MEGA65

USER'S GUIDE

MUSEUM OF ELECTRONIC GAMES & ART

REGULATORY INFORMATION

The MEGA65 home computer and portable computer have not been subject to FCC, EC or other regulatory approvals as of the time of writing.

MEGA65 USER'S GUIDE

Published by
the MEGA Museum of Electronic Games and Art, Germany.
and
Flinders University, Australia.

WORK IN PROGRESS

November 6, 2019

Contents

CHAPTER 1

Introduction

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130
131 132 133 134 135 136 137

Congratulations on your purchase of one of the most long-awaited computers in the history of computing. The MEGA65 is a community designed computer, based on the never-released Commodore® 65[1] computer, that was first designed in 1989, and intended for public release in 1990. Twenty-eight years have passed since then, but the simple, friendly nature of the 1980s home computers is still something that hasn't been recreated. These were computers that were simple enough that you could understand not just how to work with your computer, but how computers themselves work.

Many of the people who grew up using the home computers of the 1980s now have exciting and rewarding jobs in many companies, in part because of what they learnt about computers in the comfort of their own home. We want to give you that same opportunity, to experience the joy of learning how to use computers to solve all sorts of problems: writing a letter to a friend, working out how much tax you owe, inventing new things, or discovering how the universe works. This is why we made the **MEGA65**.

The MEGA65 team thinks that owning a computer should be like owning a home: You don't just use a home, you change things big and small to really make it your own, and maybe even renovate it or add on a room or two. In this guide we will show you how to more than just hang your own pictures on the wall, but instead how you can dream up new ways of using the powerful capabilities of computers by coding your own computer programmes, and even changing the computer itself!

To help you have fun with your MEGA65, we will show you how to use the exciting **graphics** and **sound** capabilities of the MEGA65. But the MEGA65 isn't just about writing your own programmes. It can also run many of the thousands of games and other programmes that were created for the Commodore® 64™[2] computer.

Welcome to the world of the **MEGA65**!

[1] Commodore is a trademark of C= Holdings

[2] Commodore 64 is a trademark of C= Holdings,

PART I

GETTING TO KNOW YOUR MEGA65

CHAPTER 2

SETUP

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130

UNPACKING AND CONNECTING THE MEGA65

Time to set up your MEGA65 home computer. The box contains the following:

- MEGA65 computer.
- Power supply (black box with socket for mains supply).
- This book, the MEGA65 User's Guide.

In addition, to be able to use your MEGA65 computer:

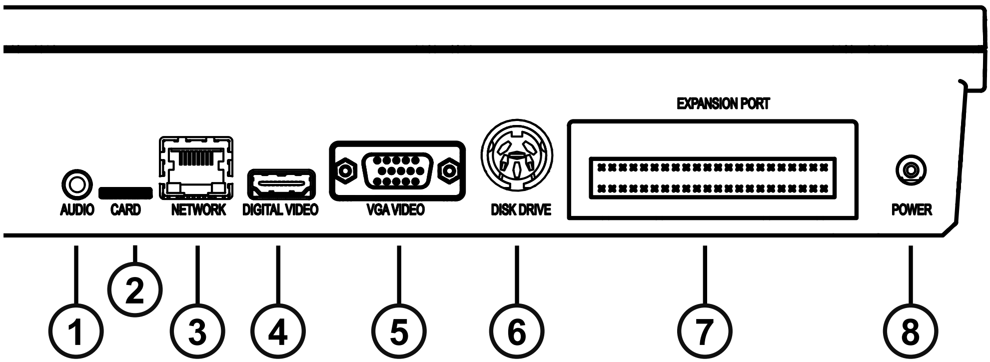
- A television or computer monitor with a VGA or digital video input, that is capable of displaying an image with 800x600 pixel resolution at 50Hz or 60Hz.
- A VGA video cable, or;
- A digital video cable.

These items are not included with the MEGA65.

You may also want to use the following to get the most out of your MEGA65:

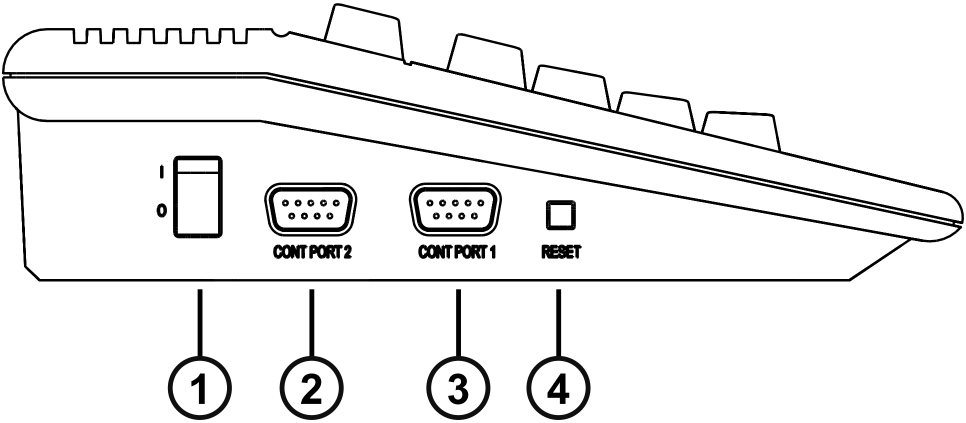
- 3.5mm mini-jack audio cable and suitable speakers or hifi system, so that you can enjoy the sound capabilities of your MEGA65.
- RJ45 ethernet cable (regular network cable) and a network router or switch. This allows use of the high-speed networking capabilities of your MEGA65.

REAR CONNECTIONS



1	3.5mm Audio Mini-Jack
2	SDCard
3	Network LAN Port
4	Digital Video Connector
5	VGA Video Connector
6	External Floppy Disk Drive
7	Cartridge Expansion Port
8	DC Power-In Socket

SIDE CONNECTIONS

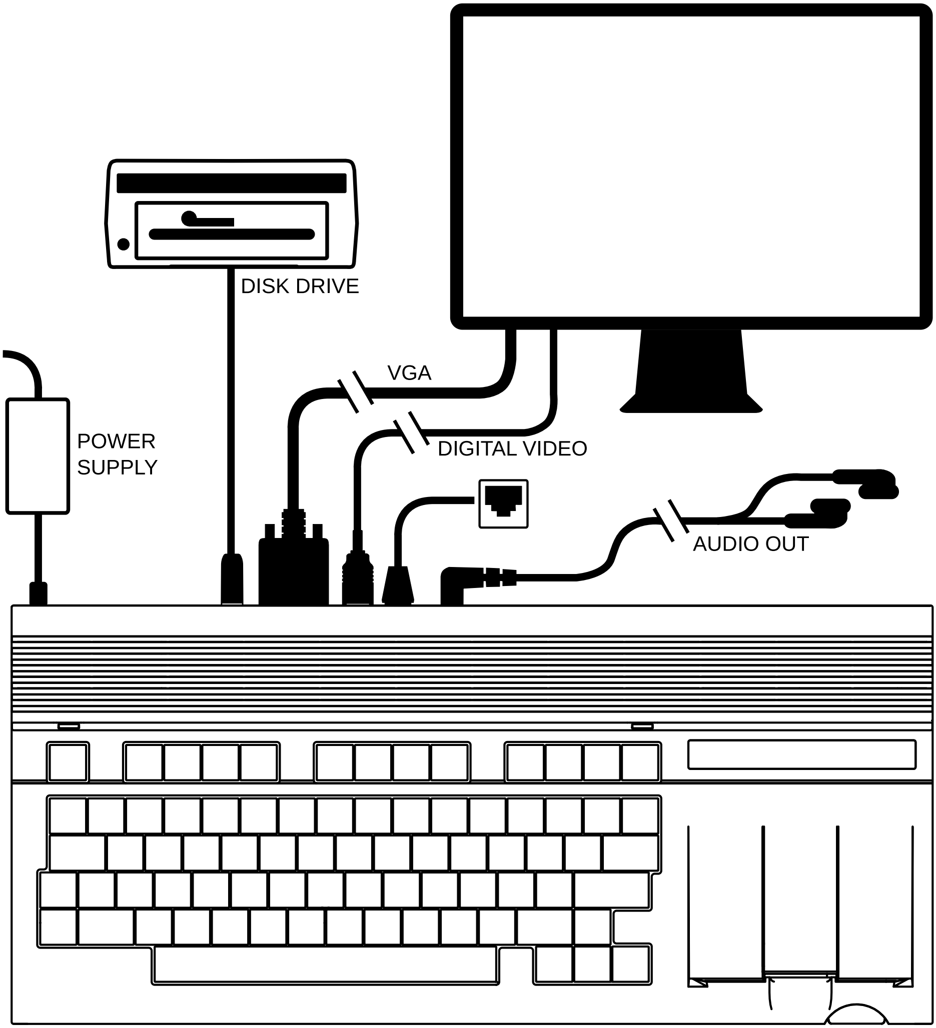


1	Power Switch
2	Controller Port 2
3	Controller Port 1
4	Reset Button

Various peripherals can be connected to Controller Ports 1 and 2 such as joysticks or paddles.

INSTALLATION

Connecting your MEGA65 to a screen and peripherals



1. Connect the power supply to the Power Supply socket of the MEGA65.

2. If you have a VGA monitor and a VGA cable, connect one end to the VGA port of the MEGA65 and the other end into your VGA monitor.

3. If you have a TV or monitor with a Digital Video connector, connect one end of your cable to the Digital Video port of the MEGA65, and the other into the Digital Video port of your monitor. If you own a monitor with a DVI socket, you can purchase a DVI to Digital Video adapter.

OPTIONAL CONNECTIONS

1. The MEGA65 houses an internal 3.5" floppy disk drive. You can also connect older Commodore® IEC serial floppy drives to the MEGA65: the Commodore® 1541, 1571 or 1581. Connect one end of your IEC cable to the Commodore® floppy disk drive and the other end to the Disk Drive socket of the MEGA65. You can also connect SD2IEC devices and PI1541's. It is possible to daisy-chain additional floppy disk drives or Commodore® compatible printers.

2. You can connect your MEGA65 to a network using a standard ethernet cable.

3. For enjoying audio from your MEGA65, you can connect a 3.5mm stereo mini-jack cable to an audio amplifier or speaker system. If your system has RCA connectors you will need to purchase a 3.5mm mini-jack to twin RCA adapter cable. The MEGA65 also has a built in amplifier to allow connecting headphones.

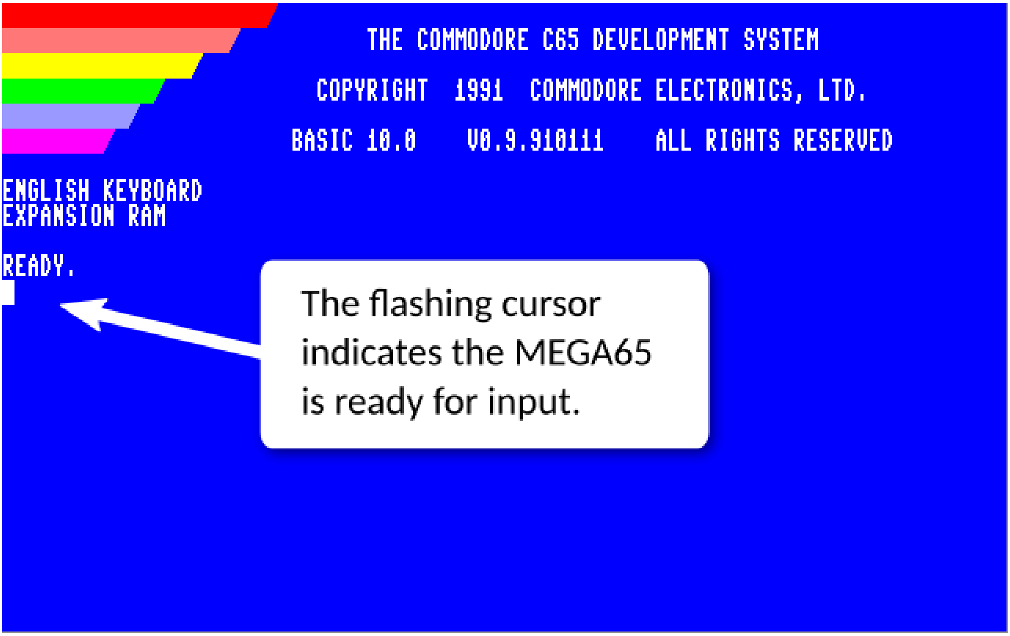
4. A Secure Digital Card or SDCard (SDHC and SDXC) can be inserted into the rear of the MEGA65 as a drive.

OPERATION

Using the MEGA65

1. Turn on the computer by using the switch on the left hand side of the MEGA65.

2. After a moment, the following will be displayed on your TV or monitor:



THE CUSROR

The flashing square underneath the READY prompt is called the cursor. The cursor indicates that the computer is ready to accept input. Pressing keys on the keyboard will print that character onto the screen. The character will be printed in the current cursor position, and then the cursor advances to the next position.

You can type commands, for example: telling the computer to load a program. You can even start entering program code.

CHAPTER 3

GETTING STARTED

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130
131 132 133 134 135 136 137

KEYBOARD

Now that you have everything connected, it's time to get familiar with the MEGA65 keyboard.

You may notice that the keyboard is a little different from the standard used on computers today. While most keys will be in familiar positions, there are some specialised keys, and some with special graphic symbols marked on the front.

Here's a brief description of how some of these special keys function.

Command Keys

The Command Keys are: RETURN, SHIFT, CTRL, and RESTORE.

RETURN

Pressing the RETURN key enters the information you have typed into the MEGA65's memory. The computer will either act on a command, store some information, or return you an error if you made a mistake.

SHIFT

The two SHIFT keys are located on the left and the right. They work very much like Shift on a regular keyboard, however they also perform some special functions too.

In upper case mode, holding SHIFT and pressing any key with a graphic symbol on the front produces the right hand symbol on that key. For example, SHIFT and J prints the character.

In lower case mode, pressing SHIFT and a letter key prints the upper case letter on that key.

Holding both shift keys down when turning the machine on activates the Utility Menu. You can format the SD card or enter the MEGA65 Configuration Utility to select the default video mode and other settings.

Finally, holding the SHIFT key and pressing a Function key accesses the function shown on the front of that key. For example: SHIFT and F1 activates **F2**.

SHIFT LOCK

In addition to the Shift key is SHIFT LOCK. Press this key to lock down the Shift function. Now any key you press prints the character to the screen as if you were holding down Shift. That includes special graphic characters.

CTRL

CTRL is the Control key. Holding down Control and pressing another key allows you to perform Control Functions. For example, holding CTRL and one of the number keys allows you to change text colours.

There are some examples of this in the Screen Editor chapter, and all the Control Functions are listed in Appendix C Control codes.

If a program is being listed to the screen, holding CTRL slows down the display of each line on the screen.

Holding CTRL and pressing * enters the Matrix Mode Debugger.

RUN/STOP

Normally, pressing the RUN STOP key stops execution of a program. Holding SHIFT while pressing RUN STOP loads the first program from disk.

Programs are able to disable the RUN STOP key.

You can boot your machine into the machine code monitor by holding down RUN STOP and pressing reset on the MEGA65.

RESTORE

The computer screen can be restored to a clean state without clearing the memory by holding down the RUN STOP key and tapping RESTORE.

Programs are able to disable this key combination.

Enter the Freeze Menu by holding RESTORE for more than one second. You can access the machine code monitor via the Freeze menu.

THE CURSOR KEYS

At the bottom right hand of the keyboard are the cursor keys. These four directional keys allow you move the cursor to any position for onscreen editing.

The cursor moves in the direction indicated on the keys: ← ↑ → ↓

However, it is also possible to move the cursor up using SHIFT and ↓. In the same way you can move the cursor left using SHIFT and →.

You don't have to keep pressing a cursor key over and over. When moving the cursor a long way, you can keep the key pressed in. When you are finished, release the key.

INSerT/DELete

This is the INSERT / DELETE key. When pressing INST DEL, the character to the left is deleted, and all characters to the right are shifted one position to the left.

To insert a character, hold the SHIFT key and press INST DEL. All the characters are shifted to the right. This allows you to type a letter, number or any other character into the newly inserted space.

CLeaR/HOME

Pressing the CLR HOME key returns the cursor into the top left-most position of the screen.

If holding SHIFT and pressing CLR HOME clears the entire screen and places the cursor into the top left-most position of the screen.

MEGA KEY

The key or the MEGA key provides a number of different functions and special utilities.

Holding the SHIFT key and pressing switches between lower and upper case character modes.

Holding and pressing any key with graphic symbols on the front prints the left-most graphic symbol to the screen.

Holding and pressing any key that shows a single graphic symbol on the front prints that graphic symbol to the screen.

Holding and pressing a number key switches to one of the colours in the second range.

Holding and pressing **TAB** enters the Matrix Mode Debugger.

When turning on the MEGA65 or pressing the reset button on the side, while holding switches the MEGA65 into C64 mode.

NO SCROLL

If a program is being listed to the screen, pressing **NO SCRL** freezes the screen output. Not available in C64 mode.

Function Keys

There are seven Function keys available for use by software applications, **F1** **F3** **F5** **F7** **F9** **F11** and **F13** to perform functions with a single press.

Hold **SHIFT** to access **F2** through to **F14** as shown on the front of each Function key.

Only Function keys **F1** to **F8** are available in C64 mode.

HELP

The **HELP** key can be used by software and acts as an **F15** / **F16** key.

ALT

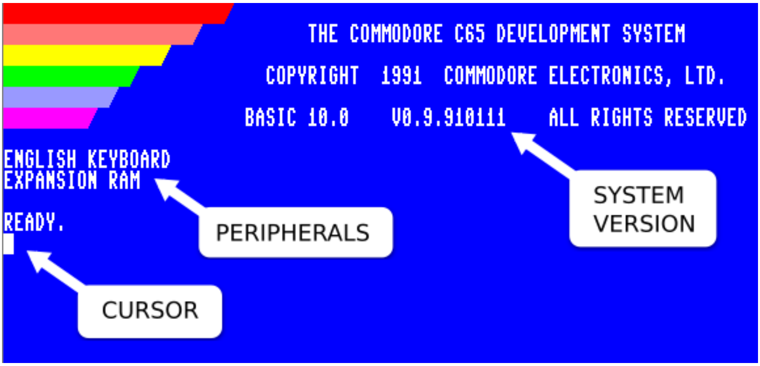
The **ALT** held while pressing other keys can be used by software to perform functions. Not available in C64 mode.

CAPS LOCK

The **CAPS LOCK** works like **SHIFT LOCK** in C65 and MEGA65 modes, but only modifies the alphabet keys. Also, holding the **CAPS LOCK** down forces the processor to run at the maximum speed. This can be used, for example, to speed up loading from the internal disk drive or SD card, or to greatly speed up the depacking process after a program is run. This can reduce the loading and depacking time from many seconds to as little as a 10th of a second.

THE SCREEN EDITOR

When your turn on your MEGA65, or reset it, the editor screen will appear.



The colour bars in the top left hand of the screen can be used as a guide to help calibrate the colours of your display. The screen also displays the name of the system, the copyright notice and what version and revision of BASIC is contained in the Read-only Memory.

Also displayed is the type of keyboard and whether or not there is additional hardware present, such as a RAM expansion.

Finally, you will see the READY prompt and the flashing cursor.

You can begin typing keys on the keyboard and the characters will be printed under the cursor. The cursor itself advances after each key press.

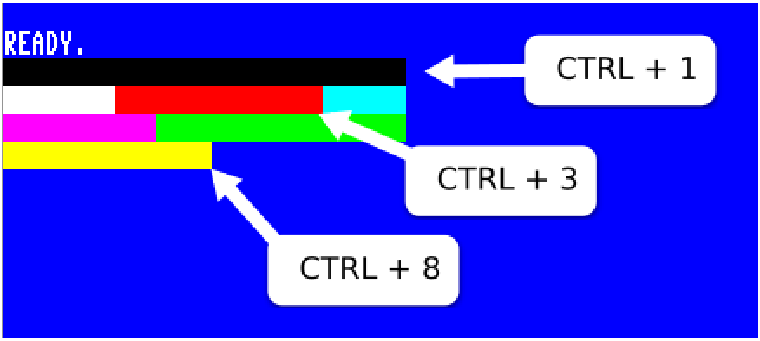
You can also produce reverse text or colour bars by holding down the **CTRL** key and pressing the **9** key or the **R** key. This enters reverse text mode.

Try holding down the **SPACE BAR**. A white bar will be drawn across the screen.

You can even change the current colour by holding down the **CTRL** key and pressing a number key. Try key **8** and then hold down the **SPACE BAR** again. A yellow bar will be drawn.

Change the bar to a number of other colours.

You will get an effect something like:



You can turn off the reverse text mode by holding CTRL and pressing the 0 key.

By pressing any keys, the characters will be typed out in the chosen colour.

There are a further eight colours available via the M key. Hold the M key and press a key from 1 to 8 to change to one of the secondary colours. For even more colours, see Escape Sequences in Appendix C.

Functions

Functions using the CTRL key are called **Control Functions**. Functions using the M key are called **Mega Functions**. There are also functions called by using the SHIFT key. These are (not surprisingly) called **Shift Functions**.

Lastly, using the ESC key are **Escape Sequences**.

ESC Sequences

Escape sequences are performed a little differently than a Control function or a Shifted function. Instead of holding the modifier key down, an Escape sequence is performed by pressing the ESC key once, then pressing the desired key code.

For example: to switch between 40/80 column mode, press and release the ESC key. Then press the X key.

You can see all the available Escape Sequences in Appendix C. We will cover some examples of these shortly.

There are more modes available. You can create flashing text by holding the CTRL key and pressing the O key. Any characters you press will flash. Turn flash mode off by pressing ESC then O.

EDITOR FUNCTIONALITY

The MEGA65 screen can allow you to do advanced tabbing, and moving quickly around the screen in many ways to help you to be more productive.

Press the HOME key to go to the home position on the screen. Hold the CTRL key down and press the W key several times. This is the **Word Advance function** which jumps your cursor to the next word, or printable character.

You can set custom tab positions on the screen for your convenience. Press HOME and then → to the fourth column. Hold down CTRL and press the X key to set a tab. Move another 20 positions to the right again, and do CTRL and X again to set a second tab.

