writing of permanent data in the BasicCard: you just declare data of storage type **Eeprom**, and the BasicCard operating system does the rest. Although the Terminal program contains no genuine EEPROM data, this useful feature is available in Terminal programs as well, if they were loaded from a ZeitControl Image File (or Debug File). **Eeprom** data in a Terminal program is written back to the image file in two circumstances:

1. On program exit, if the appropriate options were specified:
   - in the **ZCMDTerm** Terminal Program debugger, checking the **Save EEPROM** entry in the **Terminal Settings** dialog box;
   - with the **–W** parameter on the **ZCMSim** command line (see **6.9.2 The P-Code Interpreter ZCMSim.exe**).
2. When the Terminal program executes a **Write Eeprom** statement (see **3.22.7 Saving Eeprom Data**).

Note: The **Write Eeprom** statement is only valid if the Terminal program is running in the **ZCMSim** P-Code interpreter or the **ZCMDTerm** Terminal Program debugger. Programs containing **Write Eeprom** statements can't be compiled into executable files.

3. The ZC-Basic Language

The ZC-Basic programming language is a fully functional, modern Basic, with function and subroutine calls, user-defined data types, file I/O, and pre-processor directives. In addition, it has some special features for the smart card environment, including command definition and invocation, I/O encryption, and file access control.

In this chapter, the following conventions are observed:

- ZC-Basic keywords are printed in **bold text**.
- Statement fields that must be supplied by the programmer are printed in *italic text*.
- Programming examples are printed in `fixed-width bold text`.
- Optional statement fields are enclosed in [square brackets].
- Alternatives are separated by a vertical bar and enclosed in braces, e.g. { **ByVal | ByRef** }.

File I/O in ZC-Basic is described in **Chapter 4: Files and Directories**.

3.1 The Source File

A ZC-Basic Application must consist of a single compilation unit – there is no linking stage. This lets the compiler work out the storage requirements of the whole program, so that it can use the limited RAM as efficiently as possible. You may, however, split your source into several files and **#Include** them all in a master source file.

The source consists of *lines*, which may be logically extended with the line continuation character '**_**' (underscore). Each line consists of *statements*, separated from each other with '**:**' (colon). A comment character '**'**' (single quote) causes the rest of the line to be ignored (unless it occurs inside a string). The **Rem** keyword may also be used to introduce a comment, but it is only allowed at the beginning of a statement. For instance:

```
X = 0           '     Comment introduced by comment character
                Rem   OK to use Rem on its own line...
Y = 0 : Z = 0 : Rem   ...but here we need the colon
```

3.2 Tokens

At the lowest level, a source program consists of a sequence of *tokens*. There are four kinds of token: constants, identifiers, reserved words, and special symbols. Except for string constants, tokens may not contain spaces or tabs.

A constant can be an integer, a floating-point number, or a string. Integer constants are decimal by default; the prefixes **&O** (or just **&**), **&H**, and **&B** denote octal,hexadecimal, and binaryconstants respectively. Integer constants have the range –9223372036854775808 to +9223372036854775807 (or -2^{63} to $2^{63}-1$).

If a constant contains a decimal point or an exponent (E or e), it is a floating-point constant. ZC-Basic supports single- and double-precision floating-point numbers. Floating-point numbers are stored in IEEE denormalised format:

- single-precision numbers have an 8-bit exponent and a 23-bit mantissa, which gives a precision of 7 decimal places, and a range of 1.401298E–45 to 3.402823E+38;
- double-precision numbers have an 11-bit exponent and a 52-bit mantissa, which gives a precision of 16 decimal places, and a range of 4.940656E-324 to 1.797693E+308.

Another way to specify a floating-point constant is as a bit representation: an octal, hexadecimal, or binary integer constant, followed by the special character '**!**' (for a single-precision constant) or '**#**' (for a double-precision constant). For instance, **&HBFF0000000000000#** represents the double-precision value –1.0.

A string constant is any sequence of printable characters enclosed in double quotes '"'. To include non-printable characters in a string constant, use **Chr$()**; the double quote itself is **Chr$(34)**. For example:

```
X$ = Chr$(34) + "STRING" + Chr$(34) + Chr$(10) ' 10 = new line
```

The special syntax **Chr$**($c_1$, c_2, ..., c_n), where c_i are all constants between 0 and 255, is an abbreviation for

Chr$**($c_1$) + **Chr$(c_2) + ... + **Chr$**($c_n$)

This defines a constant string consisting of the characters c_1 through c_n.

Variables, procedures, etc. must be given names, or *identifiers*. In ZC-Basic, an identifier consists of letters (**A-Z**, **a-z**) and digits (**0-9**), followed by an optional type character (**@**, **%**, **&**, **^**, **!**, **#**, **$**). It may be any length. An identifier must start with a letter. The type character specifies the data type of a function or variable, as follows:

Character:	`@`	`%`	`&`	`^`	`!`	`#`	`$`
Data type:	**Byte**	**Integer**	**Long**	**Long64**	**Single**	**Double**	**String**

If a type character is not present, the default type is **Integer** (but you can change this default behaviour with **DefByte**, **DefLng** etc – see **3.23.2 DefType Statement**). Case is not significant in ZC-Basic, so **ABC**, **AbC**, and **abc** are considered identical. An identifier must not clash with a *reserved word*, which is a word with a pre-defined meaning.

Here is a list of the reserved words in ZC-Basic:

```
Abs          Access       And          Append       ApplicationID
As           Asc          At           ATR          ATS
Base         Binary       ByRef        Byte         ByVal
Call         Case         ChDir        ChDrive      Chr$
Close        Cls          Command      Const        CurDir
CurDrive     Declare      DefByte      DefDbl       DefInt
DefLng       DefLng64     DefSng       DefString    Dim
Dir          Disable      Do           Dynamic      Eeprom
Else         ElseIf       Enable       Encryption   End
EOF          Erase        Exit         Explicit     File
For          FreeFile     Function     Get          GetAttr
GoSub        GoTo         Hex$         If           Implicit
Input        Integer      Is           Key          Kill
LBound       LCase$       Left$        Len          Let
Line         Lock         Log          Long         Loop
LTrim$       Mid$         MkDir        Mod          Name
Next         Not          On           Open         Option
Or           Output       OverflowCheck Print       Private
Public       Put          Random       Randomize    Read
ReadOnly     ReDim        Rem          Return       Right$
RmDir        Rnd          Rol          Rol@         Ror
Ror@         RTrim$       Seek         Select       SetAttr
Shared       Shl          Shr          ShrL         Single
Space$       Spc          Sqrt         Static       Step
Str$         String       String$      Sub          Tab
Then         To           Trim$        Type         UBound
UCase$       Unlock       Until        Val!         Val&
ValH         Wend         While        Write        Xor
```

The following System procedures are also reserved:

```
CardInReader CardReader   Certificate  CloseCardReader
DES          Encryption   InKey$       PcscCount    PcscReader
ResetCard    Time$        WTX
```

In addition to constants, identifiers, and reserved words, the following special symbols are recognised:

(	Left parenthesis	)	Right parenthesis	_	Underscore (line continuation)
+	Plus	-	Minus	'	Single quote (comment character)
*	Multiply	/	Divide	#	Pre-processor directive or file number
,	Comma	:	Colon	"	Double quote (string delimiter)
=	Equals	<>	Not equals	.	Full stop or Period
<	Less than	>	Greater than	;	Semi-colon
<=	Less than or equal to	>=	Greater than or equal to		

3.3 Pre-Processor Directives

Pre-processor directives are instructions to the **ZCMBasic** compiler. For instance, they tell the compiler which lines of source code to compile, and whether these lines should be written to the list file if a listing is requested. They can also be used to specify various command-line parameters in the source code itself – in this case, the compiler accepts the first occurrence of the parameter, so directives in the source code are overridden by parameters on the command line. For instance, the directive

#Stack 32

in the source code is overridden by the ZCMBasic command-line parameter **-S40**.

A pre-processor directive begins with the hash character '**#**', which must be the first character on the input line (excluding spaces and tabs).

3.3.1 Source File Inclusion

The directive

#Include *filename*

causes the named file to be included and compiled as if it was part of the source file itself. Included files can themselves contain **#Include** directives, nested to any depth. If *filename* contains any space characters, it must be enclosed in double quotes ("*filename*"); otherwise the quotes are optional. The compiler looks for the file in the following directories:

- first, the directory of the including file;
- next, directories specified in **–I** parameters, in the order that they appear in the command line (see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**);
- next, the current directory;
- next, directories specified in the Windows® Registry variable "**HKEY_CURRENT_USER\Software\ZeitControl\BasicCardV8\ZCINC"**;
- finally, directories specified in the **ZCINC** environment variable.

The **ZCINC** Windows® Registry variable can be set from the **BCDevEnv** Development Environment, via menu item **Options | Environment**.

3.3.2 Constant Definition

The statement

Const *constantname=expression* [,*constantname=expression*,...]

defines one or more constants. *expression* can be an integer, floating point, or string constant.

3.3.3 Library Inclusion

The directive

> **#Library** *filename*

loads a ZeitControl Plug-In Library for the Enhanced BasicCard. See **Chapter 7: System Libraries** for a list of currently available libraries. The compiler looks for the **#Library** file in the same directories as it looks for **#Include** files – see **3.3.1 Source File Inclusion** for details.

Notes:

- ZeitControl provides a definition file *library***.def** for each library file *library***.lib**. The definition file contains the appropriate **#Library** directive, along with all the required declarations. You should normally just **#Include** this definition file, rather than loading the library yourself with a **#Library** directive.
- Terminal programs, and Professional and MultiApplication BasicCard programs don't need the **#Library** directive, as they use a different mechanism for loading Libraries – see **3.14.2 System Library Procedures**.

3.3.4 Conditional Compilation

Sections of code can be included or excluded according to the values of constants defined earlier (or on the compiler command line):

> **#If** *condition1*
>   *code block 1*
> [**#ElseIf** *condition2*
>   *code block 2*]
> [**#ElseIf** *condition3*
>   *code block 3*]
> **. . .**
> [**#Else**
>   *code block n*]
> **#EndIf**

where *condition1, condition2,...* are constant numerical expressions, which may include symbols defined in **Const** statements or on the compiler command line (with the "–**D***symbol*" parameter – see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**). *Code block i* is compiled if *condition i* is the first non-zero condition.

Instead of testing the value of a numerical expression, you can test whether a constant symbol has been defined:

> **#IfDef** *symbol1*
>   *code block 1*
> [**#ElseIfDef** *symbol2*
>   *code block 2*]
> [**#ElseIfDef** *symbol3*
>   *code block 3*]
> **. . .**
> [**#Else**
>   *code block n*]
> **#EndIf**

The directives **#IfNotDef** and **#ElseIfNotDef** have the opposite sense to directives **#IfDef** and **#ElseIfDef** respectively.

#EndIf has the alternative form **#End If** (with a space) for compatibility with the Basic **End If** statement.

See also **3.3.12 Pre-Defined Constants**.

3.3.5 Listing Directives

You can cause sections of code (or complete included files) to be omitted from the listing file with the directive

#NoList

The **#NoList** directive is cancelled by **#List**.

3.3.6 Card State

By default, a single-application BasicCard is switched to state **TEST** after a ZC-Basic program is downloaded. You can override this with the **#State** directive:

#State { **LOAD | PERS | TEST | RUN** }

This is equivalent to the command-line parameter **–S***state* (see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**).

3.3.7 Number of Open File Slots

Each open file in a ZC-Basic program is assigned an *open file slot*. The maximum number of files that can be opened simultaneously is equal to the number of open file slots:

Terminal Program	*MultiApplication BasicCard*	*Professional BasicCard*	*Enhanced BasicCard*
32	**10**	**4**	**2**

In the Professional and Enhanced BasicCards, this number can be overridden with the **#Files** directive:

#Files *nFiles*

with $0 \le nFiles \le 16$. This number includes files opened in the BasicCard program *and* BasicCard files opened from a Terminal program. If *nFiles* is non-zero, the amount of RAM used by the file system is (6 * *nFiles* + 7) bytes.

3.3.8 Stack Size

The **#Stack** directive specifies the size of the P-Code stack:

#Stack *stack-size*

This is equivalent to the compiler command-line parameter **–S***stack-size* (see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**). If no stack size is specified, the compiler works out for itself how big the stack should be.

3.3.9 Heap Size

In a MultiApplication BasicCard program, the **#Heap** directive specifies the size of the Application heap:

#Heap *heap-size*

This is equivalent to the compiler command-line parameter **–H***heap-size* (see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**).

The Application heap contains the Application's **Eeprom** strings and **Eeprom** dynamic arrays. If no heap size is specified, the heap is made just big enough to contain the strings and arrays that are initialised in the source code. If the source code contains uninitialised **Eeprom** strings or dynamic arrays, but no **#Heap** directive is present, the compiler issues an appropriate warning.

3.3.10 Message Directive

You can output a message at any point during compilation with

#Message *message*

The message is printed to the screen, and compilation continues unaffected.

3.3.11 Error Directive

You can define your own compiler error messages with the **#Error** directive. For instance:

```
#If MaxLineLength > 80
  #Error MaxLineLength too big (max 80)
#EndIf
```

Then if anybody tries to compile the program with **MaxLineLength** defined as 100, say, the compiler will issue the error message "**#Error MaxLineLength too big (max 80)**" and stop compilation.

3.3.12 Pre-Defined Constants

According to the target machine type, one of the following constants is pre-defined by the compiler (and has the value 1):

TerminalProgram	**EnhancedBasicCard**
ProfessionalBasicCard	**MultiAppBasicCard**

For instance:

```
#IfNotDef EnhancedBasicCard
  #Error This program must be compiled for the Enhanced BasicCard!
#EndIf
```

In BasicCard programs, constants **CardOSName**, **CardMajorVersion**, and **CardMinorVersion** are also defined. For instance, in a program compiled for the Professional BasicCard ZC7.4 Rev C, they take the values "ZC7.4 REV C", 7, and 4 respectively.

3.3.13 The #Pragma Directive

Various card-specific or terminal-specific options can be selected using the **#Pragma** directive. At the time of writing, the following options are available in some or all environments:

Screen Size

#Pragma ScreenWidth=*width***, ScreenHeight=***height*

Sets the size of a Terminal program's Console window. You don't have to specify both **ScreenWidth** and **ScreenHeight** ; one of them may be absent.

Protocol Specification

#Pragma ATR (*ATR-Spec***)**

where *ATR-Spec* defines the **ATR** (Answer To Reset) that the card sends on reset. See **3.21.1 Customised ATR** for the format of *ATR-Spec*.

In the **ZC6**-series MultiApplication BasicCard, protocol selection is implemented via the reserved file "**ATR**" **–** see **5.4.1 ATR File** for details.

In the **ZC7**- and **ZC8**-series BasicCards, you can specify the ATS (Answer To Selection) that the card returns when contactless protocol is activated:

#Pragma ATS (*ATS-Spec***)**

See **3.21.2 Customised ATS** for the format of *ATS-Spec*.

SW1-SW2 = &H9XXX Allowed

#Pragma Allow9XXX

Normally, if **SW1-SW2 <> &H9000**, and **SW1 <> &H61**, then **ODATA** is not sent – see **8.6 Commands and Responses**. You can override this behaviour in some BasicCards with this option: if **SW1-SW2** has the form **&H9***XXX*, then **ODATA** is sent in the response. This behaviour is enabled for every command. See **7.15.5 Communications** for an alternative method.

At the time of writing, this option is available in Professional BasicCards **ZC5.4** (from **REV B**), **ZC5.5** and **ZC5.6** (all revisions), all **ZC7**-series BasicCards, and MultiApplication BasicCard **ZC6.5**.

Catch Undefined Commands

In the MultiApplication BasicCard, if a Default Application is defined, it can be configured to catch all commands that the currently selected Application doesn't recognise. Enable this option with

#Pragma CatchUndefinedCommands

in the source code of the Default Application. See **5.2.3 Catching Undefined Commands** for more information.

Erasable CodeBlocks

#Pragma CodeBlock "*name***"**

Procedures in an erasable CodeBlock can be deleted after use to free up EEPROM – see **3.14.3 Erasable CodeBlocks**.

Contactless UID

In the **ZC7**- and **ZC8**-series BasicCards, you can specify the properties of the UID (Unique Identifier) that the card responds with during the contactless Card Selection protocol:

#Pragma UID (*param* [, *param*] **)**

where *param* is either **Random** (for a random UID, different every time); or one of **Single**, **Double**, or **Triple** (for a 4-, 7-, and 10-byte UID). The card contains a unique 7-byte UID, and the default value is **UID (Double)**. If **Triple** is specified, then **Random** is automatically assumed (because the card's UID is only 7 bytes); if **Random** is specified alone, then **Single** is assumed (according to the ISO standard).

Processor Speed

In Enhanced BasicCards from **REV C**, and all **ZC7**- and **ZC8**-series BasicCards, you can specify the initial processor speed of the card:

- The Enhanced BasicCard directive takes a positive integer parameter, which is converted to the nearest supported processor speed in megahertz:

  #Pragma Clock (*MHz***)**

  See **7.15.10 Power Management** for a list of supported processor speeds.
- The **ZC7**- and **ZC8**-series BasicCard directive has two formats, for contact and contactless protocols:

  #Pragma Clock ([*C*],[*R*],[*D*]**)** for contact protocols
  #Pragma RFClock ([*C*],[*R*],[*D*]**)** for contactless (Radio-Frequency) protocol

  C, *R*, and *D* are the speeds of the CPU, the RSA/EC co-processor, and the DES/AES co-processor, in MHz. See **7.15.10 Power Management** for a list of allowed values.

Extended Lc/Le

The **ZC7**- and **ZC8**-series series BasicCards support extended $\mathbf{L_c}$/$\mathbf{L_e}$ protocol — commands and responses can be up to 2048 bytes in length. You can control the use of extended $\mathbf{L_c}$/$\mathbf{L_e}$ in a Terminal program at three levels:

1. with **#Pragma** *CommandLength,* to set the default for the remainder of the program;
2. in a command declaration, to set the default for a particular command (see **3.14.1 Command Declarations**);
3. in a command call (see **3.15.3 Calling a Command**).

The *CommandLength* parameter is one of:

Long Command	Extended $\mathbf{L_c}$/$\mathbf{L_e}$ protocol is used for all commands.
Enable Long Command	Extended $\mathbf{L_c}$/$\mathbf{L_e}$ protocol is used when required.
Disable Long Command	Extended $\mathbf{L_c}$/$\mathbf{L_e}$ protocol is never used.

This statement is allowed in a BasicCard program, although it has no effect.

3.4 Data Storage

All variables in a ZC-Basic program belong to one of four *data storage* classes**:** **Eeprom**, **Public**, **Static**, or **Private**.

3.4.1 Eeprom data

EEPROM is the BasicCard's equivalent of a hard disk. It retains its contents while the card is powered down in the customer's wallet. EEPROM contains your ZC-Basic program (compiled into P-Code), directories and files, and all permanent variables (such as the customer's name or the credit balance in the card). For example:

```
Eeprom CustomerName$ = "" ' We don't know customer's name yet
Eeprom Balance& = 500     ' Free 5-euro bonus for new members
```

If you don't specify an initial value, the data will be initialised to zero. This initialisation takes place when the program (P-Code and data) is downloaded to the card.

Eeprom data has global scope – it can be accessed by all procedures in the program.

3.4.2 Public and Static data

The RAM data area contains **Public** and **Static** data, that retains its value as long as the BasicCard remains powered up in the card reader (or until another Application is selected in the MultiApplication BasicCard). **Public** data has global scope; **Static** data has local scope – it can only be accessed by the procedure that declared it.

Public and **Static** data can be initialised, just like **Eeprom** data. The initialisation takes place whenever the card is powered up (or in the MultiApplication BasicCard, whenever the Application is selected).

3.4.3 Private data

Data declared in a procedure as **Private** exists only until the procedure returns. It is allocated on the P-Code stack every time the procedure is called. It has local scope. **Private** data can be initialised with constant values:

```
Private LoopCounter = 100
```

This initialisation takes place every time the procedure is called. Uninitialised **Private** data is set to zero when the procedure is called.

You don't have to declare every variable before you use it. If the compiler meets a variable name that it doesn't recognise, it implicitly declares it as **Private** and issues a warning message – unless you have overridden this behaviour with the **Option Explicit** statement (see **3.23.4 Explicit Declaration of Variables and Arrays**), or by declaring the procedure itself **Static** (see **3.13 Procedure Definition**).

3.5 Data Types

ZC-Basic supports the following data types:

Byte	1-byte unsigned integer. Range: 0 to 255.
Integer	2-byte signed integer. Range: –32768 to +32767.
Long	4-byte signed integer. Range: –2147483648 to +2147483647.
Long64	8-byte signed integer. Range: –9223372036854775808 to 9223372036854775807.
Single	4-byte single-precision floating-point number (denormalised IEEE format: 1 sign bit, 8-bit exponent, and 23-bit mantissa with implied msb=1 unless exponent is zero). Precision: 7 decimal digits. Range: +/–1.401298E–45 to +/–3.402823E+38.
Double	8-byte double-precision floating-point number (denormalised IEEE format: 1 sign bit, 11-bit exponent, and 52-bit mantissa with implied msb=1 unless exponent is zero). Precision: 7 decimal digits. Range: +/–4.940656E-324 to +/–1.797693E+308.
String	Character string. Its maximum length is 16384 in a Terminal program, 2048 in **ZC7**- and **ZC8**-series BasicCards, and 254 in other BasicCards.
String*n	Fixed-length string, **n** bytes long. It has the same maximum length as a **String**.

Data types **Long64** and **Double** are available in Terminal programs, and in **ZC7-** and **ZC8-**series BasicCards. You may also define your own data types – see **3.8 User-Defined Types**.

3.6 Arrays

An array in ZC-Basic can belong to any of the four data storage classes (**Eeprom**, **Public**, **Private**, **Static**), and its elements may be of any type (**Byte**, **Integer**, **Long**, **Long64**, **Single**, **Double**, **String**, **String*n**, or a user-defined type). It may have up to 32 dimensions. In Enhanced BasicCard programs, the upper and lower bounds for each dimension are subject to the constraints:

$-32 \le \textit{lower bound} \le 31$ and $\textit{lower bound} \le \textit{upper bound} \le \textit{lower bound} + 1023$

All arrays are either **Dynamic** or fixed-size. The upper and lower bounds of a fixed-size array must be constant expressions, and can't be changed. The bounds of a **Dynamic** array can be any integer expression, and the array can be re-sized at any time with a **ReDim** statement. However, the number of dimensions of a **Dynamic** array can't be changed.

If any of the subscripts in an array access is out of bounds, a run-time P-Code error is generated.

The **ReDim** statement has the following syntax:

ReDim *array* **(***bounds* [**,** *bounds***, . . .**]**)** [**As** *type*] [**,** *array* **(***bounds* [**,** *bounds***, . . .**]**)** [**As** *type*]**, . . .**]

array If *array* has already been declared, it must be a **Dynamic** array, and one *bounds* specifier must be present for each dimension. (In this case, **As** *type* is not required, but if present it must match the type as originally declared.) If *array* has not yet been declared, then the **ReDim** statement does double duty as a data declaration statement. In other words, the statement

> **ReDim** *array* **(***bounds* [**,** *bounds***, . . .**]**)** [**As** *type*]

is expanded to

> **Dim Dynamic** *array* **(** [**, , . . .**]**)** [**As** *type*]
> **ReDim** *array* **(***bounds* [**,** *bounds***, . . .**]**)**

(The **Dim** statement is described in **3.7 Data Declaration**.)

bounds The *bounds* specifier gives the upper and lower bounds for each dimension, in the form [*lower-bound* **To**] *upper-bound.* If *lower-bound* is not given, it defaults to 0, unless otherwise specified in an **Option Base** statement (see **3.23.3 Array Subscript Base**).

An array can be cleared with the **Erase** statement:

> **Erase** *array* [**,** *array***, . . .**]

If *array* is fixed-size, all its elements are set to zero. If *array* is **Dynamic**, its data area is freed. In either case, if the elements of *array* are of type **String**, they are all freed.

3.7 Data Declaration

Data items and arrays are declared and initialised in a *data declaration statement.* A data declaration statement consists of a sequence of data declarations separated by commas. Data may optionally be initialised with constant values:

storage-class [**Dynamic**] [**ReadOnly**] *data-declaration* [=*value*] [**,** *data-declaration* [=*value*]**, . . .**]

storage-class — This can be **Eeprom**, **Public**, **Private**, or **Static**. The keyword **Dim** is also allowed; outside a procedure, **Dim** is a synonym for **Public**, and inside a procedure, it has the same meaning as **Private** (or **Static** in a procedure declared as **Static**).

Dynamic — If the **Dynamic** keyword is present, then all arrays declared in the statement are **Dynamic** arrays.

ReadOnly — The **ReadOnly** keyword has two functions in a data declaration:

- the compiler disallows any attempts to write to **ReadOnly** data;
- in the 60.5-Kb **ZC5.6** BasicCard, **Eeprom ReadOnly** data is stored in the **CONST** region, which can be located in Flash memory. This allows full utilisation of the Flash and Eeprom memory in the card.

In addition, **ReadOnly** can be used in a procedure parameter declaration – see **3.16.2 Read-Only Parameters**.

data-declaration — This field takes one of two forms:

1. For scalar (non-array) data, *data-declaration* has the form

name [**As** *type*] [**At** *address*]

The type of the variable *name* is determined as follows:

- by *type* if [**As** *type*] is present;
- otherwise, by the last character of *name* if it belongs to the following list:

Character:	`@`	`%`	`&`	`^`	`!`	`#`	`$`
Data type:	**Byte**	**Integer**	**Long**	**Long64**	**Single**	**Double**	**String**

- otherwise, by the initial character of *name*, as specified in the most recent **DefType** statement (see **3.23.2 DefType Statement**).

By default, all initial characters are assigned to **Integer** type in ZC-Basic, as if by the statement **DefInt A–Z**.

The address of the variable *name* is automatically assigned by the compiler, unless overridden by [**At** *address*]. If present, *address* takes the form *var*[+*constant*], where *var* is the name of a previously declared variable. The new variable must be entirely contained within the previously-declared variable.

2. If an array is being declared, *data-declaration* has the form

> *array* **(***bounds* [**,** *bounds***, . . .**]**)** [**As** *type*]

The type of the elements of the array is determined as described above for scalar variables. The form of the bounds specifier is described in the previous section under **ReDim**. There is an additional possibility – the empty array syntax:

> *array* **([, . . .])** [**As** type]

This declares a **Dynamic** array, while deferring the allocation of the array to a later time. The following example declares empty **Dynamic** arrays **A1**, **A2**, and **A3** with one, two, and three dimensions respectively:

```
Dim A1()
Dim A2(,)
Dim A3(,,)
```

Otherwise, *array* is **Dynamic** if (i) the **Dynamic** keyword was specified; or (ii) any of its bounds is non-constant.

As a special case, if initialisation data is present, then the last (or only) subscript bound may be omitted – it is calculated from the number of elements in the initialisation list. For instance:

```
Byte A@() = 7,8,9
Integer B(1 To 3,) = 101,102,103,104,105
```

Assuming **Option Base** has not been set (see *3.23.3 Array Subscript Base*), this is the same as

```
Byte A@(0 To 2) = 7,8,9
Integer B(1 To 3, 0 To 1) = 101,102,103,104,105,0
```

If no initialisation data is present, the data item or array is initialised to zero (or empty strings in the case of **String** data). In ZC-Basic, any type of data may be initialised, with two exceptions: **Dynamic** arrays with non-constant initial bounds, and **Private Dynamic** arrays. Initialisation data must be constant. If an array is initialised, the data must be specified in the order of the array elements, with the leftmost subscript varying the fastest ('column-major' order). For instance, the following example initialises each element of a 2x2 **String** array to contain an ASCII description of itself:

```
Option Base 1 ' Set lower bound of arrays to 1
Private X$(2,2) = "X$(1,1)", "X$(2,1)", "X$(1,2)", "X$(2,2)"
```

If the end of the initialisation data is reached before the array has been filled, the rest of the array is initialised to zero (or empty strings for a **String** array).

Fixed-length **String*n** data can be initialised in two ways: as a string, or as a list of bytes. These two ways can be combined, but the string must be the last data item in the list. For example:

```
Eeprom S1 As String*5 = "ABC" ' Padded with two NULL bytes
Public S2 As String*3 = &H81, &H82, &H83
Private S3 As String*7 = 3, 4, "XYZ"
Rem   This is equivalent to:
Rem       Private S3 As String*7 = 3, 4, 88, 89, 90, 0, 0
```

3.8 User-Defined Types

ZC-Basic supports the user definition of structured data types:

> **Type** *type-name*
> *member-name* [**As** *type*] [**,** *member-name* [**As** *type*]**, . . .**]
> *member-name* [**As** *type*] [**,** *member-name* [**As** *type*]**, . . .**]
> **. . .**
> **End Type**

type-name and *member-name* are regular identifiers. The *type* of each member can be **Byte**, **Integer**, **Long**, **Single**, **String*n**, or another user-defined type. It may not be an array, or a **String** of variable length. The total size of all the members must not exceed 254 bytes.

If *var* is a variable or array element of type *type-name,* then the members of *var* are referred to using the syntax *var***.***member-name* (as in the 'C' programming language). For example:

```
Type Point: X!, Y!: End Type ' Character '!' => type Single...
Type Rectangle
    Area As Single ' ...or the type can be declared explicitly
    TopLeft As Point
    BottomRight As Point
End Type

Sub Area (R As Rectangle)
    Width! = R.BottomRight.X! - R.TopLeft.X!
    Height! = R.BottomRight.Y! - R.TopLeft.Y!
    R.Area = Width! * Height!
End Sub
```

A user-defined type can be copied as a unit, with a single assignment statement:

```
Public UnitSq As Rectangle = 0,0,0,1,1 ' BottomRight = (1.0,1.0)
Call Area (UnitSq) ' Fill in the Area
Public RA(10) As Rectangle
For I = 1 To 10 : RA(I) = UnitSq : Next I
```

Variables or array elements of the same user-defined type can be compared for equality using = and <> (but the comparison operators < , > , <= , and >= are not allowed).

3.9 Expressions

An *expression* is built up by applying *operations* to *terms*. For example:

```
X + 5        ' Apply '+' (addition) to terms X and 5
A(I) * Rnd   ' Apply '*' (multiplication) to terms A(I) and Rnd
S$ + "0"     ' Apply '+' (concatenation) to terms S$ and "0"
```

A term can be one of the following:

- A constant: the type of a constant term is **Byte**, **Integer**, **Long**, or **Long64** (depending on the value of the constant) for whole-number expressions, **Single** or **Double** for floating-point expressions, and **String** for string constants.
- A scalar variable, an array element, or a member of a variable or array element of user-defined type.
- A function call. This can be a user-defined function or command, or a built-in function (such as **Abs**, **Sqrt**, **LBound**, **Chr$**, or **CurDir**).
- An array name, with no parentheses (or an empty pair of parentheses). This returns the address of the data area of the array, so that you can check whether a dynamic array has been allocated or not. For instance:

```
Eeprom Dynamic A() ' Declare an Integer array
...
If A = 0 Then Redim A (10) ' or 'If A() = 0...'
```

An expression has one of the following types: **Byte**, **Integer**, **Long**, **Long64**, **Single**, **Double**, **String**, *boolean*, or *user-defined*. A boolean expression is an expression of type **Integer** that is the result of a comparison; it takes the value **True** (–1) or **False** (0). Normally a boolean expression is treated the same as an **Integer** expression; any exceptions are noted below.

3.9.1 Numerical Expressions

If *expr1* and *expr2* are numerical expressions (i.e. expressions of type **Byte**, **Integer**, **Long**, **Long64, Single, Double**, or boolean), the following operations are allowed, grouped in descending order of priority:

Group 1	– *expr1* + *expr1*	Unary minus Unary plus (has no effect)
Group 2	**Not** *expr1*	Bitwise complement
Group 3	*expr1* * *expr2* *expr1* / *expr2* *expr1* **Mod** *expr2*	Multiplication Division Remainder
Group 4	*expr1* + *expr2* *expr1* – *expr2*	Addition Subtraction
Group 5	*expr1* **Shl** *expr2* *expr1* **Shr** *expr2* *expr1* **ShrL** *expr2* *expr1* **Rol** *expr2* *expr1* **Ror** *expr2* *expr1* **Rol@** *expr2* *expr1* **Ror@** *expr2*	Shift Left Shift Right (arithmetical, with sign preserved) Shift Right Logical (with sign bit cleared) Rotate Left Rotate Right Rotate Byte Operand Left Rotate Byte Operand Right
Group 6	*expr1* < *expr2* *expr1* <= *expr2* *expr1* > *expr2* *expr1* >= *expr2*	**True** if *expr1* is less than *expr2* **True** if *expr1* is less than or equal to *expr2* **True** if *expr1* is greater than *expr2* **True** if *expr1* is greater than or equal to *expr2*
Group 7	*expr1* = *expr2* *expr1* <> *expr2*	**True** if *expr1* is equal to *expr2* **True** if *expr1* is not equal to *expr2*
Group 8	*expr1* **And** *expr2*	Bitwise **And**
Group 9	*expr1* **Xor** *expr2*	Bitwise exclusive-or
Group 10	*expr1* **Or** *expr2*	Bitwise **Or**

The priority of an operator determines the order of the operations. For instance, **3 + –5 * 7** is evaluated as **3 + ((–5) * 7)** , and **A Or B And C** is evaluated as **A Or (B And C)**.

Numerical Operators

Groups 1, 3, and 4 are the *numerical operators*. The type of the resulting expression is determined as follows:

- If *expr1* or *expr2* is **Double**, then the other is converted to **Double** if necessary; the resulting expression if of type **Double**.
- Otherwise, if *expr1* or *expr2* is **Single**, then the other is converted to **Single** if necessary; the resulting expression if of type **Single**.
- Otherwise, if *expr1* or *expr2* is **Long64**, then the other is converted to **Long64** if necessary; the resulting expression if of type **Long64**.
- Otherwise, if *expr1* or *expr2* is **Long**, then the other is converted to **Long** if necessary; the resulting expression if of type **Long**.
- Otherwise, *expr1* and *expr2* are converted to **Integer**; the resulting expression is of type **Integer**.

Note: Even if *expr1* and *expr2* are both **Byte** expressions, they are converted to **Integer** before any operation is performed. (This means that the only expressions of type **Byte** are those consisting of a single term.)

Shift/Rotate Operators

The *shift/rotate operators* in Group 5 are currently available in Terminal programs; in Professional BasicCards **ZC5.4** (from **REV J**), **ZC5.5**, **ZC5.6**, and **ZC7**-series; and in MultiApplication BasicCards. *expr2* is treated as an unsigned **Integer** (so, for instance, *expr1* **Shl** *expr2* will always be zero if *epxr2* < 0 or *expr2* > 63). These operators never generate an overflow error.

Comparison Operators

Groups 6 and 7 are the *comparison operators*. Exactly the same conversions are applied as for the numerical operators, but the type of the resulting expression is boolean.

Bitwise Operators

Groups 2, 8, 9, and 10 are the *bitwise* operators. Bitwise operations are never performed on **Single** or **Double** expressions; if *expr1* or *expr2* is **Single** (resp. **Double**), it is converted to **Long** (resp. **Long64**) before a bitwise operation is performed. If both *expr1* and *expr2* are of boolean type, then the result is also of boolean type.

There is a special rule concerning the evaluation of expressions of boolean type:

> If *expr1* and *expr2* are both of boolean type, and one of the expressions
>
> *expr1* **And** *expr2* *expr1* **Or** *expr2*
>
> occurs in the program, then *expr2* is not evaluated if the value of the whole expression can be deduced from the value of *expr1* alone.

In other words:

- if *expr1* is **False**, then "*expr1* **And** *expr2*" is always **False** as well, so *expr2* is not evaluated;
- if *expr1* is **True**, then "*expr1* **Or** *expr2*" is always **True** as well, so *expr2* is not evaluated.

This is important if the evaluation of *expr2* has any side-effects. For instance:

```
If X! = 0 Or F(1/X!) > 100 Then GoTo 100
```

If **X!** is zero, then **1 / X!** is not evaluated (which would otherwise cause a run-time error), and the function **F** is not called (which might have had side effects, such as changing **Public** data).

3.9.2 String Expressions

If either *expr1* or *expr2* is of type **String**, then the other must be of type **String** as well: there are no mixed numerical/string operations. The following string operations are allowed:

Group 1	*expr1* + *expr2*	String concatenation
	expr1 **Xor** *expr2*	Byte-by-byte **Xor**
Group 2	*expr1* < *expr2*	**True** if *expr1* is less than *expr2*
	expr1 <= *expr2*	**True** if *expr1* is less than or equal to *expr2*
	expr1 > *expr2*	**True** if *expr1* is greater than *expr2*
	expr1 >= *expr2*	**True** if *expr1* is greater than or equal to *expr2*
Group 3	*expr1* = *expr2*	**True** if *expr1* is equal to *expr2*
	expr1 <> *expr2*	**True** if *expr1* is not equal to *expr2*

The resulting expression is of **String** type after string concatenation and **Xor** (Group 1), and of boolean type after string comparison (Groups 2 and 3). The comparison operations in Group 2 are performed by finding the first characters that differ in the two strings, and comparing their ASCII values. In ASCII, all lower-case letters are greater than all upper-case letters, so for instance "abc" is greater than "XYZ". For case-insensitive comparison, use **UCase$** or **LCase$** to convert both arguments to the same case. For example:

```
If UCase$(S1$) > UCase$(S2$) Then T$ = S1$: S1$ = S2$: S2$ = T$
```

The byte-by-byte **Xor** operator is available in Terminal programs, and in **ZC7-** and **ZC8-**series BasicCards from **REV D**.

3.9.3 Expressions of User-Defined Type

The only operation allowed on user-defined types is comparison for equality:

Group 1	*expr1* = *expr2*	**True** if *expr1* is equal to *expr2*
	expr1 <> *expr2*	**True** if *expr1* is not equal to *expr2*

The resulting expression is of boolean type.

3.10 Assignment Statements

An assignment statement has the form

[**Let**] *var* **=** *expression*

where *var* is a scalar variable, or an array element, or a member of a variable or array element of user-defined type. The **Let** keyword is optional. The following rules apply:

- If *var* has numerical type (**Byte**, **Integer**, **Long**, **Long64**, **Single**, or **Double**), then *expression* must have numerical type.
- If *var* has type **String** or **String*n**, then *expression* must have type **String**.
- If *var* has a user-defined type, then *expression* must have the same user-defined type.

There are four special string assignment statements:

[**Let**] **Mid$ (***string*, *start* [, *length*]**)** = *expression*

[**Let**] **Left$ (***string*, *length***)** = *expression*

[**Let**] **Right$ (***string*, *length***)** = *expression*

[**Let**] *string* **(***n***)** = *expression*

Mid$ overwrites *length* characters of *string* with the value *expression*, starting from position *start*. (The first character in the string has position 1.) A value of *start* less than 1 results in a run-time error; a value of *start* greater than the length of *string* is not an error, but no characters are copied. If *length* is absent, or if *start+length* is greater than the length of *string*, the whole of rest of the string is overwritten.

Left$ overwrites the first *length* characters of *string* with the value *expression*. If *length* is greater than the length of *string*, the whole of *string* is overwritten.

Right$ overwrites the last *length* characters of *string* with the value *expression*. If *length* is greater than the length of *string*, the whole of *string* is overwritten.

In ZC-Basic, *string* **(***n***)** is shorthand for **Mid$ (***string*, *n*, 1**)**. So the last statement in the above list assigns the first character of *expression* to the *n*th character of *string*.

In the first three string assignment statements, only the first *length* characters of *expression* are copied into *string.* If *length* is greater than the length of *expression*, then the destination sub-string is filled out with NULL characters (i.e. ASCII zeroes).

3.11 Type Casting

Since Version 7.10 of the software, the ZC-Basic compiler implements a simple form of type casting. This was required for the new Transaction Manager System Library (see **7.5 The TMLib Transaction Manager Library**), but it can be useful in other contexts too.

3.11.1 Casting a Variable

Let *var* be a scalar variable (or an array element, or a member of a variable or array element of user-defined type) that is of fixed size, i.e. not of type **String** (although it may be a fixed-length string).

Then the expression

var **As** *type*

is interpreted as a variable at the same address as *var*, of type *type*. If *type* is **String**, then it is understood as **String***n*, where *n* is the size of *var*. The size of *type* must be no greater than the size of *var*. For instance:

```
Private S$ As String*4 = Chr$(4,3,2,1)
Print S$ As Integer
```

An **Integer** is two bytes, so this displays 1027, which is the decimal representation of &H0403, the first two bytes of **S$**.

Type casting may also be used on the left-hand side of an assignment statement. For instance:

```
Private X As Long
X As Integer = &H4142
```

This sets the two most significant bytes of **X** to &H4142.

Instead of type casting, you can usually achieve the same result using

var **At** *address*

in the data declaration – see **3.7 Data Declaration**. For instance, the second example above could be written:

```
Private X As Long
Private X% At X
X% = &H4142
```

But you can't use this to access an array element – for that, you need type casting with **As**.

3.11.2 Casting a Constant to a String

Let *expr* be a constant numeric expression (of type **Byte**, **Integer**, **Long**, **Long64**, **Single**, **or Double**). Then the expression

expr **As String***n*

denotes a constant, fixed-length string (of length *n*), whose rightmost bytes agree with the least significant bytes of *expr*. The expression

expr **As String***type*

is short for

expr **As String*Len(***type***)**

For instance:

expr **As String*Integer '** Same as **String*Len(Integer)**, i.e. **String*2**

3.12 Program Control

3.12.1 Exit Statements

An **Exit** statement jumps out of an enclosing block of code, according to the type of the statement:

Exit For	Jumps to the statement following the innermost current **For**-loop.
Exit While	Jumps to the statement following the innermost current **While**-loop.
Exit Do	Jumps to the statement following the innermost current **Do**-loop.
Exit Case	Jumps to the statement following the next **End Select**.
Exit Sub	Returns from a subroutine to the calling procedure.
Exit Function	Returns from a function to the calling procedure.
Exit Command	Returns from a BasicCard command to the caller in the Terminal program.

Exit Exits the program. **Exit** in a Terminal program returns to the operating system; **Exit** in a BasicCard program returns to the caller in the Terminal program.
Note: The **Exit** statement (with no parameters) exits the program immediately, without freeing **Private** strings and arrays. This is not a problem in the Terminal program, but it can cause **pcOutOfMemory** errors in subsequent commands in a BasicCard program, until the card is reset. So you should only use such an **Exit** statement in a BasicCard program if you detect an error condition that prevents the card from continuing the command-response session.

3.12.2 Labels

There are two types of label in ZC-Basic: named labels, and line numbers. A named label is an identifier followed by a colon. A line number is simply a decimal number, which may or may not be followed by a colon. A label, of either type, may only be accessed from within the procedure that defines it. Label names and line numbers must be unique within each procedure, but the same name or line number can be used in two different procedures.

3.12.3 GoTo

The simplest program control statement is the **GoTo** statement:

> **GoTo** *label*
> **. . .**
> *label***:**

The program continues execution at the statement following *label*.

Note: You can’t use **GoTo** to jump from one procedure to another.

3.12.4 GoSub

A procedure can call its own private subroutines with the **GoSub** statement. Such a private subroutine is not a procedure; it has no parameters, and no data of its own. It is simply a part of the procedure that defines it. It returns with the **Return** statement:

> **GoSub** *label*
> **. . .**
> *label***:**
>   *subroutine-code*
> **Return** [*return-label*]

If *return-label* is specified in the **Return** statement, the subroutine returns there; otherwise it returns to the statement following the **GoSub** call.

3.12.5 If-Then-Else

The **If** statement executes code depending on the value of a conditional expression:

> **If** *condition* **Then**
>   *code block*
> **End If**

The full form of the **If-Then-Else** block is as follows:

> **If** *condition 1* **Then**
>   *code block 1*
> [**ElseIf** *condition 2* **Then**
>   *code block 2*]
> [**ElseIf** *condition 3* **Then**
>   *code block 3*]
> **. . .**
> [**Else**
>   *code block n*]
> **End If**

Each condition is a numerical expression. *code block i* is executed if *condition i* is the first non-zero (true) condition. If all the conditions are zero (false), then *code block n* is executed, if present.

Single-Line If-Then-Else

If **Then** or **Else** is followed by *code block* without an intervening statement boundary (i.e. a colon or a new line), then the **If-Then-Else** block is terminated at the next new line (by generating an **End If** statement if necessary). This is called a *single-line* If-Then-Else block. For instance:

```
If X = 0 Then GoTo 100

If X = 0 Then Y = 0 : If Z = 0 Then GoTo 100 ' Can be nested

If X < 0 Then
  X = 0
ElseIf X > 50 Then X = 50

If X > 0 Then
  X = 0
Else X = X + 1
```

These are equivalent to:

```
If X = 0 Then
  GoTo 100
End If

If X = 0 Then
  Y = 0
  If Z = 0 Then
    GoTo 100
  End If
End If

If X < 0 Then
  X = 0
ElseIf X > 50 Then
  X = 50
End If

If X > 0 Then
  X = 0
Else
  X = X + 1
End If
```

3.12.6 For-Loop

The **For**-loop executes a block of code a specified number of times:

> **For** *loop-var* = *start* **To** *end* [**Step** *increment*]
> [*code block*]
> [**Exit For**]
> [*code block*]
> **Next** [*loop-var*]

loop-var	A numerical variable, used to count the number of times the **For**-loop has been executed.
start	A numerical expression, the initial value of *loop-var*.
end	A numerical expression. The **For**-loop terminates when *loop-var* passes this value. More precisely: If *increment* ≥ 0, then the **For**-loop terminates when *loop-var* > *end.* If *increment* < 0, then the **For**-loop terminates when *loop-var* < *end.*
increment	The amount by which *loop-var* is incremented after each execution of the **For**-loop. If [**Step** *increment*] is absent, *increment* takes the value 1.

The **Exit For** statement breaks out of the **For**-loop to the statement following the **Next** instruction.

loop-var is optional in the **Next** statement (but it can be useful as a reminder if the loop is large).

If **For**-loops are nested, the **Next** statement can specify more than one loop variable. For example:

```
For I = 1 To 10: For J = 1 To 10: A(I,J) = 0 : Next I, J
```

Any **Exit For** statement, even in the innermost loop, breaks out to the statement following the **Next** statement. So the following example prints only the value **11**:

```
For I = 1 To 2 : For J = 1 To 2
  Print 10*I + J : Exit For
Next I, J
```

However, this example prints **11** and **21**:

```
For I = 1 To 2 : For J = 1 To 2
  Print 10*I + J : Exit For
Next J : Next I
```

3.12.7 While-Loop and Do-Loop

The **While**-loop is executed as long as *condition* is non-zero:

> **While** *condition*
> [*code block*]
> [**Exit While**]
> [*code block*]
> **Wend**

The **Do**-loop has more flexibility:

> **Do** [{**While | Until**} *condition*]
> [*code block*]
> [**Exit Do**]
> [*code block*]
> **Loop** [{**While | Until**} *condition*]

The optional [{**While | Until**} *condition*] may appear at the beginning or the end of the **Do**-loop, but not both. If it appears at the end, then the loop is always executed at least once. If neither is present, then the loop is executed endlessly until left by some other means (such as **Exit Do** or **GoTo**).

3.12.8 Select Case

Select Case executes one of several blocks of code, depending on the value of a test expression:

> **Select Case** *test-expression*
> **Case** *case-test* [**,** *case-test***, . . .**]
>   [*code block*]
>   [**Exit Case**]
>   [*code block*]
> **Case** *case-test* [**,** *case-test***, . . .**]
>   [*code block*]
>   [**Exit Case**]
>   [*code block*]
> **. . .**
> [**Case Else**
>   [*code block*]
>   [**Exit Case**]
>   [*code block*]]
> **End Select**

test-expression	An expression of any type (numerical, **String**, or user-defined)
case-test	This takes one of three forms:

expression	**True** if *test-expression* = *expression*
expr1 **To** *expr2*	**True** if *expr1* ≤ *test-expression* ≤ *expr2*
[**Is**] *op expr*	**True** if *test-expression op expr*, where *op* is one of the six comparison operators: < <= > >= = <> The **Is** keyword is optional.

If *test-expression* is of user-defined type, only the first of these three forms is valid.

The **Select Case** statement executes the code following the first **Case** statement that contains a *case-test* that is **True**. If more than one such **Case** statement exists, only the first is executed. If no such **Case** statement exists, then the code following the **Case Else** statement is executed (and if there is no **Case Else** statement, none of the code in the **Select Case** block is executed). The **Exit Case** statement jumps to the statement following **End Select**.

3.12.9 Computed GoTo and Computed GoSub

You can jump to one of a list of labels depending on the value of a test expression:

> **On** *expression* { **GoTo** | **GoSub** } *label1* [**,** *label2***, . . . ,** *labeln*]

expression — An expression of type **Integer**. If it is equal to r, with $1 \le r \le n$, then **GoTo** *labelr* or **GoSub** *labelr* is executed. If *expression* < 1 or *expression* > n, execution proceeds with the following statement.

3.13 Procedure Definition

A typical ZC-Basic program consists mainly of procedure definitions. Each procedure is either a **Subroutine**, a **Function**, or a **Command**. The **Private** and **Static** variables declared in a procedure belong to that procedure alone, and can't be accessed from other procedures (such variables are said to have local scope); **Public** and **Eeprom** variables can be accessed from all procedures (they have global scope).

3.13.1 Subroutine

The simplest procedure type is the subroutine. A subroutine returns no value to the caller, except through its arguments. A subroutine definition is as follows:

[**Static**] **Sub** *proc-name* **(**[*param-def*, *param-def*, **. . .**]**)**
  [*procedure code*]
  [**Exit Sub**]
  [*procedure code*]
End Sub

Static If the **Static** keyword is present in the definition, undeclared variables in the procedure have **Static** storage class, instead of **Private**.

param-def [{**ByVal** | **ByRef**}] [**ReadOnly**] *param-name*[**()**] [**As** *type*], where *param-name* is a variable name by which the parameter is accessed in *procedure-code*. See **3.16 Procedure Parameters** for a full discussion of parameters.

3.13.2 Function

A **Function** is a **Subroutine** that returns a value to the caller. A function definition is as follows:

[**Static**] **Function** *proc-name* **(**[*param-def*, *param-def*, **. . .**]**)** [**As** *type*]
  [*procedure code*]
  [*proc-name* = *expression*]
  [**Exit Function**]
  [*procedure code*]
End Function

Static If the **Static** keyword is present in the definition, undeclared variables in the procedure have **Static** storage class, instead of **Private**.

param-def [{**ByVal** | **ByRef**}] [**ReadOnly**] *param-name*[**()**] [**As** *type*], where *param-name* is a variable name by which the parameter is accessed in *procedure-code*. See **3.16 Procedure Parameters** for a full discussion of parameters.

The return type of the function is determined as if *proc-name* were a variable name: from "**As** *type*" if present; otherwise from the last character in *proc-name* if it is a type character (**@**, **%**, **&**, **!**, or **$**); otherwise from the first character in *proc-name*. (The type characters are defined in **3.2 Tokens**.) A function can have any return type that is not an array.

Inside the function, *proc-name* behaves like a **Private** variable. It is initialised to zero when the function is called, and its value is returned to the caller when the function exits.

3.13.3 Command

A command is defined like a subroutine, but you must specify the two ID bytes (**CLA** and **INS**) by which the command will be invoked:

[**Static**] **Command** [*CLA*] [*INS*] *proc-name* **(**[*PreSpec*,] [*param-def*, *param-def*, **. . .**] [**,** *PostSpec*]**)**
  [*procedure code*]
  [**Exit Command**]
  [*procedure code*]
End Command

Static If the **Static** keyword is present in the definition, undeclared variables in the procedure have **Static** storage class, instead of **Private**.

CLA The 'Class' byte. All the pre-defined commands in the BasicCard have **CLA=&HC0**, so you should normally avoid this value for your own commands, unless you specifically want to override a pre-defined command. If *CLA* is not present, **CLA** must be present in *PreSpec*.

INS — The 'Instruction' byte. The compiler accepts any value; but in a card that uses the **T=0** protocol, this byte must be even, and the top nibble may not be **6** or **9**. If *INS* is not present, **INS** must be present in *PreSpec*.

PreSpec — Pre-parameter specification. It may contain the following terms, in the following order, and separated by commas:

CLA=*constant*	An alternative way of specifying **CLA**
INS=*constant*	An alternative way of specifying **INS**
Lc=0	Only relevant under the **T=0** protocol

In a Professional BasicCard using the **T=0** protocol, **Lc=0** defines the command as having no incoming data – a **Case 2** command in the terminology of **8.3.2 APDU Transmission by T=0**. You only need to use this if:

- you are implementing a pre-existing **T=0** command specification; or
- you want to minimise **T=0** communications overhead to improve performance.

param-def — [{**ByVal** | **ByRef**}] [**ReadOnly**] *param-name* [**As** *type*], where *param-name* is a variable name (but not an array name) by which the parameter is accessed in *procedure-code*. See **3.16 Procedure Parameters** for a full discussion of parameters.

PostSpec — Post-parameter specification, only relevant under the **T=0** protocol. You only need to use this if:

- you are implementing a pre-existing **T=0** command specification; or
- you want to minimise **T=0** communications overhead to improve performance.

It may take one of two forms:

Disable Le
Input Le

Disable Le defines the command as having no outgoing data – a **Case 3** command in the terminology of **8.3.2 APDU Transmission by T=0**.

Input Le is used to distinguish the two sub-cases of **Case 4** commands – *Case 4S.2* and *Case 4S.3* in **8.3.6 Case 4: Incoming and Outgoing Data**. In *Case 4S.2* commands, **ResponseLength** is specified by the Terminal program in the **Le** parameter, so the Terminal program must send **Le** before the command is executed; in *Case 4S.3* commands, the BasicCard decides for itself what **ResponseLength** should be. **Input Le** defines the command as a *Case 4S.2* command.

Notes:

1. The special syntax "[**Static**] **Command Else** *proc-name* **(**[*param-def*, *param-def*, **. . .**]**)**" defines a *default command* in the card, that is called when the BasicCard receives a command with unrecognised *CLA* and *INS*.
2. In some cards (currently **ZC5.4** from **REV J**, **ZC5.5** from **REV E**, **ZC5.6**, **ZC6.5**, and **ZC7**- and **ZC8-**series), if the Application contains a subroutine **ClaInsFilter**(), this subroutine is called whenever a command is received, before the BasicCard operating system looks for a match for **CLA** and **INS**. If you modify **CLA** or **INS** in this subroutine, the card will behave as if the modified values had been received.
3. Arrays are not allowed as **Command** parameters.
4. A **Command** definition is only valid in a BasicCard program; it is not allowed in a Terminal program.
5. Some obsolete **T=0** card readers expect the card to send an Acknowledge byte even if the command has no incoming or outgoing data (a Type 1 command). You can tell the card to do this by specifying [**Static**] **Command** [*CLA*] [*INS*] *proc-name* **(Enable Ack)**.

3.14 Procedure Declaration

The compiler can't process a procedure call unless it knows what kinds of parameters the procedure accepts. It knows this if the procedure has already been defined:

```
Function Square (X!) As Single
  Square = X! * X!
End Function

Sub S()
  Y! = Square (5.5)     ' OK - Square already defined
End Sub
```

But the compiler won't accept the following:

```
Sub S()
  Y! = Square (5.5)     ' Error - Square not defined yet
End Sub

Function Square (X!) As Single
  Square = X! * X!
End Function
```

To call a procedure before it is defined, you must provide a *procedure declaration* that tells the compiler what it needs to know. A procedure declaration starts with the word **Declare**:

Declare Sub *proc-name* **(**[*param-def*, *param-def*, **. . .**]**)**
Declare Function *proc-name* **(**[*param-def*, *param-def*, **. . .**]**)** [**As** *type*]
Declare Command [*CLA*] [*INS*] *proc-name* **(**[*PreSpec*,] [*param-def*, *param-def*, **. . .**] [, *PostSpec*]**)**

If a declaration and a definition of the same procedure occur in the program, then they must match. More precisely:

- for a **Function**, the return type in the declaration must match the return type in the definition;
- for a **Command**, *CLA* and *INS* must be the same in the declaration and the definition;
- the types of the parameters must match exactly;
- the parameter-passing method (**ByVal** or **ByRef**, and **ReadOnly**) and must be the same for each parameter.

However, the names of the parameters don't need to match. Parameter names in a procedure declaration are just place-holders; the only restriction is that they may not be reserved words (see **3.2 Tokens** for a list of reserved words). For example:

```
Declare Function Square (Z!) As Single

Sub S()
  Y! = Square (5.5)             ' OK - Square declared
End Sub

Function Square (X!) As Single ' OK – matches declaration
  Square = X! * X!
End Function
```

3.14.1 Command Declarations

A **Command** declaration has the following general form:

Declare Command [*CLA*] [*INS*] *proc-name* **(**[*PreSpec***,**] [*param-def***,** *param-def***, . . .**] [**,** *PostSpec*]**)**

The *param-def* fields are the same as in **Function** and **Sub** declarations. The *PreSpec* and *PostSpec* fields are available for users who need precise control over the **T=0** and **T=1** Command APDU parameters; otherwise they are not required.

CLA The 'Class' byte. All pre-defined commands in the BasicCard have **CLA=&HC0**, so you should normally avoid this value for your own commands, unless you want to override a pre-defined command. If *CLA* is not present, **CLA** must be present in *PreSpec*, either here or in the procedure call – see **3.15.3 Calling a Command**.

INS The 'Instruction' byte. The compiler accepts any value; but in a card that uses the **T=0** protocol, this byte must be even, and the top nibble may not be **6** or **9**. If *INS* is not present, **INS** must be present in *PreSpec*, either here or in the procedure call – see **3.15.3 Calling a Command**.

PreSpec Pre-parameter specification. This field may contain any of the following terms, in the following order, and separated by commas:

CommandLength
CLA=*constant*
INS=*constant*
P1=*constant*
P2=*constant*
P1P2=*constant*
Lc=*constant*

Each *constant* is a **Byte** expression, except **P1P2** and **Lc**, which are of type **Integer**. See **8.6 Commands and Responses** for definitions of these terms.

CommandLength controls the use of extended L_c/L_e protocol, which allows commands and responses up to 2048 bytes in length (such commands are accepted by **ZC7**-series Professional BasicCards only). It is one of the following:

Long [**Command**]	Always use extended L_c/L_e for this command
Enable Long [**Command**]	Use extended L_c/L_e if necessary
Disable Long [**Command**]	Never use extended L_c/L_e for this command

The *CommandLength* parameter is allowed in all programs, although it only has an effect in Terminal programs.

PostSpec Post-parameter specification. If present, this field takes one of the following forms:

Le=*constant*
Disable Le

Here, *constant* is an **Integer** expression; **Disable Le** specifies that **Le** is absent from the command. See **8.6 Commands and Responses** for a definition of **Le**.

*3.14.2 System Library ProceduresIn Terminal programs, and Professional and MultiApplication BasicCard programs, Library procedures are called via the **SYSTEM** instruction. They are declared as follows:*

In Terminal programs, and Professional and MultiApplication BasicCard programs, Library procedures are called via the **SYSTEM** instruction. They are declared as follows:

Declare Sub *SysCode SysSubcode proc-name* **(**[*param-def***,** *param-def***, . . .**]**)**
Declare Function *SysCode SysSubcode proc-name* **(**[*param-def***,** *param-def***, . . .**]**)** [**As** *type*]

SysCode The System Library identifier, a **Byte** between **&HC0** and **&HFF**.
SysSubcode The procedure sub-code, any **Byte** value.

3.14.3 Erasable CodeBlocks

Sometimes a subroutine or function is only called during card initialisation or personalisation, and is never needed again. Such a procedure can be compiled into an erasable CodeBlock, and deleted when no longer needed, to free up the EEPROM for later use. First, you must define the CodeBlock:

#Pragma CodeBlock "*name***" [, "***name***"***...***]**

This defines one or more CodeBlocks for later use. Then, to put a procedure into a CodeBlock, simply specify the CodeBlock name, in quotes, before the procedure name. For example:

Declare Sub "*codeblockname***"** *subname* **(***paramlist***)**

Such a procedure is called in the usual way. *No check is made to determine whether the CodeBlock still exists* – you have to check this yourself.

To erase a CodeBlock:

Erase "*name***"**

After this, calling any procedure in the CodeBlock will have unpredictable (and probably fatal) effects.

To check whether a procedure in a CodeBlock still exists:

If (*procname***) Then …**

Note that the procedure name is required here, not the CodeBlock name.

Notes:

- This facility is not available for the **ZC6-**series MultiApplication BasicCard.
- A Command cannot be put into a CodeBlock.

3.15 Procedure Calls

3.15.1 Calling a Subroutine

The recommended way to call a subroutine is

Call *procedure-name* **(** [[{**ByVal** | **ByRef**}] *expression***,** [{**ByVal** | **ByRef**}] *expression***, . . .**] **)**

The expressions in the list must match the parameters in the subroutine declaration (or definition) in number and type. (See **3.16 Procedure Parameters** below for a fuller explanation.) If the subroutine takes no parameters, then the parentheses are optional:

Call *procedure-name* [**()**]

Alternatively, ZC-Basic accepts the older subroutine call syntax (with parentheses not allowed):

procedure-name [[{**ByVal** | **ByRef**}] *expression***,** [{**ByVal** | **ByRef**}] *expression***, . . .**]

3.15.2 Calling a Function

A **Function** call returns a value, that can be used as a term in an expression. For example:

```
X! = X! + Square (X!+1)
```

A **Function** can also be called just as if it were a **Subroutine**, in which case the return value is simply discarded.

3.15.3 Calling a Command

A **Command** is called as if it were a **Function** – although it is defined as if it were a **Subroutine**. The reason for this is that the Terminal program automatically returns the command status word (**SW1–SW2**) as if it were the return value of a function. This command status word should always be checked, as it is possible that communications were disrupted for some reason before the command could be successfully completed in the BasicCard.

A **Command** call has the following general form:

var = *command-name* **(**[*PreSpec***,**] *arg-list* [**,** *PostSpec*]**)**

where the *arg-list* field is the same as in **Function** and **Sub** calls. The *PreSpec* and *PostSpec* fields are available for users who need precise control over the **T=0** and **T=1** Command APDU parameters; otherwise they are not required.

PreSpec Pre-parameter specification. This field may contain any of the following terms, in the following order, and separated by commas:

CommandLength
CLA=*expr*
INS=*expr*
P1=*expr*
P2=*expr*
P1P2=*expr*
Lc=*expr*

Each *expr* is a **Byte** expression, except **P1P2** and **Lc**, which is are of type **Integer**. See **8.6 Commands and Responses** for definitions of these terms.

CommandLength controls the use of extended Lc/Le protocol, which allows commands and responses up to 2048 bytes in length in ZC7-series Professional BasicCards. It is one of the following:

Long [**Command**]	Use extended $\mathbf{L_c/L_e}$ for this command call
Enable Long [**Command**]	Use extended $\mathbf{L_c/L_e}$ if necessary
Disable Long [**Command**]	Don't use extended $\mathbf{L_c/L_e}$ for this command call

PostSpec Post-parameter specification. If present, this field takes one of the following forms:

Le=*expr*
Disable Le

Here, *expr* is an **Integer** expression; **Disable Le** specifies that **Le** is absent from the command. See **8.6 Commands and Responses** for a definition of **Le**.

An alternative method of calling a command:

Call *command-name* **(**[*PreSpec***,**] *arg-list* [**,** *PostSpec*]**)**

In this case, the command status word is available in the pre-defined variables **SW1**, **SW2**, and **SW1SW2**.

3.16 Procedure Parameters

3.16.1 Parameter Passing

In traditional Basic, procedure parameters are passed *by value* or *by reference*. Passing by value means that the procedure receives its own copy of the parameter; any changes it makes to this copy are lost when the procedure returns. Passing by reference means that the address (or 'reference') of the parameter is passed to the procedure; knowing its address, the called procedure can change the value of a variable in the calling procedure.

In general, ZC-Basic can't do this, because the BasicCard can't change the value of a variable in the Terminal program directly. However, it uses a *write-back* mechanism to achieve the same effect (and it retains the keywords **ByVal** and **ByRef**, although they are not strictly accurate). With the exception of **String** and array parameters, all parameters are passed by value (in the traditional sense); the value of each parameter is pushed onto the P-Code stack before the procedure is called. The parameters are then referenced like **Private** variables in the called procedure, and can be read or written directly. Then when the procedure returns to the caller, any parameters that were passed **ByRef** are copied back from the stack into their original locations.

By default, all parameters are passed **ByRef** (in the ZC-Basic sense). If the **ByVal** keyword is specified in the procedure definition or declaration, then the following parameter is passed by value, and not written back when the procedure returns. (The **ByRef** keyword is also allowed here, although it is superfluous.) The parameter-passing method specified in the procedure definition or declaration can be overridden for a particular procedure call by specifying **ByVal** or **ByRef** in front of a parameter. (Here **ByRef** is not superfluous if the parameter was specified as **ByVal** in the procedure definition or declaration.)

For the write-back mechanism to be invoked for a given parameter, the parameter-passing method must be **ByRef**, and the expression in the procedure call must be an *assignable* expression – an expression that can appear on the left-hand side of an assignment statement. If you don't want a variable to be changed by a called procedure, you can specify **ByVal**, or you can enclose the variable in parentheses (which is a valid expression, but not an assignable expression). An example may make this clearer:

```
Declare Sub S (X, ByVal Y, ByRef Z) ' 'ByRef' redundant here
Private A, B, C
Call S (A, B, C)                    ' A and C can change
Call S (ByVal A, ByRef B, C)        ' B and C can change
Call S (A+1, B, (C))      ' Nothing can change – 'A+1' and '(C)'
                          ' are not assignable expressions
```

For information on the maximum total size of a parameter list, see **3.24.1 Parameter Size Limits**.

3.16.2 Read-Only Parameters

If a parameter is declared with the **ReadOnly** keyword, then:

- the compiler will check that the parameter is not changed within the procedure;
- the compiler can perform certain optimisations (in particular with **String** parameters) that would not otherwise be possible.

For instance, if a **String** variable is declared **ReadOnly** and is passed as a parameter, then a copy of the string must be made unless the parameter is also declared **ReadOnly**. Also, an array that was declared as **ReadOnly** can only be passed to a procedure if the parameter is also declared **ReadOnly**.

3.16.3 String Parameters

There is an important difference between parameters of type **String** and parameters of type **String*n**. The former occupy 3 or 4 bytes on the P-Code stack, the latter occupy **n** bytes. So you should usually use **String** parameters rather than **String*n** parameters. However, a variable-length string parameter to a **Command** is only allowed if it is the last (or only) parameter; any other string parameters must be of fixed-length **String*n** type.

Note: You can pass a fixed-length string in a **String** parameter, or a variable-length string in a **String*n** parameter; the compiler performs the necessary conversions. The parameter type only determines how the string is passed to the procedure.

For more information on **String** parameters, see **3.24.3 String Parameter Format**.

3.16.4 Array Parameters

An array parameter takes up just two bytes on the P-Code stack (the address of the array descriptor is passed to the procedure – see **3.24.2 Array Descriptor Format**).

An array parameter is specified in a procedure definition or declaration by a pair of parentheses after the parameter name:

*param-name***()** [**As** *type*]

The parentheses must be empty. To pass an array parameter in a procedure call, the array name is sufficient; an empty pair of parentheses after the array name is optional. The type of the array must match exactly the type of the parameter. For example:

```
Declare Sub S (A() As Integer) ' Parentheses required here
Dim X (10) As Integer, Y (20) As Long
Call S (X)     ' OK
Call S (X())   ' Also OK - parentheses optional in call
Call S (Y)     ' Error - Y is Long array, not Integer array
```

The number of dimensions of the array is checked at run-time. The following code will compile, but will generate a run-time error:

```
Declare Sub S (A() As Integer)
Dim X (5, 5, 5)
Call S (X)
...
Sub S (A() As Integer)
A (2, 2) = 0 ' Run-time error - parameter X has 3 dimensions
```

3.16.5 Parameters of User-Defined Type

A parameter of user-defined type is passed to a procedure by pushing every member onto the P-Code stack. The P-Code stack occupies precious RAM, so you should avoid passing large user-defined types as procedure parameters. Otherwise, a parameter of user-defined type behaves just like a parameter of numerical type.

3.17 Built-in Functions

3.17.1 Numerical Functions

Abs(*X*) Returns the absolute value of *X* (that is to say, *X* or –*X*, whichever is positive). The type of the result is the type of *X*, unless *X* is **Byte**, in which case **Abs(*X*)** has type **Integer**.

Rnd Returns a random number of type **Long**: –2147483648 ≤ **Rnd** ≤ 2147483647. See **3.19 Random Number Generation**.

Sqrt(*X*) Returns the square root of *X*. The result is of type **Single**.

3.17.2 Array Functions

LBound(*array* [, *dim*])
UBound(*array* [, *dim*]) These two functions return the lower and upper bounds of subscript *dim* in the given array. If *dim* is not present, the lower or upper bound for the first subscript is returned. The result is of type **Integer**.

3.17.3 String Functions

string **(*n*)** Returns a string of length 1, containing the *n*th character of *string*. (The first byte of the string has position 1.) It is shorthand for **Mid$(*string*, *n*, 1)**.

Asc(*string*) Returns the ASCII value of the first character of *string*, as a **Byte**.

Chr$(*char-code*) Returns a string of length 1, containing the ASCII character with the given *char-code*.

Chr$**($c_1$, c_2, ..., c_n) This is short for **Chr$**($c_1$) + **Chr$(c_2) + ... + **Chr$**($c_n$). Each c_i must be a constant between 0 and 255.

Hex$(*val*) Returns a string containing the hexadecimal representation of the **Long** number *val*.

InStr(*start*, *s1*, *s2*) Returns the offset of the first occurrence of string *s2* in **Mid$(*s1*, *start*)**, or 0 if not found. *start* must be at least 1. This function is available in Terminal programs and in **ZC7**- and **ZC8-**series BasicCards. It is implemented as part of the **MISC** System Library; you must include MISC.DEF to use it.

Left$(*string*, *len*)	Returns the first *len* bytes of *string*.
LCase$(*string*)	Returns *string* with all upper-case letters converted to lower-case.
Len(*string*)	Returns the length of *string*, as a **Byte**.
LTrim$(*string*)	Returns *string* with leading spaces and NULL bytes removed.
Mid$(*string*, *start*[, *len*])	Returns *len* bytes of *string*, starting from position *start*. (The first byte of the string has position 1.) If *start* > **Len(*string*)**, the empty string is returned. If *start* + *len* > **Len(*string*)** , or if *len* is absent, then the whole of *string* from position *start* is returned. If *start* ≤ **0** or *len* < **0**, a run-time error is generated.
Right$(*string*, *len*)	Returns the last *len* bytes of *string*.
RTrim$(*string*)	Returns *string* with trailing spaces and NULL bytes removed.
Space$(*len*)	Returns a string containing *len* space characters (ASCII 32).
Str$(*val*)	Returns a string containing the decimal representation of *val*. If *val* is of type **Single**, its value is given to 7 significant figures.
String$(*len*, *char*)	Returns a string consisting of *len* characters with ASCII value *char*. If *char* is itself a string, then the returned string consists of *len* copies of the first character of *char*.
Trim$(*string*)	Returns *string* with leading and trailing spaces and NULL bytes removed.
UCase$(*string*)	Returns *string* with all lower-case letters converted to upper-case.
Val&(*string*[, *len*])	Returns the decimal number represented by *string*, as a **Long** value. If *len* is present, it must be a variable (not an array element). This variable is set to the number of characters used.
Val!(*string*[, *len*])	Returns the decimal number represented by *string*, as a **Single** value. If *len* is present, it must be a variable (not an array element). This variable is set to the number of characters used.
ValH(*string*[, *len*])	Returns the hexadecimal number represented by *string*, as a **Long** value. If *len* is present, it must be a variable (not an array element). This variable is set to the number of characters used.

3.17.4 Encryption Functions

Key(*keynum*)	Returns key number *keynum* as a string. If no such key exists, a zero-length string is returned. This function may also appear on the left of an assignment statement: **Key(*keynum*) = *string*** In the MultiApplication BasicCard, this function is not available; keys can only be accessed via **COMPONENT** System Library procedures. In the Terminal program, **Key** is a pre-defined, **Static** array of strings: **Key(0 To 255) As String**. In the Enhanced and Professional BasicCards, only keys declared in **Declare Key** statements can be accessed, and the length of each key is fixed; see **3.18.3 Key Declaration** for details.
DES(*type*, *key*, *block$*)	Performs a single DES block encryption or decryption operation, returning the result as an 8-byte string. *key* is either a key number from 0 to 255, or a string containing a cryptographic key. *block$* is a string at least 8 bytes long. See **3.18.6 DES Encryption Primitives** for more information.
Certificate(*key*, *data*)	Returns a **DES**-based cryptographic certificate of *data*, as an 8-byte string. *key* is either a key number from 0 to 255, or a string containing a cryptographic key. See **3.18.7 Certificate Generation** for more information.

3.17.5 Other Functions

Len(*variable***)**	Returns the size, in bytes, of a scalar variable (arrays are not allowed).
Len(*type***)**	Returns the size of a data type (e.g. **Integer**, or a user-defined type).

3.18 Encryption

The BasicCard contains a sophisticated mechanism for the encryption and decryption of commands and responses. For full details of the algorithms, see **Chapter 9: Encryption Algorithms**.

3.18.1 Implementing Encryption in the MultiApplication BasicCard

Encryption and key handling are necessarily more complex processes in a MultiApplication environment than in a single-application environment. See **Chapter 5: The MultiApplication BasicCard** for more information.

3.18.2 Implementing Encryption in a Single-Application BasicCard

To implement the encryption mechanism for single-application BasicCard commands:

1. Use the **KeyGen** program to generate a key file, containing cryptographic keys.
2. Include the generated key file in both the Terminal program and the BasicCard program.
3. Include the file **Commands.def** in the Terminal program, to define the **StartEncryption**, **ProEncryption**, and **EndEncryption** commands.
4. In the Terminal program, turn automatic encryption on and off as follows:

   Enhanced BasicCard:

   > **Call StartEncryption (P1=***algorithm***, P2=***keynum*, **Rnd)**
   > **Call EndEncryption()**

   Professional BasicCard:

   > **Call ProEncryption (P1=***algorithm***, P2=***keynum*, **Rnd**, **Rnd)**
   > **Call EndEncryption()**

   Or, if you don't know the card type in advance:

   > **Call AutoEncryption (***keynum*, *keyname$***)**
   > **Call EndEncryption()**

   The **AutoEncryption** subroutine is defined in **Commands.def**. The algorithm is selected according to the key length and the card type. The *keyname$* parameter is the pathname of the key, and is only required for the MultiApplication BasicCard. For use with single-application BasicCards, this parameter can be empty:

   > **Call AutoEncryption (***keynum*, "")

That's all you have to do. An example program is given in **9.10 .**

The program running in the BasicCard will usually want to know whether encryption is currently in force. It can check this through the pre-defined variables **Algorithm** and **KeyNumber**, which contain the two parameters **P1** and **P2** that were passed in the most recent **StartEncryption** command. If encryption is not in force, both these variables have the value zero.

3.18.3 Key Declaration

In a Terminal program or a single-application BasicCard program, the **Declare Key** statement declares a cryptographic key (the **KeyGen** program outputs its keys as **Declare Key** statements in the key file):

Declare Key *keynum* [**(***length* [**,** *counter*]**)**] [= *b1*, *b2*, *b3*, . . .]

keynum — The key number, by which the key can be specified (for example, in a **StartEncryption** command). It can take any value from 0 to 255, except in Enhanced BasicCard programs, where 255 is not allowed.

length — The length of the key. If absent, the key length defaults to 8 bytes. If an initial value field (*b1*, *b2*, *b3*, . . .) is present, and no length is specified, the key length is set to the number of bytes in the initial value field. (If the length is specified, the initial value field is padded with zeroes to the required length.)

counter — The error counter for the key ($0 \le$ *counter* ≤ 15). If *counter* is zero, the key is initially disabled. If *counter* is absent, the error counter for the key is initially inactive. See **3.18.5 Key Error Counter** for details.

Note: the *counter* parameter is allowed in all programs, but it is ignored in Terminal programs. This allows the same key file to be used in all programs in an application.

b1, *b2*, *b3*, . . . — The initial value of the key. If no initial value is provided, the key is initialised to zeroes. The key may be changed later, in one of three ways:

- with **Key(***keynum***) =** *string* (see **3.17.4 Encryption Functions**);
- with the **Read Key File** statement in a Terminal program (see **3.18.4 Run-Time Key Configuration**);
- with the **BCKeys** program in an Enhanced BasicCard (see **6.9.5 The Key Loader BCKeys.**).

3.18.4 Run-Time Key Configuration

The Terminal program can load keys from a key file at run-time, with the statement

Read Key File *filename*

If this command fails, the File System variable **FileError** contains a non-zero error code indicating the reason for the failure – see **4.12 The Definition File FILEIO.DEF** for a list of error codes.

Except in MultiApplication BasicCard programs, keys can also be accessed as strings via the **Key(***keynum***)** function. See **3.17.4 Encryption Functions** for details.

3.18.5 Key Error Counter

In a BasicCard, each cryptographic key has an error counter. If the error counter for a particular key is active, it limits the number of times that a Terminal program can attempt to guess the key. For example, suppose the error counter for key *keynum* has an initial value of 10. Whenever the BasicCard receives a command that is encrypted with key *keynum*:

- if the encryption is invalid, the error counter is decremented, and the BasicCard returns the status code **SW1-SW2 = swRetriesRemaining+*X*** (&H63C0+*X*), where *X* is the new value of the error counter. When the error counter reaches zero the key is disabled, until an **Enable Key** command is executed in the BasicCard program (see below);
- if the encryption is valid, the error counter is reset to its initial value (in this case, 10);
- if the key is disabled (i.e. the error counter is already zero), the BasicCard responds with status code **SW1-SW2 = swKeyDisabled** (&H6614).

So the Terminal program is given 10 chances, after which no more commands encrypted with key *keynum* are accepted.

In a single-application BasicCard, there are two commands for setting a key's error counter:

Enable Key *keynum* [**(***counter***)**]

Enables the key. If *counter* is present, the error counter for the key is activated, and its initial value is set to **Max** (*counter*, 15). If *counter* is absent, or equal to 255, the error counter is deactivated (i.e. the key will remain enabled regardless of how many times a command is badly encrypted with the key).

Disable Key *keynum*

Disables the key, until a subsequent **Enable Key** command is executed.

Notes:

1. This error counter mechanism only applies to the encryption of commands. Even if a key is disabled, it can always be used from within a single-application BasicCard program. ZC-Basic functions that use cryptographic keys are listed in **3.17.4 Encryption Functions**.
2. In a MultiApplication BasicCard program, the **WriteComponentAttr** System Library procedure is used to enable and disable keys.

3.18.6 DES Encryption Primitives

DES message encryption and decryption is based on the six block encryption primitives $\mathbf{E_K}$, $\mathbf{D_K}$, $\mathbf{EDE2_K}$, $\mathbf{DED2_K}$, $\mathbf{EDE3_K}$, and $\mathbf{DED3_K}$, as defined in **9.1 The DES Algorithm**. In Terminal programs, and all BasicCards with **DES** support, these primitives are available to the ZC-Basic programmer via the **DES** function:

result$ **= DES(***type***,** *key***,** *block$***)**

type The type of primitive, as follows:

+1 or **+56:**	$\mathbf{E_K}$**(***block***)**	**Single DES** encryption (8-byte key required)
–1 or **–56:**	$\mathbf{D_K}$**(***block***)**	**Single DES** decryption (8-byte key required)
+3 or **+112:**	$\mathbf{EDE2_K}$**(***block***)**	**Triple DES-EDE2** encryption (16-byte key required)
–3 or **–112:**	$\mathbf{DED2_K}$**(***block***)**	**Triple DES-EDE2** decryption (16-byte key required)
+168:	$\mathbf{EDE3_K}$**(***block***)**	**Triple DES-EDE3** encryption (24-byte key required)
–168:	$\mathbf{DED3_K}$**(***block***)**	**Triple DES-EDE3** decryption (24-byte key required)

Cards that support the **Triple Des-EDE3** algorithm (currently, Professional BasicCards **ZC5.x** and **ZC7.x**, and MultiApplication BasicCard **ZC6.5**) accept all values; other cards accept only ±1 and ±3. (The values 56, 112, and 168 denote the number of significant bits in the key.)

key Either a key number from 0 to 255, or a string containing a cryptographic key.

block$ A string containing, as its first 8 bytes, the block to encrypt or decrypt. If shorter than 8 bytes, P-Code error **pcBadStringCall** (&H0D) is generated.

result$ The 8-byte result of the DES encryption or decryption function.

3.18.7 Certificate Generation

The Terminal program, and all BasicCards with **DES** support, can generate "digital certificates" using cryptographic keys. A digital certificate is an electronic verification of a piece of data. Suppose you have a network of dealers, who can unload cash credits from the cards that you issue to your customers, in return for goods and services that they provide. At the end of the week, they come to you to exchange these electronic cash credits for real money. How can you be sure that the dealers are honest?

Digital certificates are the answer. To unload credits from a customer's card, the dealer sends a message saying "I am dealer number *A*, and I want *B* credits". The customer's BasicCard will have its own ID number *C*, and it can maintain a transaction counter *D*, which it increments after each transaction. The BasicCard program puts these four numbers *A*, *B*, *C*, and *D* together into a string or a user-defined variable, and generates a certificate using a secret key not known to the dealer or the customer. This certificate is then returned to the dealer, who shows it to you to claim reimbursement for the credits. You can write a Terminal program to check that *A*, *B*, *C*, and *D* really do generate the correct certificate with the secret key. And because the key is known only to you and the BasicCard, you know that the dealer hasn't forged the certificate.

To generate a certificate:

S$ = **Certificate(***key***,** *data***)**

where *key* is a key number from 0 to 255 or a string containing a cryptographic key, and *data* is the data to be verified – either an expression of type **String**, or a fixed-length variable or array element. Depending on the key length, this generates a **Triple DES-EDE3** certificate (24-byte key; cards **ZC5.x**, **ZC6.5**, and **ZC7.x**), or a **Triple DES-EDE2** certificate (16-byte key), or a **Single DES** certificate (8-byte key). The result, *S$*, is always 8 bytes long. The certificate generation algorithm is described in **9.3 Certificate Generation Using DES**.

3.19 Random Number Generation

The **Rnd** built-in function returns a 4-byte random number. The Terminal and the various BasicCards have different mechanisms for random number generation.

3.19.1 The Terminal

The Terminal program initialises its random number generator with a seed based on the system clock. This ensures that the **Rnd** function returns a different sequence every time a program runs. You can override this behaviour with the **Randomize** command:

Randomize *seed*

where *seed* is any expression of type **Long** or **String**.

You might want to do this for the following reasons:

- to generate a predictable sequence of random numbers while developing a program, to make debugging easier;
- to use a more unpredictable seed than the system clock, for better security.

Note: The default behaviour of the random number generator is good enough for the encryption algorithms used in communication with the BasicCard – these algorithms don't depend critically on the unpredictability of the initial values **RA** and **RB** (see **8.9.11 The START ENCRYPTION Command** for details). However, they do depend critically on the secrecy of the keys used, and for this purpose we provide a high-quality random number generation mechanism in the **KeyGen** program (see **6.9.4 The Key Generator KeyGen.**).

3.19.2 The Enhanced BasicCard

Each Enhanced BasicCard has a unique serial number burnt into its memory. The first time in its life that the BasicCard generates a random number, this serial number is used as the seed. The seed is then updated and stored in EEPROM for the next random number generation. This ensures that:

- each BasicCard generates a different sequence of random numbers;
- a given BasicCard doesn't generate the same sequence each time it is reset.

The **Randomize** command is not available in the Enhanced BasicCard.

Note: The BasicCard simulators in the **ZCMSim** and **ZCMDCard** programs do generate the same sequence of random numbers each time they run. This is because they have no access to a unique serial number to seed the generation mechanism. But when the program is downloaded to a genuine BasicCard, the random number sequence will become unpredictable.

3.19.3 The Professional and MultiApplication BasicCards

These cards have a hardware random number generator, so **Rnd** returns a truly random number.

3.20 Error Handling

If the P-Code interpreter in the BasicCard detects a run-time error, such as arithmetic overflow or insufficient memory, it calls the **ErrorHandler** procedure. If there is no procedure with this name in the program, it exits with the status code **SW1 = sw1PCodeError** (&H64). **SW2** contains the P-Code error code (see **8.8.2 BasicCard P-Code Interpreter** for a list of these error codes). The **ErrorHandler** procedure may perform clean-up operations, but it cannot cause execution to be resumed at the statement that caused the error. The pre-defined variable **PCodeError** contains the P-Code error code.

The address of the instruction where the error occurred is passed to the **ErrorHandler** procedure as an **Integer** parameter, so you can access it by declaring e.g.

Sub ErrorHandler (PC As Integer)

3.21 BasicCard-Specific Features

3.21.1 Customised ATR

When the BasicCard is reset, it provides information about itself by means of the **ATR** (Answer To Reset). The ATR contains technical information about the communication parameters that the card uses, followed by up to fifteen bytes (the 'Historical Characters') by which the card can identify itself. For example, the Historical Characters in the Enhanced BasicCard are of the form "**BasicCard ZC***vvv*", where *vvv* is the firmware version number of the card. See **8.2 Answer To Reset** for more information on the ATR.

In a single-application BasicCard program, or a **ZC8**-series MultiApplication BasicCard, you can override the card's built-in ATR with the following pre-processor directive:

#Pragma ATR (*ATR-Spec***)**

To override the default ATR in a **ZC6**-series MultiApplication BasicCard, create a file in the Root Directory with the name "**ATR**" (see **5.4.1 ATR File**), and initialise the file contents with a statement of the form:

ATR (*ATR-Spec***)**

In both cases, *ATR-Spec* is a comma-separated list of communication parameters, some of which take values:

param [= *val*] [, *param* [= *val*] , ...]

The following parameters are supported; for the meanings of these parameters, see **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*:

General Parameters

Direct or **Inverse**	Character coding convention
T=0 and/or **T=1**	The protocols supported by the card
T=15	Forces **T=15** to change the way the extra guard time is calculated
HB = *string*	Historical Bytes

Global Interface Parameters

FI = *val* or **F** = *val*	Clock rate conversion factor
DI = *val* or **D** = *val*	Baud rate adjustment factor
N = *val*	Extra guard time
TA2 = *val*	Specific mode byte
XI = *val*	Clock stop indicator
UI = *val*	Class indicator
GI = *val* or **G** = *val*	Clock factor

T=0 *Parameters*

WI = *val*	Work waiting time in tenths of a second

T=1 *Parameters*

IFSC = *n*	Information field size for the card
CWI = *val* or **CWT** = *val*	Character waiting time
BWI = *val* or **BWT** = *val*	Block waiting time
CRC or **LRC**	Error detection code

Most of these parameters affect only the content of the ATR – they are ignored by the card itself. The exceptions are **Inverse**, which at the time of writing is supported by BasicCards **ZC5.4**, **ZC5.5**, **ZC6.5**, and the **ZC7-** and **ZC8-** series; and **T=0**/**T=1**, which are supported by all Professional and MultiApplication BasicCards.

Alternatively, you can specify the ATR as a sequence of bytes, with the statement

Declare Binary ATR = *data*

Here *data* must have a total length ≤ 31. Use this feature with care, as an invalid ATR can make the card unusable. You should at the very least try out the ATR in a simulated BasicCard before testing it in a real card.

Certain cards expect a flag byte as the last byte (which doesn't count towards the 31-byte length restriction). Examples of valid ATR's can be found in the file **BasicCardV8\Inc\ATRList.def**, supplied with the distribution kit. Unless you know exactly what you are doing, you should only use this statement with data supplied by ZeitControl.

3.21.2 Customised ATS

When a **ZC7-** or **ZC8**-series BasicCard is activated by a contactless reader, it responds with an **ATS** (Answer To Select). The ATS is similar to the ATR, but it contains less information. You can override the card's built-in ATS with the following pre-processor directive:

#Pragma ATS (*param*=*val*, *param*=*val*, ...**)**

param	*val*
FSC	Frame Size for Card (16, 24, 32, 40, 48, 64, 96, 128, or 256)
FSCI	Encodes **FSC** (0 ≤ *val* ≤ 8)
TA1	Encodes the transmission speeds supported by the card
FWI	Frame Waiting time Integer
SFGI	Start-up Frame Guard Time
TC1	2 if card supports CID, 0 if not (default 2)
HB	Historical Bytes (a **String** constant)

Refer to the Internationl Standard **ISO/IEC 14443-4**: *Transmission protocol* for details.

