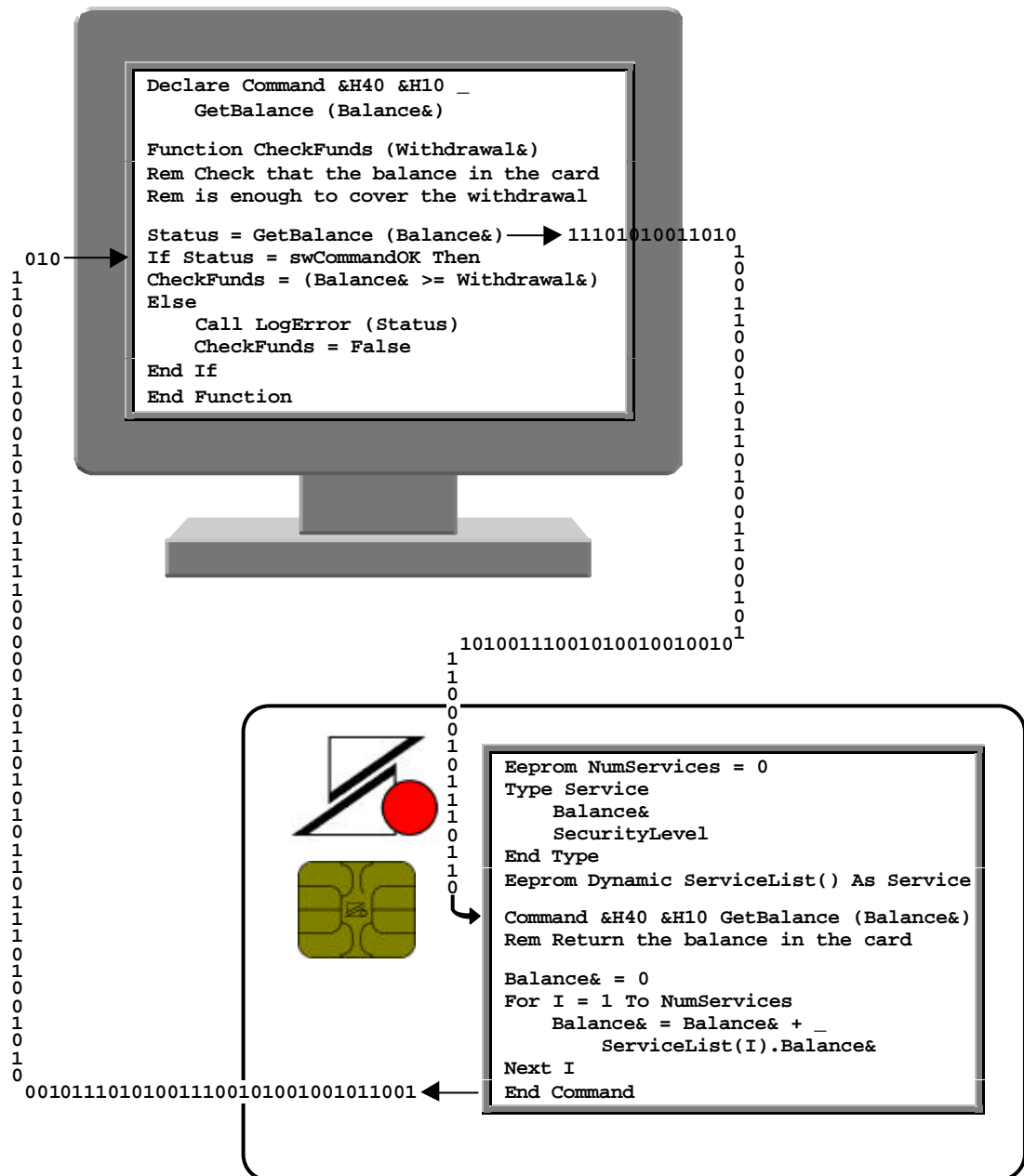


BasicCard



The Compact and Enhanced BasicCards

The ZeitControl Compact and Enhanced BasicCards

Document version 2.72

17th May 1999

Author: Tony Guilfoyle

e-mail: development@ZeitControl.de

Copyright© ZeitControl cardsystems GmbH
Siedlerweg 39
D-32429 Minden
Germany

Tel: +49 (0) 571-50522-0

Fax: +49 (0) 571-50522-99

Web sites:

<http://www.ZeitControl.de>

<http://www.BasicCard.com>

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 just a few Deutschmarks. However, until now the cost of developing software for smart cards has been out of all proportion to the cost of the hardware. A typical development project might take six months and cost half a million marks. This has been the major barrier to the widespread use and acceptance of smart cards.

But now 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. With ZeitControl's BasicCard, the development cycle of writing code, downloading, and testing takes a few minutes instead of weeks.

The products described in this document are the Compact BasicCard and the Enhanced BasicCard. Both BasicCards contain 256 bytes of RAM, and user-programmable EEPROM: 1 kilobyte in the Compact BasicCard, and up to 16 kilobytes in the Enhanced BasicCard (depending on the version). 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 – in the Compact BasicCard, permanent data takes the form of Basic variables, but the Enhanced BasicCard contains a directory-based file system as well. The RAM contains run-time data and the P-Code stack.

How much Basic code can you squeeze into 1 kilobyte of EEPROM? 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 100-200 lines of source code for the Compact BasicCard. And the Enhanced BasicCard can hold up to sixteen times as much.

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 MS-DOS® and Microsoft® Windows® 95. These support packages let you test your software even if you don't have a card reader, by simulating the BasicCard in the PC. (The Windows® 95 software package contains a fully-functional, split-screen symbolic debugger, that can run Terminal and BasicCard programs simultaneously.) So you can try out your idea for a smart card application without it costing you a pfennig.

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, according to the T=1 protocol defined in ISO/IEC standards 7816-3 and 7816-4. But this is completely invisible to the Basic programmer – all you have to do is define a command in the card, and program it as if it was 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 &H01` (&H is the hexadecimal prefix):

```

Eeprom CustomerCredits ' Declare a permanent Integer variable
Command &H20 &H01 GetCustomerCredits (Credits)
    Credits = CustomerCredits
End Command

```

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

```

Const swCommandOK = &H9000
Declare Command &H20 &H01 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. The return value from a command should always be checked, even if the command itself has no error conditions – for instance, the card may have been removed from the reader.

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

The **Compact BasicCard** contains 9K of ROM code, 1K of EEPROM, and 256 bytes of RAM. The ROM code contains:

- 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;
- a command dispatcher built around the structures defined in **ISO/IEC 7816-4: *Interindustry commands for interchange (CLA INS P1 P2 [Lc IDATA] [Le])***;
- built-in commands for loading EEPROM, enabling encryption, etc.;
- a Virtual Machine for the execution of ZeitControl's P-Code;
- code for the automatic encryption and decryption of commands and responses, using the Shrinking Generator algorithm designed by D. Coppersmith, H. Krawczyk, and Y. Mansour.

In addition, the Compact BasicCard operating system requires 71 bytes of RAM and 35 bytes of EEPROM for its own use. The remainder is available for the user's ZC-Basic program.

The **Enhanced BasicCard** contains: 17K of ROM code; up to 16K (**BasicCard ZC2.4**) of EEPROM; and 256 bytes of RAM. Of this, the Enhanced BasicCard operating system requires 107 bytes of RAM and 338 bytes of EEPROM for its own use. As well as the components listed above for the Compact BasicCard, the ROM code in the Enhanced BasicCard contains:

- code for the encryption and decryption of commands and responses using **DES**, the internationally recognised Data Encryption Standard (Single DES and Triple DES are supported);
- a directory-based, DOS-like file system;
- IEEE-compatible floating-point arithmetic.

In addition, the following EEPROM libraries are available for the Enhanced BasicCard:

- **EC-160**: 160-bit Elliptic Curve Cryptography (proposed IEEE standard P1363);
- **SHA-1**: Secure Hash Algorithm, revision 1 (Federal Information Processing Standard FIPS 180-1).

The **software support package** consists of:

- **ZCBASIC**, a compiler for the ZC-Basic programming language;
- **ZCDD**, a split-screen 'Double Debugger' that runs under Windows® 95, for debugging Terminal code and BasicCard code simultaneously;
- **ZCDOS**, a P-Code interpreter that runs compiled ZC-Basic programs under MS-DOS®. **ZCDOS** runs your Terminal program, and can either run your BasicCard program simultaneously in a simulated BasicCard, or communicate over the serial port with a genuine BasicCard;
- **BCLOAD**, for downloading P-Code to the BasicCard;
- **KEYGEN**, a program that generates random keys and primitive polynomials for use in encryption;
- **BCKEYS**, for downloading cryptographic keys to the BasicCard.

Contents

Part I: User's Guide

1. The BasicCard	4
1.1 Processor Cards	4
1.2 Programmable Processor Cards	5
1.3 BasicCard Features	6
1.4 BasicCard Program Layout	6
1.5 BasicCard Versions	9
2. The Terminal	10
2.1 The Terminal Program	10
2.2 Terminal Program Layout	10
3. The ZC-Basic Language	12
3.1 The Source File	12
3.2 Tokens	12
3.3 Pre-Processor Directives	14
3.4 Data Storage	16
3.5 Data Types	17
3.6 Arrays	18
3.7 Data Declaration	18
3.8 User-Defined Types	20
3.9 Expressions	20
3.10 Assignment Statements	23
3.11 Program Control	23
3.12 Procedure Definition	27
3.13 Procedure Calls	28
3.14 Procedure Parameters	30
3.15 Built-in Functions	31
3.16 Encryption	33
3.17 Random Number Generation	36
3.18 Error Handling	37
3.19 BasicCard-Specific Features	37
3.20 Terminal-Specific Features	38
3.21 Miscellaneous Features	42
3.22 Technical Notes	43
4. Files and Directories	45
4.1 Directory-Based File Systems	45
4.2 The Enhanced BasicCard File System	46
4.3 File System Commands	47
4.4 Directory Commands	49
4.5 Creating and Deleting Files	53
4.6 Opening and Closing Files	53
4.7 Writing To Files	55
4.8 Reading From Files	56
4.9 File Locking and Unlocking	57

4.10	Miscellaneous File Operations	59
4.11	File Definition Sections	59
4.12	The Definition File FILEIO.DEF	60
5.	Support Software	62
5.1	Hardware Requirements	62
5.2	Installation	62
5.3	The MS-DOS [®] Support Package	62
5.4	The ZeitControl Double Debugger for Windows [®] 95	70
6.	Plug-In Libraries	76
6.1	EC-160: The Elliptic Curve Library	76
6.2	SHA-1: The Secure Hash Algorithm Library	78
6.3	MATH: Mathematical Functions	82
6.4	MISC: Miscellaneous Procedures	83

Part II: Technical Reference

7.	Communications	86
7.1	The T=1 Protocol	86
7.2	Commands and Responses	87
7.3	Status Bytes SW1 and SW2	89
7.4	Pre-Defined Commands	91
7.5	The Command Definition File COMMANDS.DEF	106
8.	Encryption Algorithms	108
8.1	The DES Algorithm	108
8.2	Implementation of DES in the Enhanced BasicCard	109
8.3	Certificate Generation Using DES	111
8.4	The SG-LFSR Algorithm	111
8.5	Implementation of SG-LFSR in the Compact BasicCard	111
8.6	SG-LFSR with CRC	112
8.7	Encryption – a Worked Example	113
9.	The ZC-Basic Virtual Machine	122
9.1	The BasicCard Virtual Machine	122
9.2	The Terminal Virtual Machine	122
9.3	The P-Code Stack	123
9.4	Run-Time Memory Allocation	123
9.5	Data Types	124
9.6	P-Code Instructions	125
9.7	The SYSTEM Instruction	131
10.	Output File Formats	134
10.1	ZeitControl Image File Format	134
10.2	ZeitControl Debug File Format	137
10.3	List File Format	141
10.4	Map File Format	142
Index		144

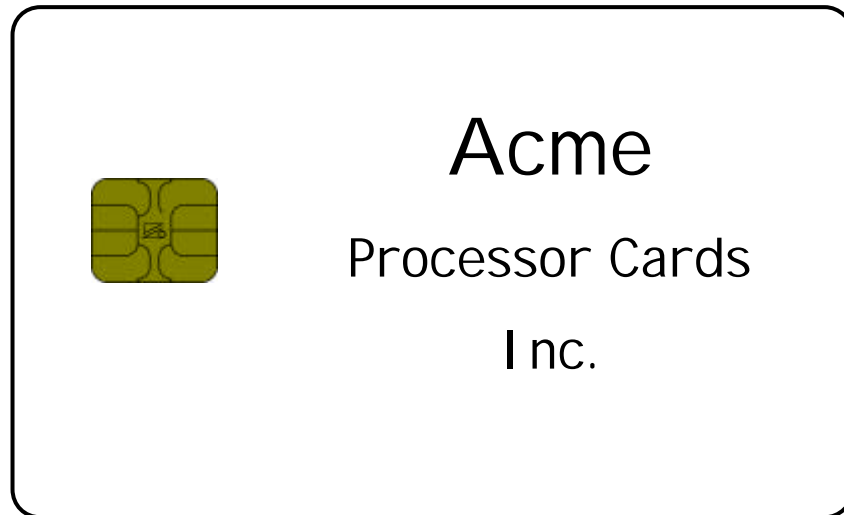
Part I

User's Guide

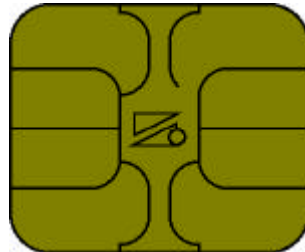
1. The BasicCard

1.1 Processor Cards

A processor card looks like this:



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

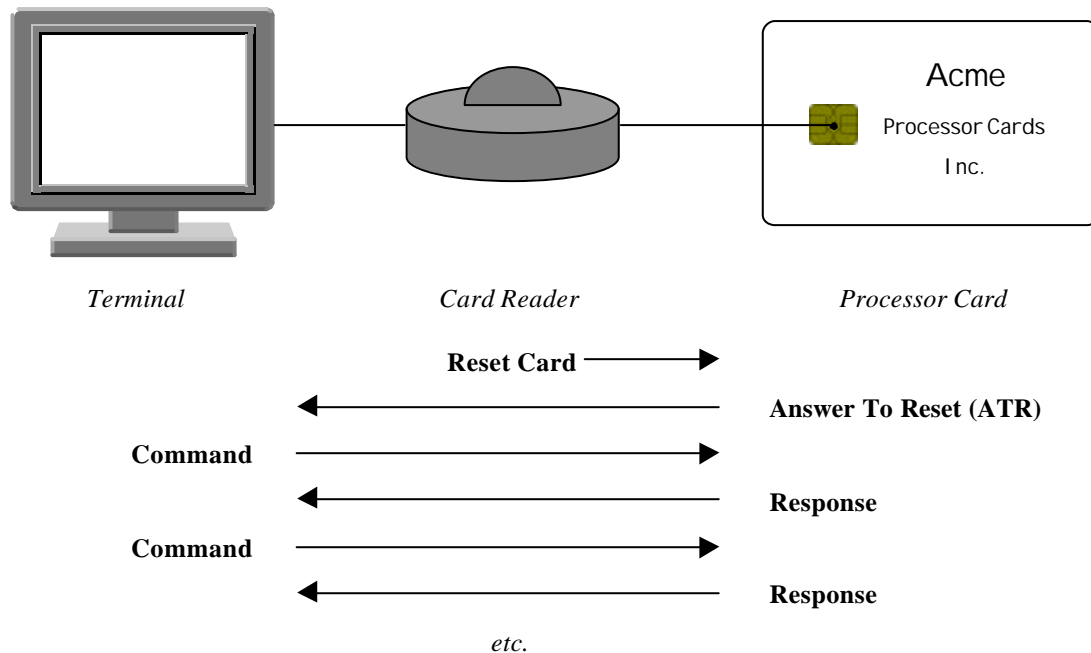


This area has the same layout as a standard telephone card. However, a telephone card contains only memory, while a processor card contains a CPU as well – in effect, a complete miniature computer. A typical processor card today might contain 8-32 kilobytes of ROM (Read-Only Memory) for the operating system machine code, 1-16 kilobytes of EEPROM (Electrically Erasable, Programmable Read-Only Memory) for the data in the card, and 256 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 7: Communications**.

1.2 Programmable Processor Cards

Until recently, 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=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 Java or Basic (the two languages currently available on the market). 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. The BasicCard

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 both versions of the BasicCard cost less than half as much as any other currently-available programmable processor card.
- 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 Enhanced BasicCard looks just like a regular diskette. Encryption has been made as simple as possible – you just turn it on or off. And EEPROM data is read and written just like RAM data.
- Secure* The Enhanced BasicCard uses the well-known **DES** standard, and the **EC-160** Elliptic Curve Cryptography library is provided. The Compact BasicCard uses the Shrinking Generator algorithm designed by D. Coppersmith, H. Krawczyk, and Y. Mansour ("The Shrinking Generator", *Advances in Cryptology – CRYPTO '93 Proceedings*, Springer-Verlag, 1994): a full description is given in **8.4 The SG-LFSR Algorithm**, and C++ source code is provided in the distribution kit. The security of the BasicCard implementation is enhanced by our cryptographic key generation program – see **5.3.4 The Key Generator KEYGEN.EXE** for more information.
- 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. See **7.2 Commands and Responses** for more information.

The operating systems in both 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.
The **T=1** protocol defines the structure and duration of the bits and bytes that constitute the messages in a command-response session. See **7.1 The T=1 Protocol** for more information.
- Pre-defined commands for downloading programs and data to the BasicCard, enabling automatic encryption, etc.
These commands are described in **7.4 Pre-Defined Commands**.
- A Virtual Machine for the execution of ZeitControl's P-Code.
The compiler **ZCBASIC.EXE** 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.EXE** Card Loader program. Then the Virtual Machine in the BasicCard executes the P-Code instructions at run-time.

1.4 BasicCard Program Layout

BasicCard programs are written in the ZC-Basic language, which is a modern procedure-oriented Basic, with special features for the processor card environment. It is described in **Chapter 3: The ZC-Basic Language**.

A program consists of *initialisation code* followed by *procedure definitions*. Programs for the Enhanced BasicCard can also contain optional *file definition sections*.

1.4.1 Initialisation Code

The first block of code that is not contained inside a procedure definition is *initialisation code*: it gets executed when the first user-defined command is called from the Terminal. 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.

1.4.2 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 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 **7.2 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 operating system in the card 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; 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. For convenience of access, 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**.)

1. The BasicCard

- Your own **SW1-SW2** error codes can take any values. However, for **ISO** compliance, the high nibble of **SW1** should be **6**, i.e. **SW1 = &H6X**. You should also avoid assigning new meanings to ZC-Basic's own error codes. ZC-Basic's error codes are listed in **7.3 Status Bytes SW1 and SW2**; you can avoid any clashes if you use **SW1 = &H6B, &H6C, or &H6F**.

1.4.3 File Definition Sections

The Enhanced BasicCard contains a DOS-like file system, with directories organised in a tree structure. There are several ways to access files and directories in the Enhanced BasicCard.

- 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 under DOS or Windows. There are also some special statements for setting access conditions on files and directories, to restrict access from Terminal programs. These access conditions can depend on cryptographic keys, user passwords, etc.
- From a Terminal program, the BasicCard looks just like a diskette, with the special drive name "@:". If the access conditions permit it, you can create, read, and write files and directories in the Enhanced BasicCard as if it was a floppy disk.
- You can initialise directory structures and files in a BasicCard program with File Definition Sections.

1.4.4 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 the Enhanced BasicCard, you can store permanent data in files, but in the Compact BasicCard permanent data must be stored as **Eeprom** data. An example of **Eeprom** data 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 the Enhanced BasicCard as well as the Compact BasicCard.

Writing to EEPROM can take up to 6 milliseconds, so the possibility is always present that the card will lose power in the middle of the write operation. The Enhanced BasicCard automatically logs all EEPROM write operations, to enable it to recover in the event of power loss. The Compact BasicCard has no such recovery mechanism, so EEPROM data may be left in an inconsistent state. In the Compact BasicCard, therefore, important **Eeprom** data should be duplicated to protect against possible corruption if the card is powered down during an EEPROM write operation. For example:

```
Eeprom Balance As Long : Rem A very important piece of data  
Eeprom ShadowBalance As Long  
Eeprom Committed = False  
...  
Command &H80 &H30 ChangeBalance (NewBalance As Long)  
    ShadowBalance = NewBalance  
    Committed = True  
    Balance = ShadowBalance  
    Committed = False  
End Command
```

Then in the initialisation code:

```
If Committed Then  
    Balance = ShadowBalance  
    Committed = False  
End If
```

This technique guarantees that **Balance** will never be left in an inconsistent state.

Note: In the Compact BasicCard, power loss during memory allocation can lead to corruption of the EEPROM heap. For this reason, we recommend that you avoid **ReDim** statements and assignment of

variable-length strings in all Compact BasicCard code that may be executed after the card is issued to the end user. (The Enhanced BasicCard always protects itself against heap corruption, so no such caution is necessary in Enhanced BasicCard code.)

1.5 BasicCard Versions

At the time of writing, the development software recognises the following versions of the BasicCard:

1.5.1 *Compact BasicCard*

- | | |
|------------------------|--|
| BasicCard ZC1.1 | Contains 1K of user-programmable EEPROM. Available since June 1998. |
| BasicCard ZC1.2 | Contains 1K of user-programmable EEPROM. This version was produced as a temporary measure while version ZC1.1 was in the manufacturing phase. It is no longer issued by ZeitControl, but is fully supported by the software. |

1.5.2 *Enhanced BasicCard*

- | | |
|------------------------|---|
| BasicCard ZC2.0 | Contains 610 bytes of user-programmable EEPROM. This version was produced as a temporary measure while version ZC2.3 was in the manufacturing phase. It is no longer issued by ZeitControl, but is fully supported by the software. |
| BasicCard ZC2.1 | Contains 2K of user-programmable EEPROM. Available mid-1999. |
| BasicCard ZC2.2 | Contains 4K of user-programmable EEPROM. Available mid-1999. |
| BasicCard ZC2.3 | Contains 8K of user-programmable EEPROM. This is the first production version of the Enhanced BasicCard. Available since February 1999. |
| BasicCard ZC2.4 | Contains 16K of user-programmable EEPROM. Available mid-1999. |

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 another ZC-Basic program running in a (real or simulated) BasicCard – the **BasicCard** program.

The compiler can create executable files and image files from a Terminal program source file – see **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE** for details.

2.1.1 Executable Files

The compiler can create standard executable files (files with **.EXE** extension), that will run as programs under MS-DOS® or Windows® 95. Such programs can't communicate with a simulated BasicCard – if they call any BasicCard commands, then a real BasicCard must be present. Also, 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 **3.20.10 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 **ZCDOS** P-Code interpreter can then run this Terminal program together with a BasicCard program running in a real or simulated BasicCard – see **5.3.2 The P-Code Interpreter ZCDOS.EXE** for details.

The **ZCDD** Double Debugger for Windows® 95 works with ZeitControl Debug Files (with **.DBG** extension), which are simply ZeitControl Image Files with debugging information included. Image files and debug files are described in **Chapter 10: Output File Formats**.

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 – see **5.3.2 The P-Code Interpreter ZCDOS.EXE**. This program can run the BasicCard program simultaneously in the PC in a simulated BasicCard, or it can communicate with a genuine BasicCard via a card reader – a ZeitControl Chipi® card reader connected to the serial 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 **1.4 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 **7.3 Status Bytes SW1 and SW2** for a list of ZC-Basic status codes. The file **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**.

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 Windows® 95 Double Debugger, checking the **Save Terminal EEPROM** entry in the **Preferences** menu;
 - with the **-W** parameter on the **ZCDOS** command line (see **5.3.2 The P-Code Interpreter ZCDOS.EXE**).
2. When the Terminal program executes a **Write Eeprom** statement (see **3.20.7 Saving Eeprom Data**).

Note: The **Write Eeprom** statement is only valid if the Terminal program is running in the **ZCDOS** P-Code interpreter or the Windows® 95 Double 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, file access control, and EEPROM write logging.

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 program 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 256 bytes of 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 **&**) and **&H** denote octal and hexadecimal constants respectively. Integer constants have the range –2147483648 to +2147483647.

If a constant contains a decimal point or an exponent (E or e), it is a floating-point constant. ZC-Basic supports only single-precision floating-point numbers. Floating-point numbers are stored in IEEE denormalised format, with an 8-bit exponent and a 23-bit mantissa. This gives a precision of 7 decimal places, and a range of 1.401298E–45 to 3.402823E+38.

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
```

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	Single	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 **3.21.2 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	Base
Binary	ByRef	Byte	ByVal	Call
CardInReader	CardReader	Case	Certificate	ChDir
ChDrive	Chr\$	Close	Cls	Command
Const	CurDir	CurDrive	Declare	DefByte
DefInt	DefLng	DefSng	DefString	DES
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
InKey\$	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	Peek
Poke	Polynomials	Print	Private	Public
Put	Random	Randomize	Read	ReDim
Rem	ResetCard	Return	Right\$	RmDir
Rnd	RTrim\$	Seek	Select	SetAttr
Shared	Single	Space\$	Spc	Sqrt
Static	Step	Str\$	String	String\$
Sub	Tab	Then	Time\$	To
Trim\$	Type	UBound	UCase\$	Unlock
Until	Val!	Val&	ValH	WEnd
While	Write	WTX	Xor	

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

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

3. The ZC-Basic Language

3.3 Pre-Processor Directives

Pre-processor directives are instructions to the **ZCBASIC** 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.