Press the HOME key to go back to the home position. Hold the CTRL key and press the I key. This is the **Forward Tab function**. Your cursor will tab to the fourth position. Press CTRL and I again. Your cursor will move to position 8. Why? By default, every 8th position is already set as a tabbed position. So the 4th and 20th positions have been added to the existing tab positions. You can continue to press the CTRL and I keys to the 16th and 20th positions.

To find the complete set of Control codes, see Appendix C Control codes.

Creating a Window

You can set a window on the MEGA65 working screen. Move your cursor to the beginning of the "BASIC 10.0" text. Press **ESC**, then press **T**. Move the cursor 10 lines down and 15 to the right.

Press the **ESC** key, then **B**. Anything you type will be contained within this window.

To escape from the window back to the full screen, press the **HOME** key twice.

Extras

Long press on **RESTORE** to go into the Freeze Menu. Then press **J** to switch joystick ports without having to physically swap the joystick to the other port.

Go to **Fast mode** with poke 0, 65 or go to the freeze menu.

MEGA and **SHIFT** switches between text uppercase and lowercase for the entire display.

PART II

FIRST STEPS IN CODING

CHAPTER 4

How Computers Work

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130

Computers can do amazing things, and you can make them do amazing things for you, too. But to do that, you need to understand how computers work. This can be hard to find out these days, because computers are now so complicated that it isn't obvious just by looking at them and using them. The MEGA65 is designed to be simple enough that you can learn how computers work as you use it. But we don't want to leave you to have to work out everything for yourself. This chapter will help you to understand how computers work, and then use that knowledge to help you make computers do what *you* want them to do.

COMPUTERS ARE JUST A PILE OF SWITCHES

What are computers *really*? Well, the answer to that question is quite simple, if a little surprising: Computers really are just made of lots of switches. These switches work like the switches you use to turn a light on or off. Light switches connect or disconnect the power supply to a light. The switches in computers connect or disconnect circuits in the computer to power. But computers can do a lot more than just switch on and off, so something else must be going on. That something else is also a bit of a surprise: A computer can turn its own switches on and off by itself. Let's explore how this simple idea of switches that can turn themselves on and off makes a computer.

Computers are full of switches. When you hear people talking about microns and nanometres with regard to computers, they are often talking about the size of the little switches that make up the computer chips. These switches are called transistors. The switches in the main chip of the MEGA65, for example, are 28 nanometres long. That's about 100,000 times thinner than a single strand of hair. This is good, because a computer might need millions or billions of switches in its design.

PART III

SOUND AND GRAPHICS

PART IV

APPENDICES

APPENDICES

APPENDIX A

ACCESSORIES

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130
131 132 133 134 135 136 137

APPENDIX **B**

BASIC 10 Command Reference

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130
131 132 133 134 135 136 137

FORMAT OF COMMANDS, FUNCTIONS AND OPERATORS

This appendix describes each of the commands, functions and other callable elements of BASIC 10. Some of these can take one or more arguments, that is, pieces of input that you provide as part of the command or function call. Some also require that you use special keywords. Here is an example of how commands, functions and operators will be described in this appendix:

KEY <numeric expression>,<string expression>

In this case, KEY is what we call a **keyword**. That just means a special word that BASIC understands. Keywords are always written in CAPITALS, so that you can easily recognise them.

The **<** and **>** signs mean that whatever is between them must be there for the command, function or operator to work. In this case, it tells us that we need to have a **numeric expression** in one place, and a **string expression** in another place. We'll explain what there are a bit more in a few moments.

You might also see square brackets around something, for example, **[,numeric expression]**. This means that whatever appears between the square brackets is optional, that is, you can include it if you need to, but that the command, function or operator will work just fine without it. For example, the `CIRCLE` command has an optional numeric argument to indicate if the circle should be filled when being drawn.

The comma, and some other symbols and punctuation marks just represent themselves. In this case, it means that there must be a comma between the **numeric expression** and the **string expression**. This is what we call syntax: If you miss something out, or put the wrong thing in the wrong place, it is called a syntax error, and the computer will tell you if you have a syntax error by giving a `?SYNTAX ERROR` message.

There is nothing to worry about getting an error from the computer. Instead, it is just the computer's way of telling you that something isn't quite right, so that you can more easily find and fix the problem. Error messages like this can't hurt the computer or damage your program, so there is nothing to worry about. For example, if we accidentally left the comma out, or replaced it with a full-stop, the computer will respond with a syntax error, like this:

```
KEY 8"FISH"

?SYNTAX ERROR
```

```
KEY 8,"FISH"

?SYNTAX ERROR
```

It is very common for commands, functions and operators to use one or more **"expression"**. An expression is just a fancy name for something that has a value. This could be a string, such as `"HELLO"`, or a number, like `23.7`, or it could be a calculation, that might include one or more functions or operators, such as `LEN("HELLO") * (3 XOR 7)`. Generally speaking, expressions can result in either a string or numeric result. In this case we call the expressions either string expressions or numeric expressions. For example, `"HELLO"` is a **string expression**, while `23.7` is a **numeric expression**.

It is important to use the correct type of expression when writing your programs. If you accidentally use the wrong type, the computer will give you a `?TYPE MISMATCH ERROR`, to say that the type of expression you gave doesn't match what it expected, that is, there is a mismatch between the type of expression it expected, and the one you gave. For example, we will get a `?TYPE MISMATCH ERROR` if we type the following command, because `"POTATO"` is a string expression instead of a numeric expression:

```
KEY "POTATO","SOUP"
```

You can try typing this into the computer yourself now, if you like.

COMMANDS

Commands are statements that you can use directly from the **READY.** prompt, or from within a program, for example:

```
PRINT "HELLO"
HELLO

10 PRINT "HELLO"
RUN
HELLO
```

ABS

Token: $B6

Format: **ABS(x)**

Usage: The numeric function **ABS(x)** returns the absolute value of the numeric argument **x**.
x = numeric argument (integer or real expression).

Remarks: The result is of real type.

Example: Using **ABS**

```
PRINT ABS(-123)
123
PRINT ABS(4.5)
4.5
PRINT ABS(-4.5)
4.5
```

AND

Token: $AF

Format: operand **AND** operand

Usage: The boolean **AND** operator performs a bitwise logical AND operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16 bit integer. Logical operands are converted to 16 bit integer using $FFFF, decimal -1 for TRUE and $0000, decimal 0, for FALSE.

```
0 AND 0  ->  0
0 AND 1  ->  0
1 AND 0  ->  0
1 AND 1  ->  1
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **AND**

```
PRINT 1 AND 3
1
PRINT 128 AND 64
0
```

In most cases the **AND** will be used in **IF** statements.

```
IF (C >= 0 AND C < 256) THEN PRINT "BYTE VALUE"
```

APPEND

Token: $FE $0E

Format: **APPEND# lfn, filename [,D drive] [,U unit]**

Usage: Opens an existing sequential file of type SEQ or USR for writing and positions the write pointer at the end of the file.

lfn = **l**ogical **f**ile **n**umber
1 <= lfn <= 127: line terminator is CR
128 <= lfn <= 255: line terminator is CR LF

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: `APPEND#` functions similar to the `DOPEN#` command, except that if the file already exists, the existing content of the file will be retained, and any `PRINT#` commands made to the open file will cause the file to grow longer.

Example: Open file in append mode:

```
APPEND#5,"DATA",U9
APPEND#130,(DD$),U(UN%)
APPEND#3,"USER FILE,U"
APPEND#2,"DATA BASE"
```

ASC

Token: $C6

Format: **ASC**(string)

Usage: Takes the first character of the string argument and returns its numeric code value. The name is apparently chosen to be a mnemonic to ASCII, but the returned value is in fact the so called PETSCII code.

Remarks: **ASC** returns a zero for an empty string, which behaviour is different to BASIC 2, where ASC("") gave an error. The inverse function to **ASC** is **CHR$**.

Example: Using **ASC**

```
PRINT ASC("MEGA")
77
PRINT ASC("")
0
```

ATN

Token: $C1

Format: **ATN**(numeric expression)

Usage: Returns the arc tangent of the argument. The result is in the range ($-\pi/2$ to $\pi/2$)

Remarks: A multiplication of the result with $180/\pi$ converts the value to the unit "degrees". **ATN** is the inverse function to **TAN**.

Example: Using **ATN**

```
PRINT ATN(0.5)
 .463647609
PRINT ATN(0.5) * 180 / π
 26.5650512
```

AUTO

Token: $DC

Format: **AUTO [step]**

Usage: Enables faster typing of BASIC programs. After submitting a new program line to the BASIC editor with the RETURN key, the AUTO function generates a new BASIC line number for the entry of the next line. The new number is computed by adding **step** to the current line number.

step = line number increment

Typing **AUTO** with no argument switches this fuction off.

Example: `AUTO 10` – use AUTO with increment 10
`AUTO` – switch AUTO off

BACKGROUND

Token: $FE $3B

Format: **BACKGROUND** colour

Usage: Sets the background colour of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Example: `BACKGROUND 4` – select background colour cyan.

Colours: **Index and RGB values of colour pallette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

BACKUP

Token: $F6

Format: **BACKUP D source TO D target [,U unit]**

Usage: Used on dual drive disk units only (e.g. 4040, 8050, 8250). The backup is done by the disk unit internally.

source = drive # of source disk (0 or 1).
target = drive # of target disk (0 or 1).

Remarks: The target disk is formatted and a identical copy of the source disk is written.
This command cannot be used for unit to unit copies.

Example: `BACKUP D0 TO D1` – copy disk in drive 0 to drive 1 on unit 8 (default).
`BACKUP D1 TO D0, U9` – copy disk in drive 1 to drive 0 on unit 9.

BANK

Token: $FE $02

Format: **BANK** banknumber

Usage: Selects the memory configuration for BASIC commands, that use 16-bit addresses. These are LOAD, SAVE, PEEK, POKE, WAIT and SYS. See system memory map for details.

Remarks: A value > 127 selects memory mapped I/O. The default value for the bank number is 128.

Example: `BANK 1` – select memory configuration 1.

BEGIN

Token: $FE $18

Format: **BEGIN** ... **BEND**

Usage: The **BEGIN** and **BEND** keywords act like a pair of brackets around a compound statement to be executed after a **THEN** or **ELSE** keyword. This overcomes the single line limitation of the standard **IF** ... **THEN** ... **ELSE** clause.

Remarks: Do not jump with **GOTO** or **GOSUB** into a compound statement. It may lead to unexpected results.

Example: Using **BEGIN** and **BEND**

```
10 GET A$
20 IF A$>="A" AND A$<="Z" THEN BEGIN
30 PW$=PW$+A$
40 IF LEN(PW$)>7 THEN 90
50 BEND :REM IGNORE ALL EXCEPT (A-Z)
60 IF A$<>CHR$(13) GOTO 10
90 PRINT "PW=";PW$
```

BEND

Token: $FE $19

Format: **BEGIN** ... **BEND**

Usage: The **BEGIN** and **BEND** keywords act like a pair of brackets around a compound statement to be executed after a **THEN** or **ELSE** keyword. This overcomes the single line limitation of the standard **IF** ... **THEN** ... **ELSE** clause.

Remarks: The example below shows a quirk in the implementation of the compound statement. If the condition evaluates to **FALSE**, execution does not resume right after **BEND** as it should, but at the beginning of next line. Test this behaviour with the following program:

Example: Using **BEGIN** and **BEND**

```
10 IF Z > 1 THEN BEGIN:A$="ONE"
20 B$="TWO"
30 PRINT A$;" ";B$;:BEND:PRINT " QUIRK"
40 REM EXECUTION RESUMES HERE FOR Z <= 1
```

BLOAD

Token: $FE $11

Format: **BLOAD filename [,B bank] [,P address] [,D drive] [,U unit]**

Usage: "Binary LOAD" loads a file of type PRG into RAM at address P and bank B.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address can be used to overrule the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: If the loading process tries to load beyond the address $FFFF, an 'OUT OF MEMORY' error occurs.

Example: Using **BLOAD**

```
BLOAD "ML DATA", B0, U9
BLOAD "SPRITES"
BLOAD "ML ROUTINES", B1, P32768
BLOAD (FN$), B(BA%), P(PA), U(UN%)
```

BOOT

Token: $FE $1B

Format: **BOOT filename [,B bank] [,P address] [,D drive] [,U unit]**
BOOT SYS
BOOT

Usage: **BOOT filename** loads a file of type PRG into RAM at address P and bank B and starts executing the code at the load address.

BOOT SYS loads the boot sector from sector 0, track 1 and unit 8 to address $0400 on bank 0 and performs a JSR $0400 afterwards (Jump To Subroutine).

The **BOOT** command with no parameter tries to load and execute a file named AUTOBOOT.C65 from the default unit 8. It's short for **RUN "AUTOBOOT.C65"**.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address can be used to overrule the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: **BOOT SYS** copies the contents of one physical sector (two logical sectors) = 512 bytes from disc to RAM, filling RAM from $0400 to $05ff.

Example: Using **BOOT**

```
BOOT SYS
BOOT (FN$), B(BA%), P(PA), U(UN%)
BOOT
```

BORDER

Token: $FE $3C

Format: **BORDER** colour

Usage: Sets the border colour of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Example: `BORDER 5` – select background colour magenta.

Colours: **Index and RGB values of colour pallette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

BOX

Token: $E1

Format: **BOX X0,Y0, X1,Y1, X2,Y2, X3,Y3, SOLID**

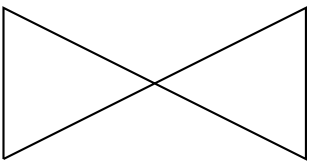
Usage: Draws a quadrangle by connecting the coordinate pairs 0 -> 1 -> 2 -> 3 -> 0. The quadrangle is drawn using the current drawing context set with SCREEN, PALETTE and PEN. The quadrangle is filled, if the parameter SOLID is 1.

Remarks: A quadrangle is a geometric figure with four sides and four angles. A box is a special form of a quadrangle, with all four angles at 90 degrees. Rhomboids, kites and parallelograms are special forms too. So the name of this command is misleading, because it can be used to draw all kind of quadrangels, not only boxes.
It is possible to draw bowtie shapes.

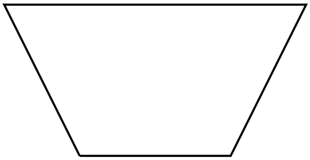
Example: Using **BOX**

```
BOX 0,0, 160,0, 160,80, 0,80
```

```
BOX 0,0, 160,80, 160,0, 0,80
```



```
BOX 0,0, 160,0, 140,80, 20,80
```



BSAVE

Token: $FE $10

Format: **BSAVE filename ,P start TO end [,B bank] [,D drive] [,U unit]**

Usage: "Binary SAVE" saves a memory range to a file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)** If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS.

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

start is the first address, where the saving begins. It becomes also the load address, that is stored in the first two bytes of the PRG file.

end Is the address, where the saving stops. **end-1** is the last address to be used for saving.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to 8.

Remarks: The length of the file is **end - start + 2**.

Example: Using **BSAVE**

```
BSAVE "ML DATA", P 32768 TO 33792, B0, U9
BSAVE "SPRITES", P 1536 TO 2058
BSAVE "ML ROUTINES", B1, P(DEC("9000")) TO (DEC("A000"))
BSAVE (FN$), B(BA%), P(PA) TO (PE), U(UN%)
```

BUMP

Token: $CE $03

Format: **b = BUMP(type)**

Usage: Used to detect sprite-sprite (type=1) or sprite-data (type=2) collisions. the return value **b** is a 8-bit mask with one bit per sprite. The bit position corresponds with the sprite number. Each bit set in the return value indicates, that the sprite for this position was involved in a collision since the last call of **BUMP**. Calling **BUMP** resets the collision mask, so you get always a summary of collisions encountered since the last call of **BUMP**.

Remarks: It's possible to detect multiple collisions, but you need to evaluate sprite coordinates then to detect which sprite collided with which one.

Example: Using **BUMP**

```
S% = BUMP(1) : REM SPRITE-SPRITE COLLISION
IF (S% AND 6) = 6) THEN PRINT "SPRITE 1 & 2 COLLISION"

S% = BUMP(2) : REM SPRITE-DATA COLLISION
IF (S% <> 0) THEN PRINT "SOME SPRITE HIT DATA REGION"
```

sprite	return	mask
0	1	0000 0001
1	2	0000 0010
2	4	0000 0100
3	8	0000 1000
4	16	0001 0000
5	32	0010 0000
6	64	0100 0000
7	128	1000 0000

BVERIFY

Token: $FE $28

Format: **BVERIFY filename [,P address] [,B bank] [,D drive] [,U unit]**

Usage: "Binary VERIFY" compares a memory range to a file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address is the address, where the comparison begins. If the parameter P is omitted, it is the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to 8 .

Remarks: **BVERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. In direct mode the command exits either with the message **OK** or with **VERIFY ERROR**. In program mode a **VERIFY ERROR** either stops execution or enters the **TRAP** error handler, if active.

Example: Using **BVERIFY**

```
BVERIFY "ML DATA", P 32768, B0, U9
BVERIFY "SPRITES", P 1536
BVERIFY "ML ROUTINES", B1, P(DEC("9000"))
BVERIFY (FN$), B(BA%), P(PA), U(UN%)
```

CATALOG

Token: $FE $0C

Format: **CATALOG [filepattern] [,R] [,D drive] [,U unit]**

Usage: Prints a listing of the specified disk.

The **R** (Recoverable) parameter includes files in the directory, which are flagged as deleted but are still recoverable.

filepattern is either a quoted string, for example: **"da*"** or a string expression in parentheses, e.g. **(DI$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The command **CATALOG** is a synonym for **DIRECTORY** or **DIR** and produces the same listing. The **filepattern** can be used to restrict the listing. The wildcard characters `*` and `?` may be used. Adding a `,T=` to the pattern string, with T specifying a filetype P,S,U or R (for PRG,SEQ,USR,REL) restricts the output to that filetype.

Example: Using **CATALOG**

```
CATALOG
  0 "BLACK SMURF     " BS  2A
508 "STORY PHOBOS"         SEQ
27  "C8096"                PRG
25  "C128"                 PRG
104 BLOCKS FREE.

DIRECTORY "*,T=S"
  0 "BLACK SMURF     " BS  2A
508 "STORY PHOBOS"         SEQ
104 BLOCKS FREE.
```

CHANGE

Token: $FE $0C

Format: **CHANGE "find" TO "replace" [,from-to]**

Usage: Used in direct mode only. It searches the line range if specified or the whole BASIC program else. At each occurrence of the "find string" the line is listed and the user prompted for an action:
'Y' <RETURN> do the change and find next string
'N' <RETURN> do **not** change and find next string
'*' <RETURN> change this and all following matches
<RETURN> exit command, don't change.

Remarks: Instead of the quote (") each other character may be used as delimiter for the findstring and replacestring. Using the quote as delimiter finds text strings, that are not tokenized and therefore not part of a keyword.
`CHANGE "LOOP" TO "OOPS"` will not find the BASIC keyword `LOOP`, because the keyword is stored as token and not as text. However `CHANGE &LOOP& TO &OOPS&` will find and replace it (probably spoiling the program).

Example: Using **CHANGE**

```
CHANGE "XX$" TO "UU$", 2000-2700
CHANGE &IN& TO &OUT&
```

CHAR

Token: $E0

Format: **CHAR column, row, height, width, direction, string [, address of character set]**

Usage: Displays text on a graphic screen. It can be used for all resolutions.

column is the start position of the output in horizontal direction. One column is 8 pixels wide, so a screen width of 320 has a column range 0 -> 39, while a width of 640 has a range of 0 -> 79.

row is the start position of the output in vertical direction. Other than column, its unit is pixel with top row having the value 0.

height is a factor applied to the vertical size of the characters. 1 is normal size (8 pixels) 2 is double size (16 pixels), and so on.

width is a factor applied to the horizontal size of the characters. 1 is normal size (8 pixels) 2 is double size (16 pixels), and so on.

direction controls the printing direction:
1: up
2: right
4: down
8: left

The optional **address of character set** can be used to select a character set different from the default character set at $29800, which is the set with upper/lower characters.

string is a string constant or expression which will be printed.

Remarks: Control characters, for example: cursor movement codes, will be ignored (neither printed nor interpreted).

Example: Using **CHAR**

```
CHAR 304,196, 1,1,2,  "MEGA 65"
```

will printh the text "MEGA 65" on the centre of a 640 x 400 graphic screen.

CHR$

Token: $C1

Format: **CHR$(numeric expression)**

Usage: Returns a string of length one character using the argument to insert the character having this value as PETSCII code.

Remarks: The argument range is 0 -> 255, so this function may also be used to insert control codes into strings. Even the NULL character, with code 0, is allowed.
CHR$ is the inverse function to **ASC**.

Example: Using **CHR$**

```
10 QUOTE$   = CHR$(34)
20 ESCAPE$  = CHR$(27)
30 PRINT QUOTE$;"MEGA 65";QUOTE$ : REM PRINT "MEGA 65"
40 PRINT ESCAPE$;"Q";       : REM CLEAR TO END OF LINE
```

CIRCLE

Token: $E2

Format: **CIRCLE xcentre, ycentre, radius, [,solid]**

Usage: A special case of the **ELLIPSE** command using the same value for horizontal and vertical radius.

xcentre x coordinate of centre in pixels.

ycentre y coordinate of centre in pixels.

radius radius of the circle in pixels.

solid will fill the circle if not zero.

Remarks: The **CIRCLE** command is used to draw circles on screens with an aspect ratio 1:1 (for example: 320 x 200 or 640 x 400). On other resolutions (like: 640 x 200) the shape will degrade to an ellipse.

Example: Using **CIRCLE**

```
10 REM USE A 640 X 400 SCREEN
20 CIRCLE 320,200,100
30 REM DRAW CIRCLE IN THE CENTRE OF THE SCREEN
```

CLOSE

Token: $A0

Format: **CLOSE channel**

Usage: Closes an input or output channel, that was established before by an **OPEN** command.

channel is a value in the range 0 -> 255.

Remarks: Closing open files before the program stops is very important, especially for output files. This command flushes output buffers and updates directory informations on disks. Failing to **CLOSE** can corrupt files and disks. BASIC does NOT automatically close channels or files when the program stops.

Example: Using **CLOSE**

```
10 OPEN 2,8,2,"TEST,S,W"
20 PRINT#2,"TESTSTRING"
30 CLOSE 2 : REM OMITTING CLOSE GENERATES A SPLAT FILE
```

CLR

Token: $9C

Format: **CLR**

Usage: Resets all pointers, that are used for management of BASIC variables, arrays and strings. The runtime stack pointers are reset and the table of open channels is reset. A **RUN** command performs **CLR** automatically.

Remarks: **CLR** should not be used inside loops or subroutines because it destroys the return address. After a **CLR** all variables are unknown and will be initialized at the next usage.

Example: Using **CLR**