3.21.3 Application ID

The BasicCard has a pre-defined command **GET APPLICATION ID** (see **8.9.10 The GET APPLICATION ID Command**). You can use this command to check that the BasicCard in the card reader contains your application. To configure an Application ID:

Declare ApplicationID = *data*

data Any sequence of **Byte** and **String** constants, with a total length ≤ 127.

3.21.4 Enabling and Disabling Encryption Algorithms

In a single-application BasicCard, you can enable or disable individual encryption algorithms:

{**Enable | Disable**} **Encryption** [*AlgorithmID* [, *AlgorithmID*, . . .]]

AlgorithmID The ID of an encryption algorithm. If no algorithm is specified, all available algorithms are enabled or disabled. The following algorithms (defined in Commands.def) can be enabled or disabled:

Enhanced BasicCard

AlgSingleDes	**&H21**	**Single DES**
AlgTripleDes	**&H22**	**Triple DES**

Professional BasicCard

AlgSingleDesCrc	**&H23**	**Single DES** with CRC-32
AlgTripleDesEDE2Crc	**&H24**	**Triple DES-EDE2** with CRC-32
AlgTripleDesEDE3Crc	**&H25**	**Triple DES-EDE3** with CRC-32
AlgAes128	**&H31**	**AES** with 128-bit key
AlgAes192	**&H32**	**AES** with 192-bit key
AlgAes256	**&H33**	**AES** with 256-bit key
AlgEaxAes128	**&H41**	**EAX** with **AES-128**
AlgEaxAes192	**&H42**	**EAX** with **AES-192**
AlgEaxAes256	**&H43**	**EAX** with **AES-256**
AlgOmacAes128	**&H81**	**OMAC** with **AES-128**
AlgOmacAes192	**&H82**	**OMAC** with **AES-192**
AlgOmacAes256	**&H83**	**OMAC** with **AES-256**

For maximum security, you should disable any encryption algorithms that you don't plan to use.
Notes:

- This directive is executed when the program is compiled, and it lasts for the lifetime of the card. Algorithms can't be enabled or disabled at run-time.
- Different Professional BasicCards support different combinations of the twelve algorithms listed above.

3.21.5 Asking the Terminal for More Time

The BasicCard has a **BWT** (Block Waiting Time) of 12.8 seconds – see **8.4 The T=1 Protocol** for more information. If a command is going to take longer than this to complete, it must request more time, otherwise the caller will time out. It does this with a **WTX** (Waiting Time Extension) statement:

WTX *BWT-units*

BWT-units Any expression of type **Byte**: the number of multiples of **BWT** requested. **WTX** requests are not cumulative – each request cancels all previous requests.

Note: Some card readers treat 255 as a special value. If in doubt, don't use this value – use 254 instead.

In the **T=0** protocol, the *BWT-units* parameter is ignored, and a single **NULL** byte (**&H60**) is sent. This resets the **WWT** (Work Waiting Time) time-out period – see **8.3 The T=0 Protocol** for more information.

The contactless **T=CL** protocol requires 1 ≤ *BWT-units* ≤ 59; the card accepts values outside this range, truncating them automatically.

3.21.6 Pre-Defined Variables

The BasicCard operating system has several internal variables that can be accessed from the ZC-Basic language. The following are all **Public** variables (in RAM), of type **Byte** (but **Lc** and **Le** are of type **Integer** in the **ZC7**-series Professional BasicCard):

CLA	Class byte – first byte of two-byte **CLA INS** command identifier.
INS	Instruction byte – second byte of two-byte **CLA INS** command identifier.
P1	Parameter 1 of 4-byte **CLA INS P1 P2** command header.
P2	Parameter 2 of 4-byte **CLA INS P1 P2** command header.
Lc	Length of **IDATA** field in command.
Le	Expected length of **ODATA** field in response (supplied by caller).
ResponseLength	Actual length of **ODATA** field in response (supplied by called command).
SW1	First status byte in response field **SW1-SW2**.
SW2	Second status byte in response field **SW1-SW2**.

Algorithm	ID of currently active encryption algorithm. Commands can check this byte to ascertain whether an appropriate encryption mechanism is in force. If no encryption is currently active, **Algorithm** is zero. See **3.21.4 Enabling and Disabling Encryption Algorithms** for a list of algorithm IDs.
KeyNumber	(Single-application BasicCards only) The number of the cryptographic key being used by the currently active encryption algorithm. If no encryption is currently active, **KeyNumber** is zero (but zero is also a valid key number, so you should not use **KeyNumber** to check whether encryption is active – use **Algorithm** for this purpose).
PCodeError	If a run-time error occurs, and the program contains a subroutine with the name **ErrorHandler**, then this subroutine is called. The error code is available to the **ErrorHandler** subroutine in the variable **PCodeError**.
FileError	The most recent error code generated by the file system (Enhanced and Professional BasicCards only).

The following **Integer** variables are defined:

P1P2	Concatenation of **P1** and **P2**.
SW1SW2	Concatenation of **SW1** and **SW2**.
LibError	The most recent library procedure error (only the Professional and MultiApplication BasicCards pre-define this variable – an Enhanced BasicCard program declares it in the *library*.**def** file).
SMKeyCID	(MultiApplication BasicCard only) The Component ID of the Key being used by the currently active encryption/authentication algorithm as a result of a **START ENCRYPTION** command. If none is currently active, **SMKeyCID** is zero.
ExtAuthKeyCID	(MultiApplication BasicCard only) The Component ID of the Key used in the most recent **EXTERNAL AUTHENTICATE** command, if successful.
VerifyKeyCID	(MultiApplication BasicCard only) The Component ID of the Key used in the most recent **VERIFY** command, if successful.

3.22 Terminal-Specific Features

3.22.1 Screen Output

Screen output uses the **Cls** and **Print** statements in conjunction with the four pre-defined variables **FgCol**, **BgCol**, **CursorX**, and **CursorY** (see **3.22.9 Pre-Defined Variables**).

The **Cls** command clears the screen, and sets **CursorX** and **CursorY** to 1:

Cls

The **Print** statement:

Print [*field* | *separator*] [*field* | *separator*] **. . .**

field	Any **Byte**, **Integer**, **Long**, **Single**, or **String** expression	
separator	‘**;**’ (semi-colon)	Leaves the output column unchanged.
	‘**,**’ (comma)	Advances the output column to the next output field (an output field is 14 characters wide).
	Spc(*n***)**	Prints *n* space characters.
	Tab(*n***)**	Advances the output column to position *n.*

After the print statement, the cursor advances to the start of the next line, unless the last character is a separator. (So you can stay on the same output line by adding a semi-colon at the end of the command.)

3.22.2 Keyboard Input

InKey$	Returns a string containing 0, 1, or 2 bytes. • 0 bytes: no character is waiting in the keyboard buffer. • 1 byte: a regular ASCII key was pressed. • 2 bytes: an extended-ASCII key was pressed. In this case, the first byte indicates which auxiliary keys were down (**&H01=Shift**, **&H02=Ctrl**, **&H04=Alt**), and the second byte contains the extended-ASCII code.
Line Input *X$*	Reads a line from the keyboard into the string variable *X$*, until the carriage return key is pressed. Extended-ASCII keys are ignored.
Input *variable-list*	Reads the variables in the list from the keyboard. If the list contains more than one variable, the user must separate the values with commas or spaces. This statement can also appear on the right-hand side of an assignment statement: *n* = **Input** *variable-list* This returns the number of variables in the list that were successfully input.

3.22.3 Communications

Four functions are provided for determining the status of the card reader and card. These functions return a status code in **SW1–SW2**, just like command calls:

CardReader [(*name$*)]

Attempts to detect a card reader via the configured serial port. If a string parameter is passed, the identification string of the card reader is returned.

Status Codes in SW1-SW2:

swCommandOK	Card reader detected
swNoCardReader	Card reader not detected
swCardReaderError	Invalid response from card reader

CardInReader

Returns **swCommandOK** (&H9000) if a card is in the card reader.

Status Codes in SW1-SW2:

swCommandOK	Card is in card reader
swNoCardReader	Card reader not detected
swCardReaderError	Invalid response from card reader
swNoCardInReader	No card in reader

ResetCard [(*ATR$*)]

Attempts to reset the card, returning **swCommandOK** (&H9000) if the card responded with a valid Answer To Reset. If a string parameter is passed, the Historical Bytes of the Answer To Reset are returned. See also **3.21.1 Customised ATR**.

Status Codes in SW1-SW2:

swCommandOK	Valid Answer To Reset received
swNoCardReader	Card reader not detected
swCardReaderError	Invalid response from card reader
swNoCardInReader	No card in reader
swT1Error	**T=1** protocol error (see**8.4 The T=1 Protocol**)
swCardError	Invalid response from card
swCardTimedOut	Card failed to send an ATR within the prescribed time

CloseCardReader

Makes the card reader that is attached to the current **ComPort** available to other programs. No error codes are generated.

3.22.4 PC/SC Functions

Two functions provide information about the PC/SC-compatible card readers configured in the system:

nReaders = **PcscCount**

Returns the number of configured PC/SC card readers, as an **Integer**.

Status codes in SW1-SW2:

swNoPcscDriver	The PC/SC driver is not installed in the system.
swPcscError	The PC/SC driver returned an unexpected error code.

ReaderName = **PcscReader (***ReaderNum***)**

Returns the name of PC/SC card reader *ReaderNum*, as a **String**. If *ReaderNum* is zero, the name of the default PC/SC reader is returned. To access PC/SC reader number *ReaderNum*, set the pre-defined variable **ComPort** to *ReaderNum*+100.

Status codes in SW1-SW2:

swNoCardReader	*ReaderNum* is less than zero or greater than *nReaders*.
swNoPcscDriver	The PC/SC driver is not installed in the system.
swPcscError	The PC/SC driver returned an unexpected error code.

Note: To configure a default PC/SC reader, add the reader's name to the Windows® system registry, in the field "HKEY_CURRENT_USER\Software\ZeitControl\BCPCSC\Default" (you can do this with the Windows® system tool Regedit.Exe). If no such field is found, reader number 1 is the default.

3.22.5 I/O Logging

The **Open Log File** statement initiates the logging of all I/O between the Terminal program and the BasicCard program:

Open Log File *filename*

Previous contents of the log file are destroyed. If the file open fails, the pre-defined variable **FileError** is set to a non-zero value – see **4.12 The Definition File FILEIO.DEF** for error codes. The statement

Close Log File

Ends I/O logging and closes the log file.

3.22.6 Date and Time

The string function **Time$** returns a 24-character string containing the current date and time in fixed format:

`"Ddd Mmm DD HH:MM:SS YYYY"` (for example: `"Wed Jul 07 15:50:35 2004"`).

3.22.7 Saving Eeprom Data

The statement

Write Eeprom [**(** *filename*)]

writes the permanent **Eeprom** data in the Terminal program to a disk file. If *filename* is not given, the data is written back to the original image file (or debug file). If the file couldn't be opened for any reason, the pre-defined variable **FileError** is set to a non-zero value – see **4.12 The Definition File FILEIO.DEF** for a list of error codes.

Note: The **Write Eeprom** statement is only valid if the Terminal program is running in the **ZCMSim** P-Code interpreter or the **ZCMDTerm** Terminal Program debugger. Programs containing **Write Eeprom** statements can't be compiled into executable files.

3.22.8 Automatic Encryption

{ **Enable | Disable** } **Encryption**

The P-Code interpreter that runs the Terminal program monitors all commands to the BasicCard, watching for **START ENCRYPTION** and **END ENCRYPTION** commands. If it sees a well-formed **START ENCRYPTION** command that receives a valid response from the BasicCard, it automatically turns on encryption of commands and decryption of responses, until it sees an **END ENCRYPTION** command. If for any reason you want to disable this monitor, you can do it with a **Disable Encryption** command. You can turn the monitor back on at any time with **Enable Encryption**.

3.22.9 Pre-Defined Variables

The Terminal P-Code interpreter contains the following **Public** pre-defined variables, of type **Byte**:

ComPort The number of the COM port that the card reader is attached to. To specify PC/SC card reader number *n*, set **ComPort =** ***n***+100 (or **ComPort =** 100 for the default PC/SC reader – see **3.22.4 PC/SC Functions** for details).

Note: The value of **ComPort** at program start-up is taken from the environment variable **ZCPORT**, if it exists; otherwise the Windows® Registry variable **ZCPORT** in the directory **HKEY_CURRENT_USER\Software\ZeitControl\BasicCardV8**, if it exists; otherwise it takes the value 1.

ResponseLength The length of the **ODATA** field in the last response received from the card.

SW1 First byte of **SW1-SW2** status field in the last response received from the card.

SW2 Second byte of **SW1-SW2** status field in the last response received from the card.

Algorithm ID of currently active encryption algorithm. Commands can check this byte to ascertain whether the appropriate encryption mechanism is in force. If no encryption is currently active, **Algorithm** is zero. See **3.21.4 Enabling and Disabling Encryption Algorithms** for a list of algorithm IDs.

PCodeError If a run-time error occurs, and the program contains a subroutine with the name **ErrorHandler**, then this subroutine is called. The error code is available to the **ErrorHandler** subroutine in the variable **PCodeError**.

FgCol Foreground colour for **Print** statements to the screen (0-15).

BgCol Background colour for **Print** statements to the screen (0-15).

CursorX X-coordinate of text cursor (1-80).

CursorY Y-coordinate of text cursor (1-25).

FileError The most recent error code generated by a file I/O operation.

nParams Number of command-line parameters (see **6.9.2 The P-Code Interpreter ZCMSim.exe**).

Two **Integer** variables are defined:

KeyNumber The number (for a single-application BasicCard) or the Component ID (for a MultiApplication BasicCard) of the cryptographic key being used by the currently active encryption algorithm. If no encryption is currently active, **KeyNumber** is zero (but zero is also a valid key number, so you should not use **KeyNumber** to check whether encryption is active – use **Algorithm** for this purpose).

SW1SW2 Concatenation of **SW1** and **SW2**.

Two **String** arrays are defined:

Param$(1 To nParams) Command-line parameters passed to the **ZCDOS** program (see **6.9.2 The P-Code Interpreter ZCMSim.exe**).

Key(0 To 255) Cryptographic keys.

3.23 Miscellaneous Features

This section lists all the ZC-Basic statements that are not covered in the preceding sections or in **Chapter 4: Files and Directories**.

3.23.1 Overflow Checking

{ **Enable** | **Disable** } **OverflowCheck**

Normally, if the result of an arithmetic operation is too big or too small to be represented in the target type, a P-Code error is generated. You can enable or disable this overflow checking with **Enable OverflowCheck** or **Disable OverflowCheck**. These statements are executed at run-time, and don't apply to the whole program. To disable overflow checking for the whole program, put **Disable OverflowCheck** in your initialisation code.

Note: This statement only affects whole-number arithmetic (**Byte**, **Integer**, **Long**, and **Long64** data types). Floating-point overflow checking (**Single** and **Double** data types) cannot be turned off.

3.23.2 DefType Statement

A **DefType** statement specifies the default type of variables, arrays, and functions that begin with a certain letter or range of letters:

{ **DefByte** | **DefInt** | **DefLng** | **DefLng64** | **DefSng** | **DefDbl** | **DefString** } *range* [**,** *range***, . . .**]

range Either a single letter, or a range of letters separated by a minus sign (e.g. **I–N**). The case of the letter(s) is not significant.

The initial setting is **DefInt A–Z**, i.e. all variables, arrays, and functions have type **Integer** by default.

3.23.3 Array Subscript Base

An array subscript range takes the form

[*lower-bound* **To**] *upper-bound*

If the optional *lower-bound* is missing, it defaults to **0**. You can change this default value with the **Option Base** command, which applies to all subsequent array declarations:

Option Base *subscript-base*

subscript-base Any constant expression. In the Enhanced BasicCard, it must satisfy

$-32 \le \textit{subscript-base} \le +31$

Or you can specify that the lower bounds of array subscripts must always be explicitly declared, with

Option Base Explicit

3.23.4 Explicit Declaration of Variables and Arrays

By default, ZC-Basic allows implicit declaration of variables and arrays:

- If it meets a variable that it doesn't recognise in an expression or an assignment statement, it will treat it as a newly-declared variable. The type of the variable is determined from its name, as described in **3.7 Data Declaration**.
- If a **ReDim** statement contains an unrecognised array name, the compiler inserts an implicit **Dim** statement to declare the array.

The Basic programming language has always behaved this way. However, this can be dangerous, as it accepts mis-typed variable names as new variables. In the following example, this results in **TransactionState** ending with the value **1** instead of **13**:

```
TransactionState = 12
...
TransactionState = TransatcionState + 1
```

The compiler will issue a warning message whenever it implicitly declares a variable in this way. You can override this behaviour in two ways:

Option Explicit

This tells the compiler not to accept variables or array names that haven't been explicitly declared. It applies only to following code; preceding code can contain implicit declarations.

Option Implicit

This tells the compiler to accept implicitly-declared variables without issuing a warning message.

3.24 Technical Notes

3.24.1 Parameter Size Limits

In the **ZC7-** and **ZC8-**series BasicCards, the variables in a parameter list can have a total size of up to 32767 bytes – in other words, no practical limit. In earlier cards, the maximum total size of all the parameters in a procedure call is approximately 128 bytes. More precisely, the compiler checks that the sum of the following contributions is ≤ 128:

- the total size of all the fixed-length parameters (including **String*n**);
- 2 bytes for each parameter of array type;
- 3 or 4 bytes for each **String** parameter, depending on whether strings longer than 254 bytes are supported (or 2 bytes for the final **String** parameter to a **Command**);
- for a **Function**, the size of the return value (2 bytes if this is a **String**);
- 2 bytes for the return address (unless it's a **Command**);
- the frame overhead (2 bytes for the Enhanced BasicCard, otherwise 4 bytes).

3.24.2 Array Descriptor Format

An array in ZC-Basic consists of a fixed-length *array descriptor*, and a *data area* (which is of variable length if the array is **Dynamic**). The array descriptor contains the following fields:

- The address of the data area (0 if not allocated). This field is 2, 3, or 4 bytes.
- The size of each element. This field is 1 or 2 bytes.
- The number of dimensions, as a **Byte**. The top bit, if set, indicates a Dynamic array.
- For each dimension, a range: either (inclusive) lower and upper bounds (**Long** in a Terminal program, **Integer** in the Professional and MultiApplication BasicCards); or a 16-bit packed range in the Enhanced BasicCard:

LO(n) (6 bits, –32 to +31)	**RANGE(n)** (10 bits, 0-1023)

  In this case, the upper bound of subscript(*i*) is equal to **LO(*i*)** + **RANGE(*i*)**.

In Terminal programs, and Professional and MultiApplication BasicCard programs, **LO(*i*)** and **HI(*i*)** are 2-byte integers, so the descriptor occupies **4*n + 4** bytes.

3.24.3 String Parameter Format

A variable of type **String** is a pointer to a (*len*, *data*) pair:

address (2, 3, or 4 bytes)	*len* (1 or 2 bytes)	*data* (*len* bytes)

Addresses are 4 bytes in a Terminal program; 3 bytes in a **ZC7-** or **ZC8-**series BasicCard; and 2 bytes in other cards. In a Terminal program, or a **ZC7**- or **ZC8**-series BasicCard, strings can be longer than 254 bytes, so *len* is 2 bytes. Otherwise *len* is 1 byte.

A variable of type **String*n** requires just **n** bytes of storage:

data (**n** bytes)

A procedure parameter of type **String*n** also takes up **n** bytes on the P-Code stack.

However, a procedure parameter of type **String** is rather more complicated. Two requirements must be fulfilled:

- A procedure can change the value of a **String** variable passed as a parameter;
- A **String*n** variable can be passed as a **String** parameter.

So a **String** parameter contains a length field of 1 or 2 bytes, and a 2-, 3-, or 4-byte address field, taking up 3-6 bytes on the P-Code stack. If a fixed-length **String*n** variable was passed, then the length field contains the length **n** and the next two bytes contain the address of the data. Otherwise, the length field contains **&HFF** (resp. **&HFFxx**) and the address field contains the address of the pointer (not the address of the data). So if the address of the data has to be changed because the string increases in length, the **String** variable can be updated to point to the new data.

3.24.4 Memory Allocation in Single-Application BasicCards

The ZC-Basic compiler calculates the sizes of all the memory regions in RAM and EEPROM. Any memory left over is assigned to the two heaps, **RAMHEAP** and **EEPHEAP**. These regions are for run-time memory allocation. (See **10.5 Run-Time Memory Allocation** for the format of the allocated memory blocks.)

The ZC-Basic P-Code interpreter uses run-time memory allocation for three kinds of data: variable-length **String** data, **Dynamic** arrays, and files. Files and **Eeprom** data are allocated as **Permanent** blocks in **EEPHEAP**. Other data is allocated in **RAMHEAP** if there is room, but if not, it is allocated as **Temporary** blocks in **EEPHEAP**. All **Temporary** blocks are freed the next time the BasicCard is reset or the Terminal program is started. EEPROM writes require up to 6 milliseconds to complete, so a BasicCard program runs more slowly when it has to use **EEPHEAP** in this way.

See **5.2.4 Memory Allocation** for information on memory allocation in the MultiApplication BasicCard.

4. Files and Directories

4.1 Directory-Based File Systems

Everybody who owns a PC is familiar with directory-based file systems. Each disk drive has a special directory, called the *root directory*, which contains data files and sub-directories. These sub-directories themselves can contain data files and sub-directories, and so on. This determines a tree of directories, in which any directory in the tree can contain data files and sub-directories. The directory containing a given data file or sub-directory is called its *parent* directory. (*directory* is the traditional term, which is used throughout this chapter; Microsoft Windows® calls its directories *folders*.)

4.1.1 File and Directory Names

Under Windows®, filenames can be up to 255 characters long, and may contain any printable character (including the space character), except the following:

\	Backslash	/	Slash	:	Colon	*	Asterisk
?	Question mark	"	Double quote	<	Left angle-bracket	>	Right angle-bracket
		Vertical bar						

Case is not significant when referring to an already existing file or directory. So if a file has the name "FILE.NAM", you can access it as "File.Nam" or "FiLe.nAm" or whatever. However, Windows® retains the case of the characters specified when the file was originally named. So if you create a file as "File.Nam" and then ask for a directory listing, Windows® lists it as "File.Nam".

4.1.2 Path Names

Each file and directory can be uniquely identified by a *full path name*. This consists of the disk drive name, followed by every sub-directory on the path from the root directory to the parent directory, followed by the name of the file or directory itself. The disk drive name is a letter A-Z followed by a colon, e.g. "`C:`" or "`A:`". (Lower-case letters may also be used to refer to disk drives, but a drive name returned by a ZC-Basic function will always be upper-case.) The drive name is immediately followed by a backslash character (this signifies the root directory); and subsequent directory names in the path are separated by backslash characters '`\`'. For example, a full path name might be "`C:\1997 Clients\Account Data`".

To save having to give the full path name every time, every disk drive in the system has a *current directory*, and the system as a whole has a *current drive*. If the disk drive name is missing from the front of a path name, the current drive is assumed. And if the first character after the disk drive name is not a backslash, then the chain of directories is followed starting from the current directory for the drive, instead of the root directory. Such a path name is called a *relative path name*. For instance, suppose the current drive is "`C:`", and the current directories for drives "`A:`" and "`C:`" are "`\Clients.97`" and "`\Programs\CPP`" respectively. Then the relative path names "`A:August\TOTALS.DAT`" and "`Headers\SUM.H`" expand to the full path names "`A:\Clients.97\August\TOTALS.DAT`" and "`C:\Programs\CPP\Headers\SUM.H`" respectively.

The directory names "`.`" and "`..`" have special meanings: "`.`" denotes the current position in the chain of directories, and "`..`" denotes the parent directory. So "`.\`" in a path has no effect, and "`..\`" goes back to the previous directory in the chain. For instance, in the previous example, the path name "`..\Basic\FILEIO.BAS`" expands to "`C:\Programs\CPP\..\Basic\FILEIO.BAS`", which is the same as "`C:\Programs\Basic\FILEIO.BAS`". The single-dot notation is useful when a directory is required as a parameter to a file system operation; for example, the ZC-Basic statement `Name "..\FileList" As ".\"` moves the file "`FileList`" from the parent directory to the current directory.

4.2 The BasicCard File System

The BasicCard contains a directory-based file system, with the same file-naming rules as those described in the previous section for Windows® (except that the maximum length of a full path name is 254 characters). The BasicCard has one root directory, so path names don't begin with a disk drive name. With the exception of the commands **CurDrive**, **ChDrive**, and **SetAttr**, the ZC-Basic file and directory commands available to a BasicCard program are the same as those available to a Terminal program.

4.2.1 File Access from a Terminal Program

If the BasicCard allows it, files and directories in the card can be accessed from a Terminal program, just as if the card was a disk drive. The card has the special drive name "`@:`". Suppose the BasicCard contains a file "`\Transport\Bus\Credits`". Then the full path name of this file from the point of view of the Terminal program is "`@:\Transport\Bus\Credits`". And if the Terminal program sets the current drive to "`@:`" and the current directory to "`\Transport`", it can refer to the file as simply "`Bus\Credits`". The full range of file and directory commands is available to the Terminal program for accessing BasicCard files and directories, subject to appropriate access being granted.

Each file or directory in the BasicCard has its own access conditions, specifying the circumstances under which the Terminal program is allowed read and write access.

If the card is a MultiApplication BasicCard, the access conditions also specify which Applications are allowed read and write access. For information on the MultiApplication BasicCard file security mechanism, see **5.1.3 Component Access Control**.

In a single-application BasicCard, these access conditions can be set and changed with **Lock** and **Unlock** statements. There are three types of access condition: **Read**, **Write**, and **Custom**. The following general rules apply to file and directory access in a single-application BasicCard:

- **Read** and **Write** access to all files and directories is available to the BasicCard program at all times.
- **Read** and **Write** access to all files and directories is available to the Terminal program as long as the BasicCard is in state **LOAD** or **PERS** (see **8.9.1 States of the BasicCard**).
- Otherwise, to access a file or directory from the Terminal program, **Read** access is required to all directories in the path from the root to the parent. To delete a file or directory, or to change its access conditions, **Write** access is required to the file or directory, and to its parent directory. (In particular, when the card is in state **TEST** or **RUN**, the Terminal program can never change the root directory's access conditions, because the root directory has no parent.)
- If a **Custom** lock is placed on a file or directory, it is locked against **Read** and **Write** access every time the card is reset. It can only be unlocked from within the BasicCard program, after which the file's regular **Read** and **Write** access conditions apply until the next reset. So you can write a command that unlocks a particular file if the Terminal program sends the correct PIN number, for instance.

The **Read** and **Write** access conditions on a file or directory can be:

- **Allowed** – access is allowed from the Terminal program;
- **Forbidden** – access is forbidden from the Terminal program; or
- **Keyed** – access is allowed only if encryption with the appropriate key is enabled.

Read and **Write** access conditions and key numbers can be set independently of each other. If access is **Keyed**, up to two keys can be specified – if encryption with either of the two keys is enabled, access is allowed. So to access a **Keyed** file from a Terminal program, you must first call **StartEncryption** with the appropriate algorithm and key number – see **3.18.1 Implementing Encryption**.

The encryption algorithm must be as strong as the key allows:

Key Length	*Permitted Algorithms*
8-15 bytes	**Single DES**
16-23 bytes	**AES-128, EAX-128, Triple DES**
24-31 bytes	**AES-192, EAX-192, 3-Key Triple DES**
32 bytes or longer	**AES-256, EAX-256**

Note: The default access conditions on the root directory are **Read=Allowed** and **Write=Forbidden**.

4.2.2 Pre-Defined Files and Directories

In a BasicCard program, you can pre-define directories and data files using **Dir** and **File** statements. The compiler constructs the appropriate structures in EEPROM for downloading to the card. See **4.11 File Definition Section** for details.

4.2.3 Storage Requirements

In the BasicCard, data files and directories are stored in EEPROM. To make efficient use of the limited space available, you should know how much memory is used. A data file or directory allocates space for its header and its name; a data file owns data blocks as well:

- A directory header requires 14 bytes of EEPROM; a data file header requires 20 bytes (23 bytes in the **ZC7-** and **ZC8-**series cards).
- The name of a file or directory takes up *n*+3 bytes of EEPROM, where *n* is the number of characters in the name.
- Each data block in a data file uses *n*+5 bytes of EEPROM (*n*+6 bytes in the **ZC7-** and **ZC8-**series cards), where *n* is the block length specified when the file was created. (The default block length is 32 bytes.) These blocks are allocated automatically when data is written to a file. *Note:* Contiguous data blocks are merged if they are also contiguous in EEPROM; this saves the overhead of 5 or 6 bytes per block. So if you are creating a file that is going to be written to just once, you can achieve optimum EEPROM usage by specifying a block length of 1 byte.

As well as these EEPROM requirements, the file system in the Enhanced and Professional BasicCards uses (6 * *nFiles* + 7) bytes of RAM (9 * *nFiles* + 8 bytes in the **ZC7-** and **ZC8-**series cards), where *nFiles* is the number of open file slots configured (see **3.3.7 Number of Open File Slots**).

4.3 File System Commands

This chapter describes all the file system commands available to the ZC-Basic programmer. There are three cases that the ZC-Basic *interpreter* must distinguish:

1. A Terminal program accessing the file system in the PC (disk drives "`A:`" through "`Z:`").
2. A Terminal program accessing the BasicCard file system (disk drive "`@:`").
3. A BasicCard program accessing its own BasicCard file system (no disk drive).

However, these cases all look the same to the ZC-Basic *programmer*. Apart from the disk drive names, there are no differences, unless explicitly noted in the command descriptions that follow.

After each command, its required access conditions are listed. These access conditions apply to a Terminal program (if the BasicCard is in state **TEST** or **RUN**), and to an Application running in the MultiApplication BasicCard. They don't apply to an Application running in a single-application BasicCard; such an Application has access to all files and directories.

All file system commands return a status byte in the pre-defined variable **FileError**. A zero value (**feFileOK**) indicates success. A non-zero value is an error code, and indicates the first error that occurred since this variable was last set to zero. (It is reset to zero every time a new command is received from the Terminal program; you may also set it to zero yourself if you want to continue after an error.) Error codes for each command are listed below.

As well as the error codes documented below under individual commands, there are some general error codes that apply to all commands:

feInvalidDrive	In cases 1 and 2 above (Terminal program), a disk drive name in a path was not a letter or "`@:`".
feBadFilename	A filename contains an invalid character, or is too long (see **4.1.1 File and Directory Names**).
feBadFilenum	A file number is out of range. In ZC-Basic, an open file is referred to by a file number. In a Terminal program, this number must be between 0 and 32 inclusive (with 0 indicating the screen or keyboard). In a BasicCard program, the number must be between 1 and the number of open file slots (see **3.3.7 Number of Open File Slots**).
feFileNotFound	A file or directory specified in a path name does not exist.
feFileNotOpen	The file number passed to the command is not associated with an open file. *Note:* This need not be the result of a programming error. If a Terminal program opens a file in the BasicCard, and then calls a BasicCard command, the BasicCard command can close all files unilaterally – including remotely-opened files – by using the **Close** command with no parameters. This is so that the BasicCard program can always find a free open file slot when it needs one.
feAccessDenied	The access conditions on a file or directory do not allow the execution of the command.
feBadFileChain	The file system in the BasicCard is corrupted.
feBadParameter	An invalid parameter value was passed to the command.
feOutOfMemory	The BasicCard has insufficient free EEPROM to execute the command.
feUnexpectedError	An operating system command in the PC returned an unexpected error code when a file system function was called.
feCommsError	In case 2 above (Terminal program accessing the BasicCard file system), the command failed because of a communications failure with the BasicCard. The status bytes describing the communications failure can be found in the pre-defined variables **SW1** and **SW2**.
feNoFileSystem	The card has no file system installed, either because no program has yet been downloaded to the card, or because the file system was disabled with a **#Files 0** directive (see **3.3.7 Number of Open File Slots**).

Definitions of these error codes and other constants are contained in the file **FileIO.def**. This file is supplied in the distribution kit, and is listed in **4.12 The Definition File FILEIO.DEF**.

4.4 Directory Commands

4.4.1 Creating a Directory

The **MkDir** command creates a new directory (but see also **4.11 File Definition Sections**):

MkDir *path*

path The path name of the new directory. A final backslash '\' is optional.

Access Conditions:

Write access to the parent directory is required. The **Read** and **Write** access conditions of the new directory are the same as those of the parent directory.

Error Codes:

feFileNotFound	The parent directory does not exist.
feFileAlreadyExists	A file or directory with the given path name already exists.
feNameTooLong	The full path name of the directory would be longer than 254 characters.

4.4.2 Deleting a Directory

The **RmDir** command deletes an existing directory. The directory must be empty before it can be deleted:

RmDir *path*

path The path name of the directory. A final backslash '\' is optional.

Access Conditions:

Single-application BasicCard: **Write** access is required, both to the directory and to its parent directory.
MultiApplication BasicCard: **Delete** access is required (but not to the parent directory).

Error Codes:

feFileNotFound	The directory does not exist.
feNotDirectory	The file is a data file, not a directory. Use **Kill** to delete data files.
feDirNotEmpty	The directory is not empty, and therefore can't be deleted.

4.4.3 Setting the Current Directory

The **ChDir** command sets the current directory.

ChDir *path*

path The path name of the new current directory. A final backslash '\' is optional.

Note (Terminal programs only): If the path contains a disk drive name, the current directory for that disk drive is changed, but the current disk drive is *not* changed. Use **ChDrive** to change the current disk drive.

Access Conditions:

Read access to the directory is required.

Error Codes:

feFileNotFound	The directory does not exist.
feNotDirectory	The file is a data file, not a directory.

4.4.4 Retrieving the Current Directory

The **CurDir** function returns the path of the current directory as a **String**:

S$ = **CurDir** [(*drive*)]

drive The disk drive for which the current directory is requested. The first character must be a letter ('A-Z' or 'a-z'), or the character '@'. If absent, the current directory of the current disk drive is returned.

Note: The optional *drive* parameter is accepted only in Terminal programs.

Access Conditions:

No access conditions are required for this command.

Error Codes:

feInvalidDrive	The disk drive specified in the *drive* parameter does not exist.

4.4.5 Renaming a File or Directory

The **Name** command renames a file or directory, or moves it to a new directory, or both. It cannot be used to move a file from one disk drive to another.

Name *OldPath* **As** *NewPath*

OldPath	The old path name of the file or directory.
NewPath	The new path name. If no backslash appears in *NewPath*, the file or directory is renamed without being moved. If *NewPath* ends with a backslash character '\', the file or directory is moved without being renamed.

Access Conditions:

Write access is required (i) to the file or directory being renamed, (ii) to its parent directory, and (iii) to the destination directory if different from the current parent directory.

Error Codes:

feFileNotFound	The file specified in *OldPath* does not exist, or the directory specified in *NewPath* does not exist.
feFileAlreadyExists	The file specified in *NewPath* already exists.
feNameTooLong	The operation would result in a file or directory in the BasicCard with a full path name longer than 254 bytes.
feRenameError	One of the following error conditions: • *OldPath* is the root directory, which cannot be renamed. • *NewPath* and *OldPath* are on different disk drives.
feRecursiveRename	The directory in *NewPath* is a sub-directory of *OldPath*, so the rename operation would result in an endless loop in the directory tree.

4.4.6 Searching for Files

Use the **Dir** command to search for files and directories matching a given wild-card specification. This has two forms:

nFiles = **Dir (** *filespec* **)**	Returns the number of matching files and directories, as an **Integer**.
file$ = **Dir (** *filespec*, *n* **)**	Returns the name of the *n*th matching file or directory, as a **String**.

filespec	The path name of the file(s) to search for. The last component of the path may contain the wild-card characters '`?`' (matching any single character) and '`*`' (matching any sequence of zero or more characters). For example, "`A*`" finds all filenames that start with the character '`A`' or 'a', and "`*=?`" finds all filenames whose penultimate character is '`=`'.
n	The number of the matching file, $1 \le n \le nFiles$.

Notes:

1. If *filespec* refers to a file or files in the PC, the first **Dir** command for a given *filespec* saves all the matching files in memory. This list is retained for future **Dir** commands of the second form that have the same *filespec* parameter (unless a ZC-Basic command intervenes that can change the directory contents). This is a major speed improvement in most cases. However, if another process changes the directory contents, ZC-Basic won't know about it, and will continue to use the original list. You can override this at any time and re-load the list from the disk, by calling a **Dir** command of the first form.
2. The BasicCard uses a case-insensitive matching algorithm that treats the full stop (period) character '`.`' no differently from any other character (unlike Microsoft Windows®). However, as a special case, the wild-card string "`*.*`" matches all files and directories.

Access Conditions:

Read access to the parent directory is required.

Error Codes:

feBadFilename	*filespec* is not a valid path name (this error code is also returned if *filespec* contains wild-card characters in any component except the last).
feBadFilenum	*n* is less than 1 or greater than *nFiles.*

4.4.7 Setting the Attributes of a File or Directory

The **SetAttr** command sets the attributes of a file or directory:

SetAttr *filename***,** *attributes*

filename The path name of the file or directory.

attributes A bit map of the attributes to set. The attributes available depend on the host operating system. See **4.4.8 Retrieving the Attributes of a File or Directory** for details.

Note: This command is available in Terminal programs only.

Access Conditions:

Access conditions are not relevant for **SetAttr** – a BasicCard file has no attributes that can be changed.

Error Codes:

feRemoteFile *filename* is a BasicCard file, so it has no attributes that can be changed.

4.4.8 Retrieving the Attributes of a File or Directory

The **GetAttr** command returns the attributes of a file or directory:

attributes **= GetAttr (** *filename***)**

filename The path name of the file or directory.

attributes A bit map of the attributes of the file or directory. The BasicCard file system supports two attributes:

faDirectory	Indicates that the file is a directory, and not a data file.
faCardFile	Indicates that the file or directory is in the BasicCard.

The Terminal program also supports the following attributes:

faReadOnly	Indicates a read-only file.
faHiddenFile	Indicates a hidden file.
faSystemFile	Indicates a system file.
faArchived	Indicates that file has been backed up since last changed.
faNormal	Indicates that no other attribute bits are set.
faTemporary	Indicates that file is being used for temporary storage.

These constants are defined in the file FILEIO.DEF.

Access Conditions:

Read access is required to the parent directory (but not to the file itself).

4.4.9 Setting the Current Disk Drive

The **ChDrive** command sets the current disk drive.

ChDrive *drive*

drive The disk drive for which the current directory is requested. The first character must be a letter ('A-Z' or 'a-z'), or the character '@'.

Note: This command is available in Terminal programs only.

Access Conditions:

No access conditions are required for this command.

Error Codes:

feInvalidDrive	The disk drive specified in the *drive* parameter does not exist.

4.4.10 Retrieving the Current Disk Drive

The **CurDrive** function returns the current disk drive as a single-character **String** containing an upper-case letter 'A-Z' or the character '@':

S$ = **CurDrive**

Note: This command is available in Terminal programs only.

Access Conditions:

No access conditions are required for this command.

4.5 Creating and Deleting Files

4.5.1 Creating a File

There is no special command to create a new file (but BasicCard files can be defined at compile time – see **4.11 File Definition Sections**). A file is created simply by opening a non-existent file for output, using the **Open** command (see **4.6.1 Opening a File**). A file can't be created in this way if *mode* is **Input** or *access* is **Read**.

4.5.2 Deleting a File

The **Kill** command deletes an existing file:

Kill *filename*

filename The name of the file.

Access Conditions:

Single-application BasicCards: **Write** access is required, both to the file and to its parent directory.
MultiApplication BasicCard: **Delete** access is required (but not to the parent directory).

Error Codes:

feFileNotFound	The file does not exist.
feNotDataFile	The file is a directory, not a data file. Use **RmDir** to delete directories.
feFileOpen	The file can't be deleted, because it is currently open.

4.6 Opening and Closing Files

4.6.1 Opening a File

In traditional Basic, the programmer has to specify *filenum*, the number of the open file slot. But in the BasicCard file system, with open file slots shared between the BasicCard program and the Terminal program, the programmer can't always know which file slots are in use. So ZC-Basic allows an alternative form of the **Open** command, where the operating system automatically selects a free open file slot. (This is equivalent to calling **FreeFile** to select an open file slot, followed by a traditional **Open** command.)

Traditional form:

Open *filename* [**For** *mode*] [**Access** *access*] [*lock*] **As** [#] *filenum* [**Len=***recordlen*] [**Align=***alignment*]

Alternative form:

filenum = **Open** *filename* [**For** *mode*] [**Access** *access*] [*lock*] [**Len=***recordlen*] [**Align=***alignment*]

filename The path name of the file to be opened.

mode If *mode* is **Input**, **Output**, or **Append**, the file is opened for sequential I/O, in which all write operations take place at the end of the file. If *mode* is **Binary** or **Random**, write operations can take place anywhere in the file, overwriting existing data:

Input Opens the file for sequential input.
Output Opens the file for sequential output. Existing data is destroyed.
Append Opens the file for sequential output and sets the file pointer to the end of the file. Existing data in the file is preserved.

Binary Opens the file for random access by file position, using **Get** and **Put**.
Random Opens the file for random access by record number, using **Get** and **Put**.

If the *mode* parameter is absent, its value depends on the *access* parameter: **Input** for **Access Read**, **Output** for **Access Write**, and **Append** for **Access Read Write**. If both *mode* and *access* are absent, *mode* defaults to **Input** and *access* defaults to **Read.**

access Specifies which types of operations will be executed on the file. It takes the value **Read**, **Write**, or **Read Write**.

- If *mode* is **Input**, then *access*, if present, must be **Read**.
- If *mode* is **Output**, then *access*, if present, must be **Write**.
- If *mode* is **Append**, then *access*, if present, must be **Write** or **Read Write**.
- If *mode* is **Binary** or **Random**, then *access* can take any value; it defaults to **Read Write**.

lock For a file in the PC, this parameter specifies whether the file can be opened simultaneously by other processes. For a file in the BasicCard, it specifies whether the file can be opened simultaneously from the Terminal program and the BasicCard program. It also determines whether a file can be opened simultaneously under different open file slots in the same program. The *lock* parameter can take the following values:

Shared Allows simultaneous read and write operations by other processes.
Lock Read Prevents simultaneous read operations by other processes.
Lock Write Prevents simultaneous write operations by other processes.
Lock Read Write Prevents simultaneous access by other processes (the default).

filenum The number of an open file slot, by which read and write operations will be executed. In the Terminal program, *filenum* must be between 1 and 32 inclusive. In the BasicCard program, *filenum* must be 1 or 2, unless the number of open file slots has been configured with the **#Files** directive (see **3.3.7 Number of Open File Slots**).

recordlen Record length or block length.

- If the file is being created, this parameter specifies the size of its data blocks (see **4.2.3 Storage Requirements** for more information). If absent (or zero), the data block size for the new file is 32 bytes. If present, it must be ≤ 16381.
- If *access* is **Random**, this parameter specifies the record length of the file. This record length must be between 1 and 254 inclusive.

alignment The alignment of each data block in the file (**ZC7-** and **ZC8-**series BasciCards only). An integer between 1 and 16, representing a power of 2 between 2 and 65536. This option was intriduced for the **ZC8-**series MultiApplication BasicCard, so that Application files could be aligned on 16-byte boundaries for the processor's Memory

Management Unit. The compiler applies this attribute automatically to Application files, so you should not normally need to use this option.

Access Conditions:

If the file already exists, the access conditions required depend on the *access* parameter: **Read**, **Write**, or **Read Write**. If the file is being created, **Write** access to the parent directory is required, and the **Read** and **Write** access conditions on the new file are the same as those of the parent directory.

Error Codes:

feFileNotFound	The file does not exist, and could not be created, because: • the parent directory does not exist; or • *mode* is **Input**; or • *access* is **Read**.
feNotDataFile	The file is a directory, not a data file.
feFileOpen	(Traditional form only) Open file slot number *filenum* is already in use.
feTooManyOpenFiles	(Alternative form only) There are no more free open file slots.
feTooManyCardFiles	(Terminal program only) An attempt was made to open a BasicCard file from a Terminal program, but there are no more free open file slots in the BasicCard.
feNameTooLong	(BasicCard file system only) The file can't be created, because its full path name would be longer than 254 characters.
feRecordTooLong	Either *access* is **Random**, and *recordlen* is greater than 254; or the file is being created, and *recordlen* is greater than 8191.
feBadParameter	Either *access* is **Random**, and *recordlen* is less than 1 (or absent); or the file is being created, and *recordlen* is less than 0.
feSharingViolation	The file is already open, and the required shared access is not available.
feInvalidAlignment	*alignment* is not an integer between 1 and 16.

4.6.2 Closing Files

The **Close** command closes one or more files:

Close [[#] *filenum* [**,** [#] *filenum* **, . . .**]]

Note: If no parameters are supplied, all open files are closed. (But the P-Code interpreter automatically closes all files on program exit.) If the BasicCard program closes all open files in this way, even files that were opened from the Terminal program are closed. In this way, the BasicCard program can always find a free open file slot when it needs one.

4.7 Writing To Files

4.7.1 Writing to Sequential Files

If a file was opened for writing, with a *mode* parameter equal to **Output** or **Append**, it can be written to with a **Print** or **Write** command. All write operations take place at the end of the file.

The **Print** command outputs data to a sequential file in human-readable format. It has the same format as the **Print** command for displaying data on the screen (see **3.22.1 Screen Output**), except for the initial #*filenum* parameter:

Print #*filenum***,** [*field* | *separator*] [*field* | *separator*] **. . .**

filenum	The *filenum* parameter to the **Open** command by which the file was opened.	
field	Any **Byte**, **Integer**, **Long**, **Single**, or **String** expression	
separator	'**;**' (semi-colon)	Leaves the output column unchanged.
	'**,**' (comma)	Advances the output column to the next output field (an output field is 14 characters wide).
	Spc(*n***)**	Prints *n* space characters.
	Tab(*n***)**	Advances the output column to position *n*.

A new-line character is added at the end, unless the last character is a separator. (So you can stay on the same output line by adding a semi-colon at the end of the command.)

The **Write** command writes data to a sequential file, in a binary format that is specific to ZC-Basic. If a sequence of values is written to a file with **Write** statements, then the same values can subsequently be read from the file using ZC-Basic **Input** statements (see **4.8.1 Reading from Sequential Files**).

Write [#] *filenum*, *expression-list*

filenum	The *filenum* parameter to the **Open** command by which the file was opened.
expression-list	A list of expressions separated by commas. Expressions can be of numerical, string, or user-defined type.

Access Conditions:

The file must have been opened with the *access* parameter equal to **Write** or **Read Write**.

Error Codes:

feInvalidMode	The file was not opened with *mode* equal to **Output** or **Append**.
feInvalidAccess	The file was not opened with *access* equal to **Write** or **Read Write**.

4.7.2 Writing to Binary and Random Files

The **Put** command is used to write to files that were opened with *mode* equal to **Binary** or **Random**. The write operation takes place at the current file position, overwriting any existing data at that position. After the **Put** command, the current file position advances to the next character (for **Binary** files) or the next record (for **Random** files):

Put [#] *filenum*, [*pos*], *data*

filenum	The *filenum* parameter to the **Open** command by which the file was opened.
pos	A record number for **Random** files, and a character position for **Binary** files. If *pos* is not present ("**Put** [#] *filenum*, , *data*"), the variable is written to the current file position.
data	A variable or array element, or a **String** expression.

Access Conditions:

The file must have been opened with the *access* parameter equal to **Write** or **Read Write**.

Error Codes:

feInvalidMode	The file was not opened with *mode* equal to **Binary** or **Random**.
feInvalidAccess	The file was not opened with *access* equal to **Write** or **Read Write**.
feSeekError	*pos* is an invalid file position.

4.8 Reading From Files

4.8.1 Reading from Sequential Files

If a file was opened for reading, with a *mode* parameter equal to **Input** or **Append**, it can be read with a **Line Input** statement, an **Input** function, or an **Input** statement.

Line Input #*filenum*, *X$*	Reads a string from the file, up to the next new-line character or end-of-file, or until the maximum string length is reached. The new-line character, if read, is discarded.
X$ = **Input** (*len*, [#] *filenum*)	The **Input** function reads a given number of characters from the file into a string.
Input #*filenum*, *variable-list*	The **Input** statement reads a list of variables from a file, expecting them in the format produced by a corresponding **Write** statement (see

4.7.1 Writing to Sequential Files). This statement can also appear on the right-hand side of an assignment statement:

n = **Input** #*filenum*, *variable-list*

This returns the number of variables in the list that were successfully input.

filenum	The *filenum* parameter to the **Open** command by which the file was opened.
X$	A variable or array element of type **String**.
len	The number of characters to read.
variable-list	A list of variables or array elements, separated by commas.

Access Conditions:

The file must have been opened with the *access* parameter equal to **Read** or **Read Write**.

Error Codes:

feInvalidMode	The file was not opened with *mode* equal to **Input** or **Append**.
feInvalidAccess	The file was not opened with *access* equal to **Read** or **Read Write**.
feReadError	The end of file was reached before enough bytes were read.

4.8.2 Reading from Binary and Random Files

The **Get** command is used to read from files that were opened with *mode* equal to **Binary** or **Random**. The read operation takes place at the current file position. After the **Get** command, the current file position advances to the next character (for **Binary** files) or the next record (for **Random** files):

Get [#] *filenum*, [*pos*], *variable* [, *len*]

filenum	The *filenum* parameter to the **Open** command by which the file was opened.
pos	A record number for **Random** files, and a character position for **Binary** files. If *pos* is not present (e.g. "**Get** *filenum*, , *variable*"), the read operation takes place at the current file position.
variable	A variable or array element. If this is of type **String**, it must be followed by the *len* parameter; otherwise the *len* parameter must be absent.
len	The number of characters to read, in the case that *variable* is of type **String**.

Access Conditions:

The file must have been opened with the *access* parameter equal to **Read** or **Read Write**.

Error Codes:

feInvalidMode	The file was not opened with *mode* equal to **Binary** or **Random**.
feInvalidAccess	The file was not opened with *access* equal to **Read** or **Read Write**.
feSeekError	File position *pos* does not exist.
feReadError	The end of file was reached before enough bytes were read.

4.9 File Locking and Unlocking

The commands in this section are valid only for files in single-application BasicCards.

4.9.1 Setting Read and Write Access Conditions

The **Read** and **Write** access conditions of a file or directory are changed with the following commands:

Read Lock *filename* [**Key** = *k1* [, *k2*]]
Read Unlock *filename*

Write Lock *filename* [**Key** = *k1* [, *k2*]]
Write Unlock *filename*

Read Write Lock *filename* [**Key** = *k1* [, *k2*]]
Read Write Unlock *filename*

filename	The path name of the file or directory.
k1, *k2*	The key numbers required to access the file or directory.

- The **Lock** command with no parameters sets the **Read** and/or **Write** access conditions of the specified file or directory to **Forbidden**.
- The **Lock** command with *k1* or *k2* specified sets the **Read** and/or **Write** access conditions of the specified file or directory to **Keyed** – the file can't be read or written from the Terminal program unless command/response encryption is currently active.
- The **Unlock** command sets the **Read** and/or **Write** access conditions of the specified file or directory to **Allowed**.

Access Conditions:

Write access is required to the file or directory, and to its parent directory.

Error Codes:

feNotRemoteFile	*filename* is not a BasicCard file or directory.

4.9.2 Setting and Unlocking a Custom Lock

If a file or directory has a **Custom** lock, it can't be read or written from a Terminal program unless the BasicCard program explicitly unlocks it. This allows access to a file or directory to be subject to any conditions, such as the presentation of a valid customer PIN number by the Terminal.

To set a **Custom** lock:

Lock *filename*

To unlock a **Custom** lock (BasicCard program only):

Unlock *filename*

Notes:

1. Once a **Custom** lock is set, it can never be permanently removed. A **Custom** lock is for ever.
2. If a **Custom** lock is unlocked, it can only be accessed from the Terminal program until the card is reset. After the card is reset, the BasicCard program must unlock the file or directory again before the Terminal program can access it.

Access Conditions:

For the "**Lock** *filename*" command, **Write** access is required to the file or directory, and to its parent directory. The "**Unlock** *filename*" command is not allowed in a Terminal program, so access conditions are not relevant.

Error Codes:

feNotRemoteFile	*filename* is not a BasicCard file or directory.
feTooManyCustomLocks	The maximum allowed number of **Custom** locks are already in place. (The implementation of the **Custom** lock mechanism in the BasicCard limits the number of locked files to 125.)

4.9.3 Retrieving the Access Conditions on a File or Directory

The access conditions on a file or directory can be obtained with the **Get Lock** command:

Get Lock *filename***,** *LockInfo*

filename The path name of the file or directory.

LockInfo A variable of user-defined type or a fixed-length string, at least seven bytes long. A suitable user-defined type **LockInfo** is defined in FILEIO.DEF:

```
Type LockInfo
  ReadLock As Byte
  WriteLock As Byte
  CustomLock As Byte
  ReadKey1@, ReadKey2@
  WriteKey1@, WriteKey2@
End Type
```

ReadLock and **WriteLock** can be **liAllowed**, **liForbidden**, **liKeyed1**, or **liKeyed2**. If **liKeyed1** or **liKeyed2**, then **ReadKey1@** etc. contain the appropriate key numbers.

CustomLock can be **liAllowed**, **liUnlocked**, or **liLocked**.

Access Conditions:

Read access is required to the parent directory.

Error Codes:

feNotRemoteFile *filename* is not a BasicCard file or directory.

4.10 Miscellaneous File Operations

filenum **= FreeFile** Returns a free *filenum* for use in a traditional **Open** statement. Returns –1 if no more file numbers are available, with error code **feTooManyOpenFiles**.

Seek [#] *filenum***,** *pos* Sets the file pointer to position *pos* (of type **Long**) for the next read or write operation on file *filenum*. *pos* is a record number for files opened with *mode*=**Random**; otherwise it is a byte count. Records and bytes are numbered from 1.

Note: If the file contains less than *pos*–1 bytes (or records), **Seek** fails with error code **feSeekError**, unless the file was opened for output in random access mode (*mode*=**Binary** or *mode*=**Random**, with **Write** access specified). In this case, the file is filled with zeroes to the required length.

Seek ([#] *filenum***)** Returns the read/write position for file *filenum*, as a **Long** value.

Len (#*filenum***)** Returns the length of file *filenum* in bytes, as a **Long** value.

EOF ([#] *filenum***)** Returns **True** if the end of file has been reached.

4.11 File Definition Sections

Using File Definition Sections, files and directories can be defined in the source code of the BasicCard program, to be created by the compiler. Files and directories so defined are downloaded to the BasicCard together with the BasicCard program itself. A File Definition Section begins with a **Dir** command and ends with the matching **End Dir** command. It may occur anywhere in a BasicCard program; it may contain only File Definition statements, not regular ZC-Basic statements. A program may contain any number of File Definition Sections.

This section describes the statements available in single-application BasicCard programs. File Definition Sections in a MultiApplication BasicCard program allow a much richer set of statements, including Component Definitions and Application Loader commands. See **5.5 Application Loader Definition Section** for more information.

4.11.1 Directory Definition

Dir *path*
    Lock Definitions
    File Definitions
    Sub-directory Definitions
End Dir

path The path name of the directory. It may be a new directory or an existing directory.

Lock Definitions	**Lock** and **Unlock** statements for the *path* directory. These have the same format as the statements described in **4.9 File Locking and Unlocking**, but without the *filename* parameter.
File Definitions	Definitions of files contained in the *path* directory (see **4.11.2 File Definition**).
Sub-directory Definitions	Nested Directory Definitions, defining sub-directories of the *path* directory. Each nested Directory Definition must end with its own **End Dir** statement.

File Definitions and nested Directory Definitions may occur in any order.

4.11.2 File Definition

A File Definition may occur only inside a Directory Definition. It ends with the next **File** or **Dir** statement, or with the **End Dir** statement of the enclosing Directory Definition.

File *filename* [**Len** = *blocklen*]
    Lock Definitions
    Data Definitions
    Input *inputfile*

filename The path name of the file.

blocklen The size of the new file's data blocks (see **4.2.3 Storage Requirements** for more information). If absent, *blocklen* defaults to 32. The special value **Len=0** sets the data block length to the length of the initial data, so that initially the file occupies exactly one data block.

Lock Definitions **Lock** and **Unlock** statements for the file. These have the same format as the statements described in **4.9 File Locking and Unlocking**, but without the *filename* parameter.

Data Definitions The initial data contained in the file. A Data Definition statement looks like this:

expr [**As** *type*] [**(***repeat-count***)**] [**,** *expr* [**As** *type*] [**(***repeat-count***)**]**, . . .**]

expr	Any constant expression of numerical or string type.
type	A data type. If absent, it defaults to the smallest data type that can contain *expr*. If *type* is a fixed-length string longer than *expr*, it is padded with NULL characters (ASCII zeroes) to the required length.
(*repeat-count***)**	The number of copies of *expr* to store in the file.

Note: To store a new-line character in the data, use the constant 10.

Input *inputfile* Copies the contents of file *inputfile* byte-for-byte into the BasicCard file. The compiler looks for *inputfile* in the same directories as it looks for **#Include** files – see **3.3.1 Source File Inclusion** for details.

4.12 The Definition File FILEIO.DEF

```
Rem  FILEIO.DEF
Rem
Rem  Declarations for ZC-Basic File I/O

#IfNotDef FileioDefIncluded ' Prevent multiple inclusion
Const FileioDefIncluded = True

#IfDef CompactBasicCard
#Error File I/O is not suported in the Compact BasicCard!
#EndIf

Rem  FileError codes

Const feFileOK              = 0
Const feInvalidDrive        = 1
Const feBadFilename         = 2
Const feBadFilenum          = 3
Const feFileNotFound        = 4
Const feFileNotOpen         = 5
Const feOpenError           = 6
Const feSeekError           = 7
Const feReadError           = 8
Const feWriteError          = 9
Const feCloseError          = 10
Const feInvalidMode         = 11
Const feInvalidAccess       = 12
Const feRenameError         = 13
Const feAccessDenied        = 14
Const feSharingViolation    = 15
Const feFileAlreadyExists   = 16
Const feNotDataFile         = 17
Const feNotDirectory        = 18
Const feDirNotEmpty         = 19
Const feBadFileChain        = 20
Const feFileOpen            = 21
Const feNameTooLong         = 22
Const feRecordTooLong       = 23
Const feTooManyOpenFiles    = 24
Const feTooManyCardFiles    = 25
Const feCommsError          = 26
Const feRemoteFile          = 27
Const feNotRemoteFile       = 28
Const feRecursiveRename     = 29
Const feInvalidFromKeyboard = 30
Const feBadParameter        = 31
Const feOutOfMemory         = 32
Const feNoFileSystem        = 33
Const feUnexpectedError     = 34
Const feNotImplemented      = 35
Const feTooManyCustomLocks  = 36
Const feBadKeyFile          = 37
Const feInvalidAlignment    = 38

Rem  File Attribute bits

Const faDirectory  = &H0010
Const faCardFile   = &H0040
```

```
#IfDef TerminalProgram

Const faReadOnly   = &H0001
Const faHiddenFile = &H0002
Const faSystemFile = &H0004
Const faArchived   = &H0020
Const faNormal     = &H0080
Const faTemporary  = &H0100

#EndIf

#IfNotDef MultiAppBasicCard

Rem  LockInfo defined type, for GET LOCK statement

Type LockInfo
  ReadLock As Byte       ' liAllowed, liKeyed1, liKeyed2, or liForbidden
  WriteLock As Byte      ' liAllowed, liKeyed1, liKeyed2, or liForbidden
  CustomLock As Byte     ' liAllowed, liUnlocked, or liLocked
  ReadKey1@, ReadKey2@   ' Key number(s) for ReadLock
  WriteKey1@, WriteKey2@ ' Key number(s) for WriteLock
End Type

Rem  LockInfo constants

Const liAllowed   = 0
Const liKeyed1    = 1
Const liKeyed2    = 2
Const liForbidden = 3
Const liUnlocked  = 1
Const liLocked    = 2

#EndIf ' MultiAppBasicCard

#EndIf ' FileioDefIncluded
```

5. The MultiApplication BasicCard

The **ZC6-** and **ZC8-**series **MultiApplication BasicCards** are a natural extension of the single-application Professional BasicCard family. The MultiApplication BasicCard was designed with two aims in mind:

- to retain the ease of programming that is such an attractive feature of the BasicCard;
- to let multiple Applications coexist in a single BasicCard without compromising their security.

These two aims have been achieved by retaining the ZC-Basic language essentially unchanged, with the additional concept of the Security Component.

5.1 Components

A Security Component (or Component for short) resembles a file, in that it has a name, resides in a directory, and can contain data. (In fact, in the MultiApplication BasicCard a file can be thought of as just another type of Component.)

5.1.1 Component Types

There are five Component types:

- **File** A data file or directory, just as in the Professional BasicCard.
- **ACR** Access Control Rule. An ACR defines the conditions by which a Component may be accessed. It is the only Component type that does not require a name.
- **Privilege** A Privilege can be granted to an Application (or to the Terminal program) to allow it access to a Component.
- **Flag** A Flag can be switched On or Off by an authorised Application, and then queried by an ACR to verify access conditions.
- **Key** A Key can be any length up to 255 bytes (**ZC6-**series) or 32767 bytes (**ZC8-**series). When you create a Key, you specify the uses to which the key can be put (for example External Authentication), and the cryptographic algorithms that it may be used in (for example AES-128).

5.1.2 Component Properties

A Component name follows the same rules as a file name. Two Components in the same directory may have the same name if they are of different types. A Component may have *Attributes* and *Data*, which can be read and written separately if the requisite access conditions are satisfied. The format of a Component's Attributes and Data depends on its type, and on whether the Component is being created, written, or read. The various formats are described in the following sections.

Each Component has a unique two-byte Component ID, or CID, that is assigned by the BasicCard operating system when the Component is created. This ID is required as a parameter in a number of **COMPONENT** System Library procedures. This Library provides two procedures, **FindComponent** and **ComponentName**, for obtaining the CID of a Component from its name and vice versa.

The top four bits of a CID determine the type of the Component; the value ((CID **Shr** 8) **And** &HF0) is equal to one of the following constants, defined in the file **Componnt.def**:

ctFile	**&H10**
ctACR	**&H20**
ctPrivilege	**&H30**
ctFlag	**&H40**
ctKey	**&H70**

5.1.3 Component Access Control

Access to a Component is controlled according to its Access Control Rule, or ACR. The ACR specifies the conditions under which the various types of access are allowed. An ACR may be assigned to any Component; an ACR itself, being a Component, may also be protected with a (different) ACR.

The MultiApplication BasicCard defines five access types:

Read	Required to read a Component's data (or the contents of a directory)
Write	Required to write a Component's data (or to create a Component in a directory)
Execute	Required to select an Application
Delete	Required to delete a Component, or to write its attributes
Grant	Required to grant a Privilege to an Application (or to the Terminal program)

The conditions under which each access type is allowed may be separately specified. See **ACR Definition 5.5.5** for information on how to define an ACR in the source code of an Application; see **7.4 The COMPONENT Library** and **5.9.2 ACRs** for details on how to create an ACR dynamically at run-time.

5.2 Applications

From the point of view of the MultiApplication BasicCard, an Application is just an executable file. But from the point of view of the programmer, an Application will also contain various Components – data files, keys, ACR's etc. This section concentrates on the Application as an executable file; for information on how to bundle an Application with the Components that it needs, see **5.5 Application Loader Definition Section.**

5.2.1 Application Files

An Application file is an executable file, that contains compiled ZC-Basic code for the execution of commands. It must satisfy certain conditions:

- the first four bytes must be "**ZCAF**";
- it must be at least 37 bytes long;
- (**ZC8**-series only) it must be aligned to at least 16 bytes, with **Align=4** or greater (normally the ZCMBasic compiler does this for you);
- (**ZC6**-series only) it must be allocated as a contiguous block of EEPROM.

In addition, it must satisfy the **ExecutableAcr** if this ACR is configured. In the **ZC8-**series card, this means that the **CardConfigExecutableAcr** data item is configured (see **5.3 Card Configuration in ZC8-Series Cards**); in the **ZC6-**series card, it means that there exists an ACR in the Root directory with the name "**Executable**".

An Application file contains compiled code for all the commands that the Application supports. It also contains the **Eeprom** data used by the Application. Such data is not shareable between Applications; if different Applications want to share data, they must use the File System. If an Application uses **Eeprom** strings or dynamic arrays, then it needs its own Heap, which also resides in the Application file.

An Application file can be created in one of two ways:

- with an **Application** *filename$* statement in the File Definition Section;
- with the "**–OA**" compiler command-line option.

The first option embeds the file in an Image file or Debug file for use by the Application Loader; the second option creates an Application file in the host computer (which can then be loaded "by hand").

5.2.2 Selecting an Application

When the MultiApplication BasicCard is reset, the operating system looks to see whether the card contains a *Default Applicatio*n. In the **ZC8-**series card, this means that the **CardConfigDefaultApp** data item is configured (see **5.3 Card Configuration in ZC8-Series Cards**); in the **ZC6-**series card, it means that there exists an Application file in the Root directory with the name "**DefaultApp**". If such a file exists, it is selected, and becomes the *Current Application*. (If no such file exists, then there is initially no Current Application.) The Current Application is the Application file whose command table is searched when a command is received. If a match is found, then the code for the matched command is executed.

Subsequently, the Current Application can be changed by selecting a new Application. This is done by calling the System Library procedure **SelectApplication** (*filename$*), either from the Terminal program or from within the card. If an Application selects another Application in this way, then the previous Application's code is no longer accessible, so code after the **SelectApplication** call will not get executed unless the Application selection fails for some reason.

To select an Application, **Execute** access is required to the Application file.

5.2.3 Catching Undefined Commands

If the card contains a Default Application, it can be configured to catch undefined commands. This means that if a command is received that is not supported by the Current Application, then the Default Application's command table is searched for a match. If an undefined command is caught by the Default Application in this way, then the Current Application is closed, and the Default Application becomes the new Current Application.

To configure an Application to catch undefined commands:

#Pragma CatchUndefinedCommands

This statement is allowed in any Application, but it has no effect except in the Default Application.

5.2.4 Memory Allocation

The MultiApplication BasicCard has three types of heap for memory allocation:

- The Global Heap is for Files and Components, including Application Files. It occupies the whole of the available EEPROM in the card.
- Each Application has its own EEPROM Heap, which is an area in the Application File for the Application's **Eeprom String** variables and **Eeprom** dynamic arrays. Its size can be configured with the **#Heap** statement, or in the **ZCMDCard** BasicCard Debugger.
- The RAM heap is for an Application's temporary (**Public** and **Private**) **String** variables and dynamic arrays. It is cleared whenever an Application is selected. Its size depends on the sizes of the Application's stack and fixed-length temporary data; the three regions **RAMHEAP**, **STACK**, and **RAMDATA** together occupy about 1100 bytes in MultiApplication BasicCard **ZC6.5**, and about 4000 bytes in MultiApplication BasicCard **ZC8.6**.

To see the exact lengths of an Application's EEPROM and RAM heaps, ask the compiler to generate a Map file. To find out the amount of free memory available in each heap, see **7.15.9 Free Memory**.

5.3 Card Configuration in ZC8-Series Cards

The **ZC8**-series MultiApplication BasicCard introduces many configurable options that are absent in the **ZC6**-series card, most of them to do with contactless protocol and Mifare™. To streamline the configuration mechanism, all the configurable data items that apply to the whole card have been collected into the Card Configuration data area. Unless explicitly configured, all these data items are empty; in this case, the card will use a default value. (There is a difference between an empty data item and an item that has been configured to have the default value: if the **ConfigAcr** specifies **Write Once**, then only empty data items can be written.)

Each data item is read and written using a **String** parameter. This includes single-byte data items, which are read and written via a string of length 1. To delete a data item, simply write an empty string.

There are three ways to write a Card Configuration item:

- use a **#Pragma** directive in the source file;
- call the **LCWriteCardConfig (***Tag@* **,** *Value$***)** Loader Command in the source file;
- call the **WriteCardConfig (***Tag@* **,** *Value$***)** System Library procedure from a Terminal program or a BasicCard application.

The *Tag@* values are defined in the **Componnt.def** file, as follows:

```
Const CardConfigConfigAcr           = 1
Const CardConfigExecutableAcr       = 2
Const CardConfigMifareAcr           = 3
Const CardConfigATR                 = 4
Const CardConfigATS                 = 5
Const CardConfigRFClock             = 6
Const CardConfigClock               = 7
Const CardConfigUIDFlags            = 8
Const CardConfigInverseConvention   = 9
Const CardConfigDisableRF           = 10
Const CardConfigEnableMifare        = 11
Const CardConfigSAKATQA             = 12
Const CardConfigCardID              = 13
Const CardConfigDefaultApp          = 14
Const CardConfigECFilename          = 15
Const CardConfigECCurveName         = 16
Const CardConfigRsaFastPrKOps       = 17
Const CardConfigRsaDisableFastPrKOps = 18
Const CardConfigDSACompatibilityMode = 19
```

These items are explained below.

To read the value of a Card Configuration data item, call **ReadCardConfig (***Tag@***)**, which returns a **String** value. If the data item has not been configured, **ReadCardConfig** returns the default value, unless the top bit of *Tag@* is set, in which case an empty string is returned for unconfigured data items. The constant

```
Const CardConfigReadConfiguredOnly   = &H80
```

is defined in **Componnt.def** to use in combination with a data item tag.

ConfigAcr

Default value: none
#Pragma directive: **#Pragma ConfigAcr** *ACRName$*

If this data item is non-empty, then it contains the name of an ACR, which must be satisfied before the Configuration Data can be accessed (this includes the **ConfigAcr** data item itself). Like any ACR, the **ConfigAcr** can contain separate **Read**, **Write**, and **Delete** conditions. If the data item is non-empty, but the ACR doesn't exist, then no access is allowed.

ExecutableAcr

Default value: none
#Pragma directive: **#Pragma ExecutableAcr** *ACRName$*

If this data item is non-empty, then it contains the name of an ACR, which must be satisfied by an Application before it can be selected. If the data item is non-empty, but the ACR doesn't exist, then access is allowed (in contrast to **ConfigAcr** and **MifareAcr**).

MifareAcr

Default value: none
#Pragma directive: **#Pragma MifareAcr** *ACRName$*

If this data item is non-empty, then it contains the name of an ACR, which must be satisfied by an Application before it can access the Mifare™ data blocks via the **Mifare™** System Library (see **7.13 The Mifare™ Library**). If the data item is non-empty, but the ACR doesn't exist, then no access is allowed.

ATR

Default value: depends on card type and revision
#Pragma directive: **#Pragma ATR (***ATR-Spec***) –** see **3.21.1 Customised ATR** for details

If this data item is non-empty, it is sent as the ATR (Answer To Reset) when the card is reset by an ISO-7816 card reader.

ATS

Default value: depends on card type and revision
#Pragma directive: **#Pragma ATS (***ATS-Spec***) –** see **3.21.2 Customised ATS** for details

If this data item is non-empty, it is sent as the ATS (Answer To Select) when the card is selected by an ISO-14443 contactless card reader.

RFClock

Default value: **#Pragma RFClock (18, 18, 4)**, encoded as **Chr$(&H58)**
#Pragma directive: **#Pragma RFClock (**[*C*],[*R*],[*D*]**)**

If this data item is non-empty, it sets the speeds of the CPU, the Crypto Co-processor, and/or the **DES** and **AES** co-processors when the card is selected by an ISO-14443 contactless card reader. See **7.15.10 Power Management** for the meaning of *C*, *R*, and *D*.

Clock

Default value: **#Pragma Clock (31, 72, 0)**, encoded as **Chr$(&HEA)**
#Pragma directive: **#Pragma Clock (**[*C*],[*R*],[*D*]**)**

If this data item is non-empty, it sets the speeds of the CPU, the Crypto Co-processor, and/or the **DES** and **AES** co-processors when the card is reset by an ISO-7816 card reader. See **7.15.10 Power Management** for the meaning of *C*, *R*, and *D*.

UIDFlags

Default value: depends on card type and revision
#Pragma directive: **#Pragma UID (** *param* [, *param*] **)**

If this data item is non-empty, it specifies the properties of the UID (Unique Identifier) that the card responds with during the contactless Card Selection protocol. See **3.3.13 The #Pragma Directive** for details.

InverseConvention

Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma InverseCnvention**

If this data item is set to a non-zero value, the card will use the Inverse Convention when communicating with an ISO-7816 card reader.

DisableRF

Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma DisableRF**

If this data item is set to a non-zero value, contactless communication is disabled (but Mifare™ communication is still allowed, if **EnableMifare** is set).

EnableMifare

Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma EnableMifare**

If this data item is set to a non-zero value, Mifare™ capability is enabled in the card. (Even if this data item is not set, the Mifare™ data blocks can still be accessed from an Application via the **Mifare™** System Library – see **7.13 The Mifare™ Library**.)

SAKATQA

Default value: **Chr$(0)**
#Pragma directive: **#Pragma SAKATQA (***SAK*, *ATQA0*, *ATQA1***)**

This data item contains communication parameters for the contactless protocol card selection sequence. Refer to the Internationl Standard **ISO/IEC 14443**: *Proximity Cards* for details.

CardID

Default value: none
#Pragma directive: **#Pragma CardID** *CardID$*

If this data item is set, then *CardID$* is sent in response to a **GET APPLICATION ID** command with **P1=&H00**, **P2=&H02** – see **8.9.10 The GET APPLICATION ID Command**.

DefaultApp

Default value: none
#Pragma directive: **#Pragma DefaultApp** *AppFilename$*

If this data item is non-empty, then it contains the filename of the Default Application, which is automatically selected whenever the card is reset. See **5.2.2 Selecting an Application** for more information.

ECFilename

Default value: none
#Pragma directive: **#Pragma ECFilename** *ECFilename$*

If this data item is non-empty, then the Elliptic Curve Domain Parameters for the **EC167**, or **EC211,** or **EC-p** System Library are loaded from it automatically whenever the card is reset. The file may also contain pre-computed data for speeding up Elliptic Curve operations. Either the data in the file must occupy a single contiguous block, or the file must have at least 16-byte alignment (set with **Align=4**). See **5.4.3 Elliptic Curve Domain Parameters** for more information.

ECCurveName

Default value: none
#Pragma directive: **#Pragma ECCurveName** *CurveName$*

If you want to use one of the pre-defined Elliptic Curves as the default, then you can specify it by its *Curve Name*. This is one of the following (case is significant):

"EC167-1" to **"EC167-5"**
"EC211-1" to **"EC211-5"**
"ECp-1" to **"ECp-19"**

RsaFastPrKOps

Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma RsaFastPrKOps**

Switch fast private-key operations on or off. See **7.1.14 Fast Private-Key Operations** for details.

RsaDisableFastPrKOps
Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma RsaDisableFastPrKOps**

Disable fast private-key operations. See **7.1.14 Fast Private-Key Operations** for details.

DSACompatibilityMode
Default value: **Chr$(0)**, i.e. **False**
#Pragma directive: **#Pragma DSACompatibilityMode**

If this data item is set to a non-zero value, then the old, non-standard-compliant version of the Elliptic Curve DSA algorithm is used, to ensure compatibility with earlier versions of the BasicCard. See ???? for more information.

5.4 Special Files in ZC6-Series Cards

Certain filenames have special meanings in the **ZC6-**series MultiApplication BasicCard.

5.4.1 ATR File

If a file with the name "**ATR**" exists in the Root Directory, its contents are used as the Answer To Reset, sent by the BasicCard whenever it is reset by the Terminal program. The complete ATR – protocol definition bytes and Historical Characters – must be included in the file, with a trailing flag byte. The special syntax

ATR (*ATR-Spec***)**

in a File Definition denotes a string constant that lets you specify the ATR in the same way as the **#Pragma ATR** directive – see **3.21.1 Customised ATR** for the format of *ATR-Spec*.

The following example configures a MultiApplication BasicCard to use the **T=0** protocol:

```
#Include ATRList.def

Dir "\" ' Root directory
  File "ATR" Lock Read: Always ' Make the file read-only
    ATR (T=0)
End Dir
```

Use this feature with care, as an invalid ATR can make the card unusable. You should at the very least try out the ATR in a simulated BasicCard before testing it in a real card.

5.4.2 Card ID File

If a file with the name "**CardID**" exists in the Root Directory, its contents are sent in response to a **GET APPLICATION ID** command with **P1=&H00**, **P2=&H02** – see **8.9.10**.

5.4.3 Elliptic Curve Domain Parameters

If a file with the name "**ECDomainParams**" exists in the Root Directory, the Elliptic Curve Domain Parameters for the **EC167** or **EC211** System Library are loaded from it automatically whenever the card is reset. The file may also contain pre-computed data for speeding up Elliptic Curve operations. The data in this file must occupy a single contiguous data block in EEPROM. Suitable data files are provided in the `\BasicCardV8\Lib\Curves` directory. For example:

```
Dir "\" ' Root directory
  File "ECDomainParams" Lock=Read:Always
    #Include \BasicCardV8\Lib\Curves\EC167-4.64
End Dir
```

This loads the 167-bit Elliptic Curve number 4, with 64 pre-computed points.

5.5 Application Loader Definition Section

An Application will typically require various Components, such as data files and keys, to be created before it can work properly. Creating these Components, and downloading the Application file, will often require a complicated sequence of cryptographic operations, such as **EXTERNAL AUTHENTICATE** commands. This process can be automated by defining it in the source file of the Application itself, in an Application Loader Definition Section. The statements in this Section are saved in the Image file, for interpretation by the Application Loader.

An Application Loader Definition Section is actually an enhanced version of the File Definition Section described in **4.11 File Definition Sections**. (Before reading this Section, you may want to review File Definition Sections.) It consists of a Directory Definition, that can contain File Definitions, nested Directory Definitions, Component Definitions, and Loader Commands.

5.5.1 Common Component Attributes

All Components have the following three attributes in common:

Ref=*ref* Specifies a reference number between 1 and 65535 by which the Component may be referred to later in the Loader Definition Section. This number must be unique.

Lock=*ACR* Specifies the ACR of the Component. *ACR* is either (i) the pathname of a previously defined ACR; or (ii) the Reference number of a previously defined ACR; or (iii) an ACR Specification. In case (iii), the Application Loader will create an Anonymous ACR.

If a Component has no ACR, anybody can read, write, or delete it. This is usually a bad idea, so every Component definition is required to contain a **Lock** attribute. However, you can specifically request an unprotected Component, with **Lock=Open**.

Create=*option* where *option* is one of the following:

Always The Component is always created. If the Component already exists in the card, the Application Loader signals an error and fails.

Once If the Component doesn't already exist in the card, it is created. Otherwise the attributes of the existing Component are checked against the attributes specified in the Component definition; if they don't match, the Application Loader signals an error and fails. No such check is performed on the Component's data.

Update If the Component doesn't already exist in the card, it is created. If the Component already exists, its attributes and data are updated to match the attributes specified in the Component definition.

Never The Component is never created. If the Component does not already exist in the card, the Application Loader signals an error and fails. If any attributes are specified in the Component definition, they are checked against the attributes of the existing Component; if they don't match, the Application Loader signals an error and fails.

If no **Create** attribute is present in a Component Definition, the default is **Create=Update** for directories, and **Create=Always** for other Component types (but this default can be overridden by **Option Create**=*option*).

These attributes will be referred to as *common-attribute* in the following paragraphs.

5.5.2 Directory Definition

Dir *name$* [*common-attribute common-attribute*...]
 [*common-attribute common-attribute*... | *component-definition* | *loader-command*]
End Dir [**Lock=***ACR*]

component-definition is one of:
 Directory Definition
 Data File Definition
 Application File Definition
 ACR Definition
 Privilege Definition
 Flag Definition
 Key Definition

See **5.5.9 Loader Commands** for information on *loader-command*.

The reason that "**End Dir Lock=***ACR*" may be useful is that it lets you assign a Lock to a Directory that depends on a Key or an ACR that belongs to the Directory itself. For instance,

```
Dir "MyApp"
  Key "MyKey" Lock=Never Usage=kuExtAuth Algorithm=AlgAes128
    "(16-byte secret)"
End Dir Lock = Read:Always; Write:ExtAuth("MyKey")
```

5.5.3 Data File Definition

File *name$* [*attribute attribute...*]
 [*attribute attribute...* | *data* | **Input** *inputfile*]
 [*attribute attribute...* | *data* | **Input** *inputfile*]
 ...

attribute	*common-attribute* | **Len=***blocklen* | **Align=***alignment* As a special case, **Len=0** sets *blocklen* to the initial length of the file.
data	Data to be stored in the file. See **4.11.2 File Definition** for details.
Input *inputfile*	Name of file to be included byte-for-byte in the BasicCard file.

5.5.4 Application File Definition

This is a special case of a Data File Definition. It defines a file which is to contain the compiled code and data of the Application.

Application *name$* [*attribute attribute...*]
 [*attribute attribute*...]
 [*attribute attribute*...]

attribute *common-attribute* | **Len=***blocklen* | **Align=***alignment*

No data statement is allowed. An Application file must satisfy certain conditions, which depend on the BasicCard version:

- In a **ZC6-**series MultiApplication BasicCard, an Application File must be allocated in a single contiguous block, which the compiler ensures by setting *blocklen* to the length of the file, as if by **Len=0**. So although **Len=***blocklen* is allowed here, it should usually be absent.
- In a **ZC8-**series MultiApplication BasicCard, an Application File must have an alignment of at least 16=2^4 bytes, which the compiler ensures by setting *alignment* equal to 4. So although **Align=***alignment* is allowed here, it is not required; if present, *alignment* should be at least 4.

5.5.5 ACR Definition

ACR *name$* [*common-attribute common-attribute...*]
[*common-attribute common-attribute... | condition*]
[*common-attribute common-attribute... | condition*]
...

condition	One of the following:	*When satisfied*
	Always	Always
	Never	Never
	ACR **And** *ACR* **And** ... **And** *ACR*	If all *ACR*'s in the list are satisfied
	ACR **Or** *ACR* **Or** ... **Or** *ACR*	If at least one *ACR* in the list is satisfied
	qualified-list	See below
	Not *ACR*	If *ACR* is not satisfied
	(*ACR*)	If *ACR* is satisfied
	Write Once	If the Component data field is empty
	Verify (*Key*)	If the **VERIFY** command has been called with *Key*
	ExtAuth (*Key*)	If the **EXTERNAL AUTHENTICATE** command has been called with *Key*
	SMEnc (*Key*)	If the **START ENCRYPTION** command has been called with *Key* for an Encryption algorithm (**EAX, AES, DES**)
	SMMac (*Key*)	If the **START ENCRYPTION** command has been called with *Key* for an Authentication algorithm (**OMAC**)
	Privilege (*Privilege*)	If the current Application file (or the Terminal program for external access) has been granted the given *Privilege*
	Flag (*Flag*)	If the given *Flag* is set
	Signed (*Key*)	If the current Application file was signed using *Key*, in an **AUTHENTICATE FILE** command or during Secure Transport
	Application (*File*)	If *File* is the current Application
	SecTrans (*Key*)	If Secure Transport with *Key* is active

qualified-list has the form

access-type-list **:** *ACR* **;** *access-type-list* **:** *ACR* **;** ... [*access-type-list* **:**] *ACR*

where *access-type-list* is a list of access types (**Read**, **Write**, **Execute**, **Delete**, **Grant**) separated by commas. If the last *ACR* in the list is not preceded by an *access-type-list*, it applies to all access types not previously mentioned. If every *ACR* is preceded by an *access-type-list*, then access types not occurring in the list are forbidden.

The corresponding list in **5.9.2 ACRs** gives the binary data format of these ACR types.

The *condition* (i.e. the meaning of the ACR) must occur on a single line, except that multiple *access-type-list* specifications may be split into separate lines. For example:

```
ACR "MyACR" Lock=Never
    Read, Execute: Always
    Write: Verify ("MyPassword")
    ExtAuth ("MyKey")
```

Here, **ExtAuth ("MyKey")** becomes the access condition for the unspecified access types (**Delete** and **Grant**).

5.5.6 Privilege Definition

A Privilege has no special attributes, and no data:

Privilege *name$* [*common-attribute common-attribute*...]
[*common-attribute common-attribute*...]

5.5.7 Flag Definition

Flag *name$* [=*value*] [*attribute attribute*...]
[*attribute attribute*...]

value The initial value of the Flag (the Flag will be set if *value* is non-zero).

attribute *common-attribute* | **SetAttr**=*bitmask*

The *bitmask* values are defined in **5.9.4 Flags**.

5.5.8 Key Definition

Key *name$* [(*error-counter*[, *reset-value*])] [*attribute attribute*...]
[*attribute attribute*... | *data*]
[*attribute attribute*... | *data*]
...

error-counter The initial value of the Key's Error Counter.

reset-value The reset value of the Key's Error Counter. If absent, it is set equal to *error-counter*.

attribute *common-attribute* | **Usage=***usage-list* | **Algorithm=***algorithm-list*

data The value of the Key. This is a **Binary Data Field**, which can take the following forms:

- a **String** constant
- **LCIndexedKey** (*LookupTime*, *Index*)

  The Key takes its value from a **Declare Key** *Index* statement (see **3.18.3 Key Declaration**). *LookupTime* is one of the values **ltCompileTime** or **ltLoadTime** defined in **Componnt.def**. If **ltCompileTime**, the Key is evaluated by the compiler from a **Declare Key** statement in the source code; if **ltLoadTime**, the Key is evaluated by the Application Loader from a **Declare Key** statement read in an **LCReadKeyFile** command (see **5.5.9 Loader Commands**).
- **LCSerialNumber** (*LookupTime*)

  The Key takes the value of the 8-byte Serial Number of the card. *LookupTime* is one of the values **ltCompileTime** or **ltLoadTime** defined in **Componnt.def**. If **ltCompileTime**, the compiler uses the Serial Number defined in the command-line parameter **–Nxxxxxxxxxxxxxxxx**, where each **x** is a hexadecimal digit. If **ltLoadTime**, the Application Loader asks the card for its Serial Number via the **GET APPLICATION ID** command.

  Notes

  1. This feature is expected to be more useful as the *Seed* parameter to **LCBuildKey** than as a way of assigning a card's Serial Number to a Key. See **5.6 Secure Transport** for an example.
  2. A simulated BasicCard has the Serial Number **0123456789ABCDEF**. The **ZCMDCard** BasicCard debugger uses this value when compiling a MultiApplication BasicCard program. To specify a different value, the **ZCMBasic** command-line compiler must be used.

- **LCBuildKey** (*Key*, *Len*, *Seed*)

  This function generates *Len* bytes of data from *Key* and *Seed*, using the **SHA-1** Secure Hash Algorithm. The *Seed* parameter is itself a Binary Data Field, which may take any of the forms defined in this paragraph. For example, if *Key* is a Master Key known only to the card issuer, and *Seed* is the card's Serial Number, then this function can be used to generate card-specific keys, for Secure Transport and other uses. See **5.6 Secure Transport** for an example of this.

- **LCKey** (*Key*)

  This function returns the value of *Key*.

- **LCPublicKey** (*PrivateKey*, *Algorithm*)

  The Key takes the value of the Public Key corresponding to the given *PrivateKey*. The *PrivateKey* parameter is a Binary Data Field, which the compiler must be able to evaluate (i.e. *LookupTime*=**ltLoadTime** is not allowed). *Algorithm* must be one of **AlgEC167NR**, **AlgEC211NR**, **AlgEC167DSA**, **AlgEC211DSA**, **AlgECpNR**, **AlgECpDSA**, **AlgRSAPSS**, or **AlgRSAPKCS1** (but only the first two are allowed in a **ZC6-**series card).

  The *PrivateKey* parameter is not stored in the Image file.