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 **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE**);
- finally, the current directory.

3.3.2 Library Inclusion

The directive

```
#Library filename
```

loads a ZeitControl Plug-In Library. See **Chapter 6: Plug-In 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.

Note: 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.

3.3.3 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 **"-Dsymbol"** parameter – see **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE**). *Code block i* is compiled if *condition i* is non-zero.

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**.

3.3.4 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**.

3.3.5 Card State

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

```
#State { LOAD | TEST | RUN }
```

This is equivalent to the command-line parameter **-Sstate** (see **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE**).

3.3.6 Number of Open File Slots

Each open file in a ZC-Basic program is assigned an *open file slot*. The Terminal program has 32 open file slots, so the maximum number of files that can be opened simultaneously is fixed at 32. In the Enhanced BasicCard, the default number of open file slots is 2, but this can be overridden with the **#Files** directive:

```
#Files nFiles
```

with $0 \leq nFiles \leq 16$. This number includes files opened in the BasicCard program *and* BasicCard files opened from a Terminal program. The amount of RAM used by the file system is $(6 * nFiles + 7)$ bytes (unless *nFiles* is zero, in which case no file system is installed, so no RAM is required).

3.3.7 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 **-Sstack-size** (see **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE**). If no stack size is specified, the compiler works out for itself how big the stack should be.

3. The ZC-Basic Language

3.3.8 EEPROM Size

The **#Eeprom** directive specifies the start and end addresses of the EEPROM in the BasicCard:

```
#Eeprom [start] [To end]
```

This is equivalent to the compiler command-line parameters **-ES***start* and **-EE***end* (see **5.3.1 The ZC-Basic Compiler ZCBASIC.EXE**). Normally the compiler knows these addresses; you only need to specify them if you have a non-standard BasicCard configuration.

3.3.9 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.

3.3.10 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.

3.3.11 Block Waiting Time

In a BasicCard program, the **BWT** field in the **ATR** can be specified with

```
#BWT n
```

where *n* is a power of 2 between 1 and 512 inclusive. This Block Waiting Time specifies the time that the card is given to execute a command, before the card reader returns with status **swCardTimedOut**. It is expressed in tenths of a second (giving a maximum of 51.2 seconds). Its default value is 16 (1.6 seconds) in a Compact BasicCard, and 128 (12.8 seconds) in an Enhanced BasicCard.

3.3.12 Pre-Defined Constants

According to the target machine (Terminal, Compact BasicCard, or Enhanced BasicCard), one of the following constants is pre-defined by the compiler (and has the value 1):

TerminalProgram	CompactBasicCard	EnhancedBasicCard
-----------------	------------------	-------------------

For instance:

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

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 pocket. EEPROM contains your ZC-Basic program (compiled into P-Code), directories and files (in the Enhanced BasicCard), 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-Mark 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. **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 every time the card is powered up.

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** – unless you have overridden this behaviour with the **Option Explicit** statement (see 3.21.4 **Explicit Declaration of Variables and Arrays**), or by declaring the procedure itself **Static** (see 3.12 **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.
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. <i>Note:</i> The Single data type is not supported in the Compact BasicCard. You may store Single data in the Compact BasicCard, but you can't perform floating-point arithmetic operations or string conversions. However, all floating-point operations and string conversions are supported in the Enhanced BasicCard.
String	Character string, up to 254 bytes long. Requires $n+3$ bytes of storage, where n is the length of the string – a 2-byte pointer to an $(n+1)$ -byte (length, data) pair.
String*n	Fixed-length string, n bytes long, where n is a constant between 1 and 254. Requires n bytes of storage.

You may also define your own data types – see 3.8 **User-Defined Types**.

3. The ZC-Basic Language

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**, **Single**, **String**, **String*n**, or a user-defined type). It may have up to 32 dimensions. The upper and lower bounds for each dimension are subject to the constraints:

$$-32 \leq \text{lower bound} \leq 31 \quad \text{and} \quad \text{lower bound} \leq \text{upper bound} \leq \text{lower bound} + 1023$$

All arrays are either **Dynamic** or **Fixed**. The upper and lower bounds of a **Fixed** 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 3.21.3 Array Subscript Base).

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

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

If *array* is **Fixed**, 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**] *data-declaration* [=initial-value] [, *data-declaration* [=initial-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.

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 | Single | String |
- otherwise, by the initial character of *name*, as specified in the most recent **DefType** statement (see 3.21.2 **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. This feature lets you write to any address in RAM or EEPROM – use it at your own risk!

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.

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**, or **Long** (depending on the value of the constant) for whole-number expressions, **Single** 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**, **Single**, **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**, **Single**, or *boolean*), the following operations are allowed, grouped in descending order of priority:

Group 1	$- \text{expr1}$	Unary minus
	$+ \text{expr1}$	Unary plus (has no effect)
Group 2	Not <i>expr1</i>	Bitwise complement
Group 3	<i>expr1</i> * <i>expr2</i>	Multiplication
	<i>expr1</i> / <i>expr2</i>	Division
	<i>expr1</i> Mod <i>expr2</i>	Remainder
Group 4	<i>expr1</i> + <i>expr2</i>	Addition
	<i>expr1</i> – <i>expr2</i>	Subtraction
Group 5	<i>expr1</i> < <i>expr2</i>	True if <i>expr1</i> is less than <i>expr2</i>
	<i>expr1</i> <= <i>expr2</i>	True if <i>expr1</i> is less than or equal to <i>expr2</i>
	<i>expr1</i> > <i>expr2</i>	True if <i>expr1</i> is greater than <i>expr2</i>
	<i>expr1</i> >= <i>expr2</i>	True if <i>expr1</i> is greater than or equal to <i>expr2</i>
Group 6	<i>expr1</i> = <i>expr2</i>	True if <i>expr1</i> is equal to <i>expr2</i>
	<i>expr1</i> <> <i>expr2</i>	True if <i>expr1</i> is not equal to <i>expr2</i>
Group 7	<i>expr1</i> And <i>expr2</i>	Bitwise And
Group 8	<i>expr1</i> Xor <i>expr2</i>	Bitwise exclusive-or
Group 9	<i>expr1</i> Or <i>expr2</i>	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)**.

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

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

Groups 5 and 6 are the *comparison operators*. Exactly the same conversions are applied as for the numerical operators, but the type of the resulting expression is boolean.

3. The ZC-Basic Language

Groups 2, 6, 7, and 8 are the *bitwise* operators. Bitwise operations are never performed on **Single** expressions; if *expr1* or *expr2* is **Single**, it is converted to **Long** 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, for instance, have changed **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	<i>expr1</i> + <i>expr2</i>	String concatenation
	<i>expr1</i> < <i>expr2</i>	True if <i>expr1</i> is less than <i>expr2</i>
Group 2	<i>expr1</i> <= <i>expr2</i>	True if <i>expr1</i> is less than or equal to <i>expr2</i>
	<i>expr1</i> > <i>expr2</i>	True if <i>expr1</i> is greater than <i>expr2</i>
	<i>expr1</i> >= <i>expr2</i>	True if <i>expr1</i> is greater than or equal to <i>expr2</i>
Group 3	<i>expr1</i> = <i>expr2</i>	True if <i>expr1</i> is equal to <i>expr2</i>
	<i>expr1</i> <> <i>expr2</i>	True if <i>expr1</i> is not equal to <i>expr2</i>

The resulting expression is of **String** type after string concatenation (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$
```

3.9.3 Expressions of User-Defined Type

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

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

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**, or **Single**), 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 Program Control

3.11.1 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 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. <i>Note:</i> 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. The ZC-Basic Language

3.11.2 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.

3.11.3 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.

3.11.4 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.

3.11.5 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 condition1 Then
    code block 1
[Elseif condition2 Then
    code block 2]
[Elseif condition3 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 non-zero (true). If all the conditions are zero (false), then *code block n* is executed.

If there are any statements on the same line after the **Then** of the initial **If**-statement, then this is a *single-line If*-statement. In this case, the **If-Then-Else** block is terminated not with **End If**, but with the end of the line. (This is the only place in the ZC-Basic language where a colon is not equivalent to an end of line.) For instance:

```

If X = 0 Then GoTo 100
If X < 0 Then X = 0 : ElseIf X > 50 Then X = 50

```

This is equivalent to

```

If X = 0 Then
    GoTo 100
End If

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

```

3.11.6 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]

```

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

```

Note: The **Exit For** statement breaks out of only the innermost **For**-loop, even if the **Next** statement specifies more than one loop variable. So the following example prints the values **11** and **21**:

```

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

```

3.11.7 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

```

3. The ZC-Basic Language

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**).

3.11.8 Select Case

The **Select Case** statement 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:

<i>expression</i>	True if <i>test-expression</i> = <i>expression</i>
<i>expr1 To expr2</i>	True if <i>expr1</i> <= <i>test-expression</i> <= <i>expr2</i>
[Is] <i>op expr</i>	True if <i>test-expression op expr</i> , where <i>op</i> 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**.

3.11.9 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 $1 \leq r \leq n$, then **GoTo** *labelr* or **GoSub** *labelr* is executed. If *expression* < 1 or *expression* > *n*, execution proceeds with the following statement.

3.12 Procedure Definition

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

3.12.1 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]
    [procdedure 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**}] *param-name*[(*i*)] [**As type**], where *param-name* is a variable name by which the parameter is accessed in *procedure-code*. See **3.14 Procedure Parameters** for a full discussion of parameters.

3.12.2 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**}] *param-name*[(*i*)] [**As type**], where *param-name* is a variable name by which the parameter is accessed in *procedure-code*. See **3.14 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 return type **Byte**, **Integer**, **Long**, **Single**, or **String**. Other types (including fixed-length strings) are not allowed.

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.

3.12.3 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 ([param-def, param-def, ...])
    [procedure code]
    [Exit Command]
    [procdedure code]
End Command
```

3. The ZC-Basic Language

<i>CLA</i>	The ‘Class’ byte. All the pre-defined commands in the BasicCard have CLA=&HC0 , so you should avoid this value for your own commands, unless you want to override a pre-defined command.
<i>INS</i>	The ‘Instruction’ byte. To be ISO-compliant, you should make this an even number, although the compiler accepts any value.
Static	If the Static keyword is present in the definition, undeclared variables in the procedure have Static storage class, instead of Private .
<i>param-def</i>	[[ByVal ByRef]] param-name[0] [As type] , where <i>param-name</i> is a variable name by which the parameter is accessed in <i>procedure-code</i> . See 3.14 Procedure Parameters for a full discussion of parameters.

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 *CLA* and *INS* are not recognised.
2. A **Command** parameter may not be an array.
3. A **Command** definition is only valid in a BasicCard program; it is not allowed in a Terminal program.
4. If a **Command** parameter is a variable-length string, it must be the last (or only) parameter in the list. In the Compact BasicCard, the compiler must know how long this string can be, so that it can make sure the P-Code stack is large enough; you can specify a maximum length for the string with the special syntax:

param-name <= *maxlen*

For example:

```
Command &H20 &H00 SetUserName(UserID, Name$<=25)
```

In the absence of this special syntax, *maxlen* defaults to 40. (The Enhanced BasicCard uses a more flexible mechanism, and the length of the string is limited only by the requirement that the total parameter list be no larger than 255 bytes. So this special syntax is not required.)

3.13 Procedure Calls

3.13.1 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 consists of the word **Declare**, followed by a procedure definition as defined in the preceding section:

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

In the case of a **Command** called from a Terminal program, the procedure definition does not appear in the current program at all. In this case, a procedure declaration is obligatory. Otherwise, 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**) 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

```

In **Command** declarations, a special *override syntax* is available, for changing the default values of the ISO-defined command parameters **P1**, **P2**, **Lc**, and **Le**. This is not necessary if you are declaring a command that was written in ZC-Basic, but it is useful for declaring the pre-defined commands in the BasicCard, or commands in a foreign card to be called from a ZC-Basic Terminal program:

Declare Command *CLA INS proc-name* ([**P1**=*expr*,] [**P2**=*expr*,] [**Lc**=*expr*,] *arg-list* [, **Le**=*expr*])

where *arg-list* is the list of parameters that are required for a regular procedure call. In the absence of this override syntax, **P1** and **P2** are initialised to zero, and **Lc** and **Le** are initialised to the length of *arg-list*. Any or all of **P1**, **P2**, **Lc**, and **Le** may be overridden in this way, but they must appear in the correct order, with **P1**, **P2**, and **Lc** before *arg-list*, and **Le** after it. You can also initialise **P1** and **P2** simultaneously with **P1P2**=*expr*. And you can specify that no **Le** byte be sent, with **Disable Le**.

For more information on these command parameters, see **Chapter 7: Communications**.

3.13.2 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 *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*, ...]

3.13.3 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. The ZC-Basic Language

3.13.4 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.

In addition, the same override syntax is available in a command call as in a command declaration, for setting the values of the ISO-defined command parameters **P1**, **P2**, **Lc**, and **Le**:

```
var = command-name ([P1=expr,] [P2=expr,] [Lc=expr,] arg-list [, Le=expr])
```

You can also initialise **P1** and **P2** simultaneously with **P1P2=expr**. And you can specify that no **Le** byte be sent, with **Disable Le**.

For more information on these command parameters, see **Chapter 7: Communications**.

3.14 Procedure Parameters

3.14.1 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.)

In order for the write-back mechanism to be invoked for a given parameter, the parameter-passing method must be **ByRef**, and the expression in the function call must be a variable, an array element, or a member of a variable or array element of user-defined type. In other words, it 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
```

3.14.2 String Parameters

There is an important difference between parameters of type **String** and parameters of type **String*n**. The former take up 3 bytes on the P-Code stack, the latter take up **n** bytes. So you should where possible 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 **3.22.2 String Parameter Format**.

3.14.3 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 **3.22.1 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
```

3.14.4 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. (This is not as economical as passing an address, but it enables **Command** parameters to have user-defined type.) 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.

3.15 Built-in Functions

3.15.1 Numerical Functions

- | | |
|----------------|--|
| Abs(X) | Returns the absolute value of <i>X</i> (that is to say, <i>X</i> or $-X$, whichever is positive). The type of the result is the type of <i>X</i> , unless <i>X</i> is Byte , in which case Abs(X) has type Integer . |
| Rnd | Returns a random number of type Long : $-2147483648 \leq \text{Rnd} \leq 2147483647$. See 3.17 Random Number Generation . |
| Sqrt(X) | Returns the square root of <i>X</i> . The result is of type Single . |

3. The ZC-Basic Language

3.15.2 Array Functions

LBound(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**.
UBound(array [, dim])

3.15.3 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*.

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

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. *Note*: If *val* is of type **Single**, use of this statement in an Enhanced BasicCard program will reduce the amount of user-programmable EEPROM available – see **3.22.4 Single-to-String Conversion** for details.

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. *Note*: Use of this statement in an Enhanced BasicCard program will reduce the amount of user-programmable EEPROM available – see **3.22.4 Single-to-String Conversion** for details.

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.

3.15.4 Encryption Functions

Note: These functions are not available in the Compact BasicCard.

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 Terminal program, **Key** is a pre-defined, **Static** array of strings: **Key(0 To 255) As String**. In the Enhanced BasicCard, only keys declared in **Declare Key** statements can be accessed, and the length of each key is fixed; see **3.16.2 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 **3.16.6 DES Encryption Primitives** for more information.

Certificate(key, data) Returns a 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 **3.16.7 Certificate Generation** for more information.

3.15.5 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).

Peek(address) Returns the contents of memory location *address*, as a **Byte** value.

Poke address, val Sets the contents of *address* to the **Byte** *val*. Use at your own risk!

3.16 Encryption

3.16.1 Implementing Encryption

The Compact and Enhanced BasicCards contain a sophisticated mechanism for the encryption and decryption of commands and responses. For full details of the algorithms, see **Chapter 8: Encryption Algorithms**. To implement this mechanism for your commands:

1. Use the **KEYGEN** program to generate a key file, containing cryptographic keys (and primitive polynomials for the **SG-LFSR** algorithm if you are programming for the Compact BasicCard).
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** and **EndEncryption** commands.
4. In the Terminal program, turn automatic encryption on and off with the statements

Call StartEncryption (P1=algorithm, P2=keynum)

Call EndEncryption()

That's all you have to do. An example program is provided in **8.7 Encryption – a Worked Example**.

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.

3.16.2 Key Declaration

The **Declare Key** statement declares a cryptographic key (the **KEYGEN** program outputs its keys as **Declare Key** statements in the key file):

3. The ZC-Basic Language

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 inclusive.

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.)

Note: In the Compact BasicCard, all keys are 8 bytes long.

counter The error counter for the key ($0 \leq \text{counter} \leq 15$). If *counter* is zero, the key is initially disabled. If *counter* is absent, the error counter for the key is initially inactive. See **3.16.5 Key Error Counter** for details.

Note: the *counter* parameter is allowed in all programs, but it is ignored except in Enhanced BasicCard 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* in a Terminal program or an Enhanced BasicCard program (see **3.15.4 Encryption Functions**);
- with the **Read Key File** statement in a Terminal program (see **3.16.4 Run-Time Key Configuration**);
- with the **BCKEYS** program in a BasicCard (see **5.3.5 The Key Loader BCKEYS.EXE**).

Note: For **Triple DES** encryption (not supported in the Compact BasicCard), 16-byte keys are required.

3.16.3 Polynomial Declaration

The encryption algorithm described in **8.4 The SG-LFSR Algorithm** requires two primitive polynomials, of degree 31 and 32. (This is the encryption algorithm used by the Compact BasicCard – the Enhanced BasicCard uses the **DES** algorithm.) You don't need to know what a primitive polynomial is, because the **KEYGEN** program generates them for you, and outputs them to the key file as a **Declare Polynomials** statement:

Declare Polynomials = *PolyA*&, *PolyS*&

PolyA& A primitive polynomial of degree 31, the generator of the Linear Feedback Shift Register **A**.

PolyS& A primitive polynomial of degree 32, the generator of the Linear Feedback Shift Register **S**.

The polynomials may be initialised at compile time, or later – with the **Read Key File** statement in a Terminal program, or with the **BCKEYS** program in a BasicCard.

3.16.4 Run-Time Key Configuration

The Terminal program can load keys and/or polynomials 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.

In Terminal programs and Enhanced BasicCard programs, keys can also be accessed as strings via the **Key**(*keynum*) function. See **3.15.4 Encryption Functions** for details.

3.16.5 Key Error Counter

In the Enhanced 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 Enhanced 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 an Enhanced BasicCard program, two commands are available 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 for the key 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.

Note: This error counter mechanism only applies to the encryption of commands. Even if a key is disabled, it can always be used from within an Enhanced BasicCard program. ZC-Basic functions that use cryptographic keys are listed in **3.15.4 Encryption Functions**.

3.16.6 DES Encryption Primitives

DES message encryption and decryption is based on the four block encryption primitives E_K , D_K , E_K^3 , and D_K^3 , as defined in **8.1 The DES Algorithm**. In Terminal programs and Enhanced BasicCard programs, these primitives are available to the ZC-Basic programmer via the **DES** function:

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

type The type of primitive: +1, -1, +3, or -3, as follows:

+1:	$E_K(block)$	Single DES encryption
-1:	$D_K(block)$	Single DES decryption
+3:	$E_K^3(block)$	Triple DES encryption
-3:	$D_K^3(block)$	Triple DES decryption

key Either a key number from 0 to 255, or a string containing a cryptographic key. The key must be at least 8 bytes long for types +1 and -1, and at least 16 bytes long for types +3 and -3.

block\$ An 8-byte string containing the block to encrypt or decrypt. If longer than 8 bytes, only the first 8 bytes are used; if shorter than 8 bytes, P-Code error **pcBadStringCall** (&H0D) is generated.

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

3.16.7 Certificate Generation

The Terminal program and the Enhanced BasicCard 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

3. The ZC-Basic Language

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:

$S\$ = \text{Certificate}(key, data)$

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. This generates a **Triple DES** certificate if key number *key* is 16 bytes or longer, otherwise a **Single DES** certificate. The result, *S\$*, is always 8 bytes long. The certificate generation algorithm is described in **8.3 Certificate Generation Using DES**.

3.17 Random Number Generation

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

3.17.1 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 **7.4.11 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 **5.3.4 The Key Generator KEYGEN.EXE**).

3.17.2 The BasicCard

Each 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 BasicCard.

Note: The BasicCard simulators in the **ZCDOS** and **ZCDD** 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.

3.18 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 **7.3.2 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.

In the Enhanced BasicCard, 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)

3.19 BasicCard-Specific Features

3.19.1 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. The historical characters in the BasicCard are of the form '**BasicCard ZC_{vvv}**', where *vvv* is the version number of the card. You can supply your own historical characters with the **Declare ATR** statement:

Declare ATR = data

data Any sequence of **Byte** and **String** constants, with a total length <= 15.

3.19.2 Application ID

The BasicCard has a pre-defined command **GET APPLICATION ID** (see **7.4.10 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.

3.19.3 Enabling and Disabling 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 algorithm IDs are available:

Compact BasicCard:	&H11	SG-LFSR
	&H12	SG-LFSR with CRC
Enhanced BasicCard:	&H21	Single DES
	&H22	Triple DES

For maximum security, you should disable any encryption algorithms that you don't plan to use. (The most secure algorithms are &H12 in the Compact BasicCard, and &H22 in the Enhanced BasicCard.)

Note: This command 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.

3.19.4 Asking the Terminal for More Time

The BasicCard has a **BWT** (Block Waiting Time) of 1.6 seconds (see **7.1 The T=1 Protocol** for more information). If a command is going to take longer than 1.6 seconds to complete, it must request more

3. The ZC-Basic Language

time, otherwise the caller will time out (but see **3.20.9 Giving the Card More Time**). 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.

3.19.5 Pre-Defined Variables

The BasicCard operating system has a number of internal variables that can be accessed from the ZC-Basic language. Most of these have to do with communications – see **Chapter 7: Communications** for details. The following are all **Public** variables (in RAM) of type **Byte**:

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 3.19.3 Enabling and Disabling Encryption Algorithms for a list of algorithm IDs.
KeyNumber	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 BasicCard only).

Two **Integer** variables are defined:

P1P2	Concatenation of P1 and P2 .
SW1SW2	Concatenation of SW1 and SW2 .

3.20 Terminal-Specific Features

3.20.1 Screen Output

Screen output uses the **Cls** and **Print** statements in conjunction with the four pre-defined variables **FgCol**, **BgCol**, **CursorX**, and **CursorY** (see **3.20.10 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

<i>separator</i>	‘;’ (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(<i>n</i>)	Prints <i>n</i> space characters.
	Tab(<i>n</i>)	Advances the output column to position <i>n</i> .

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.)

3.20.2 Keyboard Input

InKey\$	Returns a string containing 0, 1, or 2 bytes: 0 bytes if there is no character waiting in the keyboard buffer; 1 if a regular key was pressed; 2 if an extended-ASCII key was pressed (in which case the first byte is zero).
Line Input X\$	Reads a line from the keyboard into the string variable X\$, until the carriage return key is pressed.
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.

3.20.3 Communications

Three 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. If the BasicCard is being simulated in the PC, the words “Simulated Card Reader” are returned in the *name\$* parameter.

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 **3.19.1 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 7.1 The T=1 Protocol)
swCardError	Invalid response from card

3. The ZC-Basic Language

swCardTimedOut Card failed to send an ATR within the prescribed time

3.20.4 PC/SC Functions

Two functions are provided for obtaining 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.

3.20.5 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.

3.20.6 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 Jun 24 15:50:35 1998").

3.20.7 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 ZCDOS P-Code interpreter or the Windows[®] 95 Double Debugger. Programs containing **Write Eeprom** statements can't be compiled into executable files.

3.20.8 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**.

3.20.9 Giving the Card More Time

Sometimes the BasicCard needs more than the Block Waiting Time to execute a command. In principle, the card is responsible for requesting more time, which it does with a **WTX** statement – see **3.19.4 Asking the Terminal for More Time**. However, you can also override the default Block Waiting Time from the Terminal program with a **WTX** statement:

WTX *seconds*

seconds Any expression of type **Byte**: the number of seconds to give the card before timing out. Unlike **WTX** requests in the BasicCard program, this time-out value remains in effect until explicitly cancelled (by **WTX 0**). If *seconds* is equal to 255, the card is given unlimited time to respond.

The Terminal program waits for a response from the card until *both* time-outs (those set by the BasicCard program and the Terminal program) have expired.

Note: This feature is only available if **ComPort** ≤ 4 – that is, if you are accessing a ZeitControl Chipi[®] card reader via the serial port. The PC/SC standard interface does not support this feature. See **3.3.11 Block Waiting Time** for an alternative method of increasing time-outs.

3.20.10 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 **3.20.4 PC/SC Functions** for details.).

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 **3.19.3 Enabling and Disabling Encryption Algorithms** for a list of algorithm IDs.

KeyNumber 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**.

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.

3. The ZC-Basic Language

nParams Number of command-line parameters (see **5.3.2 The P-Code Interpreter ZCDOS.EXE**).

One **Integer** variable is defined:

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

Two **String** arrays are defined:

Param\$(1 To nParams) Command-line parameters passed to the **ZCDOS** program (see **5.3.2 The P-Code Interpreter ZCDOS.EXE**).

Key(0 To 255) Cryptographic keys.

3.21 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**.

3.21.1 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. (So if you want to disable overflow checking for the whole program, then **Disable OverflowCheck** should appear in your initialisation code.)

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

3.21.2 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** | **DefSng** | **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.