```
10 A=5: P$="MEGA 65"
20 CLR
30 PRINT A;P$

0
READY.
```

CMD

Token: $9D

Format: **CMD channel [,string]**

Usage: Redirects the standard output from screen to the channel. This enables to print listings and directories or other screen outputs. It is also possible to redirect this output to a disk file or a modem.

channel must be opened by the **OPEN** command.

The optional **string** is sent to the channel before the redirection begins and can be used, for example, for printer setup escape sequences.

Remarks: The **CMD** mode is stopped by a **PRINT# channel** or by closing the channel with **CLOSE channel**. It is recommended to use a **PRINT# channel** before closing, to make sure, that the output buffer is flushed.

Example: Using **CMD** to print a program listing:

```
OPEN 4,4
LIST
PRINT#4
CLOSE 4
```

COLLECT

Token: $F3

Format: **COLLECT [,D drive] [,U unit]**

Usage: Rebuilds the **BAM** (Block Availabilty Map) deleting splat files and marking unused blocks as free.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: While this command is useful for cleaning the disk from splat files (for example: write files, that weren't properly closed) it is dangerous for disks with boot blocks or random access files. These blocks are not associated with standard disk files and will therefore be marked as free too and may be overwritten by further disk write operations.

Example: Using **COLLECT**

```
COLLECT
COLLECT U9
COLLECT D0, U9
```

COLLISION

Token: $FE $17

Format: **COLLISION type [,linenumber]**

Usage: Enables or disables an user programmed interrupt handler. A call without linenumber disables the handler, while a call with linenumber enables it. After the execution of **COLLISION** with linenumber a sprite collision of the same type, as specified in the **COLLISION** call, interrupts the BASIC program and perform a **GOSUB** to **linenumber** which is expected to contain the user code for handling sprite collisions. This handler must give control back with a **RETURN**.

type specifies the collision type for this interrupt handler:
1 = sprite - sprite collision
2 = sprite - data - collision
3 = light pen

linenumber must point to a subroutine which holds code for handling sprite collision and ends with a **RETURN**.

Remarks: It is possible to enable interrupt handler for all types, but only one can execute at any time. A interrupt handler cannot be interrupted by another interrupt handler. Functions like **BUMP**, **RSPPOS** and **LPEN** may be used for evaluation of the sprites which are involved and their positions.

Example: Using **COLLISION**

```
10 COLLISION 1,70 : REM ENABLE
20 SPRITE 1,1 : MOVSPR 1,120,  0 : MOVSPR 1,0#5
30 SPRITE 2,1 : MOVSPR 2,120,100 : MOVSPR 2,180#5
40 FOR I=1 TO 50000:NEXT
50 COLLISION 1 : REM DISABLE
50 END
70 REM SPRITE <-> SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

COLOR

Token: $E7

Format: **COLOR <ON|OFF>**

Usage: Enables or disables handling of the character attributes on the screen. If **COLOR** is **ON**, the screen routines take care for both character RAM and attribute RAM. E.g. if the screen is scrolled for text, the attributes are scrolled too, so each character keeps his attribute or colour. If **COLOR** is **OFF**, the attribute or colour RAM is fixed and character movement is only done for screen characters. This speeds up screen handling, if moving characters with different colours is not intended.

Example: `COLOR ON` – with colour/attribute handling
`COLOR OFF` – no colour/attribute handling

CONCAT

Token: $FE $13

Format: **CONCAT appendfile [,D drivea] TO targetfile [,D drive] [,U unit]**

Usage: The **CONCAT** (concatenation) appends the contents of **appendfile** to the **targetfile**. Afterwards **targetfile** contains the contents of both files, while **appendfile** remains unchanged.

appendfile is either a quoted string, for example: **"data"** or a string expression in parentheses, for example: **(FN$)**

targetfile is either a quoted string, for example: **"safe"** or a string expression in parentheses, for example: **(FS$)**

If the disk unit has dual drives, it is possible to apply the **CONCAT** command to files, which are stored on different disks. In this case, it is necessary to specify the drive# for both files in the command. This is necessary too, if both files are stored on drive#1.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The **CONCAT** commands is executed in the DOS of the disk drive. Both files must exist and no pattern matching is allowed. Only sequential files of type **SEQ** may be concatenated.

Example: Using **CONCAT**

```
CONCAT "NEW DATA" TO "ARCHIVE" ,U9
CONCAT "ADDRESS",D0 TO "ADDRESS BOOK",D1
```

CONT

Token: $9A

Format: **CONT**

Usage: Used to resume program execution after a break or stop caused by an **END** or **STOP** statement or by pressing the **STOP KEY**. This is a useful debug tool. The BASIC program may be stopped and variables can be examined and even changed. The **CONT** statement then resumes execution.

Remarks: **CONT** cannot be used, if the program stops due to errors. Also any editing of the program inhibits continuation. Stopping and continuation can spoil the screen output or interfere with input/output operations.

Example: Using **CONT**

```
10 I=I+1:GOTO 10
RUN

BREAK IN 10
READY.
PRINT I
 947
CONT
```

COPY

Token: $FE $13

Format: **COPY source [,D drives] TO target [,D drive] [,U unit]**

Usage: Copies the contents of **source** to the **target**. It is used to copy either single files or, by using wildcard characters, multiple files.

source is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**.

target is either a quoted string, e.g. **"backup"** or a string expression in parentheses, e.g. **(FS$)**

If the disk unit has dual drives, it is possible to copy files from disk to disk. In this case, it is necessary to specify the drive# for source and target in the command. This is necessary too, if both files are stored on drive#1.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The **COPY** command is executed in the DOS of the disk drive. It can copy all regular file types (PRG, SEQ, USR, REL). The source file must exist, the target file must not exist. if source and target are on the same disk, the target filename must be different fom the source file name.

Example: Using **COPY**

```
COPY "*",D0 TO D1         :REM COPY ALL FILES
COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
COPY "*.TXT" TO D1       :REM PATTERN COPY
```

COS

Token: $BE

Format: **COS**(numeric expression)

Usage: The **COS** function returns the cosine of the argument. The argument is expected in units of **[radians]**. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of **[degrees]** can be converted to **[radians]** by multiplication with $\pi/180$.

Example: Using **COS**

```
PRINT COS(0.7)
 .764842187

X=60:PRINT COS(X * π / 180)
 .500000001
```

DATA

Token: $83

Format: **DATA** [list of constants]

Usage: Used to define constants which can be read by **READ** statements somewhere in the program. All type of constants (integer, real, strings) are allowed, but no expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.
A **RUN** command initializes the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is in the responsibility of the programmer, that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.
The **RESTORE** command may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming style to put large amount of **DATA** statements at the end of the program. Otherwise **GOTO** and **GOSUB** statements, with target lines lower than the current one, start their search for linenumber at the beginning of the program and have to skip through **DATA** lines wasting time.

Example: Using **DATA**

```
10 READ NA$, VE
20 READ N%:FOR I=2 TO N%:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;"   VERSION:";VE
40 PRINT "N-POINT GAUSS-LEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
30 STOP
80 DATA "MEGA 65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

DCLEAR

Token: $FE $15

Format: **DCLEAR [,D drive] [,U unit]**

Usage: Sends an initialise command to the specified unit and drive.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The DOS inside the disk unit will close all open files, clear all channels, free buffers and reread the BAM. This command should be used together with a **DCLOSE** to make sure, that the computer and the drive agree on the status, otherwise strange side effects may occur.

Example: Using **DCLEAR**

```
DCLOSE   :DCLEAR
DCLOSE U9:DCLEAR U9
DCLOSE U9:DCLEAR D0, U9
```

DCLOSE

Token: $FE $0F

Format: **DCLOSE [#channel] [,U unit]**

Usage: Closes a single file or all files for the specified unit.

channel = channel # assigned with the **DOPEN** statement.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

The **DCLOSE** command is used either with a channel argument or a unit number, but never both.

Remarks: It is important to close all open files before the program ends. Otherwise buffers will not be freed and even worse, open write files will be incomplete (splat files) and no more usable.

Example: Using **DCLOSE**

```
DCLOSE#2 :REM CLOSE FILE ASSIGNED TO CHANNEL 2
DCLOSE U9:REM CLOSE ALL FILES OPEN ON UNIT 9
```

DEC

Token: $D1

Format: **DEC(string expression)**

Usage: Returns the decimal value of the argument, that is written as a hex string. The argument range is "0000" to "FFFF" or 0 to 65535 respectively. The argument must have 1-4 hex digits.

Remarks: Allowed digits in uppercase/graphics mode are: 0123456789ABCDEF and in lowercase/uppercase mode: 0123456789abcdef.

Example: Using **DEC**

```
PRINT DEC("D000")
 53248
POKE DEC"600"),255
```

DEF FN

Token: $96

Format: **DEF FN name(real variable)**

Usage: Defines a single statement user function with one argument of real type returning a real value. The definition must be executed before the function can be used in expressions. The argument is a dummy variable, which will be replaced by the argument in the function usage.

Remarks: The value of the dummy variable will not be changed and the variable may be used in other context without side effects.

Example: Using **DEF FN**

```
10 PD = π / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "####";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
   0  1.00  0.00
  90  0.00  1.00
 180 -1.00  0.00
 270  0.00 -1.00
 360  1.00  0.00
```

DELETE

Token: $F7

Format: **DELETE [line range]**
DELETE filename [,D drive] [,U unit] [,R]

Usage: Used either to delete a range of lines from the BASIC program or to delete a disk file.

line range consist of the first and the last line to delete or a single line number. If the first number is omitted, the first BASIC line is assumed. The second number in the range specifier defaults to the last BASIC line.

filename is either a quoted string, for example: **"safe"** or a string expression in parentheses, for example: **(FS$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

R = Recover a previously deleted file. This will only work, if there were no write operations between deletion and recovery, which may have altered the contents of the file.

Remarks: The **DELETE filename** command works like the **SCRATCH filename** command.

Example: Using **DELETE**

```
DELETE 100      :REM DELETE LINE 100
DELETE 240-350  :REM DELETE ALL LINES FROM 240 TO 350
DELETE 500-     :REM DELETE FROM 500 TO END
DELETE -70      :REM DELETE FROM START TO 70

DELETE "DRM",U9 :REM DELETE FILE DRM ON UNIT 9
```

DIM

Token: $86

Format: **DIM name(limits) [,name(limits)]...**

Usage: Declares the shape, the bounds and the type of a BASIC array. As a declaration statement it must be executed only once and before any usage of the declared arrays. An array can have one or more dimensions. One dimensional arrays are often called vectors while two or more dimensions define a matrix. The lower bound of a dimension is always zero, while the upper bound is declared. The rules for variable names apply for array names too. There are integer arrays, real arrays and string arrays. It is legal to use the same identifier for scalar variables and array variables. The left parenthesis after the name identifies array names.

Remarks: Integer arrays consume two bytes per element, real arrays five bytes and string arrays three bytes for the string descriptor plus the length of the string.
If an array identifier is used without previous declaration, an implicit declaration of an one dimensional array with limit 10 is performed.

Example: Using **DIM**

```
10 DIM A%(8)   :REM ARRAY OF 9 ELEMENTS
20 DIM XX(2,3) :REM ARRAY OF 3x4 = 12 ELEMENTS
30 FOR I=0 TO 8:A%(I)=PEEK(256+I):NEXT
40 FOR I=0 TO 2:FOR J=0 TO 3:READ XX(I,J):NEXT J,I
50 END
60 DATA 1,-2,3,-4,5,-6,7,-8,9,-10,11,-12
```

DIRECTORY

Token: $EE

Format: **DIRECTORY [filepattern] [,R] [,D drive] [,U unit]**

Usage: Prints a listing of the specified disk and may be abbreviated to **DIR**.

The **R** (Recoverable) parameter includes files in the directory, which are flagged as deleted but are still recoverable.

filepattern is either a quoted string, e.g. **"da*"** or a string expression in parentheses, e.g. **(DI$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The command **DIRECTORY** is a synonym for **CATALOG** or **DIR** and produces the same listing. The **filepattern** can be used to restrict the listing. The wildcard characters '*' and '?' may be used. Adding a ",T=" to the pattern string, with T specifying a filetype P,S,U or R (for PRG,SEQ,USR,REL) restricts the output to that filetype.

Example: Using **DIRECTORY**

```
DIRECTORY
  0 "BLACK SMURF     " BS  2A
508 "STORY PHOBOS"         SEQ
27  "C8096"                PRG
25  "C128"                 PRG
104 BLOCKS FREE.

DIR "*,T=S"
  0 "BLACK SMURF     " BS  2A
508 "STORY PHOBOS"         SEQ
104 BLOCKS FREE.
```

DISK

Token: $FE $40

Format: **DISK command [,U unit]**

Usage: Sends a command string to the specified disk unit.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

command is a string expression.

Remarks: The command string is interpreted by the disk unit and must be compatible to the used DOS version. Read the disk drive manual for possible commands.

Example: Using **DISK**

```
DISK "I0"   :REM INITIALIZE DISK IN DRIVE 0
DISK "U0>9" :REM CHANGE UNIT# TO 9
```

DLOAD

Token: $F0

Format: **DLOAD filename [,D drive] [,U unit]**

Usage: "Disk LOAD" loads a file of type PRG into memory reserved for BASIC program source.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The load address, stored in the first two bytes of the file is ignored. The program is loaded into the BASIC memory. This enables loading of BASIC programs, that were saved on other computers with different memory configurations. After loading the program is relinked and ready to run or edit. It is possible to use DLOAD in a running program (Called overlay or chaining). Then the new loaded program replaces the current one and the execution starts automatically on the first line of the new program. Variables, arrays and strings from the current run are preserved and can be used by the new loaded program.

Example: Using **DLOAD**

```
DLOAD "APOCALYPSE"
DLOAD "MEGA TOOLS",U9
DLOAD (FN$),U(UN%)
```

DMA

Token: $FE $23

Format: **DMA command [,length, source, target, sub]**

Usage: The **DMA** ("Direct Memory Access") command is the fastest method to manipulate memory areas using the DMA controller.

command 0 = copy, 1 = mix, 2 = swap, 3 = fill

length = number of bytes

source = 24bit address of read area or fill byte

target = 24bit address of write area

sub = sub command

Remarks: The **DMA** controller has access to the whole 8 MB address range using 24 bit addresses. The block size is limited to 64K.

Example: Using **DMA**

```
DMA 3, 2000,   32,0,  2048,0 :REM FILL SCREEN WITH BLANKS
DMA 0, 2000, 2048,0, 16384,1 :REM COPY SCREEN TO $14000
```

DMODE

Token: $FE $35

Format: **DMODE jam,complement,inverse,stencil,style,thick**

Usage: "Display MODE" sets several parameter of the graphical context for drawing commands.

jam	0 - 1
complement	0 - 1
inverse	0 - 1
stencil	0 - 1
style	0 - 3
thick	1 - 8

DO

Token: $EB

Format: **DO** ... **LOOP**
DO [<**UNTIL | WHILE**> <logical expr.>]
. . . statements [**EXIT**]
LOOP [<**UNTIL | WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1 -> 100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

DOPEN

Token: $FE $0D

Format: **DOPEN# lfn, filename [,L[reclen]] [,W] [,D drive] [,U unit]**

Usage: Opens a file for reading, writing or modifying.

lfn = **l**ogical **f**ile **n**umber
1 <= lfn <= 127: line terminator is CR
128 <= lfn <= 255: line terminator is CR LF

L indicates, that the file is a relative file, which is opened for read-/write and random access. The reclength is mandatory for creating realative files. For existing relative files, the reclen is used as a safety check, if given.

W opens a file for write access. The file must not exist.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: `DOPEN#` may be used to open all file types. The sequential file type **SEQ** is default. The relative file type **REL** is chosen by using the **L** parameter. Other file types must be specified in the filename, e.g. by adding ",P" to the filename for program files or ",U" for USR files.

The usage of the "save-and-replace" character '@' at the beginning of the filename is not recommended, because many Commodore disk drives have a bug, that can cause data loss when using this feature.

Example: Using **DOPEN**

```
DOPEN#5,"DATA",U9
DOPEN#130,(DD$),U(UN%)
DOPEN#3,"USER FILE,U"
DOPEN#2,"DATA BASE",L240
OPENN#4,"MYPROG,P" : REM OPEN PRG FILE
```

DPAT

Token: $FE $36

Format: **DPAT type [,number, pattern, ...]**

Usage: "Drawing PATtern" sets pattern of the graphical context for drawing commands.

type	0 - 63
number	1 - 4
pattern	0 - 255

DSAVE

Token: $EF

Format: **DSAVE filename [,D drive] [,U unit]**

Usage: "Disk SAVE" saves a BASIC program to a file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)** The maximum length of the filename is 16 characters. If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The **DVERIFY** can be used after **DSAVE** to check, if the saved program on disk is identical to the program in memory.

Example: Using **DSAVE**

```
DSAVE "ADVENTURE"
DSAVE "ZORK-I",U9
DSAVE "DUNGEON",D1,U10
```

DVERIFY

Token: $FE $14

Format: **DVERIFY filename [,D drive] [,U unit]**

Usage: "Disk VERIFY" compares a BASIC program in memory with a disk file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: **DVERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. The command exits either with the message **OK** or with **VERIFY ERROR**.

Example: Using **DVERIFY**

```
DVERIFY "ADVENTURE"
DVERIFY "ZORK-I",U9
DVERIFY "DUNGEON",D1,U10
```

EL

Format: **EL** is a reserved system variable

Usage: **EL** has the value of the line, where the latest BASIC error occurred or the value -1 if there was no error.

This variable is typically used in a TRAP routine, where the error line is taken from **EL**.

Example: Using **EL**

```
10 TRAP 100

100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME
```

ELLIPSE

Token: $FE $30

Format: **ELLIPSE xcentre, ycentre, xradius, yradius, [,solid]**

Usage: As the name says, it draws an ellipse.

xcentre x coordinate of centre in pixels.

ycentre y coordinate of centre in pixels.

xradius x radius of the ellipse in pixels.

yradius y radius of the ellipse in pixels.

solid will fill the ellipse if not zero.

Remarks: The **ELLIPSE** command is used to draw ellipses on screens with various resolutions. It can also be used to draw circles.

Example: Using **ELLIPSE**

```
10 REM USE A 640 X 400 SCREEN
20 ELLIPSE 320,200,100,150
30 REM DRAW ELLIPSE IN THE CENTRE
```

ELSE

Token: $D5

Format: **IF expression THEN true clause ELSE false clause**

Usage: The **ELSE** keyword is part of an **IF** statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using **ELSE**

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

END

Token: $80

Format: **END**

Usage: Ends the execution of the BASIC program. The **READY.** prompt appears and the computer goes into direct mode waiting for keyboard input.

Remarks: **END** does **not** clear channels or close files. Also variable definitions are still valid after **END**. The program may be continued with the **CONT** statement. After executing the very last line of the program **END** is executed automatically.

Example: Using **END**

```
10 IF V < 0 THEN END : REM NEGATIVE NUMBERS END THE PROGRAM
20 PRINT V
```

ENVELOPE

Token: $FE $0A

Format: **ENVELOPE n, [attack,decay,sustain,release, waveform,pw]**

Usage: Used to define the parameters for the synthesis of a musical instrument.

n = envelope slot (0 -> 9)

attack = attack rate (0 -> 15)

decay = decay rate (0 -> 15)

sustain = sustain rate (0 -> 15)

release = release rate (0 -> 15)

waveform = (0:triangle, 1:sawtooth, 2:square/pulse, 3:noise, 4:ring modulation)

pw = pulse width (0 -> 4095) for waveform = pulse.

There are 10 slots for storing tunes, preset with following values:

n	A	D	S	R	WF	PW	Instrument
0	0	9	0	0	2	1536	piano
1	12	0	12	0	1		accordion
2	0	0	15	0	0		calliope
3	0	5	5	0	3		drum
4	9	4	4	0	0		flute
5	0	9	2	1	1		guitar
6	0	9	0	0	2	512	harpsichord
7	0	9	9	0	2	2048	organ
8	8	9	4	1	2	512	trumpet
9	0	9	0	0	0		xylophone

Example: Using **ENVELOPE**

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"
20 VOL 8
30 TEMPO 100
40 PLAY "C D E F G A B"
50 PLAY "U5 V1 C D E F G A B"
```

ERASE

Token: $FE $2A

Format: **ERASE filename [,D drive] [,U unit] [,R]**

Usage: Used to erase a disk file.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

R = Recover a previously erased file. This will only work, if there were no write operations between erasion and recovery, which may have altered the contents of the file.

Remarks: The **ERASE filename** command works like the **SCRATCH filename** command.

The success and the number of erased files can be examined by printing or using the system variable DS$. The second last number, which reports the track number in case of an disk error, now reports the number of successfully erased files.