Multiple *data* statements are allowed, as long as they can all be evaluated at compile time; the values are concatenated.

usage-list is a list of Key Usage values, separated by commas. The values specify the uses to which the key may be put. In general, for maximum security, it is advisable to avoid using a given key for more than one purpose. The following Key Usage values are defined in **Componnt.def**:

Const kuVerify	**= 1**	Password Verification
Const kuExtAuth	**= 2**	External Authentication
Const kuSMEnc	**= 3**	Secure Messaging with Encryption algorithm
Const kuSMMac	**= 4**	Secure Messaging with Authentication algorithm
Const kuSign	**= 5**	Digital Signature and File Authentication
Const kuIntAuth	**= 6**	Internal Authentication
Const kuSecTrans	**= 7**	Secure Transport of Files and Keys

algorithm-list is a list of Algorithm IDs, separated by commas. The IDs specify the cryptographic algorithms that the key may be used with. The following Algorithm IDs, defined in **AlgID.def**, are accepted by all MultiApplication BasicCards:

Const AlgSingleDesCrc	**= &H23**
Const AlgTripleDesEDE2Crc	**= &H24**
Const AlgTripleDesEDE3Crc	**= &H25**
Const AlgAes128	**= &H31**
Const AlgAes192	**= &H32**
Const AlgAes256	**= &H33**
Const AlgEaxAes128	**= &H41**
Const AlgEaxAes192	**= &H42**
Const AlgEaxAes256	**= &H43**
Const AlgOmacAes128	**= &H81**
Const AlgOmacAes192	**= &H82**
Const AlgOmacAes256	**= &H83**
Const AlgEC167NR	**= &HC1**
Const AlgEC211NR	**= &HC2**
Const AlgEC167DSA	**= &HC3**
Const AlgEC211DSA	**= &HC4**

The following Algorithm IDs are acceped in **ZC8-**series cards only:

Const AlgECpNR	**= &HE1**
Const AlgECpDSA	**= &HE2**
Const AlgRSAPSS	**= &HE3**
Const AlgRSAPKCS1	**= &HE4**

In the **ZC8-**series MultiApplication BasicCard, you can specify a hash algorithm to use with algorithms ≥ **AlgEC167NR**:

Const AlgSigHashDefault	**= &H00** (see below)
Const AlgSigHashSha1	**= &H08**
Const AlgSigHashSha224	**= &H0C**
Const AlgSigHashSha256	**= &H10**
Const AlgSigHashSha384	**= &H14**
Const AlgSigHashSha512	**= &H18**

AlgSigHashDefault means **SHA-1** with **AlgEC167NR** or **AlgEC16DSA**, and **SHA-256** otherwise.

5.5.9 Loader Commands

Loader Commands are directives to the Application Loader. To use Loader Commands:

#Include LoadComm.def

The ZC-Basic compiler embeds the Loader Commands in the Image file. The Application Loader reads them from the Image file and executes them, in the order that they occur in the Application Loader Definition Section. They will typically be interleaved with Component Definitions. In the list of Loader Commands given below, the parameters take the following form:

File — A filename or File Reference number

Key, *Privilege* — Either a constant string containing the pathname of a previously defined Component, or a constant integer which is the Reference number of a previously defined Component. (Reference numbers are assigned with the **Ref=***ref* attribute.)

Algorithm — A cryptographic algorithm ID. A list of algorithm IDs can be found in the previous section**.**

LCReadKeyFile (*filename$*)

Read the Key file into the **Key()** array. The Key file must be present on the host computer when the Application Loader runs. This is useful in conjunction with the *index* parameter in a Key Definition – see **5.5.8 Key Definition**.

LCEC167SetCurve (*DomainParams* **As String*64**)

DomainParams is a string constant that contains a copy of an **EC167DomainParams** structure. File **EC167Crv.str** in the Lib\Curves directory contains string constants **EC167Curve1String** through **EC167Curve5String** for the five pre-defined Elliptic Curves. This procedure must be called before using 167-bit Elliptic Curve operations in the Application Loader Section.

LCEC211SetCurve (*DomainParams* **As String*82**)

DomainParams is a string constant that contains a copy of an **EC211DomainParams** structure. File **EC211Crv.str** in the Lib\Curves directory contains string constants **EC211Curve1String** through **EC211Curve5String** for the five pre-defined Elliptic Curves. This procedure must be called before using 211-bit Elliptic Curve operations in the Application Loader Section.

LCECpSetCurve (*CurveIndex@*)

CurveIndex@ is a byte from 1 to 19, the index of the pre-defined **EC-p** curve. This procedure must be called before using **EC-p** Elliptic Curve operations in the Application Loader Section.

LCStartSecureTransport (*Key*, *Algorithm*)

Start Secure Transport of Files and Keys, using the given Key and Algorithm. All Files and Keys will be stored in the Image File in encrypted form, for decryption by the BasicCard. This deactivates the current Application in the card, and disables Application selection until **LCEndSecureTransport**() is called. See **5.6 Secure Transport** and **8.9.33 The SECURE TRANSPORT Command** for more information.

Valid algorithms: **AlgEaxAes128**, **AlgEaxAes192**, **AlgEaxAes256**.

LCEndSecureTransport()

End Secure Transport of Files and Keys.

LCStartEncryption (*Key*, *Algorithm*)

Call the **START ENCRYPTION** command (see **8.9.11**) with the given Key and Algorithm. All algorithms from **AlgSingleDesCrc** (**&H23**) to **AlgOmacAes256** (**&H83**) are valid.

LCEndEncryption()

Call the **END ENCRYPTION** command (see **8.9.12**).

LCExternalAuthenticate (*Key*, *Algorithm*)

Call the **EXTERNAL AUTHENTICATE** command (see **8.9.16**) with the given Key and Algorithm.

Valid algorithms: **AlgSingleDesCrc**, **AlgTripleDesEDE2Crc**, **AlgTripleDesEDE3Crc**, **AlgAes128**, **AlgAes192**, **AlgAes256**.

LCInternalAuthenticate (*Key*, *Algorithm*)

Call the **INTERNAL AUTHENTICATE** command (see **8.9.17**) with the given Key and Algorithm.

Valid algorithms: **AlgSingleDesCrc**, **AlgTripleDesEDE2Crc**, **AlgTripleDesEDE3Crc**, **AlgAes128**, **AlgAes192**, **AlgAes256**.

LCVerify (*Key*)

Call the **VERIFY** command (see **8.9.18**) with the given Key.

LCGrantPrivilege (*Privilege*, *File*)

Call the **GRANT PRIVILEGE** command (see **8.9.29**) with the given Privilege and File.

LCAuthenticateFile (*Key*, *Algorithm*, [*PrivateKey*,] *File*)

Call the **AUTHENTICATE FILE** command with the given parameters. The signature is computed at compile time, so the Key and the contents of the File must be available to the compiler. The *PrivateKey* parameter is required if *Algorithm* is ≥ **AlgEC167NR**. See **5.8 File Authentication** and **8.9.30The AUTHENTICATE FILE Command** for more information.

Valid algorithms: **AlgOmacAes128**, **AlgOmacAes192**, **AlgOmacAes256**, **AlgEC167NR**, **AlgEC167DSA**, **AlgEC211NR, AlgEC211DSA**; and in **ZC8-**series cards: **AlgECpNR, AlgECpDSA, AlgRSAPSS, AlgRSAPKCS1**.

LCCheckSerialNumber ()

Check that the card's Serial Number matches that specified in the compiler's **–N** parameter. If not, the Application Loader issues an appropriate error message and fails. The Application Loader uses the **GET APPLICATION ID** command (see **8.9.10**) to read the card's serial number.

LCWriteCardConfig (*Tag@*, *Value$*)

(**ZC8-**series cards only) Write *Value$* to the Card Configuration data item *Tag@*. See **5.3 Card Configuration in ZC8-Series Cards** for a list of data item tags.

5.6 Secure Transport

The MultiApplication BasicCard allows an Application to be loaded at any time. To control the conditions under which this happens, you can set access conditions on the directories of the card, using ACR's. And to ensure the secrecy of the Files and Keys that are loaded, you can use the Secure Transport mechanism. This encrypts the data fields of all Files and Keys in the Image file, using a Key known only to the card and to the issuer. The Application Loader does not need to know this Key, so the encrypted data remains secret.

5.6.1 An Example

The Secure Transport Key will typically be loaded into the card by the card issuer, at card initialisation time. This is a secure environment, so the data need not be encrypted. The following example creates a Secure Transport Key in the card that depends on the card's Serial Number. First, generate a Master Key file using the **KeyGen** utility (see **6.9.4 The Key Generator KeyGen.**). For example:

```
KeyGen –K100(16) MK.DAT
```

Next, use the Master Key to build a Secure Transport Key for each card:

```
#Include Componnt.def
#Include LoadComm.def

#Include MK.DAT

Dir "\" Create=Update

  Key "Master Key" Create=Never
    LCIndexedKey (ltCompileTime, 100)

  Key "Secure Transport Key" Lock=Never
    Usage=kuSecTrans Algorithm=AlgEaxAes128
    LCBuildKey ("Master Key", 16, LCSerialNumber (ltLoadTime))

End Dir Lock Read:Always; Write:SecTrans("Secure Transport Key")
```

Key "**Master Key**" is needed by the Application Loader, and so it must be stored (unencrypted) in the Image file. As this Image file is only used at card initialisation time, this does not compromise the Key's security. Key "**Secure Transport Key**" is calculated by the Application Loader, using the Serial Number that it reads from the card; only this Key is loaded into the card.

(Instead of including **MK.DAT** at compile time, it could have been read at load time, as follows:

```
Call LCReadKeyFile ("MK.DAT")
Key "Master Key" Create=Never
  LCIndexedKey (ltLoadTime, 100)
```

In a secure environment, there is nothing to choose between these two methods.)

Create=Update is required in the Directory Definition, because we change the ACR attribute of the root directory to **Read:Always; Write:SecTrans("Secure Transport Key")**. This ensures that only Applications compiled with Secure Transport enabled can be loaded into the card.

Now the card contains a Secure Transport Key, and can be issued to customers. To load an Application into the card at a later time (and in a different place):

```
#Include Componnt.def
#Include LoadComm.def

#Include MK.DAT

Dir "\"

  Key "Master Key" Create=Never
    LCIndexedKey (ltCompileTime, 100)

  Key "Secure Transport Key" Ref=1 Create=Never
    LCBuildKey ("Master Key", 16,_
      LCSerialNumber (ltCompileTime))

  Call LCStartSecureTransport (1, AlgEaxAes128)

    Rem  Load the Application here...

  Call LCEndSecureTransport()

End Dir
```

This must be compiled with the card's Serial Number specified in the command line, with the parameter **–Nxxxxxxxxxxxxxxxx**. (The card's Serial Number is an 8-byte string, returned by the command **GET APPLICATION ID** with **P2=3** – see **8.9.10 The GET APPLICATION ID Command**.) Neither of the Keys is stored in the Image file. The Application's Files and Keys are stored in encrypted form, using a Key known only to the card issuer and the BasicCard, so the Image file can safely be sent to the customer, for example as an e-mail attachment.

5.6.2 Automatic File Authentication

The encryption algorithm used, **EAX**, also authenticates the data it encrypts. So the Secure Transport mechanism can be used to authenticate Files "for free". To do this, simply set

Usage = kuSecTrans, kuSign

when the Secure Transport Key is created. Then all downloaded Files will automatically be flagged as Signed by the Secure Transport Key. This means that the Access Control Rule

Signed ("Secure Transport Key")

will be satisfied whenever the signed Application is running.

5.7 Secure Messaging

Secure Messaging is the encryption or authentication of commands and responses. This is handled in the BasicCard family by the **START ENCRYPTION** and **END ENCRYPTION** commands. The MultiApplication BasicCard is no exception, but the command parameters are slightly different, due to the different way that Keys are represented. In a Terminal program or a single-application BasicCard, a Key is indexed by a key number from 0 to 255, and Secure Messaging is activated by

Call StartEncryption (P1=*key*, **P2=***algorithm*, **Rnd)**

if the encryption algorithm requires four bytes of initialisation data, or

Call ProEncryption (P1=*key*, **P2=***algorithm*, **Rnd, Rnd)**

if eight bytes are required (for Triple DES and AES-based algorithms). The Terminal program interpreter has access to the key, and automatically activates Secure Messaging when it sees the **START ENCRYPTION** command.

In the MultiApplication BasicCard, the following steps are required:

- find the CID of the Key from its name, using **FindComponent**;
- tell the Terminal program interpreter the value of the Key with the given CID, using **AddIndexedKey**;
- call the **START ENCRYPTION** command.

The following procedure, defined in Commands.def, performs the necessary steps:

```
Sub SMEncryptionByName (KeyName$, KeyVal$, Algorithm@)
```

If you know the CID, you can save time by calling the following procedure:

```
Sub SMEncryptionByCID (KeyCID%, KeyVal$, Algorithm@)
```

The source code for these procedures is available in Commands.def.

5.8 File Authentication

This section illustrates File Authentication using **OMAC** Message Authentication and **EC211** Elliptic Curve cryptography. It shows how to configure a card so that only authenticated files can be loaded as Applications, and how to authenticate an Application so that it can be loaded in such a card. The source files described here are available in the `BasicCardV8\Examples\AuthFile` directory.

OMAC authentication is faster than Elliptic Curve authentication, but Elliptic Curve authentication is more secure, as it doesn't require the Authentication Key to be stored in the BasicCard. The same is true of RSA authentication (which is available in the **ZC8-**series card only).

See **5.6.2 Automatic File Authentication** for another method of File Authentication (which, like **OMAC**, requires the Authentication Key to be stored in the BasicCard).

5.8.1 File Authentication Using OMAC

Suppose we decide to use the algorithm **AlgOmacAes128** (**OMAC** with **AES-128**) to authenticate files. For this we need a 16-byte Authentication Key, which we can generate with the KeyGen utility:

KeyGen –K1(16) OmacKey

This creates a file **OmacKey.bas** containing a 16-byte Key. The following source code in **OmacInit.bas** configures the BasicCard so that only files authenticated with this Key can be loaded:

```
Option Explicit
#Include Componnt.def
#Include OmacKey.bas

Dir "\"

  Key "Authentication Key" Lock=Never
    Usage=kuSign Algorithm=AlgOmacAes128
    LCIndexedKey (ltCompileTime, 1) ' Key(1) from OmacKey.bas

  ACR "Executable" Lock=Read:Always ' Special name "Executable"
    Signed ("Authentication Key")

End Dir
```

We can compile this and load it into a simulated BasicCard file **OmacCard.img** with the following commands:

ZCMBasic –CM –OI OmacInit
ZCMSim –C\BasicCardV8\MultiApp\ZC65_A.mcf –AOmacInit –D –WCOmacCard

Now we can create and authenticate a simple Application in **OmacApp.bas**:

```
Option Explicit
#Include Componnt.def
#Include LoadComm.def
#Include OmacKey.bas

Dir "\"

  Key "Authentication Key" Ref=100 Create=Never
    LCIndexedKey (ltCompileTime, 1)

  Application "MyApp" ' No Lock until file is authenticated
  Call LCAuthenticateFile (100, AlgOmacAes128, "MyApp")
  Application "MyApp" Create=Update Lock=Execute:Always

End Dir

Command &HA0 &H00 TestMyApp (S$)
  S$ = "TestMyApp"
End Command
```

The Application Loader doesn't need to know the value of "**Authentication Key**", so it is not stored in the Image file. (The compiler issues a warning whenever a Key that is used for File Authentication is also stored in the Image file.) Now we compile this Application and load it into **OmacCard.img**:

ZCMBasic –CM –OI OmacApp
ZCMSim –COmacCard –AOmacApp –D –WC

To check that everything has worked, the following Terminal program **AppTest.bas** selects Application "**MyApp**" and calls its command:

```
Option Explicit
#Include Commerr.def

Declare Command &HA0 &H00 TestMyApp (S$)

ResetCard : Call CheckSW1SW2()
Call SelectApplication ("MyApp") : Call CheckSW1SW2()

Private S$
Call TestMyApp (S$) : Call CheckSW1SW2()
Print S$ ' This should print "TestMyApp"
```

To compile and run this program:

ZCMBasic –OI AppTest
ZCMSim –COmacCard AppTest

This should print:

TestMyApp

5.8.2 File Authentication Using Elliptic Curve Cryptography

The **ZC6**-series MultiApplication BasicCard can authenticate Files with Elliptic Curve algorithms **EC167NR**, **EC167DSA**, **EC211NR**, and **EC211DSA**. It uses data hashing algorithm **SHA-1** with **EC167NR**, and **SHA-256** with **EC211NR**.

The **ZC8**-series MultiApplication BasicCard can authenticate Files with Elliptic Curve algorithms **EC167NR**, **EC167DSA**, **EC211NR**, **EC211DSA**, **EcpNR**, and **EcpDSA**. It can use any of the data hashing algorithms **SHA-1**, **SHA-224**, **SHA-256**, **SHA-384**, and **SHA-512**.

This section illustrates File Authentication using the Elliptic Curve algorithm **EC211**. ZeitControl provides five Elliptic Curves for use with this algorithm; we use Curve 3 for this project. First we use the KeyGen utility to generate a 27-byte Key, for use as our Private Key:

KeyGen –K1(27) EC211Key

This creates a file **EC211Key.bas** containing a 27-byte Key. The following source code in **EC211Init.bas** configures the BasicCard so that only files authenticated with this Key can be loaded. The card must be configured to load Curve 3 automatically whenever it is reset; in the **ZC6-**series card, this is done by creating the file "**ECDomainParams**" in the Root directory, and in the **ZC8-**series card, it is done with a **#Pragma** directive.

```
Option Explicit
#Include Componnt.def
#Include LoadComm.def

#Include Curves\EC211Crv.Str
#Include EC211Key.Bas ' EC211 Private Key

Dir "\"

  #If CardMajorVersion = 8

    Call LCWriteCardConfig (CardConfigECCurveName, "EC211-3")

  #Else

    File "ECDomainParams" Lock=Read:Always
      Rem  Use Curve 3, with 128 pre-computed points:
      #Include Curves\EC211-3.128

  #EndIf

  Rem  Let the compiler know the ECDomainParameters:
  Call LCEC211SetCurve (EC211Curve3String)
```

```
    Rem  The BasicCard needs the Public Key corresponding to
    Rem  the Private Key (Key(1)) in ECKey.bas:
    Key "ECPublicKey" Lock=Read:Always
      Usage=kuSign Algorithm=AlgEC211
      LCPublicKey (LCIndexedKey (ltCompileTime, 1), AlgEC211)

    ACR "Executable" Lock=Read:Always ' Special name "Executable"
      Signed ("ECPublicKey")

  End Dir
```

Only the Public Key is stored in the BasicCard; the Private Key is not required.

We can compile this and load it into a simulated BasicCard file **EC211Card.img** with the following commands:

For a **ZC6-**series card:

ZCMBasic –CM6 –OI EC211Init
ZCMSim –C\BasicCardV8\MultiApp\ZC65_A.mcf –AEC211Init –D –WCEC211Card

For a **ZC8-**series card:

ZCMBasic –CM8 –OI EC211Init
ZCMSim –C\BasicCardV8\MultiApp\ZC86_D.mcf –AEC211Init –D –WCEC211Card

Now we can create and authenticate a simple Application in **EC211App.bas**:

```
Option Explicit
#Include Componnt.def
#Include LOADCOMM.DEF

#Include Curves\EC211Crv.Str
#Include EC211Key.Bas ' EC211 Private Key

Dir "\"

  Call LCEC211SetCurve (EC211Curve3String)

  Key "ECPublicKey" Create=Never

  Application "MyApp" ' No Lock until file is authenticated
  Call LCAuthenticateFile ("ECPublicKey", AlgEC211,_
    LCIndexedKey (ltCompileTime, 1), "MyApp")
  Application "MyApp" Create=Update Lock=Execute:Always

End Dir

Command &HA0 &H00 TestMyApp (S$)
  S$ = "TestMyApp"
End Command
```

No Keys are stored in the Image file; the Private Key is only required by the compiler, and the Public Key is assumed to have already been created in the BasicCard. Now we compile this Application and load it into **EC211Card.img**:

ZCMBasic –CM –OI EC211App
ZCMSim –CEC211Card –AEC211App –D –WC

To check that everything has worked, use the **AppTest** program described in the previous section:

ZCMSim –CEC211Card AppTest

As before, this should print:

TestMyApp

The directory `BasicCardV8\Examples\AuthFile` also contains files **EC167Key.bas**, **EC167Init.bas**, and **EC167App.bas**, to illustrate File Authentication using the **EC167** algorithm.

5.9 Component Details

To use the procedures in the **COMPONENT** System Library (described in **7.4 The COMPONENT Library**), you need to know the internal structure of each Component type. This section describes these structures. Every Component type has attributes, and some Component types have data as well. The format of a Component's attributes depends not only on the Component type, but on whether the attributes are being created, written, or read. All the structures described below are declared as user-defined types in **Componnt.def**.

In the **COMPONENT** System Library, attributes are read and written as **String** parameters. Use type casting (*var* **As** *type* – see **3.11 Type Casting**) to convert to and from the relevant structure type. For example:

```
#Include Componnt.def

Function AcrType (CID%) As Byte

  Rem  User-defined type for reading the attributes of an ACR:
  Private Attr As AcrReadAttributes

  Rem  Read the attributes into Attr
  Attr As String = ReadComponentAttr (CID%)

  Rem  Now the attributes can be accessed as structure members:
  AcrType = Attr.AcrType@

End Function
```

5.9.1 Files

In the MultiApplication BasicCard, a File is just a Component of type **ctFile**. It can be accessed as a File, via the standard ZC-Basic file commands, or as a Component, via the **COMPONENT** System Library procedures.

File Attribute Format

The Attribute format depends on whether the Component is a Directory or a Data file.

For **CreateComponent**:

Offset	*Length*	*Directory*	*Data file*	
0	**2**	`ACRCID%`	`ACRCID%`	CID of Component's ACR
2	**1**	`Attributes@`	`Attributes@`	&H80 for Directory; 0 for Data file
3	**2**		`BlockLen%`	Length of allocation block

For **WriteComponentAttr**:

Offset	*Length*	*Directory*	*Data file*	
0	**2**	`ACRCID%`	`ACRCID%`	CID of Component's ACR

For **ReadComponentAttr**:

Offset	*Length*	*Directory*	*Data file*	
0	**2**	`ACRCID%`	`ACRCID%`	CID of Component's ACR
2	**1**	`Attributes@`	`Attributes@`	&H80 for Directory; 0 for Data file
3	**2**		`BlockLen%`	Length of allocation block
5	**2**		`FileLen%`	Length of file

Six corresponding user-defined types can be found in **Componnt.def**:

```
DirectoryCreateAttributes      DataFileCreateAttributes
DirectoryWriteAttributes       DataFileWriteAttributes
DirectoryReadAttributes        DataFileReadAttributes
```

File Data Format

File data can not be read or written using procedures from the Component System Library. The standard File I/O commands must be used instead.

5.9.2 ACRs

An Access Control Rule, or ACR, defines the access conditions for a Component. An ACR may have a name, or it may be anonymous.

Anonymous ACRs allow complex ACRs to be built in a single statement; the compiler and the Application Loader construct the necessary sub-components automatically. For example, the statement

File "ABC" Lock = Read: Always; Write: Write Once; Delete: Verify ("MyPassword")

in a Component Definition Section creates four Anonymous ACRs:

Always
Write Once
Verify ("MyPassword")
Read: Always; Write: Write Once; Delete: Verify ("MyPassword")

When an Anonymous ACR is created, the BasicCard looks for a match among its existing Anonymous ACRs. If a match is found, the existing ACR is used. This relies on the fact that an Anonymous ACR can never be overwritten or deleted. An Anonymous ACR must have an **ACRCID%** equal to zero.

ACR Attribute Format

For **CreateComponent** and **ReadComponentAttr**:

Offset	*Length*		
0	**2**	`ACRCID%`	CID of Component's ACR
2	**1**	`AcrType@`	As defined in *ACR Data Format* below

For **WriteComponentAttr**:

Offset	*Length*		
0	**2**	`ACRCID%`	CID of Component's ACR

Three corresponding user-defined types can be found in **Componnt.def**:

```
AcrCreateAttributes
AcrReadAttributes
AcrWriteAttributes
```

ACR Data Format

The format of ACR data depends on the ACR type, which is one of the following:

Type	*Name*	*Data*	*When satisfied*
&H01	**acrAlways**	None	Always
&H02	**acrNever**	None	Never
&H03	**acrAnd**	*ACR*, *ACR*,...	If all ACR's in the list are satisfied
&H04	**acrOr**	*ACR*, *ACR*,...	If at least one ACR in the list is satisfied
&H05	**acrQualified**	(*AT*,*ACR*), (*AT*,*ACR*),...	If the ACR corresponding to the current Access Type AT is satisfied
&H06	**acrNot**	*ACR*	If *ACR* is not satisfied
&H07	**acrIndirect**	*ACR*	If *ACR* is satisfied
&H10	**acrWriteOnce**	None	If the Component data field is empty
&H20	**acrVerify**	*Key*	If the **VERIFY** command has been called with *Key*
&H30	**acrExtAuth**	*Key*	If the **EXTERNAL AUTHENTICATE** command has been called with *Key*
&H40	**acrSMEnc**	*Key*	If the **START ENCRYPTION** command has been called with *Key* for an Encryption algorithm (**EAX, AES, DES**)

Type	*Name*	*Data*	*When satisfied*
&H50	**acrSMMac**	*Key*	If the **START ENCRYPTION** command has been called with *Key* for an Authentication algorithm (**OMAC**)
&H60	**acrPrivilege**	*Privilege*	If the current Application file (or the Terminal program for external access) has been granted the given *Privilege*
&H70	**acrFlag**	*Flag*	If the given *Flag* is set
&H80	**acrSigned**	*Key*	If the current Application file was signed using *Key*, in an **AUTHENTICATE FILE** command or during Secure Transport
&H90	**acrApp**	*File*	If *File* is the current Application
&HA0	**acrSecTrans**	*Key*	If Secure Transport with *Key* is active

ACR, *Key*, *Privilege*, and *Flag* parameters are two-byte CID's. *AT* is a one-byte Access Type from the following list (the constants are defined in **Componnt.def**):

&H01	**atRead**
&H02	**atWrite**
&H04	**atExecute**
&H08	**atDelete**
&H10	**atGrant**

The corresponding list in **5.5.5 ACR Definition** gives the definition syntax of these ACR types, for use in the Application Loader Definition Section.

5.9.3 Privileges

A Privilege is essentially just a name. It has no data, and its only attribute is its **ACRCID%**. The corresponding user-defined type `PrivilegeAttributes` can be found in **Componnt.def**.

5.9.4 Flags

A Flag can be either On or Off, and its value can be tested as an access condition in an ACR.

Flag Attribute Format

By default, a flag is cleared whenever the card is reset. The following attribute bits are defined in **Componnt.def**:

&H02	**faPermanent**	The flag retains its value when the card is reset or powered down.
&H04	**faClearOnNewApp**	The flag is cleared whenever an Application is selected.
&H08	**faClearOnCommand**	The flag is cleared whenever the card receives a command.

A Flag's attributes are the same for all library procedures:

Offset	*Length*		
0	**2**	`ACRCID%`	CID of Flag's ACR
2	**1**	`Attributes@`	The Flag's attribute bits

The corresponding user-defined type `FlagAttributes` can be found in **Componnt.def**.

Flag Data Format

The value of the Flag is stored in bit 0 of the **Attributes@** byte, but it can also be read or written as data, as follows:

- **CreateComponent** — The *data$* parameter must be empty; the initial value of the Flag is taken from the **Attributes@** byte.
- **WriteComponentAttr** — The new value of the Flag is taken from the **Attributes@** byte.
- **ReadComponentAttr** — The value of the Flag is not returned.
- **WriteComponentData** — The *data$* parameter contains a single byte. The Flag is set if this byte is non-zero.
- **ReadComponentData** — A string of length 1 is returned, equal to **Chr$**(0) or **Chr$**(1).

5.9.5 Keys

A Key has three configurable attributes in addition to its **ACRCID%**: a Key Usage Mask, an Algorithm Mask, and an Error Counter.

Key Usage Mask

In general, a cryptographic Key should only be used for a single purpose. In the MultiApplication BasicCard, each Key has a Key Usage Mask that specifies what the key can be used for. The Key Usage values **kuVerify** etc., defined in **Componnt.def**, have corresponding bitmasks, as follows:

Constant	*Value*	*Mask*	*Usage*
kuVerify	**1**	**&H0001**	**VERIFY** command
kuExtAuth	**2**	**&H0002**	**EXTERNAL AUTHENTICATE** command
kuSMEnc	**3**	**&H0004**	**START ENCRYPTION** with Encryption algorithm
kuSMMac	**4**	**&H0008**	**START ENCRYPTION** with Authentication algorithm
kuSign	**5**	**&H0010**	**AUTHENTICATE FILE**
kuIntAuth	**6**	**&H0020**	**INTERNAL AUTHENTICATE** command
kuSecTrans	**7**	**&H0040**	**SECURE TRANSPORT** command

Algorithm Mask

As well as the Key Usage mask, a Key has an Algorithm Mask that specifies the cryptographic algorithms that the key may be used for. The Algorithm IDs defined in AlgID.DEF have corresponding bitmasks, as follows:

Algorithm ID	*Value*	*Mask*	*Algorithm*
AlgSingleDesCrc	**&H23**	**&H0001**	Single **DES** with 8-byte key
AlgTripleDesEDE2Crc	**&H24**	**&H0002**	Triple **DES-EDE2** with 16-byte key
AlgTripleDesEDE3Crc	**&H25**	**&H2000**	Triple **DES-EDE3** with 24-byte key
AlgAes128	**&H31**	**&H0004**	**AES** with 16-byte key
AlgAes192	**&H32**	**&H0008**	**AES** with 24-byte key
AlgAes256	**&H33**	**&H0010**	**AES** with 32-byte key
AlgEaxAes128	**&H41**	**&H0020**	**EAX** using **AES** with 16-byte key
AlgEaxAes192	**&H42**	**&H0040**	**EAX** using **AES** with 24-byte key
AlgEaxAes256	**&H43**	**&H0080**	**EAX** using **AES** with 32-byte key
AlgOmacAes128	**&H81**	**&H0100**	**OMAC** using **AES** with 16-byte key
AlgOmacAes192	**&H82**	**&H0200**	**OMAC** using **AES** with 24-byte key
AlgOmacAes256	**&H83**	**&H0400**	**OMAC** using **AES** with 32-byte key
AlgEC167NR	**&HC1**	**&H0800**	**EC-167** (Nyberg-Rueppel)
AlgEC211NR	**&HC2**	**&H1000**	**EC-211** (Nyberg-Rueppel)
AlgEC167DSA	**&HC3**	**&H4000**	**EC-167** (Digital Signature Algorithm)
AlgEC211DSA	**&HC4**	**&H8000**	**EC-211** (Digital Signature Algorithm)
AlgECpNR	**&HE1**	**&H10000**	**EC-p** (Nyberg-Rueppel)
AlgECpDSA	**&HE2**	**&H20000**	**EC-p** (Digital Signature Algorithm)
AlgRSAPSS	**&HE3**	**&H40000**	**RSA** (Probabilistic Signature Scheme)
AlgRSAPKCS1	**&HE4**	**&H80000**	**RSA** (PKCS-1)
AlgSigHashSha224Mask	**&H0C**	**&H100000**	**SHA-224**
AlgSigHashSha256Mask	**&H10**	**&H200000**	**SHA-256**
AlgSigHashSha384Mask	**&H14**	**&H400000**	**SHA-384**
AlgSigHashSha512Mask	**&H18**	**&H800000**	**SHA-256**

Error Counter

To prevent attempts to guess the value of a Key by repetition, a Key should normally be configured with an Error Counter. This is a counter that is decremented by one each time the Key is unsuccessfully used in a cryptographic algorithm. If the counter reaches zero, the Key is disabled until it is reinstated via a **WriteComponentAttr** command. Whenever the Key is successfully used, its Error Counter is reset to the configured value; so the initial value of the Error Counter is the number of *consecutive* unsuccessful uses that are allowed before the Key is disabled.

If this Error Counter mechanism is not required, set **ECResetValue@** to zero, and the Key will never be disabled.

Key Attribute Format

The format of the *attr$* parameter is the same for all library procedures, but differs according to the BasicCard version. For the **ZC6-**series MultiApplication BasicCard:

Offset	*Length*		
0	**2**	`ACRCID%`	CID of Key's ACR
2	**2**	`UsageMask%`	The Key Usage Mask
4	**2**	`AlgorithmMask%`	The Algorithm Mask (2 bytes)
6	**1**	`ErrorCounter@`	The current value of the Error Counter
7	**1**	`ECResetValue@`	The Error Counter value after successful use

The corresponding user-defined type `KeyAttributes` can be found in **Componnt.def**.

For the **ZC8-**series MultiApplication BasicCard:

Offset	*Length*		
0	**2**	`ACRCID%`	CID of Key's ACR
2	**2**	`UsageMask%`	The Key Usage Mask
4	**4**	`AlgorithmMask&`	The Algorithm Mask (4 bytes)
8	**1**	`ErrorCounter@`	The current value of the Error Counter
9	**1**	`ECResetValue@`	The Error Counter value after successful use

The corresponding user-defined type `ZC8KeyAttributes` can be found in **Componnt.def**.

6. Support Software

This document describes Version 8.15 of the ZeitControl MultiDebugger software support package. All the software described in this chapter is available free of charge from our web site at www.BasicCard.com.

6.1 Hardware Requirements

No special hardware is required to develop programs in ZC-Basic – the support software can simulate the BasicCard inside your PC, so you can compile and test software on any system running Windows® XP or later.

Once the software is written and tested, you will need a PC/SC-compatible card reader, and one or more BasicCards. ZeitControl offers a selection of card readers – see our web site for details. A development kit containing CyberMouse reader, BasicCards, and printed documentation is available from ZeitControl – contact us at Sales@ZeitControl.de.

6.2 Installation

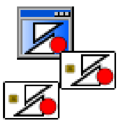
Please obtain the latest version of our development software before installing it. The latest version is available free of charge from our web site at www.BasicCard.com. Installation instructions can be found there.

To install the BasicCard software from the CD, run the program `BasicPro\Setup.exe`. The software is installed in the directory `C:\BasicCardV8` unless you specify a different destination.

6.3 File Types

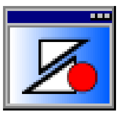
To use the development software effectively, it helps to have a clear idea of the roles played by the different types of files used by the system. We can arrange the files in a three-level hierarchy: *Project Files*, *Program Files*, and *Source Files*. There is a corresponding software hierarchy: development environment **BCDevEnv**; debuggers **ZCMDTerm/ZCMDCard**; and compiler **ZCMBasic**:

Level 1: Project Files

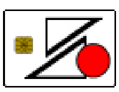


***.ZCP** Project Files
BCDevEnv.exe BasicCard Development Environment

Level 2: Program Files

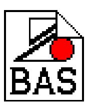


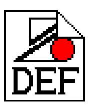
***.ZCT** Terminal Program Files
ZCMDTerm.exe ZeitControl Terminal Program Debugger



***.ZCC** BasicCard Program Files
ZCMDCard.exe ZeitControl BasicCard Program Debugger

Level 3: Source Files





***.BAS** ZC-Basic Source Files
***.DEF** ZC-Basic Definition Files
ZCMBasic.exe ZeitControl ZC-Basic Compiler

This hierarchy is not strictly enforced – you can run the debuggers outside the development environment if you just want to test a simple program; or you can compile a program from the Win32 console command line if you don't need to debug it.

***.ZCP** *Project Files*

A Project File simply lists all the Program Files that belong to a single project. What constitutes a project is up to you; the simplest projects contain one Terminal Program File and one BasicCard Program File, but bigger projects may contain two or three Terminal Program Files and a dozen or so BasicCard Program Files.

***.ZCT** *Terminal Program Files*

A Terminal Program File contains:

- compiler options for a Terminal Program, including Source File, Include Paths, and Pre-Defined Constants;
- run-time options, such as initial COM Port and Terminal Program command-line parameters;
- the positions of the various windows.

***.ZCC** *BasicCard Program Files*

A BasicCard Program File can be thought of as a Virtual BasicCard. It contains:

- compiler options for a BasicCard Program, including Source File (or multiple Source Files for a MultiApplication BasicCard), Card Type, Include Paths, and Pre-Defined Constants;
- the EEPROM contents of the Virtual BasicCard;
- the COM Port of the Virtual Card Reader that the program occupies;
- the positions of the various windows.

You can have more than one BasicCard Program File for a given source program, each with its own Virtual EEPROM. And you can run more than one **ZCMDCard** BasicCard Debugger at a time, as long as no two debuggers occupy the same Virtual Card Reader COM Port.

***.BAS** *and* ***.DEF** *ZC-Basic Source Files*

In our example programs, we make the distinction between .BAS files, which contain code, and .DEF files, which contain only definitions and declarations. This distinction is purely conventional; the compiler doesn't treat the two file types differently.

ZC-Basic Source Files are described in **Chapter Error: Reference source not found: Error: Reference source not found**.

In addition, the **ZCMBasic** Compiler produces the following two file types as output (among others – see **6.9.1 The ZC-Basic Compiler ZCMBasic.** for details):

***.IMG** *Image Files*

An Image File contains a compiled Terminal Program or BasicCard Program, with no symbolic debug information. Its contents are described in **11.1 ZeitControl Image File Format**. Two command-line programs accept Image Files as input (and Debug Files too, if the **.DBG** file extension is explicitly given):

- the **ZCMSim** P-Code Interpreter, which requires a Terminal Program Image File, and optionally one or more BasicCard Program Image Files;
- the **BCLoad** Download Program, which downloads a BasicCard Image File to a BasicCard.

See **6.9.2 The P-Code Interpreter ZCM** and **6.9.3 The Card Loader BCLoad.** for details.

***.DBG** *Debug Files*

A Debug File contains all the information in an Image File, plus symbolic debug information for the debuggers **ZCMDTerm** and **ZCMDCard**. Its contents are described in **11.2 ZeitControl Debug File Format**.

6.4 Physical and Virtual Card Readers

Whenever you access a BasicCard or a Card Reader from a ZC-Basic Terminal Program, ZeitControl's P-Code Interpreter uses the current value of the **ComPort** variable to determines where to look for the Card Reader. The meaning of the **ComPort** variable depends on the program that contains the P-Code Interpreter: this can be an executable file, the **ZCMSim** P-Code Interpreter, or the **ZCMDTerm** Terminal Program Debugger.

6.4.1 ComPort in an Executable File

A ZC-Basic program compiled into an executable file accepts the following values for the **ComPort** variable:

1 ≤ **ComPort** ≤ 4:	Physical Card Reader on serial port COM1-COM4
100 ≤ **ComPort** ≤ 199:	PC/SC Card Reader – see **3.22.4 PC/SC Functions**
201 ≤ **ComPort** ≤ 204:	Virtual Card Reader running in the **ZCMDCard** debugger
251 ≤ **ComPort** ≤ 254:	Virtual Contactless Reader running in the **ZCMDCard** debugger

6.4.2 ComPort in the ZCMSim P-Code Interpreter

The **ZCMSim** P-Code Interpreter accepts the same values for the **ComPort** variable as an executable file, as listed in the previous section. In addition, **ComPort** may be set to any of the **–P** parameters specified on the command line, in which case the corresponding simulated BasicCard is accessed – see **6.9.2 The P-Code Interpreter ZCMSim.exe**.

6.4.3 ComPort in the ZCMDTerm Terminal Program Debugger

The **ZCMDTerm** Terminal Program Debugger accepts the following values for the **ComPort** variable:

1 ≤ **ComPort** ≤ 4:	Physical or Virtual Card Reader
100 ≤ **ComPort** ≤ 199:	PC/SC Card Reader – see **3.22.4 PC/SC Functions**
201 ≤ **ComPort** ≤ 204:	Virtual Card Reader running in the **ZCMDCard** debugger
251 ≤ **ComPort** ≤ 254:	Virtual Contactless Reader running in the **ZCMDCard** debugger

If 1 ≤ **ComPort** ≤ 4, then **ZCMDTerm** has to decide whether to access a Physical or a Virtual Card Reader. It does this on the basis of the settings in the **Setting | Terminal Programs... | Card Reader Options** dialog box section. In this dialog box, each of COM1 through COM4 can be set to **Real**, **Auto**, or **Virtual**:

Real	Physical Card Reader is accessed
Auto	Virtual Card Reader if available, otherwise Physical Card Reader
Virtual	Virtual Card Reader running in the **ZCMDCard** debugger

To enable communication between the Terminal Program and a BasicCard program running in the **ZCMDCard** BasicCard Program debugger, the **ZCMDCard** debugger must know which COM Port to attach to. You can specify this in one of two ways:

- in **ZCMDCard**, via the **Car | Insert in Virtual Reader...** dialog box;
- in **ZCMDCard** or **BCDevEnv**, via the **Setting | BasicCard Program... | Virtual Reader** dialog box.

The first of these is temporary; the second is permanent for the given BasicCard Program File.

6.5 Windows®-Based Software

The Windows®-based software consists of the following programs:

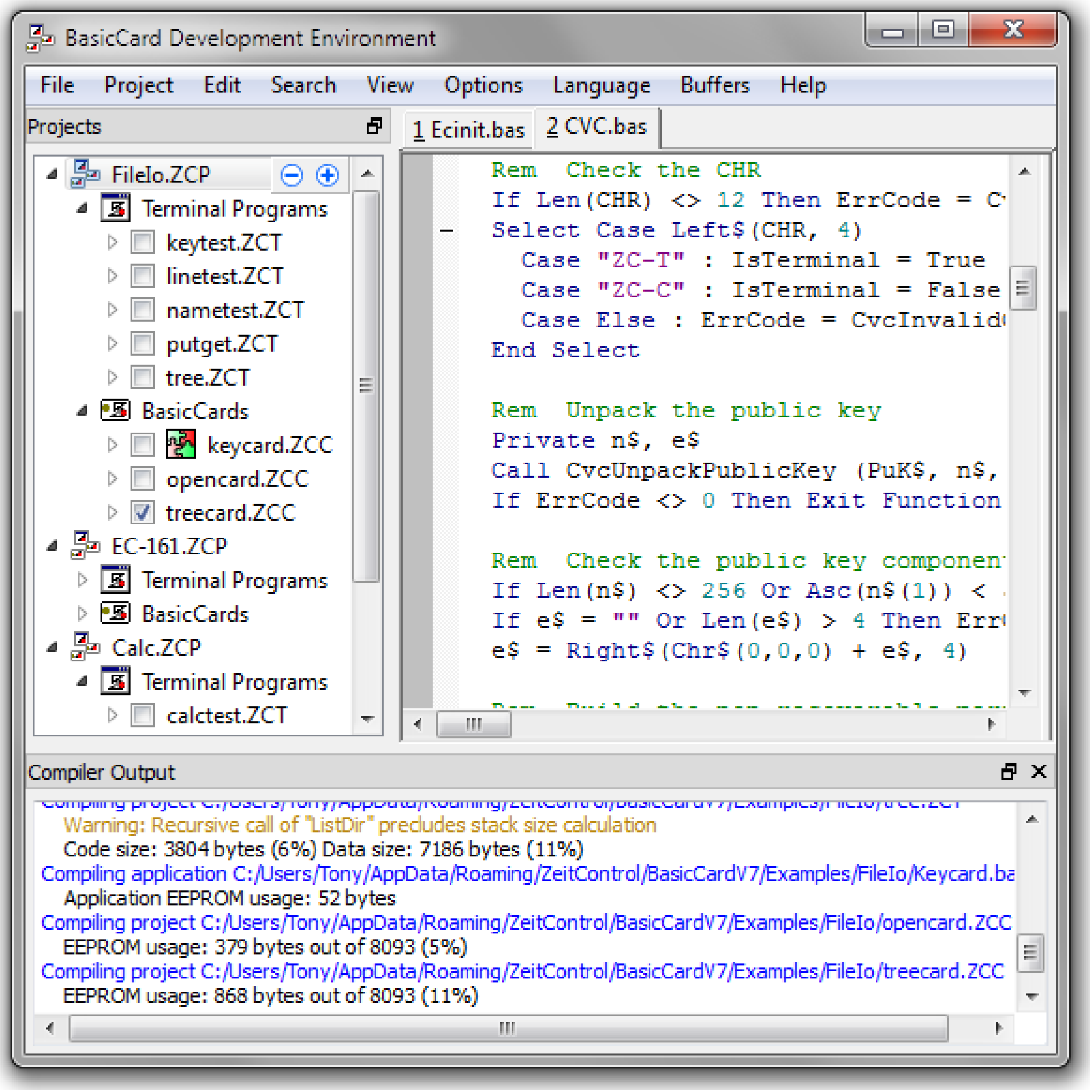
- **BCDevEnv**, the BasicCard Development Environment. This program manages projects, creating and maintaining ZeitControl Project files, with **.ZCP** extension. It also contains a built-in text editor, the SciTE editor (details available from http://www.scintilla.org/SciTE.html).
- **ZCMDTerm**, a source-level symbolic debugger for Terminal programs. It can communicate with one or more **ZCMDCard** debuggers, and one or more physical card readers. It uses ZeitControl Terminal Program files, with **.ZCT** extension, to store the information that it needs to compile and run Terminal Programs.
- **ZCMDCard**, a source-level symbolic debugger for BasicCard programs. It waits for commands from the Terminal debugger **ZCMDTerm**, executes the commands under the control of the user, and sends its responses back to the Terminal debugger. It can also download BasicCard programs to a real BasicCard. It uses ZeitControl BasicCard Program files, with **.ZCC** extension, to store the information that it needs to compile and run BasicCard Programs.

6.6 The BCDevEnv BasicCard Development Environment

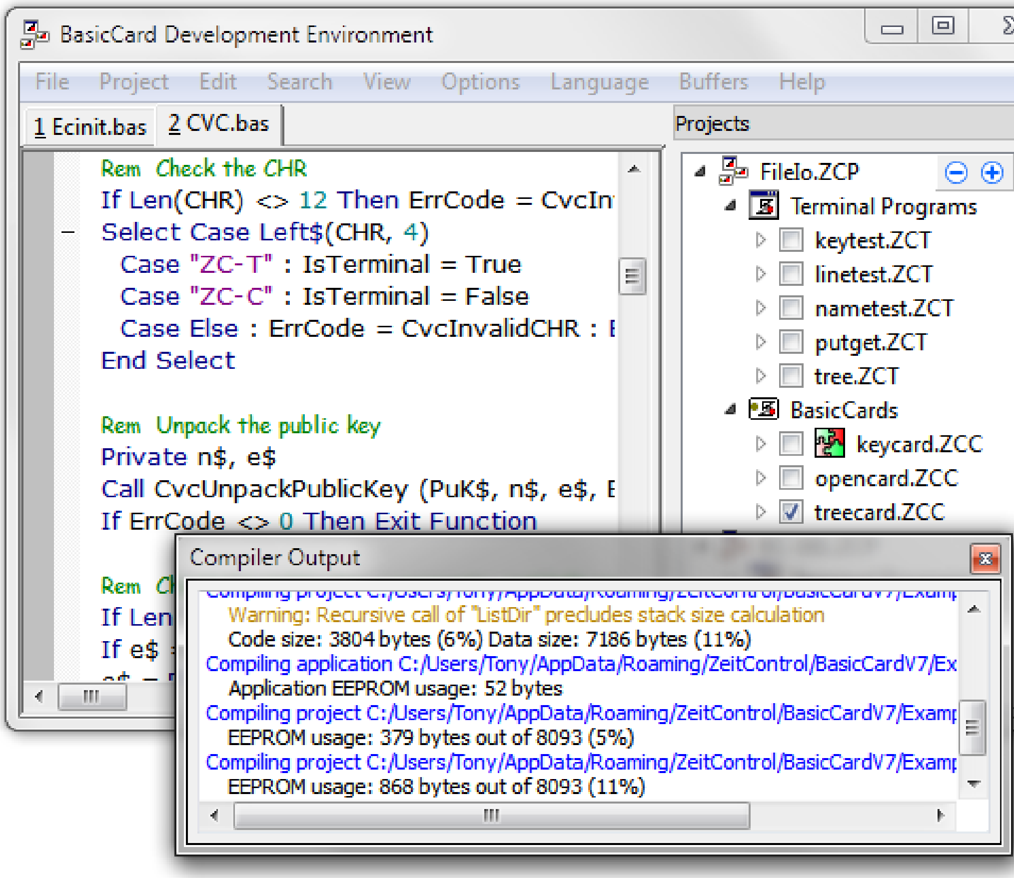
The **BCDevEnv** BasicCard Development Environment program manages projects, creating and maintaining ZeitControl Project files, with **.ZCP** extension. It also contains a built-in SciTE text editor.

The main window contains three child windows:

- the **Projects** window displays a list of projects, each of which contains a set of Terminal programs and a set of BasicCard programs;
- the **Compiler Output** window displays the results of compilations;
- the **SciTE Editor** window contains the SciTE text editor.



The **Projects** and **Compiler Output** windows are dockable – they can be picked up with the mouse and dragged to different positions, or detached from the main window:



The **Compiler Output** window can be opened and closed from the **View** menu, but the **Projects** window is always visible.

6.6.1 The Projects Window

You can use the **Project** menu to add a new or existing project to this window. Or you can add an example project from the **Help | Example Projects** menu. A project contains Terminal Programs and BasicCards, which in turn contain source files; all these are displayed in a tree structure. Clicking the checkbox next to a Terminal Program or BasicCard item adds it to the list of programs that are started when **Start Checked** is clicked in the context-sensitive menu.

The 'minus' and 'plus' buttons  collapse or expand the whole **Projects** hierarchy by one level.

The **Projects** window uses the following icons to indicate the type of an item:

 BCDevEnv project

 All Terminal programs belonging to a project

 All BasicCards belonging to a project

 MultiApplication BasicCard

 Application in a MultiApplication BasicCard

Right-clicking on any item in the tree brings up the context-sensitive menu. The contents of this menu depend on the type of item that is clicked. It contains some of the following items:

Settings

Open the **Project Settings** dialog box – see **6.6.4 The Project Settings Dialog Box**.

Add New Terminal Program / Add New BasicCard

Add a new Terminal Program or BasicCard to the project. First you are prompted for the name of the new project file, then you are taken to the **Project Settings** dialog box to configure the new program.

Add Existing Terminal Program... / Add Existing BasicCard...

Add an existing Terminal Program (from a .ZCT file) or BasicCard (from a .ZCC file) to the project.

Build / Build All / Build All Terminal Programs / Build All BasicCards / Build All Applications

Compile the program or programs, but only if they are out of date.

Rebuild / Rebuild All / Rebuild All Terminal Programs / Rebuild All BasicCards / Rebuild All Applications

Compile the program or programs, even if they are up to date.

Start / Start All / Start All Terminal Programs / Start All BasicCards

Start the program or programs in a **ZCMDTerm** Terminal debugger or a **ZCMDCard** BasicCard debugger. An item that has been started by **BCDevEnv** in this way is displayed in red. Another way to start a single item is to double-click on it.

Start Checked

Start all the checked items (those whose check-box has been clicked).

Stop / Stop All / Stop All Terminal Programs / Stop All BasicCards

Stop the program or programs displayed in red.

Note: This is not guaranteed to succeed – to stop a debugger, **BCDevEnv** sends a **WM_CLOSE** message to its main window, and if the settings have changed, the debugger will open a Save/Discard/Cancel dialog box to ask the user what to do. If the user selects Cancel, the debugger will not close.

Remove Project

Remove the project from the **Projects** window. You are given the option of deleting the .ZCP project file permanently.

Remove from Project

Remove the program from its parent project. You are given the option of deleting the .ZCT or .ZCC program file permanently.

Remove from Card

Remove an application from a MultiApplication BasicCard.

Edit

Open a source file in the **SciTE Editor** window. Another way to open a source file for editing is to double-click on it.

Show in Explorer

Show the file in Windows Explorer.

6.6.2 The Compiler Output Window

This window displays the results of all **Build** and **Rebuild** commands. Warning messages are displayed in light brown, and error messages in red. If a warning or error message contains a filename and/or a line number, then clicking on the message will open the file in the **SciTE Editor** window at the given line number.

Right click in this window to open a menu:

Copy	Copy the selected text to the clipboard
Select All	Select the contents of the window
Clear	Clear the contents of the window
Save to File	Save the contents of the window to a text file

6.6.3 The SciTE Editor Window

The **SciTE Editor** window contains an embedded copy of SciTE (the **Sci**ntilla **T**ext **E**ditor). This editor will be familiar to many users. It has been retained almost unchanged in **BCDevEnv**, so that regular users of SciTE can use it as is. The only significant differences are:

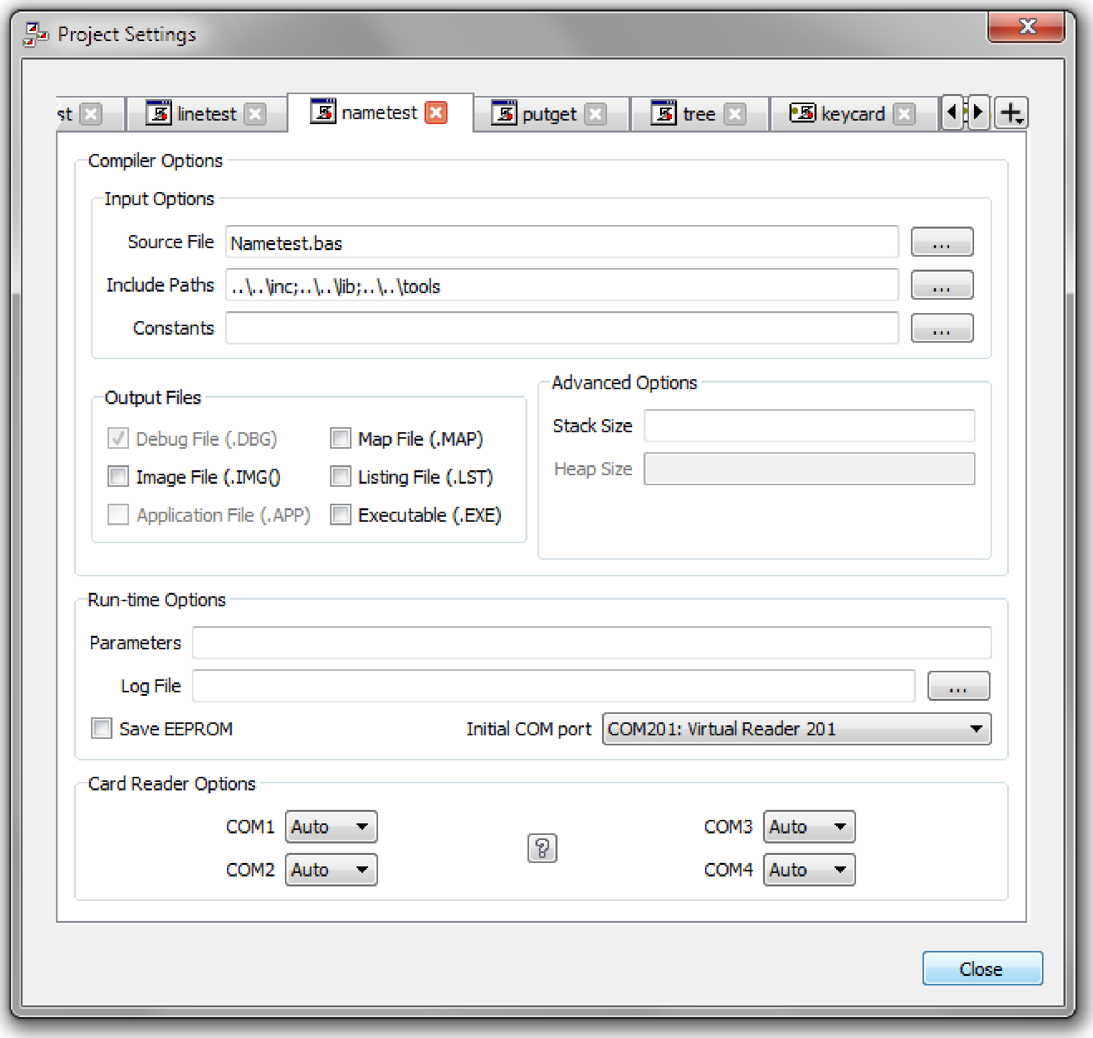
- The ZC-Basic language is supported. Syntax highlighting, folding, auto-completion (Ctrl-Space), and calltips (on open left parenthesis) are implemented. Ctrl-Z cancels an unwanted auto-complete.
- The **Tools** menu has been removed.

To learn about the SciTE editor, select menu item **Help | SciTE Editor Help**. This displays SciTE's own help page.

6.6.4 The Project Settings Dialog Box

This dialog box lets you configure all the programs that belong to a project. To activate it, select **Settings** from the **Projects** window context-sensitive menu (see **6.6.1 The Projects Window**). To add a new program to the project, click on the 'plus' button at top right.

This dialog box cannot run if any of its constituent programs are running in a debugger. In this case, **BCDevEnv** asks you whether it should stop the running programs, or cancel the dialog box invocation.



This dialog box combines the **Terminal Settings** and the **Card Settings** dialog boxes. See **6.7.6 The Terminal Settings Dialog Box** and **6.8.2 The Card Settings Dialog Box** for more information.

6.6.5 BCDevEnv Menus

Most of the menus belong to the SciTE editor. You can access the SciTE documentation via the **Help | SciTE Editor Help** menu item. Two menus are specific to **BCDevEnv**:

The **Project** menu contains the following items:

New Project	Add a new project to the **Projects** window
Open Project...	Add an existing project to the **Projects** window
Remove All	Remove all projects from the **Projects** window
Refresh	Bring the **Projects** window up to date
Stop All	Stop all running debuggers

The **Help** menu contains the following items:

BasicCard Manual	Display this manual in your PDF file viewer (for instance, Adobe Reader)
Example Projects ▸	Show a list of example projects that can be opened in the **Projects** window
SciTE Editor Help	Display the SciTE editor documentation in your web browser
About BCDevEnv	Display the version number of the running **BCDevEnv** program
About Qt	Display information about **Qt**, the cross-platform development tool from Nokia that was used to implement ZeitControl's software
About SciTE	Display information about the SciTE editor

The **Options** menu contains one item that is specific to **BCDevEnv**:

Environment...

Set the following Registry variables in **HKEY_CURRENT_USER\Software\ZeitControl\BasicCardV8**:

ZCPORT	The initial value of **ComPort** variable in Terminal programs
ZCINC	The directories searched by the **ZCMBasic** compiler for included files
ZCZOOM	The default text size in all programs. Initially, text size is 8pt.

6.7 The ZCMDTerm Terminal Program Debugger

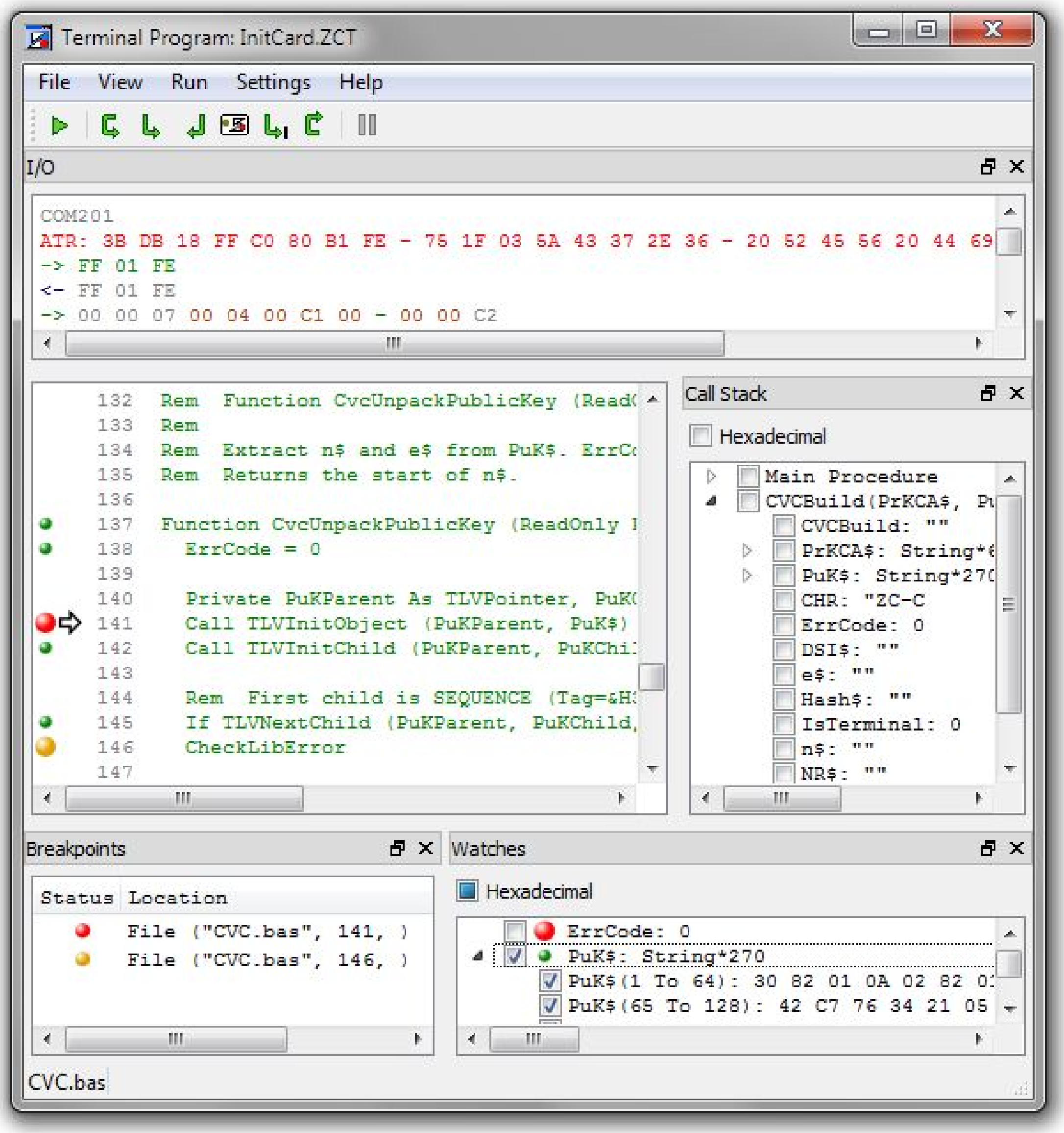
The **ZCMDTerm** ZeitControl Terminal Program Debugger is a source-level symbolic debugger for Terminal programs. It can communicate with one or more **ZCMDCard** debuggers, and one or more physical card readers. It uses ZeitControl Terminal Program files, with **.ZCT** extension, to store the information that it needs to compile and run Terminal Programs.

The main window consists of a **Source** window, four dockable child windows, and three floating windows.

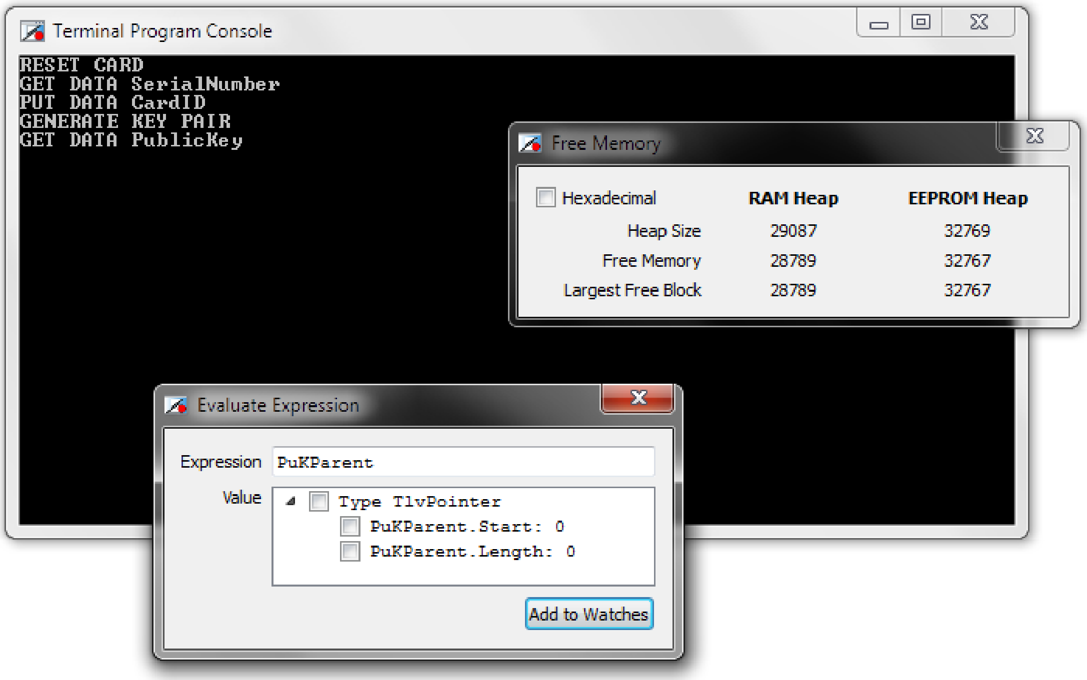
The dockable child windows can be opened and closed from the **View** menu, or detached from the main window:

- the **I/O** window displays communication between Terminal program and BasicCards;
- the **Call Stack** window displays the current call stack along with parameters and local variables;
- the **Breakpoints** window displays user-specified breakpoints;
- the **Watches** window displays values of user-specified data items.

These windows are described in more detail below.



- The **Console** window displays the console output from the Terminal program. This window is displayed whenever a Terminal program is loaded. Closing this window (by clicking on the Close icon at the top right) will close the **ZCMDTerm** Terminal debugger.
- The **Evaluate Expression** window evaluates an expression entered by the user. Clicking the checkbox next to an item toggles between decimal and hexadecimal display for that item and all its sub-items. This window can be displayed in four ways: from the **Run | Evaluate...** menu item; by pressing the F10 shortcut key; by selecting an expression in the **Source Window** with the mouse or the keyboard; or by double-clicking on a variable name in the **Source Window**.
- The **Free Memory** window displays the amount of memory available in the various heaps. This window can be opened and closed from the **View** menu.



6.7.1 The ZCMDTerm Source Window

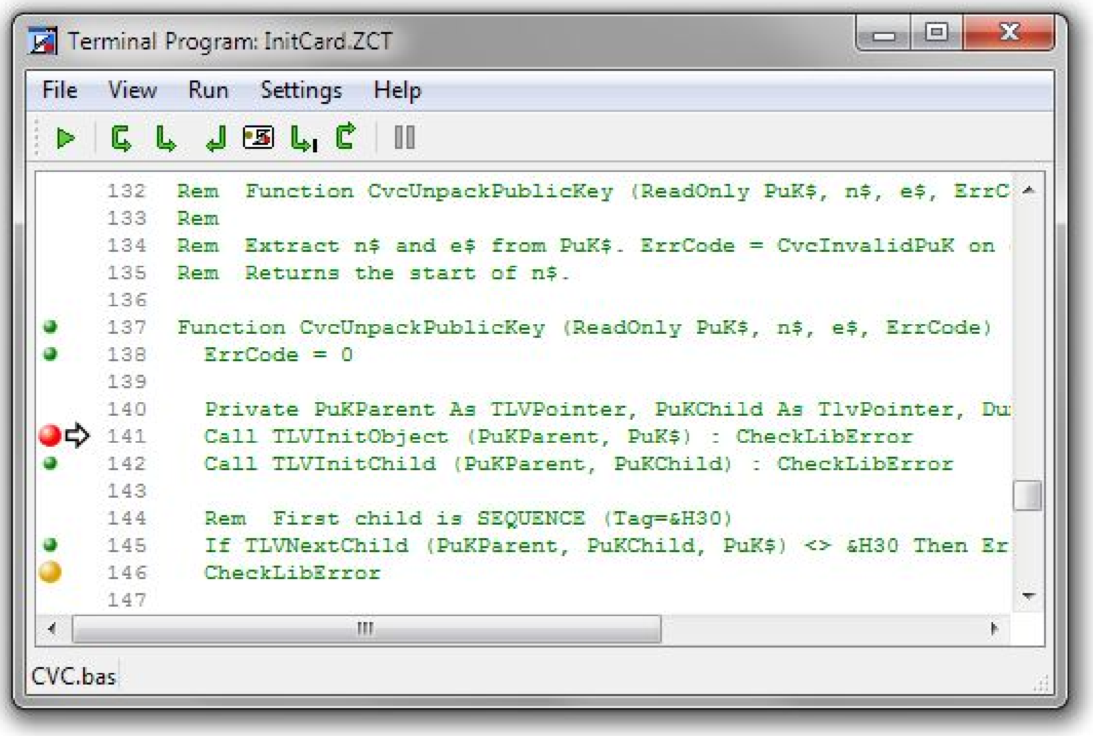
The **Source** window displays the source code of the Terminal program, and functions as a source-level debugger. A screenshot is on the next page.

The numbers in grey are line numbers. To the right of these is the source code, in green; to the left is a column of breakpoint markers. Green markers shows source lines that contain executable code; clicking on the marker sets a breakpoint there. Red markers show enabled breakpoints, and yellow markers show disabled breakpoints; right-click on a breakpoint marker to enable or disable it. The arrow at line 141 shows the current execution point.

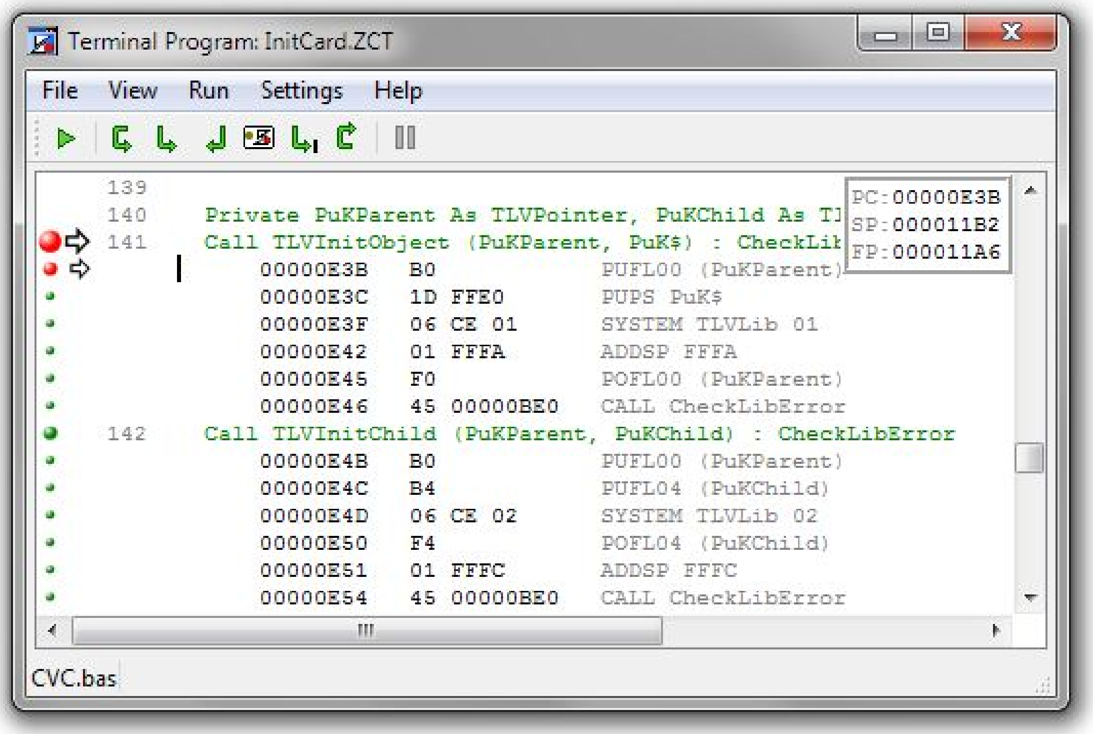
The icons in the toolbar read, from left to right:

Run F9	Run until program exit, or a breakpoint is reached
Step Over F8	Execute one source line, stepping over any procedure call
Step Into F7	Execute one source line, stepping into any procedure call
Step Out Of F6	Return to the calling procedure
Step to Card F5	Jump to the **ZCMDCard** BasicCard debugger
Run to Cursor F4	Run to the current cursor position
Restart F3	Restart the Terminal program
Pause F2	Pause whichever debugger is currently running

F9 through F2 are shortcut keys. These actions are also available from the **Run** menu.



Selecting the **View | P-Code** menu item displays the program as interleaved source lines and P-Code instructions:

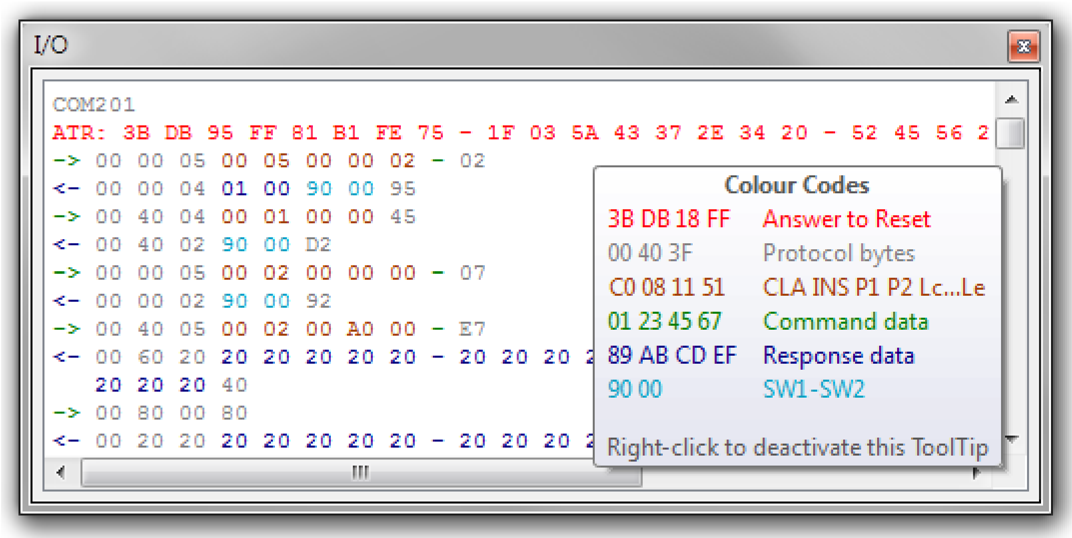


Breakpoints can be set on individual P-Code instructions in this mode.

The small arrow points to the current P-Code instruction. The box in the top right corner of the **Source** window is the **P-Code Register** window, which shows the contents of the three P-Code registers in hexadecimal. By default, this window is shown whenever P-Code is displayed, but you can change this behaviour in the **View | Registers ▸** sub-menu.

6.7.2 The ZCMDTerm I/O Window

The **I/O** window displays all communication between the Terminal program and BasicCards. The **I/O** window can be opened or closed from the **View** menu.



The data is displayed in hexadecimal, colour-coded according to its role in the current protocol. Move the mouse into the window to display the **Colour Codes** tooltip, which tells you the meaning of each colour. (You can enable and disable this tooltip in the context-sensitive menu.)

Right-click to bring up the context-sensitive menu:

Copy	Copy the selected text to the clipboard
Select All	Select the contents of the window
Clear	Clear the contents of the window
Save to File	Save the contents of the window to a text file
Deactivate ToolTip	Deactivate the **Colour Codes** tooltip

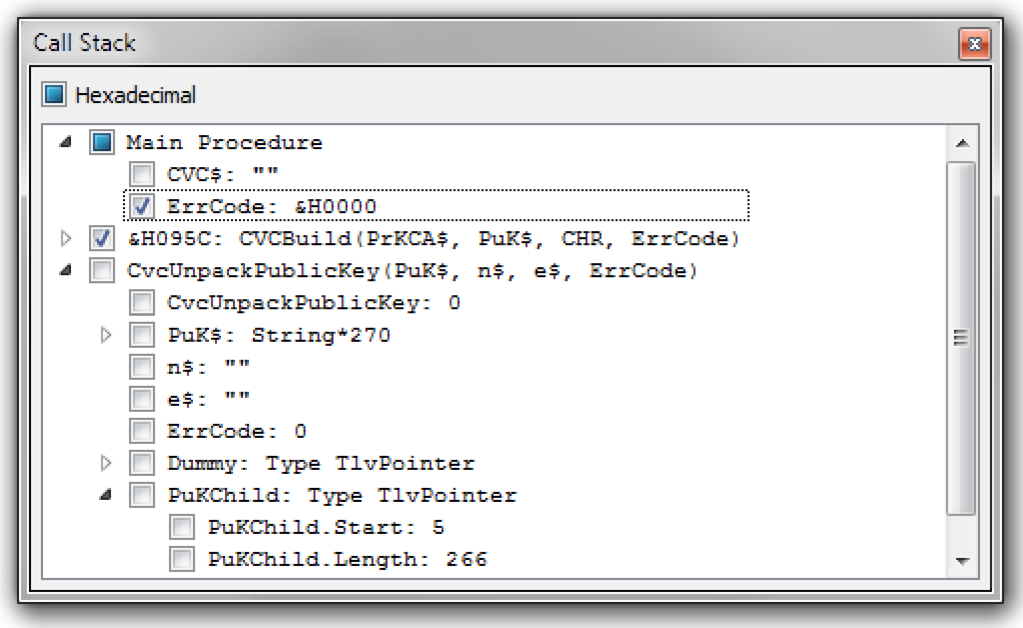
For performance reasons, the **I/O** window only displays the last 32 kilobytes of traffic. If you need a complete record of all traffic, specify a Log File in the **Terminal Settings** dialog box (see **6.7.6 The Terminal Settings Dialog Box**).

6.7.3 The ZCMDTerm Call Stack Window

The **Call Stack** window displays a list of the procedures in the call stack, from the initialisation code to the currently executing procedure. For each procedure in the call stack, it displays the parameters and variables of the procedure. Double-click on a procedure name (or click the Expand Sub-tree icon next to it) to show the parameters and variables.

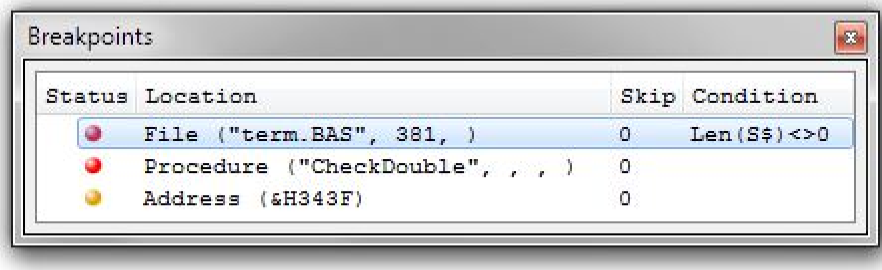
Compound types (arrays and user-defined types) are displayed in a tree structure. Clicking on the checkbox next to an item toggles between decimal and hexadecimal for that item and all its sub-items. If some (but not all) of an item's sub-items are checked, the item's checkbox indicates an intermediate state.

The **Call Stack** window can be opened or closed from the **View** menu. It can also be opened with the **View | Local Variables** menu item, which displays it at the currently executing procedure.



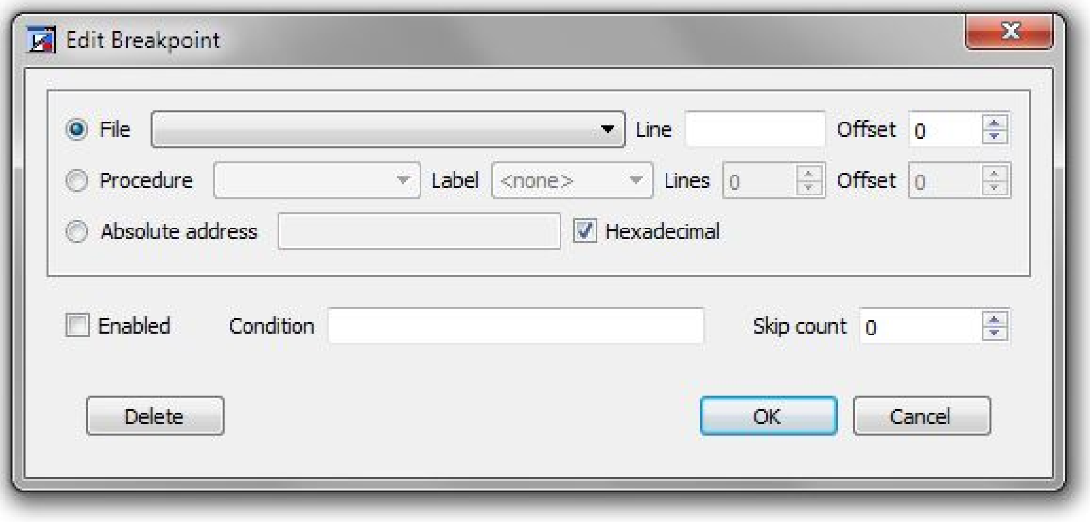
6.7.4 The ZCMDTerm Breakpoints Window

The **Breakpoints** window lists all user-specified breakpoints. On program exit, these breakpoints are saved in the Terminal program .ZCT file, to be activated the next time the program is run.



- The red marker denotes an enabled breakpoint, and the yellow marker denotes a disabled breakpoint. A disabled breakpoint is always skipped.
- The **Skip** value specifies the number of times an enabled breakpoint is skipped over before the debugger halts; it defaults to zero.
- The **Condition**, if present, sets the condition that has to be met for the debugger to halt at the breakpoint if it is enabled. The debugger will halt if either the condition is **True** or it could not be evaluated.

To add a breakpoint, either click on the breakpoint column in the **Source** window, or right-click in the **Breakpoints** window and select **Add...** The second method brings up the **Edit Breakpoint** window:



A breakpoint can be specified in one of three ways:

- **File**: Source file and line number

  The optional *Offset* parameter specifies the offset of the breakpoint instruction within the line.

- **Procedure**: Procedure name

  Optional *parameters* are:
  - *Label* The name of a label in the procedure, user-defined or compiler-generated
  - *Lines* The number of lines between the label and the breakpoint
  - *Offset* Offset of the breakpoint instruction within the line

- **Absolute address**: The address of the breakpoint instruction

When a breakpoint is added, the debugger will automatically select the most economical representation, but you can override this choice. Note that the choce is immaterial until the source file changes and the program is re-compiled.

After a breakpoint has been added to the list, it can be edited by right-clicking on its marker in the **Source** window, or by double-clicking on it in the **Breakpoints** window.

6.7.5 The ZCMDTerm Watches Window

The **Watches** window displays the values of user-entered expressions. On program exit, these expressions are saved in the Terminal program .ZCT file, to be displayed the next time the program is run. The **Watches** window can be opened or closed from the **View** menu.

To add a new watch expression:

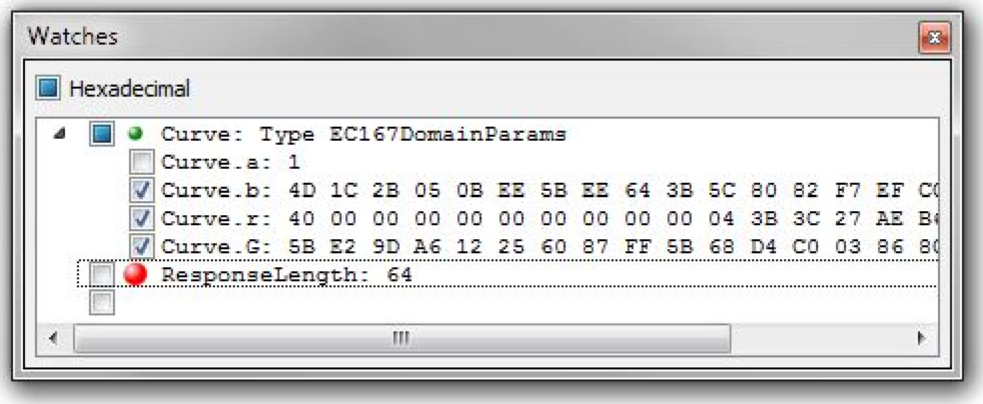
- click **Add to Watches** in the **Evaluate Expression** window; or
- move the cursor to the last line, and enter an expression in the edit box.

To edit an expression:

- double-click on the expression; or
- select the expression with the up- or down-arrow key, and press Enter.

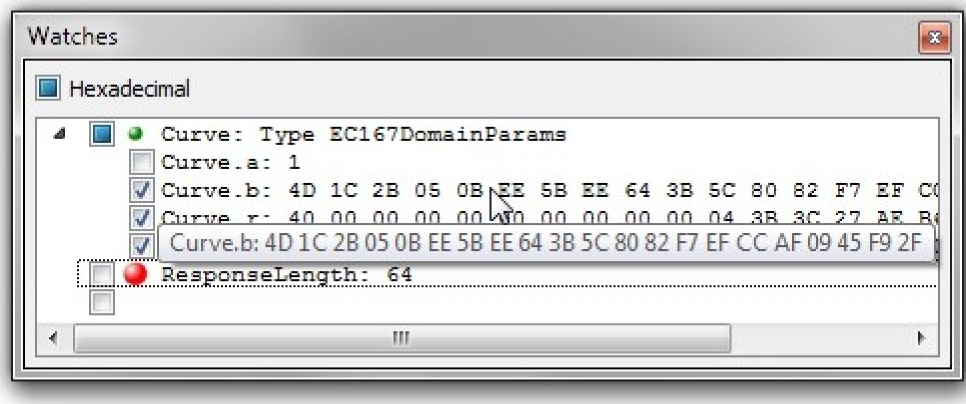
Only top-level items can be edited.

You can copy all or part of the **Watches** window to the clipboard by right-clicking to bring up a menu.



A green or red marker is displayed next to each top-level item. A red marker indicates that the item is a *Watchpoint*, which means that the program will stop if it detects a change in the value of the item (or any of its sub-items). Click on this marker to toggle it from red to greeen.

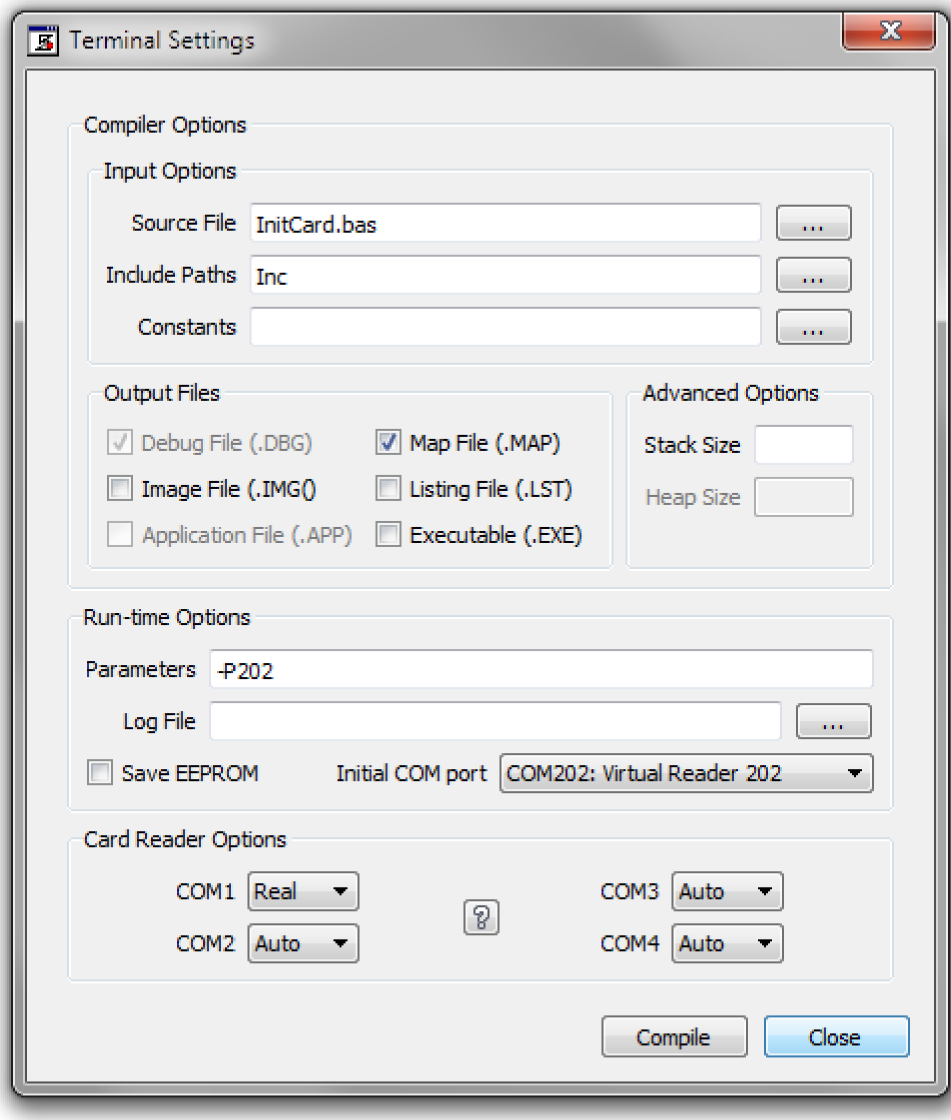
If an item is too long to fit in the window, just move the mouse cursor to it, and a tooltip will appear containing the whole of the item (you might need to click on the window first, to transfer the focus):



Like the **Call Stack** window, an item and all its sub-items can be displayed in hexadecimal by clicking on the item's checkbox.