3.21.3 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 with a value from -32 to +31.

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

Option Base Explicit

3.21.4 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
```

You can catch all such errors by using the **Option Explicit** statement:

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.

3.22 Technical Notes

3.22.1 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**). If an array has **n** dimensions, then its descriptor occupies **2*n + 4** bytes:

Address of data area (0 if not allocated) (2 bytes)		
Size of each element (1 byte)	D	n (7 bits)
LO(1) (6 bits)	RANGE(1) (10 bits)	
...	...	
LO(n) (6 bits)	RANGE(n) (10 bits)	

D This bit is **1** for **Dynamic** arrays, **0** for **Fixed** arrays.

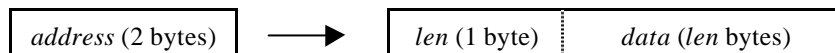
LO(i) Lower bound for subscript(*i*): $-32 \leq \mathbf{LO}(i) \leq 31$.

RANGE(i) Range for subscript(*i*): $0 \leq \mathbf{RANGE}(i) \leq 1023$.

The upper bound of subscript(*i*) is equal to **LO(i)** + **RANGE(i)**.

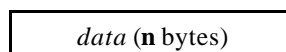
3.22.2 String Parameter Format

A variable of type **String** is a 2-byte pointer to a (*len*, *data*) pair:



This requires *len*+3 bytes of storage (unless *len* is zero, in which case the pointer itself is zero, so only 2 bytes are required).

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



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.

3. The ZC-Basic Language

So a **String** parameter takes up 3 bytes on the P-Code stack. If a fixed-length **String*n** variable was passed, then the first of these bytes contains the length **n** (0-254) and the next two bytes contain the address of the data. Otherwise, the first byte contains 255 (**&HFF**) and the next two bytes contain 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. (By the way, this is the reason for the 254-byte length restriction on all strings.)

3.22.3 Memory Allocation in the BasicCard

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 **9.4 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 (in the Enhanced BasicCard only). 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.

3.22.4 Single-to-String Conversion

The operating system in the Enhanced BasicCard consists of 17.7K of code; the chip, however, contains only 17K of ROM. The last 705 bytes contain the Single-to-String conversion routines. If an Enhanced BasicCard program requires these routines, the ZCBASIC compiler automatically loads them into EEPROM (in the **STRVAL** region – see **9.1.3 Memory Layout in the BasicCard**). This means, of course, that the amount of EEPROM available for your code and data is reduced by 705 bytes.

If any of the following ZC-Basic statements occur in an Enhanced BasicCard program, this **STRVAL** region will be loaded:

- **Str\$(val)** with a *val* parameter of type **Single**;
- **Val!(string)** (**String** to **Single** conversion);
- **Print** to file, with a parameter of type **Single**.

Note: Enhanced BasicCard version **ZC2.0** does not support these Single-to-String conversion statements.

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.

Both MS-DOS[®] and Windows[®] 95 support such directory-based file systems. As far as the ZC-Basic programmer is concerned, the only important difference between the two systems is the format of file and directory names. (Windows[®] 95 calls its directories *folders*, but they will be called directories in this chapter.)

4.1.1 File and Directory Names

MS-DOS[®] file and directory names must be of the form *filename.ext*, where *filename* and *ext* can contain up to 8 and 3 characters respectively. The only characters allowed are the upper-case letters A-Z, the digits 0-9, and the following characters (spaces are not allowed):

_	Underscore	^	Caret	-	Hyphen	@	At-sign
\$	Dollar sign	~	Tilde	{	Left brace	}	Right brace
!	Exclamation mark	#	Hash sign	`	Single quote	'	Apostrophe
%	Percent sign	&	Ampersand	(Left parenthesis)	Right parenthesis

Under Windows[®] 95, 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						

In both MS-DOS[®] and Windows[®] 95, 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. The difference is that MS-DOS[®] stores filenames in upper case, whereas Windows[®] 95 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, MS-DOS[®] lists it as "FILE.NAM", but Windows[®] 95 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:". (As with MS-DOS[®] file names, 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" under Windows[®] 95, or "C:\CLIENTS.97\ACC_DATA.DAT" under MS-DOS[®].

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

4. Files and Directories

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 name 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 Enhanced BasicCard File System

The Enhanced BasicCard contains a directory-based file system, with the same file-naming rules as those described in the previous section for Windows® 95 (except that the maximum length of a full path name is 254 characters). The Enhanced 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 Enhanced BasicCard allows it, files and directories in the card can be accessed from a Terminal program, just as if the card was a diskette. The card has the special drive name “@:”. Suppose the Enhanced 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. 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:

- **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** (see 7.4.1 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. The encryption algorithm must be **Triple DES** for keys at least 16 bytes long, and **Single DES** for shorter keys. So to access a **Keyed** file from a Terminal program, you must first call **StartEncryption** with the appropriate algorithm and key number – see **3.16.1 Implementing Encryption**.

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

4.2.2 Pre-Defined Files and Directories

In an Enhanced 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 Enhanced 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 13 bytes of EEPROM; a data file header requires 19 bytes.
- The name of a file or directory takes up $n+2$ bytes of EEPROM, where n is the number of characters in the name.
- Each data block in a data file uses $n+4$ bytes of EEPROM, 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 4 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 BasicCard uses $(6 * nFiles + 7)$ bytes of RAM, where $nFiles$ is the number of open file slots configured (see **3.3.6 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 only when the Terminal program attempts to access a file or directory in a BasicCard that is in state **TEST** or **RUN**.

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

4. Files and Directories

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, zero is not allowed; the maximum number allowed defaults to 2, but this can be overridden with a #Files directive (see 3.3.6 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. <i>Note:</i> 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 <ul style="list-style-type: none">• it’s a Compact BasicCard; or• no program has yet been downloaded to the card; or• the file system was disabled with a #Files 0 directive (see 3.3.6 Number of Open File Slots).

Definitions of these error codes, as well as all the other constants that appear in this chapter, 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:

Write access is required, both to the directory and to its 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. Files and Directories

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 <i>drive</i> parameter does not exist.
feNameTooLong	The full path name of the current directory is longer than 254 characters (Terminal program only).

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.

Note: Under MS-DOS®, directories can be renamed, but not moved.

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 <i>OldPath</i> does not exist, or the directory specified in <i>NewPath</i> does not exist.
feFileAlreadyExists	The file specified in <i>NewPath</i> 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: <ul style="list-style-type: none">• <i>OldPath</i> is the root directory, which cannot be renamed.• <i>NewPath</i> and <i>OldPath</i> are on different disk drives.• An attempt was made to move a directory under MS-DOS®.
feRecursiveRename	The directory in <i>NewPath</i> is a sub-directory of <i>OldPath</i> , 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, $1 \leq n \leq nFiles$.

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. ZC-Basic uses the host operating system to match wild-card specifications in the PC. MS-DOS® and Windows® 95 handle wild-card characters a little differently, due to the differences in what constitutes a valid filename, but "*" . "*" matches all files and directories in both systems.
3. The Enhanced BasicCard uses a case-insensitive matching algorithm that treats the full stop (period) character '.' no differently from any other character (unlike MS-DOS® and Windows® 95). 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 this command, as 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. Files and Directories

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 attributes that can be returned depend on the host operating system, as follows:

- 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.

- MS-DOS[®] supports these two attributes, plus the following:

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.

- Windows[®] 95 supports all the above attributes, plus the following:

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:

Write access is required, both to the file and to its 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**]

Alternative form: *filenum* = **Open** *filename* [**For** *mode*] [**Access** *access*] [*lock*] [**Len=recordlen**]

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**.

4. Files and Directories

<i>access</i>	Specifies which types of operations will be executed on the file. It takes the value Read , Write , or Read Write . <ul style="list-style-type: none"> • If <i>mode</i> is Input, then <i>access</i>, if present, must be Read. • If <i>mode</i> is Output, then <i>access</i>, if present, must be Write. • If <i>mode</i> is Append, then <i>access</i>, if present, must be Write or Read Write. • If <i>mode</i> is Binary or Random, then <i>access</i> can take any value; it defaults to Read Write.
<i>lock</i>	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 <i>lock</i> parameter can take the following values: <p>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).</p>
<i>filenum</i>	The number of an open file slot, by which read and write operations will be executed. In the Terminal program, <i>filenum</i> must be between 1 and 32 inclusive. In the BasicCard program, <i>filenum</i> must be 1 or 2, unless the number of open file slots has been configured with the #Files directive (see 3.3.6 Number of Open File Slots).
<i>recordlen</i>	Record length or block length. <ul style="list-style-type: none"> • 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 ≤ 8191. • If <i>access</i> is Random, this parameter specifies the record length of the file. This record length must be between 1 and 254 inclusive.

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: <ul style="list-style-type: none"> • the parent directory does not exist; or • <i>mode</i> is Input; or • <i>access</i> is Read.
feNotDataFile	The file is a directory, not a data file.
feFileOpen	(Traditional form only) Open file slot number <i>filenum</i> 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 <i>access</i> is Random , and <i>recordlen</i> is greater than 254; or the file is being created, and <i>recordlen</i> is greater than 8191.
feBadParameter	Either <i>access</i> is Random , and <i>recordlen</i> is less than 1 (or absent); or the file is being created, and <i>recordlen</i> is less than 0.
feSharingViolation	The file is already open, and the required shared access is not available.

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 **3.20.1 Screen Output**), except for the initial *#filenum* parameter:

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

<i>filenum</i>	The <i>filenum</i> parameter to the Open command by which the file was opened.
<i>field</i>	Any Byte , Integer , Long , Single , or String expression
<i>separator</i>	; (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(<i>n</i>) Prints <i>n</i> space characters.
	Tab(<i>n</i>) Advances the output column to position <i>n</i> .

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.)

Note: Use of this statement in an Enhanced BasicCard program with a parameter of type **Single** will reduce the amount of user-programmable EEPROM available – see **3.22.4 Single-to-String Conversion** for details.

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*

<i>filenum</i>	The <i>filenum</i> parameter to the Open command by which the file was opened.
<i>expression-list</i>	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 <i>mode</i> equal to Output or Append .
feInvalidAccess	The file was not opened with <i>access</i> equal to Write or Read Write .

4. Files and Directories

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*

<i>filenum</i>	The <i>filenum</i> parameter to the Open command by which the file was opened.
<i>pos</i>	A record number for Random files, and a character position for Binary files. If <i>pos</i> is not present (“ Put [#] <i>filenum</i> , , <i>data</i> ”), the variable is written to the current file position.
<i>data</i>	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 <i>mode</i> equal to Binary or Random .
feInvalidAccess	The file was not opened with <i>access</i> equal to Write or Read Write .
feSeekError	<i>pos</i> 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 254 characters have been read (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.

<i>filenum</i>	The <i>filenum</i> parameter to the Open command by which the file was opened.
<i>X\$</i>	A variable or array element of type String .
<i>len</i>	The number of characters to read.
<i>variable-list</i>	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 <i>mode</i> equal to Input or Append .
feInvalidAccess	The file was not opened with <i>access</i> 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*]

<i>filenum</i>	The <i>filenum</i> parameter to the Open command by which the file was opened.
<i>pos</i>	A record number for Random files, and a character position for Binary files. If <i>pos</i> is not present (e.g. " Get <i>filenum</i> , , <i>variable</i> "), the read operation takes place at the current file position.
<i>variable</i>	A variable or array element. If this is of type String , it must be followed by the <i>len</i> parameter; otherwise the <i>len</i> parameter must be absent.
<i>len</i>	The number of characters to read, in the case that <i>variable</i> 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 <i>mode</i> equal to Binary or Random .
feInvalidAccess	The file was not opened with <i>access</i> equal to Read or Read Write .
feSeekError	File position <i>pos</i> 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 the Enhanced BasicCard.

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 DES 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	<i>filename</i> is not a BasicCard file or directory.
------------------------	---

4. Files and Directories

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 Enhanced 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

<i>filenum</i> = FreeFile	Returns a free <i>filenum</i> for use in a traditional Open statement. Returns -1 if no more file numbers are available, with error code feTooManyOpenFiles .
Seek [#] <i>filenum</i> , <i>pos</i>	Sets the file pointer to position <i>pos</i> (of type Long) for the next read or write operation on file <i>filenum</i> . <i>pos</i> is a record number for files opened with <i>mode</i> = Random ; otherwise it is a byte count. Records and bytes are numbered from 1. <i>Note:</i> If the file contains less than <i>pos</i> -1 bytes (or records), Seek fails with error code feSeekError , unless the file was opened for output in random access mode (<i>mode</i> = Binary or <i>mode</i> = Random , with Write access specified). In this case, the file is filled with zeroes to the required length.
Seek ([#] <i>filenum</i>)	Returns the read/write position for file <i>filenum</i> , as a Long value.
Len (# <i>filenum</i>)	Returns the length of file <i>filenum</i> in bytes, as a Long value.
EOF ([#] <i>filenum</i>)	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 an Enhanced BasicCard program; it may contain only File Definition statements, not regular ZC-Basic statements. A program may contain any number of File Definition Sections.

4.11.1 Directory Definition

Dir *path*

Lock Definitions

File Definitions

Sub-directory Definitions

End Dir

<i>path</i>	The path name of the directory. It may be a new directory or an existing directory.
<i>Lock Definitions</i>	Lock and Unlock statements for the <i>path</i> directory. These have the same format as the statements described in 4.9 File Locking and Unlocking , but without the <i>filename</i> parameter.
<i>File Definitions</i>	Definitions of files contained in the <i>path</i> directory (see 4.11.2 File Definition).
<i>Sub-directory Definitions</i>	Nested Directory Definitions, defining sub-directories of the <i>path</i> 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. Files and Directories

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

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.

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.

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 supported 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
```


4.12 The Definition File FILEIO.DEF

```
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

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

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 ' FileioDefIncluded
```

5. Support Software

This document describes Version 2.70 of the software support package. All the software described in this chapter is available free of charge from our web site at www.BasicCard.com.

5.1 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 MS-DOS® or Windows® 95 system. Once the software is written and tested, you will need a Chipi® card reader from ZeitControl, and one or more BasicCards. The card reader is available in a stand-alone desktop version, or as an internal device that fits into the diskette drive bay in the PC. A development kit containing Chipi reader, BasicCards, and printed documentation is available from ZeitControl – contact us at Sales@ZeitControl.de.

5.2 Installation

To install the BasicCard software from the CD, just copy the `BasicCrd` directory to your hard disk, together with all its subdirectories. For example, if the CD is in drive E: , the following MS-DOS® command installs the software on drive C:

```
XCOPY E:\BasicCrd C:\BasicCrd\ /S
```

If you have downloaded `BasicCrd.zip` from our web site, you should unzip it from the root directory, enabling the “restore Directory structure” option (in PKUNZIP, this is `-d`; in WinZip, check “Use Folder Names” in the **Extract** dialog box.). This will create the `BasicCrd` directory for you.

You can add the `BasicCrd` directory to your PATH environment variable if you like, but this is not necessary in order to compile and run the example programs.

5.3 The MS-DOS® Support Package

The following programs make up the MS-DOS® support package:

- **ZCBASIC**, a compiler for the ZC-Basic programming language.
- **ZCDOS**, a P-Code interpreter that runs compiled ZC-Basic programs under MS-DOS®. **ZCDOS** runs your Terminal program, and can either run your BasicCard program simultaneously in a simulated BasicCard, or communicate over the serial port with a genuine BasicCard.
- **BCLOAD**, for downloading P-Code to the BasicCard.
- **KEYGEN**, a program that generates random keys and primitive polynomials for use in encryption.
- **BCKEYS**, for downloading keys to the 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.

Notes:

- Three of these programs – **ZCDOS**, **BCLOAD**, and **BCKEYS** – communicate with the Chipi® card reader via the serial port. However, 16-bit DOS programs can experience problems accessing the serial port under Windows® 95. If you are running any of these programs in a DOS box under Windows® 95, you should use the Win32 Console versions **WZCDOS**, **WBCLOAD**, and **WBCKEYS**.

5.3 The MS-DOS® Support Package

- The 16-bit DOS versions of these programs will run in a DOS box under Windows® 95, but they will only accept filenames that conform to the MS-DOS “8.3” convention – eight-character filename plus three-character extension. The Win32 Console version of the compiler, **WZCBASIC**, accepts long filenames and filenames with spaces. (If a filename contains spaces, it must be enclosed in quotation marks on the command line. For example: **WZCBASIC -OI "Hello World"** compiles the file “**Hello World.BAS**” and creates the file “**Hello World.IMG**”.) This version is also needed to create Win32 Console executable files.

5. Support Software

5.3.1 The ZC-Basic Compiler ZCBASIC.EXE

The compiler **ZCBASIC.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:

ZCBASIC [*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:

- Ctype** Compiles code for the given virtual machine type:
 - CT** Terminal (the default).
 - CC1** or **-CC2** Compact BasicCard version **ZC1.1** or **ZC1.2**.
 - CE0** to **-CE4** Enhanced BasicCard version **ZC2.0** to **ZC2.4**.

See **1.5 BasicCard Versions** for more information.

- Dsymbol[=val]** Defines *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.

- ED[exe-file]** Creates an executable file that will run under MS-DOS®.

- EW[exe-file]** Creates a Win32 Console executable file that will run in a DOS box under Windows® 95. (This option is available only in the **WZCBASIC** Win32 Console compiler.)

If no file extension is supplied, *exe-file.exe* is created. If *exe-file* is absent, *input-file.exe* is created.

- ES start** Defines the start address of EEPROM, as a hexadecimal number.

- EEend** Defines the end address of EEPROM, as a hexadecimal number.

These parameters are only required if compiling for a custom BasicCard with a non-standard configuration. (See also **3.3.8 EEPROM Size**.)

- Ipath** Adds *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.

- OI[image-file]** Generates 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 **10.1 ZeitControl Image File Format**.

- OD[debug-file]** Generates 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 **10.2 ZeitControl Debug File Format**.

- 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 **10.3 List File Format**.

- OM[map-file]** Generates 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 **10.4 Map File Format**.

- OE[error-file]** Writes 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.

- Sstack-size** Sets 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 **-Sstack-size** (or with the **#Stack** pre-processor directive – see **3.3.7 Stack Size**).

5.3 The MS-DOS® Support Package

–*Sstate* Switches the card into the specified state after the P-Code is downloaded. See also **3.3.5 Card State**. Only the first letter of *state* is significant:

First letter of <i>state</i> :	‘L’	‘T’	‘R’
New card state:	LOAD	TEST	RUN

Note: The **WZCBASIC** Win32 Console version of the compiler should be used if:

- any of your source files have Windows-style long names; or
- you want to create a Win32 Console executable file; or
- the **ZCBASIC** compiler runs out of memory during compilation.

5. Support Software

5.3.2 The P-Code Interpreter ZCDOS.EXE

The program **ZCDOS.EXE** loads and runs a compiled ZC-Basic Terminal program from a ZeitControl Image File (or Debug File). It can also simultaneously run a BasicCard program in a simulated BasicCard, or it can communicate with a genuine BasicCard through the serial port. To run the **ZCDOS** program:

ZCDOS [*param* [*param* ...]] *image-file* [*P1\$* [*P2\$* ...]]

Parameters *before* the image-file name are processed by the **ZCDOS** 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 **3.20.10 Pre-Defined Variables**.

image-file The image file output by the compiler. If no file extension is supplied, *image-file.img* is assumed.

param One of the following:

- Ccard-file** The image file of the BasicCard program. If this parameter is present, **ZCDOS** simulates a BasicCard in the PC.
- 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.
- Pcom-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.)
- 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*.

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.)

Note: 16-bit DOS programs can experience problems accessing the serial port under Windows® 95. If you are running this program in a DOS box under Windows® 95, you should use the Win32 Console version **WZCDOS.EXE**.

5.3.3 The Card Loader BCLOAD.EXE

The program **BCLOAD.EXE** downloads P-Code and data to the 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.)

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 <i>state</i> :	'L'	'T'	'R'
New card state:	LOAD	TEST	RUN

Notes:

- The ZC-Basic source code for this program is supplied on the distribution disk, in the BasicCrd\Source\BCLoad directory. The two versions **BCLOAD.EXE** and **WBCLOAD.EXE** were compiled with the COMPILE.BAT command file in the same directory.
- 16-bit DOS programs can experience problems accessing the serial port under Windows® 95. If you are running this program in a DOS box under Windows® 95, you should use the Win32 Console version **WBCLOAD.EXE**.

5. Support Software

5.3.4 The Key Generator *KEYGEN.EXE*

The program **KEYGEN.EXE** generates cryptographic keys and primitive polynomials for the encryption and decryption of commands and responses. It creates a ZC-basic source file containing **Declare Key** and/or **Declare Polynomial** statements. This file can be **#Included** in the source code of the Terminal and BasicCard programs, or it can be downloaded separately to the BasicCard using the **BCKEYS** Key Loader program. The program prompts the user to press keys on the keyboard at random; the cryptographic keys and polynomials 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 **3.16.2 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.

-P Generates two random primitive polynomials for use by the **SG-LFSR** encryption algorithms.

-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 and polynomials in *key-file* are preserved, unless overridden by **-K** or **-P**.

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 and polynomials 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 BasicCrd\Source\Keygen.

5.3.5 The Key Loader BCKEYS.EXE

The program **BCKEYS.EXE** downloads cryptographic keys and/or polynomials to a BasicCard. The following conditions apply to the downloading of keys and polynomials:

- 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** and/or **Declare Polynomials** statements. The **KEYGEN** program can generate key files for you – see **5.3.4 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, all the keys in *key-file* are downloaded.
- P** Downloads the polynomials to the BasicCard.
If neither **-K** nor **-P** appears on the command line, then all the keys and polynomials 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 <i>state</i> :	'L'	'T'	'R'
New card state:	LOAD	TEST	RUN

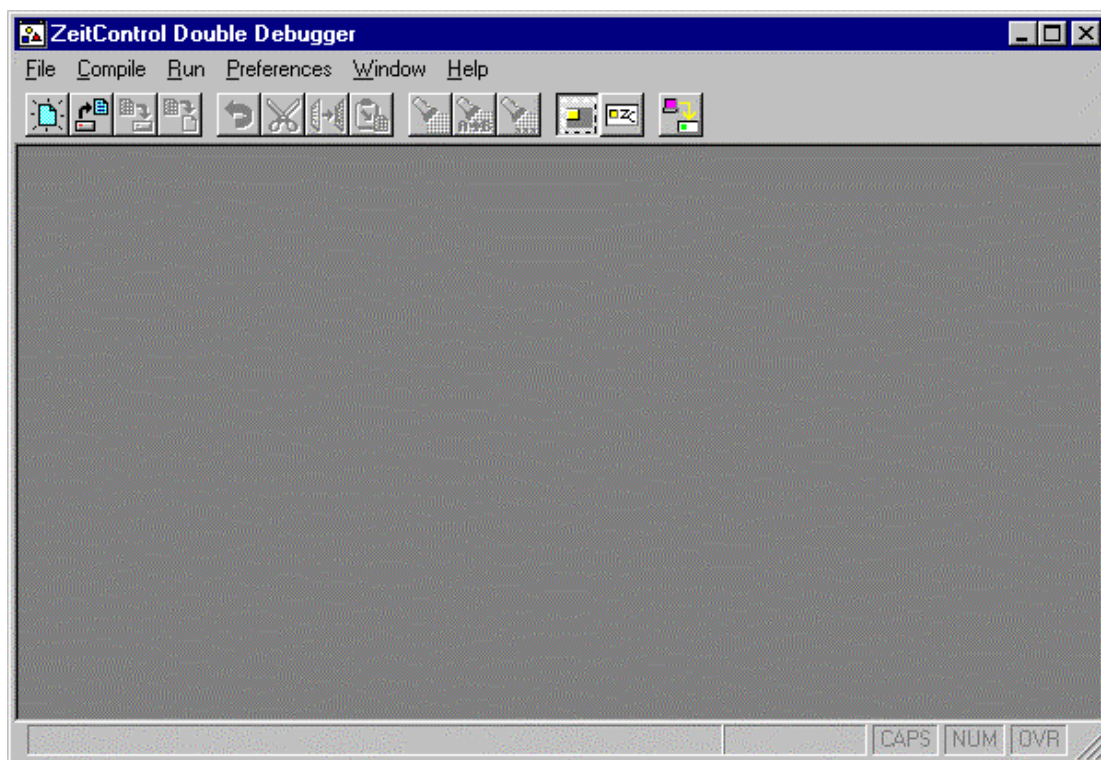
Note: 16-bit DOS programs can experience problems accessing the serial port under Windows® 95. If you are running this program in a DOS box under Windows® 95, you should use the Win32 Console version **WBCKEYS.EXE**.

5.4 The ZeitControl Double Debugger for Windows® 95

The ZeitControl Double Debugger, **ZCDD.EXE**, is a development environment for ZC-Basic applications. It contains the following components:

- a text editor, for creating ZC-Basic source files;
- the ZC-Basic compiler **WZCBASIC.EXE**;
- a BasicCard simulator, for running a BasicCard program in a PC;
- a split-screen debugger, for debugging a Terminal program and a BasicCard program simultaneously;
- the card loader **WBCLOAD.EXE**, for downloading a ZC-Basic program into a BasicCard;
- communication software for running a command-response session with a BasicCard, via a ZeitControl Chipi® card reader in the serial port.