Example: Using **ERASE**

```
SCRATCH "DRM",U9 :REM SCRATCH FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00
SCRATCH "OLD*"   :REM SCRATCH ALL FILES BEGINNING WTH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

ER

Format: **ER** is a reserved system variable

Usage: **ER** has the value of the latest BASIC error occurred or the value -1 if there was no error.

This variable is typically used in a TRAP routine, where the error number is taken from **ER**.

Example: Using **ER**

```
10 TRAP 100

100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER);" ERROR"
110 RESUME
```

ERR$

Token: $D3

Format: **ERR$(number)**

Usage: Used to convert an error number to an error string.

number is a BASIC error number (1 -> 41).

This function is typically used in a TRAP routine, where the error number is taken from the reserved variable **ER**.

Remarks: Arguments out of range (1 -> 41) will produce an 'ILLEGAL QUANTITY' error.

Example: Using **ERR$**

```
10 TRAP 100

100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER);" ERROR"
110 RESUME
```

EXIT

Token: $FD

Format: **EXIT**

Usage: Exits the current **DO .. LOOP** and continues execution at the first statement after the next **LOOP** statement.

Remarks: In nested loops **EXIT** exits only one loop continuing executing in the next outer loop if there is one.

Example: Using **EXIT**

```
10 DO
20 INPUT "ENTER YOUR AGE";AGE%
30 IF AGE% < 18 THEN EXIT
40 INPUT "ENTER YOUR CREDIT CARD #";CR$
50 LOOP UNTIL LEN(CR$) = 12
60 IF AGE% >= 18 THEN GOSUB 1000:REM VALIDATE CREDIT CARD
70 IF AGE% <  18 THEN PRINT "TOO YOUNG":END
```

EXP

Token: $BD

Format: **EXP**(numeric expression)

Usage: The **EXP** (EXPonential function) computes the value of the mathematical constant Euler's number **e** = **2.71828183** raised to the power of the argument.

Remarks: An argument greater than 88 produces an OVERFLOW ERROR:

Example: Using **EXP**

```
PRINT EXP(1)
 2.71828183

PRINT EXP(0)
 1

PRINT EXP(LOG(2))
 2
```

FAST

Token: $FE $25

Format: **FAST**

Usage: Sets the system speed to maximum (3.58 MHz). The system default is **FAST**. However after using **SLOW** for access to slow devices, **FAST** can be used to return to fast mode.

Example: Using **FAST**

```
10 SLOW
20 GOSUB 1000:REM DO SOME SLOW I/O
30 FAST
```

FILTER

Token: $FE $03

Format: **FILTER [freq, lp, bp, hp, res]**

Usage: Sets the parameters for soundfilter.

freq = filter cut off frequency (0 -> 2047)

lp = low pass filter (0:off, 1:on)

bp = band pass filter (0:off, 1:on)

hp = high pass filter (0:off, 1:on)

resonance = resonance (0 -> 15)

Remarks: Missing parameter keep their current value. The effective filter is the sum of of all filter settings. This enables band reject and notch effects.

Example: Using **FILTER**

```
FILTER 1023,1,0,0,10 :REM LOW PASS
FILTER 1023,0,1,0,10 :REM BAND PASS
FILTER 1023,0,0,1,10 :REM HIGH PASS
```

FIND

Token: $FE $2B

Format: **FIND "string" [,from-to]**

Usage: **FIND** is an editor command and can be used in direct mode only. It searches the line range (if specified) or the whole BASIC program else. At each occurence of the "find string" the line is listed with the string highlighted. The <NO-SCROLL> key can be used to pause the output.

Remarks: Instead of the quote (") each other character may be used as delimiter for the find string. Using the quote as delimiter finds text strings, that are not tokenized and therefore not part of a keyword.
`FIND "LOOP"` will not find the BASIC keyword `LOOP`, because the keyword is stored as token and not as text. However `FIND &LOOP&` will find it.

Example: Using **FIND**

```
FIND "XX$", 2000-2700
FIND &ER&
```

FN

Token: $A5

Format: **FN name(numeric expression)**

Usage: The **FN** functions are user defined functions, that accept a numeric expression as argument and return a real value. They must be defined with **DEF FN** before the first usage.

Example: Using **FN**

```
10 PD = π / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "####";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
   0  1.00  0.00
  90  0.00  1.00
 180 -1.00  0.00
 270  0.00 -1.00
 360  1.00  0.00
```

FOR

Token: $81

Format: **FOR index=start TO end [STEP step] ... NEXT [index]**

Usage: The **FOR** statement starts the definition of a BASIC loop with an index variable.

The **index** variable may be incremented or decremented by a constant value on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments **end** must be greater or equal than **start**, for negative increments **end** must be less or equal than **start**.

It is bad programming style to change the value of the index variable inside the loop or to jump into or out of the loop body with **GOTO**.

Example: Using **FOR**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

FOREGROUND

Token: $FE $39

Format: **FOREGROUND colour**

Usage: Sets the foreground colour (text colour) of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Example: `FOREGROUND 8` - select foreground colour yellow.

Colours: **Index and RGB values of colour palette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

FRE

Token: $B8

Format: **FRE(mode)**

Usage: Returns the number of free bytes for modes 0 and 1.

FRE(0) returns the number of free bytes in bank 0, which is used for BASIC program source.

FRE(1) returns the number of free bytes in bank 1, which is the bank for BASIC variables, arrays and strings. A usage of **FRE(1)** also triggers the "garbage collection", a process, that collects used strings at the top of the bank, thereby defragmenting string memory.

FRE(2) returns the number of expansion RAM banks, that are available RAM banks above the standard RAM banks 0 and 1, that are used by BASIC.

Example: Using **FRE**:

```
10 PM = FRE(0)
20 VM = FRE(1)
30 EM = FRE(2)
40 PRINT PM;" FREE FOR PROGRAM"
50 PRINT VM;" FREE FOR VARIABLES"
60 PRINT EM;" EXPANSION RAM BANKS"
```

GET

Token: $A1

Format: **GET string variable**

Usage: Gets the next character from the keyboard queue. If the queue is empty an empty string is assigned to the variable, otherwise a one character string is created and assigned to the string variable. This command does not wait for keyboard input, so it's useful to check for key presses in regular intervals or loops.

Remarks: It is syntactically OK to use **GET** with a numerical variable, but this is dangerous, because hitting a non numerical key will produce an error message and stop the program. The command **GETKEY** is similar, but waits until a key was hit.

Example: Using **GET**:

```
10 DO: GET A$: LOOP UNTIL A$ <> ""
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GET#

Token: $A1 '#'

Format: **GET# channel, list of string variables**

Usage: Reads as many bytes as necessary from the channel argument and assigns strings of length one to each variable in the list. This is useful to read characters or bytes from an input stream one by one.

Remarks: All values from 0 to 255 are valid, so this command can also be used to read binary data. A value of 0 generates a string of length 1 containing CHR$(0) as character value.

Example: Using **GET#**:

```
10 OPEN 2,8,0,"$0,P"          :REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP :REM CAN'T READ
20 GET#2,D$,D$                :REM DISCARD LOAD ADDRESS
25 DO                         :REM LINE LOOP
30 : GET#2,D$,D$              :REM DISCARD LINE LINK
35 : IF ST THEN EXIT          :REM END-OF-FILE
40 : GET#2,L$,H$              :REM FILE SIZE BYTES
45 : S=ASC(L$) + 256*ASC(H$)  :REM FILE SIZE
45 : LINE INPUT#2, F$         :REM FILE NAME
50 : PRINT S;F$               :REM PRINT FILE ENTRY
55 LOOP
60 CLOSE 2
```

GETKEY

Token: $A1 $F9 (GET token and KEY token)

Format: **GETKEY string variable**

Usage: Gets the next character from the keyboard queue. If the queue is empty the program waits until a key is hit. Then a one character string is created and assigned to the string variable.

Remarks: It is syntactically OK to use **GETKEY** with a numerical variable, but this is dangerous, because hitting a non numerical key will produce an error message and stop the program.

Example: Using **GETKEY**:

```
10 GETKEY A$ :REM WAIT AND GET CHARACTER
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GO64

Token: $CB $36 $34 (GO token and 64)

Format: **GO64**

Usage: Switches the computer to the C64 compatible mode. In direct mode a security prompt "ARE YOU SURE?" is printed, which must be responded with 'Y' to continue.

Example: Using **GO64**:

```
GO64
ARE YOU SURE?
```

GOSUB

Token: $8D

Format: **GOSUB line**

Usage: The **GOSUB** (GOto SUBroutine) command continues program execution at the given BASIC line number, saving the current BASIC program counter and line number on the runtime stack. This enables the resume of the execution after the **GOSUB** statement, once a **RETURN** statement in the called subroutine was executed. Calls to subroutines via **GOSUB** may be nested but the end of the subroutine code must always be a **RETURN**. Otherwise a stack overflow may occur.

Remarks: Unlike other programming languages, this BASIC version does not support arguments or local variables for subroutines.
Programs can be optimised by grouping subroutines at the beginning of the program source. The **GOSUB** calls will then have low line numbers with only few digits to decode. Also the subroutines will be found faster, because the search for subroutines starts very often at the start of the program.

Example: Using **GOSUB**:

```
 10 GOTO 100 :REM TO MAIN PROGRAM
 20 REM *** SUBROUTINE DISK STATUS CHECK ***
 30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
 40 RETURN
 50 REM *** SUBROUTINE PROMPT Y/N ***
 60 DO:INPUT "CONTINUE (Y/N)";A$
 70 LOOP UNTIL A$="Y" OR A$="N"
 80 RETURN
 90 *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOSUB 30: IF DD THEN DCLOSE#2:GOSUB 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GOTO

Token: $89 (GOTO) or $CB $A4 (GO TO)

Format: **GOTO line**
GO TO line

Usage: Continues program execution at the given BASIC line number. The **GOTO** command written as a single word executes faster than the **GO TO** command.

Remarks: The new line number will be searched by scanning the BASIC source linearly upwards. If the target line number is higher than the current one, the search starts from the current line upwards. If the target line number is lower, the search starts from the start of the program. Knowing this mechanism it is possible to optimise the runtime by grouping often used targets at the start of the program.

Example: Using **GOTO**:

```
 10 GOTO 100 :REM TO MAIN PROGRAM
 20 REM *** SUBROUTINE DISK STATUS CHECK ***
 30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
 40 RETURN
 50 REM *** SUBROUTINE PROMPT Y/N ***
 60 DO:INPUT "CONTINUE (Y/N)";A$
 70 LOOP UNTIL A$="Y" OR A$="N"
 80 RETURN
 90 *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOTO 30: IF DD THEN DCLOSE#2:GOTO 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GRAPHIC

Token: $DE

Format: **GRAPHIC CLR**

Usage: Initialises the BASIC graphic system. It clears the graphics memory and screen and sets all parameters of the graphics context to the default values.

Remarks: A second form of the **GRAPHIC** command, which serves as an interface to internal subroutines may be added later.

Example: Using **GRAPHIC**:

```
10 GRAPHIC CLR         :REM INITIALIZE
20 SCREEN DEF 1,1,1,2  :REM 640 X 400 X 2
30 SCREEN SET 1,1      :REM VIEW IT
40 SCNCLR 0            :REM CLEAR SCREEN
50 LINE 50,50,590,350  :REM DRAW LINE
```

HEADER

Token: $F1

Format: **HEADER diskname [,lid] [,D drive] [,U unit]**

Usage: Used to format or clear a diskette or disk.

diskname is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(DN$)** The maximum length of the diskname is 16 characters.

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: For new diskettes or disks, which are not already formatted it is absolutely necessary to specify the disk ID with the parameter **lid**. This switches the format command to the full format, which writes sector IDs and erases all contents. This will need some time, because every block on the disk will be written.
If the **lid** parameter is omitted, a quick format will be performed. This is only possible, if the disk is formatted already. A quick format writes a new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, the blocks are not overwritten, so contents may be recovered with the **ERASE R** command.

Example: Using **HEADER**

```
HEADER "ADVENTURE",IBS
HEADER "ZORK-I",U9
HEADER "DUNGEON",D1,U10
```

HELP

Token: $EA

Format: **HELP**

Usage: When the BASIC program stops due to an error, the **HELP** command can be used to list the erroneous line and highlight the statement, that caused the error stop.

Remarks: Displays BASIC errors. For errors in disk I/O one should print the disk status variable **DS** or the disk status string **DS$**.

Example: Using **HELP**

```
10 A=1.E20
20 B=A+A:C=EXP(A):PRINT A,B,C
RUN

?OVERFLOW ERROR IN 20
READY.
HELP

20 B=A+A:C=EXP(A):PRINT A,B,C
```

The erroneous statement is highlighted or underlined like:
20 B=A+A:C=EXP(A):PRINT A,B,C

HEX$

Token: $D2

Format: **HEX$(numeric expression)**

Usage: Returns a four character string in hexadecimal notation converted from the argument. The argument must be in the range 0 -> 65535 corresponding to the hex numbers 0000 -> FFFF.

Remarks: If real numbers are used as arguments, the fractional part will be cut off, not rounded.

Example: Using **HEX$**:

```
PRINT HEX$(10),HEX$(100),HEX$(1000.9)
000A      0064      03E8
```

HIGHLIGHT

Token: $FE $39

Format: **HIGHLIGHT colour**

Usage: Sets the colour to be used for the "highlight" text attribute. The colour index must be in the range 1 to 16. (See colour table).

Remarks: The highlight text attribute is used to mark text in listings generated by the **HELP FIND CHANGE** commands.

Example: `HIGHLIGHT 8` – select highlight colour yellow.

Colours: **Index and RGB values of colour palette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

IF

Token: $8B

Format: **IF expression THEN true clause ELSE false clause**

Usage: Starts a conditional execution statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using **IF**

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

INPUT

Token: $85

Format: **INPUT [prompt <,|;>] variable list**

Usage: Prints an optional prompt string and question mark to the screen, flashes the cursor and waits for user input from the keyboard.

prompt = string expression to be printed as prompt. It may be omitted.
If the separator between prompt and variable list is a comma, the cursor is placed directly after the prompt. If the separator is a semicolon, a question mark and a space is added to the prompt.

variable list = list of one or more variables, that receive the input.

The input will be processed after the user hits RETURN.

Remarks: The user must take care to enter the correct type of input matching variable types. Also the number of input items must match the number of variables. Entering non numeric characters for integer or real variables will produce a TYPE MISMATCH ERROR. Strings for string variables have to be put in quotes if they contain spaces or commas. Many programs, that need a safe input routine use **LINE INPUT** and use an own parser, in order to avoid program breaks by wrong user input.

Example: Using **INPUT**:

```
 10 DIM N$(100),A%(100),S$(100):
 20 DO
 30 INPUT "NAME, AGE, SEX";NA$,AG%,SE$
 40 IF NA$="" THEN 30
 50 IF NA$="END" THEN EXIT
 60 IF AG% < 18 OR AG% > 100 THEN PRINT "AGE?":GOTO 30
 70 IF SE$ <> "M" AND SE$ <> "F" THEN PRINT "SEX?":GOTO 30
 80 REM CHECK OK: ENTER INTO ARRAY
 90 N$(N)=NA$:A%(N)=AG%:S$(N)=SE$:N=N+1
100 LOOP UNTIL N=100
110 PRINT "RECEIVED";N;" NAMES"
```

INPUT#

Token: $84

Format: **INPUT# chnannel, variable list**

Usage: Reads a record from an input device, e.g. a disk file or a RS232 device and assigns the read data to the variables in the list.

channel = channel number assigned by a **DOPEN** or **OPEN** command.

variable list = list of one or more variables, that receive the input.

The input record must be terminated by a RETURN character and must be not longer than the input buffer (160 characters).

Remarks: The type and number of data in a record must match the variable list. Reading non numeric characters for integer or real variables will produce a FILE DATA ERROR. Strings for string variables have to be put in quotes if they contain spaces or commas.
The command **LINE INPUT#** may be used to read a whole record into a single string variable.

Example: Using **INPUT#**:

```
 10 DIM N$(100),A%(100),S$(100):
 20 DOPEN#2,"DATA"
 30 FOR I=0 TO 100
 40 INPUT#2,N$(I),A%(I),S$(I)
 50 IF ST=64 THEN 80:REM END OF FILE
 60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
 70 NEXT I
 80 DCLOSE#2
110 PRINT "READ";I;" RECORDS"
```

INSTR

Token: $D4

Format: **INSTR(haystack, needle [,start])**

Usage: Locates the position of the string expression ”needle” in the string expression ”haystack” and returns the index of the first occurrence or zero, if there is no match.

The string expression **haystack** is searched for the occurrence of the string expression **needle**.

The optional argument **start** is an integer expression, which defines the starting position for the search in **haystack**. If not present it defaults to one.

Remarks: If either string is empty or there is no match the function returns zero.

Example: Using **INSTR**:

```
I = INSTR("ABCDEF","CD")      : REM I = 3
I = INSTR("ABCDEF","XY")      : REM I = 0
I = INSTR("ABCDEF","E",3)     : REM I = 5
I = INSTR("ABCDEF","E",6)     : REM I = 0
I = INSTR(A$+B$,C$)
```

INT

Token: $B5

Format: **INT(numeric expression)**

Usage: Searches the greatest integer value, that is less or equal to the argument and returns this value as a real number. This function is **NOT** limited to the typical 16-bit integer range (-32768 -> 32767), because it uses real arithmetic. The allowed range is therefore determined by the size of the real mantissas
(32-bit) : (-2147483648 -> 2147483647).

Remarks: It is not necessary to use the **INT** function for assigning real values to integer variables, because this conversion will be done implicitly, but then for the 16-bit range.

Example: Using **INT**:

```
X  = INT(1.9)       :REM X = 1
X  = INT(-3.1)      :REM X = -4
X  = INT(100000.5)  :REM X = 100000
N% = INT(100000.5)  :REM ?ILLEGAL QUANTITY ERROR
```

JOY

Token: $CF

Format: **JOY(port)**

Usage: Returns the state of the joystick for the selected port (1 or 2). Bit 7 contains the state of the fire button. The stick can be moved in eight directions, which are numbered clockwise starting at the upper position.

	left	centre	right
up	8	1	2
centre	7	0	3
down	6	5	4

Example: Using **JOY**:

```
 10 N = JOY(1)
 20 IF N AND 128 THEN PRINT "FIRE! ";
 30 REM                N   NE  E   SE  S   SW  W   NW
 40 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
 50 GOTO 10
100 PRINT "GO NORTH"    :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"     :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"    :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"     :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

KEY

Token: $F9

Format: **KEY [ON | OFF | number, string]**

Usage: The function keys can either send their keycode when pressed, or a string assigned to this key. After power up or reset this feature is activated and the keys have default assignments.

KEY OFF: switch off function key strings. The keys will send their character code if pressed.

KEY ON: switch on function key strings. The keys will send assigned strings if pressed.

KEY: list current assignments.

KEY number, string assigns the string to the key with that number.

Default assignments:

key	number	string
F1	1	"GRAPHIC"
F2	2	"DLOAD"+CHR$(34)
F3	3	"DIRECTORY"+CHR$(13)
F4	4	"SCNCLR"+CHR$(13)
F5	5	"DSAVE"+CHR$(34)
F6	6	"RUN"+CHR$(13)
F7	7	"LIST"+CHR$(13)
F8	8	"MONITOR"+CHR$(13)
HELP	15	"HELP"+CHR$(13)
RUN	16	"RUN"+CHR$(34)+"*"+CHR$(34)+CHR$(13)

Remarks: The sum of the lengths of all assigned strings must not exceed 240 characters. Special characters like RETURN or QUOTE are entered using their codes with the CHR$(code) function.

Example: Using **KEY**:

```
KEY ON                  :REM ENABLE  FUNCTION KEYS
KEY OFF                 :REM DISABLE FUNCTION KEYS
KEY                     :REM LIST ASSIGNMENTS
KEY 2,"PRINT π"+CHR$(14) :REM ASSIGN PRINT PI TO F2
```

LEFT$

Token: $C8

Format: **LEFT$(string, n)**

Usage: Returns a string containing the first **n** characters from the argument **string**. If the length of **string** is equal or less than **n**, the result string will be identical to the argument string.

string = a string expression

n = a numeric expression (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using **LEFT$**:

```
PRINT LEFT$("MEGA-65",4)
MEGA
```

LEN

Token: $C3

Format: **LEN(string)**

Usage: Returns the length of the string.

string = a string expression

Remarks: Unprintable characters and even the NULL character are counted too.

Example: Using **LEN**:

```
PRINT LEN("MEGA-65"+CHR$(13))
8
```

LET

Token: $88

Format: **LET variable = expression**

Usage: The **LET** statement is obsolete and not needed. Assignment to variables can be done without using **LET**.

Example: Using **LET**:

```
LET A=5  :REM LONGER  AND SLOWER
A=5      :REM SHORTER AND FASTER
```

LINE

Token: $E5

Format: **LINE xbeg,ybeg,xend,yend**

Usage: Draws a line on the current graphics screen from the coordinate (xbeg/ybeg) to the coordinate (xend/yend). All currently defined modes and values of the graphic context are used.

Example: Using **LINE**:

```
10 GRAPHIC CLR         :REM INITIALIZE
20 SCREEN DEF 1,1,1,2  :REM 640 X 400 X 2
30 SCREEN SET 1,1      :REM VIEW IT
40 SCNCLR 0            :REM CLEAR SCREEN
50 LINE 50,50,590,350  :REM DRAW LINE
```

LIST

Token: $9B

Format: **LIST [line range]**

Usage: Used to list a range of lines from the BASIC program.

line range consist of the first and the last line to list or a single line number. If the first number is omitted, the first BASIC line is assumed. The second number in the range specifier defaults to the last BASIC line.

Remarks: The **LIST** command's output can be redirected to other devices via the **CMD** command.

Example: Using **LIST**

```
LIST 100      :REM LIST LINE 100
LIST 240-350  :REM LIST ALL LINES FROM 240 TO 350
LIST 500-     :REM LIST FROM 500 TO END
LIST -70      :REM LIST FROM START TO 70
```

LOAD

Token: $93

Format: **LOAD filename [,U unit [,flag]]**

Usage: This command is obsolete in BASIC-10, where the commands **DLOAD** and **BLOAD** are better alternatives.

The **LOAD** loads a file of type PRG into RAM bank 0, which is also used for BASIC program source.

filename is either a quoted string, e.g. **"prog"** or a string expression.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

If **flag** has a non zero value, the file is loaded to the address, which is read from the first two bytes of the file. Otherwise it is loaded to the start of BASIC memory and the load address in the file is ignored.

Remarks: This command is implemented in BASIC-10 to keep it backward compatible to BASIC-2.

Example: Using **LOAD**

```
LOAD "APOCALYPSE"
LOAD "MEGA TOOLS",9
LOAD "*",8,1
```

LOCATE

Token: $E6

Format: **LOCATE x,y**

Usage: Moves the graphical cursor to the specified position on the current graphic screen.

Remarks: The graphical cursor is not visible, it's just the starting point for follow up graphics commands. The current position can be examined with the **RDOT** function.

Example: Using **LOCATE**

```
LOCATE 8,16     :REM set cursor to X=8 and Y=16
```

LOG

Token: $BC

Format: **LOG**(numeric expression)

Usage: Computes the value of the natural logarithm of the argument. The natural logarithm uses Euler's number **e = 2.71828183** as base, not the number 10 which is typically used in log functions on a pocket calculator.

Remarks: The log function with base 10 can be computed by dividing the result by log(10).

Example: Using **LOG**

```
PRINT LOG(1)
 0

PRINT LOG(0)
 ?ILLEGAL QUANTITY ERROR

PRINT LOG(4)
 1.38629436

PRINT LOG(100) / LOG(10)
 2
```

LOOP

Token: $EC

Format: **DO** ... **LOOP**
DO [<**UNTIL | WHILE**> <logical expr.>]
. . . statements [**EXIT**]
LOOP [<**UNTIL | WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1 -> 100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

LPEN

Token: $CE $04

Format: **LPEN(coordinate)**

Usage: This function requires the use of a CRT monitor or TV and a light pen. It will not work with a LCD or LED screen. The lightpen must be connected to port 1.