6.7.6 The Terminal Settings Dialog Box

This dialog box lets you configure all the settings for a Terminal program. Select the **Settings | Terminal Program...** menu item to open:



Items in the **Compiler Options** section correspond to command-line options in the **ZCMBasic** compiler – see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**:

Source File	The main source file of the Terminal program
Include Paths	The –**I** parameter: search path for included files
Constants	The –**D** parameter: pre-defined constants
Output Files	The –**O** parameter: which output files to generate
Stack Size	The –**S** parameter: the size of the Terminal program P-Code stack

Include Paths and **Constants** can contain multiple entries separated by semi-colons; or you can click on the 'ellipsis' button ... to type each entry on a separate line.

Items in the **Run-time Options** section:

Parameters	Command-line parameters passed to the Terminal program in the **Param$** array
Log File	Log file of all communication between the Terminal program and BasicCards
Save EEPROM	Save EEPROM back to the .DBG file – see **2.2.4 Permanent Data**
Initial COM port	The initial value of the **ComPort** variable

The **Initial COM port** value can also be set via the **Settings | COM Port ▸** menu item.

The **Card Reader Options** section tells the Terminal program how to look for card readers on COM1 through COM4 – see **6.4.3 ComPort in the ZCMDTerm Terminal Program Debugger**.

All these settings are saved in the .ZCT file on program exit.

6.7.7 ZCMDTerm Menus

The **File** menu contains the following items:

New Terminal Program...	Create a new Terminal Program File
Open Terminal Program...	Open an existing Terminal Program File
Save	Save the current Terminal Program File
Save As...	Save the current Terminal Program File under a new name
Edit *current file*	Edit the file showing in the **Source** window
Edit...	Edit a text file in the **BCDevEnv** Professional Development Environment
Edit Source ▸	Edit a source file from the current Terminal Program
Compile...	Open the **Terminal Settings** dialog box
Exit	Exit the **ZCMDTerm** program

The **View** menu contains the following items:

Source File ▸	Display a selected source file in the **Source** window
Procedure ▸	Display a selected ZC-Basic procedure in the **Source** window
Execution Point	Display the code at the current PC
Breakpoints	Open the **Breakpoints** window for viewing and editing breakpoints
Watches	Open the **Watches** window for monitoring program data
Local Variables	View the current procedure and its local data in the **Call Stack** window
Call Stack	View all active procedures and their local data in the **Call Stack** window
I/O	Open the **I/O** window to show I/O between Terminal and BasicCard
Free Memory	Display the **Free Memory** window showing free space in the heap
P-Code	Display P-Code instructions and registers in the **Source** window
Registers ▸	Specify when the **P-Code Register** window is shown
Zoom ▸	Adjust the text size of all windows
Run/Step toolbar	Show or hide the toolbar containing the green Run/Step icons

The **Run** menu contains the following items:

Run	Run until program exit, or a breakpoint is reached
Step Over	Execute one source line, stepping over any procedure call
Step Into	Execute one source line, stepping into any procedure call
Step Out Of	Return to the calling procedure
Step to Card	Jump to the **ZCMDCard** BasicCard debugger
Run to Cursor	Run to the current cursor position
Restart	Restart the Terminal program
Pause	Pause whichever debugger is currently running
Evaluate...	Evaluate an expression in the Evaluate window

Most of these items are also available as toolbar icons.

The **Settings** menu contains the following items:

COM Port	Set the initial value of the **ComPort** variable
Terminal Program...	Open the **Terminal Settings** dialog box
Environment...	Set Windows® Registry variables **ZCPORT**, **ZCINC**, and **ZCZOOM**

The **Help** menu contains the following items:

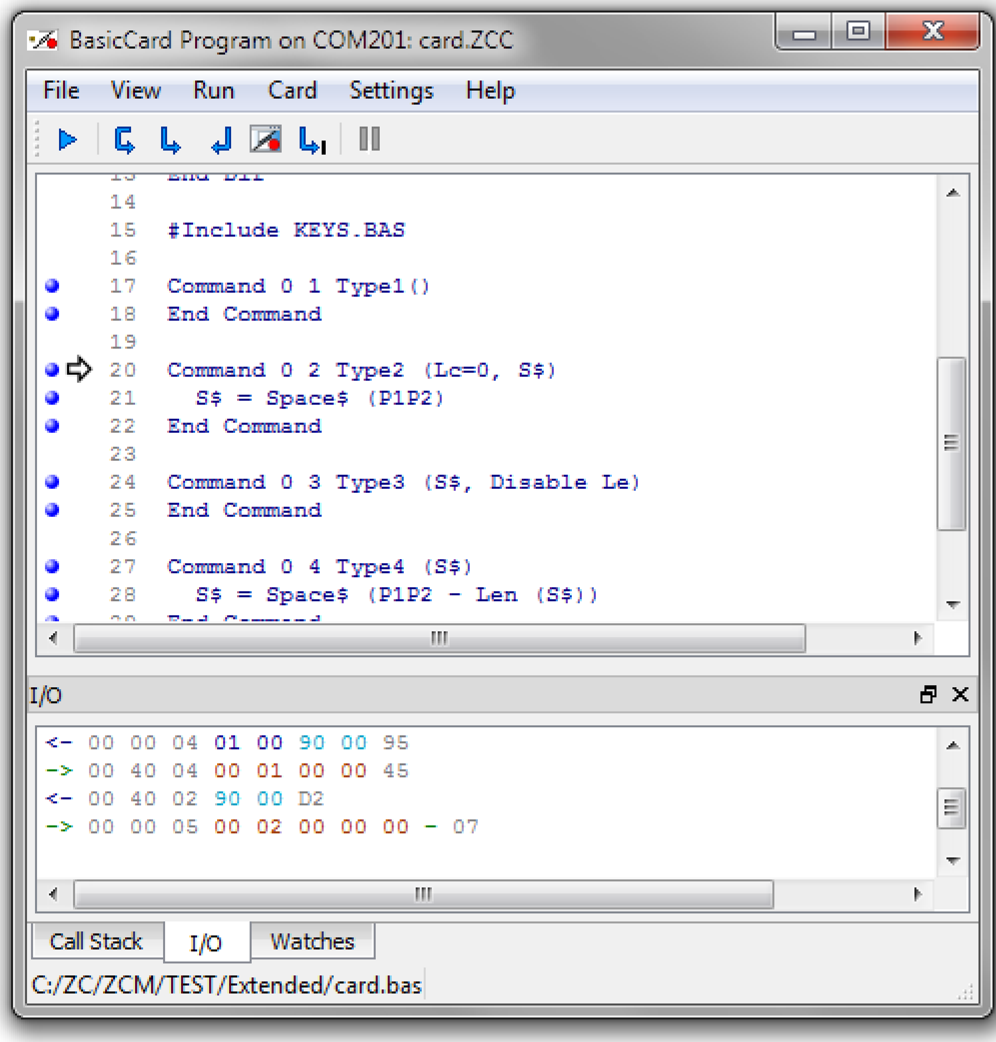
BasicCard Manual	Display this manual in your PDF file viewer (for instance, Adobe Reader)
About ZCMDTerm...	Display the version number of the running **ZCMDTerm** program
About Qt...	Display information about **Qt**, the cross-platform development tool from Nokia that was used to implement ZeitControl's software

6.8 The ZCMDCard BasicCard Debugger

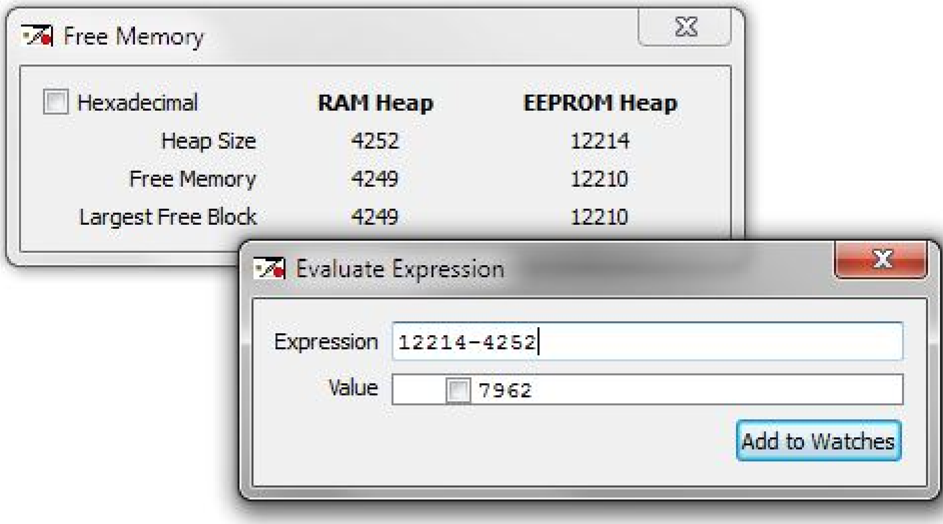
The **ZCMDCard** ZeitControl BasicCard Program Debugger is a source-level symbolic debugger for BasicCard programs. It waits for commands from the Terminal debugger **ZCMDTerm**, executes the commands under the control of the user, and sends its responses back to the Terminal debugger. It can also download BasicCard programs to a real BasicCard. It uses ZeitControl BasicCard Program files, with **.ZCC** extension, to store the information that it needs to compile and run BasicCard Programs.

The **ZCMDCard** debugger is very similar to the **ZCMDTerm** debugger. This section describes the features that differ.

The dockable windows are unchanged: the **I/O** window, **Call Stack** window, **Breakpoints** window, and **Watches** window are the same as in the **ZCMDTerm** debugger. Here they are shown as tabbed windows, which you can set up simply by dragging them into the same area of the main window:

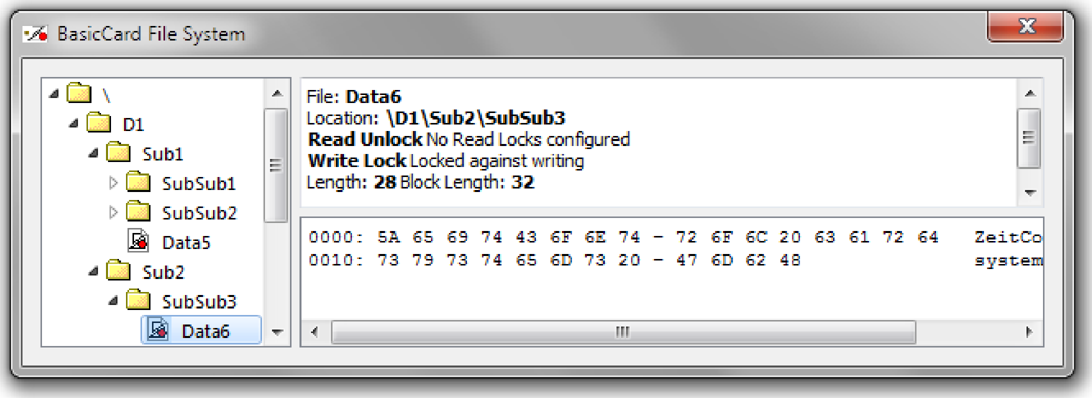


The floating **Evaluate Expression** and **Free Memory** windows are also unchanged:

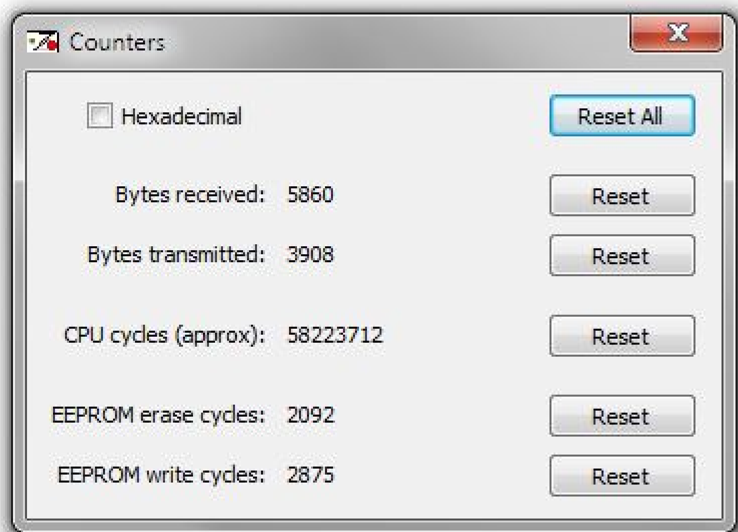


Two new floating windows are available in the **ZCMDCard** debugger: the **File System** window, and the **Counters** window.

- The **File System** window shows the BasicCard file system. It shows the directory tree in the left-hand pane, and the file properties and file contents in the two right-hand panes:

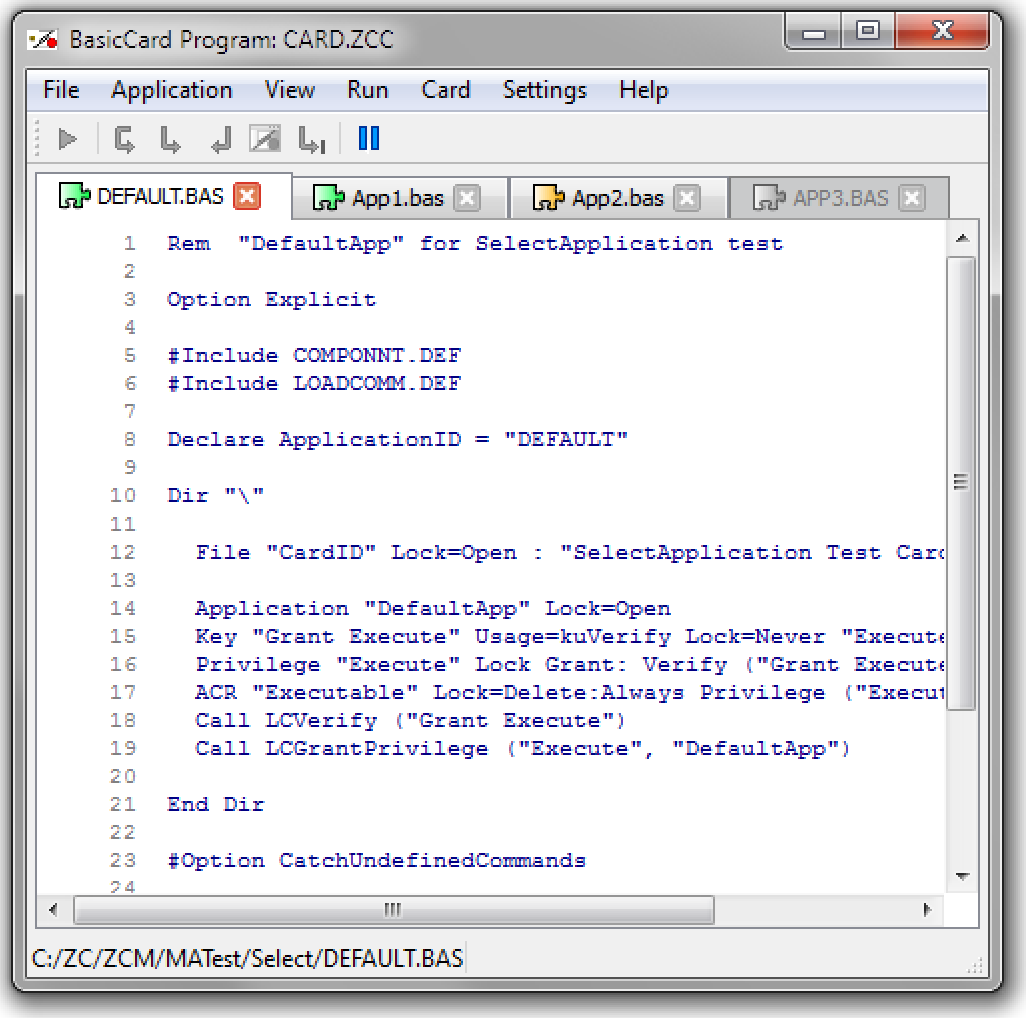


- The **Counters** window shows various performance-related counters:



6.8.1 MultiApplication Source Window

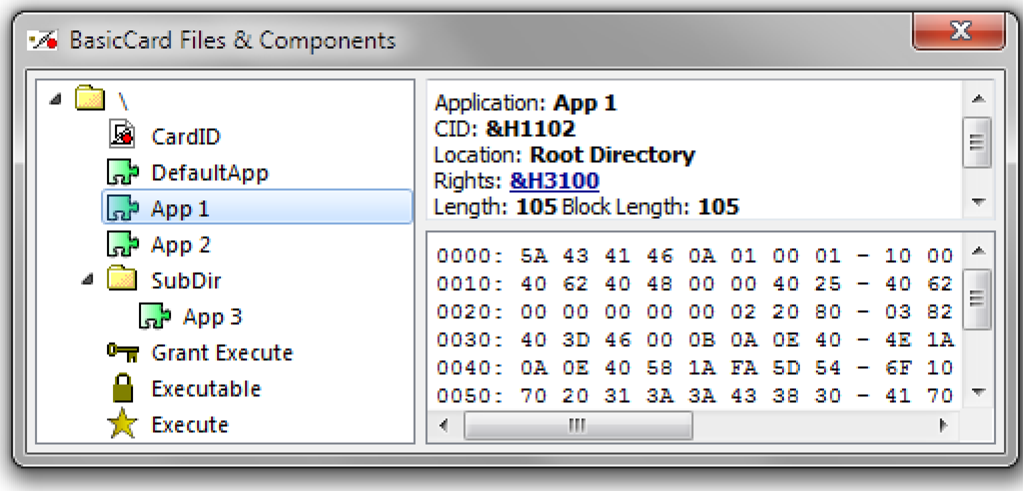
If the MultiApplication BasicCard is selected in the **Card Settings** dialog box (see **6.8.2 The Card Settings Dialog Box**), then the **Source** window displays each application in a separate tab:



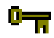
Applications that have been loaded into the card are shown with a green icon. Applications that have been compiled, but not yet loaded, are shown with an amber icon. Applications that have not yet been compiled are shown with a disabled tab (but in the **Application** menu, they are shown with a red icon).

The order of the tabs is important: it determines the order that the applications are loaded into a card when the **Load All** and **Download to Real Card** commands are executed. To change the tab order, just drag the tabs to their desired positions.

In the MultiApplication BasicCard, the **File System** window becomes the **Files & Components** window:

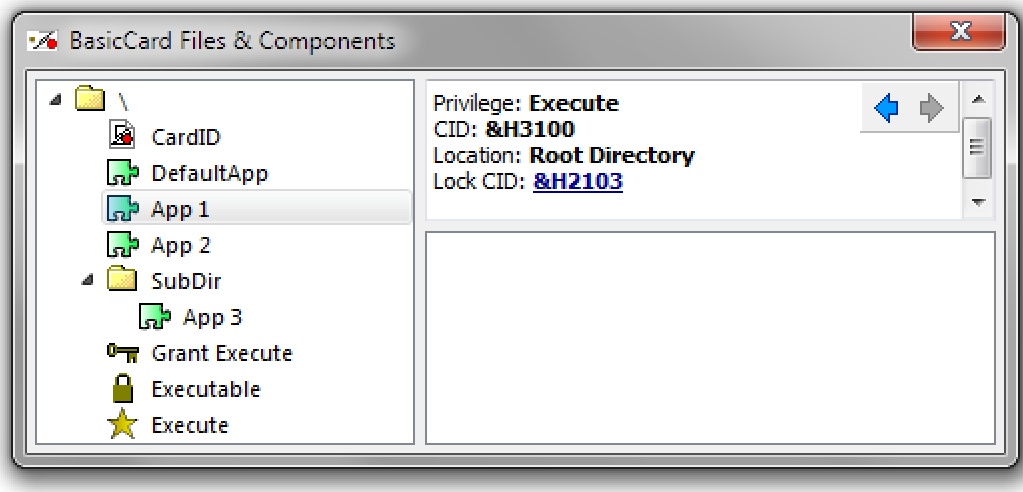


The type of each item is indicated by its icon:

	Directory		Privilege
	Data file		Flag
	Application		Cryptographic key
	ACR		

See **5.1.1 Component Types** for an explanation of these terms.

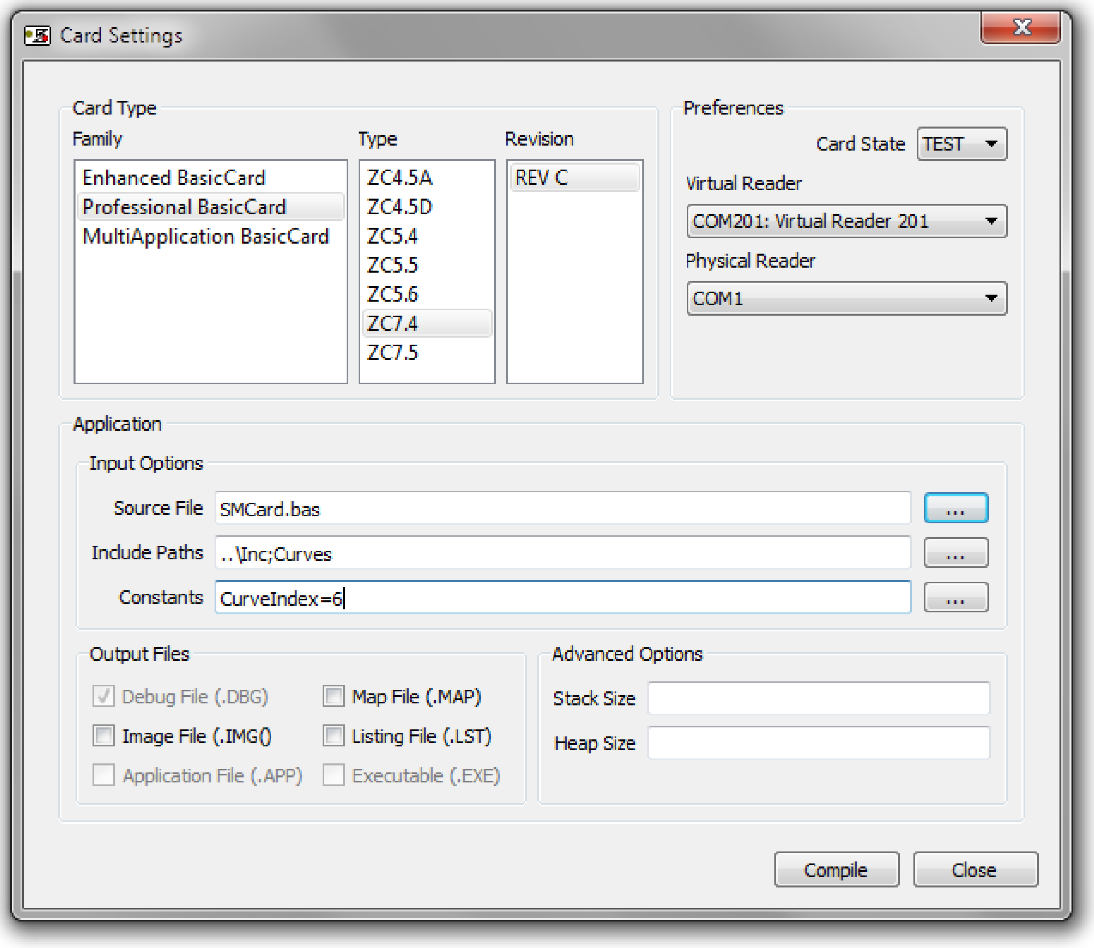
Click on the **&H3100** link to display the properties and contents of the component with that CID:



If you follow a chain of links like this, you can use the 'arrow' buttons  to navigate backwards and forwards in the chain.

6.8.2 The Card Settings Dialog Box

This dialog box lets you configure all the settings for a BasicCard program. Select the **Settings | BasicCard Program...** menu item to open:



The **CardType** section sets the type and revision of the BasicCard.

Items in the **Preferences** section:

Card State	The card state (**LOAD**, **TEST**, or **RUN**)
Virtual Reader	The COM port of the virtual reader – see **6.4 Physical and Virtual Card Readers**
Physical Reader	The COM port to use when downloading to a real card

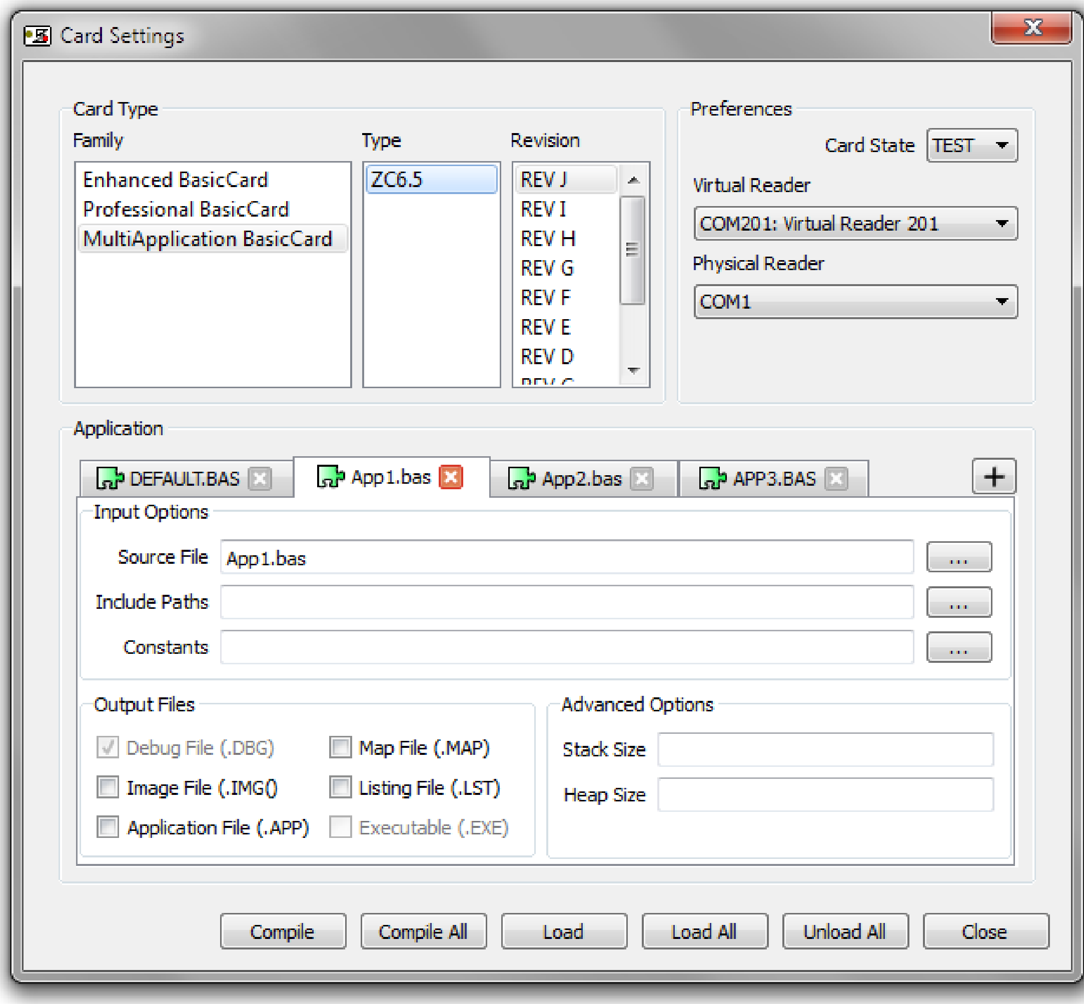
Items in the **Application** section correspond to command-line options in the **ZCMBasic** compiler – see **6.9.1 The ZC-Basic Compiler ZCMBasic.exe**:

Source File	The main source file of the Terminal program
Include Paths	The –**I** parameter: search path for included files
Constants	The –**D** parameter: pre-defined constants
Output Files	The –**O** parameter: which output files to generate
Stack Size	The –**S** parameter: the size of the BasicCard program P-Code stack
Heap Size	The **–H** parameter: application heap size in MultiApplication card

Include Paths and **Constants** can contain multiple entries separated by semi-colons; or you can click on the 'ellipsis' button ... to type each entry on a separate line.

All these settings are saved in the .ZCC file on program exit.

If MultiApplication BasicCard is selected in the **Card Type** section, the dialog box looks like this:



To change the order of the tabs, and thus the order in which the applications are loaded, just drag each tab to its desired position.

6.8.3 *ZCMDCard Menus*

The **File** menu contains the following items:

New BasicCard...	Create a new BasicCard File
Open BasicCard...	Open an existing BasicCard File
Save	Save the current BasicCard File
Save As...	Save the current BasicCard File under a new name
Edit *current file*	Edit the file showing in the **Source** window
Edit...	Edit a text file in the **BCDevEnv** Professional Development Environment
Edit Source ▸	Edit a source file from the current BasicCard program
Compile...	Open the Card **Settings** dialog box
Exit	Exit the **ZCMDCard** program

The **Application** menu is visible if the MultiApplication BasicCard has been selected in the **Card Settings** dialog box. It contains the following items:

Add...	Add an Application to the Application List
Load All	Load all Applications in the Application List into the virtual BasicCard
Unload All	Unload all Applications to create an empty virtual BasicCard

In addition, it contains a menu item for each Application in the Application List, with the following sub-menu:

View	View the Application's source code
Compile...	Compile the Application
Load...	Load the Application into the virtual BasicCard
Remove...	Remove the Application from the Application List

The **View** menu contains the following items:

Source File ►	Display a selected source file in the **Source** window
Procedure ►	Display a selected ZC-Basic procedure in the **Source** window
Execution Point	Display the code at the current PC
Breakpoints	Open the **Breakpoints** window for viewing and editing breakpoints
Watches	Open the **Watches** window for monitoring program data
Local Variables	View the current procedure and its local data in the **Call Stack** window
Call Stack	View all active procedures and their local data in the **Call Stack** window
I/O	Open the **I/O** window to show I/O between Terminal and BasicCard
File System	View files and directories in the BasicCard
Files & Components	View files and components in a MultiApplication BasicCard
Free Memory	Display the **Free Memory** window showing free space in the heap
Counters	Display the **Counters** window showing various performance counters
P-Code	Display P-Code instructions and registers in the **Source** window
Registers ►	Specify when the **P-Code Register** window is shown
Zoom ►	Adjust the text size of all windows
Run/Step toolbar	Show or hide the toolbar containing the blue Run/Step icons

The **Card** menu contains the following items:

Insert in Virtual Reader ►	Attach **ZCMDCard** to a Virtual Card Reader COM Port
Remove from Virtual Reader	Release the Virtual Card Reader COM Port
Download to Real Card...	Download the BasicCard program to a real BasicCard

The **Run** menu contains the following items:

Run	Run until program exit, or a breakpoint is reached
Step Over	Execute one source line, stepping over any procedure call
Step Into	Execute one source line, stepping into any procedure call
Step Out Of	Return to the calling procedure
Step to Terminal	Jump to the **ZCMDTerm** Terminal program debugger
Run to Cursor	Run to the current cursor position
Pause	Pause whichever debugger is currently running
Evaluate...	Evaluate an expression in the Evaluate window

Most of these items are also available as toolbar icons.

The **Settings** menu contains the following items:

BasicCard Program...	Open the **Card Settings** dialog box
Environment...	Set Windows® Registry variables **ZCPORT**, **ZCINC**, and **ZCZOOM**

The **Help** menu contains the following items:

BasicCard Manual	Display this manual in your PDF file viewer (for instance, Adobe Reader)
About ZCMDCard...	Display the version number of the running **ZCMDCard** program
About Qt...	Display information about **Qt**, the cross-platform development tool from Nokia that was used to implement ZeitControl's software

6.9 Command-Line Software

The following programs are run from a Win32 command-line console:

- **ZCMBasic**, a compiler for the ZC-Basic programming language.
- **ZCMSim**, a P-Code interpreter that runs compiled ZC-Basic programs. **ZCMSim** runs a Terminal program, and can run BasicCard programs simultaneously in simulated BasicCards, or communicate via a card reader with genuine BasicCards.
- **BCLoad**, for downloading P-Code to the BasicCard.
- **KeyGen**, a program that generates random keys for use in encryption.
- **BCKeys**, for downloading keys to the Enhanced BasicCard.

Each of these programs takes a filename as its main parameter. Other command-line parameters begin with '–' (minus sign) followed by one or more option letters, sometimes followed by data. No spaces are allowed between the minus sign and the option letters, or between the option letters and the data. Option letters may be upper or lower case.

ZCMBasic, **ZCMSim**, and **BCLoad** support parameter input files: if any command-line parameter has the form '*@filename*', subsequent parameters are read from the given file, one line at a time. Empty lines, and lines whose first non-space character is a single quote (the ZC-Basic comment character), are ignored. To specify a parameter that begins with the '@' character, simply repeat the '@' character; for example, "@@X" is passed to the program as "@X", and is not treated as a parameter file. This feature is also active for executable files created by the **ZCMBasic** compiler.

Notes:

- Three of these programs – **ZCMSim**, **BCLoad**, and **BCKeys –** communicate with a card reader, via a serial port or the PC/SC driver. The default value of the COM port is taken from the environment variable **ZCPORT**; or the Windows® Registry variable "**HKEY_CURRENT_USER\Software\ZeitControl\BasicCardV8\ZCINC"** if this environment variable does not exist; or 1 if neither of these variables exists. (To specify PC/SC reader number *n*, set the COM port to 100+*n*.)
- If a filename parameter contains spaces, it must be enclosed in quotation marks on the command line. (For example: `ZCMBasic -OI "Hello World"` compiles the file "`Hello World.BAS`" and creates the file "`Hello World.IMG`".)

6.9.1 The ZC-Basic Compiler ZCMBasic.exe

The compiler **ZCMBasic.exe** takes ZC-Basic source code as input, and produces P-Code as output. It compiles the entire program in one pass; there is no linking stage. To run the compiler:

ZCMBasic [*param* [*param* **. . .**]] *input-file* [*param* [*param* **. . .**]]

input-file The ZC-Basic source file. If no file extension is supplied, *input-file***.bas** is assumed.

param One of the following:

–C*type* Compile code for the given virtual machine type:

–CT or **–C0**	Terminal (the default).
–CE*n* or **–C3.***n*	Enhanced BasicCard version **ZC3.*n***
–CF*filename*	Professional BasicCard with Configuration File *filename*. If no file extension is supplied, *filename***.zcf** is assumed.
–CM	MultiApplication BasicCard

See Sections **1.6**–**1.9** for information about the different BasicCard types.

The compiler looks for the configuration file *filename* in the following directory order:

1. the current directory;
2. the Windows® Registry variable "**HKEY_CURRENT_USER\Software\ZeitControl\BasicCardV8\ZCCONFIG";**
3. The environment variable **ZCCONFIG**.

–D*symbol*[=*val*] Define *symbol* as if the source program contained the statement **Const** *symbol*=*val*. The *val* parameter must be an integer or a string; arithmetic expressions are not allowed. If *val* is absent, it defaults to 1.

–E[*exe-file*] Create an executable file that will run in a DOS box under Microsoft Windows®. If no file extension is supplied, *exe-file***.exe** is created. If *exe-file* is absent, *input-file***.exe** is created.

–H*heap-size* Set the Heap size of an Application for the MultiApplication BasicCard. See **5.2.4 Memory Allocation** for more information.

–I*path* Add *path* to the list of directories to search for **#Include** files (see **3.3.1 Source File Inclusion**). A closing backslash in *path* is optional. Multiple paths may be supplied, separated by semicolons.

–N*serial-number* Set the 8-byte Serial Number of a MultiApplication card, for use by the Component Section parser in the compiler. *serialnumber* must consist of 16 hexadecimal digits.

–OI[*image-file*] Generate an image file. If no file extension is supplied, *image-file***.img** is created. If *image-file* is absent, *input-file***.img** is created.

The image file is described in **11.1 ZeitControl Image File Format**.

–OD[*debug-file*] Generate a debug information file. If no file extension is supplied, *debug-file***.dbg** is created. If *debug-file* is absent, *input-file***.dbg** is created.

The debug file is described in **11.2 ZeitControl Debug File Format**.

–OA[*app-file*] Generate an Application file. If no file extension is supplied, *app-file***.app** is created. If *app-file* is absent, *input-file***.app** is created. The output file is a byte-for-byte copy of the Application file in the BasicCard. (Normally you won't need to create Application files, as the Application Loader finds all the information it needs in the image file.)

–OL[*list-file*] Generates a list file. If no file extension is supplied, *list-file***.lst** is created. If *list-file* is absent, *input-file***.lst** is created.

The list file is described in **11.4 List File Format**.

–OM[*map-file*] Generate a map file. If no file extension is supplied, *map-file***.map** is created. If *map-file* is absent, *input-file***.map** is created.

The map file is described in **11.5 Map File Format**.

–OE[*error-file*] Write all error messages to a file. If *error-file* already exists, it is deleted before compilation begins. If no file extension is supplied, *error-file***.err** is created. If *error-file* is absent, *input-file***.err** is created.

–S*stack-size* Set the size of the P-Code stack. Normally the compiler can work out for itself how big the stack has to be. But if the program contains recursive procedure calls or recursive **GoSub** calls, the compiler must guess the stack size, because it can't know how deep the recursion will go. You can override this guess with **–S***stack-size* (or with the **#Stack** pre-processor directive – see **3.3.8 Stack Size**).

–S*state* Switch the card into the specified state after the P-Code is downloaded. See also **3.3.6 Card State**. Only the first letter of *state* is significant:

First letter of *state*:	**'L'**	**'P'**	**'T'**	**'R'**
New card state:	**LOAD**	**PERS**	**TEST**	**RUN**

–V Display the version number of the compiler.

–X Delete all existing output files before compiling. Without this option, if a compilation error occurs, existing files are left untouched.

The following options are for use by ZeitControl applications, and are listed for completeness; you would not normally need to use them:

–OC[*clist-file*] Generate a command-list file, containing a declaration of every command defined in a BasicCard program. If no file extension is supplied, *clist-file***.clf** is created. If *clist-file* is absent, *input-file***.clf** is created.

–OT[*ctree-file*] Generate a call-tree file, a list of all procedure call dependencies. If no file extension is supplied, *ctree-file***.ctr** is created. If *ctree-file* is absent, *ctree-file***.ctr** is created.

–F*output* Set the output format, one of:

Console	The default
Debugger	Displays memory usage statistics
Ide	Displays errors and warnings only
FileList	Displays a list of all dependent source files

Only the first letter of *output* is significant.

6.9.2 The P-Code Interpreter ZCMSim.exe

The program **ZCMSim.exe** loads and runs a compiled ZC-Basic Terminal program from a ZeitControl Image File (or Debug File). It can also simultaneously run one or more BasicCard programs in simulated BasicCards, or it can communicate with real BasicCards via physical card readers. And for the MultiApplication BasicCard, it has a built-in Application Loader. To run the **ZCMSim** program:

> **ZCMSim** [*param* [*param* **. . .**]] *image-file* [*P1$* [*P2$* **. . .**]]

Parameters *before* the image-file name are processed by the **ZCMSim** program, as described below. Parameters *after* the image-file name (*P1$*, *P2$*,...) are passed to the Terminal program via the pre-defined **String** array **Param$(1 To nParams)** – see **3.22.9 Pre-Defined Variables**.

image-file The image file output by the compiler. If no file extension is supplied, *image-file***.img** is assumed. (So if this is a Debug File, the **.dbg** extension must be present.)

Note: The *image-file* parameter must be present, unless **ZCMSim** is functioning as an Application Loader for the MultiApplication BasicCard.

param One of the following:

–C*card-file*	The image file of a BasicCard program. If this parameter is present, **ZCMSim** simulates a BasicCard in the PC. If no file extension is supplied, *card-file***.img** is assumed. See also Note 2 below.
–P*com-port*	The number of the COM port that the card reader is attached to. (This can also be set from within the Terminal program itself, via the **ComPort** pre-defined variable.) This parameter may appear more than once – see Note 1 below.
–L[*log-file*]	Generate a log file, containing the commands sent to the card and their responses. If no file extension is supplied, *log-file***.log** is created. If *log-file* is absent, *image-file***.log** is created.
–A*card-file*	The image file of a MultiApplication BasicCard Application. The Application Loader will be invoked to load the Application into the (real or simulated) MultiApplication BasicCard. If no file extension is supplied, *card-file***.img** is assumed.
–E	Erase contents of MultiApplication card before loading the Application.
–W	Write the EEPROM data back to the image file(s) when the Terminal program exits. The Terminal program EEPROM data is written back to *image-file*. If the **–C** parameter is present on the command line, the EEPROM data in the simulated BasicCard program is written back to *card-file* when the Terminal program exits.
–WT[*new-file*]	Write the Terminal program EEPROM data back to *new-file* when the Terminal program exits. If no file extension is supplied, *new-file***.img** is created. If *new-file* is absent, the EEPROM data is written back to *image-file*.
–WC[*new-file*]	Write the EEPROM data in the simulated BasicCard program back to *new-file* when the Terminal program exits. If no file extension is supplied, *new-file***.img** is created. If *new-file* is absent, the EEPROM data is written back to *card-file*.
–D	Display the Application Loader commands on the screen as they are executed. (For use in conjunction with the –**A** parameter.)
–V	Display the version number.

P1$*, *P2$,... These parameters are separated by spaces or tabs. To pass a space or tab in a parameter, enclose it in quotation marks; to pass a quotation mark in a parameter, precede it with a backslash. (Backslashes not followed by quotation marks are passed as is.)

Notes:

1. If multiple **–P** parameters are present:

   - **–C** and **–WC** apply to the card on the most recently specified COM port;
   - the **ComPort** variable is set from the last **–P** parameter.

   For instance, to communicate with a simulated BasicCard program on COM1 and a real BasicCard in the default PC/SC reader (COM100):

   ZCMSim –P1 –C*card-file* **–P100** *image-file*

2. If *card-file* is a ZeitControl Configuration File (with .ZCF or .MCF extension), an empty card of the appropriate type is simulated. This is for the MultiApplication BasicCard – the BasicCard type is not available from the image file of an Application (which is independent of the specific BasicCard OS), so this information must be supplied separately, via the –**C** parameter. For example, to test Applications **App1** and **App2** with Terminal program **Term**:

   ZCMSim –C\BasicCardV8\MultiApp\ZC65_A.MCF –AApp1 –AApp2 –D Term

   In this case only, the *image-file* parameter may be absent; **ZCMSim** then functions as a stand-alone Application Loader. If the *card-file* parameter is also absent, **ZCMSim** will attempt to load the Applications into a real MultiApplication BasicCard. For instance:

   ZCMSim –P100 –AApp1 –AApp2 –D

6.9.3 The Card Loader BCLoad.exe

The program **BCLoad.exe** downloads P-Code and data to a single-application BasicCard.

The ZC-Basic compiler produces a ZeitControl Image File as output, containing P-Code and data in binary form. To run the **BCLoad** program:

BCLoad [*param* [*param* **. . .**]] *image-file* [*param* [*param* **. . .**]]

image-file The image file output by the compiler. If no file extension is supplied, *image-file***.img** is assumed. (A debug file is also allowed here; in this case. the **.dbg** extension must be supplied.)

param One of the following:

–D Displays the commands on the screen as they are executed.

–L[*log-file*] Generates a log file, containing the commands sent to the card and their responses. If no file extension is supplied, *log-file***.log** is created. If *log-file* is absent, *image-file***.log** is created.

–E[*error-file*] Writes all error messages to a file. If *error-file* already exists, it is deleted before the download begins. (So if *error-file* exists after the program exits, it means that an error occurred.) If no file extension is supplied, *error-file***.err** is created. If *error-file* is absent, *image-file***.err** is created.

–P*com-port* The number of the COM port that the card reader is attached to.

–S*state* Switches the card into the specified state after the download. Only the first letter of *state* is significant:

First letter of *state*:	**'L'**	**'P'**	**'T'**	**'R'**
New card state:	**LOAD**	**PERS**	**TEST**	**RUN**

Notes:

1. The ZC-Basic source code for this program is supplied on the distribution disk, in the `BasicCardV8\Source\BCLoad` directory. **BCLoad.exe** was compiled with the **Compile.bat** command file in the same directory.

2. To download an Application to the MultiApplication BasicCard, use the built-in Application Loader in **ZCMSim** or **ZCMDCard**.

6.9.4 The Key Generator KeyGen.exe

The program **KeyGen.exe** generates cryptographic keys for the encryption and decryption of commands and responses. It creates a ZC-Basic source file containing **Declare Key** statements. This file can be **#Included** in the source code of the Terminal and BasicCard programs, or it can be downloaded separately to an Enhanced BasicCard using the **BCKeys** Key Loader program. The program prompts the user to press keys on the keyboard at random; the cryptographic keys are generated from this user input, after hashing with the **MD5** algorithm (see R.L. Rivest, "The MD5 Message Digest Algorithm", RSA Data Security, Inc., April 1992). To run the **KeyGen** program:

KeyGen [*param* [*param* **. . .**]] *key-file* [*param* [*param* **. . .**]]

key-file The name of the key file to create or update. If no file extension is supplied, *key-file***.bas** is assumed.

param One of the following:

–K*key*[**(***len*[**,** *count*]**)**]	*key* is a key number between 0 and 255; *len* is a key length between 1 and 255; and *count* is the initial value of the error counter for the key, between 0 and 15 (see **3.18.3 Key Declaration**). If *len* is absent, it defaults to 8; if *count* is absent, the error counter for the key is disabled. You can create multiple keys by specifying the **–K** parameter more than once.
–Q	Generates random numbers quickly, without requiring keyboard input from the user. *Note:* This feature is provided for convenience of use during the development of an application. Keys and polynomials generated with the **–Q** parameter should not be used in a released application, as this might compromise the security of the encryption algorithms.
–U	*key-file* is updated, rather than being created from scratch – existing keys in *key-file* are preserved, unless overridden by **–K** .

Note: The generation of cryptographic keys is a delicate business. The security of the encryption algorithms used by the BasicCard relies on the secrecy of the keys generated by the **KeyGen** program, which in turn relies on the quality of the random number generator. To foster confidence in the security of our product, we provide the C++ source code of the **KeyGen** program in the directory `BasicCardV8\Source\KeyGen`.

6.9.5 The Key Loader BCKeys.exe

The program **BCKeys.exe** downloads cryptographic keys to an Enhanced BasicCard. The following conditions apply:

- The BasicCard must be in state **LOAD** (or switchable to state **LOAD**);
- The BasicCard must already have been loaded with P-Code and data by the **BCLoad** program;
- All keys that you want to download must have been declared in the ZC-Basic source code, with **Declare Key** statements.

The program takes a key file as input. This is a ZC-Basic source file that contains only **Declare Key** statements. The **KeyGen** program can generate key files for you – see **6.9.4 The Key Generator KeyGen.exe**.

To run the **BCKeys** program:

BCKeys [*param* [*param* **. . .**]] *key-file* [*param* [*param* **. . .**]]

key-file The key file, as described above. If no file extension is supplied, *key-file***.bas** is assumed.

param One of the following:

–K[*key*] *key* is a key number between 0 and 255. You can download multiple keys by specifying this parameter more than once. If *key* is absent, or if no **–K** parameter is given,all the keys in *key-file* are downloaded.

–L[*log-file*] Generates a log file, containing the commands sent to the card and their responses. If no file extension is supplied, *log-file***.log** is created. If *log-file* is absent, *key-file***.log** is created.

–D Displays the commands on the screen as they are executed.

–P*com-port* The number of the COM port that the card reader is attached to.

–S*state* Switches the card into the specified state after the download. Only the first letter of *state* is significant:

First letter of *state*:	**‘L’**	**‘T’**	**‘R’**
New card state:	**LOAD**	**TEST**	**RUN**

Note: State **PERS** is not available in Enhanced BasicCards, so it is not allowed here.

7. System Libraries

The functionality of the ZC-Basic language can be extended using ZeitControl System Libraries.

In Terminal programs, and Professional and MultiApplication BasicCards, the System Libraries are built into the Operating System; in the Enhanced BasicCard, System Libraries are implemented as Plug-In Libraries that are loaded into EEPROM only if they are needed. A ZeitControl Plug-In Library File *library*.**lib** is provided for each Enhanced BasicCard Plug-In Library. In all cases, to use a library:

#Include *library*.**def**

This loads the library if necessary, and declares its procedures and data.

The following System Libraries are currently available:

Name	*Description*	*Terminal*	*Enhanced BasicCard*	*Professional BasicCard*	*Multi-Application BasicCard*
RSA	RSA Public-Key Cryptography	✓		*	
EC-p	512-bit Elliptic Curve Cryptography	✓		*	
EC–211	211-bit Elliptic Curve Cryptography	✓		*	✓
EC–167	167-bit Elliptic Curve Cryptography	✓		*	✓
EC–161	161-bit Elliptic Curve Cryptography	✓	✓		
COMPONENT	Security Component handling	✓			✓
TMLib	Transaction Manager library	✓		*	
Crypto	Cryptographic schemes	✓		*	
BigInt	Big integer arithmetic	✓		*	
AES	Advanced Encryption Standard	✓	✓	*	✓
EAX	Encryption with Authentication	✓		*	✓
OMAC	Message Authentication	✓		*	✓
SHA	Secure Hash Algorithms	✓	✓	✓	✓
TLVLib	Tag-Length-Value library	✓		*	
Mifare	Mifare™ block read and write functions			*	*
MATH	Mathematical functions	✓			
MISC	Miscellaneous procedures	✓	✓	✓	✓

* These System Libraries are available in some, but not all, BasicCards. See **Professional and MultiApplication BasicCard Datasheet** for the latest information.

These libraries are supplied with the distribution kit, in the `BasicCardV8\Lib` directory. The program **LIBVER**, in the same directory, displays the name and version number of an Enhanced BasicCard Plug-In Library file.

In the descriptions of the individual libraries, error codes may be defined. These error codes are signalled via the **LibError** variable. The **ZCMBasic** compiler automatically declares this variable if any libraries are included that can return an error code. **LibError** contains the most recent error code signalled by a library procedure. A library procedure never sets **LibError** back to zero; if you want to continue after detecting a library error, you should set **LibError** to zero yourself.

A library error code is a 2-byte value of the form &H4XXX, with the high nibble equal to 4. Therefore, unless you are using the **T=0** protocol (and at the cost of strict ISO compatibility), you can return **LibError** in the **SW1SW2** status word if a library error occurs in a BasicCard program. For example:

```
Sub CheckLibError()
  If LibError = 0 Then Exit Sub
  SW1SW2 = LibError
  LibError = 0 ' Reset LibError for the next command
  Exit
End Sub
```

7.1 RSA: The Rivest-Shamir-Adleman Library

The **RSA** library implements Rivest-Shamir-Adleman public-key cryptography. It is based on the document **PKCS #1 v2.1: RSA Cryptography Standard** from RSA Data Security, Inc, available at http://www.rsa.com/rsalabs/node.asp?id=2125 . The following operations are supported:

- on-card private/public key pair generation, with public key length up to 4096 bits;
- encryption and decryption;
- digital signature generation and verification.

7.1.1 Overview

In **RSA** cryptography, a private key consists of three numbers (p, q, e), where p and q are prime numbers and e is a number relatively prime to p-1 and q-1. The corresponding public key consists of the two numbers (n, e), where n is the product of p and q.

The private exponent d is the inverse of e modulo (p-1)(q-1). Mathematically, this means that for any number m, m^{ed} is equal to m modulo n. If Alice wants to send a message m to Bob that only Bob can decrypt, Alice computes $c = m^e$ modulo n using Bob's public key (n_B, e_B). Bob can then recover m modulo n (and therefore m, if m is less than n), as follows:

- using p and q, compute the private exponent d;
- compute $m = c^d$ modulo n.

Similarly, if Alice wants to sign a message m, she computes the private exponent d using her own private key (p_A, q_A, e_A), and then computes the signature $s = m^d$ modulo n. Anyone who has Alice's public key (n_A, e_A) can verify that $s^e = m$ modulo n; and therefore that whoever created the signature s had knowledge of Alice's private key (and was therefore presumably Alice herself).

The security of the **RSA** system rests on the difficulty of recovering p and q if only their product n is known: the *factorisation problem*. If I know n and e, but I don't know p and q, then I can't calculate the private exponent d. The difficulty of the factorisation problem depends on the size of n. Current state-of-the-art factoring methods can expect to factor a 768-bit public key in a matter of months; 1024-bit public keys are expected to resist factorisation for a few more years; and 2048-bit keys are expected to be secure for the foreseeable future.

The **RSA** System Library consists of two separate, independent sets of procedures:

- The **RSA-4096** Procedure Set. This is the extended implementation, for the **ZC7**-series Professional BasicCard. It supports keys up to 4096 bits long. All procedure names begin with **RsaEx**.

- The **RSA-1024** Procedure Set. This is the original implementation, for the **ZC4-**series Professional BasicCard. It supports keys up to 1024 bits long. All procedure names begin with **Rsa**.

Both Procedure Sets are available in Terminal programs.

7.1.2 RSA Cryptographic Primitives

Four cryptographic primitives are defined in **PKCS #1 v2.1: RSA Cryptography Standard**:

RSAEP ((n, e), m**)**	**RSA** Encryption Primitive: $c = m^e$ modulo n
RSADP ((n, d), c**)**	**RSA** Decryption Primitive: $m = c^d$ modulo n
RSASP1 ((n, d), c**)**	**RSA** Signature Primitive 1: $s = m^d$ modulo n
RSAVP1 ((n, e), s**)**	**RSA** Verification Primitive 1: $m = s^e$ modulo n

RSAEP and **RSAVP1** are functionally identical, as are **RSADP** and **RSASP1**. Both Procedure Sets implement these cryptographic primitives.

7.1.3 RSA Cryptographic Schemes

The **PKCS #1 v2.0: RSA Cryptography Standard** defines four Cryptographic Schemes, two each for for encryption/decryption and signature/verification.

Encryption Schemes

As described in **PKCS #1 v2.0: RSA Cryptography Standard**, an *encryption scheme* consists of an *encryption operation* and a *decryption operation*. Two encryption schemes are defined: **RSAES-PKCS1-v1_5** and **RSAES-OAEP**. The second of these is available only in the **RSA-4096** Procedure Set.

The **RSAES-PKCS1-v1_5** encryption operation is:

> **RSAES-PKCS1-v1_5-ENCRYPT (**(*n*, *e*), *M***)**
>
> where (*n*, *e*) is the recipient's public key, and *M* is the message to encrypt.

The **RSAES-PKCS1-v1_5** decryption operation is:

RSAES-PKCS1-v1_5-DECRYPT (*K*, *C***)**

where *K* is the recipient's private key, and *C* is the ciphertext to decrypt.

The **RSAES-OAEP** encryption operation is:

> **RSAES-OAEP-ENCRYPT (**(*n*, *e*), *M*, *L***)**
>
> where (*n*, *e*) is the recipient's public key, *M* is the message to encrypt, and *L* is the label to be associated with the message.

The **RSAES-OAEP** decryption operation is:

> **RSAES-OAEP-DECRYPT (***K*, *C*, *L***)**
>
> where *K* is the recipient's **private** key, *C* is the ciphertext to decrypt, and *L* is the label associated with the message.

The **RSA** Plug-In Library uses **SHA-1** or **SHA-256** as the hash function for the **RSAES-OAEP** encryption scheme.

Signature Schemes With Appendix

As described in **PKCS #1 v2.1: RSA Cryptography Standard**, a *signature scheme with appendix* consists of a *signature generation operation* and a *signature verification operation*. Two signature schemes with appendix are defined: **RSASSA-PKCS1-v1_5** and **RSASSA-PSS**. The second of these is available only in the **RSA-4096** Procedure Set.

The **RSASSA-PKCS1-v1_5** signature generation operation is:

> **RSASSA-PKCS1-v1_5-SIGN (***K*, *M***)**
>
> where *K* is the recipient's private key and *M* is the message to be signed.

The **RSASSA-PKCS1-v1_5** signature verification operation is:

> **RSASSA-PKCS1-v1_5-VERIFY (**(*n*, *e*), *M*, *S***)**
>
> where (*n*, *e*) is the signer's public key, *M* is the message whose signature is to be verified, and *S* is the signature to be verified.

The **RSASSA-PSS** signature generation operation is:

> **RSASSA-PSS-SIGN (***K*, *M***)**
>
> where *K* is the recipient's private key and *M* is the message to be signed.

The **RSASSA-PSS** signature verification operation is:

> **RSASSA-PSS-VERIFY (**(*n*, *e*), *M*, *S***)**
>
> where (*n*, *e*) is the signer's public key, *M* is the message whose signature is to be verified, and *S* is the signature to be verified.

The **RSA** Plug-In Library uses hash function **SHA-1** for signature scheme **RSASSA-PKCS1-v1_5**, and **SHA-1** or **SHA-256** for signature scheme **RSASSA-PSS**.

7.1.4 The RSA-4096 Procedure Set

The **RSA-4096** Procedure Set represents large integers as ZC-Basic strings; the first byte in the string (with subscript 1) is the most significant byte. Public and Private Keys are encoded in BER-TLV format as a sequence of integers, stored in a ZC-Basic string. In ASN-1 syntax, using the terminology of **PKCS #1 v2.1: RSA Cryptography Standard**:

```
RsaExPublicKey ::= SEQUENCE {
      modulus           INTEGER, -- n
      publicExponent    INTEGER  -- e
}

RsaExPrivateKey ::= SEQUENCE {
      nbits             INTEGER, -- Exact bit length of n = p*q
      prime1            INTEGER, -- p
      prime2            INTEGER, -- q
      publicExponent    INTEGER, -- e
      exponent1         INTEGER, -- d mod (p-1)
      exponent2         INTEGER, -- d mod (q-1)
      coefficient       INTEGER  -- (inverse of q) mod p
}
```

The ASN-1 syntax presented here is for information only – you don't need to understand it to use the library procedures.

The **RSA-4096** Procedure Set is part of the **RSA** library. To load it:

#Include RSA.DEF

The file RSA.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

The following procedures are provided:

Key Generation
Sub RsaExGenerateKey (*nBits*, *pBits*, *eBits*, *e$*, *PrK$***)**
Sub RsaExPublicKey (*PrK$*, *PuK$***)**
Sub RsaExConstructKey (*p$*, *q$*, *e$*, *PrK$***)**
Function RsaExGeneratePrime (*Bytelen*, *MSW***) As String**
Function RsaExPseudoPrime (*n$*, *nRounds***)**

Cryptographic Primitives
Sub RsaExEncryptRaw (*Mess$*, *PuK$***)**
Sub RsaExDecryptRaw (*Mess$*, *PrK$***)**

Encryption Schemes
Sub RsaExPKCS1Encrypt (*Mess$*, *PuK$***)**
Function RsaExPKCS1Decrypt (*Mess$*, *PrK$***)**
Sub RsaExOAEPEncrypt (*HashLen*, *Mess$*, *L$*, *PuK$***)**
Function RsaExOAEPDecrypt (*HashLen*, *Mess$*, *L$*, *PrK$***)**

Signature Schemes
Sub RsaExPKCS1Sign (*Hash$*, *PrK$*, *Sig$***)**
Function RsaExPKCS1Verify (*Hash$*, *PuK$*, *Sig$***)**
Sub RsaExPSSSign (*Hash$*, *SaltLen*, *PrK$*, *Sig$***)**
Function RsaExPSSVerify (*Hash$*, *SaltLen*, *PuK$*, *Sig$***)**

These procedures are described in the following sections.

7.1.5 RSA-4096 Key Generation

In the **RSA-4096** Procedure Set, the bit lengths of p and q can be any numbers from 80 to 2048 inclusive – p and q do not need to be the same length. There is usually no reason to want different bit lengths for p and q, but some RSA profiles require this capability. Unless you have a reason not to, you should specify equal lengths for p and q.

The bit length of the public exponent *e* can be any number between 2 and the bit length of min (*p*, *q*) inclusive. The speed of public-key operations depends linearly on the bit length of *e*, so unless you have a reason for it, you should not choose a bit length greater than 32. Alternatively, you can provide a specific public exponent.

To generate a private key:

Sub RsaExGenerateKey (*nBits*, *pBits*, *eBits*, *e$*, *PrK$*)

nBits	The bit length of the modulus *n*. This can be any number from 160 to 4096 inclusive. For high security combined with fast execution times, we recommend a value of 2048.
pBits	The bit length of *p*. If *pBits* is non-zero, the bit length of *q* will equal *nBits* – *pBits*; if *pBits* is zero, the bit lengths of *p* and *q* will both equal (*nBits* + 1) / 2. In all cases, the bit lengths of *p* and *q* must be between 80 and 2048 inclusive.
eBits	The bit length of *e*. If e*Bits* is non-zero, a random public exponent *e* is generated, of bit length *eBits*; in this case, the *e$* parameter must be equal to the empty string.
e$	A specific value for the public exponent *e*. If *e$* is not equal to the empty string, it is used as the public exponent; in this case, the *eBits* parameter must be zero.
PrK$	The generated private key, in BER-TLV format.

Notes:

1. In every case, *p* and *q* satisfy abs($\log_2 p - \log_2 q$) > 0.1 (this is required by some RSA profiles).
2. If *eBits* is zero, and *e$* is the empty string, then the default value 65537 is used as the public exponent.
3. Most users will not need to set *pBits*, *eBits*, or *e$*, in which case the procedure call reduces to:

```
Call RsaExGenerateKey (nBits, 0, 0, "", PrK$)
```

To generate a public key from a private key:

Sub RsaExPublicKey (*PrK$*, *PuK$*)

PrK$	The private key, in BER-TLV format.
PuK$	The corresponding public key, in BER-TLV format.

For backward compatibility with the **RSA-1024** Procedure Set, the following procedure constructs a private key in BER-TLV format, from its component parts:

Sub RsaExConstructKey (*p$*, *q$*, *e$*, *PrK$*)

p$	The first prime factor of *n*.
q$	The second prime factor of *n*.
e$	The public exponent.
PrK$	The constructed private key, in BER-TLV format.

Two procedures are provided for users who want to generate their own primes:

Function RsaExGeneratePrime (*Bytelen*, *MSW*) As String

Bytelen	The length of the prime number in bytes.
MSW	The most significant word. It must be at least 256.

Function RsaExPseudoPrime (*x$*, *nRounds*)

Returns **True** (&HFFFF) if *x$* is a Rabin-Miller pseudo-prime, i.e. it passes *nRounds* rounds of a Rabin-Miller primality test. If *nRounds* is zero, the function selects a value which ensures that the probability of error is less than 1 in 2^{100} (and this value is returned in the *nRounds* parameter).

7.1.6 RSA-4096 Cryptographic Primitives

*Cryptographic primitives **RSAEP** and **RSAVP1***

RsaExEncryptRaw (*Mess$, PuK$***)**

Mess$	The message to encrypt.
PuK$	The public key (*n*, *e*) in BER-TLV format.

This procedure computes *Mess$* ^{e} modulo *n*, returning the result in *Mess$*.

In a BasicCard program, the following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n* is shorter than 160 bits
RsaKeyTooLong	*n* is longer than 4096 bits
RsaBadProcParams	*Mess$* is longer than *n*

*Cryptographic primitives **RSADP** and **RSASP1***

RsaExDecryptRaw (*Mess$*, *PrK$***)**

Mess$	The message to encrypt.
PrK$	The private key in BER-TLV format.

This procedure computes *Mess$* $^{d\$}$ modulo *n*, returning the result in *Mess$*.

In a BasicCard program, the following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*p* or *q* is shorter than 80 bits
RsaKeyTooLong	*p* or *q* is longer than 2048 bits
RsaBadProcParams	*Mess$* is longer than *n*

7.1.7 RSA-4096 Encryption Schemes

*Encryption Scheme **RSAES-PKCS1-v1_5***

To encrypt a message using the **RSAES-PKCS1-v1_5-ENCRYPT** encryption operation:

Call RsaExPKCS1Encrypt (*Mess$*, *PuK$***)**

Mess$	The message to be encrypted. It must be at least 11 bytes shorter than *n$*. The encrypted message is returned in *Mess$*.
PuK$	The public key (*n*, *e*) in BER-TLV format.

The following error code is returned in the **LibError** variable:

RsaBadProcParams	*Mess$* is not at least 11 bytes shorter than *n$*.

To decrypt a message using the **RSAES-PKCS1-v1_5-DECRYPT** decryption operation:

MessageValid = **RsaExPKCS1Decrypt (***Mess$*, *PrK$***)**

Mess$	The message to be decrypted. It must be the same length as *n$* (where *n* = *pq* is the public-key modulus). The decrypted message is returned in *Mess$*.
PrK$	The private key of the recipient, in BER-TLV format.
MessageValid	**True** if *Mess$* was successfully decrypted.

The following error code is returned in the **LibError** variable:

RsaBadProcParams	*Mess$* is not the same size as *n*.

*Encryption Scheme **RSAES-OAEP***

The **RSAES-OAEP** scheme accepts a *label* as input. The same label must be specified for encryption and decryption. The label can be any arbitrary string, and need not be secret; if in doubt, use the empty string “”.

To encrypt a message using the **RSAES-OAEP-ENCRYPT** operation:

Call Sub RsaExOAEPEncrypt (*HashLen*, *Mess$*, *L$*, *PuK$***)**

HashLen	The size in bytes of the hash function output: 20 for **SHA-1**, or 32 for **SHA-256**.
Mess$	The message to be encrypted. It must be at least 42 bytes shorter than *n$*. The encrypted message is returned in *Mess$*.
L$	The label associated with the message. Any string is accepted.
PuK$	The public key (*n*, *e*) in BER-TLV format.

The following error code is returned in the **LibError** variable:

RsaBadProcParams	*Mess$* is not at least 42 bytes shorter than *n$*.

To decrypt a message using the **RSAES-OAEP-DECRYPT** operation:

MessageValid = **RsaExOAEPDecrypt (***HashLen*, *Mess$*, *L$*, *PrK$***)**

HashLen	The size in bytes of the hash function output: 20 for **SHA-1**, or 32 for **SHA-256**.
Mess$	The message to be decrypted. It must be the same length as *n$* (where $n = pq$ is the public-key modulus). The decrypted message is returned in *Mess$*.
L$	The label associated with the message. It must match the *L$* parameter to the **RsaExOAEPEncrypt** procedure.
PrK$	The private key in BER-TLV format.
MessageValid	**True** if *Mess$* was successfully decrypted.

The following error code is returned in the **LibError** variable:

RsaBadProcParams	Mess$ is not the same size as n$.

7.1.8 RSA-4096 Signature Schemes

*Signature scheme **RSASSA-PKCS1-v1_5***

To generate a signature using the **RSASSA-PKCS1-v1_5-SIGN** signature generation operation:

Call RsaExPKCS1Sign (*Hash$*, *PrK$*, *Sig$***)**

Hash$	The hash of the data to be signed – 20 bytes (**SHA-1**) or 32 bytes (**SHA-256**).
PrK$	The private key in BER-TLV format.
Sig$	The signature calculated by **RsaExPKCS1Sign**. It has the same size as *n*.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n* is shorter than 160 bits
RsaBadProcParams	*Hash$* is not 20 or 32 bytes long

To verify a signature using the **RSASSA-PKCS1-v1_5-VERIFY** signature verification operation:

SignatureValid = **RsaExPKCS1Verify (***Hash$*, *PuK$*, *Sig$***)**

Hash$	The hash of the data that was signed – 20 bytes (**SHA-1**) or 32 bytes (**SHA-256**).
PuK$	The public key (*n*, *e*) in BER-TLV format.
Sig$	The signature to be verified.
SignatureValid	**True** if the signature is valid.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n* is shorter than 160 bits
RsaBadProcParams	*Hash$* is not 20 or 32 bytes long

*Signature scheme **RSASSA-PSS***

To generate a signature using the **RSASSA-PSS-SIGN** signature generation operation:

Call RsaExPSSSign (*Hash$*, *SaltLen*, *PrK$*, *Sig$***)**

Hash$	The hash of the data to be signed – 20 bytes (**SHA-1**) or 32 bytes (**SHA-256**).
SaltLen	Byte length of random 'salt' – see **PKCS #1 v2.0** for details. We suggest a value of 16, although any positive value is allowed, subject to *SaltLen* + **Len(***Hash$***)** + 1 being less than the size of the RSA modulus *n*.
PrK$	The private key in BER-TLV format.
Sig$	The signature calculated by **RsaExPKCS1Sign**. It has the same size as *n*.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n* is shorter than 160 bits
RsaBadProcParams	*Hash$* is not 20 or 32 bytes long

To verify a signature using the **RSASSA-PKCS1-v1_5-VERIFY** signature verification operation:

SignatureValid = **RsaExPSSVerify (***Hash$*, *SaltLen*, *PuK$*, *Sig$***)**

Hash$	The hash of the data that was signed – 20 bytes (**SHA-1**) or 32 bytes (**SHA-256**).
SaltLen	Length of random 'salt' generated by **RsaExPSSSign**.
PuK$	The public key (*n*, *e*) in BER-TLV format.
Sig$	The signature to be verified.
SignatureValid	**True** if the signature is valid.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n* is shorter than 160 bits
RsaBadProcParams	*Hash$* is not 20 or 32 bytes long

7.1.9 The RSA-1024 Procedure Set

The **RSA-1024** Procedure Set represents large integers as ZC-Basic strings; the first byte in the string (with subscript 1) is the most significant byte.

The **RSA-1024** Procedure Set is part of the **RSA** library. To load it:

#Include RSA.DEF

The file RSA.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

The following procedures are provided:

Key Generation
Function RsaPseudoPrime (*x$*, *nRounds***)**
Sub RsaGenerateKey (*nBits*, *eBits*, *p$*, *q$*, *e$***)**
Function RsaPublicKey (*p$*, *q$***) As String**

Cryptographic Primitives
Sub RsaEncrypt (*Mess$*, *n$*, *e$***)**
Sub RsaDecrypt (*Mess$*, *p$*, *q$*, *e$***)**

Encryption Scheme
Sub RsaPKCS1Encrypt (*Mess$*, *n$*, *e$***)**
Function RsaPKCS1Decrypt (*Mess$*, *p$*, *q$*, *e$***)**

Signature Scheme
Sub RsaPKCS1Sign (*Hash$*, *p$*, *q$*, *e$*, *Sig$***)**
Function RsaPKCS1Verify (*Hash$*, *n$*, *e$*, *Sig$***)**

These procedures are described in the following sections.

7.1.10 RSA-1024 Key Generation

To generate a private key:

Call RsaGenerateKey (*nBits*, *eBits*, *p$*, *q$*, *e$***)**

nBits	Length of public key *n*. Set *nBits* = 1024 for maximum security. In a BasicCard program, *nBits* must be a multiple of 16, with 496 ≤ *nBits* ≤ 1024. In a Terminal program, *nBits* can be any number between 16 and 4064.
eBits	Length of public exponent *e*. In a BasicCard program, *eBits* must be a multiple of 8, with 8 ≤ *eBits* ≤ 32. In a Terminal program, *eBits* can be any number between 8 and 2032. If *nBits* is 1024, we recommend *eBits* = 32.
p$*, *q$, *e$*	The private key (*p*, *q*, *e*).

RsaGenerateKey uses the Rabin-Miller primality test, as described in **IEEE P1363: Standard Specifications for Public Key Cryptography**. The number of Rabin-Miller rounds depends on *nBits*; it is chosen so that the probability of a given factor being composite is less than 1 in 2^{100}.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	In a BasicCard program: *nBits* < 496. In a Terminal program: *nBits* < 16.
RsaKeyTooLong	In a BasicCard program: *nBits* > 1024. In a Terminal program: *nBits* > 4064.
RsaBadProcParams	In a BasicCard program: *nBits* is not a multiple of 16, or *eBits* is not a multiple of 8, or *eBits* < 8, or *eBits* > 32. In a Terminal program: *eBits* < 8, or *eBits* > 2032.

To calculate the public key modulus *n* from *p* and *q*:

n$* = **RsaPublicKey (***p$, *q$***)**

The following error code is returned in the **LibError** variable:

RsaKeyTooLong	In a BasicCard program: *p$* or *q$* longer than 512 bits. In a Terminal program: *n$* longer than 2032 bits.

If you want to generate your own random numbers *p$* and *q$*, you can test them for primality with:

IsPrime = **RsaPseudoPrime (***x$*, *nRounds***)**

x$	Number to test for primality.
nRounds	Number of rounds of Rabin-Miller primality test to run. Unlike the corresponding function **RsaPseudoPrime** from the RSA-4096 Procedure Set, a value of zero is not allowed here.
IsPrime	**True** if *x$* survives *nRounds* rounds of the Rabin-Miller primality test.

7.1.11 RSA-1024 Cryptographic Primitives

Cryptographic primitives ***RSAEP*** *and* ***RSAVP1***

Call RsaEncrypt (*Mess$*, *n$*, *e$***)**

This procedure computes *Mess$* $^{e\$}$ modulo *n$*, returning the result in *Mess$*.

In a BasicCard program, the following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n$* is shorter than 248 bits
RsaKeyTooLong	*n$* is longer than 1024 bits
RsaBadProcParams	*Mess$* is longer than 1024 bits

*Cryptographic primitives **RSADP** and **RSASP1***

Call RsaDecrypt (*Mess$*, *p$*, *q$*, *e$***)**

This procedure first computes *d$* = inverse of *e$* modulo (*p$*-1)(*q$*-1). Then it computes *Mess$* $^{d\$}$ modulo *p$ q$*, returning the result in *Mess$*.

In a BasicCard program, the following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*p$* or *q$* is shorter than 248 bits
RsaKeyTooLong	*p$* or *q$* is longer than 512 bits
RsaBadProcParams	*Mess$* is longer than 1024 bits

7.1.12 RSA-1024 Encryption Scheme

To encrypt a message using the **RSAES-PKCS1-v1_5-ENCRYPT** encryption operation:

Call RsaPKCS1Encrypt (*Mess$*, *n$*, *e$***)**

Mess$	The message to be encrypted. It must be at least 11 bytes shorter than *n$*. The encrypted message is returned in *Mess$*.
n$*, *e$	The public key (*n*, *e*).

The following error code is returned in the **LibError** variable:

RsaBadProcParams	*Mess$* is not at least 11 bytes shorter than *n$*.

To decrypt a message using the **RSAES-PKCS1-v1_5-DECRYPT** decryption operation:

MessageValid = **RsaPKCS1Decrypt (***Mess$*, *p$*, *q$*, *e$***)**

Mess$	The message to be decrypted. It must be the same length as *n$* (where *n* = *pq* is the public-key modulus). The decrypted message is returned in *Mess$*.
p$*, *q$, *e$*	The private key (*p*, *q*, *e*).
MessageValid	**True** if *Mess$* was successfully decrypted.

The following error code is returned in the **LibError** variable:

RsaBadProcParams	*Mess$* is not the same size as *n$*.

7.1.13 RSA-1024 Signature Scheme

To generate a signature using the **RSASSA-PKCS1-v1_5-SIGN** signature generation operation:

Call RsaPKCS1Sign (*Hash$*, *p$*, *q$*, *e$*, *Sig$***)**

Hash$	The 20-byte **SHA-1** hash of the data to be signed.
p$, q$, e$	The private key (*p*, *q*, *e*).
Sig$	The signature calculated by **RsaPKCS1Sign**. It has the same size as *n$* (where *n* = *pq* is the public-key modulus).

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n$* is shorter than 376 bits
RsaBadProcParams	*Hash$* is not 20 bytes long

To verify a signature using the **RSASSA-PKCS1-v1_5-VERIFY** signature verification operation:

SignatureValid = **RsaPKCS1Verify (***Hash$*, *n$*, *e$*, *Sig$***)**

Hash$	The 20-byte **SHA-1** hash of the data that was signed.
n$, e$	The public key (*n*, *e*).
Sig$	The signature to be verified.
SignatureValid	**True** if the signature is valid.

The following error codes are returned in the **LibError** variable:

RsaKeyTooShort	*n$* is shorter than 376 bits
RsaBadProcParams	*Hash$* is not 20 bytes long

7.1.14 Fast Private-Key Operations

By default, all Private Key operations in **ZC7-** and **ZC8-**series BasicCards are protected against timing and power attacks using various obfuscation techniques. As a result, such operations are a little slower than they would be without these measures.

In certain contexts, such as card personalisation in a secure environment, these security measures may be unnecessary. So in **REV C** cards and above, you can disable these protection measures, temporarily or permanently:

Function RsaExSetFastPrKOps (*On%***)**

If *On%* is non-zero, fast Private Key operations are enabled (i.e. the security measures are disabled). If *On%* is zero (**False**), fast Private Key operations are disabled. The setting remains in force until **RsaExSetFastPrKOps** is called again, or the card is reset. The return value of the function is the current value of the setting.

The following error code is returned in the **LibError** variable:

RsaFastPrKOpsDisabled	*On%* is non-zero, but fast Private Key operations were disabled with **#Pragma RsaDisableFastPrKOps**.

Function RsaExGetFastPrKOps()

This function returns the current value of the setting: **True** (&HFFFF) if fast Private Key operations are enabled, **False** (0) otherwise.

To configure the card so that fast Private Key operations are automatically enabled when the card is reset (this is the least secure option):

#Pragma RsaFastPrKOps

To configure the card so that fast Private Key operations can never be enabled (this is the most secure option):

#Pragma RsaDisableFastPrKOps

7.2 The Elliptic Curve Library EC-p

Elliptic Curve Cryptography is a branch of Public Key Cryptography that is especially suitable for Smart Card implementation, for (at least) two reasons:

- the generation of private/public key pairs is very fast;
- it requires much smaller key sizes than other well-known methods for the same level of security.

There are two types of Elliptic Curve generally used in cryptography: curves over the *prime field* GF(p) for a prime number p; and curves over the *binary field* GF(2^n) for some integer n. Curves over the prime field GF(p) are implemented in the **EC-p** Library, available for **ZC7**- and **ZC8-**series BasicCards; curves over the binary field GF(2^n) are implemented in most other BasicCards – see **7.3 The Binary Elliptic Curve Libraries** for details.

A Smart Card executing Elliptic Curve arithmetic over the field GF(p) requires a co-processor if it is to combine a high level of security with fast response times. The **ZC7**- and **ZC8-**series BasicCards contain such a co-processor, making the **EC-p** Library the first choice for high-performance Elliptic Curve cryptography.

An element of the field GF(p) is simply an integer between 0 and p-1 inclusive. An Elliptic Curve E over GF(p) is defined by an equation of the form

$$y^2 = x^3 + Ax + B \bmod p$$

where A and B are elements of GF(p). The curve E consists of all points (x, y) with $x, y \in$ GF(p) 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. $\#E = kq$ for some (large) prime q. As well as A, B, and q, a point $P_0 \in E$ must be specified, of order q (that is, q is the smallest positive integer such that $[q]P_0 = \mathcal{O}$.) Field elements A and $B \in$ GF(p), integer q, and point $P_0 \in E$ constitute the *EC domain parameters*.

The **EC-p** Library supports the following Elliptic Curve operations:

- private/public key pair generation;
- session key generation;
- digital signature generation;
- digital signature verification;
- point addition and multiplication.

7.2.1 Available Curves

The **EC-p** Library offers nineteen pre-defined curves, with field sizes ranging from 160 bits to 521 bits:

- Curves 1-14 are the **Brainpool Standard Curves**, available in all **ZC7**-series cards. These curves have field sizes ranging from 160 bits to 512 bits. They are defined in the document **ECC Brainpool Standard Curves and Curve Generation v. 1.0**, which is available for download at http://www.ecc-brainpool.org/download/Domain-parameters.pdf.

- Curves 15-19 are the **NIST Recommended Elliptic Curves**, available in **ZC7**-series cards from **REV C**. These curves have field sizes ranging from 192 to 521 bits. They are defined in the document **FIPS PUB 186-3: Digital Signature Standard (DSS)**, which is available for download at http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf.

If you have to use another elliptic curve, you can specify its domain parameters in an Elliptic Curve Definition File. The format of this file is out of the scope of this document; if you would like to use your own EC domain parameters, contact ZeitControl for assistance. Example files `brainpoolp384r1.bin` (for Terminal programs) and `brainpoolp384r1.def` (for BasicCard programs) can be found in the directory `BasicCardV8\Lib\Curves`.

ZC7- and **ZC8-**series cards from **REV C** accept any Elliptic Curve Definition File that satisfies two conditions:

- the bit length of p is between 160 and 544 inclusive;
- #*E*, the order of *E*, is a prime number *q* of the same bit length as *p*.

ZC7-series **REV A** and **REV B** cards will only accept curves that satisfy the following conditions, as recommended in the Brainpool document:

- the bit length of *p* is a multiple of 32 between 160 and 512 inclusive;
- *p* is equal to 3 modulo 4;
- EC domain parameter *A* is chosen so that the equation $AZ^4 = -3 \bmod p$ has a solution Z;
- EC domain parameter *B* is not a square mod *p*;
- #*E*, the order of *E*, is a prime number *q* of the same bit length as *p*, with $q < p$.

7.2.2 Loading the EC-p Library

To load the **EC-p** Elliptic Curve Library:

#Include EC–p.DEF

The file EC-p.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

7.2.3 Selecting a Curve

Before calling any other procedures from the **EC-p** Library, a curve must be selected. To select one of the nineteen pre-defined curves:

Call ECpSetCurve (*CurveIndex***)**

where *CurveIndex* is an integer from 1 to 19:

CurveIndex	*Bit length of p*	*Curve ID in defining document*
1	160	`brainpoolP160r1`
2	160	`brainpoolP160t1`
3	192	`brainpoolP192r1`
4	192	`brainpoolP192t1`
5	224	`brainpoolP224r1`
6	224	`brainpoolP224t1`
7	256	`brainpoolP256r1`
8	256	`brainpoolP256t1`
9	320	`brainpoolP320r1`
10	320	`brainpoolP320t1`
11	384	`brainpoolP384r1`
12	384	`brainpoolP384t1`
13	512	`brainpoolP512r1`
14	512	`brainpoolP512t1`
15	192	`P-192`
16	224	`P-224`
17	256	`P-256`
18	384	`P-384`
19	521	`P-521`

Or you can specify an Elliptic Curve Definition File containing a set of EC domain parameters:

Call ECpSetCurveFromFile (*Filename$***)**

The EC domain parameters are loaded from the specified Elliptic Curve definition file.

To retrieve the bit length of the modulus *p* for the currently selected curve:

Function ECpBitLength()

7.2.4 Key Generation

In the **EC-p** library, a private key is an integer r with $0 < r < q$, where q is the order $\#E$ of the curve. It has the same size as p.

The corresponding public key is the point $K = [r]P_0$. A public key has two formats: the *expanded* format is of the form (x, y), where x and y are positive integers less than p; the *compressed* format is of the form $(x, \tilde{y})$, where $\tilde{y}$ is equal to the least significant bit of y. The expanded format is twice the size of p; the compressed format is one byte longer than p.

The expanded format is used internally, but the expanded or the compressed format can be passed to all the library procedures that require a public key; the library automatically converts to the expanded format if required. However, conversion to expanded format is a time-consuming operation. If speed is important, then public keys should be stored in expanded format. This requires more storage space, but unless many different keys have to be stored, the space overhead is small.

To generate a Key Pair consisting of a private key and a public key

Call ECpGenerateKeyPair (*PrK$*, *PuK$***)**

The private key is returned in *PrK$* and the public key is returned in *PuK$*, in expanded format.

To construct a public key from a private key

Sub ECpMakePublicKey (*PrK$*, *PuK$***)**

The public key corresponding to private key *PrK$* is returned in *PuK$*, in expanded format.

To convert a public key to compressed format

Sub ECpPackPublicKey (*PuK$***)**

Public key *PuK$* is converted to compressed format. If *PuK$* is already compressed, it is returned unchanged.

To convert a public key to expanded format

Sub ECpUnpackPublicKey (*PuK$***)**

Public key *PuK$* is converted to expanded format. If *PuK$* is already expanded, it is returned unchanged.

7.2.5 Session Key Generation

If two parties know each other's public keys, they can use them to agree on a secret value, the same size as the Elliptic Curve modulus p. This value is called the *shared secret* for the two parties; to compute it, you need to know the private key of one party and the public key of the other party. To compute the shared secret:

Call ECpSharedSecret (*PrK$*, *PuK$*, *Secret$***)**

PrK$	The private key of one party
PuK$	The public key of the other party
Secret$	The resulting shared secret

This shared secret can then be used to generate session keys for encrypting messages between the two parties; unlike the shared secret, a session key can be different on different occasions. **EC–p** uses the **SHA–256** algorithm to generate 32-byte session keys.

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$* = **ECpSessionKey (***KDP$***,** *SharedSecret$***)**

KDP$	Key Derivation Parameter, a string of any length
SharedSecret$	The shared secret value, returned by **ECpSharedSecret**
SessionKey$	The 32-byte or 20-byte session key

7.2.6 Generating a Digital Signature

Two Digital Signature variants are available in the **EC-p** Library: Nyberg-Rueppel (**NR**), and Digital Signature Algorithm (**DSA**). There is no practical difference between these two variants; they are provided to enhance compatibility with other systems.

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

*Signature$* = **ECpHashAndSignNR (***PrivateKey$***,** *Data$***)**
*Signature$* = **ECpHashAndSignDSA (***PrivateKey$***,** *Data$***)**

Signature$ The signature of the data. It is twice as long as the Elliptic Curve modulus *p*.
PrivateKey$ The signer's private key.
Data$ The data to be signed.

These functions use hash algorithm SHA-256.

To sign a longer body of data, first compute the hash function for the data (see **7.11.1 Hashing Functions**), and then:

Call ECpSignNR (*Hash$*, *PrK$*, *Sig$***)**
Call ECpSignDSA (*Hash$*, *PrK$*, *Sig$***)**

Hash$ The hash of the data to be signed: 20 bytes for SHA-1, or 32 bytes for SHA-256.
PrK$ The signer's private key.
Sig$ The signature of the data.

7.2.7 Verifying a Digital Signature

To verify a digital signature:

Status **= ECpHashAndVerifyNR (***Signature$***,** *Data$***,** *PublicKey$***)**
Status **= ECpHashAndVerifyDSA (***Signature$*, *Data$*, *PublicKey$***)**

Status **True** (&HFFFF) if the signature is valid, otherwise **False** (0).
Signature$ The signature to be verified.
Data$ The data that was signed.
PublicKey$ The signer's public key.

These functions assume that the signature was computed using hash algorithm SHA-256.

To verify a longer body of data, first compute the hash function for the data (see **7.11.1 Hashing Functions**), and then:

Function ECpVerifyNR (*Hash$*, *PuK$*, *Sig$***)**
Function ECpVerifyDSA (*Hash$*, *PuK$*, *Sig$***)**

Hash$ The hash of the data that was signed.
PuK$ The signer's public key.
Sig$ The signature to be verified.

These functions return **True** or **False** according to whether the signature is valid or not.

7.2.8 Point Addition and Verification

ZC7- and **ZC8-**series cards from **REV D** support low-level addition and multiplication of points on the Elliptic Curve. This is required to implement certain widely-used protocols, including the **Password Authenticated Connection Establlishment** (**PACE**) described in the BSI document available at **this link**. Two procedures are provided:

Sub ECpAddPoints (*P$***,** *Q$***)**

P$, Q$ Points on the current Elliptic Curve, in compressed or expanded format

Computes *P$ = P$ + Q$*. The result *P$* is in expanded format.

Sub ECpMultiplyPoint (*P$*, *n$*)

P$ A point on the current Elliptic Curve, in compressed or expanded format
n$ A non-negative integer in Big-Endian format (most significant byte first)

Computes *P$* = [*n$*]*P$*. The result *P$* is in expanded format.

7.2.9 Conformance Specification

The **EC-p** Library follows the standard **IEEE P1363: Standard Specifications for Public Key Cryptography**. In the terminology of this standard, the following schemes, primitives, and additional techniques are implemented:

Scheme	*Description*
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**, **ECVP-NR**, **ECSP-DSA**, and **ECVP-DSA**, and additional technique **EMSA1**.

Primitive	*Description*
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.
ECSP-DSA	Elliptic Curve Signature Primitive, DSA version.
ECVP-DSA	Elliptic Curve Verification Primitive, DSA version.

Additional Technique	*Description*
KDF1	Key Derivation Function. The hash function is **SHA–256: Secure Hash Standard**.
EMSA1	Encoding Method for Signatures with Appendix. The hash functions are **SHA–256: Secure Hash Standard** and **SHA–1: Secure Hash Algorithm Revision 1**.