To install the Double Debugger, see **5.2 Installation**. The Double Debugger is the program **ZCDD.EXE**. If you run this program, you should see the following:

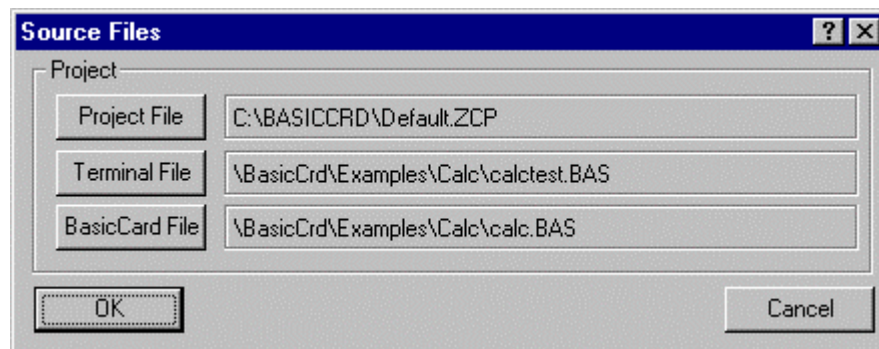


5.4.1 Debugging With Two Source Windows

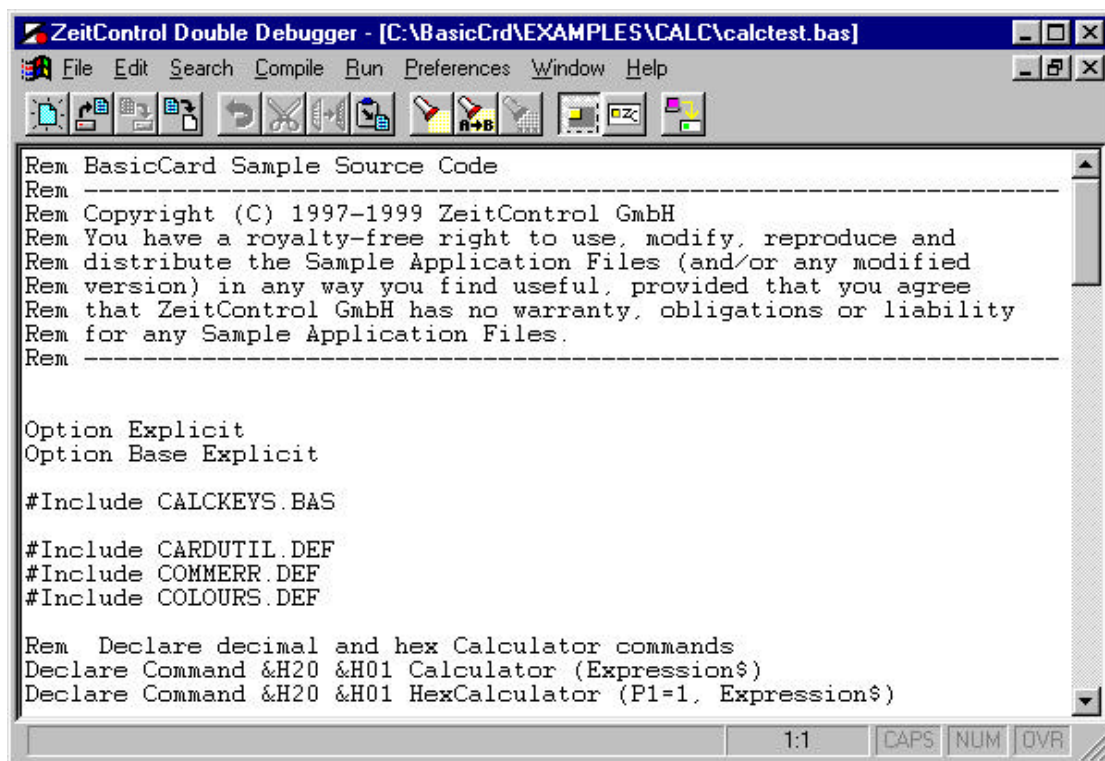
The working of the Double Debugger will be illustrated with the help of an example application. The BasicCard program **Calc.BAS** implements a **CALCULATOR** command, that evaluates ASCII expressions. The Terminal program **CalcTest.BAS** calls this command with various expressions, with and without encryption enabled. These two source files are included in the development package.

5.4 The ZeitControl Double Debugger for Windows® 95

Selecting the **Source Files...** item from the **Preferences** menu brings up the following dialog box:



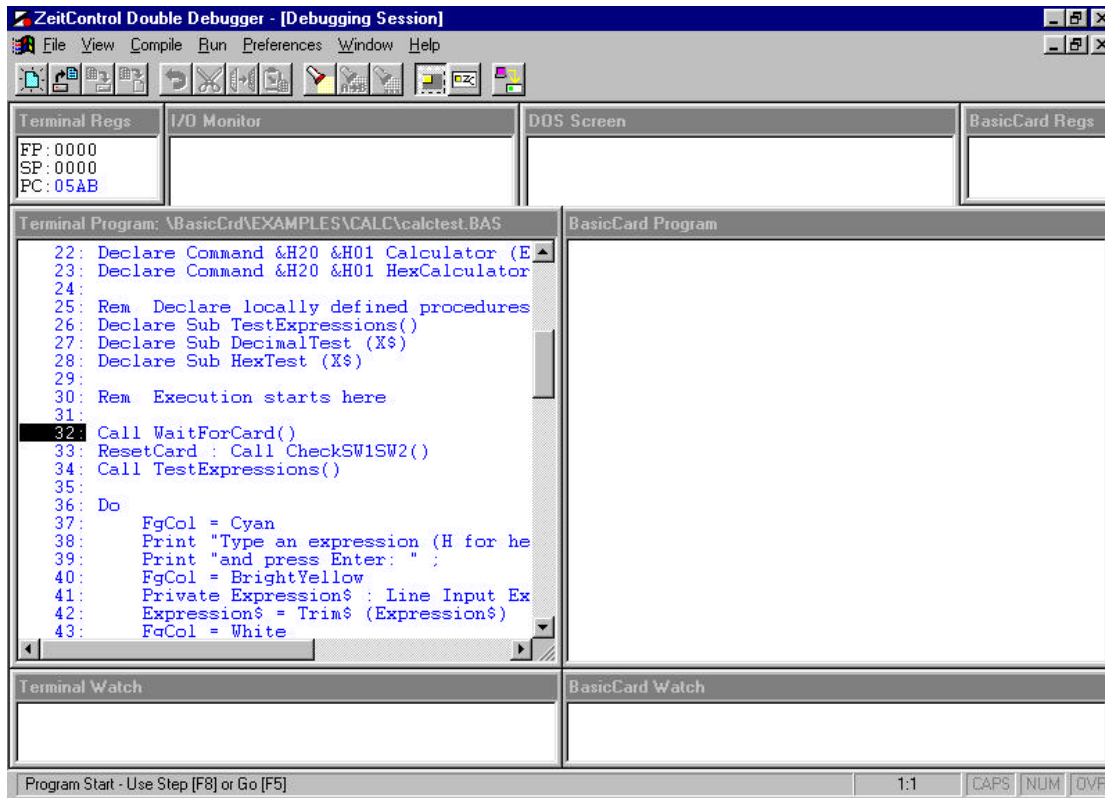
The project file **Default.ZCP** contains the names of the source files for the **CALCULATOR** demo. These source files can be edited via the **Open...** item in the **File** menu:



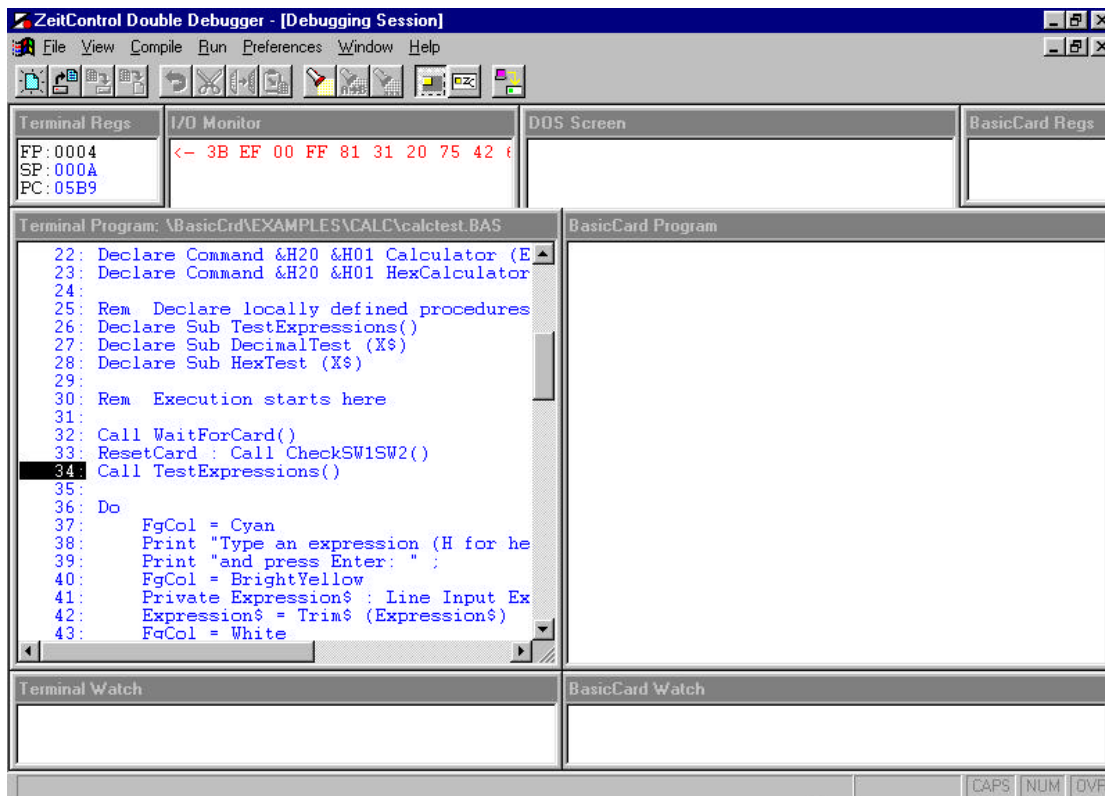
Now select the **Compile All** item in the **Compile** menu. You should see a "Compilation successful" message in the status line (bottom left in the main window).

5. Support Software

To start a debugging session, select **Start** from the **Run** menu (it's a good idea to maximise the window first):

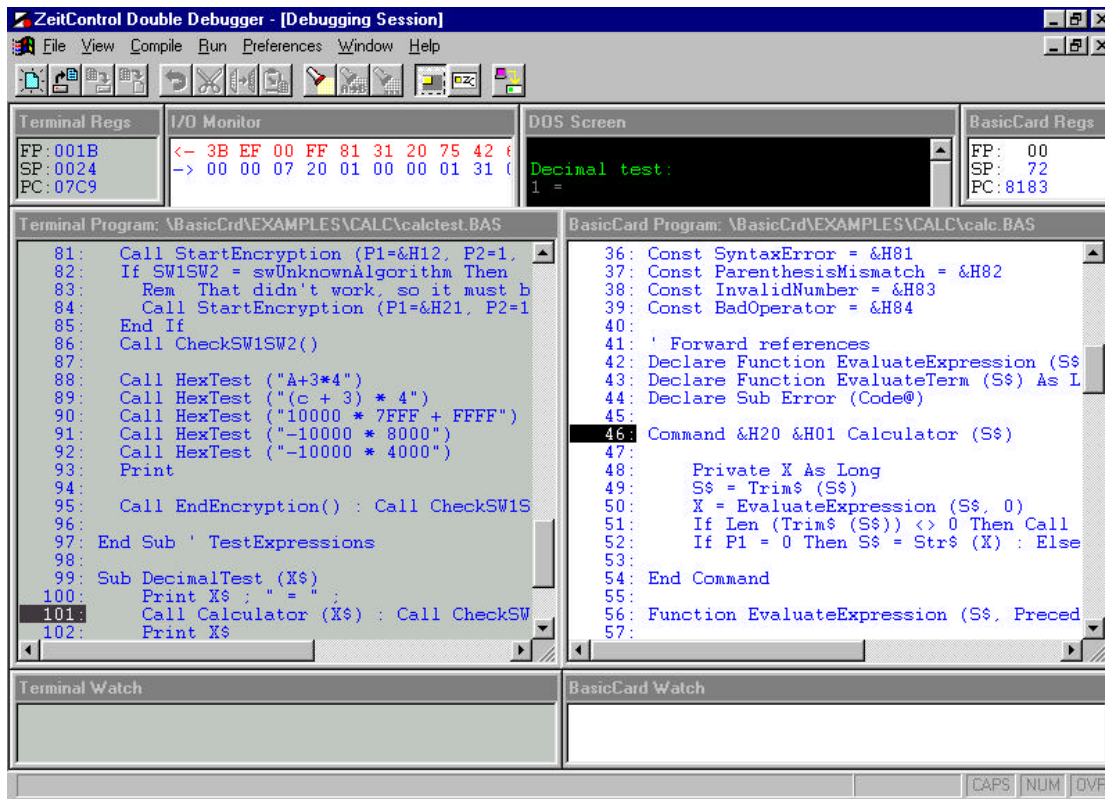


The highlighted line number shows the position of the program counter, or PC, in the source file. Initially, only the Terminal program is displayed. Pressing the **F8** key steps to the next line. Pressing **F8** again executes the **ResetCard** statement there, at which point the ATR (Answer To Reset) is displayed in the I/O Monitor window:



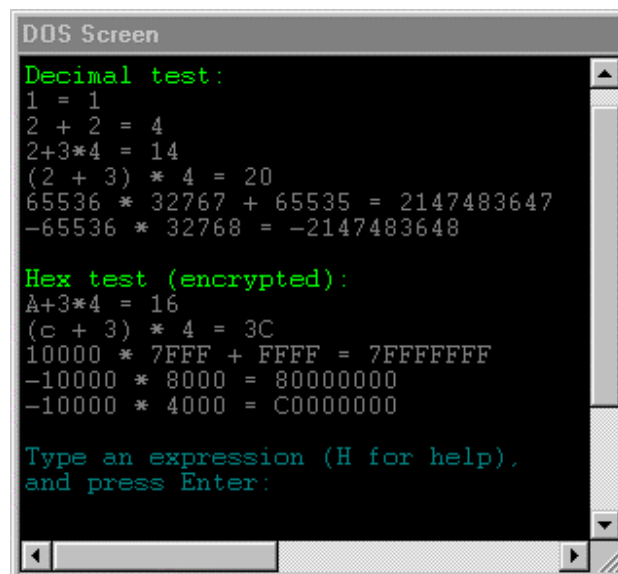
5.4 The ZeitControl Double Debugger for Windows® 95

If you press **F11** (Skip to IO), the program runs until the next I/O event, which in this case is the **Call Calculator** statement on line 101. The BasicCard program becomes the active window:



The command can be seen in the I/O Monitor, and the Terminal program has **Printed** two lines in the DOS Screen window (the third window from the left in the top row). Now you can step through the BasicCard program with **F7** (*Step In*) and **F8** (*Step Over*).

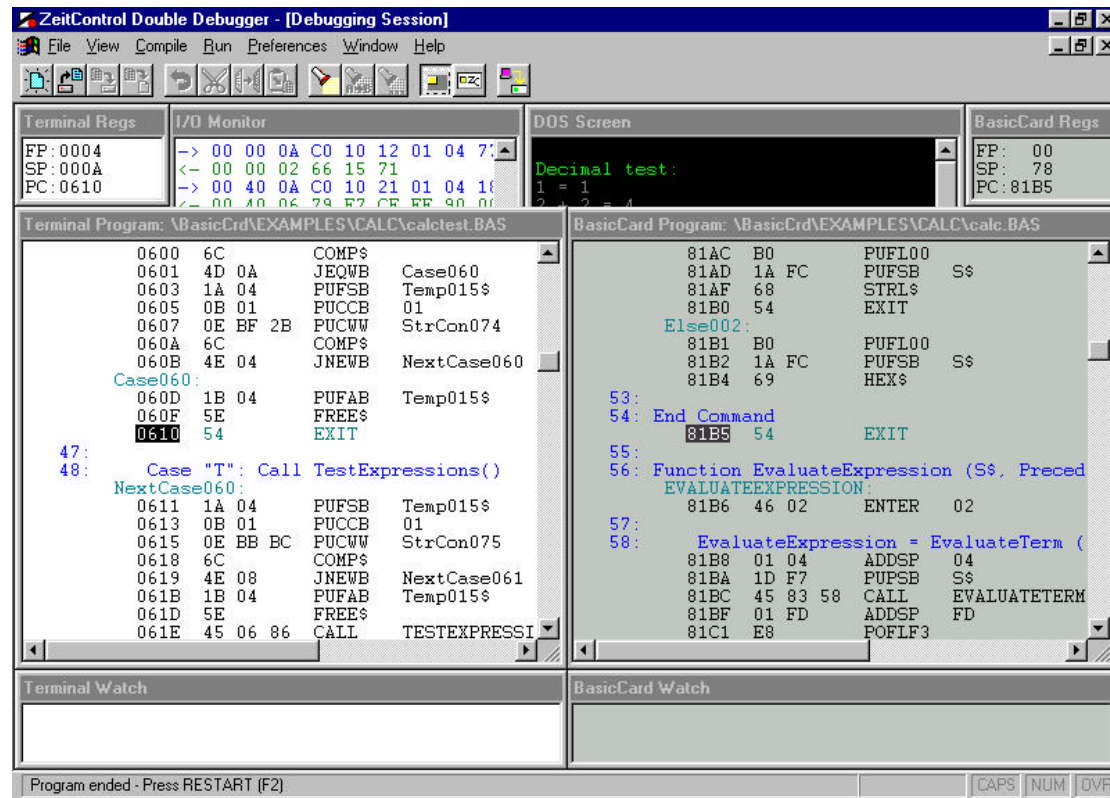
Now press **F5** to run the program to the end. The **Line Input Expression\$** statement on line 41 waits for an expression to be typed in from the keyboard. If you press the **Enter** key to input an empty line, the program will halt. The DOS Screen window should look like this:



5. Support Software

5.4.2 Viewing P-Code

P-Code instructions can be displayed in the Source window, interspersed with the ZC-Basic source code. To continue the example of the previous section, click on each of the two Source windows in turn and select the **Mixed** item in the **View** menu:



Lines numbered in the left-hand column are ZC-Basic statements from the source file. Lines beginning with a 4-digit hexadecimal address are P-Code instructions generated by the compiler.

See **9.6 P-Code Instructions** for a description of ZC-Basic P-Code. When P-Code instructions are displayed in the Source window like this, the Step Into and Step Over instructions execute one P-Code instruction, instead of one Basic statement.

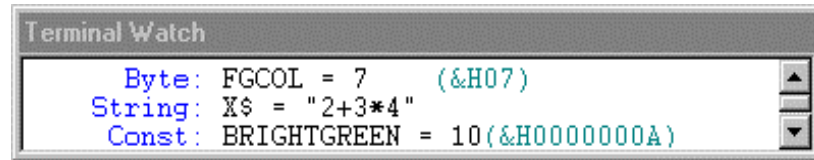
5.4.3 Step Instructions

A Step instruction is a command to the debugger to execute code until a certain condition is met. A variety of Step instructions are provided:

- **Step In (F7 key):** Execute one statement. If this statement contains a procedure call or **GoSub**, step into the called procedure or subroutine.
- **Step Over (F8 key):** Execute one statement. If this statement contains a procedure call, a command call, or a **GoSub** statement, step over the called procedure or subroutine, to the next statement in the current procedure.
- **Step Basic (F9 key):** Execute one Basic statement. This is the same as “Step In” unless the source window is in “Mixed” mode.
- **Skip to Terminal (F10 key):** Run until the Terminal program becomes active.
- **Skip to I/O (F11 key):** Run the program until the next I/O event.

5.4.4 Watch Variables

Double-clicking on a variable name anywhere in the Source window causes the variable to be added to the Watch window (situated directly underneath the Source window). The Watch window displays the values of its variables, automatically updating them if they change. For example:



To cancel a Watch variable, double-click on the variable in the Watch window.

5.4.5 Breakpoints

Double-clicking on a line in the Source window in the line-number column sets a breakpoint. This causes the debugger to halt whenever it reaches the breakpoint line. Breakpoint lines are displayed in red in the Source window. To cancel a breakpoint, double-click on the breakpoint line.

5.4.6 Programming a Real BasicCard

When your code is working correctly in the simulated BasicCard environment, it's time to test it in a real BasicCard. After plugging a ZeitControl Chipi card reader into the serial port, insert a BasicCard into the card reader and select the **Download Code** item in the **Preferences** menu. This invokes the **WBCLOAD.EXE** download program, which displays the download commands on the screen as it executes them. If all goes well, the message "Download successful" appears in the status line.

Now select the **Real BasicCard** item in the **Preferences** menu, followed by the **Restart** and **Run** items in the **Run** menu. This runs the Terminal program as before, but now the commands are sent to the BasicCard in the Chipi card reader.

6. Plug-In Libraries

In Terminal programs and Enhanced BasicCard programs, the functionality of the ZC-Basic language can be extended using ZeitControl Plug-In Libraries. For each Plug-In Library, we provide a ZeitControl Library File *library.lib* and a definition file *library.def*. To use a library:

```
#Include library.def
```

This loads the library, and declares its procedures and data.

The following ZeitControl Plug-In Libraries are currently available:

<i>Name</i>	<i>Description</i>	<i>Terminal</i>	<i>Enhanced BasicCard</i>
EC-160	160-bit Elliptic Curve Cryptography	✓	✓
SHA-1	Secure Hash Algorithm, revision 1	✓	✓
MATH	Mathematical functions	✓	
MISC	Miscellaneous procedures	✓	✓

These libraries are supplied with the distribution kit, in the `BasicCrd\Lib` directory.

In the descriptions of the individual libraries, error codes may be defined. These error codes are signalled via the **LibError** variable. The ZCBASIC 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 always a 2-byte value of the form &H4XXX, with the high nibble equal to 4. Therefore (at the cost of strict ISO compatibility) you can return **LibError** in the **SW1SW2** status word if a library error is signalled in a BasicCard program. For example:

```
Sub CheckLibError()  
  If LibError <> 0 Then  
    SW1SW2 = LibError  
    LibError = 0 ' Reset LibError for the next command  
    Exit  
  End If  
End Sub
```

6.1 EC-160: The Elliptic Curve Library

The **EC-160** library implements 160-bit Elliptic Curve Cryptography, for Terminal programs and Enhanced BasicCard programs. The following operations are supported:

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

This implementation follows the proposed standard **IEEE P1363: Standard Specifications for Public Key Cryptography**. Section **6.1.9 Conformance Specification** specifies the methods used in library **EC-160**, using the terminology of **IEEE P1363**.

6.1.1 Elliptic Curve Cryptography

Elliptic Curve Cryptography is a branch of Public Key Cryptography that is especially suitable for Smart Card implementation, for two reasons:

- the generation of private/public key pairs is simple enough to be implemented in a Smart Card;

- it requires much smaller key sizes than other well-known methods for the same level of security.

The library **EC-160** uses points with 160-bit prime order; this is considered equivalent in security to 1024-bit RSA.

To load the Elliptic Curve library:

#Include EC-160.DEF

The file EC-160.DEF is supplied with the distribution kit, in the `BasicCrd\Lib` directory.

6.1.2 Setting the Elliptic Curve Parameters

An Elliptic Curve is defined by its EC Domain Parameters; Five suitable Elliptic Curves are supplied in the directory `BasicCrd\Lib`. Choose one of these at random for your application. Files CURVE1.DEF through CURVE5.DEF contain curve definitions in ZC-Basic, for inclusion in a source program. File CURVES.BIN contains the binary data for all five curves, for run-time loading in a Terminal program.

To specify an Elliptic Curve in an Enhanced BasicCard program:

#Include CURVEX.DEF

where *X* is a number from 1 to 5. In a BasicCard program, the curve must be chosen at compile time; it can't be re-loaded at run-time.

In the Terminal program, an Elliptic Curve must be explicitly loaded using **EC160SetCurve**. 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 CURVE3.DEF
Call EC160SetCurve (EC160Params)
```

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 EC160DomainParams
Call GetCurve (Curve) : Call CheckSW1SW2()
Call EC160SetCurve (Curve)
```

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

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

```
Private Curve As EC160DomainParams
Open "CURVES.BIN" For Random As #1 Len=64
Get #1, 3, Curve ' Read Elliptic Curve #3
Close #1
Call CheckFileError()
Call EC160SetCurve (Curve)
```

If the EC domain parameters are invalid, procedure **EC160SetCurve** returns error code **EC160BadCurveParams** in variable **LibError**.

In the Terminal program, you must call **EC160SetCurve** before you call any other procedures from the **EC-160** library. If not, error code **EC160CurveNotInitialised** will be returned in variable **LibError**.

6.1.3 Key Generation

To generate a public/private key pair:

Call EC160GenerateKeyPair (Seed\$)

This procedure uses library **SHA-1** to generate a cryptographically strong pseudo-random number from *Seed\$*, for use as a private key. The 20-byte private key and its associated 21-byte public key are stored in **Eeprom** strings **EC160PrivateKey** and **EC160PublicKey**.

See **6.2.2 Pseudo-Random Number Generation** for more about pseudo-random numbers in **SHA-1**.

6. Plug-In Libraries

6.1.4 Setting an Explicit Private Key

To set an explicit value for a private key:

Call EC160SetPrivateKey (Key\$)

This procedure copies *Key\$* (reduced modulo *r*) to the 20-byte **Eeprom** string **EC160PrivateKey**, and computes the associated 21-byte **Eeprom** string **EC160PublicKey**. (*r* is explained in **6.1.8 Binary Representation Formats: EC Domain Parameters**.)