LPEN(0) returns the X position of the lightpen, the range is 60 -> 320.

LPEN(1) returns the Y position of the lightpen, the range is 50 -> 250.

Remarks: The X resolution is two pixels, **LPEN(0)** returns therefore only even numbers. A bright background colour is needed to trigger the lightpen. The **COLLISION** statement may be used to install an interrupt handler.

Example: Using **LPEN**

```
PRINT LPEN(0),LPEN(1)   :REM PRINT LIGHTPEN COORDINATES
```

MIDS

Token: $CA

Format: **variable$ = MID$(string, index, n)**
MID$(string, index, n) = string expression

Usage: **MID$** can be used either as a function, which returns a string or as a statement for inserting substrings into an existing string.

string = a string expression

index = start index (0 -> 255)

n = length of substring (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using **MID$**:

```
10 A$ = "MEGA-65"
20 PRINT MID$(A$,3,4)
30 MID$(A$,5,1) = "+"
40 PRINT A$
RUN
GA-6
MEGA+65
```

MONITOR

Token: $FA

Format: **MONITOR**

Usage: Calls the machine language monitor program, which is mainly used for debugging.

Remarks: Using the **MONITOR** requires knowledge of the CSG4510 / 6502 / 6510 CPU and the assembler language.

Example: Using **MONITOR**:

```
MONITOR
```

MOUSE

Token: $FE $3E

Format: **MOUSE ON [,port [,sprite [,pos]]]**
MOUSE OFF

Usage: Enables the mouse driver and connects the mouse at the specified port with the mouse pointer sprite.

port = mouse port 1, 2 (default) or 3 (both).

sprite = sprite number for mouse pointer (default 0).

pos = initial mouse position (x,y).

The **MOUSE OFF** command disables the mouse driver and frees the associated sprite.

Remarks: The "hot spot" of the mouse pointer is the upper left pixel of the sprite.

Example: Using **MOUSE**:

```
REM LOAD DATA INTO SPRITE #0 BEFORE USING IT
MOUSE ON, 1       :REM ENABLE  MOUSE WITH SPRITE #0
MOUSE OFF         :REM DISABLE MOUSE
```

MOVSPR

Token: $FE $06

Format: **MOVSPR sprite, x, y**
MOVSPR sprite, <+|->xrel, <+|->yrel
MOVSPR sprite, angle # speed

Usage: **MOVSPR** performs, depending on the argument format, three different tasks:

The first form **MOVSPR sprite, x, y** uses no signs for the arguments and sets the absolute position of the sprite to the screen pixel coordinates **x** and **y**.

The second form **MOVSPR sprite, <+|->xrel, <+|->yrel** uses signs '+' or '-' to indicate a relative displacement to the current position.

The third form **MOVSPR sprite, angle # speed** does not set the position of sprite **n**, but defines motion parameters. The format is recognized by putting a hash sign '#' between the last two arguments.

sprite = sprite number

x = absolute screen coordinate [pixel].

y = absolute screen coordinate [pixel].

xrel = relative screen coordinate [pixel].

yrel = relative screen coordinate [pixel].

angle = direction for sprite movement [degrees]. 0 = up, 90 = right, 180 = down, 270 = left.

speed = speed of movement (0 -> 15).

Remarks: The "hot spot" is the upper left pixel of the sprite.

Example: Using **MOVSPR**:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 MOVSPR 1,50,50 :REM SET SPRITE 1 to (50,50)
30 MOVSPR 1,45#5  :REM MOVE SPRITE 1 WITH SPEED 5 TO UPPER RIGHT
```

NEW

Token: $A2

Format: **NEW**
NEW RESTORE

Usage: Resets all BASIC parameters to their default values. After **NEW** the maximum RAM is available for program and data storage.

Because **NEW** resets parameters and pointers, but does not physically overwrite the address range of a BASIC program, that was in memory before **NEW**, it is possible to recover the program. If there were no **LOAD** operations or editing after the **NEW** command, the program can be restored with the command
NEW RESTORE.

Example: Using **NEW**:

```
NEW         :REM RESET BASIC
NEW RESTORE :REM TRY TO RECOVER NEW'ED PROGRAM
```

NEXT

Token: $82

Format: **FOR index=start TO end [STEP step] ... NEXT [index]**

Usage: Terminates the definition of a BASIC loop with an index variable.

The **index** variable may be incremented or decremented by a constant value **step** on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: The **index** variable after **NEXT** is optional. If no variable is specified, the variable for the current loop is assumed.
Several consecutive **NEXT** statements may be combined by specifying the indexes in a comma separated list. For `NEXT I:NEXT J:NEXT K` the statement `NEXT I,J,K` is equivalent.

Example: Using **NEXT**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

NOT

Token: $A8

Format: **NOT** operand

Usage: Performs a bitwise logical NOT operation on a 16 bit value. Integer operands are used as they are. Real operands are converted to a signed 16 bit integer. Logical operands are converted to 16 bit integer using $FFFF, decimal -1 for TRUE and $0000, decimal 0, for FALSE.

```
NOT 0  ->  1
NOT 1  ->  0
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **NOT**

```
PRINT NOT 3
-4
PRINT NOT 64
-65
```

In most cases the **NOT** will be used in **IF** statements.

```
OK = C < 256 AND C >= 0
IF (NOT OK) THEN PRINT "NOT A BYTE VALUE"
```

OFF

Token: $FE $24

Format: keyword **OFF**

Usage: **OFF** is a secondary keyword used in combination with primary keywords like **COLOR, KEY, MOUSE**.

Remarks: The keyword **OFF** cannot be used on its own.

Example: Using **OFF**

```
COLOR OFF :REM DISABLE SCREEN COLOUR
KEY OFF   :REM DISABLE FUNCTION KEY STRINGS
MOUSE OFF :REM DISABLE MOUSE DRIVER
```

ON

Token: $91

Format: **ON expression GOSUB line list**
ON expression GOTO line list
keyword **ON**

Usage: The **ON** keyword starts either a computed **GOSUB** or **GOTO** statement. Dependent on the value of the expression, the target for the **GOSUB** or **GOTO** is chosen from the table of line addresses at the end of the statement.

As a secondary keyword, **ON** is used in combination with primary keywords like **COLOR, KEY, MOUSE**.

expression is a positive numeric value. Real values are cut to integer.

line list is a comma separated list of valid line numbers.

Remarks: Negative values for **expression** will stop the program with an error message. The **line list** specifies the targets for values of 1,2,3,...
An expression value of zero or a value, that is greater than the number of target lines will do nothing and continue program execution with the next statement.

Example: Using **ON**

```
 10 COLOR ON :REM ENABLE SCREEN COLOUR
 20 KEY   ON :REM ENABLE FUNCTION KEY STRINGS
 30 MOUSE ON :REM ENABLE MOUSE DRIVER
 40 N = JOY(1):IF N AND 128 THEN PRINT "FIRE! ";
 60 REM               N   NE  E   SE  S   SW  W   NW
 70 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
 80 GOTO 40
100 PRINT "GO NORTH"    :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"     :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"    :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"     :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

OPEN

Token: $9F

Format: **OPEN lfn, first address [,secondary address [,filename]]**

Usage: Opens an input/output channel for a device.

lfn = **l**ogical **f**ile **n**umber
1 <= lfn <= 127: line terminator is CR
128 <= lfn <= 255: line terminator is CR LF

first address = device number. For IEC devices the unit number is the primary address. Following primary address values are posible:

unit	device
0	Keyboard
1	System default
2	RS232 serial connection
3	Screen
4-7	IEC printer and plotter
8-31	IEC disk drives

The **secondary address** has some special values for IEC disk units, 0:load, 1:save, 15:command channel. The values 2 -> 14 may be used for disk files.

filename is either a quoted string, e.g. **"data"** or a string expression. The syntax is different to the **DOPEN#** command. The **filename** for **OPEN** includes all file attributes, e.g.: "0:data,s,w".

Remarks: For IEC disk units the usage of **DOPEN** is recommended.

The usage of the "save-and-replace" character '@' at the beginning of the filename is not recommended, because many Commodore disk drives have a bug, that can cause data loss when using this feature.

Example: Using **OPEN**

```
OPEN 4,4    :REM OPEN PRINTER
CMD 4       :REM REDIRECT STANDARD OUTPUT TO 4
LIST        :REM PRINT LISTING ON PRINTER DEVICE 4
OPEN 3,8,3,"0:USER FILE,U"
OPEN 2,9,2,"0:DATA,S,W"
```

OR

Token: $B0

Format: operand **OR** operand

Usage: Performs a bitwise logical OR operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16-bit integer. Logical operands are converted to 16-bit integer using $FFFF, decimal -1 for TRUE and $0000, decimal 0, for FALSE.

```
0 OR 0  ->  0
0 OR 1  ->  1
1 OR 0  ->  1
1 OR 1  ->  1
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **OR**

```
PRINT 1 OR 3
3
PRINT 128 OR 64
192
```

In most cases the **OR** will be used in **IF** statements.

```
IF (C < 0 OR C > 255) THEN PRINT "NOT A BYTE VALUE"
```

PAINT

Token: $DF

Format: **PAINT x, y, mode [,colour]**

Usage: Performs a flood fill of an enclosed graphics area.

x, y is a coordinate pair, which must lie inside the area to be filled.

mode specifies the fill mode.
0: use the **colour** to fill the area.
1: use the colour of pixel (x,y) to fill the area.

Example: Using **PAINT**

```
 10 GRAPHIC CLR          :REM INITIALIZE
 20 SCREEN DEF 1,0,0,2   :REM 320 X 200
 30 SCREEN OPEN 1        :REM OPEN
 40 SCREEN SET 1,1       :REM MAKE SCREEN ACTIVE
 50 LINE 160,0,240,100   :REM 1ST. LINE
 60 LINE 240,100,80,100  :REM 2ND. LINE
 70 LINE 80,100,160,0    :REM 3RD. LINE
 80 PAINT 160,10,0,1     :REM FILL TRIANGLE WITH COLOUR 1
 90 GETKEY K$            :REM WAIT FOR KEY
100 SCREEN CLOSE 1       :REM END GRAPHICS
```

PALETTE

Token: $FE $34

Format: **PALETTE [screen|COLOR], colour, red, green, blue**
PALETTE RESTORE

Usage: The **PALETTE** command can be used to change an entry of the system colour palette or the palette of a screen.
PALETTE RESTORE resets the system palette to the default values.

screen = screen number (0 or 1).

COLOR = keyword for changing system palette.

colour = index to palette 0 -> 255.

red = red intensity 0 -> 15.

green = green intensity 0 -> 15.

blue = blue intensity 0 -> 15.

Example: Using **PALETTE**

```
 10 GRAPHIC CLR              :REM INITIALIZE
 20 SCREEN DEF 1,0,0,2       :REM 320 X 200
 30 SCREEN OPEN 1            :REM OPEN
 40 SCREEN SET 1,1           :REM MAKE SCREEN ACTIVE
 50 PALETTE 1,0,  0, 0, 0    :REM 0 = BLACK
 60 PALETTE 1,1, 15, 0, 0    :REM 1 = RED
 70 PALETTE 1,2,  0, 0,15    :REM 2 = BLUE
 80 PALETTE 1,3,  0,15, 0    :REM 3 = GREEN
 90 LINE 160,0,240,100       :REM 1ST. LINE
100 LINE 240,100,80,100      :REM 2ND. LINE
110 LINE 80,100,160,0        :REM 3RD. LINE
120 PAINT 160,10,0,2         :REM FILL TRIANGLE WITH BLUE (2)
130 GETKEY K$                :REM WAIT FOR KEY
140 SCREEN CLOSE 1           :REM END GRAPHICS
```

PEEK

Token: $C2

Format: **PEEK(address)**

Usage: Returns a byte value read from the 16 bit address and the current memory bank (set by **BANK**).

address = a value 0 -> 65535.

Remarks: Banks 0 -> 127 give access to RAM or ROM banks. Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using **PEEK**

```
10 BANK 128              :REM SELECT SYSTEM BANK
20 L = PEEK(DEC("02F8"))  :REM USR JUMP TARGET LOW
30 H = PEEK(DEC("02F9"))  :REM USR JUMP TARGET HIGH
40 T = L + 256 * H        :REM 16 BIT JUMP ADDRESS
50 PRINT "USR FUNCTION CALLS ADDRESS";T
```

PEN

Token: $FE $33

Format: **PEN pen colour**

Usage: Sets the colour for the graphic pen.

pen = pen number (0 -> 2)

colour = palette index.

Remarks: **PEN** defined colours are used by all following drawing commands.

Example: Using **PEN**

```
 10 GRAPHIC CLR             :REM INITIALIZE
 20 SCREEN DEF 1,0,0,2      :REM 320 X 200
 30 SCREEN OPEN 1           :REM OPEN
 40 SCREEN SET 1,1          :REM MAKE SCREEN ACTIVE
 50 PALETTE 1,0,  0, 0, 0   :REM 0 = BLACK
 60 PALETTE 1,1, 15, 0, 0   :REM 1 = RED
 70 PALETTE 1,2,  0, 0,15   :REM 2 = BLUE
 80 PALETTE 1,3,  0,15, 0   :REM 3 = GREEN
 90 PEN 0,1                 :REM PEN 0 = RED
100 LINE 160,0,240,100      :REM DRAW RED LINE
110 PEN 0,2                 :REM PEN 0 = BLUE
120 LINE 240,100,80,100     :REM DRAW BLUE LINE
130 PEN 0,3                 :REM PEN 0 = GREEN
140 LINE 80,100,160,0       :REM DRAW GREEN LINE
150 GETKEY K$               :REM WAIT FOR KEY
160 SCREEN CLOSE 1          :REM END GRAPHICS
```

PLAY

Token: $FE $04

Format: **PLAY string**

Usage: Starts playing a tune with notes and directives embedded in the argument string.

A musical note is a letter (A,B,C,D,E,F,G) which may be preceded by an optional modifier.

Possible modifieres are:

char	effect
#	sharp
$	flat
.	dotted
H	half note
I	eighth note
M	wait for end
Q	quarter note
R	pause (rest)
S	sixteenth note
W	whole note

Embedded directives consist of a letter followed by a digit:

char	directive	argument range
O	octave	0 - 6
T	tune envelope	0 - 9
U	volume	0 - 9
V	voice	1 - 3
X	filter	0 - 1

The envelope slots may be changed using the **ENVELOPE** statement. The default setting for the envelopes are:

n	A	D	S	R	WF	PW	Instrument
0	0	9	0	0	2	1536	piano
1	12	0	12	0	1		accordion
2	0	0	15	0	0		calliope
3	0	5	5	0	3		drum
4	9	4	4	0	0		flute
5	0	9	2	1	1		guitar
6	0	9	0	0	2	512	harpsichord
7	0	9	9	0	2	2048	organ
8	8	9	4	1	2	512	trumpet
9	0	9	0	0	0		xylophone

Remarks: The **PLAY** statement sets up an interrupt driven routine that starts parsing the string and playing the tune. The execution continues with the next statement with no need waiting for the tune to be finished. However this can be forced, using the 'M' modifier.

Example: Using **PLAY**

```
10 ENVELOPE 9,10,5,10,5,2,4000
20 PLAY "T9"
30 VOL 8
40 TEMPO 100
50 PLAY "C D E F G A B"
60 PLAY "U5 V1 C D E F G A B"
```

POINTER

Token: $CE $0A

Format: **POINTER(variable)**

Usage: Returns the current address of a variable or an array element in bank 1. For string variables, it is the address of the string descriptor, not the string itself. The string descriptor consists of the three bytes (length,string address low, string address high).

Remarks: The address values of arrays and their elements change for every new declaration (first usage) of scalar variables.
The addresses of strings (not their descriptors) may change at any time due to "garbage collection" in memory management.

Example: Using **POINTER**

```
10 A$="TEXT": B%=5: DIM C(100) :REM DEFINE SOME VARIABLES
20 PRINT POINTER(A$);POINTER(B%);POINTER(C(20))

1026 1033 1145
```

POKE

Token: $97

Format: **POKE address, byte [,byte ...]**

Usage: Puts on or more bytes into memory or memory mapped I/O, starting at 16-bit **address**. The current memory bank as set by **BANK** is used.

address = a value 0 -> 65535.

byte = a value 0 -> 255.

Remarks: The address is increased by one for each data byte, so a memory range may be filled with a single command.

Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using **POKE**

```
10 BANK 128                :REM SELECT SYSTEM BANK
20 POKE,DEC("02F8")),0,32  :REM SET USR VECTOR TO $2000
```

POLYGON

Token: $FE $2F

Format: **POLYGON x, y, xrad, yrad, solid, angle, sides, n**

Usage: Draws a regular **n** sided polygon. The polygon is drawn using the current drawing context set with SCREEN, PALETTE and PEN.

x,y = centre coordinates.

xrad,yrad = radius in x- and y-direction.

solid = fill (1) or outline (0).

angle = start angle.

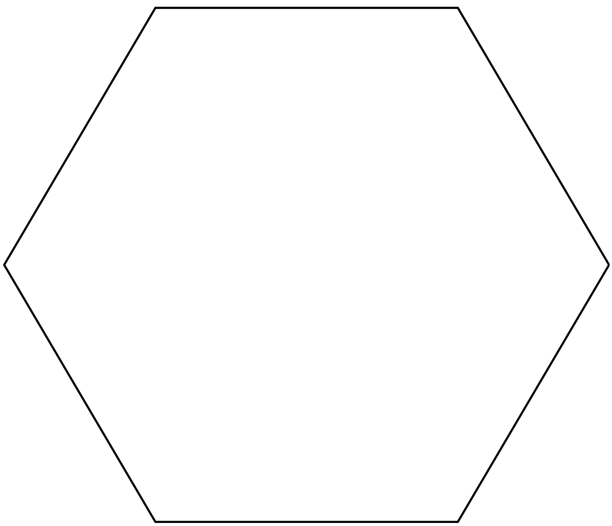
sides = sides to draw $(<= n)$.

n = number of sides or edges.

Remarks: A regular polygon is both isogonal and isotoxal, meaning all sides and angles are alike.

Example: Using **POLYGON**

```
POLYGON 320,100,50,50,0,0,6,6
```



POS

Token: $B9

Format: **POS(dummy)**

Usage: Returns the cursor column relative to the currently used window.

dummy = a numeric value, which is ignored.

Remarks: **POS** gives the column position for the screen cursor. It will not work for redirected output.

Example: Using **POS**

```
10 IF POS(0) > 72 THEN PRINT :REM INSERT RETURN
```

POT

Token: $CE $02

Format: **POT(paddle)**

Usage: Returns the position of a paddle.

paddle = paddle number 1 -> 4.

The low byte of the return value is the paddle value with 0 at the clockwise limit and 255 at the counterclockwise limit.

A value > 255 indicates the simultaneous press of the firebutton.

Remarks: Analogue paddles are noisy and inexact. The range may be less than 0 - 255 and there is some jitter in the data.

Example: Using **POT**

```
10 X = POT(1)       : REM READ PADDLE #1
20 B = X > 255      : REM TRUE (-1) IF FIRE BUTTON IS PRESSED
30 V = X AND 255    : PADDLE #1 VALUE
```

PRINT

Token: $99

Format: **PRINT arguments**

Usage: Evaluates the argument list and prints the values formatted to the current screen window. Standard formatting is used dependent on the argument type. For user controlled formatting see **PRINT USING**. Following argument types are processed:

numeric : The printout starts with a space for positive and zero values or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed either in fixed point format with typically 9 digits or in scientific format, if the value is outside the range of 0.01 -> 999999999.

string : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed.

, : A comma acts like a tabulator.

; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additinal characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions may be used in the argument list for positioning. The **CMD** command can be used for redirection.

Example: Using **PRINT**

```
10 FOR I=1 TO 10     : REM START LOOP
20 PRINT I,I*I,SQR(I)
30 NEXT
```

PRINT#

Token: $98

Format: **PRINT# channel, arguments**

Usage: Evaluates the argument list and prints the values formatted to the device assigned to **channel**. Standard formatting is used dependent on the argument type. For user controlled formatting see **PRINT# USING**. Following argument types are processed:

channel : must be opened for output by an **OPEN** or **DOPEN** statement.

numeric : The printout starts with a space for positive and zero values or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed either in fixed point format with typically 9 digits or in scientific format, if the value is outside the range of 0.01 -> 999999999.

string : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed.

, : A comma acts like a tabulator.

; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additinal characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions are not suitable for devices other than the screen.

Example: Using **PRINT#**

```
10 DOPEN#2,"TABLE",W,U9
20 FOR I=1 TO 10     : REM START LOOP
30 PRINT#2,I,I*I,SQR(I)
40 NEXT
50 DCLOSE#2
```

PRINT USING

Token: $98 $FB or $99 $FB

Format: **PRINT [# channel,] USING, format, arguments**

Usage: Parses the format string and evaluates the argument list. The values are printed following the directives of the format string.

channel : must be opened for output by an **OPEN** or **DOPEN** statement. If no channel is specified, the output goes to the screen.

format : A string which defines the rules for formatting.

numeric argument : The '#' characters set the position and with of the ouput string.
'+' or '-' set the sign option.
'.' sets the position of the decimal point.
',' can be inserted into large numbers.
'$' sets the position of the currency symbol.
^^^^ reserves place for the exponent.

string argument : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of '#' characters sets the width of the output, a '=' sign centers and a '>' character right justifies the output.

, : A comma acts like a tabulator. **;** : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additinal characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions may be used for screen output.

Example: Using **PRINT# USING**

```
10 X = 12.34: A$ = "MEGA 65"
30 PRINT USING "###.##"; X      : REM  "12.34"
40 PRINT USING "###"; X         : REM  " 12"
50 PRINT USING "####"; A$       : REM  "MEGA"
60 PRINT USING "##########"; A$ : REM  "MEGA 65   "
70 PRINT USING "=#########"; A$ : REM  " MEGA 65 "
80 PRINT USING "########>#"; A$ : REM  "   MEGA 65"
```

PUDEF

Token: $DD

Format: **PUDEF string**

Usage: Redefines up to four special characters, that are used in the **PRINT USING** routine.

string = definition string (max. 4 characters).

```
1st.: fill character
2nd.: comma separator
3rd.: decimal point
4th.: curryency symbol
```

The system default is " ,.$"

The new definition string overrides the system default and is often used for localization. A string " .," would change the punctuation to German style.