7.3 The Binary Elliptic Curve Libraries

A Binary Elliptic Curve is an elliptic curve over the binary field GF(2^n) for some n. In Smart Cards without a specialised co-processor, arithmetic in a prime field GF(p) is too slow, but binary field arithmetic is possible. So Enhanced BasicCards, **ZC5**-series Professional BasicCards, and MultiApplication BasicCards provide Binary Elliptic Curve libraries. **ZC7-** and **ZC8-**series cards from **REV D** also implement the Binary Elliptic Curve libraries, using specialised co-processor hardware. Three libraries are available:

- Library **EC–211** over the field GF(2^{211}), with 211-bit keys. This is currently considered equivalent to 2048-bit **RSA**. It is available for all **ZC5**-series Professional BasicCards, **ZC6**-series MultiApplication BasicCards, and **ZC7-** and **ZC8-**series cards from **REV D**.
- Library **EC–167** over the field GF(2^{167}), with 167-bit keys. This is currently considered equivalent to 1024-bit **RSA**. It is available for all **ZC5**-series Professional BasicCards, **ZC6**-series MultiApplication BasicCards, and **ZC7-** and **ZC8-**series cards from **REV D**.
- Library **EC–161** over the field GF(2^{168}), with 161-bit keys. See below for a discussion of the security of this library compared to the **EC–167** library. It is available for all Enhanced BasicCards.

All three libraries are available to Terminal programs.

The important difference between libraries **EC–167** and **EC–161** is not the key length (167 vs. 161), but the field exponent (167 vs. 168). In a Smart Card implementation of Elliptic Curve Cryptography, arithmetic over the underlying field must be made as fast as possible. Certain field exponents allow ingenious short cuts, speeding up the arithmetic significantly. One such exponent is 168, as used by **EC–161**. Our implementation achieves a speed-up factor of five or six; without this speed-up, Elliptic Curve Cryptography in the Enhanced BasicCard would be too slow for practical use.

However, the latest consensus among experts is that the field exponent should be a prime number, such as 211 or 167; certain composite exponents have been shown to be cryptographically weak, and the feeling is that all composite exponents (for example, 168) should therefore be avoided. So current expert opinion would not recommend library **EC–161** for applications requiring maximum security.

Each library supports the following Elliptic Curve operations:

- private/public key pair generation;
- session key generation;
- digital signature generation;
- digital signature verification (not available in the Enhanced BasicCard).

Our implementation follows the standard **IEEE P1363: Standard Specifications for Public Key Cryptography**. Section **7.3.9 Conformance Specification** specifies the methods used in the Elliptic Curve libraries, using the terminology of **IEEE P1363**.

A simple Elliptic Curve application can be found in the directory `BasicCardV8\Examples\EC`.

7.3.1 Loading an Elliptic Curve Library

To load an Elliptic Curve library:

#Include EC–*XXX*.DEF

(we use *XXX*, here and later, to denote any of 211, 167, or 161). These files are supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

7.3.2 Setting the Elliptic Curve Parameters

An Elliptic Curve is defined by its EC Domain Parameters; suitable Elliptic Curves are supplied in the directory `BasicCardV8\Lib\Curves`. Choose one of these for your application. We supply five Elliptic Curves for libraries **EC–211** and **EC–167**, and three Elliptic Curves for library **EC–161**. The *Curve Definition Files* EC211-1.16 through EC211-5.128, EC167-1.16 through EC167-5.128, and EC161-1.16 through EC161-5.64 contain curve definitions in ZC-Basic, for inclusion in a source program. File EC-*XXX*.BIN contains the binary data for all the curves for a given library, for run-time loading in a Terminal program.

Specifying an Elliptic Curve in a Enhanced or Professional BasicCard program

To specify the EC Domain Parameters to be used in an Enhanced or Professional BasicCard program:

#Include Curves\EC*XXX–C***.***N*

where *C* is a curve number from 1 to 5 (from 1 to 3 for library **EC–161**), and *N* is a power of 2 between 16 and 128 (between 16 and 64 for library **EC–161**). In an Enhanced or Professional BasicCard program, the curve must be chosen at compile time; it can't be re-loaded at run-time. This Curve Definition File loads *N* pre-computed Elliptic Curve points into EEPROM to speed up Elliptic Curve operations. The more pre-computed points, the faster the card, but the less free EEPROM space. If EEPROM space is at a premium, use 16 pre-computed points; if speed is the most important factor, use 64 or 128 pre-computed points.

Specifying an Elliptic Curve in a MultiApplication BasicCard program

To specify the EC Domain Parameters to be used in a MultiApplication BasicCard program:

Call EC*XXX***SetCurve (** *filename$***)**

where *filename$* is the name of a file in the BasicCard that contains the same data as one of the **Curves\EC***XXX–C***.***N* curve definition files. The data in this file must occupy a single contiguous data block in EEPROM. See the preceding paragraph for the meaning of *C* and *N*.

For example:

```
Dir "\ECApp" ' Start File Definition Section
  File "CurveParams" Len=0 ' Len=0 makes single contiguous block
    Lock=Read:Always  ' Read-only access for everybody
    #Include "Curves\EC211-2.64" ' Import file data
End Dir Lock=Read:Always ' End File Definition Section

Call EC211SetCurve ("\ECApp\CurveParams")
If LibError <> 0 Then ' Report error
...
```

Alternatively, if the special file "**ECDomainParams**" exists in the Root Directory, it is automatically loaded whenever the card is reset – see **5.4.3 Elliptic Curve Domain Parameters**.

Specifying an Elliptic Curve in a Terminal program

In the Terminal program, an Elliptic Curve must be explicitly loaded using **EC***XXX***SetCurve**. 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 EC211-3.16
  Call EC211SetCurve (EC211Params)
  ```

  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 EC167DomainParams
  Call GetCurve (Curve) : Call CheckSW1SW2()
  Call EC167SetCurve (Curve)
  ```

  See `BasicCardV8\Examples\EC` for an example of this.

- You can read the curve from binary files EC-*XXX*.BIN. For example:

  ```
  Private Curve As EC161DomainParams
  Open "EC-161.BIN" For Random As #1 Len=Len(EC161DomainParams)
  Get #1, 2, Curve ' Read Elliptic Curve #2
  Close #1
  Call CheckFileError()
  Call EC161SetCurve (Curve)
  ```

If the EC domain parameters are invalid, procedure **EC*XXX*SetCurve** returns error code **EC*XXX*BadCurveParams** in variable **LibError**.

In a Terminal program or a MultiApplication BasicCard program, you must call **EC*XXX*SetCurve** before you call any other procedures from the **EC–*XXX*** library. If not, error code **EC*XXX*CurveNotInitialised** will be returned in variable **LibError**.

7.3.3 Key Generation

To generate a public/private key pair:

Case 1: Terminal and single-application BasicCard programs

Call EC*XXX*GenerateKeyPair()

Case 2: MultiApplication BasicCard program

Call EC*XXX*GenerateKeyPair (*PrivateKey$*, *PublicKey$*)

This procedure generates a random private key and its associated public key, storing them in **Eeprom** strings **EC*XXX*PrivateKey** and **EC*XXX*PublicKey** (Case 1) or in the procedure parameters *PrivateKey$* and *PublicKey$* (Case 2). The **EC–211** library generates 27-byte private and public keys; the **EC–167** library generates 21-byte private and public keys; and the **EC–161** library generates a 21-byte private key and a 22-byte public key.

7.3.4 Computing a Public Key from a Private Key

Case 1: Terminal and single-application BasicCard programs

Call EC*XXX*SetPrivateKey (*PrivateKey$*)

This procedure copies *PrivateKey$* (reduced modulo *r*) to the **Eeprom** string **EC*XXX*PrivateKey**, and computes the associated **Eeprom** string **EC*XXX*PublicKey**. (*r* is explained in **7.3.8 Binary Representation Formats**: *EC Domain Parameters*.) Key lengths are as described in the previous paragraph, **7.3.3 Key Generation**.

Case 2: MultiApplication BasicCard program

PublicKey$* = **EC*XXX*MakePublicKey (*PrivateKey$)**

This function computes the public key from a specific private key.

If *PrivateKey$* is zero modulo *r*, error code **EC*XXX*BadProcParams** is returned in variable **LibError**.

7.3.5 Generating a Digital Signature

Two Digital Signature variants are available in most libraries: Nyberg-Rueppel (**NR**), and Digital Signature Algorithm (**DSA**). There is no practical difference between these two variants that we are aware of; they are provided to enhance compatibility with other systems. Different cards implement a different set of procedures:

- The Enhanced BasicCards implement only the 161-bit Nyberg-Rueppel procedures **EC161HashAndSignNR** and **EC161SignNR**.
- All other cards (**ZC5.4**, **ZC5.5**, **ZC5.6**, **ZC6.5**) implement the 167-bit Nyberg-Rueppel procedures **EC167HashAndSignNR** and **EC167SignNR**.
- The remaining procedures (167-bit DSA, 211-bit Nyberg-Rueppel, and 211-bit DSA) are implemented in the following cards:
  Professional BasicCard **ZC5.5** from **REV H**
  Professional BasicCard **ZC5.6** (all revisions)
  MultiApplication BasicCard **ZC6.5** from **REV D**

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

Case 1: Terminal and single-application BasicCard programs

Signature$* = **EC*XXX*HashAndSignNR (*Data$)**
Signature$* = **EC*XXX*HashAndSignDSA (*Data$)**

Case 2: MultiApplication BasicCard program

*Signature$* = **EC***XXX***HashAndSignNR (***PrivateKey$***,** *Data$***)**
*Signature$* = **EC***XXX***HashAndSignDSA (***PrivateKey$***,** *Data$***)**

The **EC–211** library returns a 54-byte signature; libraries **EC–167** and **EC–161** return a 42-byte signature.

To sign a longer body of data, first compute the hash function for the data (see **7.11.1 Hashing Functions**), and then:

Case 1: Terminal and single-application BasicCard programs

> *Signature$* = **EC***XXX***SignNR (***Hash$***)**
> *Signature$* = **EC***XXX***SignDSA (***Hash$***)**

Case 2: MultiApplication BasicCard program

> *Signature$* = **EC***XXX***SignNR (***PrivateKey$***,** *Hash$***)**
> *Signature$* = **EC***XXX***SignDSA (***PrivateKey$*, *Hash$***)**

In Case 1, if no private key has been set, these procedures return error code **EC***XXX***KeyNotInitialised** in variable **LibError**.

7.3.6 Verifying a Digital Signature

Different cards implement a different set of verification procedures:

- Elliptic Curve verification is not available in the Enhanced BasicCards, or in Professional BasicCard **ZC5.4 REV A-G**.
- All other cards implement the 167-bit Nyberg-Rueppel procedures **EC167HashAndVerifyNR** and **EC167VerifyNR**.
- The remaining procedures (167-bit DSA, 211-bit Nyberg-Rueppel, and 211-bit DSA) are implemented in the following cards:
  Professional BasicCard **ZC5.5** from **REV H**
  Professional BasicCard **ZC5.6** (all revisions)
  MultiApplication BasicCard **ZC6.5** from **REV D**

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 **= EC***XXX***HashAndVerifyNR (***Signature$***,** *Message$***,** *PublicKey$***)**
Status **= EC***XXX***HashAndVerifyDSA (***Signature$***,** *Message$***,** *PublicKey$***)**

Signature$	The signature to be verified: 54 bytes (**EC–211**) or 42 bytes (**EC–167** and **EC–161**)
Message$	The message that was signed
PublicKey$	The signer's public key: 27 bytes (**EC–211**), 21 bytes (**EC–167**), 22 bytes (**EC–161**)

These functions return **True** or **False** according to whether the signature is valid or not.

To verify a longer message, first compute the hash function for the message (see **7.11.1 Hashing Functions**), and then verify its signature with the function:

Status **= EC***XXX***VerifyNR (***Signature$***,** *Hash$***,** *PublicKey$***)**
Status = **EC***XXX***VerifyDSA (***Signature$*, *Hash$*, *PublicKey$***)**

If *Signature$* or *PublicKey$* are not the correct length, error code **EC***XXX***BadProcParams** is returned in variable **LibError**.

7.3.7 Session Key Generation

If two parties know each other's public keys, they can use them to agree on a secret value, 27 bytes long for the **EC–211** library and 21 bytes long for the **EC–167** and **EC–161** libraries. This value is called the *shared secret* for the two parties; to compute it, you need to know the private key of one party and the public key of the other party. To compute the shared secret:

Case 1: Terminal and single-application BasicCard programs

*SharedSecret$* = **EC***XXX***SharedSecret (***PublicKey$***)**

Case 2: MultiApplication BasicCard program

*SharedSecret$* = **EC***XXX***SharedSecret (***PrivateKey$***,** *PublicKey$***)**

PrivateKey$	The known private key (in Case 1, this must be in **EC***XXX***PrivateKey**)
PublicKey$	The other party's public key
SharedSecret$	The shared secret

If *PublicKey$* is not the correct length, or it is not a point on the curve, error **EC***XXX***BadProcParams** is returned in variable **LibError**.

This shared secret can then be used to generate session keys for encrypting messages between the two parties; unlike the shared secret, a session key can be different on different occasions. **EC–211** uses the **SHA–256** algorithm to generate 32-byte session keys; **EC–167** and **EC–161** use the **SHA–1** algorithm to generate 20-byte session keys.

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$* = **EC***XXX***SessionKey (***KDP$***,** *SharedSecret$***)**

KDP$	Key Derivation Parameter, a string of any length
SharedSecret$	The shared secret value, returned by **EC***XXX***SharedSecret**
SessionKey$	The 32-byte or 20-byte session key

Note: Generating a shared secret is a complicated calculation, which can take several seconds in some BasicCards. But once a shared secret has been generated for a given public key, session key generation is mush faster, especially if **Len(***KDP$***)** + **Len(***SharedSecret$***)** $\leq$ 55. (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.)

7.3.8 Binary Representation Formats

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

Integers

Integers in this implementation have a length of either 1 byte, 21 bytes, or 27 bytes. The first (or leftmost) byte is the most significant – in a 27-byte integer, it contains bits 215-208; in a 21-byte integer, it contains bits 167-160. The last (or rightmost) byte contains bits 7-0.

Field Elements

The library **EC–211** implements operations on Elliptic Curves over the field **GF**(2^{211}). An element of **GF**(2^{211}) is represented by 211 bits stored in 27 bytes. A Polynomial Basis field representation is used; the Field Polynomial is

$$p(t) = t^{211} + t^{11} + t^{10} + t^{8} + 1$$

The first (leftmost) byte contains the coefficients of t^{210} and t^{209}.

The library **EC–167** implements operations on Elliptic Curves over the field **GF**(2^{167}). An element of **GF**(2^{167}) is represented by 167 bits stored in 21 bytes. A Polynomial Basis field representation is used; the Field Polynomial is

$$p(t) = t^{167} + t^{6} + 1$$

The first (leftmost) byte contains the coefficients of t^{166} through t^{160}.

The library **EC–161** implements operations on Elliptic Curves over the field **GF**(2^{168}). An element of **GF**(2^{168}) 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 `BasicCardV8\Source\FldConv`.

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 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 $rG =$ O.) Field elements a and $b \in$ **GF**(2^m), integers r and k, and point $G \in E$ constitute the *EC domain parameters*.

The library **EC–211** accepts any set of EC domain parameters (a, b, r, k, G) satisfying the following:

- a is zero in all bit positions except for bits 7-0 ;
- r is exactly 211 bits long, i.e. $2^{210} < r < 2^{211}$;
- k is equal to 2.

The user-defined type **EC211DomainParams**, defined in file `BasicCardV8\Lib\EC-211.DEF`, contains curve parameters a (1 byte), b (27 bytes), r (27 bytes), and G (27 bytes), for a total of 82 bytes.

The library **EC–167** accepts any set of EC domain parameters (a, b, r, k, G) satisfying the following:

- a is zero in all bit positions except for bits 7-0 ;
- r is exactly 167 bits long, i.e. $2^{166} < r < 2^{167}$;
- k is equal to 2.

The user-defined type **EC167DomainParams**, defined in file `BasicCardV8\Lib\EC-167.DEF`, contains curve parameters a (1 byte), b (21 bytes), r (21 bytes), and G (21 bytes), for a total of 64 bytes.

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 78-72 ;
- 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 `BasicCardV8\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 Elliptic Curve cryptography:

- 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 $y^2 + x_P y = x_P^3 + ax_P^2 + b$ 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 0 (in the Polynomial Basis representation); so if we know x_P and bit 0 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 $\tilde{y}_P$.

A point P on a curve over **GF**(2^{211}) is represented by 27 bytes, with $\tilde{y}_P$ in bit 215, and x_P in bits 210-0.

A point P on a curve over **GF**(2^{167}) is represented by 21 bytes, with $\tilde{y}_P$ in bit 167, and x_P in bits 166-0.

A point P on a curve over **GF**(2^{168}) is represented by 22 bytes, with x_P in the leftmost 21 bytes (i.e. in bits 175-8), and $\tilde{y}_P$ in bit 0.

Signatures

A signature consists of two integers (c, d). Each of these integers is 27 bytes long in library **EC–211**, and 21 bytes long in libraries **EC–167** and **EC–161**, for a total signature length of 54 or 42 bytes. See **IEEE P1363** for the definitions of c and d.

7.3.9 Conformance Specification

The Binary Elliptic Curve Libraries follow the standard **IEEE P1363: Standard Specifications for Public Key Cryptography**. In the terminology of this standard, the following schemes, primitives, and additional techniques are implemented:

Scheme	*Description*	*Terminal*	*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 **ECVP-NR** (in the Terminal only), and additional technique **EMSA1**.	✓	✓

Primitive	*Description*	*Terminal*	*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.	✓	**Not Enhanced BasicCard**

Additional Technique	*Description*	*Terminal*	*BasicCard*
KDF1	Key Derivation Function. The hash function is **SHA–256: Secure Hash Standard** for library **EC–211**, and **SHA–1: Secure Hash Algorithm Revision 1** for libraries **EC–167** and **EC–161**.	✓	✓
EMSA1	Encoding Method for Signatures with Appendix. The hash function is **SHA–256: Secure Hash Standard** for library **EC–211**, and **SHA–1: Secure Hash Algorithm Revision 1** for libraries **EC–167** and **EC–161**.	✓	✓

7.4 The COMPONENT Library

This library is available in the MultiApplication BasicCard, and in the Terminal Program. See **Chapter 5: The MultiApplication BasicCard** for information on Components. Procedures in the **COMPONENT** library report errors via the **LibError** variable; a list of error codes (beginning **ce...**) can be found in **Componnt.def**. In Terminal programs, errors are also reported via **SW1SW2**. The corresponding error codes can be found in Commands.def.

To use the **COMPONENT** library:

#Include Componnt.def

The following procedures for handling Security Components are provided:

Sub SelectApplication (*filename$***)**

Select the Application contained in the given file. **Execute** access to the file is required. See **8.9.20 The SELECT APPLICATION Command** for further information.

Sub CreateComponent (*type@*, *name$*, *attr$*, *data$***)**

Create a Component. **Write** access is required to the parent directory. *name$* can be empty, if an anonymous ACR is being created. The formats of *attr$* and *data$* depend on the type of the Component; they are described in **5.9 Component Details**. See **8.9.21 The CREATE COMPONENT Command** for further information.

Sub DeleteComponent (*CID%***)**

Delete a Component. **Delete** access to the Component is required. See **8.9.22 The DELETE COMPONENT Command** for further information.

Sub WriteComponentAttr (*CID%*, *attr$***)**

Write a Component's Attributes. Both **Write** and **Delete** access to the Component are required. The format of *attr$* depends on the type of the Component; it is described in **5.9 Component Details**. See **8.9.23 The WRITE COMPONENT ATTR Command** for further information.

Function ReadComponentAttr (*CID%***) As String**

Read a Component's attributes. **Read** access to the Component's parent directory is required (but not **Read** access to the Component itself). The format of the returned string depends on the type of the Component; it is described in **5.9 Component Details**. See **8.9.24 The READ COMPONENT ATTR Command** for further information.

Sub WriteComponentData (*CID%*, *data$***)**

Write a Component's Data. **Write** access to the Component is required. See **8.9.25 The WRITE COMPONENT DATA Command** for further information.

Function ReadComponentData (*CID%***) As String**

Read a Component's data. **Read** access to the Component is required. See **8.9.26 The READ COMPONENT DATA Command** for further information.

Function FindComponent (*type@*, *name$***) As Integer**

Find a Component of a given type, and return its CID. Just like a filename, *name$* can be a full pathname (beginning with a backslash character) or a relative pathname (relative to the current directory). **Read** access is required to all directories in the path (but not to the Component itself). See **5.1 Components** for details. If the Component does not exist, **LibError** is set to **ceComponentNotFound**.

See **8.9.27 The FIND COMPONENT Command** for further information.

Function ComponentName (*CID%***) As String**

Return the full pathname of the Component with the given CID. **Read** access is required to all directories in the path (but not to the Component itself). See **8.9.28 The COMPONENT NAME Command** for further information.

Sub GrantPrivilege (*CID%*, *filename$***)**

Grant the Privilege with the given CID to the specified file. Requires **Grant** access to the Privilege, and **Write** access to the file. The Privilege is added to the file's Rights List. The file will typically be an Application, although this is not required.

If *filename$* is an empty string, the Privilege is granted to the Terminal program, and lasts until the card is reset. The Terminal program may possess up to three Privileges at once.

See **8.9.29 The GRANT PRIVILEGE Command** for further information.

Function AuthenticateFile (*KeyCID%*, *Algorithm@*, *Filename$*, *Signature$***) As Integer**

Authenticate a file with the given Key, using OMAC or EC-167 Elliptic Curve Cryptography. The Key is added to the file's Rights List. See **8.9.30 The AUTHENTICATE FILE Command** for further information.

Function ReadRightsList (*Filename$***,** *RightsList%*( )**) As Integer**

Read the Rights List of the given file into an array. The Rights List contains the CID's of the Privileges granted to the file, and the Keys with which the file has been authenticated. See **8.9.31 The READ RIGHTS LIST Command** for further information.

Sub LoadSequence (*Phase@***)**

Start or finish a Loader Sequence transaction. *Phase@* is equal to **LoadSequenceStart**, **LoadSequenceEnd**, or **LoadSequenceAbort** (defined in **Componnt.def**). If *Phase@* is equal to **LoadSequenceAbort**, and no **LoadSequenceEnd** has intervened, then all Components created since **LoadSequenceStart** are deleted. See **8.9.32 The LOAD SEQUENCE Command** for further information.

Sub SecureTransport (*KeyCID%*, *Algorithm@*, *Nonce$***)**

If *KeyCID%* is non-zero, start Secure Transport; if *KeyCID%* is zero, end Secure Transport. This procedure is available in the Terminal program only. See **8.9.33 The SECURE TRANSPORT Command** for further information.

7.5 The TMLib Transaction Manager Library

The Transaction Manager System Library provides a method of saving several EEPROM data items as an uninterruptable transaction. It is available only in **ZC7**-series BasicCards from **REV C**.

All BasicCards contain an automatic EEPROM Transaction Manager, which ensures that file operations, and changes to EEPROM data items, occur as a single transaction: this transaction will never be half-completed, which might leave the card's EEPROM in an inconsistent state. It works like this:

1. The card prepares a Transaction Log in EEPROM, which contains the write operations to be performed.
2. The card writes to a single flag-byte in EEPROM, which activates the Transaction Log.
3. The card performs the write operations on the Transaction Log.
4. The card clears the flag-byte, to deactivate the Transaction Log.

Then if Step 3 is interrupted, the flag-byte will remain set; so that the next time the card is powered up, it knows to complete Steps 3 and 4 before continuing.

Starting from **ZC7**-series **REV C** BasicCards, this functionality is available in a more general form, allowing the programmer to specify the contents of a sequence of EEPROM data items to be written as an uninterrupted unit.

To use the **TMLib** Transaction Manager library:

#Include TMLib.def

The library contains just two procedures:

Sub TMAddTransactionEntry (*Transaction$***, ReadOnly** *Dest$***, ReadOnly** *Src$***)**

Add a Transaction Entry to the *Transaction$* string. To build a list of Transaction Entries, first set *Transaction$* to the empty string, and then add entries one by one using **TMAddTransactionEntry**. When the *Transaction$* string is complete, pass it to **TMCommitTransaction** to execute the entries.

Dest$* and *Src$ are strings, but you can write to any type of EEPROM data using Type Casting (see **3.11 Type Casting** for details). For instance, to write 99 to the **Integer** array entry **A(I)**:

Call TMAddTransactionEntry (*Transaction$***, A(I) As String, 99 As String*Integer)**

Here, **String*Integer** is short for **String*Len(Integer)**, i.e. **String*2**.

Error Codes
Global variable **LibError** can take the following values (defined in **TMLib.def**):

TMTransactionTooLong	*Transaction$* would exceed the maximum allowed string length.
TMNotInEeprom	*Dest$* is not in EEPROM.

Sub TMCommitTransaction (ReadOnly *Transaction$***)**

Commit and execute the Transaction Entries stored in the *Transaction$* string. If this function returns to the caller, then all the entries have been successfully executed. If, however, the card loses power for whatever reason during this operation, then the unfinished entries will be automatically executed the next time the card is powered up.

Error Codes
Global variable **LibError** can take the following values (defined in TMLib.def):

TMInvalidTransaction	*Transaction$* is not a valid sequence of Transaction Entries.
TMTooManyStrings	*Transaction$* contains more than 32 variable-length string entries.
TMOutOfMemory	The EEPROM heap is full.

Notes

1. If the *Transaction$* string was not created by **TMAddTransactionEntry** as described above, then the results are undefined, and the card may be rendered unusable.
2. If the *Dest$* parameter to any of the **TMAddTransactionEntry** calls was an element of a dynamic array, then this array must not be re-sized before calling **TMCommitTransaction**. This would result in corruption of the EEPROM heap, rendering the card unusable.

7.6 The Crypto Library

The **Crypto** System Library contains procedures for computing Message Authentication Checksums (**MAC**s), encrypting messages, and configuring ISO Secure Messaging. It is available in the **ZC7**-series BasicCard from **REV C**, and in Terminal programs. To use the library:

#Include Crypto.def

Throughout this library, the following constants (defined in **Crypto.def**) are used to specify a Block Cipher Algorithm:

Name	*Value*	*Algorithm*	*Key Length*	*Block Length*
CryptoAlgSingleDES	**&H11**	Single DES	8	8
CryptoAlg2TDES	**&H12**	2-key Triple DES	16	8
CryptoAlg3TDES	**&H13**	3-key Triple DES	24	8
CryptoAlgAES128	**&H21**	128-bit AES	16	16
CryptoAlgAES192	**&H22**	192-bit AES	24	16
CryptoAlgAES256	**&H23**	256-bit AES	32	16

7.6.1 DES Key Parity

The least significant bit of every byte in a DES key is ignored by the DES algorithm, and may optionally be used as a parity bit. The DES standard specifies odd parity for this purpose. The BasicCard software ignores these parity bits, but two simple procedures are available for checking and setting them:

Function CryptoCheckDESKeyParity (ReadOnly *Key$***)**

Returns **True** (–1) if every byte in *Key$* has odd parity, **False** (0) otherwise.

Sub CryptoSetDESKeyParity (*Key$***)**

Sets or clears the least significant bit of every byte in *Key$*, to ensure odd parity.

7.6.2 Message Authentication Code

These procedures calculate a Message Authentication Code (MAC) for a string or a sequence of strings. The *Algorithm%* parameter is a combination of the following options:

- a Block Cipher Algorithm in bits 6-1 (as specified in the introduction to this library)
- an Authentication Mode in bits 11-9:

MacModeCBCMAC	**&H0100**
MacModeRetailMAC	**&H0200**
MacModeCMAC	**&H0300**
MacModeEMAC	**&H0400**

- an Initial Value Mode in bits 14-13:

CryptoIVPlain	**&H1000**
CryptoIVEncrypted	**&H2000**
CryptoIVNone	**&H3000**

- a Padding Suppression indicator in bit 16:

CryptoNoPadding	**&H8000**

Authentication Mode **MacModeCMAC** is the same as the **OMAC** algorithm – see **7.10 The OMAC Library**. In this mode, the Initial Value Mode must be **CryptoIVNone (**in which case the regular **OMAC** is computed) or **CryptoIVEncrypted** (in which case the **OMAC** of *IV$+Data$* is computed).

To compute a Message Authentication Code in other Authentication Modes:

1. The *Data$* string is padded to a multiple of the block length of the Block Cipher Algorithm (8 or 16 bytes – see the table above): the padding starts with a byte equal to **&H80** (unless the **CryptoNoPadding** bit is set in *Algorithm%*); then zero or more **&H00** bytes are appended.
2. *Data$* is split into blocks of 8 or 16 bytes: *Data$* = $\mathbf{M}_1 + \dots + \mathbf{M}_n$

3. *MAC* is set to **0...0** if the Initial Value Mode is **CryptoIVNone**; otherwise *MAC* is set equal to *IV$*, and then encrypted with *Key$* if the Initial Value Mode is **CryptoIVEncrypted**.
4. For $1 \le i \le n$ set *MAC = MAC* **Xor** $\mathbf{M}_i$ and encrypt *MAC* with *Key$*.
5. If Authentication Mode is **MacModeEMAC**, encrypt *MAC* with the second half of *Key$*.

In steps 3 and 4, if Authentication Mode is **MacModeRetailMAC**, the encryption algorithm is Single DES for all but the last block encryption; the last block is encrypted using the Block Cipher specified in *Algorithm%*, which must be **CryptoAlg2TDES** or **CryptoAlg3TDES**.

If Authentication Mode is **MacModeEMAC** (Encrypted MAC), then *Key$* is a concatenation of two keys: *Key$* = $K_1 + K_2$. Key K_1 is used for steps 3 and 4, and key K_2 is used for step 5. Due to space limitations in the BasicCard, this concatenated key can be at most 32 bytes long, which means that algorithms **CryptoAlg3TDES**, **CryptoAlgAES192**, and **CryptoAlgAES256** are not allowed in this mode.

7.6.3 Message Authentication Code Procedures

The following procedures are provided:

Sub CryptoMAC (ByVal *Algorithm%*, **ReadOnly** *Key$*, **ReadOnly** *IV$*, **ReadOnly** *Data$*,
MAC **As String)**

Compute the MAC of *Data$*, using *Algorithm%*, *Key$*, and *IV$*. The result is returned in *MAC*.

Sub CryptoMACStart (ByVal *Algorithm%*, **ReadOnly** *Key$*, **ReadOnly** *IV$***)**
Sub CryptoMACUpdate (ReadOnly *Data$***)**
Sub CryptoMACEnd (*MAC* **As String)**

Compute the MAC of an arbitrary concatenation of strings. Call **CryptoMACStart** once, then **CryptoMACUpdate** for each string in the sequence, then **CryptoMACEnd** to obtain the MAC.

The following error codes are returned in global variable **LibError**:

CryptoBadAlgorithm	*Algorithm%* is invalid for Message Authentication
CryptoKeyTooShort	*Key$* is too short for the selected Block Cipher Algorithm
CryptoInvalidIV	The Initial Value Mode is not **CryptoIVNone**, but *IV$* is shorter than the block length
CryptoBadCMACParams	If Authentication Mode is **MacModeCMAC**, then the Block Cipher Algorithm must be one of the AES variants; the Initial Value Mode may not be **CryptoIVPlain**; and **CryptoNoPadding** may not be set
CryptoMacState	**CryptoMACUpdate** or **CryptoMACEnd** was called without firs calling **CryptoMACStart**
CryptoKeyTooLong	The Block Cipher Algorithm selected for Authentication Mode **MacModeEMAC** requires a key longer than 16 bytes (so *Key$* would have to be longer than 32 bytes)

7.6.4 Message Encryption and Decryption

These procedures encrypt or decrypt the contents of a **String**. The *Algorithm%* parameter is a combination of the following four options:

- a Block Cipher Algorithm in bits 6-1 (as specified in the introduction to this library)
- an Encryption Mode in bits 10-9:

EncModeCBC	**&H0100**	Cipher Block Chaining mode
EncModeCFB	**&H0200**	Cipher Feedback mode
EncModeOFB	**&H0300**	Output Feedback mode

- an Initial Value Mode in bits 14-13:

CryptoIVPlain	**&H1000**
CryptoIVEncrypted	**&H2000**
CryptoIVNone	**&H3000**

- a Padding Suppression indicator in bit 16:
  CryptoNoPadding **&H8000**

To encrypt a message (here **P** denotes Plaintext and **C** denotes Ciphertext):

1. The *Data$* string is padded to a multiple of the block length of the Block Cipher Algorithm (8 or 16 bytes – see the table above): the padding starts with a byte equal to **&H80** (unless the **CryptoNoPadding** bit is set in *Algorithm%*); then zero or more **&H00** bytes are appended.
2. *Data$* is split into blocks of 8 or 16 bytes: *Data$* = $\mathbf{P}_1 + \ldots + \mathbf{P}_n$
3. Set $\mathbf{C}_0$ = **0...0** if the Initial Value Mode is **CryptoIVNone**; otherwise set $\mathbf{C}_0$ = *IV$*, and encrypt with *Key$* if the Initial Value Mode is **CryptoIVEncrypted**.
4. The blocks are processed sequentially, according to the Encryption Mode:

   EncModeCBC: For $1 \le i \le n$ $\mathbf{C}_i = \mathbf{E}(\mathbf{C}_{i-1} \textbf{ Xor } \mathbf{P}_i)$

   EncModeCFB: For $1 \le i \le n$ $\mathbf{C}_i = \mathbf{E}(\mathbf{C}_{i-1}) \textbf{ Xor } \mathbf{P}_i$

   EncModeOFB: For $1 \le i \le n$ $\mathbf{C}_i = \mathbf{E}(\mathbf{C}_{i-1} \textbf{ Xor } \mathbf{P}_{i-1}) \textbf{ Xor } \mathbf{P}_i$

The encrypted message is $\mathbf{C}_1 + \ldots + \mathbf{C}_n$. Decryption is the reverse of this process.

7.6.5 Encryption and Decryption Procedures

Two procedures are provided:

Sub CryptoEncrypt (ByVal *Algorithm%*, **ReadOnly** *Key$*, **ReadOnly** *IV$*, *Data$***)**

Sub CryptoDecrypt (ByVal *Algorithm%*, **ReadOnly** *Key$*, **ReadOnly** *IV$*, *Data$***)**

These procedures encrypt or decrypt the *Data$* string.

The following error codes may be returned by either procedure, in global variable **LibError**:

CryptoBadAlgorithm	*Algorithm%* is invalid for encryption and decryption
CryptoKeyTooShort	*Key$* is too short for the selected Block Cipher Algorithm
CryptoInvalidIV	The Initial Value Mode is not **CryptoIVNone**, but *IV$* is shorter than the block length

The following error codes may be returned by **CryptoDecrypt**:

CryptoPartialBlock	*Data$* is not a whole number of blocks
CryptoBadPadding	The padding bytes in the decrypted message are invalid

7.6.6 Secure Messaging Overview

Secure Messaging is the encryption and decryption of commands and responses between a Terminal and a smart card. There is an ISO standard for Secure Messaging: **ISO/IEC 7816-4:2005 – Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange**, available at http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=36134 . This standard cannot be implemented as it stands, because it is too general. But there are various real-world implementations of ISO-style Secure Messaging, and we have tried to provide a framework that will make it easy for you to implement any of these.

Briefly, Secure Messaging works as follows (read **8.6 Commands and Responses** first if you are not familiar with **APDU** structures). We start with a Command (say) of the form:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**Le**

Secure Messaging transforms the **IDATA** field into a sequence of Secure Messaging Fields, each of which is a **Tag-Length-Value** object:

Tag (an identifying constant)	**Length** of Value field	**Value** field

Here **Tag** is encoded in 1-2 bytes, and **Length** is encoded in 1-3 bytes – see **7.12 The TLVLib ASN.1 Library** for details.

The **Tag** field is just a number as far as the Crypto System Library is concerned, but the ISO standard recommends various tags for use in Secure Messaging. Among them are the following:

&H80, **&H81**	Plain unencrypted data
&H82, **&H83**	Encrypted data
&H86, **&H87**	Padding Indicator byte followed by encrypted data
&H8E	**MAC** (Message Authentication Code)
&H96, **&H97**	Original **Le**
&H99	**SW1-SW2**

The ISO Standard recommends that every Field with an odd-numbered Tag be included in the **MAC** calculation. The Crypto System Library does not do this automatically; you must configure it by including or omitting the **SMItemSkipMAC** flag in the *ItemHeader* (see below).

A typical Secure Messaging implementation might encrypt this Command as:

CLA'	INS	P1	P2	Lc'	TLV(IDATA')	TLV(Le)	TLV(MAC)	Le'

CLA'	**CLA** with bits 3 and 4 (**&H0C**) set, to indicate Secure Messaging with the Command Header (**CLA' INS P1 P2**) included in the **MAC** calculation
Lc'	The combined length of the three **TLV** fields
TLV(IDATA')	**&H83** *Length* **IDATA'**, where **IDATA'** is the **2TDES CBC** encryption of **IDATA**
TLV(Le)	**&H97 &H01 Le**
TLV(MAC)	**&H8E &H08 MAC**, where **MAC** is a **2TDES CBC-MAC**
Le'	Expected Response length (set to **0** by the Crypto library if **TLV(Le)** is present)

The **MAC** is calculated over the following data:

- **CLA' INS P1 P2** if the **SMIncludeHeaderInMac** option is set in the first element of **SMSpec()**
- every Secure Messaging Field of Item Type **SMItemData** or **SMItemTrailer** that does not have **SMItemSkipMAC** set in its *ItemHeader*

If any Fields are skipped, then padding is incorporated in the **MAC** calculation, so that the next Field starts on a block boundary. For this purpose, **Lc** counts as a Field; so if the **SMIncludeHeaderInMac** option is set, then padding is always incorporated after **CLA' INS P1 P2**.

7.6.7 Secure Messaging Specification

Secure Messaging is specified by means of a Secure Messaging Specification, encoded as an **Integer** array. For convenience, this array is denoted by **SMSpec()** in what follows, although it can be any user-declared **Integer** array. This array encodes the contents of Commands and Responses as a sequence of **Tag-Length-Value** Secure Messaging Fields.

The first element of the **SMSpec()** array is a bit map of Secure Messaging options (these and later constants are defined in **Crypto.def**):

SMIncludeHeaderInMac	**&H01**	**CLA INS P1 P2** is included in **MAC**
SMMacChained	**&H10**	**MAC IV** is the previous **MAC**
SMMacPreIncSSC	**&H20**	**MAC IV** is a pre-incremented **SSC** (Send Sequence Counter)
SMEncPreIncSSC	**&H40**	**ENC IV** is a pre-incremented **SSC**
SMCommonPreIncSSC	**&H80**	**MAC** and **ENC** use the same pre-incremented **SSC**

Note: See **7.6.8 Secure Messaging Procedures** for an explanation of the **PreIncSSC** options.

The rest of the **SMSpec()** array is filled with Secure Messaging Items. Each Secure Messaging Item occupies one or more array elements, and specifies the contents of a single Secure Messaging Field. The first array element in a Secure Messaging Item is the *ItemHeader*. It contains the **Tag** in bits 8-1, and the Item Type in bits 10-9:

SMItemData	**&H0100**	Plain or Encrypted data
SMItemMAC	**&H0200**	**MAC**
SMItemTrailer	**&H0300**	**Le** for commands; **SW1-SW2** for responses

An *ItemHeader* equal to zero is simply ignored. Otherwise, if **Tag** is zero, the required Tag follows in the next array element. This allows for 2-byte tags.

More than one **SMItemData** item is allowed. Successive **SMItemData** items contain successive blocks from the Source String, whose lengths can be individually specified – see **SMItemLength** below. (The Source String is the **IDATA** field of a Command, or the **ODATA** field of a response.)

The *ItemHeader* can also contain the following flags:

SMItemCommandOnly &H0400
The field is only included in Commands, not Responses

SMItemResponseOnly &H0800
The field is only included in Responses, not Commands

SMItemLength &H1000
Secure Messaging Item contains a *Length* member:

- If the Item Type is **SMItemMAC**, then *Length* is the number of bytes of the **MAC** to include in the Command or Response; if absent, the whole of the **MAC** is used.
- If the Item Type is **SMItemData**, then *Length* is the number of bytes to include from the source string. If absent, the whole of the source string is used; if *Length* is negative, all but the last *–Length* bytes of the source string are included. An error is signalled if the source string contains less than *Length* bytes (unless **SMConditionLengthIsMax** is set – see below).

Note: Most implementations of ISO-style Secure Messaging only require one **SMItemData** item, which contains the whole of the Source String. In this case, just omit the **SMItemLength** flag.

SMItemPI &H2000
This flag is for Item Type **SMItemData**. If the flag is set, then the Secure Messaging Item contains a *PaddingIndicator* byte, which is added to the start of the data.

SMItemSkipMAC &H4000
If this flag is set, then the Secure Messaging Field is not included in the **MAC** computation.

SMItemConditional &H8000
If this flag is set, then the Secure Messaging Item contains a *Condition* element, which can contain the following flags:

- **SMConditionNonEmpty &H0001**
  The Secure Messaging Field is only included if it is non-empty
- **SMConditionLengthIsMin &H0002**
  Used in conjunction with the **SMItemLength** flag; the whole of the Source String will be included, with an error signalled if it contains less than *Length* bytes.
- **SMConditionLengthIsMax &H0004**
  Used in conjunction with the **SMItemLength** flag; up to *Length* bytes from the Source String will be included.
- **SMConditionEmptyResponse &H0008**
  The Secure Messaging Field is included only if the Response **ODATA** field is empty. (This flag is ignored for Commands.)
- **SMConditionNonEmptyResponse &H0010**
  The Secure Messaging Field is included only if the Response **ODATA** field is non-empty. (This flag is ignored for Commands.)

A Secure Messaging Item contains some or all of the following elements, in the order given:

ItemHeader	Compulsory
Tag	Present if **Tag** in *ItemHeader* is zero
Condition	Present if the **SMItemConditional** flag is set in *ItemHeader*
Algorithm	Compulsory for **SMItemData** and **SMItemMAC**. If zero (valid for **SMItemData** only), the field is not encrypted.
Length	Present if the **SMItemLength** flag is set in *ItemHeader*
PaddingIndicator	Present if the **SMItemPI** flag is set in *ItemHeader*

7.6.8 Secure Messaging Procedures

In the Terminal program, Secure Messaging is used to encrypt Commands and decrypt Responses; in the BasicCard, it is used to decrypt Commands and encrypt Responses. So two sets of procedures are provided. Secure Messaging can be used in two modes: User-controlled, and Automatic. In User-controlled Secure Messaging, the user encrypts and decrypts each Command and Response by calling the appropriate Crypto System Library procedure; in Automatic Secure Messaging, the user enables Secure Messaging once, and the Operating System encrypts and decrypts all Commands and Responses automatically.

Cryptographic Parameters

Secure Messaging requires cryptographic keys and initialisation vectors. These are passed as parameters *MacKey$*, *MacIV$*, *EncKeys$*, and *EncIV$*. Two questions arise: firstly, where do these parameters come from? Secondly, how are they used?

The source of these parameters is irrelevant to the Crypto System Library – it is enough that they be the same in the Terminal and the BasicCard. But in a real-life implementation, they are typically generated by means of a key-agreement procedure between Terminal and card, involving digitally-signed certificates. An extensive example of this is provided in the directory `Examples\SM`.

Parameters *MacKey$* and *EncKey$* are the keys for the Block Cipher Algorithms specified in the **SMSpec()** array.

Parameters *MacIV$* and *EncIV$* are used to construct the Initialisation Vectors for the Message Authentication and Encryption algorithms, if the Initial Value Mode for the algorithm is not set to **CrtyptoIVNone**. This depends on the Secure Messaging option in the first element of **SMSpec()**:

- If **SMMacChained** is set, then *MacIV$* is used as the Initialisation Vector for the Message Authentication algorithm of the first message; and subsequent messages use the **MAC** of the previous message as Initialisation Vector.
- If **SMMacPreIncSSC** is set, then before each **MAC** calculation, *MacIV$* (considered as a Big-Endian integer) is incremented by 1 and used as the Initialisation Vector for the Message Authentication algorithm. *MacIV$* is called a Send Sequence Counter.
- If **SMEncPreIncSSC** is set, then before each encryption or decryption, *EncIV$* (considered as a Big-Endian integer) is incremented by 1 and used as the Initialisation Vector for the Encryption/Decryption algorithm.
- If **SMCommonPreIncSSC** is set, the same vector (*MacIV$* if non-empty, otherwise *EncIV$*) is used as the Send Sequence Counter for both Message Authentication and Encryption.

7.6.9 Terminal Program Procedures

User-Controlled Secure Messaging

Sub CryptoSMEncryptCommand (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,**_**
ReadOnly *EncKeys$*, *EncIV$*, *CLA@*, **ByVal** *INS@*, **ByVal** *P1P2%*, *IDATA$*, *Le%***)**

Encrypt a Command before sending it to the card, using Secure Messaging specification *SMSpec*(). This procedure may change the *CLA@* byte.

Sub CryptoSMDecryptResponse (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,_
ReadOnly *EncKeys$*, *EncIV$*, *ODATA$*, *SW1SW2%***)**

Decrypt a Response received from the card, using Secure Messaging specification *SMSpec*().

Automatic Secure Messaging:

Sub CryptoSMEnable (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,_
ReadOnly *EncKeys$*, *EncIV$***)**

Enable Automatic Secure Messaging using Secure Messaging specification *SMSpec*().

Sub CryptoSMConfigure (ReadOnly *SMSpec***())**

Change the Secure Messaging specification *SMSpec*() while Automatic Secure Messaging is enabled. This lets you use a different specification depending on the Command.

Sub CryptoSMDisable()

Disable Automatic Secure Messaging.

7.6.10 BasicCard Procedures

User-Controlled Secure Messaging

Function CryptoSMDecryptCommand (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,_
ReadOnly *EncKeys$*, *EncIV$*, *CLA@*, **ByVal** *INS@*, **ByVal** *P1P2%*, *IDATA$*, *Le%***)**

Decrypt a Command received from the Terminal, using Secure Messaging specification *SMSpec*(). This procedure may change the *CLA@* byte.

Sub CryptoSMEncryptResponse (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,_
ReadOnly *EncKeys$*, *EncIV$*, *ODATA$*, *SW1SW2%***)**

Encrypt a Response sent to the Terminal, using Secure Messaging specification *SMSpec*().

Automatic Secure Messaging

Sub CryptoSMEnable (ReadOnly *SMSpec***(), ReadOnly** *MacKey$*, *MacIV$*,_
ReadOnly *EncKeys$*, *EncIV$*, *Immediate%***)**

Enable Automatic Secure Messaging using Secure Messaging specification *SMSpec*(). The Response to the current Command is sent without Secure Messaging unless *Immediate%* is **True**.

Sub CryptoSMConfigure (ReadOnly *SMSpec***())**

Change the Secure Messaging specification *SMSpec*() while Automatic Secure Messaging is enabled. This lets you use a different specification depending on the Command.

Sub CryptoSMDisable (*Immediate%***)**

Disable Automatic Secure Messaging. The Response to the current Command is sent with Secure Messaging unless *Immediate%* is **True**.

Function CryptoSMStatus()

Return the Secure Messaging status, which is a combination of the following flags:

SMStatusActive	**&H01**	Secure Messaging is currently enabled
SMStatusHeaderInMAC	**&H02**	Command header **CLA INS P1 P2** was incorporated in the **MAC**
SMStatusCryptogram	**&H04**	The command or response contained an encrypted data field
SMStatusPlaintext	**&H08**	The command or response contained an unencrypted data field
SMStatusTrailer	**&H10**	The trailer (**Le** or **SW1-SW2**) was incorporated in the **MAC**
SMStatusMAC	**&H20**	The command or response contained a **MAC** field

7.6.11 Secure Messaging Examples

In the next three sections, we present three examples of Secure Messaging, and show how to configure them using the Crypto System Library. Two of the examples are taken from real life.

First, we show how to implement the Secure Messaging example from **7.6.6 Secure Messaging Overview**. The **SMSpec()** array looks like this:

Public SMSpec() =_
SMIncludeHeaderInMac + SMMacChained,_ ' Secure Messaging options
_ ' Encrypted **IDATA** field:
SMItemData + SMItemConditional + &H83,_ ' *ItemHeader* with **Tag=&H83**
SMConditionNonEmpty,_ ' *Condition*
CryptoAlg2TDES + EncModeCBC + CryptoIVNone,_ ' *Algorithm*
_ ' **Le** field (Commands only):
SMItemTrailer + SMItemCommandOnly + &H97,_' *ItemHeader* with **Tag=&H97**
_ ' **SW1-SW2** field (Responses only):
SMItemTrailer + SMItemResponseOnly + &H99,_ ' *ItemHeader* with **Tag=&H99**
_ ' **MAC** field:
SMItemMAC + &H8E,_ ' *ItemHeader* with **Tag=&H8E**
CryptoAlg2TDES + MacModeCBCMAC + CryptoIVPlain ' *Algorithm*

Before Secure Messaging is enabled, the Terminal program and the BasicCard have to agree on the session key data **MacKey$** (16 bytes), **EncKey$** (16 bytes), and **MacIV$** (8 bytes). For an example of this, see `BasicCardV8\Examples\SM`, which implements session key agreement using mutual authentication with RSA-based certificates. (This example is available from the **Help | Examples ▸ SM** menu in the **BCDevEnv** program.) Then the Terminal program enables Secure Messaging with:

Call CryptoSMEnable (SMSpec, MacKey$, MacIV$, EncKey$, "")

and the BasicCard enables Secure Messaging with:

Call CryptoSMEnable (SMSpec, MacKey$, MacIV$, EncKey$, "", False)

After this, all subsequent commands and responses are automatically sent with Secure Messaging, until a Secure Messaging error occurs or the Secure Messaging session is explicitly ended using **CryptoSMDisable**.

7.6.12 Secure Messaging Example from CWA 14890

CWA 14890 is the **CEN** Workshop Agreement **Application Interface for smart cards used as Secure Signature Creation Devices**, developed to serve as a common European legal framework for electronic signatures created by smart cards. Guidance on how to obtain this document can be found at this link. **CWA 14890** specifies a Secure Messaging protocol that requires all command/response pairs to be authenticated; in addition, some command/response pairs are encrypted. The encryption algorithm can be **CryptoAlg2TDES** or **CryptoAlgAES128**. This example uses **CryptoAlgAES128**, which has a 16-byte block size, so the command header **CLA' INS P1 P2** is padded with 12 bytes **80 00 00 00 00 00 00 00 00 00 00 00** for the **MAC** calculation.

Unencrypted commands:

If no encryption is required, a typical unsecured command looks like this:

CLA	INS	P1	P2	Lc	IDATA	Le

After Secure Messaging is applied:

CLA'	INS	P1	P2	Lc'	TLV(IDATA)	TLV(Le)	TLV(MAC)	Le

CLA' **CLA** with bits 3 and 4 (**&H0C**) set, to indicate Secure Messaging with the Command Header (**CLA' INS P1 P2**) included in the **MAC** calculation

Lc' The combined length of the two **TLV** fields

TLV(IDATA) **&H81** *Length* **IDATA**

TLV(Le) **&H97 &H01 Le**

TLV(MAC) **&H8E &H08 MAC**, where **MAC** is an **EMAC** with encrypted **IV**

The **MAC** is calculated over **CLA' INS P1 P2 80 00...00 TLV(IDATA) TLV(Le)**.

If the command is unencrypted, then so is the response:

ODATA	SW1	SW2

becomes

TLV(ODATA)	TLV(SW1 SW2)	TLV(MAC)	SW1	SW2

TLV(ODATA)	**&H81** *Length* **ODATA**
TLV(SW1 SW2)	**&H99 &H02 SW1 SW2**
TLV(MAC)	**&H8E &H08 MAC**, where **MAC** is an **EMAC** with encrypted **IV**

The **MAC** is calculated over **TLV(ODATA) TLV(SW1 SW2)**.

Encrypted commands:

If encryption is required, the unsecured command becomes:

CLA'	INS	P1	P2	Lc'	TLV(IDATA')	TLV(Le)	TLV(MAC)	Le

CLA'	**CLA** with bits 3 and 4 (**&H0C**) set, to indicate Secure Messaging with the Command Header (**CLA' INS P1 P2**) included in the **MAC** calculation
Lc'	The combined length of the two **TLV** fields
TLV(IDATA')	**&H87** *Length* **IDATA'**, where **IDATA'** is the **AES128 CBC** encryption of **IDATA**
TLV(Le)	**&H97 &H01 Le**
TLV(MAC)	**&H8E &H08 MAC**, where **MAC** is an **EMAC** with encrypted **IV**

The **MAC** is calculated over **CLA' INS P1 P2 80 00...00 TLV(IDATA') TLV(Le)**.

If the command is encrypted, then so is the response:

ODATA	SW1	SW2

becomes

TLV(ODATA')	TLV(SW1 SW2)	TLV(MAC)	SW1	SW2

TLV(ODATA')	**&H87** *Length* **ODATA'**, where **ODATA'** is the **AES128 CBC** encryption of **ODATA**
TLV(SW1 SW2)	**&H99 &H02 SW1 SW2**
TLV(MAC)	**&H8E &H08 MAC**, where **MAC** is an **EMAC** with encrypted **IV**

The **MAC** is calculated over **TLV(ODATA') TLV(SW1 SW2)**.

Secure Messaging specification in the Terminal program:

The Terminal program needs two **SMSpec()** arrays. For unencrypted commands:

```
Public SMSpecPlain() =_
    SMIncludeHeaderInMac + SMMacPreIncSSC,_      ' Secure Messaging options
    _ ' Unencrypted IDATA field:
    SMItemData + SMItemConditional + &H81,_      ' ItemHeader with Tag=&H81
        SMConditionNonEmpty,_                    ' Condition
        0,_                                      ' Algorithm (0 means don't encrypt)
    _ ' Le field (Commands only):
    SMItemTrailer + SMItemCommandOnly + &H97,_' ItemHeader with Tag=&H97
    _ ' SW1-SW2 field (Responses only):
    SMItemTrailer + SMItemResponseOnly + &H99,_  ' ItemHeader with Tag=&H99
    _ ' MAC field:
    SMItemMAC + &H8E,_                           ' ItemHeader with Tag=&H8E
        CryptoIVEncrypted + CryptoAlgAES128 + MacModeEMAC    ' Algorithm
```

For encrypted commands:

```
Public SMSpecEncrypted() =_
    SMIncludeHeaderInMac + SMMacPreIncSSC,_      ' Secure Messaging options
    _ ' Encrypted IDATA field:
    SMItemData + SMItemConditional + &H87,_      ' ItemHeader with Tag=&H87
        SMConditionNonEmpty,_                    ' Condition
        CryptoIVNone + CryptoAlgAES128 + EncModeCBC,_        ' Algorithm
    _ ' Le field (Commands only):
    SMItemTrailer + SMItemCommandOnly + &H97,_' ItemHeader with Tag=&H97
    _ ' SW1-SW2 field (Responses only):
    SMItemTrailer + SMItemResponseOnly + &H99,_  ' ItemHeader with Tag=&H99
    _ ' MAC field:
    SMItemMAC + &H8E,_                           ' ItemHeader with Tag=&H8E
        CryptoIVEncrypted + CryptoAlgAES128 + MacModeEMAC ' Algorithm
```

Secure Messaging specification in the BasicCard:

The BasicCard needs two **SMSpec()** arrays, for sending unencrypted and encrypted responses. But the BasicCard can't know in advance the type of the next command, so each of the **SMSpec()** arrays has to accept commands of either type. We do this by making all **IDATA** fields conditional on being non-empty (i.e. optional), and specifying the unwanted **ODATA** field as **SMItemCommandOnly**.

For unencrypted responses:

```
Public SMSpecPlain() =_
    SMIncludeHeaderInMac + SMMacPreIncSSC,_      ' Secure Messaging options
    _ ' Unencrypted IDATA field:
    SMItemData + SMItemConditional + &H81,_      ' ItemHeader with Tag=&H81
        SMConditionNonEmpty,_                    ' Condition
        0,_                                      ' Algorithm (0 means don't encrypt)
    _ ' Encrypted IDATA field:
    SMItemData + SMItemConditional + SMItemCommandOnly + &H87,_
        _                                        ' ItemHeader with Tag=&H87
        SMConditionNonEmpty,_                    ' Condition
        CryptoIVNone + CryptoAlgAES128 + EncModeCBC,_ ' Algorithm
    _ ' Le field (Commands only):
    SMItemTrailer + SMItemCommandOnly + &H97,_' ItemHeader with Tag=&H97
    _ ' SW1-SW2 field (Responses only):
    SMItemTrailer + SMItemResponseOnly + &H99,_  ' ItemHeader with Tag=&H99
    _ ' MAC field:
    SMItemMAC + &H8E,_                           ' ItemHeader with Tag=&H8E
        CryptoIVEncrypted + CryptoAlgAES128 + MacModeEMAC ' Algorithm
```

For encrypted responses:

```
Public SMSpecEncrypted() =_
        SMIncludeHeaderInMac + SMMacPreIncSSC,_       ' Secure Messaging options
        _ ' Unencrypted IDATA field:
        SMItemData + SMItemConditional + SMItemCommandOnly + &H81,_
                _                                     ' ItemHeader with Tag=&H81
                SMConditionNonEmpty,_                 ' Condition
                0,_                                   ' Algorithm (0 means don't encrypt)
        _ ' Encrypted IDATA field:
        SMItemData + SMItemConditional + &H87,_       ' ItemHeader with Tag=&H87
                SMConditionNonEmpty,_                 ' Condition
                CryptoIVNone + CryptoAlgAES128 + EncModeCBC,_         ' Algorithm
        _ ' Le field (Commands only):
        SMItemTrailer + SMItemCommandOnly + &H97,_' ItemHeader with Tag=&H97
        _ ' SW1-SW2 field (Responses only):
        SMItemTrailer + SMItemResponseOnly + &H99,_  ' ItemHeader with Tag=&H99
        _ ' MAC field:
        SMItemMAC + &H8E,_                            ' ItemHeader with Tag=&H8E
                CryptoIVEncrypted + CryptoAlgAES128 + MacModeEMAC ' Algorithm
```

To enable Secure Messaging:

First the Terminal program and the BasicCard must agree on the session key data **MacKey$** (32 bytes), **EncKey$** (16 bytes), and **MacIV$** (8 bytes) (the **EMAC** algorithm requires two keys, hence 32 bytes). Then the Terminal program enables Secure Messaging with:

Call CryptoSMEnable (SMSpecPlain, MacKey$, MacIV$, EncKey$, "")

and the BasicCard enables Secure Messaging with:

Call CryptoSMEnable (SMSpecPlain, MacKey$, MacIV$, EncKey$, "", False)

(You could just as well use **SMSpecEncrypted** as the first parameter here.) Then before each command, the Terminal program calls **CryptoSMConfigure** with the appropriate parameter:

Call CryptoSMConfigure (*SMSpec***)**

where *SMSpec* is **SMSpecPlain** or **SMSpecEncrypted** according to the command type.

After receiving a command, the BasicCard calls **CryptoSMStatus()** to find out which of the two forms was transmitted, and sets the appropriate **SMSpec()** array for the response:

```
If (CryptoSMStatus() And SMStatusCryptogram) <> 0 Then
   Call CryptoSMConfigure (SMSpecEncrypted)   ' Encrypted form
Else
   Call CryptoSMConfigure (SMSpecPlain)       ' Unencrypted form
End If
```

7.6.13 Secure Messaging Example from VDV Card

The VDV card is a specification for a travel card, produced by a consortium of German public transport providers, the *Verband Deutscher Verkehrsunternehmen* (see their web site at www.vdv.de). Secure Messaging is described in the document **VDV-Kernapplikation Spezifikation Nutzermedium**, available (to subscribers only) at this link.

The whole of the command is authenticated; and depending on the command, either the last 16 bytes of **IDATA** are encrypted, or no encryption is used at all. Responses to commands sent with Secure Messaging consist of the status bytes **SW1-SW2**, with no Secure Messaging fields.

Unencrypted commands:

If no encryption is required, the unsecured command looks like this:

CLA	INS	P1	P2	Lc	IDATA

After Secure Messaging is applied:

CLA'	INS	P1	P2	Lc'	TLV(IDATA)	TLV(MAC)	Le

CLA' — **CLA** with bits 3 and 4 (**&H0C**) set, to indicate Secure Messaging with the Command Header (**CLA' INS P1 P2**) included in the **MAC** calculation

Lc' — The combined length of the two **TLV** fields

TLV(IDATA) — **&H81** *Length* **IDATA**

TLV(MAC) — **&H8E &H08 MAC**, where **MAC** is a **2TDES Retail MAC** with unencrypted **IV**

Le — **0**

The **MAC** is calculated over **CLA' INS P1 P2 80 00 00 00 TLV(IDATA)**.

Encrypted commands:

If encryption is required, the unsecured command looks like this:

CLA	INS	P1	P2	Lc	LDATA	RDATA

where **RDATA** is the 16-byte field that is to be encrypted. After Secure Messaging is applied:

CLA'	INS	P1	P2	Lc'	TLV(LDATA)	TLV(RDATA')	TLV(MAC)	Le

CLA' — **CLA** with bits 3 and 4 (**&H0C**) set, to indicate Secure Messaging with the Command Header (**CLA' INS P1 P2**) included in the **MAC** calculation

Lc' — The combined length of the three **TLV** fields

TLV(LDATA) — **&H81** *Length* **LDATA**

TLV(RDATA') — **&H87 &H11 &H00 RDATA'**, where **RDATA'** is the **2TDES CBC** encryption of **RDATA** (**&H00** is the Padding Indicator)

TLV(MAC) — **&H8E &H08 MAC**, where **MAC** is a **2TDES Retail MAC** with unencrypted **IV**

Le — **0**

The **MAC** is calculated over **CLA' INS P1 P2 80 00 00 00 TLV(LDATA) TLV(RDATA')**.

Secure Messaging specification in the Terminal program:

The Terminal program needs two **SMSpec()** arrays. For unencrypted commands:

```
Public SMSpecPlain() =_
    SMMacPreIncSSC + SMIncludeHeaderInMAC,_  ' Secure Messaging options
    _ ' Unencrypted IDATA field:
    SMItemData + SMItemCommandOnly + &H81,_  ' ItemHeader with Tag=&H81
        0,_                                  ' Algorithm (0 means don't encrypt)
    _ ' MAC field:
    SMItemMac + SMItemCommandOnly + &H8E,_  ' ItemHeader with Tag=&H8E
        CryptoAlg2TDES + MacModeRetailMAC + CryptoIVPlain  ' Algorithm
```

For encrypted commands:

```
Public SMSpecEncrypted() =_
        SMMacPreIncSSC + SMIncludeHeaderInMAC,_  ' Secure Messaging options
        _ ' Unencrypted IDATA field:
        SMItemData + SMItemLength + SMItemCommandOnly + &H81,_
        _                                         ' ItemHeader with Tag=&H81
                0,_                               ' Algorithm (0 means don't encrypt)
                –16,_                             ' Length = Lc - 16
        _ ' Encrypted IDATA field:
        SMItemData + SMItemPI + SMItemCommandOnly + &H87,_
        _                                         ' ItemHeader with Tag=&H87
                CryptoAlg2TDES + EncModeCBC + CryptoIVNone + CryptoNoPadding,_
                _                                 ' Algorithm
                0,_                               ' PaddingIndicator
        _ ' MAC field:
        SMItemMac + SMItemCommandOnly + &H8E,_   ' ItemHeader with Tag=&H8E
                CryptoAlg2TDES + MacModeRetailMAC + CryptoIVPlain  ' Algorithm
```

Secure Messaging specification in the BasicCard:

The BasicCard only needs one **SMSpec()** array:

```
Eeprom SMSpec() =_
        SMMacPreIncSSC + SMIncludeHeaderInMAC,_  ' Secure Messaging options
        _ ' Unencrypted IDATA field:
        SMItemData + SMItemCommandOnly + &H81,_  ' ItemHeader with Tag=&H81
                0,_                               ' Algorithm (0 means don't encrypt)
        _ ' Encrypted IDATA field:
        SMItemData + SMItemLength + SMItemConditional + SMItemPI +_
                SMItemCommandOnly + &H87,_        ' ItemHeader with Tag=&H87
                SMConditionNonEmpty,_             ' Condition
                CryptoAlg2TDES + EncModeCBC + CryptoIVNone + CryptoNoPadding,_
                _                                 ' Algorithm
                16,_                              ' Length
                0,_                               ' PaddingIndicator
        _ ' MAC field:
        SMItemMac + SMItemCommandOnly + &H8E,_   ' ItemHeader with Tag=&H8E
                CryptoAlg2TDES + MacModeRetailMAC + CryptoIVPlain  ' Algorithm
```

To enable Secure Messaging:

After the Terminal program and the BasicCard agree on the session key data **MacKey$** (16 bytes), **EncKey$** (16 bytes), and **MacIV$** (8 bytes), the Terminal program enables Secure Messaging with:

```
Call CryptoSMEnable (SMSpecPlain, MacKey$, MacIV$, EncKey$, "")
```

and the BasicCard enables Secure Messaging with:

```
Call CryptoSMEnable (SMSpec, MacKey$, MacIV$, EncKey$, "", False)
```

(The Terminal program could just as well use **SMSpecEncrypted** as the first parameter here.) Then before each command, the Terminal program calls **CryptoSMConfigure** with the appropriate parameter:

Call CryptoSMConfigure (*SMSpec***)**

where *SMSpec* is **SMSpecPlain** or **SMSpecEncrypted** according to the command type.

The BasicCard handles both forms, due to the **SMConditionNonEmpty** field in the encrypted item specification. And it can call **CryptoSMStatus()** to find out which of the two forms was transmitted:

```
If (CryptoSMStatus() And SMStatusCryptogram) <> 0 Then... ' Encrypted form
```

7.6.14 Customer-Specific Key Procedures

Two procedures are provided for using a Customer-Specific Encryption Key.

In a Terminal program:

Sub CryptoSetCustomerKey (ReadOnly *Key$***)**

Key$ is the 32-byte Customer-Specific Key provided by ZeitControl.

In a BasicCard program:

Sub CryptoSetKDP (ReadOnly *KDP$***)**

KDP$ is the 24-byte Key Derivation Parameter provided by ZeitControl.

See **9.10 Customer-Specific Encryption Keys** for more information.

7.7 The BigInt Library

The **BigInt** System Library implements arithmetic operations on big integers, which are represented as **String** variables. It is available in the **ZC7**-series BasicCard, from **REV C**, and in Terminal programs. To use the **BigInt** System Library:

#Include BigInt.def

Integers are represented in Big-Endian format, most significant byte first. The empty string represents the integer 0. Negative integers are not allowed (but see **Function BigIntSub** below). The maximum size of an integer depends only on the maximum allowed string length, which is 2048 bytes (16384 bits) in the **ZC7-**series BasicCard, and 16384 bytes (131072 bits) in a Terminal program. Thus an integer can be as large as $2^{16384} - 1$ in a BasicCard program, and $2^{131072} - 1$ in a Terminal program. However, in a BasicCard program, the size of the input parameters to most of the arithmetic operations is further limited by the hardware. Such limitations are described below, under the **BigIntOperandTooBig** error code.

Most procedures in the library have two variants: a **Function** that returns the result as a **String**, and a **Sub** that returns the result by overwriting the first parameter in-place. This is to avoid unnecessary creation of large **String** variables. The **Sub** variant has the same name as the F**unction** variant, with **...InPlace** appended. As a rule, you should use the **Sub** variant if the first parameter will not be needed again.

The **BigInt** System Library provides the following arithmetic operations, each of which is explained in more detail below:

BigIntCompare	Compare *x* with *y*
BigIntAdd	$x + y$
BigIntSub	$x - y$
BigIntMul	$x * y$
BigIntDiv	x / y
BigIntRem	*x* **Mod** *y*
BigIntDivRemInPlace	Calculate *x* / *y* and *x* **Mod** *y* simultaneously
BigIntShiftLeft	*x* **Shl** *Shift*
BigIntShiftRight	*x* **Shr** *Shift*
BigIntAnd	*x* **And** *y*
BigIntOr	*x* **Or** *y*
BigIntXor	*x* **Xor** *y*
BigIntPower	x^e **Mod** *n*
BigIntHCF	Highest Common Factor of *x* and *y*
BigIntInvert	Inverse of *x* modulo *n*
BigIntSquareRoot	Square root of *x* modulo *p*
BigIntJacobiSymbol	Jacobi symbol $\left(\frac{a\$}{m\$}\right)$

These procedures set the **LibError** variable to a non-zero value on error. All the error codes are defined in **BigInt.def**. In the following descriptions, the error codes are those returned by the BasicCard System Library – the Terminal Program System Library sets **LibError** only when the result would be longer than 16384 bytes, or the operation is invalid. If an operation has a **Function** variant and a **Sub** variant, the error codes are described only once:

Function BigIntCompare (ReadOnly *x$*, **ReadOnly** *y$*) **As Integer** Returns –1, 0, or 1 according as *x$* is less than, equal to, or greater than *y$* *Errors* None
Function BigIntAdd (ReadOnly *x$*, **ReadOnly** *y$*) **As String** Returns *x$* + *y$* **Sub BigIntAddInPlace (***x$*, **ReadOnly** *y$***)** Computes *x$* = *x$* + *y$* *Errors* **BigIntOperandTooBig** **Len(***x$***) > &H500** or **Len(***y$***) > &H500**

Function BigIntSub (ReadOnly *x$*, ReadOnly *y$*, *Negative%*) As String
If *x$* ≥ *y$*, returns *x$* – *y$*, with *Negative%* = **False** (0)
If *x$* < *y$*, returns *y$* – *x$*, with *Negative%* = **True** (–1)

Sub BigIntSubInPlace (*x$*, ReadOnly *y$*, *Negative%*)
If *x$* ≥ *y$*, computes *x$* = *x$* – *y$*, with *Negative%* = **False** (0)
If *x$* < *y$*, computes *x$* = *y$* – *x$*, with *Negative%* = **True** (–1)

Errors **BigIntOperandTooBig** **Len(*x$*) > &H500** or **Len(*y$*) > &H500**

Function BigIntMul (ReadOnly *x$*, ReadOnly *y$*) As String
Returns *x$* * *y$*

Sub BigIntMulInPlace (*x$*, ReadOnly *y$*)
Computes *x$* = *x$* * *y$*

Errors **BigIntOperandTooBig** **Len(*x$*) > &H500** or **Len(*y$*) > &H500**
BigIntOverflow **Len(*x$* * *y$*) > &H500**

Function BigIntDiv (ReadOnly *x$*, ReadOnly *y$*) As String
Returns *x$* / *y$*

Sub BigIntDivInPlace (*x$*, ReadOnly *y$*)
Computes *x$* = *x$* / *y$*

Errors **BigIntOperandTooBig** *x$* / *y$* cannot be computed [1]
BigIntDivideByZero *y$* = 0

Function BigIntRem (ReadOnly *x$*, ReadOnly *y$*) As String
Returns *x$* **Mod** *y$*

Sub BigIntRemInPlace (*x$*, ReadOnly *y$*)
Computes *x$* = *x$* **Mod** *y$*

Errors **BigIntOperandTooBig** *x$* / *y$* cannot be computed [1]
BigIntDivideByZero *y$* = 0

Sub BigIntDivRemInPlace (*x$*, *y$*)
Computes *x$* = *x$* / *y$* and *y$* = *x$* **Mod** *y$*

Errors **BigIntOperandTooBig** *x$* / *y$* cannot be computed [1]
BigIntDivideByZero *y$* = 0

Function BigIntShiftLeft (ReadOnly *x$*, ByVal *Shift%*) As String
Returns *x$* **Shl** *Shift%*. Negative values of *Shift%* are not allowed.

Sub BigIntShiftLeftInPlace (*x$*, ByVal *Shift%*)
Computes *x$* = *x$* **Shl** *Shift%*. Negative values of *Shift%* are not allowed.

Errors **BigIntNegativeShift** *Shift%* < 0
BigIntOverflow **Len(*x$* Shl *Shift%*) > &H800**

Function BigIntShiftRight (ReadOnly *x$*, ByVal Shift%) As String
Returns *x$* **Shr** *Shift%*. Negative values of *Shift%* are not allowed.

Sub BigIntShiftRightInPlace (*x$*, ByVal *Shift%*)
Computes *x$* = *x$* **Shr** *Shift%*. Negative values of *Shift%* are not allowed.

Errors **BigIntNegativeShift** *Shift%* < 0

Function BigIntAnd (ReadOnly *x$*, **ReadOnly** *y$***) As String**
Returns bitwise *x$* **And** *y$*

Sub BigIntAndInPlace (*x$*, **ReadOnly** *y$***)**
Computes bitwise *x$* = *x$* **And** *y$*

Function BigIntOr (ReadOnly *x$*, **ReadOnly** *y$***) As String**
Returns bitwise *x$* **Or** *y$*

Sub BigIntOrInPlace (*x$*, **ReadOnly** *y$***)**
Computes bitwise *x$* = *x$* **Or** *y$*

Function BigIntXor (ReadOnly *x$*, **ReadOnly** *y$***) As String**
Returns bitwise *x$* **Xor** *y$*

Sub BigIntXorInPlace (*x$*, **ReadOnly** *y$***)**
Computes bitwise *x$* = *x$* **Xor** *y$*

Errors **BigIntOperandTooBig** **Len(***x$***) > &H500** or **Len(***y$***) > &H500**

Function BigIntPower (ReadOnly *x$*, **ReadOnly** *e$*, **ReadOnly** *n$***) As String**
Returns *x$* $^{e\$}$ **Mod** *n$*

Sub BigIntPowerInPlace (*x$*, **ReadOnly** *e$*, **ReadOnly** *n$***)**
Computes *x* = *x$* $^{e\$}$ **Mod** *n$*

Errors **BigIntOperandTooBig** **Len(***n$***) > &H200** or *x$* ≥ *n$*
BigIntInvalidOperand *n$* < 2^{159}

Function BigIntHCF (ReadOnly *x$*, **ReadOnly** *y$***) As String**
Returns **HCF(***x$*, *y$***)**, the Highest Common Factor of *x$* and *y$*

Sub BigIntHCFInPlace (*x$*, **ReadOnly** *y$***)**
Computes *x$* = **HCF(***x$*, *y$***)**

Errors **BigIntOperandTooBig** **Len(**x*$***) > &H200**; or **Len(**y*$***) > &H200**; or
both *x$* > 2^{4095} and y*$* > 2^{4095}

Function BigIntInvert (ReadOnly *x$*, **ReadOnly** *n$***) As String**
Returns the inverse of *x$* modulo *n$*; or **HCF(***x$*, *n$***)** if the inversion fails. For *n$* > 1 and *x$* > 0, the inverse of *x$* modulo *n$* is the unique *y$* with 0 < *y$* < *n$* such that *x$* * *y$* **Mod** *n$* = 1. It exists if and only if **HCF(***x$*, *n$***)** = 1.

Sub BigIntInvertInPlace (*x$*, **ReadOnly** *n$***)**
Computes *x$* = inverse of *x$* modulo *n$*; or *x$* = **HCF(***x$*, *n$***)** if the inversion fails.

Errors **BigIntOperandTooBig** *x$* ≥ 2^{2047} or *n$* ≥ 2^{2047}
BigIntDivideByZero *n$* = 0
BigIntInversionFailed **HCF(***x$*, *n$***) <>** 1

Function BigIntSquareRoot (ReadOnly *x$*, **ReadOnly** *p$***) As String**
Returns a square root of *x$* modulo *p$*, i.e. a number 0 ≤ *y$* < *p$* such that *y$* ^{2} **Mod** *p$* = *x$*. If *x$* has no square root modulo *p$*, then 0 is returned. *p$* must be an odd prime number.

Sub BigIntSquareRootInPlace (*x$*, **ReadOnly** *p$***)**
Computes *x$* = square root of *x$* modulo *p$*. If *x$* has no square root modulo *p$*, then 0 is returned. *p$* must be an odd prime number.

Errors **BigIntOperandTooBig** **Len(***p$***) > &H43**
BigIntInvalidOperand *x$* ≥ *p$*
BigIntSquareRootFailed *x$* has no square root modulo *p$*

Function BigIntJacobiSymbol (ReadOnly *a$*, ReadOnly *m$*) As Integer

Returns the Jacobi symbol $\left(\frac{a\$}{m\$}\right)$ for an odd integer *m$*. If *m$* is prime, the Jacobi symbol is equal to 0 if *a$* = 0; +1 if *a$* has a square root mod *m$;* and –1 otherwise. If *m$* is composite, see for instance http://mathworld.wolfram.com/JacobiSymbol.html for a full definition.

Errors	**BigIntOperandTooBig**	*a$/m$* cannot be computed [1], or **Len(***m$***)** > **&H43**
	BigIntInvalidOperand	*m$* is even

Notes: 1. *x$* / *y$* cannot be computed if **Len(***x$***)** + **Len(***y$***)** + *r* > 2556, where
r = min (**Len(***x$***)** – **Len(***y$***)** + 4, **Len(***y$***)**)

7.8 AES: The Advanced Encryption Standard Library

This library implements the Advanced Encryption Standard defined in Federal Information Processing Standard FIPS 197. This standard is available on the Internet, at http://csrc.nist.gov/encryption/aes/. **AES** uses the Rijndael algorithm as its cryptographic primitive. The Standard specifies three permitted key lengths: 128 bits, 192 bits, and 256 bits. All three key lengths are available to Terminal programs. All three key lengths are supported in the **ZC5**-series Professional BasicCards and **ZC6**-series MultiApplication BasicCard**s**; other versions of the BasicCard are restricted to 128-bit keys.

This library implements the Advanced Encryption Standard defined in Federal Information Processing Standard FIPS 197. This standard is available on the Internet, at http://csrc.nist.gov/encryption/aes/. **AES** uses the Rijndael algorithm as its cryptographic primitive. The Standard specifies three permitted key lengths: 128 bits, 192 bits, and 256 bits. All three key lengths are available to Terminal programs. All three key lengths are supported in the **ZC5**-series Professional BasicCards and **ZC6**-series MultiApplication BasicCard**s**; other versions of the BasicCard are restricted to 128-bit keys.

To load this library:

#Include AES.DEF

The file AES.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

The **AES** library consists of a single procedure:

Function AES (*Type%*, *Key$*, *Block$*) As String

This function encrypts or decrypts the 16-byte *Block$* with the given *Key$*, according to the *Type%* parameter:

Type%	
128	Encryption with 128-bit key. **Len (***Key$***)** must be ≥ 16.
192	Encryption with 192-bit key. **Len (***Key$***)** must be ≥ 24.
256	Encryption with 256-bit key. **Len (***Key$***)** must be ≥ 32.
-128	Decryption with 128-bit key. **Len (***Key$***)** must be ≥ 16.
-192	Decryption with 192-bit key. **Len (***Key$***)** must be ≥ 24.
-256	Decryption with 256-bit key. **Len (***Key$***)** must be ≥ 32.

The return value of the function is the encrypted or decrypted *Block$*. If *Block$* is shorter than 16 bytes, it is padded with zeroes before encryption/decryption; if it is longer than 16 bytes, it is truncated before encryption/decryption. In any case, the contents of the original *Block$* are unchanged.

The following error codes are returned in the **LibError** variable:

AesBadType	*Type%* is not ±128, ±192, or ±256.
AesUnsupportedType	*Type%* is ±192 or ±256, but the key length is not supported.
AesKeyTooShort	*Key$* is shorter than 16/24/32 bytes.

7.9 The EAX Library

EAX is an algorithm for Authenticated Encryption. See **9.6 The EAX Algorithm** for a brief description of the algorithm; a full description is available from NIST's web site, at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/. The **EAX** library is currently available for the Terminal program, **ZC5**-series and **ZC7**-series Professional BasicCards, and **ZC6**-series MultiApplication BasicCards. To use the **EAX** library:

#Include EAX.DEF

The **EAX** algorithm takes the following parameters as input:

- A block cipher algorithm. This implementation uses **AES** with key length 128, 192, or 256 bits.
- A cryptographic key for use by the block cipher algorithm.
- A Nonce. This is to ensure that subsequent invocations of **EAX** give different results, even if they encrypt the same data. The Nonce can be any string, which need not be secret, but should be different for each invocation.
- A Header. This contains data that is only authenticated, not encrypted.
- A Message. This contains the data to be encrypted and authenticated.

The algorithm encrypts the message, and computes a 16-byte Tag that authenticates the Header and the Message.

The following procedures are provided:

Function EAXInit (*Type%*, *Key$***) As String**

> This function returns an 87-byte string containing the internal state of the **EAX** algorithm. This string must be provided as the first parameter to all the other procedures in the library. The *Type%* parameter is the length of the key, in bits; it must be 128, 192, or 256. This function need only be called once for a given key.

Sub EAXProvideNonce (*EaxState* **As String**, *Key$*, *N$***)**

> String *N$* contains the Nonce. This can be any string, which need not be secret, but should be different for each invocation. Call this subroutine once for each invocation of the **EAX** algorithm. It must be called before any of the following procedures.

Sub EAXProvideHeader (*EaxState* **As String**, *Key$*, *H$***)**

> This subroutine can be called any number of times, to specify successive parts of the Header. Calls to **EAXProvideHeader** may be interleaved with calls to **EAXComputeCiphertext** or **EAXComputePlaintext**.

Sub EAXComputeCiphertext (*EaxState* **As String**, *Key$, M$***)**

> This subroutine can be called any number of times, to specify successive parts of the Message to be encrypted. The string *M$* is encrypted in place. Calls to **EAXComputeCiphertext** may be interleaved with calls to **EAXProvideHeader**.

Sub EAXComputePlaintext (*EaxState* **As String**, *Key$*, *M$***)**

> This subroutine can be called any number of times, to specify successive parts of the encrypted Message. The string *M$* is decrypted in place. Calls to **EAXComputePlaintext** may be interleaved with calls to **EAXProvideHeader**.

Function EAXComputeTag (*EaxState* **As String**, *Key$***) As String**

> Call this function at the end to compute the Tag. A 16-byte string is returned.

7.10 The OMAC Library

OMAC is an algorithm for Message Authentication. See **9.8 The OMAC Algorithm** for a brief description of the algorithm; a full description is available from NIST's web site, at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/. The **OMAC** library is currently available for the Terminal program, **ZC5**-series and **ZC7**-series Professional BasicCards, and **ZC6**-series MultiApplication BasicCard**s**. To use the **OMAC** library:

#Include OMAC.DEF

The **OMAC** algorithm takes the following parameters as input:

- A block cipher algorithm. This implementation uses **AES** with key length 128, 192, or 256 bits.
- A cryptographic key for use by the block cipher algorithm.
- A Message. This contains the data to be authenticated.

The algorithm computes a 16-byte Tag that authenticates the Message.

The simplest way to calculate the Tag is to use the following function:

Function OMAC (*Type%*, *Key$*, *Mess$***) As String**

> The *Type%* parameter is the length of the key, in bits; it must be 128, 192, or 256. This function computes the Tag for message *Mess$* and returns it as a 16-byte string.

If your message is too long to fit into a string, or if you have multiple messages to authenticate and you want to process them as fast as possible, you can use the incremental procedures:

Function OMACInit (*Type%*, *Key$***) As String**

> This function returns a 34-byte string containing the internal state of the **OMAC** algorithm. This string must be provided as the first parameter to the following library procedures. The *Type%* parameter is the length of the key, in bits; it must be 128, 192, or 256. This function need only be called once for a given key.

Sub OMACStart (*OmacState* **As String)**

> Call this subroutine once for every message, before processing the message data.

Sub OMACAppend (*OmacState* **As String**, *Key$*, *Mess$***)**

> Call this subroutine to add *Mess$* to the message being authenticated. This subroutine can be called any number of times, to authenticate a message of any length.

Function OMACEnd (*OmacState* **As String**, *Key$***) As String**

> This function computes the Tag of the message, returning it as a 16-byte string.

7.11 SHA: The Secure Hash Algorithm Library

This library implements the following Secure Hash Algorithms, as defined in the Federal Information Processing Standards document FIPS 180–2:

Algorithm	*Length of hash*	*Availability*
SHA–1	20 bytes	All programs
SHA–224	28 bytes	Terminal program and **ZC7**-series from **REV C**
SHA–256	32 bytes	Terminal program and **ZC5**-, **ZC6**-, and **ZC7**-series cards
SHA–384	48 bytes	Terminal program and **ZC7**-series from **REV C**
SHA–512	64 bytes	Terminal program and **ZC7**-series from **REV C**

These algorithms take an arbitrary message as input, and output a fixed-length hash of that message. It is supposed to be computationally infeasible to invert these algorithms. More specifically:

- given a hash, it is computationally infeasible to construct a message with that hash;
- it is computationally infeasible to construct two different messages with identical hashes.

To load this library:

#Include SHA.DEF

The file SHA.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

In what follows, *xxx* stands for one of **224**, **256**, **384**, or **512**.

SHA–1 procedure names begin with **Sha**; **SHA–***xxx* procedure names begin with **Sha***xxx*.

7.11.1 Hashing Functions

If a message is contained in a **String**, you can compute its hash with a single function call:

Function ShaHash (*S$***) As String**
Function Sha*xxx***Hash (***S$***) As String**

To hash longer messages, you must use the following procedures:

Professional and MultiApplication BasicCards:

Sub ShaStart (*HashBuff$***)**
Sub ShaAppend (*HashBuff$, S$***)**
Function ShaEnd (*HashBuff$***) As String**

Sub Sha*xxx***Start (***HashBuff$***)**
Sub Sha*xxx***Append (***HashBuff$, S$***)**
Function Sha*xxx***End (***HashBuff$***) As String**

Other Environments:

Sub ShaStart()
Sub ShaAppend (*S$***)**
Function ShaEnd() As String

Sub Sha*xxx***Start()**
Sub Sha*xxx***256Append (***S$***)**
Function Sha*xxx***End() As String**

Call **ShaStart()** (resp. **Sha***xxx***Start()**) to initialise the hashing process, then **ShaAppend (***S$***)** (resp. **Sha***xxx***Append (***S$***)**) for successive blocks of data, and finally **ShaEnd()** (resp. **Sha***xxx***End()**) to get the hash value. In the Professional and MultiApplication BasicCards, the *HashBuff$* argument is used to store the internal state of the hash algorithm; other environments have static buffers for this purpose.

7.11.2 Pseudo-Random Number Generation

The Professional and MultiApplication BasicCards have hardware random number generators; other environments must generate pseudo-random numbers in software. The Secure Hash Algorithm is one source of cryptographically strong pseudo-random numbers. To do this properly, it must be fed with some initial source of random data, for instance user key-strokes (see example program **ECTERM** in directory `BasicCardV8\Examples\EC`).

Sub ShaRandomSeed (*Seed$*)

This function mixes the given seed into the 'randomness pool'.

Function ShaRandomHash() As String

This function returns a random string, 32 bytes long in a Terminal program, 20 bytes long in an Enhanced BasicCard program. Each byte in the string is a random number between 0 and 255 inclusive.

Each time that you call **ShaRandomSeed (*Seed$*)** , the seed is mixed into the 'randomness pool'. The effect is cumulative, so the more data you mix in, the better. The ZC-Basic interpreter mixes in some data of its own each time this procedure is called:

- The Terminal program mixes in the date and time, and the elapsed CPU time for the process.
- The Enhanced BasicCard mixes in its unique serial number. So any two cards will generate different sequences, even if they are fed with the same seeds.

The Enhanced BasicCard has no other internal source of randomness, so you must send it random data from the Terminal program if cryptographically strong random numbers are required, for instance when generating key pairs for use by the **EC–161** Elliptic Curve Cryptography library.

7.12 The TLVLib ASN.1 Library

ZC7-series BasicCards starting from **REV C** contain the **TLVLib** System Library, which provides procedures to parse and build Tag-Length-Value structures as defined in the Abstract Syntax Notation standard **ASN.1** (available at http://www.itu.int/rec/T-REC-X.680-200811-I/en). This library is also available in Terminal programs.

7.12.1 Overview

A Tag-Length-Value structure has the following form:

Tag (an identifying constant)	**Length** of Value field	**Value** field

Tag Encodes a constant which identifies the contents of the **Value** field. In this implementation, it must be one or two bytes long. The top two bits encode the *class* of the tag, which has no significance for this implementation; for details, see the **ASN.1** standard. The next bit, if set, indicates that the **Value** field itself contains one or more nested Tag-Length-Value structures. The next five bits (the remaining bits of the first byte), if all set to 1, indicate a 2-byte Tag; otherwise the Tag is a single byte.