If *Key\$* is zero modulo *r*, error code **EC160BadProcParams** is returned in variable **LibError**.

Note: In the BasicCard, this procedure takes about 6 seconds to execute at a clock speed of 3.57 MHz. However, if you don't need to compute **EC160PublicKey**, you can simply copy *Key\$* to **EC160PrivateKey**, and the Elliptic Curve routines will work correctly.

6.1.5 Generating a Digital Signature

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

Signature\$ = **EC160HashAndSign** (*Message\$*)

This function returns a 40-byte string.

To sign a longer message, first compute the hash function for the message (see **6.2.1 Hashing Functions**), and then call

Signature\$ = **EC160Sign** (*Hash\$*)

If no private key has been set, these functions return error code **EC160KeyNotInitialised** in variable **LibError**.

In the BasicCard, digital signature generation takes about 6 seconds at a clock speed of 3.57 MHz.

6.1.6 Verifying a Digital Signature

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

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

Status = **EC160HashAndVerify** (*Signature\$*, *Message\$*, *PublicKey\$*)

Signature\$ The 40-byte signature to be verified

Message\$ The message that was signed

PublicKey\$ The signer's 21-byte public key

This function returns **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 **6.2.1 Hashing Functions**), and then verify its signature with the function:

Status = **EC160Verify** (*Signature\$*, *Hash\$*, *PublicKey\$*)

If *Signature\$* is not 40 bytes, or *PublicKey\$* is not 21 bytes, error code **EC160BadProcParams** is returned in variable **LibError**.

6.1.7 Session Key Generation

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

SharedSecret\$ = **EC160SharedSecret** (*PublicKey\$*)

PublicKey\$ The other party's 21-byte public key

SharedSecret\$ The 21-byte shared secret

If *PublicKey\$* is not 21 bytes long, or it is not a point on the curve, error **EC160BadProcParams** is returned in variable **LibError**.

Technical Note: If *PublicKey\$* is a point on the elliptic curve, but it does not have order r , then it is not a valid public key, and *SessionKey\$* will therefore be meaningless. This will not, however, compromise the security of your private key; the library **EC-160** protects against invalid public keys by using Secret Value Derivation Primitive **ECSVDP-DHC** with cofactor multiplication, to prevent so-called small subgroup attacks on the private key.

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

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

SessionKey\$ = **EC160SessionKey**(*KDP\$*, *SharedSecret\$*)

KDP\$ Key Derivation Parameter, a string of any length

SharedSecret\$ The shared secret value, returned by **EC160SharedSecret**

SessionKey\$ The 20-byte session key

Note: In the BasicCard, generating a shared secret is very time-consuming – it takes about 25 seconds at a clock speed of 3.57 MHz. But once a shared secret has been generated for a given public key, session key generation takes less than half a second at the same clock speed, provided **Len(KDP\$)** \leq 42. (Typically, a smart card application will only need to generate session keys for a single public key, for which the shared secret is computed just once in the card's lifetime.)

6.1.8 Binary Representation Formats

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

Integers

Integers in this implementation have a length of either 1 byte or 20 bytes. The first (or leftmost) byte is the most significant – in a 20-byte integer, in contains bits 159-152. The last (or rightmost) byte contains bits 7-0.

Field Elements

The library **EC-160** implements operations on Elliptic Curves over the field **GF**(2^m), with $m = 162$. A Polynomial Basis representation of **GF**(2^m) is used; the irreducible field polynomial $p(t)$ is given by

$$p(t) = t^m + t^{m-1} + t^{m-2} + \dots + t^2 + t + 1$$

An element of **GF**(2^m) is represented by a polynomial over **GF**(2) modulo $p(t)$, of degree $\leq m - 1$. Its binary representation occupies 162 bits stored in 21 bytes, with bits 161-160 in the first (leftmost) byte and bits 7-0 in the last (rightmost) byte. Bit i is the coefficient of t^i in the polynomial representation of the element.

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 $x, y \in \mathbf{GF}(2^m)$ 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 $b \in \mathbf{GF}(2^m)$, integers r and k , and point $G \in E$ constitute the *EC domain parameters*. (k is redundant, as it can be calculated from a, b , and r ; it is included for convenience.)

6. Plug-In Libraries

The library **EC-160** accepts any set of EC domain parameters (a, b, r, k, G) that satisfies the following conditions:

- a is a single byte (with bits 161-8 all zero);
- r is exactly 160 bits long, i.e. $2^{159} < r < 2^{160}$;
- r has at least one digit equal to zero in its base-32 expansion.

Mathematically, $\#E$ will always lie between $2^{162} - 2^{82} + 1$ and $2^{162} + 2^{82} + 1$; so if r is 160 bits long, then $k = \#E / r$ must be equal to 4, 6, or 8.

The user-defined type **EC160DomainParams**, defined in file `BasicCrd\Lib\EC-160.DEF`, contains curve parameters a (1 byte), b (21 bytes), r (20 bytes), k (1 byte), and G (21 bytes), for a total of 64 bytes.

Points on the Curve

Points on the curve play two roles in library **EC-160**:

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

If P is on the curve and $x_P \neq 0$, then $y^2 + x_P y = x_P^3 + a x_P^2 + b$ has two solutions, y_0 and y_1 . Moreover, the two expressions y_0 / x_P and y_1 / x_P differ only in bit 0; 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 $y)_P$. A point P on the curve is represented by 163 bits stored in 21 bytes, with x_P in bits 161-0, and the compressed y-coordinate $y)_P$ in bit 162.

Signatures

A signature consists of two 20-byte integers (c, d) . See **IEEE P1363** for the definitions of c and d .

6.1.9 Conformance Specification

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

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

<i>Primitive</i>	<i>Description</i>	<i>Terminal</i>	<i>Enhanced BasicCard</i>
ECSVDP-DHC	Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version with cofactor multiplication. Compatibility with ECSVDP-DH is enabled.	✓	✓
ECSP-NR	Elliptic Curve Signature Primitive, Nyberg-Rueppel version.	✓	✓
ECVP-NR	Elliptic Curve Verification Primitive, Nyberg-Rueppel version.	✓	

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

6.2 SHA-1: The Secure Hash Algorithm Library

This library implements the Secure Hash Algorithm as defined in the Federal Information Processing Standards document FIPS 180-1. The algorithm takes an arbitrary message as input, and outputs a 20-byte hash of that message. It is supposed to be computationally infeasible to invert this algorithm. More specifically:

- given a 20-byte hash, it is computationally infeasible to construct a message with that hash;
- it is computationally infeasible to construct two different messages with identical hashes.

FIPS 180-1 is available on the Internet, at www.itl.nist.gov/div897/pubs/fip180-1.htm.

The **SHA-1** library was implemented as an adjunct to the **EC-160** Elliptic Curve library. In the first place, it is specified in the proposed IEEE standard P1363 as one of the approved hashing algorithms for use in Elliptic Curve digital signature generation; and in the second place, it provides a source of cryptographically strong pseudo-random numbers, for the generation of keys and signatures.

However, it can also be used as a stand-alone library. To load this library:

#Include SHA-1.DEF

The file SHA-1.DEF is supplied with the distribution kit, in the `BasicCrd\Lib` directory.

6.2.1 Hashing Functions

If a message is contained in a **String**, you can compute its hash with a single function call:

Function ShaHash (\$\$) As String

To hash longer messages, you must use the following procedures:

Sub ShaStart()

Sub ShaAppend (\$\$)

Function ShaEnd() As String

Call **ShaStart()** to initialise the hashing process, then **ShaAppend (\$\$)** for successive blocks of data, and finally **ShaEnd()** to get the 20-byte hash value.

6.2.2 Pseudo-Random Number Generation

The Secure Hash Algorithm can be used to generate 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 `BasicCrd\Examples\EC`).

Sub ShaRandomSeed (Seed\$)

This function mixes the given seed into the 'randomness pool'.

Function ShaRandomHash() As String

This function returns a 20-byte random string. 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:

6. Plug-In Libraries

- 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 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-160** Elliptic Curve Cryptography library.

6.3 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 BasicCrD\Lib directory.

6.3.1 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.

6.3.2 Integer Rounding

Function Floor (X!) As Single	The largest integer $\leq X!$, as a Single value
Function Ceil (X!) As Single	The smallest integer $\geq X!$, as a Single value

6.3.3 Exponentiation

Function Pow (X!, Y!) As Single	$X!$ to the power $Y!$
Function (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

6.3.4 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 ($-p/2 \leq \text{ASin} (X!) \leq p/2$)
Function ACos (X!) As Single	Inverse Cosine function ($0 \leq \text{ACos} (X!) \leq p$)
Function ATan (X!) As Single	Inverse Tangent function ($-p/2 < \text{ATan} (X!) < p/2$)
Function ATan2 (Y!, X!) As Single	Inverse Tangent at (X!, Y!) ($-p < \text{ATan2} (Y!, X!) \leq p$)

6.3.5 Hyperbolic Functions

Function SinH (X!) As Single	Hyperbolic Sine: $(\text{Exp}(X!) - \text{Exp}(-X!)) / 2$
Function CosH (X!) As Single	Hyperbolic Cosine: $(\text{Exp}(X!) + \text{Exp}(-X!)) / 2$
Function TanH (X!) As Single	Hyperbolic Tangent: $\text{SinH}(X!) / \text{CosH}(X!)$

6.3.6 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

6.4 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 BasicCrd\Lib directory.

6.4.1 Suspending the Program

In a Terminal program, the following subroutine suspends execution for the specified number of milliseconds:

Sub Sleep (Milliseconds As Long)

Under Windows[®] 95 (and later versions), this frees the CPU for other processes to use. Under DOS, the program simply continues running in a tight loop until the specified time has elapsed.

6.4.2 Fast EEPROM Writes

The EEPROM in the BasicCard has an erase/write cycle time of 6 milliseconds – it takes this long to guarantee that each bit has been completely discharged and/or recharged. The BasicCard has no internal clock, so it must count instruction cycles to estimate the elapsed time. However, it has no way of knowing the clock frequency, so it must assume the worst case – it must assume that the clock is running at its maximum allowed speed. This maximum speed is specified in standard ISO/IEC 7816-3 as 5 MHz.

If in fact the card reader is generating a slower clock frequency, then EEPROM writes will take longer than they need to. For instance, most readers (including ZeitControl's Chipi[®] card reader) generate a clock frequency of 3.57 MHz; so instead of 6 milliseconds, an EEPROM write takes 8.4 milliseconds. If speed is important to you, and if you know that the clock frequency is only 3.57 MHz (or less), you can call the following procedure:

Sub FastEepromWrites()

The BasicCard operating system will then speed up its EEPROM writes, so that they take 6 milliseconds at the assumed slower clock speed. This procedure is available in an Enhanced BasicCard program only.

Warning: If in fact the card reader is running at faster than 3.57 MHz, calling this procedure may result in subsequent loss of EEPROM data through charge leakage.

Part II

Technical Reference

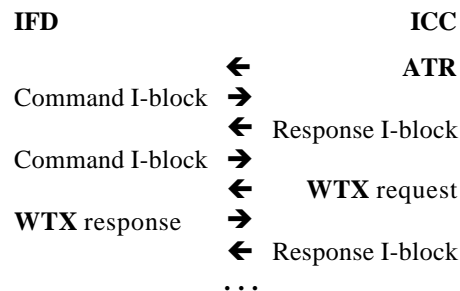
7. Communications

Note: Throughout this chapter, **bold** numbers are hexadecimal.

7.1 The T=1 Protocol

The **T=1** protocol is a 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 **ATR**; (ii) the error-free transmission of I-blocks; (iii) 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**.

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 I-block containing a command, and the **ICC** responds with an I-block 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:



7.1.1 Answer To Reset

The BasicCard sends the following **ATR**:

TS	T0	TB1	TC1	TD1	TD2	TA3	TB3	T1-TK
3B	EF	00	FF	81	31	50	45	‘BasicCard ZC _{vvv} ’

Briefly, what this means is:

TS = 3B	Direct convention (high = 1 , low = 0 ; most 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	IFSC = &H50 (information field size for the ICC)
TB3 = 45	BWT (block waiting time) = (11 + 16 * 960) ETU (= 1.6 seconds between blocks); CWT (character waiting time) = (11 + 32) ETU (= 3.33 ms between characters)
T1-TK	The historical characters (<i>vvv</i> is the BasicCard 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. Historical characters **T1-TK** can be configured in ZC-Basic with the **Declare ATR** statement – see **3.19.1 Customised ATR**.

7.1.2 Structure of an I-block

At the **T=1** level, an I-block contains the following fields. All fields are one byte, except the **INF**:



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, unless otherwise instructed with the pre-defined ASSIGN NAD command. 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. The maximum length of the INF field is equal to the IFSC parameter in the ATR . However, longer messages can be sent by means of the chaining mechanism. (This mechanism is invisible to the BasicCard programmer, so it is not described here.)
INF	Information field – the information content of the I-block. The T=1 protocol says nothing about the internal format of the INF field. See 7.2 Commands and Responses for the format of the INF fields of commands and responses.
LRC	Longitudinal redundancy check. A simple Xor of all the preceding bytes.

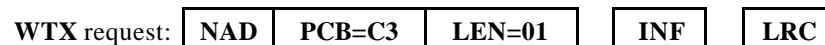
7.1.3 WTX Request

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. 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

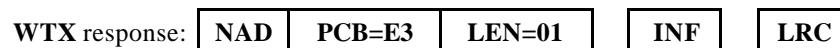
WTX BWT-units

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:



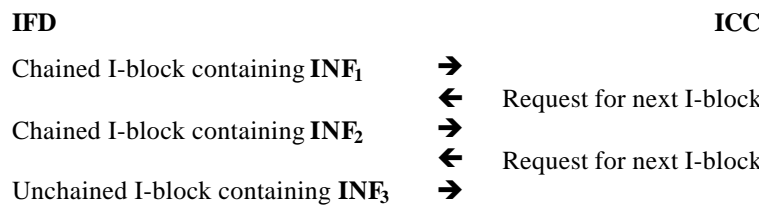
The **INF** field has length 1, and contains the value *BWT-units*. The response to this request contains an identical **INF** field:



7.2 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 previous section described the transmission of I-blocks containing **INF** fields, and mentioned that **INF** fields of successive I-blocks could be chained together to produce longer messages. The result of chaining **INF** fields together is the **APDU** (application protocol data unit). In the following example, **APDU** is the concatenation of **INF₁**, **INF₂**, and **INF₃**:

7. Communications



The **APDU** of a command has the following structure (shaded blocks are optional):



CLA	Class byte – first byte of two-byte CLA INS command identifier.
INS	Instruction byte – second byte of two-byte CLA INS command identifier. For ISO compatibility, this byte should be even.
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 special override syntax described in **3.13.4 Calling a Command**:

Call *command-name* ([**P1=expr**,] [**P2=expr**,] [**Lc=expr**,] *arg-list* [, **Le=expr**])

The **APDU** of a response has the following structure (the shaded block is optional):



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. *Note:* if **SW1-SW2 <> &H9000**, and **SW1 <> &H61**, then **ODATA** is discarded: any return values are lost.

7.3 Status Bytes SW1 and SW2

7.3.1 BasicCard Operating System

The following status codes are returned by the BasicCard operating system:

swCommandOK	9000	Command successfully completed.
sw1LeWarning	61XX	Command successfully completed, but Le was not equal to XX .
swRetriesRemaining	63CX	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	64XX	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.)
swKeyNotFound	6611	The key specified in a START ENCRYPTION command was not configured with a Declare Key statement in the BasicCard program.
swPolyNotFound	6612	The SG-LFSR algorithm was specified in a START ENCRYPTION command, but primitive polynomials were not configured with a Declare Polynomials 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; the Triple DES algorithm requires 16-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.
swBadCommandCRC	66C2	The active encryption algorithm is SG_LFSR with CRC , and the CRC in a command was invalid.
swDesCheckError	66C3	The active encryption algorithm is Single DES or Triple DES , and the authentication bytes in a command were invalid.
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 the Compact BasicCard, this is the same size as the P-Code stack; in the Enhanced BasicCard, it is 256 bytes. (In state LOAD , other limits may apply, but the software support package handles this case.)
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 further commands.

7. Communications

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.
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.

7.3.2 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 configured size.
pcDivideByZero	02	A division by zero (or a Mod with zero divisor) occurred.
pcNotImplemented	03	An unimplemented P-Code instruction was executed (e.g. a floating-point instruction in the Compact BasicCard).
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.

7.3.3 Terminal P-Code Interpreter

The P-Code interpreter in the Terminal program can return the following status codes in **SW1-SW2**:

swNoCardReader	6782	No card reader detected on the given COM port.
swCardReaderError	6783	An invalid reply was received to a card reader command.

swNoCardInReader	6784	No card is inserted in the card reader.
swCardPulled	6785	The card has been removed from the card reader.
swT1Error	6786	An unrecoverable T=1 protocol error occurred while communicating with the card.
swCardError	6787	An invalid response was received to a BasicCard command.
swCardNotReset	6788	The card has not been reset. A BasicCard must be reset before the Terminal program can send it any commands.
swKeyNotLoaded	6789	The key specified in a START ENCRYPTION command is unknown to the Terminal program.
swPolyNotLoaded	678A	The SG-LFSR algorithm was specified in a START ENCRYPTION command, but primitive polynomials have not been configured in the Terminal program.
swBadResponseCRC	678B	The active encryption algorithm is SG_LFSR with CRC , and the CRC in a response was invalid.
swCardTimedOut	678C	The card did not respond within the time allowed.
swTermOutOfMemory	678D	The Terminal program has insufficient free memory to process the response.
swBadDesResponse	678E	The active encryption algorithm is Single DES or Triple DES , and the authentication bytes in a response were invalid.
swInvalidComPort	678F	The COM port is not in the range 1-4.
swComPortNotSupported	6790	The COM port is 3 or 4; these values are allowed only in the Windows [®] 95 support software, not in the MS-DOS [®] software.

7.4 Pre-Defined Commands

7.4.1 States of the BasicCard

The 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 card can be switched from **LOAD** to **TEST** and back again any number of times, but the **RUN** state is permanent. Once the card is switched to state **RUN**, it can't be re-programmed.

7.4.2 Pre-Defined Commands – a Summary

The BasicCard operating system 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 of the card (and the version number of an Enhanced BasicCard)
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

7. Communications

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
ASSIGN NAD	16	Assign a Node Address to the card
FILE IO	18	Execute a file system operation (Enhanced BasicCard only)

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	TEST	RUN
GET STATE	✓	✓	✓	✓
EEPROM SIZE	✓	✓		
CLEAR EEPROM	✓	✓		
WRITE EEPROM	✓	✓		
READ EEPROM	✓	✓	*	*
EEPROM CRC	✓	✓		
SET STATE	✓	✓	✓	
GET APPLICATION ID			✓	✓
START ENCRYPTION			✓	✓
END ENCRYPTION			✓	✓
ECHO	✓	✓	✓	✓
ASSIGN NAD	✓	✓	✓	✓
FILE IO		✓	✓	✓

* The **READ EEPROM** command is allowed in states **TEST** and **RUN** if encryption with key number **0** is enabled (see **7.4.7 The READ EEPROM Command**)

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.

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 (Enhanced BasicCard only)
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 **ZCDD Double Debugger**.) When the application is written and tested, cards can be switched into the **RUN** state for delivery to end users.

7.4.3 The GET STATE Command

GET STATE – Get the state of the card (and the version number of an Enhanced BasicCard)

Command syntax:	CLA	INS	P1	P2	Le
	C0	00	00	00	03

Response from the Compact BasicCard:	ODATA	SW1	SW2
	<i>state</i> (1 byte)	61	01

Response from the Enhanced BasicCard:	ODATA	SW1	SW2
	<i>state</i> (1 byte), <i>version</i> (2 bytes)	90	00

This command returns the state of the BasicCard; an Enhanced BasicCard also returns the version number of the card. If present, the first byte of *version* is equal to 2 (for Enhanced BasicCard), and the second byte is equal to 0, 1, 2, 3, or 4, for card versions **ZC2.0** through **ZC2.4**.

The *state* byte is one of the following:

<i>state</i> :	00	01	02	03
State of card:	NEW	LOAD	TEST	RUN

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%)
```

Note: A Compact BasicCard will leave **Version%** unchanged; an Enhanced BasicCard (and all later versions of the BasicCard) will set the high byte of **Version%** to 2 (or higher). So the following code can be used to determine the type of a BasicCard:

```
#Include COMMANDS.DEF
Version% = &H100
Call GetState (State@, Version%)
Select Case Version% / 256
  Case 1      ' Compact BasicCard
  ...
  Case 2      ' Enhanced BasicCard
  ...
  Case Else   ' A later type that we don't recognise
  ...
End Select
```

7. Communications

7.4.4 The EEPROM SIZE Command

EEPROM SIZE – Get the address and length of EEPROM

Command syntax:

CLA	INS	P1	P2	Le
C0	02	00	00	04

Response:

ODATA	SW1	SW2
<i>start</i> (2 bytes), <i>length</i> (2 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
Call EepromSize (Start%, Length%)
```

7.4.5 The CLEAR EEPROM Command

CLEAR EEPROM – Set specified bytes to **FF**

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	04	<i>hi</i>	<i>lo</i>	02	<i>length</i> (2 bytes)

Response:

SW1	SW2
90	00

Sets *length* bytes of EEPROM to **FF**, starting from address *hi:lo*.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	Lc <> 02 , or length of IDATA <> 02
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
Call ClearEeprom (P1P2=address, Length%)
```

7. Communications

7.4.6 The WRITE EEPROM Command

WRITE EEPROM – Load data into EEPROM

Command syntax:

CLA	INS	P1	P2	Lc	IDATA
C0	06	hi	lo	len	data

Response:

SW1	SW2
90	00

Writes *data* (*len* bytes) to EEPROM starting at address *hi:lo*.

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:

```
#Include COMMANDS.DEF
Call WriteEeprom (P1P2=address, Data$)
```

7.4.7 The READ EEPROM Command

READ EEPROM – Read data from EEPROM

Command syntax:

CLA	INS	P1	P2	Le
C0	08	hi	lo	len

Response:

ODATA	SW1	SW2
len bytes	90	00

Reads *len* bytes from EEPROM starting from address *hi:lo*. 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
Call ReadEeprom (P1P2=address, Data$, Le=len)
```

7. Communications

7.4.8 The EEPROM CRC Command

EEPROM CRC – Calculate a CRC over a specified EEPROM address range

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	0A	hi	lo	02	length (2 bytes)	02

Response:

ODATA	SW1	SW2
CRC (2 bytes)	90	00

Returns the CRC of *length* bytes from address *hi:lo*. 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 the function of enabling 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.

Here is a 'C' function to calculate the CRC:

```
unsigned short CRC (unsigned char *p, unsigned int len)
{
    unsigned short crc = 0 ;
    while (len--)
    {
        unsigned char NextByte = *p++ ;
        int i ;
        for (i = 0 ; i < 8 ; i++, NextByte >>= 1)
        {
            if ((crc ^ NextByte) & 1)
            {
                crc >>= 1 ;
                crc ^= 0xCA00 ;
            }
            else crc >>= 1 ;
        }
    }
    return crc ;
}
```

Command-Specific Error Codes in SW1-SW2:

swLcLeError	Lc <> 02 or length of IDATA <> 02 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
Call EepromCRC (P1P2=address, Length%)
```

The CRC is returned in the **Length%** variable.

7.4.9 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 follows:

state:	01	02	03
New card state:	LOAD	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 > 03 or P2 <> 00

To call **SET STATE** from a Terminal program:

```
#Include COMMANDS.DEF
Call SetState (P1=State@)
```

7. Communications

7.4.10 The GET APPLICATION ID Command

GET APPLICATION ID – Get the Application ID string

Command syntax:

CLA	INS	P1	P2	Le
C0	0E	00	00	00

Response:

ODATA	SW1	SW2
<i>Application-ID</i>	61	len

This command returns the Application ID specified in the ZC-Basic source code statement:

Declare ApplicationID = *Application-ID*

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:

```
#Include COMMANDS.DEF
Call GetApplicationID (Name$)
```


7.4.11 The START ENCRYPTION Command

START ENCRYPTION – Start automatic encryption of command/response data

Command syntax:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	10	<i>algorithm</i>	<i>key</i>	04	Random number RA (4 bytes)	04

Response:	ODATA	SW1	SW2
	Random number RB (4 bytes)	90	00

This command initiates automatic encryption of command and response data fields.

The Compact BasicCard accepts the following two algorithms:

algorithm

11	SG-LFSR (Shrinking Generator – Linear Feedback Shift Register)
12	SG-LFSR with CRC

The Enhanced BasicCard accepts the following two algorithms:

algorithm

21	Single DES (Data Encryption Standard)
22	Triple DES

The Enhanced BasicCard supports automatic algorithm selection: if *algorithm* is zero, then **Single DES** is used if the key is shorter than 16 bytes, otherwise **Triple DES** is used. (The Compact BasicCard returns with **SW1-SW2 = swUnknownAlgorithm** if *algorithm* is zero.)

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, and it must be long enough – at least 16 bytes for **Triple DES**, at least 8 bytes for the other three algorithms.

For descriptions of these algorithms, and the role of **RA** and **RB**, see **Chapter 8: Encryption Algorithms**.

Command-Specific Error Codes in SW1-SW2:

swKeyNotFound	Key number <i>key</i> was not configured
swPolyNotFound	Primitive polynomials were not initialised
swKeyTooShort	Key number <i>key</i> is too short
swKeyDisabled	Key number <i>key</i> is disabled
swUnknownAlgorithm	<i>algorithm</i> is unknown, or is not enabled in the card
swAlreadyEncrypting	Encryption is already enabled
swLcLeError	Lc is present
swInvalidState	Card is not in TEST or RUN state

To call **START ENCRYPTION** from a Terminal program:

```
#Include COMMANDS.DEF
Call StartEncryption ([P1=Algorithm,] P2=KeyNumber, Rnd)
```

7. Communications

7.4.12 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()
```

7.4.13 The ECHO Command

ECHO – Echo the command data

Command syntax:

CLA	INS	P1	P2	Lc	IDATA	Le
C0	14	<i>increment</i>	00	<i>len</i>	<i>data</i>	<i>len</i>

Response:

ODATA	SW1	SW2
<i>data+increment</i>	90	00

This command simply adds *increment* to each byte in *data*. It is intended for testing communication and encryption (see **8.7 Encryption – a Worked Example**).