It is not necessary to redefine all four characters. Any length between 1 and 4 is allowed.

Remarks: **PUDEF** changes the output of **PRINT USING** only. **PRINT** and **PRINT#** are not affected. The control characters of the format string cannot be changed.

Example: Using **PUDEF**

```
10 X = 123456.78
20 PUDEF " .,"
30 PRINT USING "####,####.#"; X : REM 123.456,8
```

RCLR

Token: $CD

Format: **RCLR(colour source)**

Usage: Returns the current colour index for the selected colour source.

Colour sources are:

```
0:  40 column background
1:  graphical foreground
2:  multicolour mode 1
3:  multicolour mode 2
4:  frame colour
5:  80 column text
6:  80 column background
```

Example: Using **RCLR**

```
10 C = RCLR(5)  : REM C = colour index of 80 column text
```

RDOT

Token: $D0

Format: **RDOT(n)**

Usage: Returns information about the graphical cursor.

n = kind of information.

```
0:  x-position
1:  y-position
2:  colour index
```

Example: Using **RDOT**

```
10 X = RDOT(0)
20 Y = RDOT(1)
30 C = RDOT(2)
40 PRINT "THE COLOUR INDEX AT (";X;"/";Y;") IS:";C
```

READ

Token: $87

Format: **READ(variable list)**

Usage: Reads values from program source into variables.

variable list = any legal variables.

All type of constants (integer, real, strings) can be read, but no expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.
A **RUN** command initializes the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is in the responsibility of the programmer, that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.
The **RESTORE** command may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming style to put large amount of **DATA** statements at the end of the program. Otherwise **GOTO** and **GOSUB** statements, with target lines lower than the current one, start their search for linenumber at the beginning of the program and have to skip through **DATA** lines wasting time.

Example: Using **READ**

```
10 READ NA$, VE
20 READ N%:FOR I=2 TO N%:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;"   VERSION:";VE
40 PRINT "N-POINT GAUSS-LEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
30 STOP
80 DATA "MEGA 65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

RECORD

Token: $FE $12

Format: **RECORD#lfn, record, [,byte]**

Usage: Positions the read/write pointer of a relative file.

lfn = **l**ogical **f**ile **n**umber

record = target record (1 -> 65535).

byte = byte position in record.

This command can be used only for files of type **REL**, which are relative files capable of direct access.

The **RECORD** command positions the file pointer to the specified record number. If this record number does not exist and the disk capacity is high enough, the file is expanded to this record count by adding empty records. This is not an error, but the disk status will give the message **RECORD NOT PRESENT**.

Any INPUT# or PRINT# command will then proceed on the selected record position.

Remarks: The original Commodore disk drives all had a bug in their DOS, which could destroy data by using relative files. A recommended workaround was to issue each **RECORD** command twice, before and after the I/O operation.

Example: Using **RECORD**

```
10 REM *** READ FIRST 10 INDEXED RECORDS FROM DATA BASE
15 N = 1000: DIM IX(N)
20 DOPEN#3,"DATA INDX"
25 FOR I=1 TO N:INPUT#3,IX(I):NEXT
30 DCLOSE#3
35 DOPEN#2,"DATA BASE",L240
40 FOR J=1 TO 10
45 RECORD#2,IX(J)
50 INPUT#2,A$
55 PRINT A$
60 NEXT J
65 DCLOSE#2
```

REM

Token: $8F

Format: **REM**

Usage: Marks the rest of the line as comment.

All characters after **REM** are never executed but skipped.

Example: Using **REM**

```
10 REM *** PROGRAM TITLE ***
20 N=1000 :REM NUMBER OF ITEMS
30 DIM NA$(N)
```

RENAME

Token: $F5

Format: **RENAME old TO new [,D drive] [,U unit]**

Usage: Renames a disk file.

old is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**.

new is either a quoted string, e.g. **"backup"** or a string expression in parentheses, e.g. **(FS$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: The **RENAME** command is executed in the DOS of the disk drive. It can rename all regular file types (PRG, SEQ, USR, REL). The old file must exist, the new file must not exist. Only single files can be renamed, wild characters like '*' and '?' are not allowed. The file type cannot be changed.

Example: Using **RENAME**

```
RENAME "CODES" TO "BACKUP" :REM RENAME SINGLE FILE
```

RENUMBER

Token: $F8

Format: **RENUMBER [new, inc, old]**

Usage: Used to renumber all or a range of lines of a BASIC program.

new is the new starting line of the line range to renumber. The default value is 10.

inc is the increment to be used. The default value is 10.

old is the old starting line of the line range to renumber. The default value is the first line.

The **RENUMBER** changes all line numebrs in the chosen range and also changes all references from statements like **GOTO, GOSUB, TRAP, RESTORE, RUN** etc.

RENUMBER can be executed in direct mode only. If it detects a problem, like memory overflow, unresolved references or line number overflow (greater than 64000) it will stop with an error message and leave the program unchanged.

The command may be called with 0-3 parameters. Unspecified parameters use their default values.

Remarks: The **RENUMBER** command may need several minutes to execute for large programs.

Example: Using **RENUMBER**

```
RENUMBER            :REM NUMBERS WILL BE 10,20,30,...
RENUMBER 100,5      :REM NUMBERS WILL BE 100,105,110,115,...
RENUMBER 601,1,500  :REM RENUMBER STARTING AT 500 TO 601,602,...
```

RESTORE

Token: $8C

Format: **RESTORE [line]**

Usage: Set or reset the internal pointer for **READ** from **DATA** statements.

line is the new position for the pointer to point at. The default is the first program line.

Remarks: The new pointer target **line** needs not to contain **DATA** statements. Every **READ** will automatically advance the pointer to the next **DATA** statement.

Example: Using **RESTORE**

```
10 DATA 3,1,4,1,5,9,2,6
20 DATA "MEGA 65"
30 DATA 2,7,1,8,2,8,9,5
40 FOR I=1 TO 8:READ P:PRINT P:NEXT
50 RESTORE 30
60 FOR I=1 TO 8:READ P:PRINT P:NEXT
70 RESTORE 20
80 READ A$:PRINT A$
```

RESUME

Token: $D6

Format: **RESUME [line | NEXT]**

Usage: is used inside a **TRAP** routine to resume normal program execution after handling the exception.

line : program execution resumes at the given line number.

NEXT : the keyword NEXT resumes execution at the statement following the statement, that caused the error.

RESUME with no parameters tries to re-execute the statement, that caused the error. The **TRAP** routine should have examined and corrected the variables in this case.

Remarks: **RESUME** cannot be used in direct mode.

Example: Using **RESUME**

```
 10 TRAP 100
 20 FOR I=1 TO 100
 30 PRINT EXP(I)
 40 NEXT
 50 PRINT "STOPPED FOR I =";I
 60 END
100 PRINT ERR$(ER): RESUME 50
```

RETURN

Token: $8E

Format: **RETURN**

Usage: Returns control from a subroutine, which was called with **GOSUB** or an event handler, declared with **COLLISION**.

The execution continues at the statement following the **GOSUB** call.

In the case of the **COLLISION** handler, the execution continues at the statement where it left to call the handler.

Example: Using **RETURN**

```
 10 DOPEN#2,"DATA":GOSUB 100
 20 FOR I=1 TO 100
 30 INPUT#2,A$:GOSUB 100
 40 PRINT A$
 50 NEXT
 60 DCLOSE#2
 70 END
100 IF DS THEN PRINT DS$:STOP  :REM DISK ERROR
110 RETURN                     :REM OK
```

RGR

Token: $CC

Format: **RGR(dummy)**

Usage: Returns information about the graphic mode.

dummy = unused numeric expression.

In text mode **RGR** returns zero. in graphics mode **RGR** returns a non zero value.

Example: Using **RGR**

```
10 M = RGR(0)
20 IF M THEN CHAR 0,0,1,1,2,"TITLE": ELSE PRINT "TITLE"
```

RIGHT$

Token: $C9

Format: **RIGHT$(string, n)**

Usage: Returns a string containing the last **n** characters from the argument **string**. If the length of **string** is equal or less than **n**, the result string will be identical to the argument string.

string = a string expression

n = a numeric expression (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using **RIGHT$**:

```
PRINT RIGHT$("MEGA-65",2)
65
```

RMOUSE

Token: $FE $3F

Format: **RMOUSE xvar, yvar, butvar**

Usage: Reads mouse position and button status.

xvar = numerical variable receiving x-position.

yvar = numerical variable receiving y-position.

butvar = numerical variable receiving button status.
left button sets bit 7, while right button sets bit 0.

value	status
0	no button
1	right button
128	left button
129	both buttons

The command puts a -1 into all variables, if the mouse is not connected or disabled.

Remarks: Two active mice on both ports merge the results.

Example: Using **RMOUSE**:

```
10 MOUSE ON, 1, 1        :REM MOUSE ON PORT 1 WITH SPRITE 1
20 RMOUSE XP, YP, BU     :REM READ MOUSE STATUS
30 IF XP < 0 THEN PRINT "NO MOUSE ON PORT 1":STOP
40 PRINT "MOUSE:";XP;YP;BU
50 MOUSE OFF             :REM DISABLE MOUSE
```

RND

Token: $BB

Format: **RND(type)**

Usage: Returns a pseudo random number

This is called a ”pseudo” random number, because the numbers are not really random, but are derived from another number called ”seed” and generate reproducable sequences. The **type** argument determines, which seed is used.

type = 0: use system clock.

type < 0: use the value of **type** as seed.

type > 0: derive value from previous random number.

Remarks: Seeded random number sequences produce the same sequence for identical seeds.

Example: Using **RND**:

```
10 DEF FNDI(X) = INT(RND(0)*6)+1 :REM DICE FUNCTION
20 FOR I=1 TO 10                 :REM THROW 10 TIMES
30 PRINT I;FNDI(0)               :REM PRINT DICE POINTS
40 NEXT
```

RREG

Token: $FE $09

Format: **RREG areg, xreg, yreg, zreg, sreg**

Usage: Reads the values, that were in the CPU registers after a SYS call, into the specified variables.

areg = variable gets accumulator value.

xreg = variable gets X register value.

yreg = variable gets Y register value.

zreg = variable gets Z register value.

sreg = variable gets status register value.

Remarks: The register values after a SYS call are stored in system memory. This enables the command **RREG** to retrieve these values.

Example: Using **RREG**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

RSPCOLOR

Token: $CE $07

Format: **RSPCOLOR(n)**

Usage: Returns multicolour sprite colours.

n = 1 : get multicolour # 1.

n = 2 : get multicolour # 2.

Remarks: See also **SPRITE** and **SPRCOLOR**.

Example: Using **RSPCOLOR**:

```
10 SPRITE 1,1         :REM TURN SPRITE 1 ON
20 C1% = RSPCOLOR(1)  :REM READ COLOUR #1
30 C2% = RSPCOLOR(2)  :REM READ COLOUR #2
```

RSPPOS

Token: $CE $05

Format: **RSPPOS(sprite,n)**

Usage: Returns sprite's position and speed

sprite : sprite number.

n = 0 : get X position.

n = 1 : get Y position.

n = 2 : get speed.

Remarks: See also **SPRITE** and **MOVSPR**.

Example: Using **RSPPOS**:

```
10 SPRITE 1,1          :REM TURN SPRITE 1 ON
20 XP = RSPPOS(1,0)    :REM GET X OF SPRITE 1
30 YP = RSPPOS(1,1)    :REM GET Y OF SPRITE 1
30 SP = RSPPOS(1,2)    :REM GET SPEED OF SPRITE 1
```

RSPRITE

Token: $CE $06

Format: **RSPRITE(sprite,n)**

Usage: Returns sprite's parameter.

sprite : sprite number (0 -> 7)

n = 0 : turned on (0 or 1).

n = 1 : foreground colour (0 -> 15)

n = 2 : background priority (0 or 1).

n = 3 : X-expanded (0 or 1).

n = 4 : Y-expanded (0 or 1).

n = 5 : multicolour (0 or 1).

Remarks: See also **SPRITE** and **MOVSPR**.

Example: Using **RSPRITE**:

```
10 SPRITE 1,1          :REM TURN SPRITE 1 ON
20 EN = RSPRITE(1,0)   :REM SPRITE 1 ENABLED ?
30 FG = RSPRITE(1,1)   :REM SPRITE 1 FOREGROUND COLOUR INDEX
30 BP = RSPRITE(1,2)   :REM SPRITE 1 BACKGROUND PRIORITY
20 XE = RSPRITE(1,3)   :REM SPRITE 1 X EXPANDED ?
30 YE = RSPRITE(1,4)   :REM SPRITE 1 Y EXOANDED ?
30 MC = RSPRITE(1,5)   :REM SPRITE 1 MULTICOLOUR ?
```

RUN

Token: $8A

Format: **RUN [line number]**
RUN filename [,D drive] [,U unit]

Usage: Run a BASIC program.

If a filename is given, the program file is loaded into memory, otherwise the program that is currently in memory is used.

line number an existing line number of the program in memory.

filename is either a quoted string, e.g. **"prog"** or a string expression in parentheses, e.g. **(PR$)**. The filetype must be "PRG".

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

RUN resets first all internal pointers to their starting values. Therefore there are no variables, arrays and strings defined, the runtime stack is reset and the table of open files is cleared.

Remarks: In order to start or continue program execution without resetting everything, use the **GOTO** command.

Example: Using **RUN**

```
RUN "FLIGHTSIM" :LOAD AND RUN PROGRAM FLIGHTSIM
RUN 1000        :RUN PROGRAM IN MEMORY, START AT 1000
RUN             :RUN PROGRAM IN MEMORY
```

SAVE

Token: $94

Format: **SAVE filename [,unit]**

Usage: "Saves a BASIC program to a file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)** The maximum length of the filename is 16 characters, not counting the optional save and replace character '@' and the infile drive definition.. If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS. The filename may be preced by the drive number definition "0:" or "1:" which is only relevant for dual drive disk units.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: This is an obsolete command, implemented only for compatibility to older BASIC dialects. The command **DSAVE** should be used instead.

Example: Using **SAVE**

```
SAVE "ADVENTURE"
SAVE "ZORK-I",8
SAVE "1:DUNGEON",9
```

SCNCLR

Token: $E8

Format: **SCNCLR [colour]**

Usage: Clears a text window or screen.

SCNCLR (with no arguments) clears the current text window. The default window occupies the whole screen.

SCNCLR colour clears the graphic screen by filling it it with the **colour**.

Example: Using **SCNCLR**:

```
10 GRAPHIC CLR         :REM INITIALIZE
20 SCREEN DEF 1,1,1,2  :REM 640 X 400 X 2
30 SCREEN SET 1,1      :REM VIEW IT
40 SCNCLR 0            :REM CLEAR SCREEN
50 LINE 50,50,590,350  :REM DRAW LINE
```

SCRATCH

Token: $F2

Format: **SCRATCH filename [,D drive] [,U unit] [,R]**

Usage: Used to erase a disk file.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

R = Recover a previously erased file. This will only work, if there were no write operations between erasion and recovery, which may have altered the contents of the file.

Remarks: The **SCRATCH filename** command works like the **ERASE filename** command.

The success and the number of erased files can be examined by printing or using the system variable DS$. The second last number, which reports the track number in case of an disk error, now reports the number of successfully erased files.

Example: Using **SCRATCH**

```
SCRATCH "DRM",U9 :REM SCRATCH FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00
SCRATCH "OLD*"   :REM SCRATCH ALL FILES BEGINNING WTH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

SCREEN

Token: $FE $2E

Format:
SCREEN CLR
SCREEN DEF
SCREEN SET
SCREEN OPEN
SCREEN CLOSE

Usage: **SCREEN** performs one of five actions, selected by the secondary keyword after it.

SCREEN CLR colour clear graphics screen by filling it with **colour**.

SCREEN DEF screen, width, height, depth defines resolution parameters for the chosen screen.

SCREEN SET draw view sets screen numbers (0 or 1) for the drawing and the viewing screen.

SCREEN OPEN screen allocates resources and initializes the graphic context for the selected screen (0 or 1).

SCREEN CLOSE screen closes screen (0 or 1) and frees ressources.

Remarks: The **SCREEN** command cannot be used alone. It must always be used together with a secondary keyword.

Example: Using **SCREEN**:

```
10 GRAPHIC CLR          :REM INITIALIZE
20 SCREEN DEF 1,1,1,2   :REM SCREEN #1: 640 X 400 X 2
30 SCREEN OPEN          :REM OPEN
40 SCREEN SET 1,1       :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0         :REM CLEAR SCREEN
60 LINE 50,50,590,350   :REM DRAW LINE
70 SLEEP 10             :REM WAIT 10 SECONDS
80 SCREEN CLOSE
```

SET

Token: $FE $2D

Format: **SET DEF unit**
SET DISK old to new

Usage: **SET DEF unit** redefines the default unit for disk access, which is initialized to 8 by the DOS. Commands, that do not explicitely specify a unit, will use this default unit.

SET DISK old to new is used to change the unit number of a disk drive temporarily.

Remarks: These settings are valid until a reset or shutdown.

Example: Using **SET**:

```
DIR            :REM SHOW DIRECTORY OF UNIT 8
SET DEF 11     :REM UNIT 11 BECOMES DEFAULT
DIR            :REM SHOW DIRECTORY OF UNIT 11
DLOAD "*"      :REM LOAD FIRST FILE FROM UNIT 11
SET DISK 8 TO 9 :REM CHANGE UNIT# OF DISK DRIVE 8 TO 9
DIR U9         :REM SHOW DIRECTORY OF UNIT 9 (FORMER 8)
```

SGN

Token: $B4

Format: **SGN**(numeric expression)

Usage: The **SGN** function extracts the sign from the argument and returns it as a number:

-1 for a negative argument
0 for a zero
1 for a positive, non zero argument

Example: Using **SGN**

```
10 ON SGN(X)+2 GOTO 100,200,300 :REM TARGETS FOR MINUS,ZERO,PLUS
20 Z = SGN(X) * ABS(Y) : REM COMBINE SIGN OF X WITH VALUE OF Y
```

SIN

Token: $BE

Format: **SIN**(numeric expression)

Usage: The **SIN** function returns the sine of the argument. The argument is expected in units of **[radians]**. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of **[degrees]** can be converted to **[radians]** by multiplication with $\pi/180$.

Example: Using **SIN**

```
PRINT SIN(0.7)
 .644217687

X=30:PRINT SIN(X * π / 180)
 .5
```

SLEEP

Token: $FE $0B

Format: **SLEEP** seconds

Usage: The **SLEEP** commands pauses the execution for the given duration (1 - 65535).

Example: Using **SLEEP**

```
50 GOSUB 1000 :REM DISPLAY SPLASH SCREEN
60 SLEEP 10   :REM WAIT 10 SECONDS
70 GOTO  2000 :REM START PROGRAM
```

SLOW

Token: $FE $26

Format: **SLOW**

Usage: Slow down system clock to 1 MHz.

Example: Using **SLOW**

```
50 SLOW       :REM SET SPEED TO MINIMUM
60 GOSUB 100 :REM EXECUTE SUBROUTINE AT 1 MHZ
70 FAST       :REM BACK TO HIGH SPEED
```

SOUND

Token: $DA

Format: **SOUND voice, freq, dur [,dir ,min, sweep, wave, pulse]**

Usage: plays a sound effect.

voice = voice number (1 -> 6).

freq = frequency (0 -> 65535).

dur = duration (0 -> 32767) .

dir = direction (0:up, 1:down, 2:oscillate).

min = minimum frequency (0 -> 65535).

sweep = sweep range (0 -> 65535).

wave = waveform (0:triangle, 1:saw, 2:square, 3:noise).

pulse = pulse width (0 -> 5095).

For details on sound programming, read the **SOUND** chapter.

Remarks: The **SOUND** command starts playing the sound effect and immediately continues with the execution of the next BASIC statement, while the sound effect is played. This enables showing graphics or text and playing sounds simultaneously.

Example: Using **SOUND**

```
SOUND 1, 7382, 60   :REM PLAY SQUARE WAVE ON VOICE 1 FOR 1 SECOND
SOUND 2,  800, 3600 :REM PLAY SQUARE WAVE ON VOICE 2 FOR 1 MINUTE
SOUND 3, 4000, 120, 2, 2000, 400, 1
REM PLAY SWEEPING SAWTOOTH WAVE AT VOICE 3
```

SPC

Token: $A6

Format: **SPC(columns)**

Usage: The **SPC** function skips **columns**.
The effect is like printing **column** times a **cursor right character**.

Remarks: The name of this function is derived from **SPACES**, which is misleading. The function prints **cursor right characters** not **SPACES**. The contents of those character cells, that are skipped, will not be changed.

Example: Using **SPC**

```
10 FOR I=8 TO 12
20 PRINT SPC(-(I<10));I :REM TRUE = -1, FALSE = 0
30 NEXT I
RUN
 8
 9
10
11
12
```

SPRCOLOR

Token: $FE $08

Format: **SPRCOLOR [mc1] [,mc2]**

Usage: Sets multicolour sprite colours.
The **SPRITE** command, which sets the attributes of a sprite, sets only the foreground colour. For the setting of the additional two colours, of multicolour sprites, **SPRCOLOR** has to be used.

Remarks: See also **SPRITE**.