Length Encodes the length of the **Value** field. If the first byte is **&H82**, then the following two bytes contain the length; if the first byte is **&H81**, then the following byte contains the length; and if the first byte is ≤ **&H7F**, then it encodes the length directly. No other values are allowed.

Value The data content of the Tag-Length-Value structure. It may itself consist of nested Tag-Length-Value structures, if bit 6 of the first **Tag** byte is set.

Here is a simple example, a 512-bit RSA public key. The modulus *n*, in hexadecimal, is:

```
AC3B19C84AEE8592AA6EBE098D02EE853768F149FFA1875E203505949807B891\
F42984DD08CE658E939D378C248B7810E2E9636534101673DEF03F9274D4B8F5
```

and the exponent *e* is:

```
1305B
```

This is stored by the RSA System Library as a sequence {*n*,*e*} of two integers, in the following TLV structure:

```
30 48       -- Tag=&H30 (SEQUENCE), Length=&H48
   02 41    -- n: Tag=02 (INTEGER), Length=41
      00 AC 3B 19 C8 4A EE 85 92 AA 6E BE 09 8D 02 EE
      85 37 68 F1 49 FF A1 87 5E 20 35 05 94 98 07 B8
      91 F4 29 84 DD 08 CE 65 8E 93 9D 37 8C 24 8B 78
      10 E2 E9 63 65 34 10 16 73 DE F0 3F 92 74 D4 B8
      F5
   02 03    -- e: Tag=&H02 (INTEGER), Length=&H03
      01 30 5B
```

The values &H30 (SEQUENCE) and &H02 (INTEGER) are defined in the ASN.1 standard. Note that an INTEGER is signed, so a leading zero byte is required if the top bit is set in *n$* or *e$*.

Given *n$* and *e$*, you can build this structure using the **TLVLib** library, as follows:

```
Rem  Add leading zero if required
If (Asc(n$(1)) And &H80) <> 0 Then n$ = Chr$(0) + n$
If (Asc(e$(1)) And &H80) <> 0 Then e$ = Chr$(0) + e$

Rem  Convert to TLV INTEGERs
n$ = TLVCreateObject (&H02, n$)
e$ = TLVCreateObject (&H02, e$)

Rem  Create the TLV SEQUENCE
PublicKey$ = TLVCreateObject (&H30, n$+e$)
```

Conversely, you can recover *n$* and *e$* from *PublicKey$* with the following code:

```
Public Parent As TlvPointer, Child As TlvPointer

Rem  Set Parent.Start=1, Parent.Length=Len(PublicKey$):
Call TLVInitObject (Parent, PublicKey$)

Rem  First (and only) top-level object should be SEQUENCE:
If TLVFirstChild (Parent, Child, PublicKey$) <> &H30 Then_
  GoTo Error

Rem  Step down one level:
Parent = Child

Rem  First element of sequence is INTEGER n$:
If TLVFirstChild (Parent, Child, PublicKey$) <> &H02 Then_
  GoTo Error
n$ = Mid$(PublicKey$, Child.Start, Child.Length)

Rem  Second element of sequence is INTEGER e$:
If TLVNextChild (Parent, Child, PublicKey$) <> &H02 Then_
  GoTo Error
e$ = Mid$(PublicKey$, Child.Start, Child.Length)
```

The **TLVPointer** type is defined in **TLVLib.def** as follows:

```
Type TlvPointer
  Start as Integer
  Length as Integer
End Type
```

The **Start** and **Length** members define the **Value** field of a TLV object, relative to the enclosing string. For example, in the RSA public key above:

	Start	Length
Public key as a whole	&H01	&H4A
SEQUENCE object	&H03	&H48
Modulus n	&H05	&H41
Exponent e	&H48	&H03

7.12.2 Parsing TLV Structures

The following procedures are available for parsing a TLV object into its component parts:

Sub TLVInitObject (*Parent* **As TlvPointer, ReadOnly** *Data$***)**

Initialise *Parent* to the whole of *Data$*, for use in subsequent procedures. This procedure just sets `Parent.Start=1` and `Parent.Length=Len(Data$)`.

Sub TLVInitChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer)**

Initialise *Child* for use in **TLVNextChild** or **TLVNextMatchingChild**. This procedure just sets `Child.Start = Parent.Start` and `Child.Length=0`.

Function TLVFirstChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer, ReadOnly** *Data$***)**

The same as calling **TLVInitChild** followed by **TLVNextChild**.

Function TLVNextChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer, ReadOnly** *Data$***)**

Get the next child of *Parent.* Returns the Tag of the next child, or 0 if *Parent* has no more children.

Function TLVFirstMatchingChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer, ByVal** *Tag*, **ReadOnly** *Data$***)**

The same as calling **TLVInitChild** followed by **TLVNextMatchingChild**.

Function TLVNextMatchingChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer,**
ByVal *Tag*, **ReadOnly** *Data$***)**

Get the next child of *Parent* with the given *Tag*. Returns **True** (-1) if successful, or **False** (0) if *Parent* has no more matching children.

Function TLVLastMatchingChild (ReadOnly *Parent* **As TlvPointer,** *Child* **As TlvPointer,**
ByVal *Tag*, **ReadOnly** *Data$***)**

Get the last child of *Parent* with the given *Tag*. Returns **True** (-1) if successful, or **False** (0) if *Parent* has no matching children.

The last three procedures in this group are for searching for a particular Tag at any level of the hierarchy. This provides a quick way to scan all the nested objects in a TLV structure without having to step up and down the levels.

Sub TLVEnumInit (*Ptr* **As TlvPointer, ReadOnly** *Data$***)**

Initialise *Ptr* for use in **TLVEnumNext**.

Function TLVEnumFirst (*Ptr* **As TlvPointer, ReadOnly** *Data$***)**

The same as calling **TLVEnumInit** followed by **TLVEnumNext**.

Function TLVEnumNext (*Ptr* **As TlvPointer, ReadOnly** *Data$***)**

Get the next object in the flattened TLV tree structure. Returns the Tag of the next object, or 0 if *Data$* has no more objects

7.12.3 Building TLV Structures

The following procedures are available for creating and editing TLV structures:

Function TLVCreateObject (ByVal *Tag* **as Integer, ReadOnly** *Value$***) As String**

The return value is the TLV structure:

Tag	**Len(***Value$***)**	*Value$*

Sub TLVAddChild (ReadOnly *Parent* **As TlvPointer, ByVal** *InsertPos*,
ByVal *Tag* **as Integer, ReadOnly** *Value$, Data$***)**

Add a child with the given *Tag* and *Value$* fields to the *Parent* structure.

Parent	The Start and Length (in the *Data$* string) of the parent object to which the child will be added.
InsertPos	The position to insert the child (relative to the start of *Data$*, not the start of *Parent*).
Tag	The Tag of the child to be added.
Value$	The contents of the child object to be added.
Data$	The string to be updated.

Note: The *Parent* parameter is required to resolve ambiguity when *InsertPos* coincides with the end of a constructed object. For instance:

InsertPos `08`
Tag `02`
Value$ `AB CD`
Data$

```
30 08       -- Outer parent
   30 03    -- Inner parent
      02 01 45
   -- InsertPos is here
   02 01 97
```

Without the *Parent* parameter, we can't know whether to add the child to the inner parent:

```
30 0C
   30 07
      02 01 45
      02 02 AB CD
   02 01 97
```

or to the outer parent:

```
30 0C
   30 03
      02 01 45
   02 02 AB CD
   02 01 97
```

Sub TLVDeleteChild (ReadOnly *Child* **As TlvPointer,** *Data$***)**

Delete the *Child* object from the *Data$* string. For example, suppose *Data$* contains the following:

```
30 0C
   30 07
      02 01 45
      02 02 AB CD
   02 01 97
```

and *Child* represents the TLV object `02 01 45` (so `Child.Start=7` and `Child.Length=1`). Then after calling **TLVDeleteChild**, *Data$* will contain:

```
30 09
   30 04
      02 02 AB CD
   02 01 97
```

(and the *Child* structure will no longer be valid).

Sub TLVReplaceChild (*Child* **As TlvPointer, ByVal** *Tag* **as Integer, ReadOnly** *Value$***,** *Data$***)**

Replace *Child* with the TLV object specified by *Tag* and *Value$*. For example, suppose *Data$* contains the following:

```
30 0C
   30 07
      02 01 45
      02 02 AB CD
   02 01 97
```

and *Child* represents the TLV object `02 01 45` (so `Child.Start=7` and `Child.Length=1`). Then after calling

TLVReplaceChild (*Child***, &H10, Chr$(1,2,3),** *Data$***)**

Data$ will contain:

```
30 0E
   30 09
      10 03 01 02 03
      02 02 AB CD
   02 01 97
```

and the *Child* structure will be updated (so `Child.Start=7` and `Child.Length=3`).

One further utility procedure is provided:

Sub TLVFullObject (*Object* **As TlvPointer, ReadOnly** *Data$***)**

On entry, *Object* specifies the position and size of a Value field. On exit, *Object* specifies the position and size of the full TLV object, including the Tag and Length fields.

7.13 The Mifare™ Library

ZC7- and **ZC8-**series BasicCards from **REV D** can double as Mifare™ cards. Mifare™ is the well-known contactless processor card technology owned by NXP Semiconductors. Information about Mifare™ can be obtained from NXP (this PDF document contains a detailed technical description); many examples of real-life Mifare-based systems can be found at the MIFARE.net website.

The Mifare-capable BasicCards implement the Mifare Classic version, with 1 kilobyte of EEPROM divided into sixteen sectors numbered 0-15. Each 64-byte sector consists of four 16-byte blocks numbered 0-3. Blocks 0-2 of each sector contain data; block 3 contains two 6-byte keys **A** and **B**, and a 4-byte Access Control region **AC**, laid out as **A || AC || B**. For details on the layout and function of these fields, see the PDF document referenced in the preceding paragraph.

On delivery, each key is set to **&HFF, &HFF, &HFF, &HFF, &HFF, &HFF**.

Mifare™ functionality is disabled by default. To enable it:

#Pragma EnableMifare

The card will then function as a Mifare Classic card if it is in the proximity of a Mifare-capable card reader. It will function as a BasicCard in two cases:

- it is in a contact reader; or
- it has received a request for its ATS, from a contactless reader.

If you want to prevent the card being used as a BasicCard by a contactless reader, use the following combination:

#Pragma EnableMifare
#Pragma DisableRF

If the card is functioning as a BasicCard, then the Mifare data blocks can be read and written from the ZC-Basic program using procedures in the Mifare System Library.

Note: In a **ZC8-**series MultiApplication card, access to these Mifare procedures can be controlled through the Card Configuration parameter **MifareAcr –** see **5.3 Card Configuration in ZC8-Series Cards**. To set **MifareAcr**:

#Pragma MifareAcr *ACRName$*

To load the Mifare library:

#Include Mifare.def

Three procedures are provided. Here, *BlockIndex@* is equal to 4 * *SectorNumber* + *BlockNumber*, and *Key$* is equal to the 12-byte concatenation of *KeyA* and *KeyB*:

Sub MifareWriteBlock (*BlockIndex@***,** *Key$***,** *Data$***)**

Write a 16-byte data bock. In a MultiApplication BasicCard, if **MifareAcr** has been set, then **Write** access is required.

Function MifareReadBlock (*BlockIndex@***,** *Key$***) As String**

Read a 16-byte data bock. In a MultiApplication BasicCard, if **MifareAcr** has been set, then **Read** access is required.

Sub MifareResetSector (*SectorNum@***)**

Reset a 64-byte sector to its initial state. In a MultiApplication BasicCard, if **MifareAcr** has been set, then **Delete** access is required.

Note: ZeitControl's development software currently provides no Mifare emulation. In a simulated BasicCard, these functions have no effect. This will be fixed in the near future.

7.14 MATH: Mathematical Functions

The **MATH** library provides standard mathematical functions such as **Exp** and **Sin**. It may only be used in Terminal programs. To load this library:

#Include MATH.DEF

The file MATH.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory.

7.14.1 Error Codes

The **MATH** library procedures can signal the following error codes in **LibError**:

MathDomain	A parameter was outside the valid range, e.g. **Log (–1.0)**
MathSingularity	The function has a singularity at the given point, e.g. **Tan (MathPi / 2)**
MathOverflow	The maximum **Single** value of 3.402823E+38 was exceeded
MathUnderflow	The minimum **Single** value of 1.401298E–45 was truncated to zero
MathLossOfPrecision	Total loss of precision renders the result meaningless, e.g. **Sin (1E30)**

These constants are defined in MATH.DEF.

7.14.2 Integer Rounding

Function Floor (*X!*) As Single	The largest integer ≤ *X!*, as a **Single** value
Function Ceil (*X!*) As Single	The smallest integer ≥ *X!*, as a **Single** value

7.14.3 Exponentiation

Function Pow (*X!*, *Y!*) As Single	*X!* to the power *Y!*
Function Exp (*X!*) As Single	***e*** to the power *X!* (***e*** is the base of natural logarithms)
Function LogE (*X!*) As Single	The natural logarithm of *X!* (i.e. the logarithm to base ***e***)
Function Log10 (*X!*) As Single	The logarithm of *X!* to base 10

7.14.4 Trigonometric Functions

Function Hypot (*X!*, *Y!*) As Single	**Sqrt (** *X!* ***** *X!* **+** *Y!* ***** *Y!* **)** (with no intermediate overflow)
Function Sin (*X!*) As Single	Sine function
Function Cos (*X!*) As Single	Cosine function
Function Tan (*X!*) As Single	Tangent function **Tan (*X!*) = Sin (*X!*) / Cos (*X!*)**
Function ASin (*X!*) As Single	Inverse Sine function (**–π/2 ≤ ASin (*X!*) ≤ π/2**)
Function ACos (*X!*) As Single	Inverse Cosine function (**0 ≤ ACos (*X!*) ≤ π**)
Function ATan (*X!*) As Single	Inverse Tangent function (**–π/2 < ATan (*X!*) < π/2**)
Function ATan2 (*Y!*, *X!*) As Single	Inverse Tangent at **(*X!*, *Y!*)** (**–π < ATan2 (*Y!*, *X!*) ≤ π**)

7.14.5 Hyperbolic Functions

Function SinH (*X!*) As Single	Hyperbolic Sine: **(Exp (*X!*) – Exp (–*X!*)) / 2**
Function CosH (*X!*) As Single	Hyperbolic Cosine: **(Exp (*X!*) + Exp (–*X!*)) / 2**
Function TanH (*X!*) As Single	Hyperbolic Tangent: **SinH (*X!*) / CosH (*X!*)**

7.14.6 Mathematical Constants

The following constants are defined in MATH.DEF:

Const MathE = 2.718281828	The base ***e*** of natural logarithms
Const MathPi = 3.141592654	**π**

7.15 MISC: Miscellaneous Procedures

The **MISC** library provides miscellaneous utility procedures. To load this library:

#Include MISC.DEF

The file MISC.DEF is supplied with the distribution kit, in the `BasicCardV8\Lib` directory. It contains the following procedures, all of which are defined in more detail below:

For Terminal programs:

Timing Functions	**Sub GetDateTime (***DT* **As DateTime)**
	Function TimeInterval (*StartTime* **As DateTime,** *EndTime* **As DateTime) As Long**
	Function UnixTime() As Long
Suspending the Program	**Sub Sleep (***Milliseconds* **As Long)**
Executing a Command Line	**Sub Execute (***CommandString$***)**
CRC Calculations	**Function CRC16 (***S$***) As Integer**
	Sub UpdateCRC16 (*CRC***,** *S$***)**
	Function CRC32 (*S$***) As Long**
	Sub UpdateCRC32 (*CRC* **As Long,** *S$***)**
	Function CCITTCRC16 (*S$***) As Integer**
	Sub UpdateCCITTCRC16 (*CRC*, *S$***)**
Random String	**Sub RandomString (***S$***,** *Len***)**
Communications	**Function ProtocolType() As Byte**
	Function IsPhysicalReader() As Integer
Making a Noise	**Sub Beep (***Frequency***,** *Duration* **As Long)**

For BasicCard programs:

Hardware Data	**Function CardSerialNumber() As String**
Power Management	**Function SetProcessorSpeed (***Speed@***)**

For Professional and MultiApplication BasicCards:

Random String	**Sub RandomString (***S$***,** *Len***)**
Communications	**Function LePresent()**
	Sub SuspendSW1SW2Processing()
Free Memory	**Sub GetFreeMemory (***Mem* **As FreeMemoryData)**

For **ZC7**-series Professional BasicCards:

Communications	**Sub CommParams (***Protocol@*, *Speed@*, *ExtendedLcLe@***)**
CRC Calculations	**Function CCITTCRC16 (***S$***) As Integer**
	Sub UpdateCCITTCRC16 (*CRC*, *S$***)**

7.15.1 Timing Functions

Three timing procedures are provided, for use in Terminal programs only.

Two of these procedures take parameters of type **DateTime**, defined in MISC.DEF:

```
Type DateTime
    Year, Month, Day
    Hour, Minute, Second
    Millisecond
End Type
```

Sub GetDateTime (*DT* **As DateTime)**

Returns the current system date and time in *DT*.

Note: DT is filled in from the system clock. Under Microsoft Windows®, the system clock has a resolution of about 55 milliseconds, which is rounded to a multiple of 10. So values returned by **GetDateTime** will jump in increments of 50 or 60 milliseconds.

Function TimeInterval (*StartTime* As DateTime, *EndTime* As DateTime) As Long

Returns the time interval between *StartTime* and *EndTime*, in milliseconds. This interval will be a multiple of the system clock resolution; see note to **GetDateTime**.

For examples of the use of these procedures, see programs ECINIT.BAS and ECTEST.BAS in directory `BasicCardV8\Examples\EC`.

The third timing procedure returns the number of seconds elapsed since 1st January 1970:
Function UnixTime() As Long

7.15.2 Suspending the Program

In a Terminal program, the following subroutine suspends execution for the specified number of milliseconds:

Sub Sleep (*Milliseconds* As Long)

This frees the CPU for other processes to use.

7.15.3 Executing a Command Line

An operating system command can be executed from a Terminal program using the **Execute** subroutine:

Sub Execute (*CommandString$*)

The following error codes are returned in the **LibError** variable:

MiscFileNotFound	The command string specified a non-existent executable file
MiscNotExecutable	The command string specified a non-executable file
MiscOutOfMemory	Insufficient memory to execute the command
MiscUnexpectedError	The operating system returned an unexpected error code

These constants are defined in MISC.DEF.

Note that it is not possible to retrieve an error code generated by the command itself.

7.15.4 CRC Calculations

Function CRC16 (*S$*) As Integer	Returns the 16-bit CRC of the string *S$*
Sub UpdateCRC16 (*CRC*, *S$*)	Allows cumulative calculation of 16-bit CRC's
Function CRC32 (*S$*) As Long	Returns the 32-bit CRC of the string *S$*
Sub UpdateCRC32 (*CRC* As Long, *S$*)	Allows cumulative calculation of 32-bit CRC's
Function CCITTCRC16 (*S$*) As Integer	Returns the 16-bit CCITT CRC of the string *S$*
Sub UpdateCCITTCRC16 (*CRC*, *S$*)	Allows cumulative calculation of 16-bit CCITT CRC's

To calculate the CRC of a single **String** value, call **CRC16** or **CRC32**. To calculate CRC's for larger amounts of data, first initialise *CRC* to zero, then call **UpdateCRC16** or **UpdateCRC32** with successive values of *S$*.

The CCITT variants were introduced for the **ZC7**-series Professional BasicCard, which has a hardware CRC co-processor for calculating a 16-bit CCITT-compatible CRC. They are available in Terminal programs and **ZC7**-series BasicCards.

Here are three ‘C’ functions to calculate these CRC variants:

```
unsigned short CRC16 (unsigned char *p, unsigned int len)
  {
  unsigned short crc = 0 ;
  while (len--)
    {
    crc ^= *p++ ;
    int i ; for (i = 0 ; i < 8 ; i++)
      if (crc & 1) crc >>= 1, crc ^= 0xCA00 ;
      else crc >>= 1 ;
    }
  return crc ;
  }

unsigned long CRC32 (unsigned char *p, unsigned int len)
  {
  unsigned long crc = 0 ;
  while (len--)
    {
    crc ^= *p++ ;
    int i ; for (i = 0 ; i < 8 ; i++)
      if (crc & 1) crc >>= 1, crc ^= 0xA3000000 ;
      else crc >>= 1 ;
    }
  return crc ;
  }

unsigned short CCITTCRC16 (unsigned char *p, unsigned int len)
  {
  unsigned short crc = 0 ;
  while (len--)
    {
    crc ^= *p++ ;
    int i ; for (i = 0 ; i < 8 ; i++)
      if (crc & 1) crc >>= 1, crc ^= 0x8408 ;
      else crc >>= 1 ;
    }
  return crc ;
  }
```

7.15.5 Communications

The Terminal program can query whether the card reader on the current **ComPort** is physical or virtual:

Function IsPhysicalReader() As Integer

This function returns **True** (&HFFFF) if the reader exists and is physical, and **False** (0) if the reader doesn't exist or is virtual (i.e. simulated). Also, if the reader doesn't exist, **SW1SW2** contains an error code.

The Terminal program can detect the protocol in use by the card in the currently active **ComPort**:

Function ProtocolType() As Byte

Return value	0	**T=0** protocol active
	1	**T=1** protocol active
	2	**T=CL** contactless protocol active (simulated card only)

If contactless protocol is in use by a real card, the Terminal program is unable to detect this – PC/SC card readers simulate **T=0** or **T=1** protocol in such cases. This function only returns a value of 2 if a simulated **ZC7**-series Professional BasicCard is using contactless protocol (i.e. if 251 ≤ **ComPort** ≤ 254).

Similarly, the **ZC7**-series Professional BasicCard can detect various currently active protocol parameters:

Sub CommParams (*Protocol@*, *Speed@*, *ExtendedLcLe@***)**

Protocol@	0, 1, or 2 for **T=0**, **T=1**, or **T=CL** (contactless) protocol
Speed@	**PPS1** parameter from the **PPS** request (or 0 if no **PPS** request was received). **PPS1** sets the communication speed (but note that the meaning of this parameter depends on whether **T=CL** protocol is currently active).
ExtendedLcLe@	1 if the current command was sent using extended $\mathbf{L_c}$/$\mathbf{L_e}$ protocol (for commands and responses up to 2048 bytes in length); otherwise 0

Function LePresent()

This function returns **True** (&HFFFF) or **False** (0) according to whether the **Le** field is present in the command APDU. It is available in all current Professional and MultiApplication BasicCards. See **Chapter 8: Communications** for more information about the **Le** field.

Normally, if **SW1-SW2 <> &H9000**, and **SW1 <> &H61**, then **ODATA** is not sent – see **8.6 Commands and Responses**. You can override this behaviour in some BasicCards with the following procedure call:

Sub SuspendSW1SW2Processing()

The card will then send the **ODATA** field in the response, regardless of the value of **SW1-SW2**. This procedure only affects the current command. See **3.3.13 The #Pragma Directive** for an alternative method.

7.15.6 Making a Noise

The Terminal program can generate an audible beep with the **Beep** subroutine:

Sub Beep (*Frequency***,** *Duration* **As Long)**

The duration is in milliseconds.

Note: The *Frequency* and *Duration* parameters are only effective under Windows® NT, Windows® 2000, and later systems; they are ignored under Windows® 98 (although they must be present).

7.15.7 Random String

In the Terminal program and in all current Professional BasicCards, a **String** variable can be filled with random data:

Sub RandomString (*S$***,** *Len***)**

On return, *S$* contains *Len* bytes of random data.

7.15.8 Card Serial Number

In BasicCard programs, the card's unique Serial Number is available as an 8-byte string:

Function CardSerialNumber() As String

In Professional and MultiApplication BasicCards, this is the same number that is returned by the **GET APPLICATION ID** command when **P2 = 3** – see **8.9.10 The GET APPLICATION ID Command**.

7.15.9 Free Memory

In **ZC7**-series Professional BasicCards from **REV C**, and all MultiApplication BasicCards, you can find out the state of the various memory allocation heaps.

Free Memory in the Professional BasicCard

Sub GetFreeMemory (*Mem* **As ProFreeMemoryData)**

The **ProFreeMemoryData** structure contains the total free memory and the size of the largest free block in the RAM and EEPROM heaps:

```
Type ProFreeMemoryData
  TotalFreeRam
  LargestFreeRamBlock
  TotalFreeEeprom As Long
  LargestFreeEepromBlock As Long
End Type
```

Free Memory in the MultiApplication BasicCard

Sub GetFreeMemory (*Mem* **As FreeMemoryData)**

For each heap, the total free memory and the size of the largest free block are returned in a **HeapData** structure:

```
Type HeapData
  TotalFreeMemory%
  LargestFreeBlock%
End Type

Type FreeMemoryData
  RamHeapData As HeapData
  AppFileHeapData As HeapData
  EepromHeapData As HeapData
End Type
```

See **5.2.4 Memory Allocation** for more information on heaps in the MultiApplication BasicCard.

7.15.10 Power Management

The new high-performance chips used in the latest BasicCards run more than twice as fast as previous versions. However, they require more power the faster they run. So you can adjust the processor speed with the **SetProcessorSpeed** function. The way to do this depends on the card type, as follows:

Power Management in the Enhanced BasicCard

Enhanced BasicCards from **REV C** support the following processor speeds, in MHz:

3.75 5 7.5 10 12 15 20 30 33.75 37.5 41.25 45 48.75 52.25 56.25 60

To set the processor speed:

Function SetProcessorSpeed (*MHz@***) As Byte**

This function sets the processor speed to the nearest supported value, and returns the previous value (rounded to the nearest integer). **SetProcessorSpeed (0)** leaves the processor speed unchanged, and just returns the current value.

When the card is reset, the processor speed defaults to 60MHz. You can change this behaviour with the **#Pragma Clock** directive – see **Processor Speed** in **3.3.13 The #Pragma Directive** for details.

Power Management in the ***ZC7****- and* ***ZC8-****series BasicCards*

In the **ZC7**- and **ZC8-**series BasicCards, you can set the speeds of the various co-processors independently:

Function SetProcessorSpeed (*Clock@***) As Byte**

Clock@ is the hardware clock control register. The function returns the previous value of *Clock@*.

The clock control register contains three fields (speeds are in MHz):

C CPU speed, encoded in bits 3-0:

Speed	4	8	12	18	24	31
Encoding	4	5	6	8	9	10

R RSA/EC co-processor speed, encoded in bits 7 to 5:

Speed	4	8	18	36	48	72
Value	0	1	2	3	6	7

D DES/AES co-processor speed, encoded in bit 4:

Speed	18 or 36	4
Value	0	1

If bit 4 is set, the co-processor runs at 4MHz; otherwise, it runs at 18 or 36 MHz, depending on the CPU speed *C*.

The default clock speeds are slower in contactless RF mode than in contact mode, to reduce power consumption. In contact mode, *Clock@* is **&HEA**, so (*C*, *R*, *D*) is (31, 72, 0). In contactless mode, *Clock@* is **&H58**, so (*C*, *R*, *D*) is (18, 18, 4). You can specify different start-up values with **#Pragma Clock** and **#Pragma RFClock –** see **Processor Speed** in **3.3.13 The #Pragma Directive** for details.

Note: If contactless protocol is active, changing the processor speed of a card at run-time will cause fluctuations in the power consumption of the card, which can interfere with the RF communication. For contactless protocol, you should usually rely on the **#Pragma RFClock** directive to set the processor speed at start-up.

Power Management in Other BasicCards

The **ZC5**-series Professional BasicCard, and the **ZC6**-series MultiApplication BasicCard, require a processor speed divisor:

Function SetProcessorSpeed (*Divisor@***) As Byte**

The higher the value of *Divisor@* , the slower the processor speed. The function returns the previous value of *Divisor@*.

Divisor@ is rounded down to the nearest value that is supported by the processor: 1, 2, 4, or 8. **SetProcessorSpeed (1)** sets the maximum processor speed, and **SetProcessorSpeed (8)** sets the minimum processor speed. **SetProcessorSpeed (0)** leaves the processor speed unchanged, and just returns the current value.

Part II

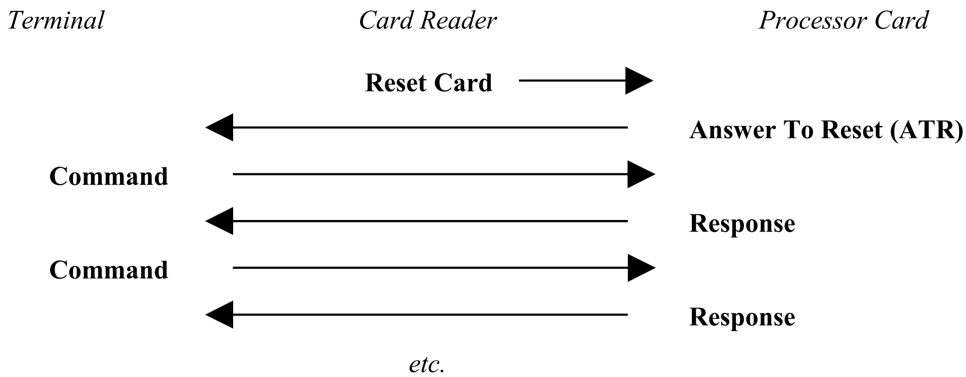
Technical Reference

8. Communications

Note: Throughout this chapter, **bold** numbers are hexadecimal.

8.1 Overview

As outlined in **1.1 Processor Cards**, communication between a Terminal and a Processor Card proceeds, via a Card Reader, as a series of Commands (initiated by the Terminal) and Responses (sent by the Processor Card). The series starts with the Card Reader sending a **Reset Card** signal to the Processor Card:



Two documents describe this process in detail:

1. **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*

This document describes the communication between the Card Reader and the Processor Card, from the bit level through the byte level to the block level. We will be concerned with three aspects:

- the structure of the **ATR**;
- the **T=0** character transmission protocol;
- the **T=1** block transmission protocol.

2. **ISO/IEC 7816-4**: *Interindustry commands for interchange*

This document describes Commands and Responses. We will be concerned with three aspects:

- the contents of Commands and Responses;
- the method by which the **T=0** protocol transmits Commands and Responses;
- the method by which the **T=1** protocol transmits Commands and Responses.

We provide a summary of these documents in the following sections. Most readers can skip these sections; they are provided mainly for users who need to program the BasicCard to be compatible with existing systems.

In these documents, a Command or Response is referred to as an **APDU** (application protocol data unit). The structure of Command and Response **APDU**'s is described in **8.6 Commands and Responses**.

8.2 Answer To Reset

With the Answer To Reset (**ATR**), the Processor Card identifies itself and indicates which protocols it supports. Most of the data in the **ATR** is not relevant to a BasicCard programmer. The following information is important:

- whether the card supports the **T=0** and/or the **T=1** protocols;
- the maximum communication speed that the card allows;
- the Historical Characters.

The Enhanced BasicCards support only the **T=1** protocol, at 9600 baud. They send the following **ATR** (the byte names are from ISO/IEC):

TS	T0	TB1	TC1	TD1	TD2	TA3	TB3	T1-TK
3B	**EF**	**00**	**FF**	**81**	**31**	**50** or **20**	**45** or **75**	'BasicCard ZC*vvv*'

Briefly, what this means is:

TS = 3B	Direct convention (high = **1**, low = **0**; least significant bit arrives first)
T0 = EF	**E** → **TB1**, **TC1**, **TD1** follow; **F** → 15 historical characters
TB1 = 00	No EEPROM programming voltage required
TC1 = FF	Waiting time between two characters = 11 **ETU**
TD1 = 81	**TD2** follows (**T=1** indication)
TD2 = 31	**TA3**, **TB3** follow (**T=1** indication)
TA3 = 50 or **20**	**IFSC** (Information Field Size) **=** &H50 in Compact card, &H20 in Enhanced card
TB3 = 45 or **75**	**CWT** (character waiting time) = (11 + 32) **ETU** (= 3.33 ms between characters) In ZC1.1, ZC3.3, and ZC3.5 cards (**TB3 = 45**): **BWT** (block waiting time) = (11 + 16*960) **ETU** (= 1.6 seconds between blocks) In later cards (**TB3 = 75**): **BWT** (block waiting time) = (11 + 128*960) **ETU** (= 12.8 seconds between blocks)
T1-TK	The historical characters (*vvv* is the BasicCard firmware version number)

An **ETU** (elementary time unit) is one bit, or 372 clock cycles. The timing figures assume a clock frequency of 3.57 MHz. All these fields can be configured in ZC-Basic with **#Pragma ATR (***ATR-Spec***)** – see **3.21.1 Customised ATR**.

The Professional BasicCards are more flexible in their capabilities; they support the **T=0** protocol as well as the **T=1** protocol, and they can run at up to 38400 baud. Here is a typical **ATR** (from the Professional BasicCard **"ZC4.5D REV C"**):

TS	T0	TA1	TB1	TC1	TD1	TC2	T1-TK
3B	**FC**	**13**	**00**	**FF**	**40**	**80**	'ZC4.5D REV C'

TS = 3B	Direct convention (high = **1**, low = **0**; most significant bit arrives first)
T0 = FC	**F** → **TA1**, **TB1**, **TC1**, **TD1** follow; **C** → 12 historical characters
TA1 = 13	**FI = 1**; **DI = 3** → maximum allowed communication speed = 38400 baud
TB1 = 00	No EEPROM programming voltage required
TC1 = FF	Waiting time between two characters = 11 **ETU**
TD1 = 40	**TC2** follows (**T=0** indication)
TC2 = 80	**WI** = 128 → **WWT** (work waiting time) = 12.8 seconds

More examples are available in the file **BasicCardV8\Inc\ATRList.Def**, supplied with the distribution kit. This file contains the **ATR** of every currently available BasicCard.

8.3 The T=0 Protocol

The **T=0** protocol is a character-level transmission protocol for integrated circuit cards with contacts, defined in the document **ISO/IEC 7816-3**: *Electronic signals and transmission protocols*. Some Professional BasicCards support the **T=0** protocol, as well as the **T=1** protocol described in the next section. **T=1** is faster, easier to use, and less error-prone; you should only use the **T=0** protocol if you are implementing a pre-existing **T=0** command set, or you need to use card readers that don't support the **T=1** protocol.

The **T=0** protocol is defined as a sequence of messages exchanged between the **IFD** (interface device) and the **ICC** (integrated circuit card). In the present context, the **IFD** is the Terminal program, and the

ICC is the BasicCard. The exchange begins when the **ICC** is powered up and responds with an **ATR** (Answer To Reset). Thereafter the **IFD** sends a **TPDU** (transmission protocol data unit) containing a Command, and the **ICC** replies with a **TPDU** containing the Response. A **TPDU** is a lower-level object than an **APDU**; we will see later how **APDU**'s are constructed from **TPDU**'s.

8.3.1 TPDU Transmission

When the **IFD** sends a Command **TPDU** and the **ICC** replies with a response **TPDU**, only one of the two **TPDU**'s may contain data. If the Command **TPDU** contains data, it is an *incoming data transfer*; if the Response **TPDU** contains data, it is an *outgoing data transfer*. The **T=0** protocol does not provide any mechanism for specifying which of the two **TPDU**'s may contain data; and in fact the protocol grinds to a halt if the **IFD** and **ICC** don't agree on the direction of data transfer.

In both cases, the **IFD** first sends a 5-byte command header:

CLA	**INS**	**P1**	**P2**	**P3**	→

CLA Class byte – first byte of two-byte **CLA INS** command identifier. This byte may not be **FF**.
INS Instruction byte – second byte of two-byte **CLA INS** command identifier. **INS** must be even, and the top nibble may not be **6** or **9**.
P1 Parameter 1 of 4-byte **CLA INS P1 P2** command header.
P2 Parameter 2 of 4-byte **CLA INS P1 P2** command header.
P3 Number of data bytes.

From the command header, the **ICC** must be able to determine whether the **IFD** expects an incoming or outgoing data transfer.

Incoming Data Transfer

Command **TPDU**:	**CLA**	**INS**	**P1**	**P2**	**P3**	$\mathbf{D_1}$	**...**	$\mathbf{D_{P3}}$
Response **TPDU**:	**SW1**	**SW2**						

The **ICC** acknowledges the 5-byte command header by echoing the **INS** byte (more variations are described in the **ISO/IEC** document, but the BasicCard does not use them):

←	**INS**

The **IFD** then sends **P3** bytes of data:

$\mathbf{D_1}$	**...**	$\mathbf{D_{P3}}$	→

The **ICC** responds with a two-byte status code:

←	**SW1**	**SW2**

where the top nibble of **SW1** is **6** or **9** (but **SW1=60** is not allowed). Status codes are described in **8.8 Status Bytes SW1 and SW2**.

Outgoing Data Transfer

Command **TPDU**:	**CLA**	**INS**	**P1**	**P2**	**P3**
Response **TPDU**:	$\mathbf{D_1}$	**...**	$\mathbf{D_{P3}}$	**SW1**	**SW2**

The **ICC** acknowledges the 5-byte command header by echoing the **INS** byte, and then sends **P3** data bytes, followed by a two-byte status code:

←	INS	D_1	...	D_{P3}	SW1	SW2

In both cases, the **ICC** may reject the command by responding immediately with **SW1-SW2** instead of echoing **INS**.

If the **WWT** work waiting time is exceeded, the **IFD** will time out. The **ICC** can restart the timer, and so delay the time out, by sending a **NULL** (**60**) byte. In a BasicCard program, this is done with the **WTX** statement:

WTX *n*

The ZC-Basic syntax requires the parameter *n*, although it is ignored if the card is using **T=0** protocol.

8.3.2 APDU Transmission by T=0

This section describes the methods defined by ISO/IEC for implementing **APDU** exchanges under **T=0**. If you are not familiar with the structure of Command and Response **APDU**'s, you should read **8.6 Commands and Responses** before continuing.

There are four cases to consider. We adhere to the notation in **ISO/IEC 7816-4**: *Interindustry commands for interchange*, **Annex A** (normative): **Transportation of APDU messages by T=0**:

Case 1: **Lc=0**, and **Le** not present: no incoming data, and no outgoing data
Case 2: **Lc=0**, and **Le** present: outgoing data only
Case 3: **Lc** non-zero, and **Le** not present: incoming data only
Case 4: **Lc** non-zero, and **Le** present: incoming and outgoing data

8.3.3 Case 1: No Incoming Data or Outgoing Data

The Command **TPDU** consists of the Command **APDU** with **P3=0** appended:

Command **APDU**:	CLA	INS	P1	P2

Command **TPDU**:	CLA	INS	P1	P2	P3=0

Response **TPDU**:	SW1	SW2

Response **APDU**:	SW1	SW2

8.3.4 Case 2: Outgoing Data Only

Case 2S.1 – Le accepted

If the **ICC** accepts the value of **Le** supplied by the **IFD**, the Command and Response **TPDU** are identical to the Command and Response **APDU**:

Command **APDU**:	CLA	INS	P1	P2	Le

Command **TPDU**:	CLA	INS	P1	P2	P3=Le

Response **TPDU**:	D_1	...	D_{P3}	SW1	SW2

Response **APDU**:	D_1	...	D_{Le}	SW1	SW2

Case 2S.2 – Le definitely not accepted

If the **ICC** does not accept **Le**, and does not want to suggest an alternative, it replies with **SW1-SW2=6700**:

Command **APDU:**

CLA	**INS**	**P1**	**P2**	**Le**

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=Le**

Response **TPDU**:

67	**00**

Response **APDU**:

67	**00**

Case 2S.3 – Le not accepted, La indicated

If the **ICC** does not accept **Le**, and has an alternatve **La** to suggest, it responds with **SW1-SW2** = **6C La**, and the **IFD** can re-issue the command to receive the outgoing data:

Command **APDU:**

CLA	**INS**	**P1**	**P2**	**Le**

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=Le**

Response **TPDU**:

6C	**La**

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=La**

Response **TPDU**:

$\mathbf{D_1}$	**...**	$\mathbf{D_{La}}$	**SW1**	**SW2**

Response **APDU**:

$\mathbf{D_1}$	**...**	$\mathbf{D_{La}}$	**61**	**La**

Case 2S.4 – Command not accepted

Command **APDU:**

CLA	**INS**	**P1**	**P2**	**Le**

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=Le**

Response **TPDU**:

SW1	**SW2**

Response **APDU**:

SW1	**SW2**

with **SW1=6***X* except **6C**, or **SW1-SW2=9***XXX* except **9000**.

8.3.5 Case 3: Incoming Data Only

The Command and Response **TPDU** are identical to the Command and Response **APDU**:

Command **APDU:**

CLA	**INS**	**P1**	**P2**	**Lc**	$\mathbf{D_1}$	**...**	$\mathbf{D_{Lc}}$

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=Lc**	$\mathbf{D_1}$	**...**	$\mathbf{D_{P3}}$

Response **TPDU**:

SW1	**SW2**

Response **APDU**:

SW1	**SW2**

8.3.6 Case 4: Incoming and Outgoing Data

The Command **TPDU** is identical to the Command **APDU**, but with **Le** removed:

Command **APDU:**

CLA	**INS**	**P1**	**P2**	**Lc**	**D_1**	**...**	**D_{Lc}**	**Le**

Command **TPDU:**

CLA	**INS**	**P1**	**P2**	**P3=Lc**	**D_1**	**...**	**D_{P3}**

Depending on the response, the **IFD** may issue a **GET RESPONSE** Command to request the outgoing data. This command has **INS**=**C0**, **P1=0**, **P2=0**, but the ISO/IEC document leaves the **CLA** byte unspecified. ZeitControl's Terminal software (the **IFC**) uses **CLA=0**; the BasicCard operating system accepts any value for **CLA** that is not a user-defined command.

Case 4S.1 – Command not accepted

Response **TPDU**:

SW1	**SW2**

Response **APDU**:

SW1	**SW2**

with **SW1=6***X* except **61**, or **SW1-SW2=9***XXX* except **9000**.

Case 4S.2 – Command accepted

Response **TPDU**:

90	**00**

The **IFD** issues a **GET RESPONSE** Command:

Command **TPDU:**

CLA=00	**INS=C0**	**P1=00**	**P2=00**	**P3=Le**

Transmission then proceeds as in *Case 2*.

Case 4S.3 – Command accepted with information added

The **ICC** accepts the command, and indicates that **Lx** bytes of outgoing data are available:

Response **TPDU**:

61	**Lx**

The **IFD** issues a **GET RESPONSE** Command, with **P3**=**min(Le,Lx)**:

Command **TPDU:**

CLA=00	**INS=C0**	**P1=00**	**P2=00**	**P3**

Transmission then proceeds as in *Case 2*.

8.4 The T=1 Protocol

The **T=1** protocol is a block-level transmission protocol for integrated circuit cards with contacts, defined in the document **ISO/IEC 7816-3**: *Electronic signals and transmission protocols.* The BasicCard contains a full implementation of this **T=1** standard, including **NAD** awareness, chaining, retries, **WTX** requests, and **IFS** requests. This section describes those parts of the **T=1** protocol that a programmer of the BasicCard might want to know: (i) the error-free transmission of I-blocks; (ii) the **WTX** request. The mechanisms for chaining, error handling, and **IFS** adjustment are hidden from the programmer, and are not described here. For a detailed definition of the **T=1** protocol, see document **ISO/IEC 7816-3**.

8.4.1 APDU Transmission by T=1

The **T=1** protocol is defined as a sequence of messages exchanged between the **IFD** (interface device) and the **ICC** (integrated circuit card). In the present context, the **IFD** is the Terminal program, and the **ICC** is the BasicCard. The exchange begins when the **ICC** is powered up and responds with an **ATR** (Answer To Reset). Thereafter the **IFD** sends an **APDU** containing a Command, and the **ICC** replies with an **APDU** containing the Response. In between receiving a command and sending its response, the **ICC** may transmit a **WTX** request (waiting time extension), to ask for more time:

IFD		**ICC**
	←	**ATR**
Command **APDU**	→	
	←	Response **APDU**
Command **APDU**	→	
	←	**WTX** request
WTX response	→	
	←	Response **APDU**
	. . .	

Each **APDU** is transmitted in one or more *I-blocks*. An I-block is the fundamental unit of transmission in the **T=1** protocol; successive I-blocks are chained together to produce the Command and Response **APDU**'s. In the following example, **APDU** is the concatenation of **INF**$_1$, **INF**$_2$, and **INF**$_3$:

IFD		**ICC**
Chained I-block containing **INF**$_1$	→	
	←	Request for next I-block
Chained I-block containing **INF**$_2$	→	
	←	Request for next I-block
Unchained I-block containing **INF**$_3$	→	

The maximum allowed length of an I-block depends on the direction of transmission, and on protocol parameters that can vary dynamically; it is typically 32-128 bytes.

8.4.2 Structure of an I-block

An I-block contains the following fields. All fields are one byte, except the **INF**:

I-block:	**NAD**	**PCB**	**LEN**		**INF**		**LRC**

NAD Node Address byte. The low nibble contains the Node Address (0-7) of the sender, and the high nibble contains the Node Address (0-7) of the intended recipient. The BasicCard responds to all Node Address values: the **NAD** of the response I-block is equal to the **NAD** of the command I-block with the high and low nibbles reversed.

PCB Protocol control byte. Alternates between **00** and **40** (unless chaining is in progress). The BasicCard programmer can ignore this byte.

LEN The length of the **INF** field in bytes.

INF Information field – the information content of the I-block. The **T=1** protocol says nothing about the internal format of the **INF** field.

LRC Longitudinal redundancy check. A simple **Xor** of all the preceding bytes.

8.4.3 WTX Request

The **BWT** (block waiting time) defined in the **ATR** tells the **IFD** how long to wait for a response before timing out. The BasicCard **ATR** defines a **BWT** of 1.6 seconds (BasicCard versions ZC1.1, ZC3.3, and ZC3.5), or 12.8 seconds (all other BasicCards). If a command is going to take longer than this, it must request more time using a **WTX** (waiting time extension) request. In ZC-Basic, this takes the form

WTX *BWT-units*

BWT-units A **Byte** expression, giving the requested time in multiples of the **BWT**. **WTX** requests are not cumulative; the time allowed is counted from the time of the request, and cancels any previous **WTX** requests.

A **WTX** request contains the following fields:

WTX request:	**NAD**	**PCB=C3**	**LEN=01**	**INF**	**LRC**

The **INF** field has length 1, and contains the value *BWT-units*. The response to this request contains an identical **INF** field:

WTX response:	**NAD**	**PCB=E3**	**LEN=01**	**INF**	**LRC**

8.5 The T=CL Contactless Protocol

The **ZC7**- and **ZC8-**series BasicCards are available in a Dual Interface version, which implements the **T=CL** Contactless protocol defined in International Standard **ISO/IEC 14443**. In the terminology of that standard, the card is a **PICC Type A**. All such cards support the **T=0** and **T=1** protocols too.

The **T=CL** protocol is similar to the **T=1** protocol described in the previous section. It is defined as a sequence of messages exchanged between the **PCD** (Proximity Coupling Device, or contactless card reader) and the **PICC** (Proximity Card). In the present context, the **PCD** is the Terminal program, and the **PICC** is the BasicCard. The exchange begins with an Anticollision Loop which the **PCD** performs when it detects a card within range; this serves to select a single card unambiguously, even if more than one card is present. Then the **PCD** asks the card for its **ATS** (Answer To Select). Thereafter the **PCD** sends an **APDU** containing a Command, and the **PICC** replies with an **APDU** containing the Response. In between receiving a command and sending its response, the **PICC** may transmit a **WTX** request (waiting time extension), to ask for more time:

PCD		**PICC**
Anticollision Loop	→	
	←	**ATS**
Command **APDU**	→	
	←	Response **APDU**
Command **APDU**	→	
	←	**WTX** request
WTX response	→	
	←	Response **APDU**
	. . .	

Each **APDU** is transmitted in one or more *I-blocks*. An I-block is the fundamental unit of transmission in the **T=CL** protocol; successive I-blocks are chained together to produce the Command and Response **APDU**'s. In the following example, **APDU** is the concatenation of **INF**$_1$, **INF**$_2$, and **INF**$_3$:

PCD		**PICC**
Chained I-block containing **INF**$_1$	→	
	←	Request for next I-block
Chained I-block containing **INF**$_2$	→	
	←	Request for next I-block
Unchained I-block containing **INF**$_3$	→	

The maximum allowed length of an I-block depends on the direction of transmission, and on protocol parameters that can vary dynamically; it ranges from 16 to 256 bytes.

8.5.1 Answer To Select

With the Answer To Select (**ATS**), the PICC identifies itself and indicates which protocol parameters it supports:

TL	**T0**	[**TA1**]	[**TB1**]	[**TC1**]	**T1-TK**	**EDC**

Briefly, what this means is:

TL	Length of the **ATS** (including **TS** itself but excluding **EDC**)
T0	Codes **FSC** (Frame Size for proximity Card) and the presence of **TA1**, **TB1**, **TC1**
TA1	Specifies the transmission speeds supported by the card
TB1	Codes **FWT** (Frame Waiting Time) and **SFGT** (Start-up Frame Guard Time)
TC1	Specifies whether the **CID** and **NAD** bytes are supported. The **ZC7**-series BasicCard supports **CID** but not **NAD**.
T1-TK	Historical Bytes
EDC	A 2-byte CRC over all preceding bytes

All these fields can be configured in ZC-Basic with **#Pragma ATS (***ATS-Spec***)** – see **3.21.2 Customised ATS**.

8.5.2 Structure of an I-block

An I-block contains the following fields:

I-block:	**PCB**	[**CID**]	[**NAD**]	**INF**	**EDC**

PCB	Protocol control byte. Indicates whether the **CID** and **NAD** fields are present, and whether the block is chained.
CID	Card Identifier byte. This optional field is used to select a card in case there is more than one active card in range of the **PCD**.
NAD	Node Address byte, for selecting one of multiple logical channels in a single card. This optional field is not supported by the BasicCard.
INF	Information field – the information content of the I-block. The **T=1** protocol says nothing about the internal format of the **INF** field.
EDC	Error Detection Code, a 2-byte CRC over all preceding bytes.

If you want to view the **T=CL** protocol at this level, you will need to run a simulated card with 251 ≤ **ComPort** ≤ 254, and look at the log file – PC/SC contactless card readers hide these details from the host computer.

From the point of view of the BasicCard programmer, the **T=CL** protocol is indistinguishable from the **T=1** protocol. In certain circumstances (for instance, when deciding whether to allow important data to be overwritten), you might want to know which protocol is currently active. You can query the protocol parameters with procedure **CommParams** in the **MISC** System Library – see **7.15.5 Communications** for details.

8.6 Commands and Responses

This section describes the contents of commands and responses, as defined in the document **ISO/IEC 7816-4**: *Interindustry commands for interchange*. The **APDU** of a command has the following structure (shaded blocks are optional):

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**Le**

CLA Class byte – first byte of two-byte **CLA INS** command identifier. If the **T=0** protocol is used, this byte may not be **FF**.

INS Instruction byte – second byte of two-byte **CLA INS** command identifier. For ISO compatibility, this byte should be even. If the **T=0** protocol is used, the top nibble may not be **6** or **9**.

P1 Parameter 1 of 4-byte **CLA INS P1 P2** command header.

P2 Parameter 2 of 4-byte **CLA INS P1 P2** command header.

Lc Length of **IDATA** field in command.

IDATA Data expected by command. In the case of a ZC-Basic command, this field contains the parameters passed by the caller.

Le Expected length of **ODATA** field in response (supplied by caller).

In the BasicCard, **CLA** and **INS** can refer to pre-defined commands (all of which have **CLA=C0**) or ZC-Basic commands (**CLA** and **INS** are specified by the programmer for each command). **P1** and **P2** are retained in the BasicCard for **ISO** compatibility; you can use them if you like, or ignore them. If you want to use them, the parameters passed to you by the caller are available as **Public Byte** variables **P1** and **P2**; and you can specify their values in commands that you call using the *PreSpec* field described in **3.15.3 Calling a Command**:

Call *command-name* **(**[**P1=***expr***,**] [**P2=***expr***,**] *arg-list***)**

The **APDU** of a response has the following structure (the shaded block is optional):

ODATA		SW1	SW2

ODATA Data returned by command. In the case of a ZC-Basic command, this field contains the parameters that were passed by the caller, as modified by the called command.

SW1 First status byte.

SW2 Second status byte.

SW1 and **SW2** are pre-defined **Public** variables of type **Byte**. Before a command is executed, they have the values **&H90** and **&H00**, which is a standard status code meaning “Command successfully completed”. If you want to return an error code to the caller, just set **SW1** and **SW2** to the appropriate values before you exit the command.

Notes:

- if **SW1-SW2 <> &H9000**, and **SW1 <> &H61**, then **ODATA** is discarded: any return values are lost. In some Professional BasicCards, you can override this behaviour – see **3.3.13 The #Pragma Directive** and **7.15.5 Communications**.
- in a card using the **T=0** protocol, the high nibble of **SW1** must be **6** or **9**.

8.7 Extended-Length Commands

In most BasicCards, the **Lc** and **Le** fields, if present, must be a single byte. But **ISO 7816-3** defines a protocol for commands and responses up to 65535 bytes long. This protocol is supported by the **ZC7**- and **ZC8-**series BasicCards. Any command that is 7 bytes or longer, with the fifth byte equal to **00**, is an extended-length command. If the command is exactly seven bytes long, then the sixth and seventh bytes constitute **Le**. Otherwise, the sixth and seventh bytes constitute **Lc**, and the **Le** field, if present, is also two bytes long.

The **ZC7**- and **ZC8-**series BasicCards allow values of **Lc** and **Le** up to 2048.

8.8 Status Bytes SW1 and SW2

8.8.1 BasicCard Operating System

The following status codes are returned by the BasicCard operating system (codes marked with ***** are returned by the MultiApplication BasicCard only):

swCommandOK	**9000**	Command successfully completed.
sw1LeWarning	**61***XX*	Command successfully completed, but **Le** was not equal to *XX*.
swRetriesRemaining	**63C***X*	A command was wrongly encrypted, and the error counter for the active key has been decremented to *X*. If *X* reaches zero, the key is disabled.
sw1PCodeError	**64***XX*	P-Code error *XX* occurred in the BasicCard. (The P-Code error codes are described in the next section.)
swEepromWriteError	**6581**	A write to EEPROM failed. (This is a hardware error.)
swBadEepromHeap	**6582**	The EEPROM heap is in an inconsistent state.
swBadFileChain	**6583**	The BasicCard File System is in an inconsistent state.
swKeyNotFound	**6611**	The key specified in a **START ENCRYPTION** command was not configured with a **Declare Key** statement in the BasicCard program.
swKeyTooShort	**6613**	The cryptographic key specified in a **START ENCRYPTION** command was too short for the algorithm. All algorithms require at least 8-byte keys; some algorithms requires 16-byte or 24-byte keys.
swKeyDisabled	**6614**	The active key has been disabled, either explicitly with a **Disable Key** statement, or automatically when its error counter reached zero.
swUnknownAlgorithm	**6615**	Parameter **P1** in a **START ENCRYPTION** command does not specify a valid algorithm.
swAlreadyEncrypting	**66C0**	A **START ENCRYPTION** command was received while encryption was already active.
swNotEncrypting	**66C1**	An **END ENCRYPTION** command was received while encryption was not active.
swDesCheckError	**66C3**	The active encryption algorithm is **Single DES** or **Triple DES**, and the authentication bytes in a command were invalid.
swCoprocessorError	**66C4**	The Crypto-Coprocessor has reported an internal error.
swAesCheckError	**66C5**	The active encryption algorithm is **AES**, and the authentication bytes in a command were invalid.
***swBadSignature**	**66C6**	An **AUTHENTICATE FILE** command contained an invalid signature.
***swBadAuthenticate**	**66C7**	Invalid **VERIFY** or **EXTERNAL AUTHENTICATE** command.
swLcLeError	**6700**	Either **Lc** has an unexpected value; or **Le** is absent when it should be present, or present when it should be absent.
swCommandTooLong	**6781**	A command will not fit in the command buffer. In most cards, this is 256 bytes; in **ZC7**-series Professional BasicCards, it is 2048 bytes.
swResponseTooLong	**6782**	The response from the card is too long to be sent.
swInvalidState	**6985**	A built-in command was called, but the state of the BasicCard is invalid for the command.
swCardUnconfigured	**6986**	The card has not been configured by ZeitControl.

swNewStateError	**6987**	The state of the BasicCard has been changed with a **SET STATE** command. After a **SET STATE** command, the BasicCard must be reset before it will accept any furhter commands.
***swBadComponentName**	**69C0**	A Component name contained an invalid character.
***swComponentNotFound**	**69C1**	A Component was not found in the BasicCard.
***swAccessDenied**	**69C2**	The required access conditions were not satisfied.
***swComponentAlreadyExists**	**69C3**	A Component with the given name already exists.
***swBadComponentChain**	**69C4**	The card's internal Component chain has become corrupted. Contact ZeitControl for assistance.
***swNameTooLong**	**69C5**	The full path name of the Component is longer than 254 characters.
***swOutOfMemory**	**69C6**	The BasicCard has insufficient free memory to execute the command.
***swInvalidACR**	**69C7**	An ACR has an unrecognised type.
***swBadComponentType**	**69C8**	A Component is not of the required type.
***swKeyUsage**	**69CD**	Current usage not enabled in Key's Usage attribute.
***swKeyAlgorithm**	**69CE**	Current algorithm not enabled in Key's Algorithm attribute.
***swTooManyTempFlags**	**69D0**	The limit of 64 temporary Flags has been reached.
***swExecutableAcrDenied**	**69D1**	The Application file does not satisfy the "**\Executable**" ACR.
***swApplicationNotFound**	**69D2**	Application file not found.
***swACRDepth**	**69D3**	Compound ACR's can be nested to a limit of at most 5 levels.
***swBadComponentAttr**	**69D4**	Attempt to write invalid Component Attributes.
***swBadComponentData**	**69D5**	Attempt to write invalid Component Data.
***swBadAppFile**	**69D6**	The file is not a valid Application file.
***swLoadSequenceActive**	**69D7**	Attempt to activate LoadSequence or delete a Component when LoadSequence is already active.
***swLoadSequenceNotActive**	**69D8**	Attempt to close or abort a non-existent LoadSequence.
***swLoadSequencePhase**	**69D9**	Invalid Phase parameter to LoadSequence command.
***swBadEaxTag**	**69DC**	Invalid **EAX** tag received during Secure Transport.
***swSecureTransportActive**	**69DD**	Attempt to activate Secure Transport when already active.
***swSecureTransportInactive**	**69DE**	Attempt to close non-existent Secure Transport session.
***swComponentReferenced**	**69DF**	Attempt to delete a Component referenced by another Component.
swP1P2Error	**6A00**	**P1** or **P2** is invalid for the command.
swOutsideEeprom	**6A02**	An invalid address was passed in **P1P2** to one of the built-in EEPROM access commands.
swDataNotFound	**6A88**	The built-in command **GET APPLICATION ID** returns this error code if no Application ID was configured in the BasicCard.
sw1LaWarning	**6C***XX*	Command successfully completed, but **La** was not equal to *XX*.
swINSNotFound	**6D00**	The **INS** byte of the command was not recognised (although the **CLA** byte was valid).
swCLANotFound	**6E00**	The **CLA** byte of the command was not recognised.
SwInternalError	**6F00**	An unexpected error condition was detected.

8.8.2 BasicCard P-Code Interpreter

If the P-Code interpreter in the BasicCard detects an error, it returns **sw1PCodeError** (**64**) in **SW1**, and the specific P-Code error in **SW2**. The P-Code error is one of the following:

pcStackOverflow	**01**	The P-Code stack has grown beyond its comfigured size.
pcDivideByZero	**02**	A division by zero (or a Mod with zero divisor) occurred.
pcNotImplemented	**03**	An unimplemented P-Code instruction (e.g. **XMIT**) was executed.
pcBadRamHeap	**04**	Corruption of RAM has left the heap in an inconsistent state.
pcBadEepromHeap	**05**	Corruption of EEPROM has left the heap in an inconsistent state.
pcReturnWithoutGoSub	**06**	A Return statement was executed with no corresponding GoSub.
pcBadSubscript	**07**	One of the subscripts in an array access was out of bounds.
pcBadBounds	**08**	One of the array subscript bounds in a ReDim statement was out of range.
pcInvalidReal	**09**	A floating-point operand was not a valid IEEE-format number.
pcOverflow	**0A**	The result of an arithmetic operation was too large or small for the destination.
pcNegativeSqrt	**0B**	An attempt was made to take the square root of a negative number.
pcDimensionError	**0C**	An array parameter did not have the expected number of dimensions.
pcBadStringCall	**0D**	An invalid parameter was passed to a string function.
pcOutOfMemory	**0E**	There was not enough free memory left to complete the instruction.
pcArrayNotDynamic	**0F**	The array parameter in a ReDim statement was not Dynamic.
pcArrayTooBig	**10**	The array size requested in a ReDim statement was too large.
pcDeletedArray	**11**	An attempt was made to access an element of a deleted array.
pcPCodeDisabled	**12**	A previous P-Code error has disabled the BasicCard. The card must be reset before it can execute P-Code again.
pcBadSystemCall	**13**	A SYSTEM instruction had an invalid sub-function code.
pcBadKey	**14**	An invalid key number was passed to a cryptographic function.
pcBadLibraryCall	**15**	An invalid Plug-In Library function was called.
pcStackUnderflow	**16**	The P-Code stack has shrunk to a negative size.

8.8.3 Terminal P-Code Interpreter

The P-Code interpreter in the Terminal program can return the following status codes in **SW1-SW2**:

swNoCardReader	**6790**	No card reader detected on the given COM port.
swCardReaderError	**6791**	An invalid reply was received to a card reader command.
swNoCardInReader	**6792**	No card is inserted in the card reader.
swCardPulled	**6793**	The card has been removed from the card reader.
swT1Error	**6794**	An unrecoverable **T=1** protocol error occurred while communicating with the card.
swCardError	**6795**	An invalid response was received to a BasicCard command.

swCardNotReset	**6796**	The card has not been reset. A BasicCard must be reset before the Terminal program can send it any commands.
swKeyNotLoaded	**6797**	The key specified in a **START ENCRYPTION** command is unknown to the Terminal program.
swCardTimedOut	**679A**	The card did not respond within the time allowed.
swTermOutOfMemory	**679B**	The Terminal program has insufficient free memory to process the response.
swBadDesResponse	**679C**	The active encryption algorithm is **Single DES** or **Triple DES**, and the authentication bytes in a response were invalid.
swInvalidComPort	**679D**	The COM port is not in the range 1-4.
swNoPcscDriver	**679F**	No PC/SC driver is installed on the PC.
swPcscReaderBusy	**67A0**	The PC/SC reader is busy.
swPcscError	**67A1**	An unexpected PC/SC error occurred.
swComPortBusy	**67A2**	Another process is using the COM port.
swBadATR	**67A3**	The BasicCard returned an invalid **ATR**.
swT0Error	**67A4**	A **T=0** protocol error occurred.
swPTSError	**67A7**	An error occurred during Protocol Type Selection.
swDataOverrun	**67A8**	The Terminal has lost characters sent by the card reader.
swBadAesResponse	**67A9**	The active encryption algorithm is **AES**, and the authentication bytes in a response were invalid.
swReservedINS	**6D80**	An attempt was made to send a forbidden **INS** in **T=0** protocol.
swReservedCLA	**6E80**	An attempt was made to send **CLA=FF** in **T=0** protocol.

8.9 Pre-Defined Commands

8.9.1 States of the BasicCard

The Enhanced BasicCard has four states:

NEW: The card is in state **NEW** before ZeitControl configures it.

LOAD: The card is in state **LOAD** when the application developer gets it.

TEST: State **TEST** lets the application developer test software in the card.

RUN: The card is in state **RUN** when it is issued to the end user.

The Professional BasicCard has five states:

NEW: The card is in state **NEW** before ZeitControl configures it.

LOAD: The card is in state **LOAD** when the application developer gets it.

PERS: State **PERS** is for initialising the file system: files can be created and accessed by anybody, but ZC-Basic code cannot be run.

TEST: State **TEST** lets the application developer test software in the card.

RUN: The card is in state **RUN** when it is issued to the end user. This state is permanent.

The card can be switched between **LOAD**, **PERS**, and **TEST** any number of times, but the **RUN** state is permanent. Once the card is switched to state **RUN**, it can't be re-programmed.

The MultiApplication BasicCard has the same five states as the Professional BasicCard:

NEW:	The card is in state **NEW** before ZeitControl configures it.
LOAD:	Reserved to ZeitControl for EEPROM initialisation.
PERS:	In state **PERS**, everybody has access to all Components, and the **CLEAR EEPROM** command is enabled.
TEST:	State **TEST** is identical to state **RUN**, except that the card can be switched back to state **PERS**.
RUN:	The card is in state **RUN** when it is issued to the end user. This state is permanent.

In the MultiApplication BasicCard, state **LOAD** is normally only available to ZeitControl; the card will usually be in state **PERS** when the application developer gets it, and can only be switched between states **PERS**, **TEST**, and **RUN**. Contact our Sales department if you need to write directly to EEPROM in state **LOAD**.

8.9.2 Pre-Defined Commands – a Summary

Single-Application BasicCards

The operating system in a single-application BasicCard contains twelve or thirteen pre-defined commands. All commands have class byte **CLA = C0**. The **INS** byte takes the values **00, 02, 04, . . . , 16, 18**, as follows:

GET STATE	**00**	Get the state and version of the card
EEPROM SIZE	**02**	Get the address and length of EEPROM
CLEAR EEPROM	**04**	Set specified bytes to FF
WRITE EEPROM	**06**	Load data into EEPROM
READ EEPROM	**08**	Read data from EEPROM
EEPROM CRC	**0A**	Calculate CRC over a specified EEPROM address range
SET STATE	**0C**	Set the state of the card
GET APPLICATION ID	**0E**	Get the Application ID string
START ENCRYPTION	**10**	Start automatic encryption of command/response data
END ENCRYPTION	**12**	End automatic encryption
ECHO	**14**	Echo the command data
FILE IO	**18**	Execute a file system operation

Most of these commands are enabled only when the BasicCard is in an appropriate state. The following table summarises which internal commands are valid in which states:

	NEW	LOAD	PERS	TEST	RUN
GET STATE	✓	✓	✓	✓	✓
EEPROM SIZE	✓	✓			
CLEAR EEPROM	✓	✓			
WRITE EEPROM	✓	✓			
READ EEPROM	✓	✓	*	*	*
EEPROM CRC	✓	✓			
SET STATE	✓	✓	✓	✓	
GET APPLICATION ID				✓	✓
START ENCRYPTION				✓	✓
END ENCRYPTION				✓	✓
ECHO	✓	✓	✓	✓	✓
FILE IO		**	✓	✓	✓

* The **READ EEPROM** command is allowed in states **PERS**, **TEST**, and **RUN** if encryption with key number **0** is enabled (see **8.9.7 The READ EEPROM Command**).

** In the Enhanced BasicCard only, the **FILE IO** command is allowed in state **LOAD**.

In state **NEW**, no checks are performed on addresses of EEPROM reads and writes. (This is to allow ZeitControl to install upgrades to the BasicCard operating system, before delivery to the application developer.)

In state **LOAD**, the EEPROM access commands are restricted to user EEPROM.

In a single-application BasicCard, these commands will typically be called at the following points in the development cycle:

1. Write and test a ZC-Basic application on the PC
2. **EEPROM SIZE** – check that the card has the expected EEPROM size
3. **CLEAR EEPROM** – set EEPROM to a known state
4. **WRITE EEPROM** – download the application to the card
5. **EEPROM CRC** – check that the EEPROM was correctly written
6. **FILE IO** – create files and directories
7. **SET STATE** to **TEST** and reset the card
8. Run the application in the card
9. **SET STATE** to **LOAD** and reset the card
10. **READ EEPROM** to check any EEPROM changes made by the application

(Most of this is handled automatically by the ZeitControl development software.) When the application is written and tested, cards can be switched into the **RUN** state for delivery to end users.

Applications in the MultiApplication BasicCard will normally be loaded by the Application Loader, built into the ZCMSim command-line interpreter and the ZCMDCard BasicCard debugger.

MultiApplication BasicCard

The operating system in the MultiApplication BasicCard contains the following commands in addition to those listed above:

GET CHALLENGE	**40**	Get a cryptographic Challenge for **EXTERNAL AUTHENTICATE**
EXTERNAL AUTHENTICATE	**42**	Authenticate the Terminal program to the BasicCard
INTERNAL AUTHENTICATE	**44**	Authenticate the BasicCard to the Terminal program
VERIFY	**46**	Verify the user's password or PIN
GET FREE MEMORY	**48**	Get the amount of free memory available in the global heap
SELECT APPLICATION	**A0**	Select an Application
CREATE COMPONENT	**A2**	Create a Component
DELETE COMPONENT	**A4**	Delete a Component
WRITE COMPONENT ATTR	**A6**	Write a Component's attributes
READ COMPONENT ATTR	**A8**	Read a Component's attributes
WRITE COMPONENT DATA	**AA**	Write a Component's data
READ COMPONENT DATA	**AC**	Read a Component's data
FIND COMPONENT	**AE**	Get the CID of a Component from its name
COMPONENT NAME	**B0**	Get the name of a Component from its CID
GRANT PRIVILEGE	**B2**	Grant a Privilege to a File
AUTHENTICATE FILE	**B4**	Authenticate a File with a Signature
READ RIGHTS LIST	**B6**	Read the Privileges and Signatures of a File
LOAD SEQUENCE	**B8**	Start, end, or abort a Load Sequence session
SECURE TRANSPORT	**BA**	Start or end a Secure Transport session
WRITE CARD CONFIG	**BC**	Write a Card Configuration data item
READ CARD CONFIG	**BE**	Read a Card Configuration data item

8.9.3 The GET STATE Command

GET STATE – Get the state and version of the card

Command syntax:

CLA	INS	P1	P2	Le
C0	**00**	**00**	**00**	**00**

Response:

ODATA	SW1	SW2
state (1 byte), *version* (n bytes)	**61**	n+1

This command returns the state and version of the BasicCard.

The *state* byte (Enhanced BasicCards):

state:	**00**	**01**	**02**	**03**
State of card:	**NEW**	**LOAD**	**TEST**	**RUN**

The *state* byte (Professional and MultiApplication BasicCards):

state:	**00**	**01**	**02**	**03**	**04**
State of card:	**NEW**	**LOAD**	**PERS**	**TEST**	**RUN**

The length of the *version* field depends on the card type, as follows:

Old Enhanced BasicCard: n = 2: major version number (**03**) followed by minor version number
Other card types: n ≥ 3: the version info is an ASCII string

Command-Specific Error Codes in SW1-SW2:

swLcLeError **Lc** is present, or **Le** is absent
swP1P2Error **P1 <> 00** or **P2 <> 00**

To call **GET STATE** from a Terminal program:

```
#Include Commands.def
Call GetState (state@, version$)
```

8.9.4 The EEPROM SIZE Command

EEPROM SIZE – Get the address and length of EEPROM

This command returns **2 * GASize** bytes (see **10.1 Address Metrics**).

Command syntax:

CLA	INS	P1	P2	Le
C0	**02**	**00**	**00**	**04** or **06**

Response:

ODATA	SW1	SW2
start (**GASize** bytes), *length* (**GASize** bytes)	**90**	**00**

Returns the start address and length of loadable EEPROM.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present, or **Le** is absent
swInvalidState	Card is not in **NEW** or **LOAD** state
swP1P2Error	**P1 <> 00** or **P2 <> 00**

To call **EEPROM SIZE** from a Terminal program:

```
#Include Commands.def

Rem  If GASize = 2:
Call EepromSize (start%, length%)

Rem  If GASize = 3:
Private start&, start$ As String*3 At start&+1
Private length&, length$ As String*3 At length&+1
Call EepromSize24 (start$, length$)
```

8.9.5 The CLEAR EEPROM Command

CLEAR EEPROM – Clear specified bytes (to **00** or **FF**, depending on the processor type)

Single-Applications BasicCards

If **GASize = 2** (see **10.1 Address Metrics**):

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**04**	*addr*	**02**	*length* (2 bytes)

If **GASize = 3**:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	**04**	**00**	**00**	**06**	*addr* (3 bytes), *length* (3 bytes)

Response:

SW1	SW2
90	**00**

Clears *length* bytes of EEPROM to **00** or **FF**, starting from address *addr*.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc <> 02** (*resp*. 6), or length of **IDATA <> 02** (*resp*. 6)
swInvalidState	Card is not in **NEW** or **LOAD** state
swOutsideEeprom	Address range not wholly contained in EEPROM

To call **CLEAR EEPROM** from a Terminal program:

```
#Include Commands.def

Rem  If GASize = 2:
Call ClearEeprom (P1P2=addr%, length%)

Rem  If GASize = 3:
Private addr&, addr$ As String*3 At addr&+1
Private length&, length$ As String*3 At length&+1
Call ClearEeprom24 (addr$, length$)
```

MultiApplication BasicCard in state **PERS**

Command syntax:

CLA	INS	P1	P2
C0	**04**	**00**	**00**

Response:

SW1	SW2
90	**00**

Clears all of EEPROM to **00** or **FF**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** present
swInvalidState	Card is not in state **NEW**, **LOAD**, or **PERS**

To call **CLEAR EEPROM** from a Terminal program for the MultiApplication BasicCard:

```
#Include Commands.def
Call ClearEeprom (Lc=0, 0)
```

8.9.6 The WRITE EEPROM Command

WRITE EEPROM – Load data into EEPROM

If **GASize = 2** (see **10.1 Address Metrics**):

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**06**	*addr*	*len*	*data*

If **GASize = 3**:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	**06**	**00**	**00**	*len+3*	*addr* (3 bytes), *data*

Response:

SW1	SW2
90	**00**

Writes *data* (*len* bytes) to EEPROM starting at address *addr*.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** <> length of **IDATA**
swInvalidState	Card is not in **NEW** or **LOAD** state
swOutsideEeprom	Address range not wholly contained in EEPROM

To call **WRITE EEPROM** from a Terminal program:

```
Rem  If GASize = 2:
Declare Command &HC0 &H06 WriteEeprom (data$, Disable Le)
Call WriteEeprom (P1P2=addr%, data$)

Rem  If GASize = 3:
Declare Command &HC0 &H06 WriteEeprom24 (addr$ As String*3,
                data$, Disable Le)
Private addr&, addr$ As String*3 At addr&+1
Call WriteEeprom24 (addr$, data$)
```

Note: For security reasons, the **WRITE_EEPROM** command is encrypted, and is not available for general use. Calling this command from a user program is likely to damage the card irreparably. For this reason, it is not included in **Commands.def**. However, it is possible to call this command with data supplied by the compiler in the Image File – see the **BCLoad.exe** source code in `BasicCardV8\Source\BCLoad` for an example of how to do this. In such cases, you must declare the **WriteEeprom** command yourself, as shown above.

8.9.7 The READ EEPROM Command

READ EEPROM – Read data from EEPROM

If **GASize = 2** (see **10.1 Address Metrics**):

Command syntax:

CLA	INS	P1P2	Le
C0	**08**	*addr*	*len*

If **GASize = 3**:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**08**	**00**	**00**	**03**	*addr* (3 bytes)	*len*

Response:

ODATA	SW1	SW2
data (*len* bytes)	**90**	**00**

Reads *len* bytes from EEPROM starting from address *addr*. If you have configured key number **00** in the card, then the **READ EEPROM** command can be called whatever the state of the card, by enabling encryption with key **00**. You should consider this option whenever the card contains data that is not available elsewhere – if the card becomes unusable for any reason, for example because of hardware errors writing to EEPROM, you can recover the data this way.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present, or **Le** is absent
swInvalidState	Card is not in **NEW** or **LOAD** state, and key **00** is not active
swOutsideEeprom	Address range not wholly contained in EEPROM

To call **READ EEPROM** from a Terminal program:

```
#Include Commands.def

Rem  If GASize = 2:
Call ReadEeprom (P1P2=addr%, data$, Le=len@)

Rem  If GASize = 3:
Private addr&, addr$ As String*3 At addr&+1
Call ReadEeprom24 (addr$, data$, Le=len@)
```

8.9.8 The EEPROM CRC Command

EEPROM CRC – Calculate a CRC over a specified EEPROM address range

If **GASize = 2** (see **10.1 Address Metrics**):

Command syntax:

CLA	INS	P1P2	Lc	IDATA	Le
C0	**0A**	*addr*	**02**	*length* (2 bytes)	**02**

If **GASize = 3**:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	**0A**	**00**	**00**	**06**	*addr* (3 bytes), *length* (3 bytes)

Response:

ODATA	SW1	SW2
CRC (2 bytes)	**90**	**00**

Returns the CRC of *length* bytes from address *addr*. All bytes must be in EEPROM. This command can be used to verify the contents of EEPROM after downloading an application to the card.

In the Enhanced BasicCard, this command also serves to enable the BasicCard file system. To access the file system while the card is still in state **LOAD**, an **EEPROM CRC** command must be sent, to let the card know that the relevant data structures have been downloaded; the **BCLoad** program does this automatically after downloading a ZC-Basic program to the BasicCard.

Warning: Do not call this command in the Enhanced BasicCard before a valid ZC-Basic program has been loaded. The card will attempt to enable a non-existent file system, which can permanently disable the card. (In the Professional BasicCards, you can call this command at any time.)

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc <> 02** (*resp*. **06**) or length of **IDATA <> 02** (*resp*. **06**) or **Le** not present
swInvalidState	Card is not in **NEW** or **LOAD** state
swOutsideEeprom	Address range not wholly contained in EEPROM

To call **EEPROM CRC** from a Terminal program:

```
#Include Commands.def

Rem  If GASize = 2:
Call EepromCRC (P1P2=addr%, length%)
```

The CRC is returned in the `length%` variable.

```
Rem  If GASize = 3:
Private addr&, addr$ As String*3 At addr&+1
Private length&, length$ As String*3 At length&+1
Call EepromCRC24 (addr$, length$, CRC%)
```

Note: If **Le** ≥ **03** (*resp*. **09**), the Professional and MultiApplication BasicCards return a 32-bit CRC. To call the 32-bit **EEPROM CRC** command from a Terminal program:

```
#Include Commands.def

Rem  If GASize = 2:
CRCHi% = length%
Call EepromCRC32 (P1P2=addr%, CRCHi%, CRCLo%)

Rem  If GASize = 3:
Private addr&, addr$ As String*3 At addr&+1
Private length&, length$ As String*3 At length&+1
Call Eeprom24CRC32 (addr$, length$, CRC&)
```

16-bit and 32-bit CRC calculations are described in **7.15.4 CRC Calculations**.

8.9.9 The SET STATE Command

SET STATE – Set the state of the card

Command syntax:

CLA	INS	P1	P2
C0	**0C**	*state*	**00**

Response:

SW1	SW2
90	**00**

This command changes the state of the card, as folllows:

Enhanced BasicCards

state:	**01**	**02**	**03**
New card state:	**LOAD**	**TEST**	**RUN**

Professional and MultiApplication BasicCards

state:	**01**	**02**	**03**	**04**
New card state:	**LOAD**	**PERS**	**TEST**	**RUN**

After this command is successfully called, no further commands are allowed until the card is reset.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** present
swInvalidState	Card is in **RUN** state
swCardUnconfigured	The card has not been configured by ZeitControl. If you see this error, contact ZeitControl for a replacement card.
swP1P2Error	**P1 = 00** or **P1 > RUN** or **P2 <> 00**

To call **SET STATE** from a Terminal program:

```
#Include Commands.def
Call SetState (P1=state@)
```

8.9.10 The GET APPLICATION ID Command

GET APPLICATION ID – Get the Application ID string

Single-Application BasicCards

Command syntax:

CLA	INS	P1	P2	Le
C0	**0E**	**00**	**00** or **03**	**00**

Response:

ODATA	SW1	SW2
Data	**61**	*len*

P2 = **00:** *Data* contains the Application ID specified in the ZC-Basic source code statement:

Declare ApplicationID = *Application-ID*

P2 = 03: In **ZC7-**series Professional BasicCards, and **ZC5**-series cards from **REV G**, *Data* contains the 8-byte hardware Serial Number of the card, as returned by the **MISC** System Library function **CardSerialNumber()**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present or **Le** is absent
swInvalidState	Card is not in **TEST** or **RUN** state
swP1P2Error	**P1 <> 00** or **P2 <> 00**
swDataNotFound	Application ID not configured

To call **GET APPLICATION ID** from a Terminal program for a single-application BasicCard:

```
#Include Commands.def
Call GetApplicationID (ApplicationID$)
```

MultiApplication BasicCards

Command syntax:

CLA	INS	P1	P2	Le
C0	**0E**	*index*	*type*	**00**

Response:

ODATA	SW1	SW2
Data	**61**	*len*

type	*Data*
0	Application ID of Application specified in *index*
1	Filename of Application specified in *index*
2	Configured CardID (*index* must be zero). For **ZC6**-series cards, the contents of the special file "**\CardID**" – see **5.4.2 Card ID File**; for **ZC8-**series cards, the Card Configuration data item **CardID** – see **5.3 Card Configuration in ZC8-Series Cards**.
3	8-byte hardware Serial Number of the card (*index* must be zero).

For *type* equal to **0** or **1**, if *index* is equal to zero, it refers to the currently selected Application. If *index* is equal to n with $n \geq 1$, it refers to the n^{th} executable Application file (according to the order in which the files were created).

To call **GET APPLICATION ID** from a Terminal program for the MultiApplication BasicCard:

```
#Include Commands.def
Call GetApplicationID (P1=index@, P2=type%, Data$)
```

8.9.11 The START ENCRYPTION Command

START ENCRYPTION – Start automatic encryption of command/response data

This command initiates automatic encryption of command and response data fields. Its format depends on the card type.

Enhanced BasicCard:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	10	*algorithm*	*key*	04	*Random number* **RA** (4 bytes)	04

Response:

ODATA	SW1	SW2
Random number **RB** (4 bytes)	90	00

Professional BasicCard:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	10	*algorithm*	*key*	len_R	*Random number* **RA** (len_R bytes)	00

Response:

ODATA	SW1	SW2
algorithm (1 byte); *Random number* **RB** (len_R bytes)	61	len_R+1

MultiApplication BasicCard:

Command syntax:

CLA	INS	P1P2	Lc	IDATA	Le
C0	10	*KeyCID*	len_R+1	*algorithm* (1 byte); *Random number* **RA** (len_R bytes)	00

Response:

ODATA	SW1	SW2
algorithm (1 byte); *Random number* **RB** (len_R bytes)	61	len_R+1

algorithm is one of the following cryptographic algorithms, defined in the file **AlgID.def**:

algorithm		len_R	*key length*
AlgSingleDes	**Single DES** (Data Encryption Standard, 8-byte key)	4	**8**
AlgTripleDes	**Triple DES-EDE2** (Data Encryption Standard, 16-byte key)	4	**10** (=16_{10})
AlgSingleDesCrc	**Single DES** with CRC	4	**8**
AlgTripleDesEDE2Crc	**Triple DES-EDE2** with CRC	4	**10** (=16_{10})
AlgTripleDesEDE3Crc	**Triple DES-EDE3** with CRC	4	**18** (=24_{10})
AlgAes128	**AES-128** (Advanced Encryption Standard, 128-bit key)	8	**10** (=16_{10})
AlgAes192	**AES-192** (Advanced Encryption Standard, 192-bit key)	8	**18** (=24_{10})
AlgAes256	**AES-256** (Advanced Encryption Standard, 256-bit key)	8	**20** (=32_{10})
AlgEaxAes128	**EAX** with **AES-128**	8	**10** (=16_{10})
AlgEaxAes192	**EAX** with **AES-192**	8	**18** (=24_{10})
AlgEaxAes256	**EAX** with **AES-256**	8	**20** (=32_{10})
AlgOmacAes128	**OMAC** with **AES-128**	0	**10** (=16_{10})
AlgOmacAes192	**OMAC** with **AES-192**	0	**18** (=24_{10})
AlgOmacAes256	**OMAC** with **AES-256**	0	**20** (=32_{10})

For descriptions of these algorithms, and the role of **RA** and **RB**, see **Chapter 9: Encryption Algorithms**.

In single-application BasicCards, *key* is the key number. It must match one of the key numbers configured in the BasicCard program with the ZC-Basic **Declare Key** statement, of length at least *key length* from the above table. If the **START ENCRYPTION** command is successful, the pre-defined variable **KeyNumber** is set equal to *key*.

In the MultiApplication BasicCard, *KeyCID* is the CID of the Key . If the **START ENCRYPTION** command is successful, the pre-defined variable **SMKeyCID** is set equal to *KeyCID*.

Algorithms supported in the Enhanced BasicCard

The Enhanced BasicCard supports algorithms **AlgSingleDes** and **AlgTripleDes**.

Algorithms supported in the Professional BasicCard

The different Professional BasicCard versions support various combinations of cryptographic algorithms. See the **Professional BasicCard Datasheet** for up to date information. At the time of writing, the following versions are available:

BasicCard	*Algorithms*
ZC5.4	**AlgAes128**, **AlgAes192**, **AlgAes256**, **AlgSingleDesCrc**, **AlgTripleDesEDE2Crc**
ZC5.5	All the algorithms in the above table from **AlgSingleDesCRC** to **AlgOmacAes256**
ZC7-series	All the algorithms in the above table from **AlgSingleDesCRC** to **AlgOmacAes256**

Algorithms supported in the MultiApplication BasicCard

The MultiApplication BasicCard supports all the algorithms in the above table from **AlgSingleDesCRC** to **AlgOmacAes256**.

Automatic Algorithm Selection

If *algorithm* is zero, then the card automatically selects the strongest algorithm that is compatible with len_R and the key length. In the Professional and MultiApplication BasicCards, the algorithm thus selected is returned in the first byte of **ODATA**.

Command-Specific Error Codes in SW1-SW2:

swKeyNotFound	Key number *key* was not configured
swKeyTooShort	Key number *key* is too short
swKeyDisabled	Key number *key* is disabled
swUnknownAlgorithm	*algorithm* is unknown, or is not enabled in the card
swAlreadyEncrypting	Encryption is already enabled
swLcLeError	*Enhanced BasicCards*: **Lc** <> **04**, or **Le** is absent
	Professional and MultiApplication BasicCards: **RA** too short, or **Le** absent
swInvalidState	Card is not in **TEST** or **RUN** state

To call **START ENCRYPTION** from a Terminal program for an Enhanced BasicCard, or a Professional BasicCard with **DES** support:

```
#Include Commands.def
Call StartEncryption ([P1=Algorithm@,] P2=KeyNumber@, Rnd)
```

To call **START ENCRYPTION** from a Terminal program for a Professional BasicCard:

```
#Include Commands.def
Call ProEncryption ([P1=Algorithm@,] P2=KeyNumber@, Rnd, Rnd)
```

Note that both forms are accepted by a Professional BasicCard with **DES** support.

To call **START ENCRYPTION** from a Terminal program for a MultiApplication BasicCard, see **5.7 Secure Messaging**.

Alternatively, for all BasicCard types, **Commands.def** defines the subroutine **AutoEncryption**, which automatically selects the correct version of the command:

```
#Include Commands.def
Call AutoEncryption (KeyNumber@, KeyName$)
```

where *KeyName$* is the name of the Key in the MultiApplication BasicCard (so it can be the empty string if the card is known to be a single-application type).

Customer-Specific Key

In **ZC7-** and **ZC8-**series BasicCards from **REV D**, if **P1** is equal to *algorithm*+8 for a valid *algorithm*, and **P2** is equal to **&HFD**, **&HFE**, or **&HFF**, then encryption with a Customer-Specific Key is activated. See **9.10 Customer-Specific Encryption Keys** for details.

8.9.12 The END ENCRYPTION Command

END ENCRYPTION – End automatic encryption

Command syntax:

CLA	INS	P1	P2
C0	12	00	00

Response:

SW1	SW2
90	00

This command ends automatic encryption of command and response data fields.

Command-Specific Error Codes in SW1-SW2:

swNotEncrypting	Encryption is not currently enabled
swLcLeError	**Lc** or **Le** present
swInvalidState	Card is not in **TEST** or **RUN** state
swP1P2Error	**P1 <> 00** or **P2 <> 00**

To call **END ENCRYPTION** from a Terminal program:

```
#Include Commands.def
Call EndEncryption()
```

8.9.13 The ECHO Command

ECHO – Echo the command data

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	14	*increment*	00	*datalen*	*data*	*resplen*

Response:

ODATA	SW1	SW2
data+increment	90	00

This command simply adds *increment* to each byte in *data*, and returns *resplen* bytes. It is intended for testing communication and encryption (see **9.10**).