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$)
```

7. Communications

7.4.14 The ASSIGN NAD Command

ASSIGN NAD – Assign a Node Address to the card

Command syntax:

CLA	INS	P1	P2
C0	16	NAD	00

Response:

SW1	SW2
90	00

If $1 \leq NAD \leq 7$, this command tells the card to respond only to those messages in which the high nibble of the first byte (the **NAD**) is equal to *NAD*. If $NAD = 0$, this command tells the card to respond to all messages. Other values of *NAD* are invalid.

Note: All commands sent by the Terminal program have **NAD=00**. The **ASSIGN NAD** command is intended for use by future versions of ZC-Basic that are capable of communicating with more than one BasicCard at a time.

Command-Specific Error Codes in SW1-SW2:

swLcLeError	Lc or Le present
swP1P2Error	P1 > 07 or P2 <> 00

To call **ASSIGN NAD** from a Terminal program:

```
#Include COMMANDS.DEF
Call AssignNAD (P1=NAD)
```

7.4.15 The FILE IO Command

FILE IO – Execute a file system operation (Enhanced BasicCard only)

Command syntax:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	18	<i>SysCode</i>	<i>filenum</i>	<i>CommandLen</i>	<i>CommandData</i>	<i>ResponseLen</i>

Response:	ODATA	SW1	SW2
	<i>status</i> (1 byte) + <i>ResponseData</i>	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 **9.7.4 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.

7. Communications

7.5 The Command Definition File COMMANDS.DEF

The file COMMANDS.DEF can be found in the directory BasicCrd\Inc. It contains:

- declarations of all the pre-defined commands;
- definitions of the ZC-Basic **SW1-SW2** status codes; and
- definitions of P-Code error codes.

See **7.3 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

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)
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 &H0C SetState()
Declare Command &HC0 &H0E GetApplicationID(Lc=0, Name$)
Declare Command &HC0 &H10 StartEncryption(RA&)
Declare Command &HC0 &H12 EndEncryption()
Declare Command &HC0 &H14 Echo(S$)
Declare Command &HC0 &H16 AssignNAD()

Rem BasicCard operating system errors

Const swCommandOK           = &H9000
Const swRetriesRemaining    = &H63C0
Const swEepromWriteError    = &H6581
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 swLcLeError           = &H6700
Const swCommandTooLong     = &H6781
Const swInvalidState        = &H6985
Const swCardUnconfigured    = &H6986
Const swNewStateError       = &H6987
Const swPlP2Error           = &H6A00
Const swOutsideEeprom       = &H6A02
Const swDataNotFound        = &H6A88
Const swINSNotFound         = &H6D00
Const swCLANotFound         = &H6E00

Rem SW1=&H61 is Le warning:

Const sw1LeWarning          = &H61
```

7.5 The Command Definition File COMMANDS.DEF

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
```

Rem Error codes generated by the Terminal

```
Const swNoCardReader         = &H6782
Const swCardReaderError      = &H6783
Const swNoCardInReader       = &H6784
Const swCardPulled           = &H6785
Const swTlError              = &H6786
Const swCardError            = &H6787
Const swCardNotReset         = &H6788
Const swKeyNotLoaded         = &H6789
Const swPolyNotLoaded        = &H678A
Const swBadResponseCRC       = &H678B
Const swCardTimedOut         = &H678C
Const swTermOutOfMemory      = &H678D
Const swBadDesResponse       = &H678E
Const swInvalidComPort       = &H678F
Const swComPortNotSupported  = &H6790
Const swNoPcscDriver         = &H6791
Const swPcscReaderBusy       = &H6792
Const swPcscError            = &H6793
```

```
#EndIf ' CommandsDefIncluded
```

8. Encryption Algorithms

The Compact BasicCard supports the following two encryption algorithms:

Algorithm ID

11	SG-LFSR (Shrinking Generator – Linear Feedback Shift Register)
12	SG-LFSR with CRC

The Enhanced BasicCard supports the following two encryption algorithms:

Algorithm ID

21	Single DES (Data Encryption Standard)
22	Triple DES

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, skip this chapter and see instead **3.16.1 Implementing Encryption**.

8.1 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, a stronger version, **Triple DES**, has become a *de facto* standard in the banking world. This algorithm is generally believed to be safe against all currently feasible 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 key **K** and an 8-byte data block **P** as input, and returns an 8-byte data block **C** as output:

$$C = E_K(P)$$

The inverse of this is the **DES** decryption function:

$$P = D_K(C)$$

(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 a **Single DES** key contains only 56 bits, although ZC-Basic requires 8-byte keys. This is usual in **DES** implementations; the top bit of each byte can be used as a parity check, or simply thrown away (which is what the BasicCard does).

The **Triple DES** algorithm takes a 16-byte key and splits it into two 8-byte keys **KL** and **KR**. Then the encryption and decryption functions are given by

$$C = E_K^3(P) = E_{KL}(D_{KR}(E_{KL}(P))) \text{ and}$$

$$P = D_K^3(C) = D_{KL}(E_{KR}(D_{KL}(C)))$$

(The four functions E_K , D_K , E_K^3 , and D_K^3 can be called directly from ZC-Basic – see 3.16.6 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.

8.2 Implementation of DES in the Enhanced BasicCard

Apart from their encryption and decryption functions (E and D versus E^3 and D^3), the implementations of **Single DES** and **Triple DES** in the Enhanced 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.)

8.2.1 The Message Encryption Functions ME_K and ME_K^3

The **Single DES** message encryption function $C = ME_K(P)$ 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 E_K and D_K ;
- an 8-byte *initialisation vector* C_0 (more about this in 8.2.3 The Initialisation Vector).

First, split the message P into 8-byte blocks P_1, P_2, \dots, P_{n-1} , plus a final block P_n that may be shorter than 8 bytes. Pad this final block with m zeroes to a length of 8 bytes (so $0 \leq m \leq 7$). Then compute, for $1 \leq i \leq n$:

$$C_i = E_K(C_{i-1} \text{ Xor } P_i)$$

(Note that the initialisation vector C_0 is needed to compute C_1 .) Then throw away the last m bytes of the *penultimate* block C_{n-1} , and concatenate the resulting blocks C_1, \dots, C_n to get the encrypted ciphertext C .

If we threw away the last m bytes of the *last* block C_n , then the message C couldn't be decrypted by its recipient. But the recipient can reconstruct the last m bytes of C_{n-1} , as follows:

The last block is computed from $C_n = E_K(C_{n-1} \text{ Xor } P_n)$

Therefore, $D_K(C_n) = C_{n-1} \text{ Xor } P_n$

which means that $C_{n-1} = D_K(C_n) \text{ Xor } P_n$

But the last m bytes of P_n are all zero, so the last m bytes of C_{n-1} are equal to the last m bytes of $D_K(C_n)$, 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 function $C = ME_K^3(P)$ is defined in exactly the same way, except that the key K is 16 bytes long, and the **Triple DES** encryption function E^3 is substituted for the **Single DES** function E .

8.2.2 The Message Decryption Functions MD_K and MD_K^3

The **Single DES** message decryption function $P = MD_K(C)$ is the inverse of ME_K . First restore the penultimate block C_{n-1} to 8 bytes, as described in the previous section. Then compute, for $1 \leq i \leq n$:

$$P_i = C_{i-1} \text{ Xor } D_K(C_i)$$

Throw away the last m bytes in P_n (which should all be zero), and concatenate all the resulting blocks P_1, \dots, P_n to get the original plaintext message P .

The **Triple DES** message decryption function $C = MD_K^3(P)$ is defined in exactly the same way, except that the **Triple DES** decryption function D^3 is substituted for the **Single DES** function D .

8. Encryption Algorithms

8.2.3 The Initialisation Vector

The initialisation vector C_0 is determined as follows:

For the first command following a **START ENCRYPTION** command, the initialisation vector 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	<i>algorithm</i>	<i>key</i>	04	<i>Random number RA</i> (4 bytes)	04

Response:	ODATA	SW1	SW2
	<i>Random number RB</i> (4 bytes)	90	00

In this case, 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, C_0 is simply the last ciphertext block C_n of the previous message.

8.2.4 Encryption of Commands Using DES

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 $C = ME_K^3(P)$:

C

- Wrap the resulting ciphertext **C** in the original command parameters:

CLA	INS	P1	P2	Lc' + 8	C	Le
------------	------------	-----------	-----------	----------------	----------	-----------

The resulting command is always exactly 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**.

8.2.5 Encryption of Responses Using DES

A response has the following structure (the shaded block is optional):

ODATA	SW1	SW2
--------------	------------	------------

Encryption consists of the following steps:

- Append six zeroes:

$$P = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \text{ODATA} & \text{SW1} & \text{SW2} & 00 & 00 & 00 & 00 & 00 & 00 \\ \hline \end{array}$$

- Encrypt the resulting response **P**, with $C = ME_K(P)$ or $C = ME_K^3(P)$:

$$C$$

- Append the original **SW1-SW2**:

$$\begin{array}{|c|c|c|} \hline C & SW1 & SW2 \\ \hline \end{array}$$

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** \neq **swCommandOK** and **SW1** \neq **sw1LeWarning**), then the response is not encrypted.

8.3 Certificate Generation Using DES

The ZC-Basic **Certificate** command is described in **3.16.7 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 **P₁ ..., P_n**, padding the last block **P_n** with zeroes if necessary. Fill the initialisation vector **C₀** with zeroes, and then compute, for $1 \leq i \leq n$:

$$\begin{aligned} C_i &= E_K(C_{i-1} \text{ Xor } P_i) && \text{(for keys } K \text{ shorter than 16 bytes)} \\ C_i &= E_K^3(C_{i-1} \text{ Xor } P_i) && \text{(for keys } K \text{ 16 bytes or longer)} \end{aligned}$$

The certificate is the final ciphertext block **C_n**.

8.4 The SG-LFSR Algorithm

This algorithm was designed by D. Coppersmith, H. Krawczyk, and Y. Mansour ("The Shrinking Generator", *Advances in Cryptology – CRYPTO '93 Proceedings*, Springer-Verlag, 1994). It uses two Linear Feedback Shift Registers, **A** and **S**, to generate a stream of bits: the registers are run in parallel until register **S** generates a **1** bit, at which point the bit generated simultaneously by register **A** is used as the next bit in the stream.

The Compact BasicCard implements this algorithm with Linear Feedback Shift Registers **A** and **S** of length 31 and 32 respectively. In order for the system to be secure against attack with registers of this size, it is necessary to use generating polynomials **PolyA** and **PolyS** that are unknown to the attacker. To this end, we supply a program for the generation of random cryptographic keys and primitive polynomials – see **5.3.4 The Key Generator KEYGEN.EXE**.

C++ source code for the **SG-LFSR** algorithm is provided in the development kit, in the directory `BasicCrD\Source\SG-LFSR`.

8.5 Implementation of SG-LFSR in the Compact BasicCard

The BasicCard implementation uses primitive polynomials **PolyA** and **PolyS** of degree 31 and 32 respectively, and a cryptographic key **K**, all of which are known only to the two communicating

8. Encryption Algorithms

parties. (The **KEYGEN** program generates random polynomials and keys – see **5.3.4 The Key Generator KEYGEN.EXE**.) The **START ENCRYPTION** command is called to enable encryption:

Command syntax:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	10	<i>algorithm</i>	<i>key</i>	04	<i>Random number RA (4 bytes)</i>	04

Response:	ODATA	SW1	SW2
	<i>Random number RB (4 bytes)</i>	90	00

The caller and responder both contribute 4-byte random numbers to the register initialisation procedure. **RA** may take any value; for maximum security, a different **RA** should be generated for each session. **RB** is generated by the BasicCard.

To describe how the encryption mechanism is initialised, we split all the parts into two-byte words: **RA(0):RA(1)**, **RB(0):RB(1)**, and **K(0):K(1):K(2):K(3)**, where **K** is the (eight-byte) key number *key*.

Then the two registers **A** and **S** are initialised as follows:

```

A(0) = (RA(0) Xor K(0)) And &H7FFF
A(1) = RB(0) Xor K(1)
S(0) = RB(1) Xor K(2)
S(1) = RA(1) Xor K(3)

```

So the initial value of each register depends on both of the random numbers, and on the key.

Zero is an invalid initialisation value, so as a final step:

```

If A(0) = 0 And A(1) = 0 Then A(1) = 1
If S(0) = 0 And S(1) = 0 Then S(1) = 1

```

Encryption starts with the first command after the **START ENCRYPTION** command is received, and remains in effect for commands and responses until an **END ENCRYPTION** command is received (the responses to the **START ENCRYPTION** and **END ENCRYPTION** commands themselves are not encrypted). A ZC-Basic command can tell what kind of encryption is currently active, by looking at the pre-defined variables **Encryption** (the algorithm ID) and **KeyNumber**. (If encryption is currently inactive, then **Encryption** is zero.) Encryption and decryption are identical, and consist of **Xor**-ing each byte with the result of the function **SG_LFSR::GetByte()** (defined in the C++ source file `BasicCrdr\Source\SG-LFSR\sg_lfsr.cpp`).

A command has the following structure (shaded blocks are optional):

CLA	INS	P1	P2	Lc	IDATA	Le
------------	------------	-----------	-----------	-----------	--------------	-----------

Only the data field **IDATA** is encrypted. The command bytes **CLA**, **INS**, **P1**, **P2**, **Lc**, and **Le** are not encrypted, for two reasons:

- The value of these bytes is often predictable. The number of predictable bytes that are encrypted should be kept as low as possible, to make it harder to break the key.
- Compatibility with ISO standards is lost if these bytes are altered.

A response has the following structure (the shaded block is optional):

ODATA	SW1	SW2
--------------	------------	------------

Again, only the data field **ODATA** is encrypted. The status bytes **SW1** and **SW2** are not encrypted.

8.6 SG-LFSR with CRC

The **SG-LFSR** algorithm is simple to implement, and runs efficiently. However, it provides no authentication for the data it encrypts – I don't need to know the key in order to send encrypted

8.7 Encryption – a Worked Example

messages. It's true that I won't know what I'm sending, and I won't understand the response. But I could still cause problems by sending random data. If authentication is important (and it usually is), then you should use encryption algorithm **12: SG-LFSR with CRC** (Cyclic Redundancy Check). The same 2-byte CRC is used as in the **EEPROM CRC** command. 'C' source code for calculating the CRC is given in **7.4.8 The EEPROM CRC Command**.

A command has the following structure (shaded blocks are optional):

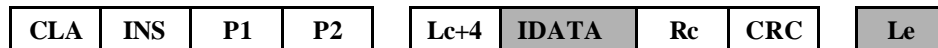


It is encrypted as follows:

- A two-byte random number **Rc** is appended to **IDATA**, and **Lc** is amended accordingly. (Without this random number, the CRC would be predictable in the case of a command with no **IDATA** field. As the CRC is later encrypted, we want to avoid this.)



- The CRC is calculated over the whole of the resulting message (**CLA INS P1 P2 Lc+2 IDATA Rc Le**). It is then appended to the two-byte random number, and **Lc** is updated accordingly.



- The resulting message is encrypted using **SG-LFSR**, as described in section 8.5.

A response has the following structure (the shaded block is optional):



It is encrypted in a similar fashion:

- A two-byte random number **Rr** is appended to **ODATA**.



- The CRC is calculated over the whole of the resulting response (**ODATA Rr SW1 SW2**), and appended to the two-byte random number.



- The resulting response is encrypted using **SG-LFSR**, as described in section 8.5.

Note: If status bytes **SW1 SW2** indicate an error (i.e. **SW1SW2** <> **swCommandOK** and **SW1** <> **sw1LeWarning**), then the response is not encrypted.

8.7 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 `BasicCrd\Examples\echotest` directory.

8.7.1 The Source Code

We ran the **KEYGEN** program to generate encryption polynomials and two cryptographic keys:

```
KEYGEN TESTKEYS -K99 -K100(16) -P
```

8. Encryption Algorithms

This produced output file TESTKEYS.BAS:

```
Declare Polynomials = &H609FBB9C,&HD23B770D
Declare Key 99 = &H3E,&H1F,&HA7,&H55,&H81,&HDB,&HC3,&H25
Declare Key 100(16) = &H83,&H24,&H24,&H59,&H86,&H8B,&H8F,&H3F,_,
    &HA0,&HC4,&H1B,&HFE,&H3E,&HF4,&HE2,&H16
```

We edited this file so that it could be included in a Compact BasicCard program:

```
Declare Polynomials = &H609FBB9C,&HD23B770D
Declare Key 99 = &H3E,&H1F,&HA7,&H55,&H81,&HDB,&HC3,&H25

#IfNotDef CompactBasicCard ' 16-bit keys not allowed in Compact BasicCard
Declare Key 100(16) = &H83,&H24,&H24,&H59,&H86,&H8B,&H8F,&H3F,_,
    &HA0,&HC4,&H1B,&HFE,&H3E,&HF4,&HE2,&H16
#EndIf
```

Then we wrote a simple ZC-Basic Terminal program ECHOTEST.BAS to send encrypted **ECHO** commands. The ECHOTEST program takes a command-line parameter 0, 1, or 2.

“ECHOTEST 0” runs with no encryption:

```
Call EchoTest()
```

“ECHOTEST 1” tests **SG-LFSR** encryption in the Compact BasicCard:

```
Rem Encryption algorithm &H11 = SG-LFSR
Call StartEncryption (P1=&H11, P2=99, Rnd) : Call CheckStatus()
Call EchoTest()
Call EndEncryption() : Call CheckStatus()

Rem Encryption algorithm &H12 = SG-LFSR with CRC
Call StartEncryption (P1=&H12, P2=99, Rnd) : Call CheckStatus()
Call EchoTest()
Call EndEncryption() : Call CheckStatus()
```

“ECHOTEST 2” tests **DES** encryption in the Enhanced BasicCard:

```
Rem Encryption algorithm &H21 = Single DES
Call StartEncryption (P1=&H21, P2=99, Rnd) : Call CheckStatus()
Call EchoTest()
Call EndEncryption() : Call CheckStatus()

Rem Encryption algorithm &H22 = Triple DES (16-byte key required)
Call StartEncryption (P1=&H22, P2=100, Rnd) : Call CheckStatus()
Call EchoTest()
Call EndEncryption() : Call CheckStatus()
```

The BasicCard program ECHOCARD.BAS just includes the key file:

```
#Include TESTKEYS.BAS
```

8.7.2 The Log Files

The COMPILE.BAT batch file in the source directory creates a Terminal program image file ECHOTEST.IMG, and two BasicCard program image files COMPACT.IMG and ENHANCED.IMG:

```
\BasicCrd\WZCBasic EchoTest -OI -I\BasicCrd\Inc
\BasicCrd\WZCBasic EchoCard -OICompact.IMG -CC1 -I\BasicCrd\Inc
\BasicCrd\WZCBasic EchoCard -OEnhanced.IMG -CE1 -I\BasicCrd\Inc
```

The SIM.BAT batch file runs the EHCOTEST program three times, and creates the I/O log files PLAIN.LOG, COMPACT.LOG, and ENHANCED.LOG:

```
\BasicCrd\WZCDos -CCompact -LPlain EchoTest 0
\BasicCrd\WZCDos -CCompact -LCompact EchoTest 1
\BasicCrd\WZCDos -CEnhanced -LEnhanced EchoTest 2
```

8.7 Encryption – a Worked Example

These were the resulting log files. (*Note:* If you run the ECHOTEST program yourself, your log files will be different, due to the different random numbers generated.)

PLAIN.LOG:

```
1:  <- 3B EF 00 FF 81 31 20 45 42 61 73 69 63 43 61 72 64 20 5A 43 31 2E 31 BE
2:  -> 00 00 09 C0 14 01 00 03 61 62 63 00 BF
   <- 00 00 05 62 63 64 61 03 02
```

COMPACT.LOG:

```
3:  <- 3B EF 00 FF 81 31 20 45 42 61 73 69 63 43 61 72 64 20 5A 43 31 2E 31 BE
4:  -> 00 40 0A C0 10 11 63 04 29 23 BE 84 04 D8
   <- 00 40 06 E1 6C D6 AE 90 00 23
5:  -> 00 00 09 C0 14 01 00 03 E5 D6 30 00 DC
   <- 00 00 05 A2 A5 92 61 03 F2
6:  -> 00 40 04 C0 12 00 00 96
   <- 00 40 02 90 00 D2
7:  -> 00 00 0A C0 10 12 63 04 52 90 49 F1 04 D1
   <- 00 00 06 F1 BB E9 EB 90 00 DE
8:  -> 00 40 0D C0 14 01 00 07 92 98 33 C7 32 39 35 00 5F
   <- 00 40 09 1F C1 1C 13 8F F0 E7 61 03 62
9:  -> 00 00 09 C0 12 00 00 04 BC E2 DD C5 99
   <- 00 00 02 90 00 92
```

ENHANCED.LOG:

```
10: <- 3B EF 00 FF 81 31 20 45 42 61 73 69 63 43 61 72 64 20 5A 43 32 2E 31 BF
11: -> 00 40 0A C0 10 21 63 04 A3 F7 76 62 04 98
   <- 00 40 06 C7 F1 B5 02 90 00 57
12: -> 00 00 11 C0 14 01 00 0B D1 2D DB 39 92 7D E3 43 EA 75 C8 00 C9
   <- 00 00 0D BE BD C6 51 0C F8 C7 F3 AF A0 CF 61 03 FB
13: -> 00 40 0D C0 12 00 00 08 34 70 7C 93 08 82 9B 89 A4
   <- 00 40 02 90 00 D2
14: -> 00 00 0A C0 10 22 64 04 FF 7D EA 1A 04 EE
   <- 00 00 06 91 82 D0 F9 90 00 AC
15: -> 00 40 11 C0 14 01 00 0B C6 40 78 CA E4 BC A2 DE 79 05 29 00 CA
   <- 00 40 0D D5 EC EB E4 B8 84 90 6F 6D 0D 8D 61 03 37
16: -> 00 00 0D C0 12 00 00 08 F6 F8 43 29 1E A9 47 38 7B
   <- 00 00 02 90 00 92
```

- 1: **ATR** (Answer To Reset) from the simulated BasicCard, including the text “**BasicCard ZC1.1**”
- 2: Unencrypted **ECHO** command and response
- 3: **START ENCRYPTION** command (algorithm = **&H11**) and response
- 4: Encrypted **ECHO** command and response (algorithm = **&H11**)
- 5: **END ENCRYPTION** command and response
- 6: **ATR** from the simulated Compact BasicCard, as in **1** above.
- 7: **START ENCRYPTION** command (algorithm = **&H12**) and response
- 8: Encrypted **ECHO** command and response (algorithm = **&H12**)
- 9: **END ENCRYPTION** command and response
- 10: **ATR** from the simulated Enhanced BasicCard, including the text “**BasicCard ZC2.1**”
- 11: **START ENCRYPTION** command (algorithm = **&H21**) and response
- 12: Encrypted **ECHO** command and response (algorithm = **&H21**)
- 13: **END ENCRYPTION** command and response
- 14: **START ENCRYPTION** command (algorithm = **&H22**) and response
- 15: Encrypted **ECHO** command and response (algorithm = **&H22**)
- 16: **END ENCRYPTION** command and response

We will look at these commands one by one, disregarding the **T=1** parameters **NAD PCB LEN . . . LRC** in every message.

8. Encryption Algorithms

8.7.3 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	00

Response:	ODATA	SW1	SW2
	62 63 64	61	03

8.7.4 START ENCRYPTION (Algorithm = &H11)

The **Rnd** function in the Terminal program returned **RA = &H2923BE84**, and the random-number generator in the BasicCard operating system returned **RB = &HE16CD6AE**. This led to the following **START ENCRYPTION** command-response pair:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	10	11	63	04	29 23 BE 84	04

Response:	ODATA	SW1	SW2
	E1 6C D6 AE	90	00

Together with the polynomials and key 99 from file KEYS.BAS:

```
Declare Polynomials = &H609FBB9C,&HD23B770D
Declare Key 99 = &H3E,&H1F,&HA7,&H55,&H81,&HDB,&HC3,&H25
```

we now have all the data we need to initialise the SG_LFSR encryptor. As described in section 8.5, we build the **A** and **S** registers from the following two-byte words:

```
RA(0) = 2923, RA(1) = BE84
RB(0) = E16C, RB(1) = D6AE
K(0) = 3E1F, K(1) = A755, K(2) = 81DB, K(3) = C325
```

Then

```
A(0) = (RA(0) Xor K(0)) And &H7FFF = 173C
A(1) = RB(0) Xor K(1) = 4639
S(0) = RB(1) Xor K(2) = 5775
S(1) = RA(1) Xor K(3) = 7DA1
```

Now the Terminal program operating system initialises its **SG-LFSR** encryptor, first with the polynomials **PolyA** and **PolyS**, and then with the registers **A** and **S**:

```
SG_LFSR Encryptor (0x609FBB9CL, 0xD23B770DL) ;
Encryptor.Initialise (0x173C4639L, 0x57757DA1L) ;
```

(C++ source code for the SG_LFSR class is provided in the development kit – see 8.4 The SG-LFSR Algorithm.) The **IDATA** and **ODATA** sections of subsequent commands and responses will be encrypted by **Xor**-ing them with successive bytes returned by `Encryptor.GetByte()`. The initialisation values given here generate the sequence:

```
84 B4 53 C0 C6 F6...
```

8.7.5 Encrypted ECHO Command (Algorithm = &H11)

From the above sequence, the **IDATA** and **ODATA** sections of the encrypted **ECHO** command and response will be:

```
61 Xor 84 = E5, 62 Xor B4 = D6, 63 Xor 53 = 30
62 Xor C0 = A2, 63 Xor C6 = A5, 64 Xor F6 = 92
```


So the **ECHO** command and response will be:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	14	01	00	03	E5 D6 30	00

Response:	ODATA	SW1	SW2
	A2 A5 92	61	03

8.7.6 END ENCRYPTION

Before calling **START ENCRYPTION** for algorithm **&H12**, the **END ENCRYPTION** command must be called to cancel the currently enabled encryption. It has no **IDATA** or **ODATA** field, so it is not affected by encryption algorithm **&H11**:

Command:	CLA	INS	P1	P2
	C0	12	00	00

Response:	SW1	SW2
	90	00

8.7.7 START ENCRYPTION (Algorithm = &H12)

This time, the **Rnd** function in the Terminal program returned **RA = &H529049F1**, and the random-number generator in the BasicCard operating system returned **RB = &HF1BBE9EB**. This led to the following **START ENCRYPTION** command-response pair:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	10	12	63	04	52 90 49 F1	04

Response:	ODATA	SW1	SW2
	F1 BB E9 EB	90	00

We repeat the process from section 8.7.4 to generate the new **A** and **S** registers:

```

RA(0) = 5290, RA(1) = 49F1
RB(0) = F1BB, RB(1) = E9EB
K(0) = 3E1F, K(1) = A755, K(2) = 81DB, K(3) = C325

A(0) = (RA(0) Xor K(0)) And &H7FFF = 6C8F
A(1) = RB(0) Xor K(1) = 56EE
S(0) = RB(1) Xor K(2) = 6830
S(1) = RA(1) Xor K(3) = 8AD4

```

So the Terminal program operating system re-initialises its **SG-LFSR** encryptor:

```
Encryptor.Initialise (0x6C8F56EEL, 0x68308AD4L) ;
```

and the sequence generated this time is

```
F3 FA 50 74 94 0F 45 7D A2 78 C8 B3 82 61 3B EE 99 40...
```

8.7.8 Encrypted ECHO Command (Algorithm = &H12)

The unencrypted **ECHO** command:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	14	01	00	03	61 62 63	00

8. Encryption Algorithms

- Add a two-byte random number **Rc**, and set **Lc = 05**:

CLA	INS	P1	P2	Lc	IDATA	Rc	Le
C0	14	01	00	05	61 62 63	B3 A6	00

- Add the CRC calculated over **C0 14 01 00 05 61 62 63 B3 A6 00**, and set **Lc = 07**:

CLA	INS	P1	P2	Lc	IDATA	Rc	CRC	Le
C0	14	01	00	07	61 62 63	B3 A6	36 70	00

- Encrypt **IDATA Rc CRC** with the **SG-LFSR** sequence **F3 FA 50 74 94 0F 45** to get the final version:

CLA	INS	P1	P2	Lc	IDATA	Rc	CRC	Le
C0	14	01	00	07	92 98 33	C7 32	39 35	00

The unencrypted response to the **ECHO** command:

Response:	ODATA	SW1	SW2
	62 63 64	61	03

- Add a two-byte random number **Rr**:

ODATA	Rr	SW1	SW2
62 63 64	DB 3C	61	03

- Add the CRC calculated over **62 63 64 DB 3C 61 03**:

ODATA	Rr	CRC	SW1	SW2
62 63 64	DB 3C	72 86	61	03

- Encrypt **ODATA Rr CRC** with the **SG-LFSR** sequence **7D A2 78 C8 B3 82 61**:

ODATA	Rr	CRC	SW1	SW2
1F C1 1C	13 8F	F0 E7	61	03

8.7.9 END ENCRYPTION

This time, the **END ENCRYPTION** command is affected by the encryption algorithm. The unencrypted **END ENCRYPTION** command:

Command:	CLA	INS	P1	P2
	C0	12	00	00

- Add a two-byte random number **Rc**, and set **Lc = 02**:

CLA	INS	P1	P2	Lc	Rc
C0	12	00	00	02	87 0C

8.7 Encryption – a Worked Example

- Add the CRC calculated over **C0 12 00 00 02 87 0C**, and set **Lc = 04**:

CLA	INS	P1	P2	Lc	Rc	CRC
C0	12	00	00	04	87 0C	44 85

- Encrypt **Rc CRC** with the **SG-LFSR** sequence **3B EE 99 40**:

CLA	INS	P1	P2	Lc	Rc	CRC
C0	12	00	00	04	BC E2	DD C5

The response is not encrypted:

Response:	SW1	SW2
	90	00

8.7.10 START ENCRYPTION (Algorithm = &H21)

This time, the **Rnd** function in the Terminal program returned **RA = &HA3F77662**, and the random-number generator in the BasicCard operating system returned **RB = &HC7F1B502**:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	10	12	63	04	A3 F7 76 62	04

Response:	ODATA	SW1	SW2
	C7 F1 B5 02	90	00

So the initialisation vector **C₀** is loaded with **A3 F7 C7 F1 B5 02 76 62**.

8.7.11 Encrypted ECHO Command (Algorithm = &H21)

The unencrypted **ECHO** command:

Command:	CLA	INS	P1	P2	Lc	IDATA	Le
	C0	14	01	00	03	61 62 63	00

- Add two zeroes:

CLA	INS	P1	P2	Lc	IDATA	Le	
C0	14	01	00	03	61 62 63	00	00 00

- Now we must encrypt the plaintext message **P = C0 14 01 00 05 61 62 63 00 00 00** using the **Single DES** message encryption function **ME_K**. Referring back to **8.2.1 The Message Encryption Functions ME_K and ME_K³**:

K = 3E 1F A7 55 81 DB C3 25

C₀ = A3 F7 C7 F1 B5 02 76 62

P₁ = C0 14 01 00 03 61 62 63

P₂ = 00 00 00 (00 00 00 00 00)

m = 5

is key number 99 from TESTKEYS.BAS;

from the **START ENCRYPTION** command;

is the first message block;

is the second message block;

is the length of padding required in **P₂**.

8. Encryption Algorithms

So we compute (you can check these in ZC-Basic, using the **DES** function):

$$C_1 = E_K(C_0 \text{ Xor } P_1) = E_K(63 \text{ E3 C6 F1 B6 63 14 01}) = \text{D1 2D DB 19 3E 80 B1 FB}$$

$$C_2 = E_K(C_1 \text{ Xor } P_2) = E_K(\text{D1 2D DB 19 3E 80 B1 FB}) = \text{39 92 7D E3 43 EA 75 C8}$$

and we throw away the last **m** bytes of C_1 to get:

$$C = ME_K(P) = \text{D1 2D DB 39 92 7D E3 43 EA 75 C8}$$

- To get the final version, **C** is wrapped in the original **CLA INS P1 P2 . . . Le**, with **Lc** adjusted appropriately:

CLA	INS	P1	P2	Lc	C	Le
C0	14	01	00	0B	D1 2D DB 39 92 7D E3 43 EA 75 C8	00

The unencrypted response to the **ECHO** command:

Response:	ODATA	SW1	SW2
	62 63 64	61	03

- Add six zeroes:

ODATA	SW1	SW2	
62 63 64	61	03	00 00 00 00 00 00

- Encrypt **P** = 62 63 64 61 03 00 00 00 00 00 00 using ME_K , where

$$K = 3E \text{ 1F A7 55 81 DB C3 25}$$

$$C_0 = 39 \text{ 92 7D E3 43 EA 75 C8}$$

$$P_1 = 62 \text{ 63 64 61 03 00 00 00}$$

$$P_2 = 00 \text{ 00 00 (00 00 00 00)}$$

$$m = 5$$

is key number 99 from TESTKEYS.BAS;

is C_2 from the **ECHO** command just received;

is the first message block;

is the second message block;

is the length of padding required in P_2 .

So we compute:

$$C_1 = E_K(C_0 \text{ Xor } P_1) = E_K(5B \text{ F1 19 82 40 EA 75 C8}) = \text{BE BD C6 6C 9E B0 59 F2}$$

$$C_2 = E_K(C_1 \text{ Xor } P_2) = E_K(\text{BE BD C6 6C 9E B0 59 F2}) = \text{51 0C F8 C7 F3 AF A0 CF}$$

and we throw away the last **m** bytes of C_1 to get:

$$C = ME_K(P) = \text{BE BD C6 51 0C F8 C7 F3 AF A0 CF}$$

- Now the original **SW1-SW2** are appended, to get:

C	SW1	SW2
BE BD C6 51 0C F8 C7 F3 AF A0 CF	61	03

8.7.12 END ENCRYPTION

The unencrypted **END ENCRYPTION** command:

Command:	CLA	INS	P1	P2
	C0	12	00	00

8.7 Encryption – a Worked Example

- Add **Lc' = 00**, **Le' = 00**, and two zeroes:

CLA	INS	P1	P2	Lc'	Le'	
C0	12	00	00	00	00	00 00

- Encrypt **P = C0 12 00 00 00 00 00 00** with **ME_K**, where

K = 3E 1F A7 55 81 DB C3 25
C₀ = 51 0C F8 C7 F3 AF A0 CF
P₁ = C0 12 00 00 00 00 00 00
m = 0

is key number 99 from TESTKEYS.BAS;
 is **C₂** from the **ECHO** response;
 is the only message block;
 is the length of padding required in **P₁**.

So we compute:

$$C_1 = E_K(C_0 \text{ Xor } P_1) = E_K(91 \ 1E \ F8 \ C7 \ F3 \ AF \ A0 \ CF) = 34 \ 70 \ 7C \ 93 \ 08 \ 82 \ 9B \ 89$$

and **C = ME_K(P)** is simply **C₁**.

- The final version:

CLA	INS	P1	P2	Lc	C
C0	12	00	00	08	34 70 7C 93 08 82 9B 89

(**Le** is not appended in this case, because it wasn't present in the unencrypted command.)

The response is not encrypted:

Response:

SW1	SW2
90	00

8.7.13 Triple DES (Algorithm = &H22)

The three commands (**START ENCRYPTION**, **ECHO**, and **END ENCRYPTION**) are encrypted in exactly the same way for **Triple DES** as for **Single DES**, with two exceptions:

- **Triple DES** requires a 16-byte key, so key number 100 is used instead of key number 99;
- the **Triple DES** message encryption function **ME_K³** is substituted for **ME_K**.

9. The ZC-Basic Virtual Machine

Note: Throughout this chapter, **bold** numbers are hexadecimal.

9.1 The BasicCard Virtual Machine

9.1.1 The Compact BasicCard

The Compact BasicCard contains **100** bytes of RAM (= 256 in decimal), and **3E0** bytes of EEPROM (= 992 in decimal). Of this, the operating system uses the first **47** bytes of RAM and the first **23** bytes of EEPROM. The memory available for use by an application written in ZC-Basic is thus **B9** bytes of RAM and **3BD** bytes of EEPROM.

9.1.2 The Enhanced BasicCard

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.)

9.1.3 Memory Layout in the BasicCard

RAM and EEPROM are divided into regions, 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 and file system work-space (Enhanced BasicCard only)	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 Enhanced BasicCard programs that use Single-to-String conversion – see **3.22.4 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 **10.4 Map File Format**.

9.2 The Terminal Virtual Machine

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)		

9.3 The P-Code Stack

The P-Code Virtual Machine has three registers:

PC	Program counter (2 bytes)
SP	Stack Pointer (BasicCard: 1 byte; Terminal: 2 bytes)
FP	Frame Pointer (BasicCard: 1 byte; Terminal: 2 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 $\mathbf{SP} + \text{frame-size} + 1$ (BasicCard) or $\mathbf{SP} + \text{frame-size} + 2$ (Terminal)
- $\mathbf{FP} = \mathbf{SP}$
- $\mathbf{SP} = \mathbf{SP} + \text{frame-size}$

The last instruction in every procedure is

LEAVE

This undoes the effect of the **ENTER** instruction before returning to the caller:

- $\mathbf{SP} = \mathbf{FP} - 1$ (BasicCard) or $\mathbf{FP} - 2$ (Terminal)
- Pop **FP**
- Pop **PC**

9.4 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

9. The ZC-Basic Virtual Machine

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			RAMHEAP block (BasicCard)		RAMHEAP block (Terminal)	
F	T	Len (14 bits)	F	Len (7 bits)	F	Len (15 bits)
Data area (Len bytes)			Data area (Len bytes)		Data area (Len bytes)	

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.

9.5 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
REAL	4-byte IEEE-format floating-point number
STRING	See <i>Strings</i> below

These types correspond to the ZC-Basic data types **Byte**, **Integer**, **Long**, **Single**, and **String** respectively. Arithmetic operations are provided for **WORD**, **LONG**, and **REAL** data; **CHAR** data must be converted to **WORD** before performing arithmetic on it.

9.5.1 Strings

There are two types of string: variable-length and fixed-length.

- A variable-length string is a 2-byte pointer to a Pascal-type string, which consists of a length byte followed by the string contents.
- A fixed-length string is a sequence of characters, whose length is known at compile time.

Both types are restricted to 254 bytes in length; if an operation would result in a longer string, it truncates the result.

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 2-byte pointer, 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 2-byte pointer, 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 2-byte pointer, in the FRAME region, to a Pascal-type string, which may be in RAMHEAP or EEPHEAP .

String parameters A **String** parameter takes up 3 bytes on the stack: a one-byte *length* followed by a two-byte *address*. If *length* ≤ 254 , the address points directly to a fixed-length string. If *length* = 255, the address is a handle, and points to a variable-length string variable. (This is the reason for the 254-byte length restriction on all strings.)

9.6 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*, or second character *c*, *w*, *l*, denote **REAL**, **CHAR**, **WORD**, and **LONG** data respectively.
- *A* is the address of an array descriptor.
- *X*\$, *Y*\$, *Z*\$ are **STRING**s.

9.6.1 Miscellaneous Instructions

Name	OpCode	Param	Description
NOP	00		No operation
ADDSP	01	<i>scDelta</i>	SP += <i>scDelta</i> . If <i>scDelta</i> > 0, 'pushed' bytes are initialised to zero.
DUP	02	<i>ucLen</i>	Push the top <i>ucLen</i> stack bytes
COMPL	03		Pop <i>slY</i> ; pop <i>slX</i> ; compare; push for WORD comparison
RAND	04		Push a LONG random number
ERROR	05	<i>ucError</i>	Generate a P-Code error condition
SYSTEM	06	<i>ucSysCode</i>	Operating system call – see 9.7 The SYSTEM Instruction .

9.6.2 Data Conversion Instructions

Name	OpCode	Description
CVTCW	07	Pop <i>ucX</i> ; <i>swY</i> = <i>ucX</i> ; push <i>swY</i>
CVTWC	08	Pop <i>swX</i> ; <i>ucY</i> = <i>swX</i> ; push <i>ucY</i>
CVTWL	09	Pop <i>swX</i> ; <i>slY</i> = <i>swX</i> ; push <i>slY</i>
CVTLW	0A	Pop <i>slX</i> ; <i>swY</i> = <i>slX</i> ; push <i>swY</i>

9. The ZC-Basic Virtual Machine

9.6.3 Data Access Instructions (Push and Pop)

Name	OpCode	Param	Description
PUCCB	0B	<i>ucConst</i>	Push constant CHAR <i>ucConst</i>
PUCWB	0C	<i>scConst</i>	Push constant <i>scConst</i> sign-extended to WORD
PUCWC	0D	<i>ucConst</i>	Push constant <i>ucConst</i> zero-extended to WORD
PUCWW	0E	<i>swConst</i>	Push constant WORD <i>swConst</i>
PURCB	0F	<i>ucAddr</i>	Push CHAR at address <i>ucAddr</i>
PURWB	10	<i>ucAddr</i>	Push WORD at address <i>ucAddr</i>
PURLB	11	<i>ucAddr</i>	Push LONG at address <i>ucAddr</i>
PURSB	12	<i>ucAddr</i>	Push STRING at address <i>ucAddr</i>
PUECW	13	<i>uwAddr</i>	Push CHAR at address <i>uwAddr</i>
PUEWW	14	<i>uwAddr</i>	Push WORD at address <i>uwAddr</i>
PUELW	15	<i>uwAddr</i>	Push LONG at address <i>uwAddr</i>
PUESW	16	<i>uwAddr</i>	Push STRING at address <i>uwAddr</i>
PUFCB	17	<i>scAddr</i>	Push CHAR at address FP + <i>scAddr</i>
PUFWB	18	<i>scAddr</i>	Push WORD at address FP + <i>scAddr</i>
PUFLB	19	<i>scAddr</i>	Push LONG at address FP + <i>scAddr</i>
PUFSB	1A	<i>scAddr</i>	Push STRING at address FP + <i>scAddr</i>
PUFAB	1B	<i>scAddr</i>	Push FP + <i>scAddr</i> as WORD
PUSAB	1C	<i>ucAddr</i>	Push SP – <i>ucAddr</i> as WORD
PUPSB	1D	<i>scAddr</i>	Push 3-byte STRING parameter at address FP + <i>scAddr</i>
PUINC	1E		Pop <i>uwAddr</i> ; push CHAR at address <i>uwAddr</i>
PUINW	1F		Pop <i>uwAddr</i> ; push WORD at address <i>uwAddr</i>
PUINL	20		Pop <i>uwAddr</i> ; push LONG at address <i>uwAddr</i>
PORCB	21	<i>ucAddr</i>	Pop CHAR at address <i>ucAddr</i>
PORWB	22	<i>ucAddr</i>	Pop WORD at address <i>ucAddr</i>
PORLB	23	<i>ucAddr</i>	Pop LONG at address <i>ucAddr</i>
POECW	24	<i>uwAddr</i>	Pop CHAR at address <i>uwAddr</i>
POEWW	25	<i>uwAddr</i>	Pop WORD at address <i>uwAddr</i>
POELW	26	<i>uwAddr</i>	Pop LONG at address <i>uwAddr</i>
POFCB	27	<i>scAddr</i>	Pop CHAR at address FP + <i>scAddr</i>
POFWB	28	<i>scAddr</i>	Pop WORD at address FP + <i>scAddr</i>
POFLB	29	<i>scAddr</i>	Pop LONG at address FP + <i>scAddr</i>
POINC	2A		Pop <i>uwAddr</i> ; pop CHAR at address <i>uwAddr</i>
POINW	2B		Pop <i>uwAddr</i> ; pop WORD at address <i>uwAddr</i>
POINL	2C		Pop <i>uwAddr</i> ; pop LONG at address <i>uwAddr</i>

9.6.4 Integer Arithmetic Instructions

Name	OpCode	Description
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 $swX \text{ Mod } swY$
MODL	36	Pop slY ; pop slX ; push $slX \text{ Mod } slY$
ANDW	37	Pop uwY ; pop uwX ; push $uwX \text{ And } uwY$
ANDL	38	Pop ulY ; pop ulX ; push $ulX \text{ And } ulY$
ORW	39	Pop uwY ; pop uwX ; push $uwX \text{ Or } uwY$
ORL	3A	Pop ulY ; pop ulX ; push $ulX \text{ Or } ulY$
XORW	3B	Pop uwY ; pop uwX ; push $uwX \text{ Xor } uwY$
XORL	3C	Pop ulY ; pop ulX ; push $ulX \text{ Xor } ulY$
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)

9. The ZC-Basic Virtual Machine

9.6.5 Program Control Instructions

(In the **ENTER** and **LEAVE** instructions, F denotes the size of the FP register: 1 in the BasicCard, 2 in the Terminal.)

Name	OpCode	Param	Description
CALL	45	$uwAddr$	Procedure call or GoSub : push PC+3 as WORD ; PC = $uwAddr$
ENTER	46	$ucFrmSiz$	Push FP ; push SP + $ucFrmSiz + F$; FP = SP ; SP = SP + $ucFrmSiz$
LEAVE	47		Return from procedure: SP = FP - F ; pop FP ; pop PC
RETURN	48		Return from GoSub : pop PC
JUMPB	49	$scDisp$	PC = PC + $scDisp + 2$
JUMPW	4A	$uwAddr$	PC = $uwAddr$
JZRWB	4B	$scDisp$	Pop swX ; if $swX = 0$ then PC = PC + $scDisp + 2$
JNZWB	4C	$scDisp$	Pop swX ; if $swX \neq 0$ then PC = PC + $scDisp + 2$
JEQWB	4D	$scDisp$	Pop swY ; pop swX ; if $swX = swY$ then PC = PC + $scDisp + 2$
JNEWB	4E	$scDisp$	Pop swY ; pop swX ; if $swX \neq swY$ then PC = PC + $scDisp + 2$
JLEWB	4F	$scDisp$	Pop swY ; pop swX ; if $swX \leq swY$ then PC = PC + $scDisp + 2$
JGTWB	50	$scDisp$	Pop swY ; pop swX ; if $swX > swY$ then PC = PC + $scDisp + 2$
JGEWB	51	$scDisp$	Pop swY ; pop swX ; if $swX \geq swY$ then PC = PC + $scDisp + 2$
JLTWB	52	$scDisp$	Pop swY ; pop swX ; if $swX < swY$ then PC = PC + $scDisp + 2$
LOOP	53	$scDisp$	Pop swX ; if $swX \geq 0$ then execute JLEWB else execute JGEWB
EXIT	54		Exit the Virtual Machine

9.6.6 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 ($swI1, swI2, \dots, swIn$)
CHKDIM	56	$ucNdims$	Pop A ; push A ; if $\text{Dim}(A) \neq ucNdims$ then execute ERROR 0C
ALLOCA	57		Pop A ; pop bounds word $uwBr$ 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
FREEAS	59		Pop string array A ; free all strings in A ; if Dynamic then deallocate A
BOUND A	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

9.6.7 String Instructions

Name	OpCode	Description
COPY\$	5D	Pop $X\$$; pop $Y\$$; $X\$ = Y\$$
FREE\$	5E	Pop 2-byte handle 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 $Y\$$; $X\$ = \text{UCASE}\$(Y\$)$
LCASE\$	66	Pop $X\$$; pop $Y\$$; $X\$ = \text{LCASE}\$(Y\$)$
STRING\$	67	Pop $X\$$; pop $ucChar$; pop $swLen$; $X\$ = \text{String}\$(swLen, ucChar)$
STRL\$	68	Pop $X\$$; pop slX ; $X\$ = \text{Str}\(slX)
HEX\$	69	Pop $X\$$; pop slX ; $X\$ = \text{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 $X\$$; $slVal = \text{Val}\&\$(X$, ucLen)$; push $slVal$; push $ucLen$
VALHL\$	6E	Pop $X\$$; $slVal = \text{ValH}\$(X$, ucLen)$; push $slVal$; push $ucLen$

9.6.8 Data Initialisation Instructions

Name	OpCode	Params	Description
RDATA	6F	$ucAddr, ucLen, data$	Copy <i>data</i> ($ucLen$ bytes) to address $ucAddr$
FDATA	70	$scAddr, ucLen, data$	Copy <i>data</i> ($ucLen$ bytes) to address FP + $scAddr$

9. The ZC-Basic Virtual Machine

9.6.9 Floating-Point Instructions

Note: These instructions are not implemented in the Compact BasicCard.

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\$ = \text{Str}\(rX)
VALR\$	7E	Pop $X\$$; $rVal = \text{Val}\!(X\$, ucLen)$; push $rVal$; push $ucLen$

9.6.10 The XMIT Command Call Instruction

Note: This instruction is available only in a Terminal program.

Name	OpCode	Params	Description
XMIT	7F	$ucType, ucLen$	Send command and process response

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
-----	-----	----	----	----	-------------------------------	----

Then the command is transmitted according to $ucType$, as follows:

$ucType$	
0	Send Lc bytes in IDATA (no Le)
1	Send Lc bytes in IDATA , followed by Le
2	The top 3 bytes of the IDATA field contain a variable-length string parameter $X\$$. Send $ucLen - 3$ bytes in IDATA , followed by $X\$$.
3	The same as $ucType = 2$, with Le appended to IDATA .
4	The top 3 bytes of the IDATA field contain a variable-length string parameter $X\$$. Send up to Lc bytes of ($ucLen - 3$ bytes followed by $X\$$).
5	The same as $ucType = 4$, with Le appended to IDATA .

9.6.11 Abbreviated Instructions

Instructions from **80** to **FF** are single-byte abbreviations of 2-byte **PUExB** / **POExB** instructions. For example, **PUFLF1** (instruction **A6**) is an abbreviation of **PUFLB F1**.

Name	OpCode	Description
PUFWED – PUFWFC	80-8F	Push WORD at address FP – (93 – OpCode)
PUFW00 – PUFW0F	90-9F	Push WORD at address FP + (OpCode – 90)
PUFLEB – PUFLFA	A0-AF	Push LONG at address FP – (B5 – OpCode)
PUFL00 – PUFL0F	B0-BF	Push LONG at address FP + (OpCode – B0)
POFWED – POWWFC	C0-CF	Pop WORD at address FP – (D3 – OpCode)
POFW00 – POWW0F	D0-DF	Pop WORD at address FP + (OpCode – D0)
POFLEB – POFLFA	E0-EF	Pop LONG at address FP – (F5 – OpCode)
POFL00 – POFL0F	F0-FF	Pop LONG at address FP + (OpCode – F0)

9.7 The SYSTEM Instruction

The **SYSTEM** P-Code instruction (OpCode **06**) calls an operating system function, according to the first parameter, *SysCode*.

9.7.1 SYSTEM Functions in the Compact BasicCard

The Compact BasicCard has just three **SYSTEM** functions:

<i>OpCode</i>	<i>SysCode</i>	<i>Name</i>	
06	00	WTX	Send a Waiting Time Extension request
06	01	CommandString	Convert a command parameter to a variable-length string
06	02	ResponseString	Convert a variable-length string to a response parameter

9.7.2 SYSTEM Functions in the Enhanced BasicCard

The Enhanced BasicCard has five **SYSTEM** functions with *SysCode* < **80**:

<i>OpCode</i>	<i>SysCode</i>	<i>Name</i>	
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	55	Key	Built-in Key() function

In addition, the Enhanced BasicCard supports the **FILE SYSTEM** functions – see **9.7.4 FILE SYSTEM Functions**.

9. The ZC-Basic Virtual Machine

9.7.3 *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 ZeitControl Chipi [®] 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

In addition, the Terminal supports the **FILE SYSTEM** functions listed in the next section.

9.7.4 *FILE SYSTEM Functions*

The file system functionality in the Terminal and the Enhanced BasicCard is implemented through the **SYSTEM** P-Code instruction. Such **FILE SYSTEM** commands all have *SysCode* \geq **80**:

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 <i>n</i> th matching filename
06	86	EraseFile	Delete a data file
06	87	RenameFile	Rename or move a file or directory

9.7 The SYSTEM Instruction

OpCode SysCode Name

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 SINGLE 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 SINGLE 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 SINGLE value to a sequential file
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)

10. 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 **ZCDOS** and **BCLOAD** programs.
- Debug file: symbolic debugging information, for use by the **ZCDD** Double Debugger.
- 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.

10.1 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. 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
'SECA'	32-byte security area
'EEPR'	EEPROM Image Section – EEPSYS , CMDTAB , PCODE , STRCON , KEYTAB , EEPDATA , and EEPHEAP regions
'LIBR'	Libraries Section – Plug-In Library directory

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
'LIBR'	Libraries Section – Plug-In Library directory

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.

10.1.1 Signature Section

<i>Length</i>	
4	'ZCIF' (“ZeitControl Image File”)
4	Total length of all remaining sections (= file length – 8)

10.1.2 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

10.1.3 Virtual Machine Type Section

Length

4	‘VMTP’
4	Section length = 02
1	Virtual Machine type: 00 = Terminal, 01 = Compact BasicCard, 02 = Enhanced BasicCard
1	Virtual Machine sub-type (00 for Terminal; 00 to 04 for BasicCards)

See **1.5 BasicCard Versions** for a list of BasicCard version numbers.

10.1.4 32-byte security area (BasicCard only)

Length

4	‘SECA’
4	Section length <i>len</i>
<i>len</i>	Data (to be padded with FF to a length of 20 bytes)

The Security Area of the BasicCard is a 32-byte area containing 16 bytes of factory-programmed ROM plus 16 bytes of PROM. This section is not downloaded to the BasicCard – it is for the simulated BasicCard in the PC.

10.1.5 P-Code Section (Terminal only)

Length

4	‘CODE’
4	Section length <i>len</i>
2	Program entry point
<i>len-2</i>	P-Code. The P-Code in the Terminal starts at address 0000 .

10. Output File Formats

10.1.6 Data Section (Terminal only)

<i>Length</i>	
4	'DATA'
4	Section length
2	Start address of RAM data
2	Length of RAM data
2	Number of records n
2	Start address of record 0
2	Length len_0 of record 0
len_0	Contents of record 0
...	
2	Start address of record $n - 1$
2	Length len_{n-1} of record $n - 1$
len_{n-1}	Contents of record $n - 1$

All RAM bytes not contained in a record must be initialised to **00**.

The Data Section contains the **RAMSYS**, **STRCON**, **RAMDATA**, and **RAMHEAP** regions.

10.1.7 EEPROM Image Section

<i>Length</i>	
4	'EEPR'
4	Section length
2	Start address of EEPROM data
2	Length of EEPROM data
2	Number of records n
2	Start address of record 0
2	Length len_0 of record 0
len_0	Contents of record 0
...	
2	Start address of record $n - 1$
2	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.

10.1.8 Libraries Section

Length

4	‘LIBR’
4	Section length
2	String table length len_{ST}
len_{ST}	String table
2	Number of modules n
	Module 0
...	
	Module $n-1$

Each module sub-section contains the following information:

Length

2	Name of module
1	Major version number of module
1	Minor version number of module
1	Major version number of interpreter needed to execute this module
1	Minor version number of interpreter needed to execute this module
2	Start address of module
2	End address of module
2	Number of Address List Entries n
2	Address List Index 0
2	Address 0
...	
2	Address List Index $n-1$
2	Address $n-1$

Note: The Libraries Section is for the various PC-based interpreters – the BCLOAD program ignores it. Each ZeitControl Plug-In Library defines its own list of addresses that the interpreter needs to know; these are the Address List Entries.

10.2 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 **‘VMTP’** Virtual Machine Type Section:

‘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

10. Output File Formats

10.2.1 Signature Section

Length

4	'ZCDF' ("ZeitControl Debug File")
4	Total length of all remaining sections (= file length – 8)

10.2.2 Files Section

This section contains the names of all the source files in the program:

Length

4	'FILE'
4	Section length
2	String table length len_{ST}
len_{ST}	String table
2	Number of files n
2	Name of file 0
...	
2	Name of file $n - 1$

10.2.3 Types Section

This section contains definitions of every data type that occurs in the program.

Length

4	'TYPE'
4	Section length
2	String table length len_{ST}
len_{ST}	String table
2	Number of type entries n
7	Type 0
...	
7	Type $n - 1$

Type format (shaded bytes are zero):

Byte	0					
Integer	1					
Long	2					
Single	3					
String	4					
String*n	5	n				
<i>Array</i>	6	<i>ElementType</i>	<i>nDims</i>			
<i>UserType</i>	7	<i>TypeName</i>	<i>nMembers</i>			
<i>Member</i>	8	<i>MemberName</i>	<i>MemberType</i>	<i>Offset</i>		

<i>ElementType, MemberType</i>	Indices of types in the Types section
<i>TypeName, MemberName</i>	Offsets in the string table
<i>nDims</i>	Number of dimensions of the array
<i>nMembers</i>	Number of members in the user-defined type
<i>Offset</i>	Offset of the member in its user-defined type <i>UserType</i>

A *UserType* entry is immediately followed by *nMembers* type entries of type *Member*.

10.2.4 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 `strcmp`). User-defined symbols are stored in upper case; symbols containing lower-case letters are compiler-generated names.

<i>Length</i>	
4	‘SYMB’
4	Section length
2	Procedure start address (0000 for the global Symbols Section)
2	Procedure end address (0000 for the global Symbols Section)
2	String table length len_{ST}
len_{ST}	String table
2	Number of symbols n
8	Symbol 0
...	
8	Symbol $n - 1$

Symbol format (shaded bytes are zero):

Const Long	0	<i>SymbolName</i>	4-byte integer		
Const Single	1	<i>SymbolName</i>	4-byte floating-point number		
Const String	2	<i>SymbolName</i>	<i>String</i>	<i>Len</i>	
<i>Label</i>	3	<i>SymbolName</i>	Address		
<i>Variable</i>	4	<i>SymbolName</i>	Address	<i>Type</i>	<i>Storage</i>

<i>SymbolName, String</i>	2-byte offsets in the string table
<i>Type</i>	Index in the Types section
<i>Storage</i>	0 = 2-byte absolute 1 = 1-byte absolute 2 = 1-byte signed, FP-relative (procedure parameters, Private data) 3 = indirect 1-byte signed, FP-relative (String and array parameters)

10. Output File Formats

10.2.5 Line Numbers Section

Line-number entries are sorted in increasing code address order.

Length

4	'LINE'
4	Section length
2	Number of line-number entries n
10	Line-number entry 0
...	
10	Line-number entry $n - 1$

Line-number entry format:

Code address (2 bytes)	File number (2 bytes)	Line number (2 bytes)	File position (4 bytes)
------------------------	-----------------------	-----------------------	-------------------------

10.2.6 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

4	'FIXU'
4	Section length
2	Number of entries in Labels table $nLabs$
6	Label entry 0
...	
6	Label entry $nLabs - 1$
2	Number of entries in Operands table $nOps$
6	Operand entry 0
...	
6	Operand entry $nOps - 1$

Label entries and Operand entries have the same format:

Code address (2 bytes)	Symbols Section (2 bytes)	Index of symbol in Symbols Section (2 bytes)
------------------------	---------------------------	--

10.3 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 Compact BasicCard version ZC1.2, with list file and map file requested:

```
ZCBASIC MONTHLEN -CC2 -OL -OM
```

The list file, **MONTHLEN.LST**:

```

❶ File: MONTHLEN.BAS
  ❷ 1 Declare ApplicationID = "Small Example Program"
❸ InitCode:
  ❹ PCODE      ❺ A849:❻ 46 00  ❷ ENTER 00
    PCODE      A84B: 6F 80 01 RDATA 80 01
                      FF          FF
    PCODE      A84F: 47          LEAVE
ApplicationID:
  ❸ EEPROMDATA A86A: 15 53 6D 61 6C 6C 20 45 78 61 6D 70 6C 65 20 50
    EEPROMDATA A87A: 72 6F 67 72 61 6D
  2 Eeprom MonthLength(1 To 12) = 31,28,31,30,31,30,31,31,30,31,30,31
❹ MONTHLENGTH:
    EEPROMDATA A880: A8 88 02 01 04 0B 00 18
MONTHLENGTH DATA:
    EEPROMDATA A888: 00 1F 00 1C 00 1F 00 1E 00 1F 00 1E 00 1F 00 1F
    EEPROMDATA A898: 00 1E 00 1F 00 1E 00 1F
  3 Const InvalidMonth = &H6F01
  4
  5 Command &H80 &H00 GetMonthLength (N)
GETMONTHLENGTH:
    PCODE      A850: 46 00      ENTER 00
    CMDTAB     A843: 01 80 00 02 A8 50
  6      If N < 1 Or N > 12 Then
    PCODE      A852: 8F          PUFWFC (N) ❶
    PCODE      A853: 0C 01      PUCWB 01
    PCODE      A855: 52 05      JLTWB True001
    PCODE      A857: 8F          PUFWFC (N)
    PCODE      A858: 0C 0C      PUCWB 0C
    PCODE      A85A: 4F 06      JLEWB Else001
  7          SW1SW2 = InvalidMonth
True001:
    PCODE      A85C: 0E 6F01    PUCWW 6F01
    PCODE      A85F: 22 45      PORWB SW1SW2
  8      Else
    PCODE      A861: 54          EXIT
  9          N = MonthLength (N)
Else001:
    PCODE      A862: 8F          PUFWFC (N)
    PCODE      A863: 0E A880    PUCWW MONTHLENGTH

```

10. Output File Formats

```

        PCODE      A866:  55          ARRAY
        PCODE      A867:  1F          PUINW
        PCODE      A868:  CF          POFWFC (N)
10      End If
11      End Command
        PCODE      A869:  54          EXIT

```

- ❶ Input filename
- ❷ Source code, with line number
- ❸ Compiler-generated label (contains lower-case letters)
- ❹ P-Code (**PCODE** is the name of the region)
- ❺ Address of P-Code instruction
- ❻ P-Code instruction and operands, in hexadecimal
- ❼ P-Code instruction and operands, in text
- ❽ Eeprom data (**EEPDATA** is the name of the region)
- ❾ User-generated label (upper-case letters)
- ❿ Implicit operand of abbreviated P-Code instruction, in parentheses

10.4 Map File Format

The map file **MONTHLEN.MAP** from the example program in the previous section, **10.3 List File Format**:

- ❶ Input file: MONTHLEN.BAS
- ❷ ===== RAM regions =====

Name	Start	End	Length
RAMSYS	00	4B	4C
STACK	4C	7F	34
RAMDATA			00
RAMHEAP	80	FF	80

- ❸ ===== EEPROM regions =====

Name	Start	End	Length
EEPSYS	A820	A842	0023
CMDTAB	A843	A848	0006
PCODE	A849	A869	0021
STRCON			0000
KEYTAB			0000
EEPDATA	A86A	A89F	0036
EEPHEAP	A8A0	ABFF	0360

- ❹ ===== Symbols by name =====

Name	Scope	Address	Type
CLA	Global	47	PUBLIC BYTE
ENCRYPTION	Global	23	PUBLIC BYTE
FALSE	Global		CONST=0000
GETMONTHLENGTH	Global	A850	COMMAND &H80 &H00
INS	Global	48	PUBLIC BYTE
INVALIDMONTH	Global		CONST=6F01
KEYNUMBER	Global	40	PUBLIC BYTE
LC	Global	4B	PUBLIC BYTE
LE	Global	44	PUBLIC BYTE
MONTHLENGTH	Global	A880	EEPROM INTEGER ARRAY

MONTHLENGTH DATA	Global	A888	ARRAY DATA
N	GETMONTHLENGTH	FC	PARAM INTEGER
P1	Global	49	PUBLIC BYTE
P1P2	Global	49	PUBLIC INTEGER
P2	Global	4A	PUBLIC BYTE
PCODEERROR	Global	41	PUBLIC BYTE
RESPONSELENGTH	Global	43	PUBLIC BYTE
SW1	Global	45	PUBLIC BYTE
SW1SW2	Global	45	PUBLIC INTEGER
SW2	Global	46	PUBLIC BYTE
TRUE	Global		CONST=FFFF

5 ===== Symbols by location =====

RAM system data:

Name	Scope	Address	Type
----	-----	-----	----
ENCRYPTION	Global	23	PUBLIC BYTE
KEYNUMBER	Global	40	PUBLIC BYTE
PCODEERROR	Global	41	PUBLIC BYTE
RESPONSELENGTH	Global	43	PUBLIC BYTE
LE	Global	44	PUBLIC BYTE
SW1SW2	Global	45	PUBLIC INTEGER
SW1	Global	45	PUBLIC BYTE
SW2	Global	46	PUBLIC BYTE
CLA	Global	47	PUBLIC BYTE
INS	Global	48	PUBLIC BYTE
P1P2	Global	49	PUBLIC INTEGER
P1	Global	49	PUBLIC BYTE
P2	Global	4A	PUBLIC BYTE
LC	Global	4B	PUBLIC BYTE

EEPROM user data:

Name	Scope	Address	Type
----	-----	-----	----
MONTHLENGTH	Global	A880	EEPROM INTEGER ARRAY
MONTHLENGTH DATA	Global	A888	ARRAY DATA

6 User code:

Name	Scope	Address	Type
----	-----	-----	----
Initialisation Code	Global	A849	SUB
GETMONTHLENGTH	Global	A850	COMMAND &H80 &H00

7 Local variables:

Name	Scope	Address	Type
----	-----	-----	----
N	GETMONTHLENGTH	FC	PARAM INTEGER

- 1 Input filename.
- 2 RAM regions: The addresses and lengths of the regions in RAM.
- 3 EEPROM regions: The addresses and lengths of the regions in EEPROM.
- 4 Symbols by name: All the symbols in alphabetical order.
- 5 Symbols by location: All the symbols, ordered according to location and address.
- 6 User code: The addresses of all the procedures and labels in the source program.
- 7 Local variables: The signed FP-relative addresses of parameters and **Private** data.

Index

A

Abs	31
Access Conditions	57
ACos Mathematical Function	82
Algorithm	38, 41
Answer To Reset	37, 86
Append mode	53
Application ID	37
Array Descriptor Format	43
Array Functions	32
Array Parameters	31
Arrays	18
As type	19
Asc	32
ASin Mathematical Function	82
ASSIGN NAD	104
Assignment Statements	23
At address	19
ATan Mathematical Function	82
ATan2 Mathematical Function	82
ATR	37, 86
ATR Declaration	37
Attributes	51
Automatic Encryption	41

B

BasicCard	6
BasicCard Versions	9
BasicCard Virtual Machine	122
BasicCard-Specific Features	37
BCKEYS.EXE	69
BCLOAD.EXE	67
BgCol	41
Binary Files	56, 57
Binary mode	53
Block Waiting Time	16, 86
Breakpoints	75
Built-in Commands	91
Built-in Functions	31
BWT	16, 86
Byte data type	17

C

Call	29
Card Loader	67
Card State	15
CardInReader	39
CardReader	39
Ceil Mathematical Function	82
Certificate	33, 36
Certificate Generation	35, 111
ChDir	49
ChDrive	52
Chr\$	32

CLA	28, 38
Class byte	28, 38
CLEAR EEPROM	95
Close File	55
Close Log File	40
Cls	38
Command Calls	30
Command Definition	27
Command-response protocol	5
COMMANDS.DEF	106
Communications	39, 86
Compact BasicCard	4
ComPort	41
Computed GoTo/GoSub	26
Conditional Compilation	14
Cos Mathematical Function	82
CosH Mathematical Function	83
CRC	98
Create File	53
CurDir	50
CurDrive	52
Current Disk Drive	52
CursorX	41
CursorY	41
Custom Lock	58

D

Data Declaration	18
Data Storage	16
Data Types	17
Data Types, P-Code	124
Date	40
Debug File Format	137
Debug File, Generating	64
Debugger	70
Declare ApplicationID	37
Declare ATR	37
Declare Key	33
Declare Polynomials	34
DefByte	42
DefInt	42
DefLng	42
DefSng	42
DefString	42
DefType Statement	42
Delete File	53
DES Algorithm	108
DES Encryption Primitives	35
Dir	51, 59
Directory Attributes	51
Directory Commands	49
Directory Definition	59
Directory Names	45
Directory-Based File Systems	45
Disable Encryption	37, 41

Disable Key	35	For-Loop	25
Disable OverflowChec	42	FreeFile	59
Disk Drive	52	Function Calls	29
Do-Loop	25	Function Definition	27
Dynamic arrays	18		
E		G	
EC-160 Library	76	GET APPLICATION ID	100
EC160GenerateKeyPair	77	Get Lock	58
EC160HashAndSign	78	GET STATE	93
EC160HashAndVerify	78	GetAttr	52
EC160SessionKey	79	GoSub.....	24
EC160SetCurve	77	GoTo	24
EC160SetPrivateKey	78		
EC160SharedSecret	78	H	
EC160Sign	78	Hex\$	32
EC160Verify	78	Hexadecimal Constants	12
ECHO	103	Hypot Mathematical Function.....	82
EEPROM CRC	98	I	
Eeprom data	16	I/O Logging	40
EEPROM Size	16	I-block (T=1 protocol)	87
EEPROM SIZE	94	If-Then-Else.....	24
Elliptic Curve Library	76	Image File Format	134
Enable Encryption	37, 41	Image File, Generating	64
Enable Key	35	Image Files	10
Enable OverflowCheck	42	Implementing Encryption.....	33
Encryption.....	33	Initialisation Code	7
Encryption Algorithms	108	InKey\$	39
Encryption Functions	33	Input	39, 56
END ENCRYPTION	102	Input mode	53
Enhanced BasicCard	4	INS	28, 38
EOF	59	Installation of Support Software	62
Error Counter.....	34	Instruction byte.....	28, 38
Error Directive.....	16	Integer data type	17
Error File, Generating	64		
Error Handling.....	37	K	
Executable File, Generating	64	Key Configuration.....	34
Executable Files	10	Key Declaration	33
Exit	23	Key Error Counter.....	34
Exp Mathematical Function.....	82	Key Generator	68
Expressions	20	Key Loader.....	69
		Keyboard Input.....	39
F		KEYGEN.EXE.....	68
fa... File Attributes	52	KeyNumber	38, 41
FastEepromWrites	83	Kill	53
fe... File System Errors	48		
FgCol	41	L	
File	60	Labels	24
File Attributes	51	LBound	32
File Definition	60	Lc	38
File Definition Sections	8	LCase\$	32
FILE IO	105	Le	38
File Names	45	Left\$	32
File System Commands	47	Len (of data).....	32
FileError	38, 41	Len (of file).....	59
FILEIO.DEF	60	LibError	76
Files and Directories	45	Libraries	76
Fixed arrays	18	Line Input	39, 56
Floor Mathematical Function.....	82	List File Format.....	141
Folders	45	List File, Generating.....	64

ZeitControl BasicCard

Listing Directives	15
LOAD state	91
Lock	58
Lock File	57
Log10 Mathematical Function	82
LogE Mathematical Function	82
Long data type	17
LTrim\$	32

M

Map File Format	142
Map File, Generating	64
MATH Library	82
Memory Allocation	44
Message Decryption Functions	109
Message Encryption Functions	109
Mid\$	32
MISC Library	83
MkDir	49

N

Name	50
NEW state	91
nParams	42
Numerical Expressions	21
Numerical Functions	31

O

Octal Constants	12
Open File	53
Open File Slots	15
Open Log File	40
Option Base	42
Option Explicit	43
Output File Formats	134
Output mode	53
Overflow Checking	42

P

P1	38
P1P2	38
P2	38
Param\$	42
Parameter Passing	30
Path Names	45
pc... P-Code Errors	90
PC/SC Functions	40
P-Code Instructions	125
P-Code Interpreter	66
P-Code Stack	123
PCodeError	38, 41
Peek	33
Permanent Data	8, 11
Plug-In Libraries	76
Poke	33
Polynomial Declaration	34
Pow Mathematical Function	82
Pre-Defined Commands	91
Pre-Defined Constants	16
Pre-Defined Files	47

Pre-Defined Variables	38, 41
Pre-Processor Directives	14
Print	38, 55
Private data	17
Procedure Calls	28
Procedure Declaration	28
Procedure Definition	27
Procedure Definitions	7
Procedure Parameters	30
Processor Cards	4
Program Layout	6
Programming processor card	5
Public data	17
Put	56

R

Random Files	56, 57
Random mode	53
Random Number Generation	36
Randomize	36
READ EEPROM	97
Read From Files	56
Read Key File	34
Read Lock	57
Read Unlock	57
Read Write Lock	57
Read Write Unlock	57
ReDim	18
Renaming Files	50
Reserved words	13
ResetCard	39
ResponseLength	38, 41
Return	24
Right\$	32
Rmdir	49
Rnd	31, 36
ROM code	4
RTrim\$	32
RUN state	91
Run-Time Memory Allocation	123

S

Save Eeprom Data	40
Screen Output	38
Searching for Files	51
Secure Hash Algorithm	81
Seek	59
Select Case	26
Sequential Files	55, 56
SET STATE	99
SetAttr	51
SG-LFSR	111
SG-LFSR with CRC	112
SHA-1 Library	81
ShaAppend	81
ShaEnd	81
ShaHash	81
ShaRandomHash	81
ShaRandomSeed	81
ShaStart	81

Shrinking Generator	111	TEST state.....	91
Sin Mathematical Function.....	82	Time\$	40
Single data type	17	Tokens	12
SinH Mathematical Function.....	83	Trim\$	32
<i>Skip to I/O</i>	74		
<i>Skip to Terminal</i>	74	U	
Sleep	83	UBound	32
Source File	12	UCase\$	32
Source File Inclusion.....	14	Unlock	58
Space\$	32	Unlock File.....	57
Sqrt	31	User-Defined Parameters	31
Stack Size	15	User-Defined Types	20
START ENCRYPTION	101		
States of the BasicCard	91	V	
Static data	17	Val!	32
<i>Step Basic</i>	74	Val&	32
<i>Step In</i>	74	ValH	32
<i>Step Over</i>	74	Versions of the BasicCard.....	9
Storage Requirements	47	Virtual Machine	122
Str\$	32		
String data type	17	W	
String Expressions	22	Watch Variables	75
String Functions	32	WBCKEYS.EXE.....	69
String Parameter Format.....	43	WBCLOAD.EXE.....	67
String Parameters	31	While-Loop.....	25
String\$	32	Write	55
String*n data type	17	Write Eeprom	40
Strings, P-Code.....	124	WRITE EEPROM	96
Subroutine Calls	29	Write Lock	57
Subroutine Definition.....	27	Write to file	55
Support Software.....	62	Write Unlock	57
sw... Status Codes	89	WTX Request.....	87
SW1	38, 89	WTX Statement	37, 41
SW1SW2	38, 42	WZCBASIC.EXE.....	64
SW2	38, 89	WZCDOS.EXE.....	66
SYSTEM Instruction.....	131		
T		Z	
T=1 Protocol.....	86	ZC-Basic Compiler.....	64
Tan Mathematical Function.....	82	ZC-Basic language	12
TanH Mathematical Function.....	83	ZCBASIC.EXE.....	64
Terminal Program.....	10	ZCDD.EXE.....	70
Terminal Program Layout.....	10	ZCDOS.EXE.....	66
Terminal Virtual Machine	122	ZeitControl Double Debugger.....	70
Terminal-Specific Features	38		