Example: Using **SPRCOLOR**:

```
10 SPRITE 1,1,2,,,,1  :REM TURN SPRITE 1 ON (FG = 2)
20 SPRCOLOR 4,5       :REM MC1 = 4, MC2 = 5
```

SPRITE

Token: $FE $07

Format: **SPRITE no [switch, colour, prio, expx, expy, mode]**

Usage: Switches a sprite on or off and sets its attributes.

no = sprite number

switch = 1:ON, 0:OFF

colour = sprite foreground colour

prio = sprite(1) or screen(0) priority

expx = 1:sprite X expansion

expy = 1:sprite Y expansion

mode = 1:multi colour sprite

Remarks: The command **SPRCOLOR** must be used to set additional colours for multi colour sprites (mode = 1)

Example: Using **SPRITE**:

```
10 COLLISION 1,70 : REM ENABLE
20 SPRITE 1,1 : MOVSPR 1,120,  0 : MOVSPR 1,0#5
30 SPRITE 2,1 : MOVSPR 2,120,100 : MOVSPR 2,180#5
40 FOR I=1 TO 50000:NEXT
50 COLLISION 1 : REM DISABLE
50 END
70 REM SPRITE <-> SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

SPRSAV

Token: $FE $16

Format: **SPRSAV source, destination**

Usage: Copies sprite data.

source = sprite number or string variable.

destination = sprite number or string variable.

Remarks: Both, source and destination can be either a sprite number or a string variable. But they must not be both a string variable. A simple string assignment can be used for such cases.

Example: Using **SPRSAV**:

```
10 BLOAD "SPRITEDATA",P1600    :REM LOAD DATA FOR SPRITE 1
20 SPRITE 1,1                  :REM TURN SPRITE 1 ON
30 SPRSAV 1,2                  :REM COPY SPRITE 1 DATA TO 2
40 SPRITE 2,1                  :REM TURN SPRITE 2 ON
50 SPRSAV 1,A$                 :REM SAVE SPRITE 1 DATA IN STRING
```

SQR

Token: $BA

Format: **SQR**(numeric expression)

Usage: The **SQR** function returns the square root of the argument.

Remarks: The argument must not be negative.

Example: Using **SQR**

```
PRINT SQR(2)
 1.41421356
```

STEP

Token: $A9

Format: **FOR index=start TO end [STEP step] ... NEXT [index]**

Usage: The **STEP** keyword is an optional part of a **FOR** loop.

The **index** variable may be incremented or decremented by a constant value on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments **end** must be greater or equal than **start**, for negative increments **end** must be less or equal than **start**.

It is bad programming style to change the value of the index variable inside the loop or to jump into or out of the loop body with **GOTO**.

Example: Using **STEP**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D
```

STOP

Token: $90

Format: **STOP**

Usage: Stops the execution of the BASIC program. A message tells the line number of the break. The **READY.** prompt appears and the computer goes into direct mode waiting for keyboard input. The program execution can be resumed with the command **CONT**.

Remarks: All variable definitions are still valid after **STOP**. They may be inspected or altered and the program may be continued with the **CONT** statement. Every editing of the program source makes continuation impossible, however.

Example: Using **STOP**

```
10 IF V < 0 THEN STOP : REM NEGATIVE NUMBERS STOP THE PROGRAM
20 PRINT SQR(V)       : REM PRINT SQUARE ROOT
```

STR$

Token: $C4

Format: **STR$(numeric expression)**

Usage: Returns a string containing the formatted value of the argument, as if it were printed to the string.

Example: Using **STR$**:

```
A$ = "THE VALUE OF PI IS " + STR$(π)
PRINT A$
THE VALUE OF PI IS 3.14159265
```

SYS

Token: $9E

Format: **SYS address [, areg, xreg, yreg, zreg, sreg]**

Usage: Calls a machine language subroutine. This can be a ROM resident kernel or BASIC subroutine or a routine in RAM, which was loaded or poked to RAM before.

The CPU registers are loaded with the arguments, if specified. Then a subroutine call **JSR address** is performed. The called routine should exit with a **RTS** instruction. Then the register contents will be saved and the execution of the BASIC program continues.

address = start address of the subroutine.

areg = variable gets accumulator value.

xreg = variable gets X register value.

yreg = variable gets Y register value.

zreg = variable gets Z register value.

sreg = variable gets status register value.

The **SYS** command uses the current bank as set with the **BANK** command.

Remarks: The register values after a **SYS** call are stored in system memory. This enables the command **RREG** to retrieve these values.

Example: Using **SYS**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

TAB

Token: $A3

Format: **TAB(column)**

Usage: Positions the cursor at **column**.
This is only done, if the target column is right of the current cursor column, otherwise nothing happens. The column count starts with 0 for the left most column.

Remarks: This function must not be confused with the **TAB** key, which advances the cursor to the next tabstop.

Example: Using **TAB**

```
10 FOR I=1 TO 5
20 READ A$
30 PRINT "* " A$ TAB(10) " *"
40 NEXT I
50 END
60 DATA ONE,TWO,THREE,FOUR,FIVE

RUN
* ONE      *
* TWO      *
* THREE    *
* FOUR     *
* FIVE     *
```

TAN

Token: $C0

Format: **TAN**(numeric expression)

Usage: Returns the tangent of the argument. The argument is expected in units of **[radians]**.

Remarks: An argument in units of **[degrees]** can be converted to **[radians]** by multiplication with $\pi/180$.

Example: Using **TAN**

```
PRINT TAN(0.7)
 .84228838

X=45:PRINT TAN(X * π / 180)
 .999999999
```

TEMPO

Token: $FE $05

Format: **TEMPO speed**

Usage: Sets the playback speed for the **PLAY** command.

speed = 1 -> 255.

The duration of a whole note is computed with $duration = 24/speed$.

Example: Using **TEMPO**

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"
20 VOL 8
30 TEMPO 24      :REM PLAY EACH NOTE FOR ONE SECOND
40 PLAY "C D E F G A B"
50 PLAY "U5 V1 C D E F G A B"
```

THEN

Token: $A7

Format: **IF expression THEN true clause ELSE false clause**

Usage: The **THEN** keyword is part of an **IF** statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using **THEN**

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

TO

Token: $A4

Format: keyword **TO**

Usage: **TO** is a secondary keyword used in combination with primary keywords like **GO, FOR, BACKUP, BSAVE, CHANGE, CONCAT, COPY, RENAME and SET DISK**

Remarks: The keyword **TO** cannot be used on its own.

Example: Using **TO**

```
10 GO TO 1000    :REM AS GOTO 1000
20 GOTO 1000     :REM SHORTER AND FASTER
30 FOR I=1 TO 10 :REM TO IS PART OF THE LOOP
40 PRINT I:NEXT  :REM LOOP END
50 COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
```

TRAP

Token: $D7

Format: **TRAP [line number]**

Usage: **TRAP** with a valid line number activates the BASIC error handler with following consequences: In case of an error the BASIC interpreter does not stop with an error message, but saves execution pointer and line number, places the error number into the system variable **ER** and jumps to the line number of the TRAP command. The trapping routine can examine **ER** and decide, whether to **STOP** or **RESUME** execution.

TRAP with no argument disables the error handler. Errors will be handled by the normal system routines.

Example: Using **TRAP**

```
 10 TRAP 100
 20 FOR I=1 TO 100
 30 PRINT EXP(I)
 40 NEXT
 50 PRINT "STOPPED FOR I =";I
 60 END
100 PRINT ERR$(ER): RESUME 50
```

TROFF

Token: $D9

Format: **TROFF**

Usage: Turns off trace mode (switched on by **TRON**).

Example: Using **TROFF**

```
 10 TRON              :REM ACTIVATE TRACE MODE
 20 FOR I=85 TO 100
 30 PRINT I;EXP(I)
 40 NEXT
 50 TROFF             :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TRON

Token: $D8

Format: **TRON**

Usage: Turns on trace mode.

Example: Using **TRON**

```
 10 TRON              :REM ACTIVATE TRACE MODE
 20 FOR I=85 TO 100
 30 PRINT I;EXP(I)
 40 NEXT
 50 TROFF             :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TYPE

Token: $FE $27

Format: **TYPE filename [,D drive] [,U unit]**

Usage: types the contents of a file containing text in PETSCII code.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN$)**

drive = drive # in dual drive disk units.
The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: This command cannot be used to type BASIC programs. Use **LIST** for programs. **TYPE** can only process SEQ or USR files containing records of PETSCII text, delimited by the **CR** = CHR$(13) character.

Example: Using **TYPE**

```
TYPE "README"
TYPE "README 1ST",U9
```

UNTIL

Token: $FC

Format: **DO** ... **LOOP**
DO [<**UNTIL | WHILE**> <logical expr.>]
. . . statements [**EXIT**]
LOOP [<**UNTIL | WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**.

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1 -> 100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

USING

Token: $FB

Format: **PRINT [# channel,] USING, format, arguments**

Usage: **USING** is a secondary keyword used after **PRINT** or **PRINT#**.

It defines the format string for the argument list. The values are printed following the directives of the format string.

channel : must be opened for output by an **OPEN** or **DOPEN** statement. If no channel is specified, the output goes to the screen.

format : A string which defines the rules for formatting.

numeric argument : The '#' characters set the position and with of the ouput string.
'+' or '-' set the sign option.
'.' sets the position of the decimal point.
',' can be inserted into large numbers.
'$' sets the position of the currency symbol.
^^^^ reserves place for the exponent.

string argument : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of '#' characters sets the width of the output, a '=' sign centers and a '>' character right justifies the output.

, : A comma acts like a tabulator. **;** : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additinal characters. A semicolon at the end of the argument list suppresses the automatic return character.

Example: Using **PRINT USING**

```
10 X = 12.34: A$ = "MEGA 65"
30 PRINT USING "###.##"; X      : REM  "12.34"
40 PRINT USING "###"; X         : REM  " 12"
50 PRINT USING "####"; A$       : REM  "MEGA"
60 PRINT USING "#########"; A$  : REM  "MEGA 65  "
70 PRINT USING "=########"; A$  : REM  " MEGA 65 "
80 PRINT USING "#######>#"; A$  : REM  "  MEGA 65"
```

USR

Token: $B7

Format: **USR(numeric expression)**

Usage: Using the function **USR(X)** in a numeric expression, puts the argument into the floating point accumulator 1 and jumps to the address $02F7 expecting the address of the machine language user routine in $02F8 - $02F9. After executing the user routine, BASIC returns the contents of the floating point accumulator 1, which should be set by the user routine..

Remarks: Banks 0 -> 127 give access to RAM or ROM banks. Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using **USR**

```
10 UX = DEC("7F00")            :REM ADDRESS OF USER ROUTINE
20 BANK 128                    :REM SELECT SYSTEM BANK
30 BLOAD "ML-PROG",P(UX)       :REM LOAD USER ROUTINE
40 POKE (DEC("2F8")),UX AND 255 :REM USR JUMP TARGET LOW
50 POKE (DEC("2F9")),UX / 256   :REM USR JUMP TARGET HIGH
60 PRINT USR(π)                :REM PRINT RESULT FOR ARGUMENT PI
```

VAL

Token: $C5

Format: **VAL**(string expression)

Usage: Converts a string to a floating point value.

This function acts like reading from a string.

Remarks: A string containing not a valid number will not produce an error but return 0 as result.

Example: Using **VAL**

```
PRINT VAL("78E2")
7800

PRINT VAL("7+5")
7

PRINT VAL("1.256")
1.256

PRINT VAL("$FFFF")
0
```

VERIFY

Token: $95

Format: **VERIFY filename [,unit [,binflag]]**

Usage: This command is obsolete in BASIC-10, where the commands **DVERIFY** and **BVERIFY** are better alternatives.

VERIFY with no **binflag** compares a BASIC program in memory with a disk file of type PRG. It does the same as **DVERIFY**, but with a different syntax.

VERIFY with **binflag** compares a binary file in memory with a disk file of type PRG. It does the same as **BVERIFY**, but with a different syntax.

filename is either a quoted string, e.g. **"prog"** or a string expression.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to `8`.

Remarks: **VERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. The command exits either with the message **OK** or with **VERIFY ERROR**.

Example: Using **VERIFY**

```
VERIFY "ADVENTURE"
VERIFY "ZORK-I",9
VERIFY "1:DUNGEON",10
```

VOL

Token: $DB

Format: **VOL volume**

Usage: Sets the volume for sound output with **SOUND** or **PLAY**.

volume = 0 (off) -> 15 (loudest).

Remarks: This volume setting affects all voices.

Example: Using **VOL**

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"
20 VOL 8
30 TEMPO 100
40 PLAY "C D E F G A B"
50 PLAY "U5 V1 C D E F G A B"
```

WAIT

Token: $92

Format: **WAIT address, andmask [, xormask]**

Usage: Pauses the BASIC program until a requested bit pattern is read from the given address.

address = the address at the current memory bank, which is read.

andmask = and mask applied.

xormask = xor mask applied.

WAIT reads the byte value from **address** and applies the masks:
result = PEEK(address) AND andmask XOR xormask

The pause ends if the result is nonzero, otherwise the reading is repeated. This may hang the computer infinitely, if the condition is never met.

Remarks: This command is typically used to examine hardware registers or system variables and wait for an event, e.g. joystick event, mouse event, keyboard press or a special raster line.

Example: Using **WAIT**

```
10 BANK 128
20 WAIT 211,1           :REM WAIT FOR SHIFT KEY BEING PRESSED
```

WHILE

Token: $ED

Format: **DO** ... **LOOP**
DO [<**UNTIL | WHILE**> <logical expr.>]
. . . statements [**EXIT**]
LOOP [<**UNTIL | WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1 -> 100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

WINDOW

Token: $FE $1A

Format: **WINDOW left, top, right, bottom [,clear]**

Usage: Sets the text screen window.

left = left column

top = top row

right = right column

bottom = bottom row

clear = clear text window flag

The row values count from 0 to 24.

The column values count from 0 to 79 or 39 depending on the screen mode.

Remarks: There can be only one window on the screen. Striking the HOME key twice or printing CHR$(19)CHR$(19) will reset the window to the default full screen.

Example: Using **WINDOW**

```
10 WINDOW 0,1,79,24      :REM SCREEN WITHOUT TOP ROW
20 WINDOW 0,0,79,24,1    :REM FULL SCREEN WINDOW CLEARED
30 WINDOW 0,12,79,24     :REM LOWER HALF OF SCREEN
40 WINDOW 20,5,59,15     :REM SMALL CENTERED WINDOW
```

APPENDIX C

Special Keyboard Controls and Sequences

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123

ASCII CODES AND CHR$

You can use the PRINT CHR$(X) statement to print a character. Below is the full table of ASCII codes you can print by index. For example, by using index 65 from the table below as: PRINT CHR$(65) you will print the letter 'A'.

You can also do the reverse with the ASC statement. For example: PRINT ASC("A") Will output 65, which matches in the ASCII code table.

CHR$	Prints	CHR$	Prints	CHR$	Prints
0		17	↓	34	"
1		18	RVS ON	35	#
2		19	CLR HOME	36	$
3		20	INST DEL	37	%
4		21		38	&
5	WHT	22		39	'
6		23		40	(
7		24		41	)
8	DISABLE SHIFT	25		42	*
9	ENABLE SHIFT	26		43	+
10		27		44	,
11		28	RED	45	-
12		29	→	46	.
13	RETURN	30	GRN	47	/
14	LOWER CASE	31	BLU	48	0
15		32	SPACE	49	1
16		33	!	50	2

CHR$	Prints	CHR$	Prints	CHR$	Prints
51	3	75	K	99	
52	4	76	L	100	
53	5	77	M	101	
54	6	78	N	102	
55	7	79	O	103	
56	8	80	P	104	
57	9	81	Q	105	
58	:	82	R	106	
59	;	83	S	107	
60	<	84	T	108	
61	=	85	U	109	
62	>	86	V	110	
63	?	87	W	111	
64		88	X	112	
65	A	89	Y	113	
66	B	90	Z	114	
67	C	91	[	115	
68	D	92		116	
69	E	93	]	117	
70	F	94	↑	118	
71	G	95	←	119	
72	H	96		120	
73	I	97		121	
74	J	98		122	

CHR$	Prints	CHR$	Prints	CHR$	Prints
123		146	RVS OFF	169	
124		147	CLR HOME	170	
125		148	INST DEL	171	
126	π	149		172	
127		150		173	
128		151		174	
129	ORG	152		175	
130		153		176	
131		154		177	
132		155		178	
133	F1	156	PUR	179	
134	F3	157	←	180	
135	F5	158	YEL	181	
136	F7	159	CYN	182	
137	F2	160	SPACE	183	
138	F4	161		184	
139	F6	162		185	
140	F8	163		186	
141	SHIFT RETURN	164		187	
142	UPPERCASE	165		188	
143		166		189	
144	BLK	167		190	
145	↑	168		191	

CONTROL CODES

Keyboard Control	Function
CTRL + 1 to 8	Choose from the first range of colours.
CTRL + T	Backspace the character immediately to the left and to shift all rightmost characters one position to the left. This is the same function as the Backspace key.
CTRL + Z	Tabs the cursor to the left.
CTRL + E	Restores the colour of the cursor back to the default white.
CTRL + Q	moves the cursor down one line at a time. This is the same function produced by the Cursor Down key.
CTRL + G	produces a bell tone.
CTRL + J	is a line feed and moves the cursor down one row. This is the same function produced by the ↓ key.
CTRL + U	backs up to the start of the previous word, or unbroken string of characters. If there are no characters between the current cursor position and the start of the line, the cursor will move to the first column of the current line.
CTRL + W	advances forward to the start of the next word, or unbroken string of characters. If there are no characters between the current cursor position and the end of the line, the cursor will move to the first column of the next line.

Keyboard Control	Function
CTRL + B	turns on underline text mode. Turn off underline mode by pressing ESC then O.
CTRL + N	changes the text case mode from uppercase to lowercase.
CTRL + M	is the carriage return. This is the same function as the RETURN key.
CTRL +]	is the same function as →.
CTRL + I	tabs forward to the right.
CTRL + X	sets or clears the current screen column as a tab position. CTRL + I or Z will jump to all positions set with X. When there are no more tab positions, the cursor will stay at the end of the line with CTRL and I, or move to the start of the line in the case of CTRL and Z.
CTRL + K	locks the uppercase/lowercase mode switch usually performed with M and SHIFT keys.
CTRL + L	enables the uppercase/lowercase mode switch that is performed with the M and SHIFT keys.
CTRL + [	is the same as pressing the ESC key.
CTRL + *	enters the Matrix Mode Debugger.

SHIFTED CODES

Keyboard Control	Function
SHIFT + INST DEL	Insert a character in the current cursor position and move all characters to the right by one position.
SHIFT + HOME	Clear home, clear the entire screen and move the cursor to the home position.

ESCAPE SEQUENCES

To perform an Escape Sequence, press and release the **ESC** key. Then press one of the following keys to perform the sequence:

Key	Sequence
X	Clears the screen and toggles between 40 and 80 column modes.
@	Clears the screen starting from the cursor to the end of the screen.
A	Enables the auto-insert mode. Any keys pressed will insert before other characters.
B	Sets the bottom-right window area of the screen at the cursor position. All typed characters and screen activity will be restricted to the area. Also see **ESC** then **T**.
C	Disables auto-insert mode, going back to overwrite mode.
D	Deletes the current line and moves other lines up one position.
E	Sets the cursor to non-flashing mode.
F	Sets the cursor to regular flashing mode.
G	Enables the bell which can be sounded using **CTRL** and **G**.
H	Disable the bell so that pressing **CTRL** and **G** will have no effect.
I	Inserts an empty line in the current cursor position and moves all subsequent lines down one position.

Key	Sequence
J	Moves the cursor to start of current line.
K	Move to end of the last non-whitespace character on the current line.
L	Enables scrolling when the cursor down key is pressed at the bottom of the screen.
M	Disables scrolling. When pressing the cursor down key at the bottom on the screen, the cursor will move to the top of the screen. The cursor is restricted at the top of the screen with the Cursor up key.
O	Cancels the quote, reverse, underline and flash modes.
P	Erases all characters from the cursor to the start of current line.
Q	Erases all characters from the cursor to the end of current line.
S	Switches the VIC-IV to colour range 16-31. These colours can be accessed with CTRL and keys 1 to 8 or M and keys 1 to 8.
T	Set top-left window area of the screen at the cursor position. All typed characters and screen activity will be restricted to the area. Also see ESC then B.
U	Switches the VIC-IV to colour range 0-15. These colours can be accessed with CTRL and keys 1 to 8 or M and keys 1 to 8.

Key	Sequence
V	Scrolls the entire screen up one line.
W	Scrolls the entire screen down one line.
X	Toggles the 40/80 column display. The screen will also clear home.
Y	Set the default tab stops (every 8 spaces) for the entire screen.
Z	Clears all the tab stops. Any tabbing with CTRL and I will move the cursor to the end of the line.
1 to 8	Choose from the second range of colours.

APPENDIX **D**

Decimal, Binary and Hexadecimal

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130

NUMBERS

Simple computer programs, such as most of the introductory BASIC programs in this book, do not require an understanding of mathematics or much knowledge about the inner workings of the computer. This is because BASIC is considered a high-level programming language. It lets us program the computer somewhat indirectly, yet still gives us control over the computer's features. Most of the time, we don't need to concern ourselves with the computer's internal architecture, which is why BASIC is user friendly and accessible.

As you acquire deeper knowledge and become more experienced, you will often want to instruct the computer to perform complex or specialized tasks that differ from the examples given in this book. Perhaps for reasons of efficiency, you may also want to exercise direct and precise control over the contents of the computer's memory. This is especially true for applications that deal with advanced graphics and sound. Such operations are closer to the hardware and are therefore considered low-level. Some simple mathematical knowledge is required to be able to use these low-level features effectively.

The collective position of the tiny switches inside the computer—whether each switch is on or off—is the state of the computer. It is natural to associate numerical concepts with this state. Numbers let us understand and manipulate the internals of the machine via logic and arithmetic operations. Numbers also let us encode the two essential and important pieces of information that lie within every computer program: *instructions* and *data*.

A program's instructions tell a computer what to do and how to do it. For example, the action of outputting a text string to the screen via the statement **PRINT** is an instruction. The action of displaying a sprite and the action of changing the screen's border color are instructions too. Behind the scenes, every instruction you give to the computer is associated with one or more numbers (which, in turn, correspond to the tiny switches inside the computer being switched on or off). Most of the time these instructions won't look like numbers to you. Instead, they might take the form of statements in BASIC.

A program's data consists of information. For example, the greeting "HELLO MEGA65!" is PETSCII character data in the form of a text string. The graphical design of a sprite might be pixel data in the form of a hero for a game. And the color data of the screen's border might represent orange. Again, behind the scenes, every piece of data you give to the computer is associated with one or more numbers. Data is sometimes given directly next to the statement to which it applies. This data is referred to as a parameter or argument (such as when changing the screen colour with a **BACKGROUND 1** statement). Data may also be given within the program via the BASIC statement **DATA** which accepts a list of comma-separated values.

All such numbers—regardless of whether they represent instructions or data—reside in the computer's memory. Although the computer's memory is highly structured, the computer does not distinguish between instructions and data, nor does it have separate areas of memory for each kind of information. Instead, both are stored in whichever memory location is considered convenient. Whether a given memory location's contents is part of the program's instructions or is part of the program's data largely depends on your viewpoint, the program being written and the needs of the programmer.