Note: The Enhanced BasicCards ignore *resplen*, always returning *datalen* bytes.

Command-Specific Error Codes in SW1-SW2:

swLcLeError **Lc** <> length of **IDATA** or **Le** not present
swP1P2Error **P2 <> 00**

To call **ECHO** from a Terminal program:

```
#Include Commands.def
Call Echo (P1=increment@, S$, Le=resplen@)
```

8.9.14 The FILE IO Command

FILE IO – Execute a file system operation

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**18**	*SysCode*	*filenum*	*CommandLen*	*CommandData*	*ResponseLen*

Response:

ODATA	SW1	SW2
status (1 byte) + *ResponseData*	**90**	**00**

This command is sent whenever the Terminal program attempts to access the file system in the BasicCard. The P-Code interpreter in the PC builds the command automatically, sends it to the BasicCard, and interprets the response. *SysCode* is the same as the *SysCode* parameter to the **SYSTEM** P-Code instruction – see **10.9.3 FILE SYSTEM Functions**. The *status* byte in the **ODATA** field is the **FileError** byte for the operation. The format of the *CommandData* and *ResponseData* fields depends on the value of *SysCode*, and is not described in this document.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc <>** length of **IDATA**, or **Le** absent
swP1P2Error	*SysCode* is not a valid file system operation

The **FILE IO** command was not designed to be called directly from a Terminal program. The P-Code interpreter calls it automatically when a file system operation is requested – see **Chapter 4: Files and Directories** for a description of the file system commands available in ZC-Basic.

8.9.15 The GET CHALLENGE Command

GET CHALLENGE – Get a cryptographic Challenge for **EXTERNAL AUTHENTICATE**

Command syntax:

CLA	INS	P1	P2	Le
C0	**40**	**00**	**00**	*ChallengeLen*

Response:

ODATA	SW1	SW2
Challenge	**90**	**00**

This command returns a random string of bytes as a Challenge for a subsequent **EXTERNAL AUTHENTICATE** command. If the Algorithm that will be used in the **EXTERNAL AUTHENTICATE** command is **AlgSingleDesCrc** or **AlgTripleDesCrc**, then *ChallengeLen* must be at least 8; if the Algorithm is **AlgAes128**, **AlgAes192**, or **AlgAes256**, then *ChallengeLen* must be at least 16. If **Le** is zero, or greater than 16, then 16 bytes are returned; this is valid for all Algorithms.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present or **Le** is absent
swP1P2Error	**P1P2 <> 0**

To call **GET CHALLENGE** from a Terminal program:

```
#Include Componnt.def
Call GetChallenge (Challenge$, Le=ChallengeLen@)
```

Note: If you need to avoid a name clash with the ISO **GET CHALLENGE** command, just define the constant **NoISONames**, at the start of the Terminal program (with **Const NoISONames = True**) or as a compiler option. You can still call the BasicCard command, using the name **ZCGetChallenge**.

8.9.16 The EXTERNAL AUTHENTICATE Command

EXTERNAL AUTHENTICATE – Authenticate the Terminal program to the BasicCard

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**42**	*KeyCID*	*n*+1	*Algorithm* (1 byte); *Response to Challenge*

Response:

SW1	SW2
90	**00**

The **EXTERNAL AUTHENTICATE** command is used to prove to the BasicCard that the Terminal program has access to a given Key. It does this by encrypting the Challenge returned by the **GET CHALLENGE** command, using the Algorithm's block encryption primitive.

Algorithm One of **AlgSingleDesCrc**, **AlgTripleDesCrc**, **AlgAes128**, **AlgAes192**, **AlgAes256**

n Block length of *Algorithm*: 8 bytes for **AlgSingleDesCrc** or **AlgTripleDesCrc**, and 16 bytes for **AlgAes128**, **AlgAes192**, or **AlgAes256**

EXTERNAL AUTHENTICATE must be the next command after **GET CHALLENGE**; any intervening command cancels the Challenge. If *Response to Challenge* is correct, the Access Condition **ExtAuth** (*KeyCID*) will be satisfied, until the next **EXTERNAL AUTHENTICATE** command is received or the card is reset.

The pre-defined variable **ExtAuthKeyCID** is set equal to *KeyCID* if this command is successful, or to zero if not.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is not equal to *n*+1, or **Le** is present
swDataNotFound	**GET CHALLENGE** was not the most recently received command, or the Challenge requested was less than *n* bytes
swKeyNotFound	A Key with the given *KeyCID* was not found
swKeyUsage	The Key's **Usage** attribute does not have **kuExtAuth** enabled
swKeyAlgorithm	The Key's **Algorithm** attribute does not have the given *Algorithm* enabled
swKeyTooShort	The Key is too short for the given *Algorithm*
swUnknownAlgorithm	*Algorithm* is not one of the five listed above
swBadAuthenticate	*Response to Challenge* is incorrect

To call **EXTERNAL AUTHENTICATE** from a Terminal program:

```
#Include Componnt.def
Call ExternalAuthenticate (P1P2=KeyCID%, Algorithm@, Response$)
```

Note: If you need to avoid a name clash with the ISO **EXTERNAL AUTHENTICATE** command, just define the constant **NoISONames**, at the start of the Terminal program (with **Const NoISONames = True**) or as a compiler option. You can still call the BasicCard command, using the name **ZCExternalAuthenticate**.

8.9.17 The INTERNAL AUTHENTICATE Command

INTERNAL AUTHENTICATE – Authenticate the BasicCard to the Terminal program

Command syntax:

CLA	INS	P1P2	Lc	IDATA	Le
C0	**44**	*KeyCID*	*n*+1	*Algorithm* (1 byte); *Challenge*	*n*

Response:

ODATA	SW1	SW2
Response to Challenge	**90**	**00**

The **INTERNAL AUTHENTICATE** command is used to prove to the Terminal program that the BasicCard has access to a given Key. It does this by encrypting *Challenge* using the Algorithm's block encryption primitive.

Algorithm One of **AlgSingleDesCrc**, **AlgTripleDesCrc**, **AlgAes128**, **AlgAes192**, **AlgAes256**

n Block length of *Algorithm*: 8 bytes for **AlgSingleDesCrc** or **AlgTripleDesCrc**, and 16 bytes for **AlgAes128**, **AlgAes192**, or **AlgAes256**

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is not equal to *n*+1, or **Le** is absent
swKeyNotFound	A Key with the given *KeyCID* was not found
swKeyUsage	The Key's **Usage** attribute does not have **kuIntAuth** enabled
swKeyAlgorithm	The Key's **Algorithm** attribute does not have the given *Algorithm* enabled
swKeyTooShort	The Key is too short for the given *Algorithm*
swUnknownAlgorithm	*Algorithm* is not one of the five listed above

To call **INTERNAL AUTHENTICATE** from a Terminal program:

```
#Include Componnt.def
Call InternalAuthenticate (P1P2=KeyCID%, Algorithm@, Challenge$)
Response$ = Chr$(Algorithm@) + Challenge$ ' Construct response
```

Note: If you need to avoid a name clash with the ISO **INTERNAL AUTHENTICATE** command, just define the constant **NoISONames**, at the start of the Terminal program (with **Const NoISONames = True**) or as a compiler option. You can still call the BasicCard command, using the name **ZCInternalAuthenticate**.

8.9.18 The VERIFY Command

VERIFY – Verify the user's password or PIN

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**46**	*KeyCID*	*n*	*Password*

Response:

SW1	SW2
90	**00**

The **VERIFY** command is used to prove to the BasicCard that the user knows a given password or PIN. The user types the password, which is sent unencrypted in the **IDATA** field (unless automatic encryption of Commands and Responses has been activated with the **START ENCRYPTION** command). If *Password* matches the data field of the given Key, the Access Condition **Verify** (*KeyCID*) will be satisfied, until the next **VERIFY** command is received or the card is reset.

The pre-defined variable **VerifyKeyCID** is set equal to *KeyCID* if this command is successful, or to zero if not.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Le** is present
swKeyNotFound	A Key with the given *KeyCID* was not found
swKeyUsage	The Key's **Usage** attribute does not have **kuVerify** enabled
swBadAuthenticate	*Password* is incorrect

To call **VERIFY** from a Terminal program:

```
#Include Componnt.def
Call Verify (P1P2=KeyCID%, Password$)
```

Note: If you need to avoid a name clash with the ISO **VERIFY** command, just define the constant **NoISONames**, at the start of the Terminal program (with **Const NoISONames = True**) or as a compiler option. You can still call the BasicCard command, using the name **ZCVerify**.

8.9.19 The GET FREE MEMORY Command

GET FREE MEMORY – Get the amount of free memory available in the global heap

Command syntax:

CLA	INS	P1	P2	Le
C0	**48**	**00**	**00**	**04**

Response:

ODATA	SW1	SW2
TotalFreeMemory (2 bytes); *LargestFreeBlock* (2 bytes)	**90**	**00**

The **GET FREE MEMORY** command returns the total free memory, and the size of the largest free block, in the card's global heap.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present, or **Le** is absent
swP1P2Error	**P1P2 <> 0**
swBadEepromHeap	The global heap is invalid. Contact ZeitControl for assistance

To call **GET FREE MEMORY** from a Terminal program:

```
#Include Componnt.def
Call GetFreeMemory (TotalFreeMemory%, LargestFreeBlock%)
```

8.9.20 The SELECT APPLICATION Command

SELECT APPLICATION – Select an Application

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	**A0**	**00**	**00**	*len*	*filename*

Response:

SW1	SW2
90	**00**

The **SELECT APPLICATION** command selects a File as the current Application.

To succeed, the caller must have **Execute** access to File *filename*. In addition, if there exists an ACR with the name "**Executable**" in the Root Directory, this ACR must be satisfied by File *filename* (not by the caller) for **Execute** access. In other words, when checking whether ACR "**Executable**" is satisfied, the three ACR types that depend on the current Application – **Privilege**, **Signed**, and **Application** – are evaluated as if *filename* were the current Application file.

If this call fails, the current Application remains unchanged.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is absent, or **Le** is present
swP1P2Error	**P1P2 <> 0**
swApplicationNotFound	File *filename* not found
swAppFileOpen	File *filename* is currently open for reading or writing
swExecutableAcrDenied	ACR "**\Executable**" exists, and is not satisfied by File *filename*
swAccessDenied	The caller does not have Execute access to File *filename*
swBadAppFile	*filename* is not a valid executable File
swSecureTransportActive	No Application may be selected during Secure Transport

To call **SELECT APPLICATION** from a Terminal program:

```
#Include Componnt.def
Call SelectApplication (filename$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **SelectApplication** is implemented as a System Library procedure, not as a Command definition.

8.9.21 The CREATE COMPONENT Command

CREATE COMPONENT – Create a Component

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**A2**	*type*	**00**	*len*	See below	**02**

Response:

ODATA	SW1	SW2
CID of new Component (2 bytes)	**90**	**00**

The **CREATE COMPONENT** command creates a Component in the BasicCard. It requires **Write** access to the Component's parent directory.

type the type of the Component: **ctFile**, **ctACR**, **ctPrivilege**, **ctFlag**, or **ctKey**.

IDATA *pathlen*, the length of *pathname* (1 byte);
pathname, the pathname of the Component (absolute, or relative to the current directory);
attrlen, the length of *attributes* (1 byte);
attributes, the Attributes field of the Component;
data, the Data field of the Component.

The length of the *data* field can be deduced from **Lc** (*datalen* = **Lc** – *pathlen* – *attrlen* – 2), so it is not required in **IDATA**. The format of the Attributes and Data fields depends on the Component type; a full description can be found in **5.9 Component Details**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is less than 2, or **Le** is absent
swP1P2Error	**P2 <> 0**
swBadComponentName	*pathlen* is invalid, or *pathname* is not a valid Component name
swBadComponentType	**P1** is not a valid Component type
swBadComponentAttr	*attrlen* is invalid, or *attributes* is invalid for the Component type
swBadComponentData	*data* is invalid for the Component type
swComponentAlreadyExists	A Component of type P1 with the given pathname already exists

To call **CREATE COMPONENT** from a Terminal program:

```
#Include Componnt.def
Call CreateComponent (type@, name$, attributes$, data$)
```

The format of the *attributes* field for each Component type is available as a user-defined structure type in **Componnt.def**. **5.9 Component Details** gives the details, and shows how to pass a user-defined structure in a **String** parameter.

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **CreateComponent** is implemented as a System Library procedure, not as a Command definition.

8.9.22 The DELETE COMPONENT Command

DELETE COMPONENT – Delete a Component

Command syntax:

CLA	INS	P1P2
C0	A4	*CID*

Response:

SW1	SW2
90	00

The **DELETE COMPONENT** command deletes a Component from the BasicCard. **Delete** access to the Component is required.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** is present
swComponentNotFound	No Component with the given *CID* exists
swLoadSequenceActive	No Component may be deleted during a Load Sequence – see **8.9.32 The LOAD SEQUENCE Command** for more information
swComponentReferenced	The Component is referenced by other Components, and may therefore not be deleted An ACR may be referenced as the **Lock** attribute of another Component; any Component type may be referenced as the parameter to an ACR. If a Privilege or Key is referenced only from the Rights List of a File, it will be automatically deleted from the Rights List, and will not generate this error. See **8.9.31 The READ RIGHTS LIST Command** for information on Rights Lists.

To call **DELETE COMPONENT** from a Terminal program:

```
#Include Componnt.def
Call DeleteComponent (CID%)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **DeleteComponent** is implemented as a System Library procedure, not as a Command definition.

8.9.23 The WRITE COMPONENT ATTR Command

WRITE COMPONENT ATTR – Write a Component's attributes

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**A6**	*CID*	*len*	*attributes*

Response:

SW1	SW2
90	**00**

The **WRITE COMPONENT ATTR** command writes the Attributes field of a Component. **Write** and **Delete** access to the Component are required. The format of a Component's Attributes field depends on the type of the Component; a full description can be found in **5.9 Component Details**.

This is the command to use if you want to change a Component's Access Control Rule. If the Component is a Flag or a Key, then it has other writable attributes; if you want to leave these attributes unchanged, you should read them with **READ COMPONENT ATTR**, change the **ACRCID%** field, and write them with **WRITE COMPONENT ATTR**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is absent or **Le** is present
swComponentNotFound	No Component with the given *CID* exists
swBadComponentAttr	The *attributes* field is invalid for the Component type

To call **WRITE COMPONENT ATTR** from a Terminal program:

```
#Include Componnt.def
Call WriteComponentAttr (CID%, attributes$)
```

The format of the *attributes* field for each Component type is available as a user-defined structure type in **Componnt.def**. **5.9 Component Details** gives the details, and shows how to pass a user-defined structure in a **String** parameter.

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **WriteComponentAttr** is implemented as a System Library procedure, not as a Command definition.

8.9.24 The READ COMPONENT ATTR Command

READ COMPONENT ATTR – Read a Component's attributes

Command syntax:

CLA	INS	P1P2	Le
C0	**A8**	*CID*	**00**

Response:

ODATA	SW1	SW2
attributes	**61**	*len*

The **READ COMPONENT ATTR** command reads the Attributes field of a Component. **Read** access is required to the Component's parent directory, but not to the Component itself. The format of a Component's Attributes field depends on the type of the Component; a full description can be found in **5.9 Component Details**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present or **Le** is absent
swComponentNotFound	No Component with the given *CID* exists

To call **READ COMPONENT ATTR** from a Terminal program:

```
#Include Componnt.def
Call ReadComponentAttr (CID%, attributes$)
```

The format of the *attributes* field for each Component type is available as a user-defined structure type in **Componnt.def**. **5.9 Component Details** gives the details, and shows how to pass a user-defined structure in a **String** parameter.

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **ReadComponentAttr** is implemented as a System Library procedure, not as a Command definition.

8.9.25 The WRITE COMPONENT DATA Command

WRITE COMPONENT DATA– Write a Component's data

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**AA**	*CID*	*len*	*data*

Response:

SW1	SW2
90	**00**

The **WRITE COMPONENT DATA** command writes the Data field of a Component. **Write** access to the Component is required. The format of a Component's Data field depends on the type of the Component; a full description can be found in **5.9 Component Details**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is absent or **Le** is present
swComponentNotFound	No Component with the given *CID* exists
swBadComponentData	The *data* field is invalid for the Component type

To call **WRITE COMPONENT DATA** from a Terminal program:

```
#Include Componnt.def
Call WriteComponentData (CID%, data$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **WriteComponentData** is implemented as a System Library procedure, not as a Command definition.

8.9.26 *The READ COMPONENT DATA Command*

READ COMPONENT DATA – Read a Component's data

Command syntax:

CLA	INS	P1P2	Le
C0	**AC**	*CID*	**00**

Response:

ODATA	SW1	SW2
data	**61**	*datalen*

The **READ COMPONENT DATA** command reads the Data field of a Component. **Read** access to the Component is required. The format of a Component's Data field depends on the type of the Component; a full description can be found in **5.9 Component Details**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError **Lc** is present or **Le** is absent
swComponentNotFound No Component with the given *CID* exists

To call **READ COMPONENT DATA** from a Terminal program:

```
#Include Componnt.def
Call ReadComponentData (CID%, data$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **ReadComponentData** is implemented as a System Library procedure, not as a Command definition.

8.9.27 The FIND COMPONENT Command

FIND COMPONENT – Get the CID of a Component from its name

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**AE**	*type*	**00**	*len*	*pathname*	**02**

Response:

ODATA	SW1	SW2
CID (2 bytes)	**90**	**00**

The **FIND COMPONENT** command finds the CID of a Component given its type and pathname. **Read** access is required to all directories in the path, but not to the Component itself.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** is absent
swP1P2Error	**P2 <> 0**
swBadComponentType	*type* is not a valid Component type
swBadComponentName	*pathname* is not a valid Component name
swComponentNotFound	No such Component exists

To call **FIND COMPONENT** from a Terminal program:

```
#Include Componnt.def
CID% = FindComponent (type@, name$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **FindComponent** is implemented as a System Library procedure, not as a Command definition.

8.9.28 The COMPONENT NAME Command

COMPONENT NAME – Get the name of a Component from its CID

Command syntax:

CLA	INS	P1P2	Le
C0	**B0**	*CID*	**00**

Response:

ODATA	SW1	SW2
pathname	**61**	*len*

The **COMPONENT NAME** command returns the full pathname of a Component given its CID. **Read** access is required to all directories in the path, but not to the Component itself.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is present, or **Le** is absent
swBadComponentType	The top four bits of *CID* do not form a valid Component type
swComponentNotFound	There is no Component with the gven CID

To call **COMPONENT NAME** from a Terminal program:

```
#Include Componnt.def
pathname$ = ComponentName (CID%)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **ComponentName** is implemented as a System Library procedure, not as a Command definition.

8.9.29 The GRANT PRIVILEGE Command

GRANT PRIVILEGE – Grant a Privilege to a File

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	**B2**	*PrivilegeCID*	*len*	*filename*

Response:

SW1	SW2
90	**00**

The **GRANT PRIVILEGE** command grants a Privilege to a File. This command requires **Grant** access to the Privilege, and **Write** access to the File. If the command is successful, *PrivilegeCID* is added to the File's Rights List; this causes the Access Condition **Privilege** (*PrivilegeCID*) to be satisfied whenever File is the currently selected Application.

If the **IDATA** field is empty, the command grants the Privilege to the Terminal program. The Terminal program is allowed to grant itself a Privilege in this way, as long as it has Grant access to the Privilege.

More precisely:

- If an operation is initiated from user code in an Application, then the Access Condition **Privilege** (*Privilege*) is satisfied if the Privilege is contained in the Rights List of the Application File.
- If an operation is initiated from the Terminal program, then the Access Condition **Privilege** (*Privilege*) is satisfied if the Privilege has been granted to the Terminal program since the card was last reset. (But the card only remembers the three most recent such Privileges.)

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is absent, or **Le** is present
swBadComponentType	*PrivilegeCID* is not a valid CID for a Component of type Privilege
swBadComponentName	*filename* is not a valid filename
swComponentNotFound	Either the Privilege or the File does not exist

To call **GRANT PRIVILEGE** from a Terminal program:

```
#Include Componnt.def
Call GrantPrivilege (PrivilegeCID%, filename$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **GrantPrivilege** is implemented as a System Library procedure, not as a Command definition.

8.9.30 The AUTHENTICATE FILE Command

AUTHENTICATE FILE – Authenticate a File with a Signature

Command syntax:

CLA	INS	P1P2	Lc	IDATA
C0	B4	*KeyCID*	*len*	See below

Response:

SW1	SW2
90	00

The **AUTHENTICATE FILE** command authenticates a File with an Elliptic Curve signature or a Message Authentication Code (MAC). It requires **Read** access to the File.

KeyCID The CID of the Key used to verify the signature or authenticate the MAC

IDATA *algorithm*, the cryptographic algorithm used to sign or authenticate the File
namelen, the length of *filename*
filename, the path name of the File
signature, the signature or MAC

Valid algorithms are **AlgEC167**, **AlgOmacAes128**, **AlgOmacAes192**, and **AlgOmacAes256**.

- If *algorithm* is equal to **AlgEC167**, then *KeyCID* is the CID of an Elliptic Curve Public Key; *signature* is a 42-byte digital signature of the contents of the File, computed using the corresponding Private Key, as if by the **EC167** System Library procedure **EC167HashAndSign** (see **7.3.5 Generating a Digital Signature** for details).
- If *algorithm* is equal to **AlgOmacAes128**, **AlgOmacAes192**, or **AlgOmacAes256**, then *signature* is the 16-byte MAC of the contents of the File, computed with the **OMAC** algorithm, as if by the System Library procedure **OMAC** (see **7.10 The OMAC Library**).

If the command is successful, *KeyCID* is added to the File's Rights List; this causes the Access Condition **Signed** (*KeyCID*) to be satisfied whenever File is the currently selected Application.

For another method of authenticating a File, see **5.6.2 Automatic File Authentication**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** is absent, or **Le** is present
swBadComponentType	*KeyCID* is not a valid CID for a Component of type Key
swBadComponentName	*filename* is not a valid filename
swKeyNotFound	The Key does not exist
swComponentNotFound	The File does not exist
swKeyUsage	The Key's **Usage** attribute does not have **kuSign** enabled
swKeyAlgorithm	The Key's **Algorithm** attribute does not have *algorithm* enabled
swKeyTooShort	The Key is too short for the given *algorithm*
swUnknownAlgorithm	*algorithm* is not one of the four listed above
swBadSignature	The signature or MAC is incorrect

To call **AUTHENTICATE FILE** from a Terminal program:

```
#Include Componnt.def
Call AuthenticateFile (KeyCID%, algorithm@, filename$, signature$)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **AuthenticateFile** is implemented as a System Library procedure, not as a Command definition.

8.9.31 The READ RIGHTS LIST Command

READ RIGHTS LIST – Read the Privileges and Signatures of a File

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**B6**	*start*	**00**	*len*	*filename*	2**nmax*

Response:

ODATA	SW1	SW2
RightsList%(*start*) to *RightsList%*(*start+n–1*)	**61**	2**n*

The **READ RIGHTS LIST** command returns the Rights List of a File. This list contains the CID of every Privilege that has been granted to the File (with the **GRANT PRIVILEGE** command or the **GrantPrivilege** System Library procedure), and every Key that has been used to authenticate the File (with the **AUTHENTICATE FILE** command or the **AuthenticateFile** System Library procedure). The Rights List is used by the MultiApplication BasicCard operating system to evaluate the Access Conditions **Privilege** (*PrivilegeCID*) and **Signed** (*KeyCID*). **Read** access is required to every directory on the path, but not to the File itself.

In principle, a File can have a Rights List with more than 127 entries; such a list is too long to be returned in the **ODATA** field. In this case, you can request the Rights List entries *RightsList%*(*start*) to *RightsList%*(*start+nmax–1*) by setting **P1** and **Le** accordingly; if **Le** is zero, *nmax* is taken to be 127. (Here the *RightsList%*() array is taken to be zero-based.)

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** is absent, or **Le** is odd
swP1P2Error	**P2 <> 0**
swBadComponentName	*filename* is not a valid filename
swComponentNotFound	File *filename* does not exist
swDataNotFound	The Rights List contains at most *start* entries, so there is no data to return

To call **READ RIGHTS LIST** from a Terminal program:

```
#Include Componnt.def
nRights% = ReadRightsList (filename$, RightsList%())
```

The **ReadRightsList** System Library procedure automatically handles the case where the number of Rights List entries is greater than 127.

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **ReadRightsList** is implemented as a System Library procedure, not as a Command definition.

8.9.32 The LOAD SEQUENCE Command

LOAD SEQUENCE – Start, end, or abort a Load Sequence session

Command syntax:

CLA	INS	P1	P2
C0	B8	*phase*	00

Response:

SW1	SW2
90	00

The **LOAD SEQUENCE** command implements a form of data commitment for use during Application loading. Sometimes the Application Loader will fail before loading is complete – for instance, the card may lose power during loading, or it may have insufficient memory to create all the required Components. In this case, none of the Application's Components that were created before the error occurred will be required. This command provides a simple method to ensure that these unwanted Components are automatically deleted.

The *phase* parameter must be **LoadSequenceStart**, **LoadSequenceEnd**, or **LoadSequenceAbort**. These constants are defined in **Componnt.def**. They are used as follows:

- Before the Application Loader starts to load an Application, it calls **LOAD SEQUENCE** with *phase*=**LoadSequenceStart**. After this, all newly created Components will be flagged as Uncommitted.
- If the Application loads successfully, the Application Loader calls **LOAD SEQUENCE** with *phase*=**LoadSequenceEnd**; these new Components will then be flagged as Permanent.
- If the Application fails to load for any reason, the Application Loader calls **LOAD SEQUENCE** with *phase*=**LoadSequenceAbort**; this tells the BasicCard to delete all Components that are flagged as Uncommitted. If the Application Loader can't do this, because the card is no longer responsive (or because the Application Loader itself lost power), then the next time the card is reset, it will delete these Components automatically.

Components cannot be deleted while a Load Sequence is active; an attempt to delete a Component will result in the error code **SW1-SW2=swLoadSequenceActive**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** or **Le** is present
swP1P2Error	**P2 <> 0**
swLoadSequencePhase	**P1** is not one ot the three values listed above
swLoadSequenceActive	*phase*= **LoadSequenceStart**, but Load Sequence is already active
swLoadSequenceNotActive	*phase*=**LoadSequenceEnd** or **LoadSequenceAbort**, but no Load Sequence is active

To call **LOAD SEQUENCE** from a Terminal program:

```
#Include Componnt.def
Call LoadSequence (phase@)
```

Note: For compatibility with the **COMPONENT** System Library in the MultiApplication BasicCard, **LoadSequence** is implemented as a System Library procedure, not as a Command definition.

8.9.33 The SECURE TRANSPORT Command

SECURE TRANSPORT – Start or end a Secure Transport session

Start session:

CLA	INS	P1P2	Lc	IDATA
C0	BA	*KeyCID*	*len*	*algorithm* (1 byte); *Nonce*

Response:

SW1	SW2
90	00

End session:

CLA	INS	P1	P2
C0	BA	00	00

Response:

SW1	SW2
90	00

The **SECURE TRANSPORT** command enables Files and Keys to be stored in an Image file in encrypted form, using a Secure Transport Key known only to the issuer and the BasicCard. While Secure Transport is active, the Access Condition **SecTrans** (*KeyCID*) is satisfied. See **5.6 Secure Transport** for an explanation of the Secure Transport mechanism.

Valid algorithms: **AlgEaxAes128**, **AlgEaxAes192**, **AlgEaxAes256**.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	**Lc** absent (**P1P2 <> 0**), or **Lc** present (**P1P2 = 0**), or **Le** present
swKeyNotFound	The Key does not exist
swKeyUsage	The Key's **Usage** attribute does not have **kuSecTrans** enabled
swKeyAlgorithm	The Key's **Algorithm** attribute does not have *algorithm* enabled
swKeyTooShort	The Key is too short for the given *algorithm*
swUnknownAlgorithm	*algorithm* is not one of those listed above
swSecureTransportActive	Attempt to start Secure Transport while already active
swSecureTransportInactive	Attempt to end a non-existent Secure Transport session

To start **SECURE TRANSPORT** from a Terminal program:

```
#Include Componnt.def
Call SecureTransport (P1P2=KeyCID%, algorithm@, Nonce$)
```

To end **SECURE TRANSPORT** from a Terminal program:

```
#Include Componnt.def
Call SecureTransport (Lc=0, 0, "")
```

8.10 The Command Definition File Commands.def

The file **Commands.def** can be found in the directory `BasicCardV8\Inc`. It contains:

- declarations of all the pre-defined commands;
- definitions of the ZC-Basic **SW1-SW2** status codes;
- definitions of P-Code error codes; and
- interface procedures tothe **START ENCRYPTION** command.

See **8.8 Status Bytes SW1 and SW2** for descriptions of the status and error codes.

Here is the file **Commands.def**:

```
Rem Pre-defined BasicCard commands

#IfNotDef CommandsDefIncluded ' Prevent multiple inclusion
Const CommandsDefIncluded = True

#Include AlgID.DEF

Declare Command &HC0 &H00 GetState(Lc=0, State@, Version$)
Declare Command &HC0 &H02 EepromSize(Lc=0, Start%, Length%)
Declare Command &HC0 &H04 ClearEeprom(Length%, Disable Le)

Rem  Since Version 3.01, the WRITE EEPROM command is no longer supported.
Rem  Use it at your own risk!
Rem
Rem  Declare Command &HC0 &H06 WriteEeprom(Data$, Disable Le)

Declare Command &HC0 &H08 ReadEeprom(Lc=0, Data$)
Declare Command &HC0 &H0A EepromCRC(Length%)
Declare Command &HC0 &H0A EepromCRC32(Lc=2, CRCHi%, CRCLo%, Le=4)
Declare Command &HC0 &H0C SetState()
Declare Command &HC0 &H0E GetApplicationID(Lc=0, Name$)
Declare Command &HC0 &H10 StartEncryption(RA&, Le=0)
Declare Command &HC0 &H10 StartEncryptionWithKDP(RA&, ReadOnly KDP$, Le=0)
Declare Command &HC0 &H10 ProEncryption(RAHi&, RALo&, Le=0)
Declare Command &HC0 &H10 ProEncryptionWithKDP(RAHi&, RALo&, ReadOnly KDP$, Le=0)
Declare Command &HC0 &H10 SMEncryption(Algorithm@, RAHi&, RALo&, Le=0)
Declare Command &HC0 &H10 SMAuthentication(Algorithm@, Le=0)
Declare Command &HC0 &H12 EndEncryption()
Declare Command &HC0 &H14 Echo(S$)
Declare Command &HC0 &H16 AssignNAD()

Rem  Commands for BasicCards with 24-bit addresses

Declare Command &HC0 &H02 Eeprom24Size(Lc=0, Start As String*3,_
  Length As String*3)
Declare Command &HC0 &H04 ClearEeprom24(Start As String*3, Length As String*3,_
  Disable Le)
Declare Command &HC0 &H08 ReadEeprom24(Lc=3, ReadOnly Address As String*3, Data$)
Declare Command &HC0 &H0A Eeprom24CRC(Lc=6, ReadOnly Address As String*3,_
  ReadOnly Length As String*3, CRC%, Le=8)
Declare Command &HC0 &H0A Eeprom24CRC32(Lc=6, ReadOnly Address As String*3,_
  ReadOnly Length As String*3, CRC&, Le=10)

Rem BasicCard operating system errors

Const swCommandOK              = &H9000
Const swRetriesRemaining       = &H63C0
Const swEepromWriteError       = &H6581
Const swBadEepromHeap          = &H6582
Const swBadFileChain           = &H6583
Const swKeyNotFound            = &H6611
```

```
Const swPolyNotFound           = &H6612
Const swKeyTooShort            = &H6613
Const swKeyDisabled            = &H6614
Const swUnknownAlgorithm       = &H6615
Const swAlreadyEncrypting      = &H66C0
Const swNotEncrypting          = &H66C1
Const swBadCommandCRC          = &H66C2
Const swDesCheckError          = &H66C3
Const swCoprocessorError       = &H66C4
Const swAesCheckError          = &H66C5
Const swBadSignature           = &H66C6
Const swBadAuthenticate        = &H66C7
Const swLcLeError              = &H6700
Const swCommandTooLong         = &H6781
Const swResponseTooLong        = &H6782
Const swInvalidState           = &H6985
Const swCardUnconfigured       = &H6986
Const swNewStateError          = &H6987
Const swSMError                = &H6988
Const swP1P2Error              = &H6A00
Const swOutsideEeprom          = &H6A02
Const swDataNotFound           = &H6A88
Const swINSNotFound            = &H6D00
Const swCLANotFound            = &H6E00
Const swInternalError          = &H6F00

Rem SW1=&H61 is Le warning:

Const sw1LeWarning             = &H61

Rem SW1=&H6C is La warning (T=0 protocol only):

Const sw1LaWarning             = &H6C

Rem P-Code interpreter errors (SW1=&H64, SW2=P-Code error)

Const sw1PCodeError            = &H64

Const pcStackOverflow          = &H01
Const pcDivideByZero           = &H02
Const pcNotImplemented         = &H03
Const pcBadRamHeap             = &H04
Const pcBadEepromHeap          = &H05
Const pcReturnWithoutGoSub     = &H06
Const pcBadSubscript           = &H07
Const pcBadBounds              = &H08
Const pcInvalidReal            = &H09
Const pcOverflow               = &H0A
Const pcNegativeSqrt           = &H0B
Const pcDimensionError         = &H0C
Const pcBadStringCall          = &H0D
Const pcOutOfMemory            = &H0E
Const pcArrayNotDynamic        = &H0F
Const pcArrayTooBig            = &H10
Const pcDeletedArray           = &H11
Const pcPCodeDisabled          = &H12
Const pcBadSystemCall          = &H13
Const pcBadKey                 = &H14
Const pcBadLibraryCall         = &H15
Const pcStackUnderflow         = &H16
Const pcInvalidAddress         = &H17

Rem Error codes generated by the Terminal
```

```
Const swNoCardReader           = &H6790
Const swCardReaderError        = &H6791
Const swNoCardInReader         = &H6792
Const swCardPulled             = &H6793
Const swT1Error                = &H6794
Const swCardError              = &H6795
Const swCardNotReset           = &H6796
Const swKeyNotLoaded           = &H6797
Const swPolyNotLoaded          = &H6798
Const swBadResponseCRC         = &H6799
Const swCardTimedOut           = &H679A
Const swTermOutOfMemory        = &H679B
Const swBadDesResponse         = &H679C
Const swInvalidComPort         = &H679D
Const swNoPcscDriver           = &H679F
Const swPcscReaderBusy         = &H67A0
Const swPcscError              = &H67A1
Const swComPortBusy            = &H67A2
Const swBadATR                 = &H67A3
Const swT0Error                = &H67A4
Const swPTSError               = &H67A7
Const swDataOverrun            = &H67A8
Const swBadAesResponse         = &H67A9
Const swZCMDCardObsolete       = &H67AA
Const swZCMDTermObsolete       = &H67AB
Const swCommandTooShort        = &H67C0
Const swCommandFormat          = &H67C1
Const swResponseTooShort       = &H67C2
Const swUnexpectedResponse     = &H67C3
Const swInvalidSetState        = &H67C4
Const swTerminalProgramRunning = &H67C5
Const swAppLoadFailure         = &H67C6
Const swReservedINS            = &H6D80
Const swReservedCLA            = &H6E80

Rem  MuliApplication BasicCard errors
Rem  (corresponding to Component Library errors in COMPONNT.DEF)

Const swBadComponentName         = &H69C0
Const swComponentNotFound        = &H69C1
Const swAccessDenied             = &H69C2
Const swComponentAlreadyExists   = &H69C3
Const swBadComponentChain        = &H69C4
Const swNameTooLong              = &H69C5
Const swOutOfMemory              = &H69C6
Const swInvalidACR               = &H69C7
Const swBadComponentType         = &H69C8
'Const swKeyNotFound              = &H69CC swKeyNotFound already exists
Const swKeyUsage                 = &H69CD
Const swKeyAlgorithm             = &H69CE
'Const swKeyDisabled              = &H69CF swKeyDisabled already exists
Const swTooManyTempFlags         = &H69D0
Const swExecutableAcrDenied      = &H69D1
Const swApplicationNotFound      = &H69D2
Const swACRDepth                 = &H69D3
Const swBadComponentAttr         = &H69D4
Const swBadComponentData         = &H69D5
Const swBadAppFile               = &H69D6
Const swLoadSequenceActive       = &H69D7
```

```
Const swLoadSequenceNotActive    = &H69D8
Const swLoadSequencePhase        = &H69D9
'Const swKeyTooShort              = &H69DA swKeyTooShort already exists
'Const swUnknownAlgorithm         = &H69DB swUnknownAlgorithm already exists
Const swBadEaxTag                = &H69DC
Const swSecureTransportActive    = &H69DD
Const swSecureTransportInactive  = &H69DE
Const swComponentReferenced      = &H69DF
Const swFileNotContiguous        = &H69E0
Const swAppFileOpen              = &H69E1

#IfDef TerminalProgram

Rem  AutoEncryption handles StartEncryption for the different card types.
Rem  To use:
Rem          Call AutoEncryption (KeyNum@, KeyName$)
Rem          Call CheckSW1SW2()
Rem
Rem  KeyNum@ is the key number, for all card types. Encrypting for the
Rem  MultiApplication BasicCard also requires the key's path name, in KeyName$.

#Include MISC.DEF
#Include COMPONNT.DEF

Sub AutoEncryption (KeyNum@, KeyName$)

  Private TryAES : TryAES = (Len (Key(KeyNum@)) >= 16)
  If TryAES Then
    Call ProEncryption (P2=KeyNum@, Rnd, Rnd)
    If SW1SW2 = swLcLeError Then Call StartEncryption (P2=KeyNum@, Rnd)
  Else
    Call StartEncryption (P2=KeyNum@, Rnd)
  End If

  Select Case SW1SW2

    Case swUnknownAlgorithm ' Compact BasicCard doesn't support P1=0
      Call StartEncryption (P1=&H12, P2=KeyNum@, Rnd)

    Case swBadComponentType ' MultiApplication BasicCard
      Private CID : CID = FindComponent (ctKey, KeyName$)
      Call AddIndexedKey (CID, Key(KeyNum@))
      If TryAES Then
        Call SMEncryption (P1P2=CID, 0, Rnd, Rnd)
      Else
        Call SMEncryption (P1P2=CID, Lc=5, 0, Rnd, 0)
      End If

  End Select

End Sub

Rem  Sub SMEncryptionByCID (KeyCID%, KeyVal$, Algorithm@)
Rem  Sub SMEncryptionByName (KeyName$, KeyVal$, Algorithm@)
Rem
Rem  These procedures activate encryption in the MultiApplication BasicCard.
Rem  SMEncryptionByName is simpler; SMEncryptionByCID is faster, saving
Rem  a call to FincComponent.

Sub SMEncryptionByCID (KeyCID%, KeyVal$, Algorithm@)
```

```
  Rem  Tell the Terminal program interpreter the value of the key
  Call AddIndexedKey (KeyCID%, KeyVal$)

  If Algorithm@ < AlgOmacAes128 Then

    Rem  Encryption algorithm - initialisation data required

    If Algorithm@ <= AlgTripleDesCrc Then ' Four-byte initialisation data
      Call SMEncryption (P1P2=KeyCID%, Lc=5, Algorithm@, Rnd, 0)
    Else                                  ' Eight-byte initialisation data
      Call SMEncryption (P1P2=KeyCID%, Algorithm@, Rnd, Rnd)
    End If

  Else

    Rem  Authentication algorithm - no initialisation data required
    Call SMEncryption (P1P2=KeyCID%, Lc=1, Algorithm@, 0, 0)

  End If

End Sub

Sub SMEncryptionByName (KeyName$, KeyVal$, Algorithm@)
  Private CID : CID = FindComponent (ctKey, KeyName$)
  If SW1SW2 = swCommandOK Then _
    Call SMEncryptionByCID (CID, KeyVal$, Algorithm@)
End Sub

#EndIf ' TerminalProgram

#EndIf ' CommandsDefIncluded
Const swFileNotContiguous        = &H69E0
Const swAppFileOpen              = &H69E1

#IfDef TerminalProgram

Rem  AutoEncryption handles StartEncryption for the different card types.
Rem  To use:
Rem          Call AutoEncryption (KeyNum@, KeyName$)
Rem          Call CheckSW1SW2()
Rem
Rem  KeyNum@ is the key number, for all card types. Encrypting for the
Rem  MultiApplication BasicCard also requires the key's path name, in KeyName$.

#Include MISC.DEF
#Include COMPONNT.DEF

Sub AutoEncryption (KeyNum@, KeyName$)

  Private TryAES : TryAES = (Len (Key(KeyNum@)) >= 16)
  If TryAES Then
    Call ProEncryption (P2=KeyNum@, Rnd, Rnd)
    If SW1SW2 = swLcLeError Then Call StartEncryption (P2=KeyNum@, Rnd)
  Else
    Call StartEncryption (P2=KeyNum@, Rnd)
  End If

  Select Case SW1SW2
```

```
    Case swUnknownAlgorithm ' Compact BasicCard doesn't support P1=0
      Call StartEncryption (P1=&H12, P2=KeyNum@, Rnd)

    Case swBadComponentType ' MultiApplication BasicCard
      Private CID : CID = FindComponent (ctKey, KeyName$)
      Call AddIndexedKey (CID, Key(KeyNum@))
      If TryAES Then
        Call SMEncryption (P1P2=CID, 0, Rnd, Rnd)
      Else
        Call SMEncryption (P1P2=CID, Lc=5, 0, Rnd, 0)
      End If

  End Select

End Sub

Rem  Sub SMEncryptionByCID (KeyCID%, KeyVal$, Algorithm@)
Rem  Sub SMEncryptionByName (KeyName$, KeyVal$, Algorithm@)
Rem
Rem  These procedures activate encryption in the MultiApplication BasicCard.
Rem  SMEncryptionByName is simpler; SMEncryptionByCID is faster, saving
Rem  a call to FincComponent.

Sub SMEncryptionByCID (KeyCID%, KeyVal$, Algorithm@)

  Rem  Tell the Terminal program interpreter the value of the key
  Call AddIndexedKey (KeyCID%, KeyVal$)

  If Algorithm@ < AlgOmacAes128 Then

    Rem  Encryption algorithm - initialisation data required

    If Algorithm@ <= AlgTripleDesCrc Then ' Four-byte initialisation data
      Call SMEncryption (P1P2=KeyCID%, Lc=5, Algorithm@, Rnd, 0)
    Else                               ' Eight-byte initialisation data
      Call SMEncryption (P1P2=KeyCID%, Algorithm@, Rnd, Rnd)
    End If

  Else

    Rem  Authentication algorithm - no initialisation data required
    Call SMEncryption (P1P2=KeyCID%, Lc=1, Algorithm@, 0, 0)

  End If

End Sub

Sub SMEncryptionByName (KeyName$, KeyVal$, Algorithm@)
  Private CID : CID = FindComponent (ctKey, KeyName$)
  If SW1SW2 = swCommandOK Then _
    Call SMEncryptionByCID (CID, KeyVal$, Algorithm@)
End Sub

#EndIf ' TerminalProgram

#EndIf ' CommandsDefIncluded
```

9. Encryption Algorithms

The Enhanced BasicCard supports the following two encryption algorithms:

Algorithm	
AlgSingleDes	**Single DES** (Data Encryption Standard, 8-byte key)
AlgTripleDes	**Triple DES-EDE2** (Data Encryption Standard, 16-byte key)

The Professional and MultiApplication BasicCards support some or all of the following encryption algorithms:

Algorithm	
AlgSingleDesCrc	**Single DES** with CRC (8-byte key)
AlgTripleDesEDE2Crc	**Triple DES-EDE2** with CRC (16-byte key)
AlgTripleDesEDE3Crc	**Triple DES-EDE3** with CRC (24-byte key)
AlgAes128	**AES-128** (Advanced Encryption Standard, 128-bit key)
AlgAes192	**AES-192** (Advanced Encryption Standard, 192-bit key)
AlgAes256	**AES-256** (Advanced Encryption Standard, 256-bit key)
AlgEaxAes128	**EAX** (Authenticated Encryption) using **AES-128**
AlgEaxAes192	**EAX** (Authenticated Encryption) using **AES-192**
AlgEaxAes256	**EAX** (Authenticated Encryption) using **AES-256**
AlgOmacAes128	**OMAC** (One-Key CBC MAC) using **AES-128**
AlgOmacAes192	**OMAC** (One-Key CBC MAC) using **AES-192**
AlgOmacAes256	**OMAC** (One-Key CBC MAC) using **AES-256**

This chapter describes these algorithms in detail, to give interested readers the opportunity to evaluate them. But you don't need to know how these algorithms work in order to use them; if you only want to know how to use them from ZC-Basic, see instead **3.18.1 Implementing Encryption**.

9.1 The DES Algorithm

The **DES** algorithm is the internationally recognised Data Encryption Standard, defined in the ANSI standard documents *X3.92-1981 (Data Encryption Algorithm)* and *X3.106-1983 (Data Encryption Algorithm – Modes of Operation)*. See these documents for a definition of the **DES** algorithm itself; for a fuller treatment, including 'C' source code, see Bruce Schneier's *Applied Cryptography* (Second Edition, John Wiley & Sons, Inc., 1996).

As you can see from the dates of the ANSI documents, the **DES** algorithm is no longer young. In fact, the original **DES** algorithm is usually referred to as **Single DES**, and must now be regarded as less than completely secure. Special-purpose hardware can be constructed for several tens of thousands of dollars, that can break **Single DES** encryption in less than a day. For this reason, stronger versions, **Triple DES-EDE2** and **Triple DES-EDE3**, have become *de facto* standards in the banking world. **Triple DES-EDE2** is generally believed to be safe against all currently feasible attacks, and **Triple DES-EDE3** is believed to be safe against any imaginable future attacks. However, **Single DES** is still used for protecting confidential but financially worthless data, such as a patient's medical records.

The original ANSI *X3.92* document defines **DES** as an encryption function that takes a 56-bit, 8-byte key **K** and an 8-byte data block **P** as input, and returns an 8-byte data block **C** as output:

$$\mathbf{C} = \mathbf{E}_{\mathbf{K}}(\mathbf{P})$$

The inverse of this is the **DES** decryption function:

$$\mathbf{P} = \mathbf{D}_{\mathbf{K}}(\mathbf{C})$$

(This notation is taken from Bruce Schneier's *Applied Cryptography*: **P** and **C** denote plaintext and ciphertext, **E** and **D** are encryption and decryption, and **K** is the key.)

Note that an 8-byte **Single DES** key contains only 56 significant bits. This is because the top bit of each byte is not used. This bit can be used as a parity check, or simply ignored (which is what the BasicCard does).

The **Triple DES-EDE2** algorithm takes a 112-bit, 16-byte key **K** and splits it into two 8-byte keys **KL** and **KR**. Then the encryption and decryption functions are given by

$$\mathbf{C = EDE2_K (P) = E_{KL} (D_{KR} (E_{KL} (P)))} \text{ and}$$

$$\mathbf{P = DED2_K (C) = D_{KL} (E_{KR} (D_{KL} (C)))}$$

The **Triple DES-EDE3** algorithm takes a 168-bit, 24-byte key **K** and splits it into three 8-byte keys **K1**, **K2**, and **K3**. Then the encryption and decryption functions are given by

$$\mathbf{C = EDE3_K (P) = E_{K3} (D_{K2} (E_{K1} (P)))} \text{ and}$$

$$\mathbf{P = DED3_K (C) = D_{K1} (E_{K2} (D_{K3} (C)))}$$

(The six functions $\mathbf{E_K}$, $\mathbf{D_K}$, $\mathbf{EDE2_K}$, $\mathbf{DED2_K}$, $\mathbf{EDE3_K}$, and $\mathbf{DED3_K}$ can be called directly from ZC-Basic – see **3.18.6 DES Encryption Primitives**.)

Given such encryption and decryption functions, there are several ways that they can be used to encrypt and decrypt a message of arbitrary length. The method used by the Enhanced BasicCard is described in the next section.

9.2 Implementation of DES in the BasicCard

Apart from their encryption and decryption functions (**E** and **D** versus $\mathbf{E^3}$ and $\mathbf{D^3}$), the implementations of **Single DES**, **Triple DES-EDE2**, and **Triple DES-EDE3** in the BasicCard are identical. To start with, we need to know how to encrypt a message that is longer than 8 bytes. (All commands and responses encrypted with **DES** in the BasicCard are at least 8 bytes long.)

9.2.1 The Message Encryption Functions $\mathbf{ME_K}$, $\mathbf{MEDE2_K}$, and $\mathbf{MEDE3_K}$

The **Single DES** message encryption function $\mathbf{C = ME_K (P)}$ is defined as follows. We are given:

- a message **P**, at least 8 bytes in length;
- an 8-byte key **K**;
- the **Single DES** encryption and decryption functions $\mathbf{E_K}$ and $\mathbf{D_K}$;
- an 8-byte *initialisation vector* $\mathbf{C_0}$ (more about this in **9.2.3 The Initialisation Vector**).

First, split the message **P** into 8-byte blocks $\mathbf{P_1}$, $\mathbf{P_2}$,..., $\mathbf{P_{n-1}}$, plus a final block $\mathbf{P_n}$ that may be shorter than 8 bytes. Pad this final block with m zeroes to a length of 8 bytes (so $0 \le m \le 7$). Then compute, for $1 \le i \le n$:

$$\mathbf{C_i = E_K (C_{i-1} \text{ Xor } P_i)}$$

(Note that the initialisation vector $\mathbf{C_0}$ is needed to compute $\mathbf{C_1}$.) Then throw away the last m bytes of the *penultimate* block $\mathbf{C_{n-1}}$, and concatenate the resulting blocks $\mathbf{C_1}$,..., $\mathbf{C_n}$ to get the encrypted ciphertext **C**.

If we threw away the last m bytes of the *last* block $\mathbf{C_n}$, then the message **C** couldn't be decrypted by its recipient. But the recipient can reconstruct the last m bytes of $\mathbf{C_{n-1}}$, as follows:

The last block is computed from	$\mathbf{C_n = E_K (C_{n-1} \text{ Xor } P_n)}$
Therefore,	$\mathbf{D_K (C_n) = C_{n-1} \text{ Xor } P_n}$
which means that	$\mathbf{C_{n-1} = D_K (C_n) \text{ Xor } P_n}$

But the last m bytes of $\mathbf{P}_n$ are all zero, so the last m bytes of $\mathbf{C}_{n-1}$ are equal to the last m bytes of $\mathbf{D}_K(\mathbf{C}_n)$, which can be computed without prior knowledge of the plaintext **P**. This trick is called *ciphertext stealing*, and it allows us to keep encrypted messages to their original size.

The **Triple DES** message encryption functions $\mathbf{MEDE2}_K$ and $\mathbf{MEDE3}_K$ are defined in exactly the same way, except that the key **K** is 16 or 24 bytes long, and the **Triple DES** encryption function $\mathbf{EDE2}_K$ or $\mathbf{EDE3}_K$ is substituted for the **Single DES** function $\mathbf{E}_K$.

9.2.2 The Message Decryption Functions MD_K, $MDED2_K$, and $MDED3_K$

The **Single DES** message decryption function $\mathbf{P} = \mathbf{MD}_K(\mathbf{C})$ is the inverse of $\mathbf{ME}_K$. First restore the penultimate block $\mathbf{C}_{n-1}$ to 8 bytes, as described in the previous section. Then compute, for $1 \le i \le n$:

$$\mathbf{P}_i = \mathbf{C}_{i-1} \ \mathbf{Xor} \ \mathbf{D}_K(\mathbf{C}_i)$$

Throw away the last m bytes in $\mathbf{P}_n$ (which should all be zero), and concatenate all the resulting blocks $\mathbf{P}_1$,..., $\mathbf{P}_n$ to get the original plaintext message **P**.

The **Triple DES** message decryption functions $\mathbf{MDED2}_K$ and $\mathbf{MDED3}_K$ are defined in exactly the same way, except that the **Triple DES** decryption function $\mathbf{DED2}_K$ or $\mathbf{DED3}_K$ is substituted for the **Single DES** function $\mathbf{D}_K$.

9.2.3 The Initialisation Vector

The initialisation vector $\mathbf{C}_0$ is determined as follows:

For the first command following a **START ENCRYPTION** command, the initialisation vector $\mathbf{C}_0$ depends on the command and response fields of the **START ENCRYPTION** command:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**10**	*algorithm*	*key*	**04**	*Random number* **RA** (4 bytes)	**04**

Response:

ODATA	SW1	SW2
Random number **RB** (4 bytes)	**90**	**00**

In this case, $\mathbf{C}_0$ consists of the first two bytes of **RA**, followed by all four bytes of **RB**, followed by the last two bytes of **RA**.

For subsequent commands and responses, $\mathbf{C}_0$ is simply the last ciphertext block $\mathbf{C}_n$ of the previous message.

9.2.4 Encryption of Commands in the Enhanced BasicCard

A command has the following structure (shaded blocks are optional):

CLA	INS	P1	P2	Lc	IDATA	Le

Encryption consists of the following steps:

- If the **Lc** or **Le** fields are absent, insert **Lc' = 00** and/or **Le'** = **00**:

CLA	INS	P1	P2	Lc'	IDATA	Le'

- Append two zeroes (the resulting command now contains at least 8 bytes):

P =

CLA	INS	P1	P2	Lc'	IDATA	Le'	00	00

- Encrypt the whole command **P**, with **C = ME**$_K$ (**P**) or **C = MEDE2**$_K$ (**P**):

C

- Wrap the resulting ciphertext **C** in the original command parameters:

CLA	**INS**	**P1**	**P2**	**Lc' + 8**	**C**	**Le**

The resulting command is 8 bytes longer than the original command. These 8 bytes of redundancy enable an authentication check to be done: the command parameters **CLA INS P1 P2 Lc' Le' 00 00** in the decrypted command must match the wrapping, otherwise the command is rejected, with **SW1-SW2 = swDesCheckError**.

9.2.5 Encryption of Responses in the Enhanced BasicCard

A response has the following structure (the shaded block is optional):

ODATA	**SW1**	**SW2**

Encryption consists of the following steps:

- Append six zeroes:

P =	**ODATA**	**SW1**	**SW2**	**00**	**00**	**00**	**00**	**00**	**00**

- Encrypt the resulting response **P**, with **C = ME**$_K$ (**P**) or **C = MEDE2**$_K$ (**P**):

C

- Append the original **SW1-SW2**:

C	**SW1**	**SW2**

The resulting response is always exactly 8 bytes longer than the original response. As with command encryption, these 8 bytes of redundancy enable an authentication check to be done on the response: if the decrypted response doesn't end with **SW1-SW2** followed by six zeroes, the response is rejected, and **SW1-SW2 = swBadDesResponse** is returned to the caller in the Terminal program.

Note: If status bytes **SW1 SW2** indicate an error (i.e. **SW1SW2 <> swCommandOK** and **SW1 <> sw1LeWarning**), then the response is not encrypted.

9.2.6 Encryption of Commands in the Professional BasicCard

The Professional BasicCard required a new encryption algorithm, because the algorithms described above for the Enhanced BasicCard are not compatible with the **T=0** protocol.

A command has the following structure (shaded blocks are optional):

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**Le**

Encryption consists of the following steps:

- Insert an **LeFlag** byte: **01** if **Le** is present, **00** if **Le** is absent:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**LeFlag**	**Le**

- If the **Le** field is absent, append **Le'** = **00**:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**LeFlag**	**Le'**

- Calculate the 32-bit **CRC** of the resulting data:

  CRC = CRC32 (**CLA || INS || P1 || P2** [|| **Lc || IDATA**] || **LeFLag || Le'**)

  The **CRC32** function is defined in **7.15.4 CRC Calculations**.

- If the **Lc** field is absent, insert **Lc' = 00**:

CLA	**INS**	**P1**	**P2**	**Lc'**	**IDATA**	**LeFlag**	**Le'**

- Append two zeroes, followed by the **CRC** (now the command tail **P** is at least 8 bytes long):

CLA	**INS**	**P1**	**P2**	**Lc'**	**P =**	**IDATA**	**LeFlag**	**Le'**	**00**	**00**	**CRC**

- Encrypt the command tail **P**, with **C = ME$_K$ (P)** , **C = MEDE2$_K$ (P)** , or **C = MEDE3$_K$ (P)**:

CLA	**INS**	**P1**	**P2**	**Lc'**	**C**

- Adjust **Lc'**, and append **Le''**:

CLA	**INS**	**P1**	**P2**	**Lc' + 8**	**C**	**Le''**

Le'' is computed as follows (this is where **T=0** compatibility comes in):

- If **Le** was absent, then **Le'' = 08**
- If **Le = 00**, then **Le'' = 00**
- Otherwise, **Le'' = Le + 08**

The resulting command is 8 or 9 bytes longer than the original command. When the BasicCard receives the command, it checks that the decrypted command tail **P** is valid, and that the **CRC** is correct. If not, the command is rejected, with **SW1-SW2 = swDesCheckError**.

9.2.7 Encryption of Responses in the Professional BasicCard

A response has the following structure (the shaded block is optional):

ODATA	**SW1**	**SW2**

Encryption consists of the following steps:

- Calculate the 32-bit **CRC** of the response:

  CRC = CRC32 ([**ODATA** ||] **SW1 || SW2**)

  The **CRC32** function is defined in **7.15.4 CRC Calculations**.

- Append two zeroes and the **CRC**:

P =	**ODATA**	**SW1**	**SW2**	**00**	**00**	**CRC**

- Encrypt the resulting response **P**, with $\mathbf{C} = \mathbf{ME}_K(\mathbf{P})$, $\mathbf{C} = \mathbf{MEDE2}_K(\mathbf{P})$, or $\mathbf{C} = \mathbf{MEDE3}_K(\mathbf{P})$:

C

- Append the original **SW1-SW2**:

C	**SW1**	**SW2**

The resulting response is 8 bytes longer than the original response. If the decrypted response doesn't end in **SW1 SW2 00 00 CRC**, the response is rejected, and **SW1-SW2 = swBadDesResponse** is returned to the caller in the Terminal program.

Note: If status bytes **SW1-SW2** indicate an error (i.e. **SW1SW2 <> swCommandOK** and **SW1 <> sw1LeWarning**), then the response is not encrypted.

9.3 Certificate Generation Using DES

The ZC-Basic **Certificate** command is described in **3.18.7 Certificate Generation**. The certificate generation algorithm is as follows:

Let **P** be the data to be signed. Append the byte **80** to **P** (this ensures that messages differing only in the number of trailing zeroes will have different certificates). Split the resulting **P** into 8-byte blocks $\mathbf{P}_1$,..., $\mathbf{P}_n$, padding the last block $\mathbf{P}_n$ with zeroes if necessary. Fill the initialisation vector $\mathbf{C}_0$ with zeroes, and then compute, for $1 \le i \le n$:

$\mathbf{C}_i = \mathbf{EDE3}_K(\mathbf{C}_{i-1}\ \mathbf{Xor}\ \mathbf{P}_i)$	(for keys **K** 24 bytes or longer, if supported)
$\mathbf{C}_i = \mathbf{EDE2}_K(\mathbf{C}_{i-1}\ \mathbf{Xor}\ \mathbf{P}_i)$	(for keys **K** 16 bytes or longer)
$\mathbf{C}_i = \mathbf{E}_K(\mathbf{C}_{i-1}\ \mathbf{Xor}\ \mathbf{P}_i)$	(for keys **K** shorter than 16 bytes)

The certificate is the final ciphertext block $\mathbf{C}_n$.

9.4 The AES Algorithm

On 28th February 2001, the US National Institute of Standards and Technology announced the Advanced Encryption Standard (**AES**), the long-awaited replacement for the **DES** standard. **AES** is described in "Draft Federal Information Processing Standard for the AES". This document is available from NIST's web site, at http://csrc.nist.gov/encryption/aes. **AES** uses the *Rijndael* algorithm, designed by Joan Daemen and Vincent Rijmen, as its cryptographic primitive. In its original specification, the Rijndael algorithm encrypts and decrypts data blocks of length 128, 192, or 256 bits, using a key of length 128, 192, or 256 bits. The **AES** specification fixes the block length at 128 bits (i.e. 16 bytes), but retains the three key length options.

AES with a 128-bit key length (or **AES-128**) is considered equal or superior in security to Triple DES. However, it is roughly six times faster. Longer key lengths are correspondingly more secure. For details of how to call the **AES** encryption primitives from a ZC-Basic program, see **7.8 AES: The Advanced Encryption Standard Library**.

9.5 Implementation of AES in the Professional BasicCard

This section parallels **9.2 Implementation of DES in the BasicCard**. Here the functions $\mathbf{E}_K$ and $\mathbf{D}_K$ are the **AES-*xxx*** encryption and decryption primitives, where *xxx* is the key length in bits: 128, 192, or 256. To start with, we need to know how to encrypt a message that is longer than 16 bytes. (All commands and responses encrypted with **AES** in the BasicCard are at least 16 bytes long.)

9.5.1 *The Message Encryption Function* ***AES-ME***$_K$

The **AES-***xxx* message encryption function **C = AES-ME**$_K$ **(P)** is defined as follows. We are given:

- a message **P**, at least 16 bytes in length;
- a 16-byte key **K**;
- the **AES-** *xxx* encryption and decryption functions $\mathbf{E_K}$ and $\mathbf{D_K}$;
- a 16-byte *initialisation vector* $\mathbf{C_0}$ (more about this in **9.5.3 The Initialisation Vector**).

First, split the message **P** into 16-byte blocks $\mathbf{P_1}$, $\mathbf{P_2}$,..., $\mathbf{P_{n-1}}$, plus a final block $\mathbf{P_n}$ that may be shorter than 16 bytes. Pad this final block with m zeroes to a length of 16 bytes (so $0 \leq m \leq 15$). Then compute, for $1 \leq i \leq n$:

$$\mathbf{C_i = E_K (C_{i-1} \text{ Xor } P_i)}$$

(Note that the initialisation vector $\mathbf{C_0}$ is needed to compute $\mathbf{C_1}$.) Then throw away the last m bytes of the *penultimate* block $\mathbf{C_{n-1}}$, and concatenate the resulting blocks $\mathbf{C_1}$,..., $\mathbf{C_n}$ to get the encrypted ciphertext **C**. For an explanation of why bytes are discarded from the penultimate block, see the description of ciphertext stealing in **9.2.1 The Message Encryption Functions ME**$_K$ **and ME**$_K^3$.

9.5.2 *The Message Decryption Function* ***AES-MD***$_K$

The **AES-***xxx* message decryption function **P = AES-MD**$_K$ **(C)** is the inverse of **AES-ME**$_K$. First restore the penultimate block $\mathbf{C_{n-1}}$ to 16 bytes, as described for **DES** in **9.2.1 The Message Encryption Functions ME**$_K$ **and ME**$_K^3$. Then compute, for $1 \leq i \leq n$:

$$\mathbf{P_i = C_{i-1} \text{ Xor } D_K (C_i)}$$

Throw away the last m bytes in $\mathbf{P_n}$ (which should all be zero), and concatenate all the resulting blocks $\mathbf{P_1}$,..., $\mathbf{P_n}$ to get the original plaintext message **P**.

9.5.3 *The Initialisation Vector*

The initialisation vector $\mathbf{C_0}$ is determined as follows:

For the first command following a **START ENCRYPTION** command, the initialisation vector $\mathbf{C_0}$ depends on the command and response fields of the **START ENCRYPTION** command:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	**10**	*algorithm*	*key*	**08**	*Random number* **RA** (8 bytes)	**00**

Response:

ODATA	SW1	SW2
algorithm (1 byte); *Random number* **RB** (8 bytes)	**90**	**00**

In this case, $\mathbf{C_0}$ consists of the first four bytes of **RA**, followed by all eight bytes of **RB**, followed by the last four bytes of **RA**.

For subsequent commands and responses, $\mathbf{C_0}$ is simply the last ciphertext block $\mathbf{C_n}$ of the previous message.

9.5.4 *Encryption of Commands*

A command has the following structure (shaded blocks are optional):

CLA	INS	P1	P2	Lc	IDATA	Le

Encryption consists of the following steps:

- Insert an **LeFlag** byte: **01** if **Le** is present, **00** if **Le** is absent:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**LeFlag**	**Le**

- If the **Le** field is absent, append **Le'** = **00**:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**LeFlag**	**Le'**

- Calculate the 32-bit **CRC** of the resulting data:

  CRC = CRC32 (**CLA || INS || P1 || P2** [|| **Lc || IDATA**] || **LeFLag || Le'**)

  The **CRC32** function is defined in **7.15.4 CRC Calculations**.

- If the **Lc** field is absent, insert **Lc' = 00**:

CLA	**INS**	**P1**	**P2**	**Lc'**	**IDATA**	**LeFlag**	**Le'**

- Append ten zeroes, followed by the **CRC** (now the command tail **P** is at least 16 bytes long):

CLA	**INS**	**P1**	**P2**	**Lc'**	**P =**	**IDATA**	**LeFlag**	**Le'**	**00**	**...**	**00**	**CRC**

- Encrypt the command tail **P**, with **C = AES-ME$_K$** (**P**):

CLA	**INS**	**P1**	**P2**	**Lc'**	**C**

- Adjust **Lc'**, and append **Le''**:

CLA	**INS**	**P1**	**P2**	**Lc' + 16**	**C**	**Le''**

Le'' is computed as follows:

- If **Le** was absent, then **Le'' = 10**
- If **Le = 00**, then **Le'' = 00**
- Otherwise, **Le'' = Le + 10**

The resulting command is 16 or 17 bytes longer than the original command. When the BasicCard receives the command, it checks that the decrypted command tail **P** is valid, and that the **CRC** is correct. If not, the command is rejected, with **SW1-SW2 = swAesCheckError**.

9.5.5 Encryption of Responses

A response has the following structure (the shaded block is optional):

ODATA	**SW1**	**SW2**

Encryption consists of the following steps:

- Calculate the 32-bit **CRC** of the response:

  CRC = CRC32 ([**ODATA ||**] **SW1 || SW2**)

  The **CRC32** function is defined in **7.15.4 CRC Calculations**.

- Append ten zeroes and the **CRC**:

P =	**ODATA**	**SW1**	**SW2**	**00**	**...**	**00**	**CRC**

- Encrypt the resulting response **P**, with $\mathbf{C} = \mathbf{AES\text{-}ME_K}\ (\mathbf{P})$:

C

- Append the original **SW1-SW2**:

C	SW1	SW2

The resulting response is 16 bytes longer than the original response. If the decrypted response doesn't end in **SW1 SW2 00...00 CRC**, the response is rejected, and **SW1-SW2 = swBadAesResponse** is returned to the caller in the Terminal program.

Note: If status bytes **SW1-SW2** indicate an error (i.e. **SW1SW2 <> swCommandOK** and **SW1 <> sw1LeWarning**), then the response is not encrypted.

9.6 The EAX Algorithm

EAX is an algorithm for Authenticated Encryption, designed by M.Bellare, P. Rogaway, and D. Wagner. A brief explanation of the algorithm follows; the full description is available from NIST's web site, at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/.

The **EAX** algorithm was designed to achieve the dual aims of secrecy and authentication, using a single cryptographic key. For encryption it uses the **CTR** algorithm; for authentication it uses a generalisation of the **OMAC** algorithm (described in **9.8 The OMAC Algorithm**), which the authors call "tweaked **OMAC**".

The **EAX** algorithm uses a block cipher $\mathbf{E_K}\ (\mathbf{B})$, which operates on blocks **B** of length **n** bits. The choice of block cipher is left to the implementer. ZeitControl's implementation of **EAX** uses **AES** as its block cipher, with key length 128, 192, or 256 bits; the block length **n** is equal to 128 bits (16 bytes) in all cases.

9.6.1 The CTR Algorithm

CTR is short for counter-mode encryption. The **CTR** algorithm is a standard encryption algorithm, that takes a Key **K** and a Nonce **N** as input parameters, and encrypts a Message **M** to produce ciphertext **C** of the same length as **M**:

$$\mathbf{C} = \mathbf{CTR_K^N(M)}$$

Suppose $\mathbf{M} = \mathbf{M_1} \parallel \mathbf{M_2} \parallel \ldots \parallel \mathbf{M_m}$, with all blocks (except possibly the last) **n** bits long. Define

$$\mathbf{S_1} = \mathbf{E_K}\ (\mathbf{N}),\ \mathbf{S_2} = \mathbf{E_K}\ (\mathbf{N} + 1),\ \ldots,\ \mathbf{S_m} = \mathbf{E_K}\ (\mathbf{N} + \mathrm{m} - 1)$$

where addition is performed modulo $2^{\mathbf{n}}$, treating **N** as an integer $0 \leq \mathbf{N} < 2^{\mathbf{n}}$. Then let

$$\mathbf{C_i} = \mathbf{M_i}\ \mathbf{Xor}\ \mathbf{S_i} \qquad (1 \leq \mathrm{i} \leq \mathrm{m})$$

with $\mathbf{C_m}$ truncated to the same length as $\mathbf{M_m}$. Then the ciphertext **C** is given by

$$\mathbf{CTR_K^N(M)} = \mathbf{C_1} \parallel \mathbf{C_2} \parallel \ldots \parallel \mathbf{C_m}$$

The Nonce **N** does not have to be secret, but it must be different for each invocation of **CTR** for a given Key **K**.

9.6.2 Tweaked OMAC

The **EAX** algorithm requires a parameterised version of the **OMAC** algorithm, which it calls "tweaked **OMAC**". The parameter is an integer *t*:

$$\mathbf{OMAC_K^{\mathit{t}}(M)} = \mathbf{OMAC_K}\ ([t]_{\mathbf{n}} \parallel \mathbf{M})$$

where $[t]_{\mathbf{n}}$ denotes the **n**-bit binary representation of *t* (with most significant bit first).

9.6.3 EAX

Now we can define the **EAX** algorithm. It takes the following items as input:

- A Key **K**, for use by the block encryption algorithm $\mathbf{E_K}$.
- A Nonce **N**, of any length. **N** does not have to be secret, but it must be different for each invocation of **EAX** for a given Key **K**. The BasicCard uses a 16-byte Nonce for the encryption of Commands and Responses.
- A Header **H**, of any length. **H** is authenticated, but not encrypted, by the **EAX** algorithm. **H** is often referred to as *Associated Data*.
- A Message **M**, which will be encrypted by the **EAX** algorithm.

EAX computes as output a ciphertext **C** and a Tag **T**, as follows:

- $\mathbf{U = OMAC_K^0(N)}$
- $\mathbf{V = OMAC_K^1(H)}$
- $\mathbf{C = CTR_K^U(M)}$
- $\mathbf{W = OMAC_K^2(C)}$
- **T = U Xor V Xor W**

We write this as

$$\mathbf{CT = EAX.Encrypt_K^{NH}(M)}$$

C is the same length as **M**; the Tag **T** is **n** bits long. (The full definition of the **EAX** algorithm, in the original paper, defines a parameter τ as the length of the desired Tag; **T** is truncated to τ bits. ZeitControl's implementation does not use this parameter.)

The **CTR** algorithm is its own inverse, so decryption follows the same steps as encryption, with $\mathbf{M = CTR_K^U(C)}$ instead of $\mathbf{C = CTR_K^U(M)}$. After the last step, the recipient can check that the computed Tag **T** is equal to the Tag received with the ciphertext. If not, the message is rejected.

9.7 Implementation of EAX in the BasicCard

The **EAX** algorithm is currently available in Professional BasicCard **ZC5.5** and MultiApplication BasicCard **ZC6.5**. It has a user-callable interface, described in **7.9 The EAX Library**; and it can also be specified in the **START ENCRYPTION** command for the authentication of Commands and Responses. The three constants **AlgEaxAes128**, **AlgEaxAes192**, and **AlgEaxAes256** are defined in the file AlgID.DEF for this purpose.

This section describes the encryption of Commands and Responses using **EAX**.

9.7.1 The Nonce

The Nonce **N** is always 16 bytes long. It is determined as follows:

For the first command following a **START ENCRYPTION** command, **N** depends on the command and response fields of the **START ENCRYPTION** command:

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	10	*algorithm*	*key*	08	*Random number* **RA** (8 bytes)	09

Response:

ODATA	SW1	SW2
algorithm (1 byte); *Random number* **RB** (8 bytes)	90	00

In this case, **N** consists of the first four bytes of **RA**, followed by all eight bytes of **RB**, followed by the last four bytes of **RA**.

For subsequent commands and responses, **N** is simply the Tag field **T** of the previous message.

9.7.2 Encryption of Commands

A command has the following structure (shaded blocks are optional):

CLA	INS	P1	P2	Lc	IDATA	Le

Encryption consists of the following steps:

- If **Le** is absent, set **Le' = 16**; if **Le** is zero, set **Le' = 0**; otherwise set **Le' = Le+16**:

CLA	INS	P1	P2	Lc	IDATA	Le'

- If **Lc** is absent, set **Lc' = 16**; otherwise set **Lc' = Lc+16**:

CLA	INS	P1	P2	Lc'	IDATA	Le'

- Authenticate **CLA || INS || P1 || P2 || Lc' || Le'**, encrypt the **IDATA** field, and compute the Tag **T**:

  H = CLA || INS || P1 || P2 || Lc' || Le'

  $\mathbf{CT = EAX.Encrypt_{K}^{NH}(IDATA)}$

- Replace **IDATA** with **C**, and insert the Tag **T**:

CLA	INS	P1	P2	Lc'	C	T	Le'

Then, if the **T=0** protocol is active, the last byte of **T** is replaced by **Le'**. The resulting command is 16-18 bytes longer than the original command. When the BasicCard receives the command, it checks that the **EAX** Tag **T** is correct. If not, the command is rejected, with **SW1-SW2 = swAesCheckError**.

9.7.3 Encryption of Responses

A response has the following structure (the shaded block is optional):

ODATA	SW1	SW2

Encryption consists of the following steps:

- Authenticate **SW1-SW2**, encrypt the **ODATA** field, and compute the Tag **T**:

  H = SW1 || SW2

  $\mathbf{CT = EAX.Encrypt_{K}^{NH}(ODATA)}$

- Replace **ODATA** with **C**, and insert the Tag **T**:

C	T	SW1	SW2

The resulting response is 16 bytes longer than the original response. If the **EAX** Tag **T** is incorrect, the response is rejected, and **SW1-SW2 = swBadAesResponse** is returned to the caller in the Terminal program.

Note: If status bytes **SW1-SW2** indicate an error (i.e. **SW1SW2 <> swCommandOK** and **SW1 <> sw1LeWarning**), then the response is not authenticated.

9.8 The OMAC Algorithm

The **OMAC** algorithm, designed by Tetsu Iwata and Kaoru Kurosawa, is a Message Authentication algorithm: it computes a Tag $\mathbf{T} = \mathbf{OMAC_K}\ (\mathbf{M})$ from a Message **M** using a Key **K**. This Tag authenticates **M** to anybody who knows **K**. In other words, if I receive a Message **M** and a Tag **T**, with **T** equal to $\mathbf{OMAC_K}\ (\mathbf{M})$, then I can be sure that

- **K** was known by whoever computed **T**;
- **M** has not been changed since it was used to compute **T**.

Only the Key **K** needs to be kept secret; **M** and **T** can be sent unencrypted.

We give a brief explanation of the **OMAC** algorithm here; a full description is available from NIST's web site, at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/. (In the published description, algorithms **OMAC1** and **OMAC2** are defined; we describe here the **OMAC1** variant, as used by the BasicCard and by the **EAX** algorithm.)

The **OMAC** algorithm uses a block cipher $\mathbf{E_K}\ (\mathbf{B})$, which operates on blocks **B** of length **n** bits. The choice of block cipher is left to the implementer. ZeitControl's implementation of **OMAC** uses **AES** as its block cipher, with key length 128, 192, or 256 bits; the block length **n** is equal to 128 bits (16 bytes) in all cases.

OMAC is an abbreviation for "One-key CBC MAC". CBC MAC is a Message Authentication algorithm that requires the length of the Message **M** to be a multiple of **n** bits; so we can write

$$\mathbf{M} = \mathbf{M}_1 \parallel \mathbf{M}_2 \parallel \ldots \parallel \mathbf{M}_m$$

where each $\mathbf{M}_i$ is **n** bits long. Then we define

$\mathbf{C}_0 = \mathbf{0}^\mathbf{n}$	($\mathbf{0}^\mathbf{n}$ denotes the block consisting of **n** zero bits)
$\mathbf{C}_i = \mathbf{E_K}\ (\mathbf{M}_i\ \mathbf{Xor}\ \mathbf{C}_{i-1})$	$(1 \le i \le m)$
$\mathbf{CBC_K}\ (\mathbf{M}) = \mathbf{C}_m$	

If **M** is not a multiple of **n** bits, we must pad it in some way. Appending zeroes is not good enough; it would fail to distinguish between messages differing only in their number of trailing zeroes. One simple method is to append 1, followed by enough zeroes to bring the length to a multiple of **n**. The disadvantage of this method is that it may require an extra call to the block encryption algorithm $\mathbf{E_K}$. The **OMAC** algorithm avoids this extra call at the cost of increased theoretical complexity, but with negligible practical overhead. Let **u** be any non-zero element of the finite field $\mathbf{GF}(2^\mathbf{n})$, and let $\mathbf{L} = \mathbf{E_K}\ (\mathbf{0}^\mathbf{n})$; we can interpret **L** as an element of the field, and multipy it by **u** to get **Lu**, $\mathbf{Lu}^2$ etc. The reason for introducing the field $\mathbf{GF}(2^\mathbf{n})$ is that it allows a concrete proof of security to be given; interested readers can consult the published description on the above web site. In practice, when **n** is equal to 128 we can choose **u** so that multiplication by **u** reduces to the following simple procedure:

- rotate **L** left by one bit;
- if the least significant bit is now 1, then **Xor** the least significant byte with **86**.

(The computation of **L** requires a call to the block encryption algorithm $\mathbf{E_K}$, which we were supposed to be trying to avoid; but this call is only required once for a given **K**, after which **L** can be re-used for subsequent messages.)

Now we can define the padding function. Suppose $\mathbf{M} = \mathbf{M}_1 \parallel \mathbf{M}_2 \parallel \ldots \parallel \mathbf{M}_m$, with all blocks (except possibly the last) **n** bits long. (If **M** itself is zero bits long, set m=1 and let $\mathbf{M}_m$ be the empty block.) Then if $|\mathbf{M}_m|$ is equal to **n**, set $\mathbf{P} = \mathbf{M}_m\ \mathbf{Xor}\ \mathbf{Lu}$; otherwise, pad $\mathbf{M}_m$ to length **n** by appending a 1 followed by $\mathbf{n} - |\mathbf{M}_m| - 1$ zeroes, and set $\mathbf{P} = (\mathbf{M}_m \parallel 100\ldots00)\ \mathbf{Xor}\ \mathbf{Lu}^2$. Then

$$\mathbf{Pad_K}\ (\mathbf{M}) = \mathbf{M}_1 \parallel \mathbf{M}_2 \parallel \ldots \parallel \mathbf{M}_{m-1} \parallel \mathbf{P}$$

The **OMAC** algorithm computes the following **n**-bit Tag:

$$\mathbf{OMAC_K}\ (\mathbf{M}) = \mathbf{CBC_K}\ (\mathbf{Pad_K}\ (\mathbf{M}))$$

9.9 Implementation of OMAC in the BasicCard

The **OMAC** algorithm is currently available in Professional BasicCard **ZC5.5** and MultiApplication BasicCard **ZC6.5**. It has a user-callable interface, described in **7.10 The OMAC Library**; and it can also be specified in the **START ENCRYPTION** command for the authentication of Commands and Responses. The three constants **AlgOmacAes128**, **AlgOmacAes192**, and **AlgOmacAes256** are defined in the file AlgID.DEF for this purpose.

This section describes the authentication of Commands and Responses using **OMAC**.

9.9.1 Authentication of Commands

A command has the following structure (shaded blocks are optional):

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**Le**

Authentication consists of the following steps:

- If **Le** is absent, set **Le' = 16**; if **Le** is zero, set **Le' = 0**; otherwise set **Le' = Le+16**:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA**	**Le'**

- If **Lc** is absent, set **Lc' = 16**; otherwise set **Lc' = Lc+16**:

CLA	**INS**	**P1**	**P2**	**Lc'**	**IDATA**	**Le'**

- Calculate the **OMAC** Tag of the resulting data:

  T = OMAC$_K$ (**CLA || INS || P1 || P2 || Lc'** [**|| IDATA**] **|| Le'**)

- Append **T** to **IDATA**:

CLA	**INS**	**P1**	**P2**	**Lc'**	**IDATA**	**T**	**Le'**

Then, if the **T=0** protocol is active, the last byte of **T** is replaced by **Le'**. The resulting command is 16-18 bytes longer than the original command. When the BasicCard receives the command, it checks that the **OMAC** Tag **T** is valid. If not, the command is rejected, with **SW1-SW2 = swAesCheckError**.

9.9.2 Authentication of Responses

A response has the following structure (the shaded block is optional):

ODATA	**SW1**	**SW2**

Authentication consists of the following steps:

- Calculate the **OMAC** Tag of the response:

  T = OMAC$_K$ ([**ODATA ||**] **SW1 || SW2**)

- Append **T** to **ODATA**:

ODATA	**T**	**SW1**	**SW2**

The result is 16 bytes longer than the original response. If the **OMAC** Tag **T** is incorrect, the response is rejected, and **SW1-SW2 = swBadAesResponse** is returned to the caller in the Terminal program.

Note: If status bytes **SW1-SW2** indicate an error (i.e. **SW1SW2 <> swCommandOK** and **SW1 <> sw1LeWarning**), then the response is not authenticated.

9.10 Customer-Specific Encryption Keys

The latest BasicCards (**ZC7-** and **ZC8-**series from **REV D**) provide a Customer-Specific Encryption Key mechanism. This is based on a secret 24-byte Master Key $\mathbf{K_M}$ contained in each BasicCard. This Master Key is not known to any single person; it is stored in distributed format among several BasicCards at ZeitControl, and at least two of these BasicCards are required to reconstitute the key.

For a modest processing fee, ZeitControl can provide a 32-byte Customer-Specific Key $\mathbf{K_C}$, which can be used as a key for command-response encryption. We envisage that the chief use for this key will be to provide a Secure Channel during card personalisation, before any application-specific cryptographic keys have been exchanged.

9.10.1 Customer Key Generation

Contact ZeitControl to obtain your own 32-byte Customer-Specific Key $\mathbf{K_C}$. We will generate:

- a unique Key Derivation Decryption Parameter **KDP**, which is a random 32-byte string;
- the Customer-Specific Key $\mathbf{K_C}$, generated from **KDP** and $\mathbf{K_M}$ as follows:

  Hash = SHA-256 (KDP)
  $\mathbf{K_C}$ **= AES-192 (**$\mathbf{K_M}$ **, Left(Hash, 16)) + AES-192 (**$\mathbf{K_M}$ **, Right(Hash, 16))**

  On request, we will split $\mathbf{K_C}$ into two parts, $\mathbf{K_A}$ and $\mathbf{K_B}$, which we will deliver separately. The Customer Key $\mathbf{K_C} = \mathbf{K_A}$ **Xor** $\mathbf{K_B}$ can then be reconsituted in a secure environment.

9.10.2 How It Works

The Terminal program has no access to $\mathbf{K_M}$, so it must be told $\mathbf{K_C}$. To do this, call the **Crypto** System Library procedure **CryptoSetCustomerKey**:

Call CryptoSetCustomerKey ($\mathbf{K_C}$**)**

The BasicCard only has to be told what **KDP** is, and it can compute $\mathbf{K_C}$ for itself. Before the Customer Key can be used for encryption, **CryptoSetKDP** must be called in the card:

Call CryptoSetKDP (KDP)

9.10.3 The START ENCRYPTION Command

To enable encryption using a Customer Key, call **START ENCRYPTION** with the following parameters:

- add 8 to the *Algorithm* selector in **P1**;
- set **P2=&HFF**.

As a special case, this format is accepted by the **ZC8-**series MultiApplication card too (the **START ENCRYPTION** format is usually different for MultiApplication cards).

If the **START ENCRYPTION** command is successful, then the pre-defined variables **Algorithm** and **KeyNumber** are set to **P1** and **P2** respectively.

9.10.4 Example Data

Before you order a Customer-Specific Key from ZeitControl, you can try out your software with the example data in **Crypto.def**. For testing your software in a real card, two 32-byte **String** constants are defined:

CryptoExampleKDP$
CryptoExampleRealCustomerKey$

The development software does not know the Master Key $\mathbf{K_M}$, so if your BasicCard program is running in the **ZCMSim** BasicCard simulator or the **ZCMDCard** BasicCard debugger, it uses a Simulated Master Key $\mathbf{K_S}$, defined as the 24-byte **String** constant

CryptoExampleSimulatedMasterKey$

The Customer Key K_C derived from **CryptoExampleKDP$** using the Simulated Master Key K_S is the 32-byte **String** constant

CryptoExampleSimulatedCustomerKey$

To simplify testing, a new function **IsPhysicalReader()** has been added to the **MISC** System Library – see **7.15.5 Communications**. Here is an example of its use:

```
#Include Misc.def

If IsPhysicalReader() Then
  Call CryptoSetCustomerKey (CryptoExampleCustomerKey$)
Else
  Call CryptoSetCustomerKey (CryptoExampleSimulatedCustomerKey$)
End If
```

9.11 Encryption – a Worked Example

This section shows the progression from ZC-Basic source code to encrypted messages. All source files are supplied with the software development kit, in the `BasicCardV8\Examples\EchoTest` directory. Two encryption algorithms are exhibited: **AlgTripleDesEDE2Crc** (**Triple DES-EDE2** with **CRC**) and **AlgEaxAes192** (**EAX** using **AES-192**).

9.11.1 The Source Code

We ran the **KeyGen** program to generate four cryptographic keys:

```
KeyGen TestKeys –K108 -K116(16) -K124(24) –K132(32)
```

This produced output file **TestKeys.bas**:

```
Declare Key 108 = &H03,&HAF,&H59,&H92,&HC9,&HE5,&H0D,&HC6
Declare Key 116(16) = &H1D,&HE1,&HFA,&HB0,&HC8,&H1F,&HC2,&HE6,_
        &H95,&H3B,&H46,&H1C,&HE7,&HFD,&HCB,&H53
Declare Key 124(24) = &HD6,&HB4,&HCE,&HAC,&H3A,&H43,&H62,&H88,_
        &HEF,&H0B,&HAD,&HF0,&H41,&H6D,&HED,&H74,_
        &H2A,&H01,&H73,&H27,&HD3,&H7F,&HCE,&H15
Declare Key 132(32) = &HC7,&H5D,&HB1,&H37,&H52,&HC0,&HB6,&HFF,_
        &H2E,&H9D,&H55,&H06,&HD2,&H07,&H81,&H57,_
        &HAC,&H0C,&H81,&H73,&H27,&HB9,&HD4,&H1C,_
        &H05,&H76,&H6D,&H52,&H0D,&H40,&H21,&H67
```

Then we wrote a ZC-Basic Terminal program **EchoTest.bas** to send encrypted **ECHO** commands. The **EchoTest** program takes a list of algorithm names as parameters. The BasicCard source file, **EchoCard.bas**, reduces to just the following statements if compiled for a single-application BasicCard:

```
#Include TestKeys.bas
Declare ApplicationID = "Single-application EchoTest"
```

The file **Compile.bat** in the `BasicCardV8\Examples\EchoTest` directory compiles the **EchoTest.bas** source file, along with a separate BasicCard image file for each card type.

Executing **Sim.bat** from this directory tests all encryption algorithms, for all card types, and generates log files for each run. We will look at a simpler example, generated by executing **DocGen.bat**:

```
..\..\ZCMSim –CPro –LExample EchoTest AlgNone AlgTripleDesEDE2Crc AlgEaxAes192
```

This sends three **ECHO** commands:

- unencrypted;
- using **Triple DES-EDE2** with **CRC**;
- using **EAX** with **AES-192**.

This creates the log file **Example.log**:

```
Port 1
ATR: 3B FB 13 00 FF 81 31 80 75 5A 43 35 2E 35 20 52 45 56 20 45 61
-> 00 00 05 C0 0E 00 00 00 CB
<- 00 00 1D 53 69 6E 67 6C 65 2D 61 70 70 6C 69 63 61 74 69 6F 6E
   20 45 63 68 6F 54 65 73 74 61 1B 3D
-> 00 40 09 C0 14 01 00 03 61 62 63 03 FC
<- 00 40 05 62 63 64 90 00 B0
-> 00 00 0A C0 10 24 74 04 9C 13 E7 F7 00 11
<- 00 00 07 24 29 72 6A 36 61 05 40
-> 00 40 11 C0 14 01 00 0B 4D 0F 9C 3E A8 19 68 C8 01 85 19 0B E8
<- 00 40 0D EF 0A F4 EB 4F 28 9D AF AE F4 3A 90 00 12
-> 00 00 0E C0 12 00 00 08 0F 73 E5 9E 4E FD 68 CA 08 CA
<- 00 00 02 90 00 92
-> 00 40 0E C0 10 42 7C 08 E0 0A 92 C8 11 F8 ED 54 00 48
<- 00 40 0B 42 D8 C5 67 B3 28 8D B0 79 61 09 C4
-> 00 00 19 C0 14 01 00 13 42 4B 97 15 2C 56 AD C5 D6 00 81 1D 99
   5B 20 45 6A A3 47 13 36
<- 00 00 15 2B 58 1C 6E D8 31 47 57 6C 33 A3 FF 8C 89 26 17 30 D5
   A2 90 00 0D
-> 00 40 16 C0 12 00 00 10 CE C0 E4 7D 8B 47 F3 9B E8 E9 3D 5D ED
   72 60 5D 10 74
<- 00 40 02 90 00 D2
```

Note: If you run the **EchoTest** program yourself, your log file will be different, due to the different random numbers generated by the Terminal program.

If we strip the **T=1** protocol bytes **NAD PCB LEN . . . LRC** from each command and response, we get the following:

```
❶ ATR: 3B FB 13 00 FF 81 31 80 75 5A 43 35 2E 35 20 52 45 56 20 45 61
❷ -> C0 0E 00 00 00
  <- 53 69 6E 67 6C 65 2D 61 70 70 6C 69 63 61 74 69 6F 6E 20 45 63
     68 6F 54 65 73 74 61 1B
❸ -> C0 14 01 00 03 61 62 63 03
  <- 62 63 64 90 00
❹ -> C0 10 24 74 04 9C 13 E7 F7 00
  <- 24 29 72 6A 36 61 05
❺ -> C0 14 01 00 0B 4D 0F 9C 3E A8 19 68 C8 01 85 19 0B
  <- EF 0A F4 EB 4F 28 9D AF AE F4 3A 90 00
❻ -> C0 12 00 00 08 0F 73 E5 9E 4E FD 68 CA 08
  <- 90 00
❼ -> C0 10 42 7C 08 E0 0A 92 C8 11 F8 ED 54 00
  <- 42 D8 C5 67 B3 28 8D B0 79 61 09
❽ -> C0 14 01 00 13 42 4B 97 15 2C 56 AD C5 D6 00 81 1D 99 5B 20 45
     6A A3 47 13
  <- 2B 58 1C 6E D8 31 47 57 6C 33 A3 FF 8C 89 26 17 30 D5 A2 90 00
❾ -> C0 12 00 00 10 CE C0 E4 7D 8B 47 F3 9B E8 E9 3D 5D ED 72 60 5D
     10
  <- 90 00
```

❶ **ATR** (Answer To Reset) from the simulated BasicCard, including the text "**ZC5.5 REV E**"
❷ **GET APPLICATION ID** command and response
❸ **ECHO** command and response
❹ **START ENCRYPTION** command (algorithm = **&H24 = AlgTripleDesEDE2Crc**) and response
❺ **ECHO** command and response, encrypted with **AlgTripleDesEDE2Crc**
❻ **END ENCRYPTION** command and response
❼ **START ENCRYPTION** command (algorithm = **&H42 = AlgEaxAes192**) and response
❽ **ECHO** command and response, encrypted with **AlgEaxAes192**
❾ **END ENCRYPTION** command and response

We will look at these commands one by one.

9.11.2 GET APPLICATION ID Command and Response

The **EchoTest** program calls **GET APPLICATION ID** to find out whether it is dealing with a single-application BasicCard or a MultiApplication BasicCard:

Command:

CLA	INS	P1	P2	Le
C0	0E	00	00	00

Response:

ODATA	SW1	SW2
"Single-application EchoTest"	61	1B

The BasicCard returns **"Single-application EchoTest"**, declared in **EchoCard.bas**.

9.11.3 Unencrypted ECHO Command and Response

The parameter "abc" is **61 62 63** in hexadecimal. The **ECHO** command adds **P1=01** to every byte:

Command:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	14	01	00	03	61 62 63	03

Response:

ODATA	SW1	SW2
62 63 64	90	00

9.11.4 START ENCRYPTION (Algorithm = AlgTripleDesEDE2Crc)

The **Rnd** function in the Terminal program returned **RA = &H4E9225DB**, and the random-number generator in the BasicCard operating system returned **RB = &H29726A36**. This led to the following **START ENCRYPTION** command-response pair (the first byte of **ODATA** confirms the choice of algorithm):

Command:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	10	24	74	04	9C 13 E7 F7	00

Response:

ODATA	SW1	SW2
24 29 72 6A 36	61	05

We build the initialisation vector C_0 from **RA** and **RB**, as described in section **9.2.3**:

C_0 **= 9C 13 29 72 6A 36 E7 F7**

9.11.5 Encrypted ECHO Command (Algorithm = AlgTripleDesEDE2Crc)

The unencrypted **ECHO** command:

Command:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	14	01	00	03	61 62 63	03

- Insert an **LeFlag** byte:

CLA	INS	P1	P2	Lc	IDATA	LeFlag	Le
C0	14	01	00	03	61 62 63	01	03

- Calculate the 32-bit **CRC** of the resulting data:

  CRC = CRC32 (C0 14 01 00 03 61 62 63 01 03) = &H9D95964E

- Append two zeroes, followed by the **CRC**:

CLA	INS	P1	P2	Lc	IDATA	LeFlag	Le		CRC
C0	14	01	00	03	61 62 63	01	03	00 00	9D 95 96 4E

- Now we must encrypt the command tail

  P = 61 62 63 01 03 00 00 9D 95 96 4E

  using the **Triple DES** message encryption function $\mathbf{MEDE2_K}$. Referring back to **9.2.1 The Message Encryption Functions $\mathbf{ME_K}$, $\mathbf{MEDE2_K}$, and $\mathbf{MEDE3_K}$** :

  K = 1D E1 FA B0 C8 1F C2 E6 95 3B 46 1C E7 FD CB 53 is key number 116 from TestKeys.bas;
  $\mathbf{C_0}$ **= 9C 13 29 72 6A 36 E7 F7** from the **START ENCRYPTION** command;
  $\mathbf{P_1}$ **= 61 62 63 01 03 00 00 9D** is the first message block;
  $\mathbf{P_2}$ **= 95 96 4E (00 00 00 00 00)** is the second message block;
  m = 5 is the length of padding required in $\mathbf{P_2}$.

  So we compute (you can check these in ZC-Basic, using the **DES** function):

  $\mathbf{C_1}$ **=** $\mathbf{EDE2_K}$ **(**$\mathbf{C_0}$ **Xor** $\mathbf{P_1}$**) =** $\mathbf{EDE2_K}$ **(FD 71 4A 73 69 36 E7 6A) = 4D 0F 9C 01 35 81 DA E8**
  $\mathbf{C_2}$ **=** $\mathbf{EDE2_K}$ **(**$\mathbf{C_1}$ **Xor** $\mathbf{P_2}$**) =** $\mathbf{EDE2_K}$ **(D8 99 D2 01 35 81 DA E8) = 3E A8 19 68 C8 01 85 19**

  and we throw away the last **m** bytes of $\mathbf{C_1}$ to get:

  C = $\mathbf{MEDE2_K}$ **(P) = 4D 0F 9C 3E A8 19 68 C8 01 85 19**

- To get the final version, **C** is wrapped in the original **CLA INS P1 P2 . . . Le**, with **Lc** and **Le** adjusted appropriately:

CLA	INS	P1	P2	Lc	C	Le
C0	14	01	00	0B	4D 0F 9C 3E A8 19 68 C8 01 85 19	0B

The unencrypted response to the **ECHO** command:

Response:

ODATA	SW1	SW2
62 63 64	90	00

- Calculate the 32-bit **CRC** of the response:

  CRC = CRC32 (62 63 64 90 00) = &HCF2CB422

- Append two zeroes and the **CRC**:

ODATA	SW1	SW2	
62 63 64	90	00	00 00 CF 2C B4 22

- Encrypt **P = 62 63 64 90 00 00 00 CF 2C B4 22** using $\mathbf{MEDE2_K}$, where

  K = 1D E1 FA B0 C8 1F C2 E6 95 3B 46 1C E7 FD CB 53 is key number 116 from TestKeys.bas;
  $\mathbf{C_0}$ **= 3E A8 19 68 C8 01 85 19** is $\mathbf{C_2}$ from the **ECHO** command just received;
  $\mathbf{P_1}$ **= 62 63 64 90 00 00 00 CF** is the first message block;

$\mathbf{P_2}$ = **2C B4 22 (00 00 00 00 00)** is the second message block;
m = 5 is the length of padding required in $\mathbf{P_2}$.

So we compute:

$\mathbf{C_1}$ = **EDE2**$_\mathbf{K}$ **(**$\mathbf{C_0}$ **Xor** $\mathbf{P_1}$**)** = **EDE2**$_\mathbf{K}$ **(5C CB 7D F8 C8 01 85 D6)** = **EF 0A F4 E4 7F A9 43 AC**
$\mathbf{C_2}$ = **EDE2**$_\mathbf{K}$ **(**$\mathbf{C_1}$ **Xor** $\mathbf{P_2}$**)** = **EDE2**$_\mathbf{K}$ **(C3 BE D6 E4 7F A9 43 AC)** = **EB 4F 28 9D AF AE F4 3A**

and we throw away the last **m** bytes of $\mathbf{C_1}$ to get:

C = MEDE2$_\mathbf{K}$ **(P) = EF 0A F4 EB 4F 28 9D AF AE F4 3A**

- Now the original **SW1-SW2** are appended, to get:

C	SW1	SW2
EF 0A F4 EB 4F 28 9D AF AE F4 3A	90	00

9.11.6 END ENCRYPTION

The unencrypted **END ENCRYPTION** command:

Command:

CLA	INS	P1	P2
C0	12	00	00

- Insert an **LeFlag** byte and append **Le' = 00**:

CLA	INS	P1	P2	LeFlag	Le'
C0	12	00	00	00	00

- Calculate the 32-bit **CRC** of the resulting data:

  CRC = CRC32 (C0 12 00 00 00 00) = &H13ED6700

- Insert **Lc' = 00**, and append two zeroes followed by the **CRC**:

CLA	INS	P1	P2	Lc'	LeFlag	Le'	
C0	12	00	00	00	00	00	00 00 13 ED 67 00

- Encrypt the command tail **P = 00 00 00 00 13 ED 67 00** with **MEDE2**$_\mathbf{K}$, where

K = 1D E1 FA B0 C8 1F C2 E6 95 3B 46 1C E7 FD CB 53
is key number 116 from TestKeys.bas;
$\mathbf{C_0}$ = **EB 4F 28 9D AF AE F4 3A** is C_2 from the **ECHO** response;
$\mathbf{P_1}$ = **00 00 00 00 13 ED 67 00** is the only message block;
m = 0 is the length of padding required in $\mathbf{P_1}$.

So we compute:

$\mathbf{C_1}$ = **EDE2**$_\mathbf{K}$ **(**$\mathbf{C_0}$ **Xor** $\mathbf{P_1}$**)** = **EDE2**$_\mathbf{K}$ **(EB 4F 28 9D BC 43 93 3A)** = **0F 73 E5 9E 4E FD 68 CA**

and **C = MEDE2**$_\mathbf{K}$ **(P)** is simply $\mathbf{C_1}$.

- Append **Le'' = 08** to get the final version:

CLA	INS	P1	P2	Lc	C	Le''
C0	12	00	00	08	0F 73 E5 9E 4E FD 68 CA	08

The response is not encrypted:

Response:

SW1	SW2
90	00

9.11.7 START ENCRYPTION (Algorithm = AlgEaxAes192)

The two calls to the **Rnd** function in the Terminal program returned **&HE00A92C8** and **&H11F8ED54**, giving **RA = E0 0A 92 C8 11 F8 ED 54**; and the random-number generator in the BasicCard returned **RB = D8 C5 67 B3 28 8D B0 79**. This led to the following **START ENCRYPTION** command-response pair (the first byte of **ODATA** confirms the choice of algorithm):

Command:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	10	42	7C	08	E0 0A 92 C8 11 F8 ED 54	00

Response:

ODATA	SW1	SW2
42 D8 C5 67 B3 28 8D B0 79	61	09

We build the Nonce **N** from **RA** and **RB**, as described in section **9.7.1**:

N = E0 0A 92 C8 D8 C5 67 B3 28 8D B0 79 11 F8 ED 54

9.11.8 Encrypted ECHO Command (Algorithm = AlgEaxAes192)

The unencrypted **ECHO** command:

Command:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	14	01	00	03	61 62 63	03

- Set **Le' = Le+10**, **Lc' = Lc+10**:

CLA	INS	P1	P2	Lc'	IDATA	Le'
C0	14	01	00	13	61 62 63	13

- Set **H = CLA || INS || P1 || P2 || Lc' || Le' = C0 14 01 00 13 13**, encrypt **IDATA**, and compute the Tag **T**:

  $\mathbf{CT = EAX.Encrypt_{K}^{NH}(IDATA)}$

  You can compute **C** and **T** in a ZC-Basic Terminal program, using the **EAX** System Library:

```
#Include EAX.DEF
#Include TestKeys.bas

Private N$ = Chr$(&HE0,&H0A,&H92,&HC8,&HD8,&HC5,&H67,&HB3,_
                  &H28,&H8D,&HB0,&H79,&H11,&HF8,&HED,&H54)
Private H$ = Chr$(&HC0,&H14,&H01,&H00,&H13,&H13)
Private C$ = Chr$(&H61,&H62,&H63)

Private EaxState$ : EaxState$ = EaxInit (192, Key(124))
Call EaxProvideNonce (EaxState$, Key(124), N$)
Call EaxProvideHeader (EaxState$, Key(124), H$)
Call EAXComputeCiphertext (EaxState$, Key(124), C$)

Private T$ : T$ = EaxComputeTag (EaxState$, Key(124))
```

  We get:

  C = 42 4B 97
  T = 15 2C 56 AD C5 D6 00 81 1D 99 5B 20 45 6A A3 47

  So the encrypted command is:

CLA	INS	P1	P2	Lc'	C
C0	**14**	**01**	**00**	**13**	**42 4B 97**

T	Le'
15 2C 56 AD C5 D6 00 81 1D 99 5B 20 45 6A A3 47	**13**

The unencrypted response to the **ECHO** command:

Response:

ODATA	SW1	SW2
62 63 64	**90**	**00**

- Set **N** equal to **T** from the **START ENCRYPTION** command:

  N = 15 2C 56 AD C5 D6 00 81 1D 99 5B 20 45 6A A3 47

- Set **H = SW1 || SW2 = 90 00**, encrypt **ODATA**, and compute the Tag **T**:

  $\mathbf{CT = EAX.Encrypt_{K}^{NH}(ODATA)}$

  We get:

  C = 2B 58 1C
  T = 6E D8 31 47 57 6C 33 A3 FF 8C 89 26 17 30 D5 A2

  So the encrypted response is:

C	T	SW1	SW2
2B 58 1C	**6E D8 31 47 57 6C 33 A3 FF 8C 89 26 17 30 D5 A2**	**90**	**00**

9.11.9 END ENCRYPTION

The unencrypted **END ENCRYPTION** command:

Command:

CLA	INS	P1	P2
C0	**12**	**00**	**00**

- Set **N** equal to **T** from the **ECHO** response:

  N = 6E D8 31 47 57 6C 33 A3 FF 8C 89 26 17 30 D5 A2

- Insert **Le' = 10**, **Lc' = 10**:

CLA	INS	P1	P2	Lc'	Le'
C0	**12**	**00**	**00**	**10**	**10**

- Set **H = CLA || INS || P1 || P2 || Lc' || Le' = C0 12 00 00 10 10**, and compute the Tag **T**:

  $\mathbf{CT = EAX.Encrypt_{K}^{NH}(“”)}$

  We get:

  T = CE C0 E4 7D 8B 47 F3 9B E8 E9 3D 5D ED 72 60 5D

  So the encrypted command is:

CLA	INS	P1	P2	Lc'
C0	12	00	00	10

T	Le'
CE C0 E4 7D 8B 47 F3 9B E8 E9 3D 5D ED 72 60 5D	10

10. The ZC-Basic Virtual Machine

Note: Throughout this chapter, **bold** numbers are hexadecimal.

10.1 Address Metrics

The ZC-Basic Virtual Machine has been implemented for BasicCards of widely differing storage capacity, as well as for a PC running Microsoft Windows. To describe this machine in its full generality, we will use various *address metrics* that have different values in different environments:

GASize The size of a *GenericAddress*, that can be used to address all data
RASize The size of a *RamAddress*, that can be used to address data in **Ram**
FOSize The size of a *FrameOffset*, for referencing stack-based data in a Procedure

10.2 The BasicCard Virtual Machine

10.2.1 The Enhanced BasicCard

Address metrics: **GASize = 2**
RASize = 1
FOSize = 1

The Enhanced BasicCard contains **100** bytes of RAM (= 256 in decimal), and up to **3FE0** bytes of EEPROM (= 16352 in decimal). Of this, the operating system uses the first **6B** bytes of RAM, and the first **15D** bytes of EEPROM. If the file system is not disabled, it requires **7** bytes of RAM, plus **6** bytes for each file slot. (Files and directories themselves are allocated from the **EEPHEAP** region.)

10.2.2 The Professional BasicCard

Address metrics: **GASize = 3** (**ZC7-**series from **REV D**) or **2** (all other versions)
RASize = 2
FOSize = 2 (**ZC7-**series from **REV D**) or **1** (all other versions)

The Professional BasicCard contains up to **12C0** bytes of RAM (= 4800 in decimal), and up to **12000** bytes of EEPROM (= 73728 in decimal). The amount of RAM and EEPROM used by the operating system varies from version to version, but the figures in **10.2.1 The Enhanced BasicCard** give a rough guide.

10.2.3 The MultiApplication BasicCard

Address metrics: **GASize = 2** (**ZC6-**series) or **3** (**ZC8-**series)
RASize = 2
FOSize = 1 (**ZC6-**series) or **2** (**ZC8-**series)

The MultiApplication BasicCard contains up to **12C0** bytes of RAM (= 4800 in decimal), and up to **12000** bytes of EEPROM (= 73728 in decimal). The amount of RAM and EEPROM used by the operating system varies from version to version, but the figures in **10.2.1 The Enhanced BasicCard** give a rough guide. When an Application's ZC-Basic code runs in the MultiApplication BasicCard, all addresses are virtual; this lets the operating system protect an Application from unauthorised access by other Applications in the same card.

10.2.4

10.2.5 Memory Layout in the BasicCard

RAM and EEPROM are divided into regions, usually in the following order:

	RAM Regions		EEPROM Regions
RAMSYS	System RAM	**EEPSYS**	System EEPROM
STACK	The P-Code stack	**STRVAL**	Single-to-String code*
RAMDATA	**Public** and **Static** data	**CMDTAB**	Command descriptor table
RAMHEAP	Run-time memory allocation	**PCODE**	The ZC-Basic program code
FILEINFO	Open file slots + file system workspace	**STRCON**	String constants
(**FRAME**)	Procedure frame (contained in **STACK**)	**KEYTAB**	Keys for encryption
		EEPDATA	**Eeprom** data
		EEPHEAP	Run-time memory allocation
		Libraries	Plug-In Libraries

* The **STRVAL** region is only present for older Enhanced BasicCard programs that use Single-to-String conversion.

The ZC-Basic compiler calculates how much static memory is required for each region, and assigns any remaining memory to **RAMHEAP** and **EEPHEAP**, for run-time memory allocation of strings, arrays, and files. The map file lists the sizes of all these regions – see **11.5 Map File Format**.

Certain Professional and MultiApplication BasicCards allocate their **EEPROM** regions in a different order; consult the map file for details.

10.3 The Terminal Virtual Machine

Address metrics: **GASize = 4**
RASize = 4
FOSize = 2

A Terminal program contains a **CODE** segment and a **DATA** segment, each of which may be up to 64 kilobytes long. The **CODE** segment contains only the **PCODE** region. The **DATA** segment contains RAM and EEPROM regions (see **2.2.4 Permanent Data** for the meaning of EEPROM data in a Terminal program). The regions occur in the following order (RAM before EEPROM):

	RAM Regions		EEPROM Regions
STACK	The P-Code stack	**EEPDATA**	**Eeprom** data
RAMSYS	System RAM	**EEPHEAP**	Run-time memory allocation
RAMDATA	**Public** and **Static** data		
RAMHEAP	Run-time memory allocation		
STRCON	String constants		
(**FRAME**)	Procedure frame (contained in **STACK**)		

10.4 The P-Code Stack

The P-Code Virtual Machine has three registers:

PC	Program counter (**GASize** bytes)
SP	Stack Pointer (**RASize** bytes)
FP	Frame Pointer (**RASize** bytes)

The P-Code stack grows upwards; the **SP** register contains the address of the first free byte on the stack. The stack contains four kinds of data:

- Command parameters, received from the I/O port (BasicCard only). These are located at the bottom of the stack.
- Procedure parameters and return addresses. Before a procedure is called, its parameters are pushed onto the P-Code stack. (If the procedure is a **Function**, space is reserved below the parameters for the function return value.)
- **FRAME** data, consisting of **Private** data and compiler-generated temporary variables. Each procedure has its own **FRAME** region, of a fixed size, that is allocated from the stack when the procedure is called. The **FP** register points to the base of the **FRAME** region.
- Intermediate results of computations. The Virtual Machine has no data registers; all computation is performed on the top of the P-Code stack.

The first P-Code instruction in a procedure is

ENTER *frame-size*

This instruction sets up the **FRAME** region as follows:

- Push **FP**
- Push **SP** + *frame-size* + size of SP (i.e. **SP** + *frame-size* + **RASize**)
- **FP** = **SP**
- **SP = SP +** *frame-size*

The last instruction in every procedure is

LEAVE

This undoes the effect of the **ENTER** instruction before returning to the caller:

- **SP = FP –** size of **FP** (i.e. **FP** – **RASize**)
- Pop **FP**
- Pop **PC**

10.5 Run-Time Memory Allocation

The Virtual Machine has two heaps for the run-time allocation of strings and arrays: **RAMHEAP** and **EEPHEAP**. Each is composed of variable-length blocks, that are either *allocated* or *free*; adjacent free blocks are concatenated as soon as they are created. In addition, an allocated block in **EEPHEAP** is either *permanent* or *temporary*. Each block consists of a *block header* followed by a *data area*. The block header contains the length of the data area, and one or two bits describing the block:

EEPHEAP block

<table><tr><th>F</th><th>T</th><th>Len</th></tr><tr><td colspan="3">Data area (Len bytes)</td></tr></table>

RAMHEAP block

<table><tr><th>F</th><th>Len</th></tr><tr><td colspan="2">Data area (Len bytes)</td></tr></table>

F = 1 if the block is free, **0** if the block is allocated.

T = 1 if the block is temporary, **0** if the block is permanent. A temporary block is automatically freed the next time the BasicCard is reset or the Terminal program is run.

A **RAMHEAP** block header is **RASize** bytes. An **EEPHEAP** block header is at least **GASize** bytes; in some cards with **GASize = 2**, it is 3 bytes.

Notes:

1. If **F** is **1**, then **T** is not used as a temporary block flag. This means that, although allocated blocks in **EEPHEAP** are limited to 16383 bytes in the case of a 2-byte header, a free block (and thus the total size of the heap) may be up to 32767 bytes long.

2. An Application in the MultiApplication BasicCard has a **RAMHEAP** region and an **EEPHEAP** region, like other cards. These regions are contained in the Application File. In addition, the operating system in the MultiApplication BasicCard has a global EEPROM heap, for Files, Components, and temporary blocks. See **5.2.4 Memory Allocation** for further information.

10.6 Data Types

The BasicCard Virtual Machine implements the following data types:

CHAR	1-byte unsigned integer
WORD	2-byte signed integer
LONG	4-byte signed integer
QUAD	8-byte signed integer
REAL	4-byte IEEE-format floating-point number
DOUBLE	8-byte IEEE-format floating-point number
STRING	See *Strings* below

These types correspond to the ZC-Basic data types **Byte**, **Integer**, **Long**, **Long64**, **Single**, **Double**, and **String** respectively. Arithmetic operations are provided for **WORD**, **LONG**, **QUAD**, **REAL**, and **DOUBLE** data; **CHAR** data must be converted to **WORD** before performing arithmetic.

Types **QUAD** and **DOUBLE** are available in Terminal programs, and **ZC7-** and **ZC8-**series BasicCards from **REV D**.

10.6.1 Strings

There are two types of string: variable-length and fixed-length.

- A variable-length string is a pointer of size **GASize**, to a Pascal-type string, which consists of a length field (one or two bytes) followed by the string contents.
- A fixed-length string is a sequence of characters, whose length is known at compile time.

The maximum length of a string (and the size of the length field in a variable-length string) depends on the operating system:

- 16384 bytes in a Terminal program (2-byte length field);
- 2048 bytes in **ZC7**- and **ZC8-** series BasicCards (2-byte length field);
- 254 bytes otherwise (1-byte length field).

If an operation would result in a string longer than this maximum, the result is truncated.

String variables take various forms, depending on the storage type:

Eeprom	A fixed-length **Eeprom** string variable is a sequence of characters in the **EEPDATA** region. A variable-length **Eeprom** string variable is a pointer of size **GASize**, in the **EEPDATA** region, to a Pascal-type string in the **EEPHEAP** region.
Public, **Static**	A fixed-length **Public** or **Static** string variable is a sequence of characters in the **RAMDATA** region. A variable-length **Public** or **Static** string variable is a pointer of size **GASize**, in the **RAMDATA** region, to a Pascal-type string, which may be in **RAMHEAP** or **EEPHEAP**. Strings are allocated from **RAMHEAP** if there is room, but if not they are allocated from **EEPHEAP**. In this case they are marked as temporary, so that they can be deleted when the BasicCard is reset or the Terminal program is restarted.
Private	A fixed-length **Private** string variable is a sequence of characters in the **FRAME** region. A variable-length **Private** string variable is a pointer of

size **GASize**, in the **FRAME** region, to a Pascal-type string, which may be in **RAMHEAP** or **EEPHEAP**.

String parameters A **String** parameter takes up **GASize+1** or **GASize+2** bytes on the stack: a one- or two-byte *length* followed by a *GenericAddress*. If the most significant byte of the *length* field is 255, the address is a handle, and points to a variable-length string variable; otherwise the address points directly to a fixed-length string. (This is the reason for the 254-byte length restriction on strings in the smaller BasicCards.)

10.7 P-Code Instructions

In this section, names in *italics* obey the following conventions:

- Initial characters *s* and *u* denote signed and unsigned values respectively.
- Initial character *r*, *d*, or second character *c*, *w*, *l*, *q*, denote **REAL**, **SINGLE**, **CHAR**, **WORD**, **LONG**, and **QUAD** data respectively.
- *GA* denotes a *GeneralAddress*, of size **GASize**.
- *RA* denotes a *RamAddress*, of size **RASize**.
- *FO* denotes a *FrameOffset* (signed by default), of size **FOSize**.
- *A* is the address of an array descriptor, of size **GASize**.
- *X$*, *Y$*, *Z$* are **STRING**s.

10.7.1 Miscellaneous Instructions

Name	*OpCode*	*Param*	*Description*
NOP	**00**		No operation
ADDSP	**01**	*FO*	**SP +=** *FO*. If *FO* **> 0**, 'pushed' bytes are initialised to zero.
DUP	**02**	*uFO*	Push the top *uFO* stack bytes
COMPL	**03**		Pop *slY* ; pop *slX* ; compare ; push for **WORD** comparison
RAND	**04**		Push a **LONG** random number
ERROR	**05**	*ucError*	Generate a P-Code error condition
SYSTEM	**06**	*ucSysCode*	Operating system call – see **10.8.5** .

10.7.2 Data Conversion Instructions

Name	*OpCode*	*Description*
CVTCW	**07**	Pop *ucX* ; *swY* = *ucX* ; push *swY*
CVTWC	**08**	Pop *swX* ; *ucY* = *swX* ; push *ucY*
CVTWL	**09**	Pop *swX* ; *slY* = *swX* ; push *slY*
CVTLW	**0A**	Pop *slX* ; *swY* = *slX* ; push *swY*

10.7.3 Data Access Instructions (Push and Pop)

Name	*OpCode*	*Param*	*Description*
PUCCB	**0B**	*ucConst*	Push constant **CHAR** *ucConst*
PUCWB	**0C**	*scConst*	Push constant *scConst* sign-extended to **WORD**
PUCWC	**0D**	*ucConst*	Push constant *ucConst* zero-extended to **WORD**
PUCWW	**0E**	*swConst*	Push constant **WORD** *swConst*
PURC	**0F**	*RA*	Push **CHAR** at address *RA*
PURW	**10**	*RA*	Push **WORD** at address *RA*
PURL	**11**	*RA*	Push **LONG** at address *RA*
PURS	**12**	*RA*	Push **STRING** at address *RA* (but see **ADDUW** below)
PUEC	**13**	*GA*	Push **CHAR** at address *GA*
PUEW	**14**	*GA*	Push **WORD** at address *GA*
PUEL	**15**	*GA*	Push **LONG** at address *GA*
PUES	**16**	*GA*	Push **STRING** at address *GA*
PUFC	**17**	*FO*	Push **CHAR** at address **FP** + *FO*
PUFW	**18**	*FO*	Push **WORD** at address **FP** + *FO*
PUFL	**19**	*FO*	Push **LONG** at address **FP** + *FO*
PUFS	**1A**	*FO*	Push **STRING** at address **FP** + *FO*
PUAF	**1B**	*FO*	Push **FP** + *FO* as **WORD**
PUAS	**1C**	*uFO*	Push **SP** – *uFO* as **WORD**
PUPS	**1D**	*FO*	Push 3-byte **STRING** parameter at address **FP** + *FO*
PUINC	**1E**		Pop *GA* ; push **CHAR** at address *GA*
PUINW	**1F**		Pop *GA* ; push **WORD** at address *GA*
PUINL	**20**		Pop *GA* ; push **LONG** at address *GA*
PORC	**21**	*RA*	Pop **CHAR** at address *RA*
PORW	**22**	*RA*	Pop **WORD** at address *RA*
PORL	**23**	*RA*	Pop **LONG** at address *RA*
POEC	**24**	*GA*	Pop **CHAR** at address *GA*
POEW	**25**	*GA*	Pop **WORD** at address *GA*
POEL	**26**	*GA*	Pop **LONG** at address *GA*
POFC	**27**	*FO*	Pop **CHAR** at address **FP** + *FO*
POFW	**28**	*FO*	Pop **WORD** at address **FP** + *FO*
POFL	**29**	*FO*	Pop **LONG** at address **FP** + *FO*
POINC	**2A**		Pop *GA* ; pop **CHAR** at address *GA*
POINW	**2B**		Pop *GA* ; pop **WORD** at address *GA*
POINL	**2C**		Pop *GA* ; pop **LONG** at address *GA*

10.7.4 Integer Arithmetic Instructions

Name	*OpCode*	*Description*
ADDUW	**12**	Pop *uwY* ; pop *uwX* ; push *uwX* + *uwY* (see *Note*)
ADDW	**2D**	Pop *swY* ; pop *swX* ; push *swX* + *swY*
ADDL	**2E**	Pop *slY* ; pop *slX* ; push *slX* + *slY*
SUBW	**2F**	Pop *swY* ; pop *swX* ; push *swX* – *swY*
SUBL	**30**	Pop *slY* ; pop *slX* ; push *slX* – *slY*
MULW	**31**	Pop *swY* ; pop *swX* ; push *swX* * *swY*
MULL	**32**	Pop *slY* ; pop *slX* ; push *slX* * *slY*
DIVW	**33**	Pop *swY* ; pop *swX* ; push *swX* / *swY*
DIVL	**34**	Pop *slY* ; pop *slX* ; push *slX* / *slY*
MODW	**35**	Pop *swY* ; pop *swX* ; push *swX* **Mod** *swY*
MODL	**36**	Pop *slY* ; pop *slX* ; push *slX* **Mod** *slY*
ANDW	**37**	Pop *uwY* ; pop *uwX* ; push *uwX* **And** *uwY*
ANDL	**38**	Pop *ulY* ; pop *ulX* ; push *ulX* **And** *ulY*
ORW	**39**	Pop *uwY* ; pop *uwX* ; push *uwX* **Or** *uwY*
ORL	**3A**	Pop *ulY* ; pop *ulX* ; push *ulX* **Or** *ulY*
XORW	**3B**	Pop *uwY* ; pop *uwX* ; push *uwX* **Xor** *uwY*
XORL	**3C**	Pop *ulY* ; pop *ulX* ; push *ulX* **Xor** *ulY*
NEGW	**3D**	Pop *swX* ; push –*swX*
NEGL	**3E**	Pop *slX* ; push –*slX*
ABSW	**3F**	Pop *swX* ; push **Abs(***swX***)**
ABSL	**40**	Pop *slX* ; push **Abs(***slX***)**
INCW	**41**	Pop *swX* ; push *swX* + 1
INCL	**42**	Pop *slX* ; push *slX* + 1
NOTW	**43**	Pop *uwX* ; push **Not(***uwX***)**
NOTL	**44**	Pop *ulX* ; push **Not(***ulX***)**

Note: The **ADDUW** instruction has the same OpCode (**12**) as the **PURSB** instruction. It is required to avoid **pcOverflow** errors during run-time address calculations in some of the Professional BasicCards, whose address space crosses the **7FFF/8000** boundary. In all such cards, a *RamAddress* is the same size as a *GeneralAddress*, so the **PUR*x*** and **POR*x*** instructions are not needed.

10.7.5 Program Control Instructions

(In the **ENTER** and **LEAVE** instructions, *F* denotes the size of the FP register, as defined in **10.4 The P-Code Stack**.)

Name	*OpCode*	*Param*	*Description*
CALL	**45**	*GA*	Procedure call or **GoSub**: push **PC+GASize+1** as *GA*; **PC** = *GA*
ENTER	**46**	*uFO*	Push **FP** ; push **SP** + *uFO* + **RASize** ; **FP = SP** ; **SP = SP** + *uFO*
LEAVE	**47**		Return from procedure: **SP = FP – RASize** ; pop **FP** ; pop **PC**
RETURN	**48**		Return from **GoSub**: pop **PC**
JUMP	**49**	*scDisp*	**PC = PC** + *scDisp* + 2
LJUMP	**4A**	*GA*	**PC** = *GA*
JZRW	**4B**	*scDisp*	Pop *swX* ; if *swX* = 0 then **PC = PC** + *scDisp* + 2
JNZW	**4C**	*scDisp*	Pop *swX* ; if *swX* <> 0 then **PC = PC** + *scDisp* + 2
JEQW	**4D**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* = *swY* then **PC = PC** + *scDisp* + 2
JNEW	**4E**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* <> *swY* then **PC = PC** + *scDisp* + 2
JLEW	**4F**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* ≤ *swY* then **PC = PC** + *scDisp* + 2
JGTW	**50**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* > *swY* then **PC = PC** + *scDisp* + 2
JGEW	**51**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* ≥ *swY* then **PC = PC** + *scDisp* + 2
JLTW	**52**	*scDisp*	Pop *swY* ; pop *swX* ; if *swX* < *swY* then **PC = PC** + *scDisp* + 2
LOOP	**53**	*scDisp*	Pop *swX* ; if *swX* ≥ 0 then execute **JLEW** else execute **JGEW**
EXIT	**54**		Exit the Virtual Machine

10.7.6 Array Instructions

Name	*OpCode*	*Param*	*Description*
ARRAY	**55**		Pop *A* ; pop subscript *swIr* for each dimension *r*, in reverse order; push address of array element *A* (*swI1, swI2, . . . , swIn*)
CHKDIM	**56**	*ucNdims*	Pop *A* ; push *A* ; if Dim(*A*) <> *ucNdims* then execute **ERROR 0C**
ALLOCA	**57**		Pop *A* ; pop array bounds for each dimension *r*, in reverse order; allocate data area of *A* and initialise all elements to 0
FREEA	**58**		Pop *A* ; if **Dynamic** then deallocate *A*, else set all elements of *A* to 0
FREEA$	**59**		Pop string array *A* ; free all strings in *A* ; if **Dynamic** then deallocate *A*
BOUNDA	**5A**		Pop *swHi* ; pop *swLo* ; push **400***swLo* + (*swHi* – *swLo*) as **WORD**
LBOUND	**5B**		Pop *A* ; pop *ucDim* ; push lower bound of subscript *ucDim* as **WORD**
UBOUND	**5C**		Pop *A* ; pop *ucDim* ; push upper bound of subscript *ucDim* as **WORD**

10.7.7 String Instructions

Name	*OpCode*	*Description*
COPY$	**5D**	Pop *X$* ; pop *Y$* ; *X$* = *Y$*
FREE$	**5E**	Pop handle *GA* to variable-length string *X$* ; *X$* = empty string
ADD$	**5F**	Pop *X$* ; pop *Z$* ; pop *Y$* ; *X$* = *Y$* + *Z$*
MID$	**60**	Pop *swLen* ; pop *swStart* ; pop *X$* ; push **Mid$(***X$*, *swStart*, *swLen***)**
LEFT$	**61**	Pop *swLen* ; pop *X$* ; push **Left$(***X$*, *swLen***)**
RIGHT$	**62**	Pop *swLen* ; pop *X$* ; push **Right$(***X$*, *swLen***)**
LTRIM$	**63**	Pop *X$* ; push **LTrim$(***X$***)**
RTRIM$	**64**	Pop *X$* ; push **RTrim$(***X$***)**
UCASE$	**65**	Pop *X$* ; pop *Y$* ; *X$* = **UCase$(***Y$***)**
LCASE$	**66**	Pop *X$* ; pop *Y$* ; *X$* = **LCase$(***Y$***)**
STRING$	**67**	Pop *X$* ; pop *ucChar* ; pop *swLen* ; *X$* = **String$(***swLen*, *ucChar***)**
STRL$	**68**	Pop *X$* ; pop *slX* ; *X$* = **Str$(***slX***)**
HEX$	**69**	Pop *X$* ; pop *slX* ; *X$* = **Hex$(***slX***)**
ASC$	**6A**	Pop *X$* ; push **Asc(***X$***)** as **CHAR**
LEN$	**6B**	Pop *X$* ; push **Len(***X$***)** as **CHAR**
COMP$	**6C**	Pop *Y$* ; pop *X$* ; compare ; push for **WORD** comparison
VALL$	**6D**	Pop *X$* ; *slVal* = **Val&(***X$, ucLen***)** ; push *slVal* ; push 1- or 2-byte *Len*
VALHL$	**6E**	Pop *X$* ; *slVal* = **ValH(***X$, ucLen***)** ; push *slVal* ; push 1- or 2-byte *Len*

10.7.8 Data Initialisation Instructions

Name	*OpCode*	*Params*	*Description*
RDATA	**6F**	*RA*, *ucLen*, *data*	Copy *data* (*ucLen* bytes) to address *ucAddr*
FDATA	**70**	*FO*, *ucLen*, *data*	Copy *data* (*ucLen* bytes) to address **FP** + *FO*

10.7.9 Floating-Point Instructions

Name	*OpCode*	*Description*
COMPR	**71**	Pop *rY* ; pop *rX* ; compare ; push for **WORD** comparison
CVTWR	**72**	Pop *swX* ; push *swX* as **REAL**
CVTRW	**73**	Pop *rX* ; push *rX* as **WORD**
CVTLR	**74**	Pop *slX* ; push *slX* as **REAL**
CVTRL	**75**	Pop *rX* ; push *rX* as **LONG**
ADDR	**76**	Pop *rY* ; pop *rX* ; push *rX* + *rY*
SUBR	**77**	Pop *rY* ; pop *rX* ; push *rX* – *rY*
MULR	**78**	Pop *rY* ; pop *rX* ; push *rX* * *rY*
DIVR	**79**	Pop *rY* ; pop *rX* ; push *rX* / *rY*
NEGR	**7A**	Pop *rX* ; push –*rX*
ABSR	**7B**	Pop *rX* ; push **Abs(***rX***)**
SQRTR	**7C**	Pop *rX* ; push **Sqrt(***rX***)**
STRR$	**7D**	Pop *X$* ; pop *rX* ; *X$* = **Str$(***rX***)**
VALR$	**7E**	Pop *X$* ; *rVal* = **Val!(***X$, ucLen***)** ; push *rVal* ; push *ucLen*

10.7.10 The XMIT Command Call Instruction

Name	*OpCode*	*Params*	*Description*
XMIT	**7F**	*ucType*, *ucLen*	Send command and process response

This instruction is available only in a Terminal program. *ucType* contains command length information *LengthInfo* in the top four bits, and command type information *TypeInfo* in the bottom four bits. Before this instruction is executed, a command must be pushed onto the P-Code stack:

CLA	**INS**	**P1**	**P2**	**Lc**	**IDATA** padded to *ucLen* bytes	**Le**

Here, **Lc** and **Le** are one byte if *LengthInfo* is zero, otherwise two bytes. The commmand is sent using extended **Lc/Le** or not, according to *LengthInfo*:

LengthInfo	
0	Don't use extended **Lc/Le** for this command
4	Use extended **Lc/Le** for this command if necessary
8	Always use extended **Lc/Le** for this command

The command is transmitted according to *TypeInfo*, as follows:

TypeInfo	
0	Send **Lc** bytes in **IDATA** (no **Le**)
1	Send **Lc** bytes in **IDATA**, followed by **Le**
2	The top 3 or 4 bytes of the **IDATA** field contain a variable-length string parameter *X$*. Send *ucLen* – 3 (resp. *ucLen* – 4) bytes in **IDATA**, followed by *X$*.
3	The same as *ucType* = **2**, with **Le** appended to **IDATA**.
4	The top 3 or 4 bytes of the **IDATA** field contain a variable-length string parameter *X$*. Send up to **Lc** bytes of (*ucLen* – 3 (resp. *ucLen* – 4) bytes followed by *X$*).
5	The same as *ucType* = **4**, with **Le** appended to **IDATA**.
7	The same as *ucType* = **3**, but *X$* was passed **ByVal**.
9	The same as *ucType* = **5**, but *X$* was passed **ByVal**.

10.7.11 Abbreviated Instructions

Instructions from **80** to **FF** are single-byte abbreviations of 2-byte **PUF***x* / **POF***x* instructions. For example, **PUFL05** (instruction **B5**) is an abbreviation of **PUFL 05**. A positive frame offset refers to a **Private** data item, and a negative frame offset (denoted *xx* in the table) refers to a procedure parameter. If the frame offset is negative, then the meaning of an abbreviated instruction depends on a *CallOverhead* parameter **COSize**, which is:

- **0C** (decimal 12) in a Terminal program;
- **04** if **GASize** is equal to 3;
- **02** otherwise.

Name	*OpCode*	*Description*
PUFW*xx* **– PUFW***xx*	**80-8F**	Push **WORD** at address **FP – (91 – OpCode) – COSize**
PUFW00 – PUFW0F	**90-9F**	Push **WORD** at address **FP + (OpCode – 90)**
PUFL*xx* **– PUFL***xx*	**A0-AF**	Push **LONG** at address **FP – (B3 – OpCode) – COSize**
PUFL00 – PUFL0F	**B0-BF**	Push **LONG** at address **FP + (OpCode – B0)**
POFW*xx* **– POFW***xx*	**C0-CF**	Pop **WORD** at address **FP – (D1 – OpCode) – COSize**
POFW00 – POFW0F	**D0-DF**	Pop **WORD** at address **FP + (OpCode – D0)**
POFL*xx* **– POFL***xx*	**E0-EF**	Pop **LONG** at address **FP – (F3 – OpCode) – COSize**
POFL00 – POFL0F	**F0-FF**	Pop **LONG** at address **FP + (OpCode – F0)**

10.8 64-Bit Extensions

Several major enhancements were made to the Virtual Machine for the **ZC7-** and **ZC8-**series **REV D** BasicCards. Chiefly, 64-bit integer and floating-point arithmetic is supported in these cards. In addition, these cards have a 24-bit address space, to allow for more than 64KB of EEPROM.

These enhancements, which are also available in Terminal programs, are known as *64-Bit Extensions*. In a Virtual Machine with 64-Bit Extensions, instruction **5A** (**BOUNDA** in Enhanced BasicCards) is the first byte of a 2-byte Extended Instruction.

10.8.1 64-Bit Extensions – QUAD and DOUBLE Instructions

Name	*OpCode*	*Param*	*Description*
COMPQ	**5A00**		Pop *sqY* ; pop *sqX* ; compare ; push for **WORD** comparison
CVTLQ	**5A01**		Pop *slX* ; *sqY* = *slX* ; push *sqY*
CVTQL	**5A02**		Pop *sqX* ; *slY* = *sqX* ; push *slY*
PURQ	**5A03**	*RA*	Push **QUAD** at address *RA*
PUEQ	**5A04**	*GA*	Push **QUAD** at address *GA*
PUFQ	**5A05**	*FO*	Push **QUAD** at address **FP +** *FO*
PUINQ	**5A06**		Pop *GA* ; push **LONG** at address *GA*
PORQ	**5A07**	*RA*	Pop **QUAD** at address *RA*
POEQ	**5A08**	*GA*	Pop **QUAD** at address *GA*
POFQ	**5A09**	*FO*	Pop **QUAD** at address **FP +** *FO*
POINQ	**5A0A**		Pop *GA* ; pop **QUAD** at address *GA*
ADDQ	**5A0B**		Pop *sqY* ; pop *sqX* ; push *sqX* + *sqY*
SUBQ	**5A0C**		Pop *sqY* ; pop *sqX* ; push *sqX* – *sqY*
MULQ	**5A0D**		Pop *sqY* ; pop *sqX* ; push *sqX* * *sqY*
DIVQ	**5A0E**		Pop *sqY* ; pop *sqX* ; push *sqX* / *sqY*
MODQ	**5A0F**		Pop *sqY* ; pop *sqX* ; push *sqX* **Mod** *sqY*
ANDQ	**5A10**		Pop *sqY* ; pop *sqX* ; push *sqX* **And** *sqY*
ORQ	**5A11**		Pop *sqY* ; pop *sqX* ; push *sqX* **Or** *sqY*
XORQ	**5A12**		Pop *sqY* ; pop *sqX* ; push *sqX* **Xor** *sqY*
NEGQ	**5A13**		Pop *sqX* ; push *–sqX*
ABSQ	**5A14**		Pop *sqX* ; push **Abs(***sqX***)**
INCQ	**5A15**		Pop *sqX* ; push *sqX* + 1
NOTQ	**5A16**		Pop *sqX* ; push **Not** *sqX*
CVTWQ	**5A17**		Pop *swX* ; push *swX* as **QUAD**
CVTQW	**5A18**		Pop *sqX* ; push *sqX* as **WORD**
CVTWD	**5A19**		Pop *swX* ; push *swX* as **DOUBLE**
CVTDW	**5A1A**		Pop *dX* ; push *dX* as **WORD**
CVTQR	**5A1B**		Pop *sqX* ; push *sqX* as **REAL**
CVTRQ	**5A1C**		Pop *rX* ; push *rX* as **QUAD**
CVTRD	**5A1D**		Pop *rX* ; push *rX* as **DOUBLE**
CVTDR	**5A1E**		Pop *dX* ; push *dX* as **REAL**
STRQ$	**5A1F**		Pop *X$* ; pop *sqX* ; *X$* = **Str$(***sqX***)**
HEXQ$	**5A20**		Pop *X$* ; pop *sqX* ; *X$* = Hex$(*sqX*)
VALQ$	**5A21**		Pop *X$* ; *sqVal* = **Val^(***X$, ucLen***)** ; push *sqVal* ; push 2-byte *Len*
VALHQ$	**5A22**		Pop *X$* ; *sqVal* = **ValH^(***X$, ucLen***)** ; push *sqVal* ; push 2-byte *Len*
COMPD	**5A23**		Pop d*Y* ; pop d*X* ; compare ; push for **WORD** comparison

Name	*OpCode*	*Param*	*Description*
CVTLD	**5A24**		Pop *slX* ; push *slX* as **DOUBLE**
CVTDL	**5A25**		Pop *dX* ; push *dX* as **LONG**
CVTQD	**5A26**		Pop *sqX* ; push *sqX* as **DOUBLE**
CVTDQ	**5A27**		Pop *dX* ; push *dX* as **QUAD**
ADDD	**5A28**		Pop *dY* ; pop *dX* ; push *dX* + *dY*
SUBD	**5A29**		Pop *dY* ; pop *dX* ; push *dX* – *dY*
MULD	**5A2A**		Pop *dY* ; pop *dX* ; push *dX* * *dY*
DIVD	**5A2B**		Pop *dY* ; pop *dX* ; push *dX* / *dY*
NEGD	**5A2C**		Pop *dX* ; push –*dX*
ABSD	**5A2D**		Pop *dX* ; push **Abs(***dX***)**
SQRTD	**5A2E**		Pop *dX* ; push **Sqrt(***dX***)**
STRD$	**5A2F**		Pop *X$* ; pop *dX* ; *X$* = **Str$(***dX***)**
VALD$	**5A30**		Pop *X$* ; *dVal* = **Val#(***X$, ucLen***)** ; push *dVal* ; push 2-byte *Len*

10.8.2 64-Bit Extensions – GenericAddress Instructions

Name	*OpCode*	*Param*	*Description*
CVTLA	**5A31**		Pop *slX* ; push *slX* as *GenericAddress*
PUCA	**5A32**	*GA*	Push *GA*
PURA	**5A33**	*RA*	Push *GenericAddress* at address *RA*
PUEA	**5A34**	*GA*	Push *GenericAddress* at address *GA*
PUFA	**5A35**	*FO*	Push *GenericAddress* at address **FP** + *FO*
PUINA	**5A36**		Pop *GA* ; push *GenericAddress* at address *GA*
PORA	**5A37**	*RA*	Pop *GenericAddress* at address *RA*
POEA	**5A38**	*GA*	Pop *GenericAddress* at address *GA*
POFA	**5A39**	*FO*	Pop *GenericAddress* at address **FP** + *FO*
POINA	**5A3A**		Pop *GA* ; pop *GenericAddress* at address *GA*
ADDA	**5A3B**		Pop *GAX* ; pop *GAY* ; push *GAX* + *GAY*
ADDAW	**5A3C**		Pop *swX* ; pop *GA*; push *GA* + *swX*
ADDAL	**5A3D**		Pop *slX* ; pop *GA*; push *GA* + *slX*

10.8.3 64-Bit Extensions – Large Array Instructions

Name	*OpCode*	*Param*	*Description*
LARRAY	**5A3E**		Pop *A* ; pop subscript *slIr* for each dimension *r*, in reverse order; push address of array element *A* (*slI1, slI2, . . . , slIn*)
LALLOCA	**5A3F**		Pop *A* ; pop **LONG** array bounds for each dimension *r*, in reverse order; allocate data area of *A* and initialise all elements to 0

*10.8.4 64-Bit Extensions – String **Xor** Instruction*

Name	*OpCode*	*Param*	*Description*
XOR$	**5A40**		Pop *X$* ; pop *Z$* ; pop *Y$* ; *X$* = *Y$* **Xor** *Z$*

10.8.5 64-Bit Extensions – Abbreviated Instructions

See **10.7.11 Abbreviated Instructions** for the meaning of **COSize**.

Name	*OpCode*	*Description*
PUFQ*xx* **– PUFQ***xx*	**A5A0-A5AF**	Push **QUAD** at address **FP – (A5B7 – OpCode) – COSize**
PUFQ00 – PUFQ0F	**A5B0-A5BF**	Push **QUAD** at address **FP + (OpCode – A5B0)**
POFQ*xx* **– POFQ***xx*	**A5E0-A5EF**	Push **QUAD** at address **FP – (A5F7 – OpCode) – COSize**
POFQ00 – POFQ0F	**A5F0-A5FF**	Push **QUAD** at address **FP + (OpCode – A5F0)**

10.9 The SYSTEM Instruction

The **SYSTEM** P-Code instruction (OpCode **06**) calls an operating system function, according to the first parameter, *SysCode:*

- a System Library procedure if *SysCode* ≥ **C0** – see **10.9.4 System Library Procedures**.
- a **FILE SYSTEM** function if **80** ≤ *SysCode* ≤ **BF** – see **10.9.3 FILE SYSTEM Functions**.
- otherwise, a built-in System Function, as described in the following paragraphs.

10.9.1 SYSTEM Functions in the BasicCard

OpCode	*SysCode*	*Name*	
06	**00**	**WTX**	Send a Waiting Time Extension request
06	**03**	**EnableKey**	Enable or disable a cryptographic key or its error counter
06	**40**	**Certificate**	Calculate a cryptographic certificate
06	**41**	**DES**	DES block encryption primitives
06	**4C**	**EnableOvCheck**	Enable overflow checking (the default)
06	**4D**	**DisableOvCheck**	Disable overflow checking
06	**55**	**Key**	Built-in **Key()** function
06	**58**	**Shift**	Shift/rotate operator

10.9.2 SYSTEM Functions in the Terminal

OpCode	*SysCode*	*Name*	
06	**00**	**WTX**	Give the card more time
06	**40**	**Certificate**	Calculate a cryptographic certificate
06	**41**	**DES**	Des block encryption primitives
06	**42**	**Cls**	Clear the screen
06	**43**	**UpdateScreen**	Update the screen
06	**44**	**InKey$**	Check for keyboard input
06	**45**	**CardReader**	Look for a card reader
06	**46**	**CardInReader**	Check whether a card is in the reader
06	**47**	**ResetCard**	Reset the card in the card reader
06	**48**	**WriteEeprom**	Write EEPROM data back to the image file
06	**49**	**KeyFile**	Load a key file
06	**4A**	**EnableEncrypt**	Enable auto-encryption (the default)
06	**4B**	**DisableEncrypt**	Disable auto-encryption
06	**4C**	**EnableOvCheck**	Enable overflow checking (the default)
06	**4D**	**DisableOvCheck**	Disable overflow checking
06	**4E**	**Time$**	Date and time as e.g. “Wed Jun 20 15:50:35 1998”
06	**4F**	**ChDrive**	Change the current disk drive
06	**50**	**CurDrive**	Retrieve the current disk drive
06	**51**	**LongSeed**	Seed the random number generator with a **LONG** value
06	**52**	**StringSeed**	Seed the random number generator with a **STRING**
06	**53**	**OpenLogFile**	Start logging of I/O to file
06	**54**	**CloseLogFile**	End logging of I/O to file
06	**56**	**PcscCount**	Number of configured PC/SC card readers
06	**57**	**PcscReaderName**	Name of a PC/SC card reader
06	**58**	**Shift**	Shift/rotate operator
06	**59**	**CloseCardReader**	Close the current card reader

10.9.3 FILE SYSTEM Functions

The file system functionality in the ZC-Basic interpreter is implemented through the **SYSTEM** P-Code instruction. Such **FILE SYSTEM** commands all have **80** ≤ *SysCode* ≤ **BF**:

OpCode	*SysCode*	*Name*	
06	**80**	**MkDir**	Create a directory
06	**81**	**RmDir**	Delete a directory
06	**82**	**ChDir**	Change the current directory
06	**83**	**CurDir**	Retrieve the current directory
06	**84**	**DirCount**	Count the filenames that match a wild-card spec
06	**85**	**DirFile**	Return the *n*th matching filename
06	**86**	**EraseFile**	Delete a data file
06	**87**	**RenameFile**	Rename or move a file or directory
06	**88**	**OpenFile**	Open a file
06	**89**	**OpenFreeFile**	Open a file after finding a free file slot for it
06	**8A**	**CloseFile**	Close a file
06	**8B**	**CloseAll**	Close all files
06	**8C**	**FreeFile**	Find a free file slot
06	**8D**	**FileLength**	Return the length of an open file
06	**8E**	**GetFilepos**	Return the read/write pointer of an open file
06	**8F**	**SetFilepos**	Set the read/write pointer of an open file
06	**90**	**EOF**	Return **True** if at the end of an open file
06	**91**	**Get**	Read from a binary file
06	**92**	**GetPos**	**Get** after setting the read/write pointer
06	**93**	**Put**	Write to a binary file
06	**94**	**PutPos**	**Put** after setting the read/write pointer
06	**95**	**StartInput**	Set the counter of matched input items to 0
06	**96**	**EndInput**	Return the counter of matched input items
06	**97**	**Read**	Read a specified number of bytes from a sequential file
06	**98**	**ReadLong**	Read a formatted **LONG** value from a sequential file
06	**99**	**ReadSingle**	Read a formatted **REAL** value from a sequential file
06	**9A**	**ReadString**	Read a formatted **STRING** from a sequential file
06	**9B**	**ReadBlock**	Read a formatted fixed-size block from a sequential file
06	**9C**	**ReadLine**	Read a line from a sequential file
06	**9D**	**WriteLong**	Write a formatted **LONG** value to a sequential file
06	**9E**	**WriteSingle**	Write a formatted **REAL** value to a sequential file
06	**9F**	**WriteString**	Write a formatted **STRING** to a sequential file
06	**A0**	**PrintLong**	Write an ASCII **LONG** value to a sequential file
06	**A1**	**PrintSingle**	Write an ASCII **REAL** value to a sequential file

OpCode	*SysCode*	*Name*	
06	**A2**	**PrintString**	Write an ASCII **STRING** to a sequential file
06	**A3**	**PrintSpaces**	Write a specified number of spaces to a sequential file
06	**A4**	**PrintTab**	Advance to the next 14-character output field
06	**A5**	**SetColumn**	Advance to a specified output column
06	**A6**	**PrintNewLine**	Print a new-line character
06	**A7**	**LockFile**	Set the access conditions on a file or directory
06	**A8**	**GetLocks**	Retrieve the access conditions on a file or directory
06	**A9**	**GetAttr**	Retrieve the attributes of a file or directory
06	**AA**	**SetAttr**	Set the attributes of a file or directory (Terminal only)
06	**AB**	**ReadStringExtended**	**ReadString** using extended **Lc/Le** (Terminal only)
06	**AC**	**ReadLineExtended**	**ReadLine** using extended **Lc/Le** (Terminal only)
06	**AD**	**ReadLong64**	Read a formatted **QUAD** value from a sequential file
06	**AE**	**WriteLong64**	Write a formatted **QUAD** value to a sequential file
06	**AF**	**PrintLong64**	Write an ASCII **QUAD** value to a sequential file
06	**B0**	**ReadDouble**	Read a formatted **DOUBLE** value from a sequential file
06	**B1**	**WriteDouble**	Write a formatted **DOUBLE** value to a sequential file
06	**B2**	**PrintDouble**	Write an ASCII **DOUBLE** value to a sequential file
06	**B3**	**OpenFileAligned**	**OpenFile** with **Align** parameter
06	**B4**	**OpenFreeFileAligned**	**OpenFreeFile** with **Align** parameter

10.9.4 System Library Procedures

Values of *SysCode* between **C0** and **FF** are reserved for System Library procedures – see **3.14.2 System Library Procedures**. For details of which codes are assigned to which procedures, see the individual *Library*.**def** files supplied with ZeitControl's development software.

11. Output File Formats

This chapter describes the formats of the various output files generated by the ZC-Basic compiler:

- Image file: program and data in binary format, for use by **ZCMSim** and **BCLoad** programs.
- Debug file: symbolic debugging information, for the **ZCMDTerm** and **ZCMDCard** debuggers.
- Application file: selectable Application file in the MultiApplication BasicCard
- List file: source program, compiled P-Code, and data in human-readable text format.
- Map file: the addresses of all symbols in the program, ordered by name and by location.

Note: Throughout this chapter, **bold** numbers are hexadecimal.

11.1 ZeitControl Image File Format

Debug and Image files consist of Sections, each of which starts with a 4-byte ASCII name, followed by a 4-byte section length. In an Image file, Sections are guaranteed to occur in the following order:

For a BasicCard program:

'ZCIF'	Signature Section – "ZeitControl Image File"
'VERS'	Version Section – File format version
'VMTP'	Virtual Machine Type Section – target machine
'CONF'	Configuration File Section (Professional BasicCard only)
'EEPR'	EEPROM Image Section – **EEPSYS**, **CMDTAB**, **PCODE**, **STRCON**, **KEYTAB**, **EEPDATA**, and **EEPHEAP** regions (absent for MultiApplication BasicCard)
'RELO'	Library relocations (certain Enhanced BasicCard versions)
'LOAD'	Program Load Section, containing the commands to download to the BasicCard
'CERT'	Code Certification Section (certain Enhanced BasicCard versions)

For a Terminal program:

'ZCIF'	Signature Section – "ZeitControl Image File"
'VERS'	Version Section – File format version
'VMTP'	Virtual Machine Type Section – target machine
'CODE'	P-Code Section – Contents of **PCODE** region
'DATA'	Data Section – **RAMSYS**, **STRCON**, **RAMDATA**, and **RAMHEAP** regions
'EEPR'	EEPROM Image Section – **EEPDATA** and **EEPHEAP** regions

Numerical 2-byte and 4-byte fields are stored lsb to msb, Intel-style (or Little-Endian). This is in contrast to the Virtual Machine, which is Big-Endian.

Some sections contain string tables. A string table consists of consecutive null-terminated strings. Whenever a name occurs in a Section field, it is to be interpreted as an offset into the string table of the current Section.

11.1.1 Signature Section

Length	
4	**'ZCIF'** ("ZeitControl Image File")
4	Total length of all remaining sections (= file length – 8)

11.1.2 Version Section

Length	
4	**‘VERS’**
4	Section length = **04**
1	Major version of software that created this file
1	Minor version of software that created this file
1	Major version of oldest software compatible with this file
1	Minor version of oldest software compatible with this file

11.1.3 Virtual Machine Type Section

Length	
4	**‘VMTP’**
4	Section length *len*
1	*MachineType*
1	*MachineSubtype*
len-2	*FurtherData*

If the image file was created for a Terminal program:
MachineType = *MachineSubtype* = **0**
FurherData: **TerminalMemorySize As Long**
ScreenWidth As Byte (optional)
ScreenHeight As Byte (optional)

If the image file was created for an Enhanced or Professional BasicCard **ZC***x.y*:
MachineType = *x*, *MachineSubtype* = *y*
FurtherData, if present, is an ASCII string containing the version ID of the card

If the image file was created for a **ZC6**- or **ZC8**-series MultiApplication BasicCard:
MachineType = **6** or **8**, *MachineSubtype* = **0**
FurtherData is empty

11.1.4 Configuration File Section

Length	
4	**‘CONF’**
4	Section length *len*
len	Full path name of **.ZCF** BasicCard Configuration File, in Unicode format

This section is present for Professional BasicCards and certain Enhanced BasicCards.

11.1.5 P-Code Section (Terminal only)

Length	
4	**‘CODE’**
4	Section length *len*
4	Program entry point
len-4	P-Code. The P-Code in the Terminal starts at address **00000000**.

11.1.6 Data Section (Terminal only)

Length	
4	**'DATA'**
4	Section length
4	Start address of RAM data
4	Length of RAM data
4	Number of records n
4	Start address of record **0**
4	Length len_0 of record **0**
len_0	Contents of record **0**
. . .	
4	Start address of record $n-1$
4	Length len_{n-1} of record $n-1$
len_{n-1}	Contents of record $n-1$

All RAM bytes not contained in a record are initialised to **00**.

The Data Section contains the **RAMSYS**, **STRCON**, **RAMDATA**, and **RAMHEAP** regions.

11.1.7 EEPROM Image Section

Length	
4	**'EEPR'**
4	Section length
4	Start address of EEPROM data
4	Length of EEPROM data
4	Number of records n
4	Start address of record **0**
4	Length len_0 of record **0**
len_0	Contents of record **0**
. . .	
4	Start address of record $n-1$
4	Length len_{n-1} of record $n-1$
len_{n-1}	Contents of record $n-1$

All EEPROM bytes not contained in a record must be initialised to **FF**.

In the Terminal, the EEPROM Image Section contains just the **EEPDATA** and **EEPHEAP** regions. In the BasicCard, it contains the **EEPSYS**, **CMDTAB**, **PCODE**, **STRCON**, **KEYTAB**, **EEPDATA**, and **EEPHEAP** regions.

11.1.8 Relocation Section

Length	
4	**'RELO'**
4	Section length
4	Number of relocations *n*
4	Length len_0 of relocation **0**
4	Address of relocation **0**
len_0	Contents of relocation **0**
. . .	
4	Length len_{n-1} of relocation $n - 1$
4	Start address of relocation $n - 1$
len_{n-1}	Contents of relocation $n - 1$

This section is used to perform Plug-In Library relocations in certain Enhanced BasicCards. It is only used for simulated cards, and not required when loading a real BasicCard.

11.1.9 Program Load Section (Single-Application BasicCards)

Length	
4	**'LOAD'**
4	Section length
1	State of BasicCard after download (from **#State** directive or **–S** parameter)
1	Size of an EEPROM address in the card, in bytes (2 or 3)
4	Number n_{WE} of **WRITE EEPROM** commands
4	Number n_{CRC} of **EEPROM CRC** commands
4	Address of **WRITE EEPROM** command **0**
4	Length len_0 of **WRITE EEPROM** command **0**
len_0	Contents of **WRITE EEPROM** command **0**
. . .	
4	Address of **WRITE EEPROM** command $n_{WE} - 1$
4	Length len_{n-1} of **WRITE EEPROM** command $n_{WE} - 1$
len_{n-1}	Contents of **WRITE EEPROM** command $n_{WE} - 1$
4	Address of **EEPROM CRC** command **0**
4	Length of **EEPROM CRC** command **0**
2	CRC of **EEPROM CRC** command **0**
. . .	
4	Address of **EEPROM CRC** command $n_{CRC} - 1$
4	Length of **EEPROM CRC** command $n_{CRC} - 1$
2	CRC of **EEPROM CRC** command $n_{CRC} - 1$

11.1.10 Application Load Section (MultiApplication BasicCard)

Length	
4	**'LOAD'**
4	Section length
1	State of BasicCard after download (from **#State** directive or **–S** parameter)
1	Loader Action code
1	Loader Action subcode
	Loader Action data
. . .	
1	Loader Action code
1	Loader Action subcode
	Loader Action data

A Debug File contains source file information between the Loader Action subcode and the Loader Action data. It consists of three 4-byte fields: File number, Line number, and File position.

A Loader Action code other than **20** is an instruction to the Application Loader. In this case, the Loader Action data consists of the number of parameters (1 byte), followed by a Parameter Field for each parameter. A Parameter Field consists of *ParamType* (1 byte), followed by the value of the Parameter:

ParamType	*Meaning*	*Format*
00	**Byte**	1 byte
01	**Integer**	2 bytes, lsb first
02	**Long**	4 bytes, lsb first
04	**String**	*len* (4 bytes) followed by *val* (*len* bytes)
10	**ctFile**	4-byte Reference number of a Component of type File
20	**ctAcr**	4-byte Reference number of a Component of type ACR
30	**ctPrivilege**	4-byte Reference number of a Component of type Privilege
40	**ctFlag**	4-byte Reference number of a Component of type Flag
70	**ctKey**	4-byte Reference number of a Component of type Key

The following table gives the parameter types of each Application Loader instruction:

Code	*Subcode*	
10	**82**	Push current directory and change directory (*Directory* **As String**)
10	**83**	Pop current directory (no parameters)
10	**49**	**LCReadKeyFile** (*KeyFile* **As String**)
C0	**10**	**LCStartEncryption** (*Key* **As ctKey**, *Algorithm* **As Byte**)
C0	**12**	**LCEndEncryption** (no parameters)
C0	**42**	**LCExternalAuthenticate** (*Key* **As ctKey**, *Algorithm* **As Byte**)
C0	**44**	**LCInternalAuthenticate** (*Key* **As ctKey**, *Algorithm* **As Byte**)
C0	**46**	**LCVerify** (*Key* **As ctKey**)
C0	**92**	**LCGrantPrivilege** (*Privilege* **As ctPrivilege**, *File* **As ctFile**)

Code	*Subcode*	
C0	**94**	**LCAuthenticateFile** (*Key* **As ctKey**, *Algorithm* **As Byte**, *File* **As ctFile**, *Signature* **As String**)
C0	**98**	**LCLoadSequence** (*Phase* **As Byte**)
C0	**9A**	**LCStartSecureTransport** (*Key* **As ctKey**, *Algorithm* **As Byte**, *Nonce* **As String**)
C0	**9B**	**LCEndSecureTransport** (no parameters)
C0	**9C**	**LCCheckSerialNumber** (*SerialNumber* **As String**)
C0	**BC**	**LCWriteCardConfig** (*Tag* **As Byte**, *Data* **As String**)

A Loader Action code of **20** is a Component Action:

Code	*Subcode*	
20	**10**	File Action
20	**20**	ACR Action
20	**30**	Privilege Action
20	**40**	Flag Action
20	**50**	Directory Action
20	**70**	Key Action

Component Action data begins with the following fields, common to all Component types:

Ref	Component Reference (4 bytes)
Create	Create Option (1 byte): **0/1/2/3 = Always/Once/Update/Never**
Name	*len* (4 bytes), *name* (*len* bytes): absent (because known) if top bit is set in *Create*
Lock	Component Reference number of ACR (4 bytes)
Spec	Bit mask of attributes specified in the source code (1 byte)

Some attributes are always present in the file, whether or not they were specified in the source code. Details are given below.

Bit 0 of *Spec* (here denoted by *Spec*.0) is set if the **Lock** field was specified in the corresponding Component Definition. The other bits of *Spec*, and the remainder of the Component Action data, depend on the Component type. See **5.9 Component Details** for background information:

Subcode 10: File Action

Spec.1	*BlockLen* (2 bytes)
Spec.3	*Alignment* (1 byte)
Spec.2	*FileLength* (4 bytes)
	Contents of file (*FileLength* bytes)

BlockLen is always present. *Alignment* is only present if *Spec*.4 is set; *FileLength* and the file contents are only present if *Spec*.2 is set.

Clarification: Spec.3 says whether *Alignment* was specified in the source file (for checking purposes); *Spec*.4 says whether the *Alignment* field is present in the file (because only **ZC8-** series cards expect an *Alignment* field).

Subcode 20: ACR Action

Spec.1	*ACRType* (1 byte)
	ACRData

These fields are described in **5.9.2 ACRs**. If *Spec*.1 is not set, then neither field is present.

Subcode 30: Privilege Action

A Privilege Action contains no further Component Action data.

Subcode 40: Flag Action

*Spec.*1 *Attributes* (1 byte)

This field is always present.

Subcode 50: Directory Action

A Directory Action contains no further Component Action data.

Subcode 70: Key Action

*Spec.*1 *UsageMask* (2 bytes)
*Spec.*2 *AlgorithmMask* (4 bytes if *Spec.*6 is set, otherwise 2 bytes)
*Spec.*3 *ErrorCounter* (1 byte)
*Spec.*4 *ECResetValue* (1 byte)
*Spec.*5 *Data*

The first four fields are always present. If *Spec.*5 is set, then the *Data* field is also present; it takes the form of a Parameter Field of type **BinaryData**, as follows:

ParamType	*Meaning*	*Format*
80	**BinaryData**	1-byte **BinaryData** sub-type code, followed by parameters:
	Subtype	
	81	**bdString**: *len* (4 bytes), *val* (*len* bytes)
	82	**bdLCIndexedKey**: *KeyIndex* (4 bytes)
	83	**bdLCSerialNumber**: no parameters
	84	**bdLCBuildKey**: *Key* (**ctKey** parameter) *Len* (**Long** parameter) *Seed* (**BinaryData** parameter)
	85	**bdLCKey**: *Key* (**ctKey** parameter)

11.1.11 Code Certification Section

This Section is only required for Enhanced BasicCards **ZC3.1**, **ZC3.2**, and **ZC3.31**.

Length	
4	**'CERT'**
4	Section length *len*
4	Start address of Certified Code
len–**4**	Code Certificate, to be sent in the **SET STATE** command

11.2 ZeitControl Debug File Format

A debug file has the same format as an image file, with additional sections containing debug information. The Signature Section has a different name:

'ZCDF' Signature Section – "ZeitControl Debug File"

The debug information sections occur immediately after the **'CONF'** Configuration File Section if present, otherwise the **'VMTP'** Virtual Machine Type Section:

'OPTS' Compiler Options Section – Options with which the source file was compiled
'FILE' Files Section – Names of all source files
'TYPE' Types Section – Descriptions of all data types used in the program
'SYMB' Symbols Sections – Labels and variables, one Section for each scope
'LINE' Line Numbers Section – Source line number information
'FIXU' Fixups Section – Cross-references

11.2.1 Signature Section

Length	
4	**‘ZCDF’** (“ZeitControl Debug File”)
4	Total length of all remaining sections (= file length – 8)

11.2.2 Compiler Options Section

This section contains the compiler options with which the source file was compiled.

Length	
4	**‘OPTS’**
4	Section length
1	–**S***state* parameter: State of the BasicCard
4	–**S***stack* parameter: Stack size requested
4	–**H***heap* parameter: Heap size requested
4	Length len_I of –**I***include-path* parameter
len_I	–**I***include-path* parameter: search paths for included files
4	Length len_D of –**D***constants* parameter
len_D	–**D***constants* parameter: command-line constant definitions
4	Length len_N of –**N***serial-number* parameter
len_N	–**N***serial-number* parameter: serial number of MultiApplication BasicCard

All these fields are present, even if they are not allowed for the given Machine Type.

11.2.3 Files Section

This section contains the names and timestamps of all the source files in the program:

Length	
4	**‘FILE’**
4	Section length
4	String table length len_{ST}
len_{ST}	String table
4	Number of files n
4	Name of file **0**
4	Number of lines in file **0**
4	Length of longest line in file **0**
4	Timestamp of file **0**
. . .	
4	Name of file n – 1
4	Number of lines in file n – 1
4	Length of longest line in file n – 1
4	Timestamp of file n – 1

11.2.4 Types Section

This section contains definitions of every data type that occurs in the program.

Length	
4	**'TYPE'**
4	Section length
4	String table length len_{ST}
len_{ST}	String table
4	Number of type entries n
	Type **0**
. . .	
	Type n – 1

Type format:

Byte	**00**			
Integer	**01**			
Long	**02**			
Long64	**03**			
Single	**04**			
Double	**05**			
String	**06**			
String*n	**07**	**n** (4 bytes)		
Array	**08**	*ElementType* (4 bytes)	*nDims* (1 byte)	
UserType	**09**	*TypeName* (4 bytes)	*nMembers* (4 bytes)	
Member	**0A**	*MemberName* (4 bytes)	*MemberType* (4 bytes)	*Offset* (4 bytes)

ElementType, *MemberType*	Indices of types in the Types section
TypeName, MemberName	Offsets in the string table
nDims	Number of dimensions of the array
nMembers	Number of members in the user-defined type
Offset	Offset of the member in its user-defined type *UserType*

A *UserType* entry is immediately followed by *nMembers* type entries of type *Member*.

11.2.5 Symbols Sections

The first Symbols Section contains global symbols. Each subsequent Symbols Section contains the local symbols for a single procedure. Symbols are sorted by name (according to the 'C' library function `stricmp`). Symbols beginning with '$' are compiler-generated names.

Length	
4	**'SYMB'**
4	Section length
4	Procedure start address (**00000000** for the global Symbols Section)
4	Procedure end address (**00000000** for the global Symbols Section)
4	String table length len_{ST}
len_{ST}	String table
4	Number of symbols n
	Symbol **0**
. . .	
	Symbol $n - 1$

Symbol format:

<table>
<tr><td>Const Long</td><td>0</td><td>SymbolName</td><td colspan="4">4-byte integer</td><td></td></tr>
<tr><td>Const Long64</td><td>1</td><td>SymbolName</td><td colspan="4">8-byte integer</td><td></td></tr>
<tr><td>Const Single</td><td>2</td><td>SymbolName</td><td colspan="4">4-byte floating-point number</td><td></td></tr>
<tr><td>Const Double</td><td>3</td><td>SymbolName</td><td colspan="4">8-byte floating-point number</td><td></td></tr>
<tr><td>Const String</td><td>4</td><td>SymbolName</td><td colspan="2">String</td><td colspan="2">Len</td><td></td></tr>
<tr><td>Label</td><td>5</td><td>SymbolName</td><td colspan="2">Address</td><td colspan="2">CodeLength</td><td>LabelFlags</td></tr>
<tr><td>Variable</td><td>6</td><td>SymbolName</td><td colspan="2">Address</td><td colspan="2">VarType</td><td>Storage</td></tr>
<tr><td>Library Proc</td><td>7</td><td>SymbolName</td><td>Code</td><td>Subcode</td><td colspan="2"></td><td></td></tr>
<tr><td>Command</td><td>8</td><td>SymbolName</td><td colspan="2">Address</td><td>CLA</td><td>INS</td><td></td></tr>
</table>

SymbolName, String	4-byte offsets in the string table
Len	4-byte string length
Address	4-byte address
LabelFlags (1 byte)	**1** = subroutine **2** = function **4** = frameless procedure
VarType	4-byte index in the Types section
Storage (1 byte)	**0** = absolute **1** = signed, FP-relative (procedure parameters, **Private** data) **2** = indirect signed, FP-relative (**String** and array parameters)
Code, Subcode	**SYSTEM** code and subcode bytes

11.2.6 Line Numbers Section

Line-number entries are sorted in increasing code address order.

Length	
4	**'LINE'**
4	Section length
4	Number of line-number entries *n*
10	Line-number entry **0**
. . .	
10	Line-number entry $n - 1$

Line-number entry format:

Code address (4 bytes)	File number (4 bytes)	Line number (4 bytes)	File position (4 bytes)

11.2.7 Fixups Section

This Section contains two tables: Labels and Operands. Entries in the Labels table give the label(s) at a given address. Entries in the Operands table give the operand of a P-Code instruction as a symbol (*Label* or *Variable*).

Length	
04	**'FIXU'**
04	Section length
04	Number of entries in Labels table *nLabs*
0C	Label entry **0**
. . .	
0C	Label entry *nLabs* – 1
04	Number of entries in Operands table *nOps*
0C	Operand entry **0**
. . .	
0C	Operand entry *nOps* – 1

Label entries and Operand entries have the same format:

Code address (4 bytes)	Symbols Section (4 bytes)	Index of symbol in Symbols Section (4 bytes)

11.3 Application File Format

An Application File in the **ZC6-**series MultiApplication BasicCard has the following format:

Length		
4	**'ZCAF'**	"ZeitControl Application File"
2	**Version**	Major/Minor Version Number (currently 10.1)
2	**CompatibilityMask**	**&H0001** for a **ZC6-**series Application File
2	**RamAddressOffset**	Virtual RAM starts here
2	**AppFileAddressOffset**	Virtual EEPROM starts here
2	**StackSize**	P-Code Stack size required by Application
2	**UserRamSize**	Size of User RAM Data + User RAM Heap
2	**ApplIDPtr**	Application ID string
2	**RamInitCodePtr**	Address of compiler-generated RAM initialisation code
2	**InitCodePtr**	Address of user's Application initialisation code
2	**CommandTablePtr**	Table of user-defined commands
2	**AppFileData**	Start of Application's **Eeprom** Data
2	**AppFileHeap**	Start of Application's **Eeprom** Heap
2	**AppFileHeapEnd**	End of Application's **Eeprom** Heap
2	**ErrorHandler**	User-defined **ErrorHandler** procedure
2	**DefaultHandler**	User-defined **Command Else** command
2	**ClaInsFilter**	User-defined **ClaInsFilter** procedure
1	**OptionMask**	**#Pragma** options

An Application File in the **ZC8-**series MultiApplication BasicCard has the following format:

Length		
4	**'ZCAF'**	"ZeitControl Application File"
2	**Version**	Major/Minor Version Number (currently 20.1)
2	**CompatibilityMask**	**&H0002** for a **ZC8-**series Application File
3	**RamAddressOffset**	Virtual RAM starts here
3	**AppFileAddressOffset**	Virtual EEPROM starts here
3	**AppFileData**	Start of Application's **Eeprom** Data
2	**EndOfRamHeap**	End of Application's startup **Ram** heap
2	**RamData**	Start of Application's user **Ram** data
2	**RamHeap**	Start of Application's **Ram** heap
3	**ApplIDPtr**	Application ID string
3	**RamInitCodePtr**	Address of compiler-generated RAM initialisation code
3	**InitCodePtr**	Address of user's Application initialisation code
3	**CommandTablePtr**	Table of user-defined commands
3	**AppFileHeap**	Start of Application's **Eeprom** Heap
3	**ErrorHandler**	User-defined **ErrorHandler** procedure
3	**DefaultHandler**	User-defined **Command Else** command
3	**ClaInsFilter**	User-defined **ClaInsFilter** procedure
1	**OptionMask**	**#Pragma** options
		Application Code and Data

11.4 List File Format

The format of the list file is illustrated by means of a small example program:

```
Declare ApplicationID = "Small Example Program"
Eeprom MonthLength(1 To 12) = 1,28,31,30,31,30,31,31,30,31,30,31
Const InvalidMonth = &H6F01
Command &H80 &H00 GetMonthLength (N)
    If N < 1 Or N > 12 Then
        SW1SW2 = InvalidMonth
    Else
        N = MonthLength (N)
    End If
End Command
```

This program was compiled for the Enhanced BasicCard version ZC3.2, with list file and map file requested:

```
ZCMBasic MonthLen -C3.2 -OL -OM
```

The list file, **`MonthLen.LST`:**

```
❶ File: MonthLen.BAS
❷      1  Declare ApplicationID = "Small Example Program"
❸ $ApplicationID:
❹         EEPDATA     81A8:  15 53 6D 61 6C 6C 20 45 78 61 6D 70 6C 65 20 50
          EEPDATA     81B8:  72 6F 67 72 61 6D
       2  Eeprom MonthLength(1 To 12) = 1,28,31,30,31,30,31,31,30,31,30,31
❺ MonthLength:
          EEPDATA     81BE:  81 C6 02 01 04 0B 00 18
  MonthLength Data:
          EEPDATA     81C6:  00 01 00 1C 00 1F 00 1E 00 1F 00 1E 00 1F 00 1F
          EEPDATA     81D6:  00 1E 00 1F 00 1E 00 1F
       3  Const InvalidMonth = &H6F01
       4  Command &H80 &H00 GetMonthLength (N)
  GetMonthLength:
❻         PCODE     ❼ 8181:❽ 46 00   ❾ ENTER 00
          CMDTAB      817B:  01 80 00 02 81 81
       5      If N < 1 Or N > 12 Then
          PCODE       8183:  8F        PUFWFC (N)❿
          PCODE       8184:  0C 01     PUCWB 01
          PCODE       8186:  52 05     JLTWB $If001
          PCODE       8188:  8F        PUFWFC (N)
          PCODE       8189:  0C 0C     PUCWB 0C
          PCODE       818B:  4F 06     JLEWB $Else001
       6          SW1SW2 = InvalidMonth
  $If001:
          PCODE       818D:  0E 6F01   PUCWW 6F01
          PCODE       8190:  22 56     PORWB SW1SW2
          PCODE       8192:  54        EXIT
       7      Else
       8          N = MonthLength (N)
  $Else001:
          PCODE       8193:  8F        PUFWFC (N)
          PCODE       8194:  0E 81BE   PUCWW MonthLength
          PCODE       8197:  55        ARRAY
          PCODE       8198:  1F        PUINW
          PCODE       8199:  CF        POFWFC (N)
       9      End If
      10  End Command
          PCODE       819A:  54        EXIT
```

❶ Input filename
❷ Source code, with line number
❸ Compiler-generated label (begins with ‘$’)
❹ Eprom data (**EEPDATA** is the name of the region)
❺ User-generated label (no initial ‘$’)
❻ P-Code (**PCODE** is the name of the region)
❼ Address of P-Code instruction
❽ P-Code instruction and operands, in hexadecimal
❾ P-Code instruction and operands, in text
❿ Implicit operand of abbreviated P-Code instruction, in parentheses

11.5 Map File Format

The map file **MonthLen.MAP** from the example program in the previous section, **11.4 List File Format:**

```
❶ Input file: MonthLen.BAS

❷ ===== RAM regions =====

  Name        Start  End    Length
  ----        -----  ---    ------
  RAMSYS      00     00     01
  STACK       70     9B     2C
  RAMDATA                   00
  RAMHEAP     9C     EC     51
  FILEINFO    ED     FF     13

❸ ===== EEPROM regions =====

  Name        Start  End    Length
  ----        -----  ---    ------
  EEPSYS      8020   817A   015B
  CMDTAB      817B   8180   0006
  PCODE       8181   81A7   0027
  STRCON                    0000
  KEYTAB                    0000
  EEPDATA     81A8   81E0   0039
  EEPHEAP     81E1   8FF4   0E14

❹ ===== Symbols by name =====

  Name                     Scope                Address Type
  ----                     -----                ------- ----
  Algorithm                Global               25      PUBLIC BYTE
  CardMajorVersion         Global                       CONST=0003
  CardMinorVersion         Global                       CONST=0002
  CLA                      Global               6B      PUBLIC BYTE
  EnhancedBasicCard        Global                       CONST=0001
  False                    Global                       CONST=0000
  FileError                Global               FA      PUBLIC BYTE
  GetMonthLength           Global               8181    COMMAND &H80 &H00
  Has24BitAddresses        Global                       CONST=0000
  Has64BitExtensions       Global                       CONST=0000
  INS                      Global               6C      PUBLIC BYTE
  InvalidMonth             Global                       CONST=6F01
  KeyNumber                Global               52      PUBLIC BYTE
  Lc                       Global               6F      PUBLIC BYTE
  Le                       Global               55      PUBLIC BYTE
  MonthLength              Global               81BE    EEPROM INTEGER ARRAY
  MonthLength Data         Global               81C6    ARRAY DATA
  N                        GetMonthLength       FC      PARAM INTEGER
  P1                       Global               6D      PUBLIC BYTE
  P1P2                     Global               6D      PUBLIC INTEGER
  P2                       Global               6E      PUBLIC BYTE
  PCodeError               Global               53      PUBLIC BYTE
  ResponseLength           Global               54      PUBLIC BYTE
  SW1                      Global               56      PUBLIC BYTE
  SW1SW2                   Global               56      PUBLIC INTEGER
  SW2                      Global               57      PUBLIC BYTE
```

```
  True                     Global                       CONST=FFFFFFFF

❺ ===== Symbols by location =====

  RAM system data:

  Name                     Scope               Address Type
  ----                     -----               ------- ----
  Algorithm                Global              25      PUBLIC BYTE
  KeyNumber                Global              52      PUBLIC BYTE
  PCodeError               Global              53      PUBLIC BYTE
  ResponseLength           Global              54      PUBLIC BYTE
  Le                       Global              55      PUBLIC BYTE
  SW1                      Global              56      PUBLIC BYTE
  SW1SW2                   Global              56      PUBLIC INTEGER
  SW2                      Global              57      PUBLIC BYTE
  CLA                      Global              6B      PUBLIC BYTE
  INS                      Global              6C      PUBLIC BYTE
  P1                       Global              6D      PUBLIC BYTE
  P1P2                     Global              6D      PUBLIC INTEGER
  P2                       Global              6E      PUBLIC BYTE
  Lc                       Global              6F      PUBLIC BYTE
  FileError                Global              FA      PUBLIC BYTE

  EEPROM user data:

  Name                     Scope               Address Type
  ----                     -----               ------- ----
  MonthLength              Global              81BE    EEPROM INTEGER ARRAY
  MonthLength Data         Global              81C6    ARRAY DATA

❻ User code:

  Name                     Scope               Address Type
  ----                     -----               ------- ----
  GetMonthLength           Global              8181    COMMAND &H80 &H00

❼ Local variables:

  Name                     Scope               Address Type
  ----                     -----               ------- ----
  N                        GetMonthLength      FC      PARAM INTEGER
```

❶ Input filename.
❷ RAM regions: The addresses and lengths of the regions in RAM.
❸ EEPROM regions: The addresses and lengths of the regions in EEPROM.
❹ Symbols by name: All the symbols in alphabetical order.
❺ Symbols by location: All the symbols, ordered according to location and address.
❻ User code: The addresses of all the procedures and labels in the source program.
❼ Local variables: The signed FP-relative addresses of parameters and **Private** data.

Index