Although BASIC is a high-level language, it still provides statements that allow programmers to manipulate the computer's memory efficiently. The statement **PEEK** lets us read the information from a specified memory location: we can inspect the contents of a memory address. The statement **POKE** lets us store information inside a specified memory location: we can modify the contents of a memory address so that it is set to a given value.

NOTATIONS AND BASES

We now take a look at numbers.

Numbers are ideas about quantity and magnitude. In order to manipulate numbers and determine relationships between them, it's important for them to have a unique form. This brings us to the idea of the symbolic representation of numbers using a positional notation. In this appendix we'll restrict our discussion to whole numbers, which are also called *integers*.

The *decimal* representation of numbers is the one with which you will be most comfortable since it is the one you were taught at school. Decimal notation uses the ten Hindu-Arabic numerals 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 and is thus referred to as a base 10 numeral system. As we shall see later, in order to express large numbers in decimal, we use a positional system in which we juxtapose digits into columns to form a bigger number.

For example, 53280 is a decimal number. Each such digit (0 to 9) in a decimal number represents a multiple of some power of 10. When a BASIC statement (such as **PEEK** or **POKE**) requires an integer as a parameter, that parameter is given in the decimal form.

Although the decimal notation feels natural and comfortable for humans to use, modern computers, at their most fundamental level, use a different notation. This notation is called *binary*. It is also referred to as a base 2 numeral system because it uses only two Hindu-Arabic numerals: 0 and 1. Binary reflects the fact that each of the tiny switches inside the computer must be in exactly one of two mutually exclusive states: on or off. The number 0 is associated with off and the number 1 is associated with on. Binary is the simplest notation that captures this idea. In order to express large numbers in binary, we use a positional system in which we juxtapose digits into columns to form a bigger number and prefix it with a % sign.

For example, %1001 0110 is a binary number. Each such digit (0 or 1) in a binary number represents a multiple of some power of 2.

We'll see later how we can use special BASIC statements to manipulate the patterns of ones and zeros present in a binary number to change the state of the switches associated with it. Effectively, we can toggle individual switches on or off, as needed.

A third notation called *hexadecimal* is also often used. This is a base 16 numeral system. Because it uses more than ten digits, we need to use some letters to represent the extra digits. Hexadecimal uses the ten Hindu-Arabic digits 0 to 9 as well as the six Latin alphabetic characters as "digits" (A, B, C, D, E and F) to represent the numbers 10 to 15. This gives a total of sixteen symbols for the numbers 0 to 15. To express a large number in hexadecimal, we use a positional system in which we juxtapose digits into columns to form a bigger number and prefix it with a $ sign.

For example, $E7 is a hexadecimal number. Each such digit (0 to 9 and A to F) in a hexadecimal number represents a multiple of some power of 16.

Hexadecimal is not often used when programming in BASIC. It is more commonly used when programming in low-level languages like machine code or assembly language. It also appears in computer memory maps and its brevity makes it a useful notation, so it is described here.

Always remember that decimal, binary and hexadecimal are just different notations for numbers. A notation just changes the way the number is written (i.e., the way it looks on paper or on the screen), but its intrinsic value remains unchanged. A notation is essentially different ways of representing the same thing. The reason that we use different notations is that each notation lends itself more naturally to a different task.

When using decimal, binary and hexadecimal for extended periods you may find it handy to have a scientific pocket calculator with a programmer mode. Such calculators can convert between bases with the press of a button. They can also add, subtract, multiply and divide, and perform various bitwise logical operations. See Appendix **??** **??** as it contains a **??** table for decimal, binary, and hexadecimal for integers between 0 and 255.

The BASIC listing for this appendix is a utility program that converts individual numbers into different bases. It can also convert multiple numbers within a specified range.

Although these concepts might be new now, with some practice they'll soon seem like second nature. We'll look at ways of expressing numbers in more detail. Later, we'll also investigate the various operations that we can perform on such numbers.

Decimal

When representing integers using decimal notation, each column in the number is for a different power of 10. The rightmost position represents the number of units (because $10^0 = 1$) and each column to the left of it is 10 times larger than the column before it. The rightmost column is called the units column. Columns to the left of it are labelled tens (because $10^1 = 10$), hundreds (because $10^2 = 100$), thousands (because $10^3 = 1000$), and so on.

To give an example, the integer 53280 represents the total of 5 lots of 10000, 3 lots of 1000, 2 lots of 100, 8 lots of 10 and 0 units. This can be seen more clearly if we break the integer up into distinct parts, by column.

Since

$$53280 = 50000 + 3000 + 200 + 80 + 0$$

we can present this as a table with the sum of each column at the bottom.

TEN THOUSANDS	**THOUSANDS**	**HUNDREDS**	**TENS**	**UNITS**
$10^4 = 10000$	$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
5	0	0	0	0
	3	0	0	0
		2	0	0
			8	0
				0
5	3	2	8	0

Another way of stating this is to write the expression using multiples of powers of 10.

$$53280 = (5 \times 10^4) + (3 \times 10^3) + (2 \times 10^2) + (8 \times 10^1) + (0 \times 10^0)$$

Alternatively

$$53280 = (5 \times 10000) + (3 \times 1000) + (2 \times 100) + (8 \times 10) + (0 \times 1)$$

We now introduce some useful terminology that is associated with decimal numbers.

The rightmost digit of a decimal number is called the least significant digit, because, being the smallest multiplier of a power of 10, it contributes the least to the number's magnitude. Each digit to the left of this digit has increasing significance. The leftmost (non-zero) digit of the decimal number is called the most significant digit, because, being the largest multiplier of a power of 10, it contributes the most to the number's magnitude.

For example, in the decimal number 53280, the digit 0 is the least significant digit and the digit 5 is the most significant digit.

A decimal number a is m *orders of magnitude* greater than the decimal number b if $a = b \times (10^m)$. For example, 50000 is three orders of magnitude greater than 50, because it has three more zeros. This terminology can be useful when making comparisons between numbers or when comparing the time efficiency or space efficiency of two programs with respect to the sizes of the given inputs.

Note that unlike binary (which uses a conventional % prefix) and hexadecimal (which uses a conventional $ prefix), decimal numbers are given no special prefix. In some textbooks you might see such numbers with a subscript instead. So decimal numbers will have a subscripted 10, binary numbers will have a subscripted 2, and hexadecimal numbers will have a subscripted 16.

Another useful concept is the idea of signed and unsigned decimal integers.

A signed decimal integer can be positive or negative or zero. To represent a signed decimal integer, we prefix it with either a $+$ sign or a – sign. (By convention, zero, which is neither positive nor negative, is given the $+$ sign.)

If, on the other hand, a decimal integer is unsigned it must be either zero or positive and does not have a negative representation. This can be illustrated with the BASIC statements **PEEK** and **POKE**. When we use **PEEK** to return the value contained within a memory location, we get back an unsigned decimal number. For example, the statement **PRINT (PEEK (49152))** outputs the contents of memory location 49152 to the screen as an unsigned decimal number. Note that the memory address that we gave to **PEEK** is itself an unsigned integer. When we use **POKE** to store a value inside a memory location, both the memory address and the value to store inside it are given as unsigned integers. For example, the statement **POKE 49152, 128** stores the unsigned decimal integer 128 into the memory address given by the unsigned decimal integer 49152.

Each memory location in the MEGA65 can store a decimal integer between 0 and 255. This corresponds to the smallest and largest decimal integers that can be represented using eight binary digits (eight bits). Also, the memory addresses are decimal integers between 0 and 65535. This corresponds to the smallest and largest decimal integers that can be represented using sixteen binary digits (sixteen bits).

Note that the largest number expressible using d decimal digits is $10^d - 1$. (This number will have d nines in its representation.)

Binary

Binary notation uses powers of 2 (instead of 10 which is for decimal). The rightmost position represents the number of units (because $2^0 = 1$) and each column to the left of it is 2 times larger than the column before it. Columns to the left of the rightmost column are the twos column (because $2^1 = 2$), the fours column (because $2^2 = 4$), the eights column (because $2^3 = 8$), and so on.

As an example, the integer %1101 0011 uses exactly eight binary digits and represents the total of 1 lot of 128, 1 lot of 64, 0 lots of 32, 1 lot of 16, 0 lots of 8, 0 lots of 4, 1 lot of 2 and 1 unit.

We can break this integer up into distinct parts, by column.

Since

%1101 0011 = %1000 0000 + %100 0000 + %00 0000 + %1 0000 + %0000 + %000 + %10 + %1

we can present this as a table with the sum of each column at the bottom.

ONE HUNDRED AND TWENTY-EIGHTS	**SIXTY-FOURS**	**THIRTY-TWOS**	**SIXTEENS**	**EIGHTS**	**FOURS**	**TWOS**	**UNITS**
$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	0	0	0	0	0	0	0
	1	0	0	0	0	0	0
		0	0	0	0	0	0
			1	0	0	0	0
				0	0	0	0
					0	0	0
						1	0
							1
1	1	0	1	0	0	1	1

Another way of stating this is to write the expression in decimal, using multiples of powers of 2.

$\%11010011 = (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

Alternatively

$\%11010011 = (1 \times 128) + (1 \times 64) + (0 \times 32) + (1 \times 16) + (0 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$

which is the same as writing

$$\%11010011 = 128 + 64 + 16 + 2 + 1$$

Binary has terminology of its own. Each binary digit in a binary number is called a *bit*. In an 8-bit number the bits are numbered consecutively with the least significant (i.e., rightmost) bit as bit 0 and the most significant (i.e., leftmost) bit as bit 7. In a 16-bit number the most significant bit is bit 15. A bit is said to be *set* if it equals 1. A bit is said to be *clear* if it equals 0. When a particular bit has a special meaning attached to it, we sometimes refer to it as a *flag*.

1	1	0	1	0	0	1	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

As mentioned earlier, each memory location can store an integer between 0 and 255. The minimum corresponds to %0000 0000 and the maximum corresponds to %1111 1111, which are the smallest and largest numbers that can be represented using exactly eight bits. The memory addresses use 16 bits. The smallest memory address, represented in exactly sixteen bits, is %0000 0000 0000 0000 and this corresponds to the smallest 16-bit number. Likewise, the largest memory address, represented in exactly sixteen bits, is %1111 1111 1111 1111 and this corresponds to the largest 16-bit number.

It is often convenient to refer to groups of bits by different names. For example, eight bits make a *byte* and 1024 bytes make a *kilobyte*. Half a byte is called a nybble. See Appendix **??** **??** for the **??** table for further information.

Note that the largest number expressible using d binary digits is (in decimal) $2^d - 1$. (This number will have d ones in its representation.)

Hexadecimal

Hexadecimal notation uses powers of 16. Each of the sixteen hexadecimal numerals has an asscociated value in decimal.

Hexadecimal Numeral	Decimal Equivalent
$0	0
$1	1
$2	2
$3	3
$4	4
$5	5
$6	6
$7	7
$8	8
$9	9
$A	10
$B	11
$C	12
$D	13
$E	14
$F	15

The rightmost position in a hexadecimal number represents the number of ones (since $16^0 = 1$). Each column to the left of this digit is 16 times larger than the column before it. Columns to the left of the rightmost column are the 16-column (since $16^1 = 16$), the 256-column (since $16^2 = 256$), the 4096-column (since $16^3 = 4096$), and so on.

As an example, the integer $A3F2 uses exactly four hexadecimal digits and represents the total of 10 lots of 4096 (because $A = 10), 3 lots of 256 (because $3 = 3), 15 lots of 16 (because $F = 15) and 2 units (because $2 = 2). We can break this integer up into distinct parts, by column.

Since

$A3F2 = $A000 + $300 + $F0 + $2

we can present this as a table with the sum of each column at the bottom.

FOUR THOUSAND AND NINETY-SIXES	TWO HUNDRED AND FIFTY-SIXES	SIXTEENS	UNITS
$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
A	0	0	0
	3	0	0
		F	0
			2
A	3	F	2

Another way of stating this is to write the expression in decimal, using multiples of powers of 16.

$$\$A3F2 = (10 \times 16^3) + (3 \times 16^2) + (15 \times 16^1) + (2 \times 16^0)$$

Alternatively

$$\$A3F2 = (10 \times 4096) + (3 \times 256) + (15 \times 16) + (2 \times 1)$$

which is the same as writing

$$\$A3F2 = 40960 + 768 + 240 + 2$$

Again, like binary and decimal, the rightmost digit is the least significant and the leftmost digit is the most significant.

Each memory location can store an integer between 0 and 255, and this corresponds to the hexadecimal numbers $00 and $FF. The hexadecimal number $FFFF corresponds to 65535—the largest 16-bit number.

Hexadecimal notation is often more convenient to use and manipulate than binary. Binary numbers consist of a longer sequence of ones and zeros, while hexadecimal is much shorter and more compact. This is because one hexadecimal digit is equal to exactly four bits. So a two-digit hexadecimal number comprises of eight bits with the low nybble equaling the right digit and the high nybble equaling the left digit.

Note that the largest number expressible using d hexadecimal digits is (in decimal) $16^d - 1$. (This number will have d $F symbols in its representation.)

OPERATIONS

In this section we'll take a tour of some familiar operations like counting and arithmetic, and we'll see how they apply to numbers written in binary and hexadecimal.

Then we'll take a look at various logical operations using logic gates. These operations are easy to understand. They're also very important when it comes to writing programs that have extensive numeric, graphic or sound capabilities.

Counting

If we consider carefully the process of *counting* in decimal, this will help us to understand how counting works when using binary and hexadecimal.

Let's suppose that we're counting in decimal and that we're starting at 0. Recall that the list of numerals for decimal is (in order) 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Notice that when we add 1 to 0 we obtain 1, and when we add 1 to 1 we obtain 2. We can continue in this manner, always adding 1:

0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5
5 + 1 = 6
6 + 1 = 7
7 + 1 = 8
8 + 1 = 9

Since 9 is the highest numeral in our list of numerals for decimal, we need some way of handling the following special addition: $9 + 1$. The answer is that we can reuse our old numerals all over again. In this important step, we reset the units column back to 0 and (at the same time) add 1 to the tens column. Since the tens columnn contained a 0, this gives us $9 + 1 = 10$. We say we “carried” the 1 over to the tens column while the units column cycled back to 0.

Using this technique, we can count as high as we like. The principle of counting for binary and hexadecimal is very much same, except instead of using ten symbols, we get to use two symbols and sixteen symbols, respectively.

Let’s take a look at counting in binary. Recall that the list of numerals for binary is (in order) just 0 and 1. So, if we begin counting at %0 and then add %1, we obtain %1 as the result:

%0 + %1 = %1

Now, the sum $\%1 + \%1$ will cause us to perform the analogous step: we reset the units column back to zero and (at the same time) add %1 to the twos column. Since the twos column contained a %0, this gives us $\%1 + \%1 = \%10$. We say we “carried” the %1 over to the twos column while the units column cycled back to %0. If we continue in this manner we can count higher.

%1 + %1 = %10
%10 + %1 = %11
%11 + %1 = %100
%100 + %1 = %101
%101 + %1 = %110
%110 + %1 = %111
%111 + %1 = %1000

Now we'll look at counting in hexadecimal. The list of numerals for hexadecimal is (in order) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. If we begin counting at $0 and repeatedly add $1 we obtain:

$0 + $1 = $1
$1 + $1 = $2
$2 + $1 = $3
$3 + $1 = $4
$4 + $1 = $5
$5 + $1 = $6
$6 + $1 = $7
$7 + $1 = $8
$8 + $1 = $9
$9 + $1 = $A
$A + $1 = $B
$B + $1 = $C
$C + $1 = $D
$D + $1 = $E
$E + $1 = $F

Now, when we compute $F + $1 we must reset the units column back to $0 and add $1 to the sixteens column as that number is "carried".

$F + $1 = $10

Again, this process allows us to count as high as we like.

Arithmetic

The standard arithmetic operations of addition, subtraction, multiplication and division are all possible using binary and hexadecimal.

Addition is done in the same way that addition is done using decimal, except that we use base 2 or base 16 as appropriate. Consider the following example for the addition of two binary numbers.

```
   %110
 + %111
 ------
  %1101
 ======
```

We obtain the result by first adding the units columns of both numbers. This gives us %0 + %1 = %1 with nothing to carry into the next column. Then we add the twos columns of both numbers: %1 + %1 = %0 with a %1 to carry into the next column. We then add the fours columns (plus the carry) giving (%1 + %1) + %1 = %1 with a %1 to carry into the next column. Last of all are the eights columns. Because these are effectively both zero we only concern ourselves with the carry which is %1. So (%0 + %0) + %1 = %1. Thus, %1101 is the sum.

Next is an example for the addition of two hexadecimal numbers.

```
  $7D
 +$69
 ----
  $E6
 ====
```

We begin by adding the units columns of both numbers. This gives us $D + $9 = $6 with a $1 to carry into the next column. We then add the sixteens columns (plus the carry) giving ($7 + $6) + $1 = $E with nothing to carry and so $E6 is the sum.

We now look at subtraction. As you might suspect, binary and hexadecimal subtraction follows a similar process to that of subtraction for decimal integers.

Consider the following subtraction of two binary numbers.

```
  %1011
 -%  110
 -------
  %  101
 =======
```

Starting in the units columns we perform the subtraction %1 - %0 = %1. Next, in the twos columns we perform another subtraction %1 - %1 = %0. Last of all we subtract the fours columns. This time, because %0 is less than %1, we'll need to borrow a %1 from the eights column of the top number to make the subtraction. Thus we compute %10 - %1 = %1 and deduct %1 from the eights column. The eights columns are now both zeros. Since %0 - %0 = %0 and because this is the leading digit of the result we can drop it from the final answer. This gives %101 as the result.

Let's now look at the subtraction of two hexadecimal numbers.

```
  $3D
 -$1F
 ----
  $1E
 ====
```

To perform this subtraction we compute the difference of the units columns. In order to do this, we note that because $D is less than $F we will need to borrow $1 from the sixteens column of the top number to make the subtraction. Thus, we compute $1D - $F = $E and also compute $3 - $1 = $2 in the sixteens column for the for the $1 that we just borrowed. Next, we compute the difference of the sixteens column as $2 - $1 = $1. This gives us a final answer of $1E.

We won't give in depth examples of multiplication and division for binary and hexadecimal notation. Suffice to say that principles parallel those for the decimal system. Multiplication is repeated addition and division is repeated subtraction.

We will, however, point out a special type of multiplication and division for both binary and hexadecimal. This is particularly useful for manipulating binary and hexadecimal numbers.

For binary, multiplication by two is simple—just shift all bits to the left by one position and fill in the least significant bit with a %0. Division by two is simple too—just shift all bits to the right by one position and fill in the most significant bit with a %0. By doing these repeatedly we can multiply and divide by powers of two with ease.

Thus the binary number %111, when multiplied by eight has three extra zeros on the end of it and is equal to %111000. (Recall that $2^3 = 8$.) And the binary number %10100, when divided by four has two less digits and equals %101. (Recall that $2^2 = 4$.)

These are called left and right *bit shifts*. So if we say that we shift a number to the left four bit positions, we really mean that we multiplied it by $2^4 = 16$.

For hexadecimal, the situation is similar. Multiplication by sixteen is simple—just shift all digits to the left by one position and fill in the rightmost digit with a $0. Division by sixteen is simple too—just shift all digits to the right by one position. By doing this repeatedly we can multiply and divide by powers of sixteen with ease.

Thus the hexadecimal number $F, when multiplied 256 has two extra zeros on the end of it and is equal to $F00. (Recall that $16^2 = 256$.) And the hexadecimal number $EA0, when divided by sixteen has one less digit and equals $EA. (Recall that $16^1 = 16$.)

Logic Gates

There exist several so-called *logic gates*. The fundamental ones are NOT, AND, OR and XOR.

They let us set, clear and invert specific binary digits. For example, when dealing with sprites, we might want to clear bit 6 (i.e., make it equal to 0) and set bit 1 (i.e., make it equal to 1) at the same time for a particular graphics chip register. Certain logic gates will, when used in combination, let us do this.

Learning how these logic gates work is very important because they are the key to understanding how and why the computer executes programs as it does.

All logic gates accept one or more inputs and produce a single output. These inputs and outputs are always single binary digits (i.e., they are 1-bit numbers).

The NOT gate is the only gate that accepts exactly one bit as input. All other gates—AND, OR, and XOR—accept exactly two bits as input. All gates produce exactly one output, and that output is a single bit.

First, let's take a look at the simplest gate, the NOT gate.

The NOT gate behaves by inverting the input bit and returning this resulting bit as its output. This is summarized in the following table.

INPUT X	OUTPUT
0	1
1	0

We write NOT x where x is the input bit.

Next, we take a look at the AND gate.

As mentioned earlier, the AND gate accepts two bits as input and produces a single bit as output. The AND gate behaves in the following manner. Whenever both input bits are equal to 1 the result of the output bit is 1. For all other inputs the result of the output bit is 0. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	0
1	0	0
1	1	1

We write x AND y where x and y are the input bits.

Next, we take a look at the OR gate.

The OR gate accepts two bits as input and produces a single bit as output. The OR gate behaves in the following manner. Whenever both input bits are equal to 0 the result is 0. For all other inputs the result of the output bit is 1. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	1

We write x OR y where x and y are the input bits.

Last of all we look at the XOR gate.

The XOR gate accepts two bits as input and produces a single bit as output. The XOR gate behaves in the following manner. Whenever both input bits are equal in value the output bit is 0. Otherwise, both input bits are unequal in value and the output bit is 1. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

We write x XOR y where x and y are the input bits.

Note that there do exist some other gates. They are easy to construct.

- NAND gate: this is an AND gate followed by a NOT gate
- NOR gate: this is an OR gate followed by a NOT gate
- XNOR gate: this is an XOR gate followed by a NOT gate

SIGNED AND UNSIGNED NUMBERS

So far we've largely focussed on unsigned integers. Unsigned integer have no positive or negative sign. They are always assumed to be positive. (For this purpose, zero is regarded as positive.)

Signed numbers, as mentioned earlier, can have a positive sign or a negative sign.

Signed numbers are represented by treating the most significant bit as a sign bit. This bit cannot be used for anything else. If the most significant bit is 0 then the result is interpreted as having a positive sign. Otherwise, the most significant bit is 1, and the result is interpreted as having a negative sign.

A signed 8-bit number can represent positive-sign numbers between 0 and 127, and negative-sign numbers between -1 and -128.

A signed 16-bit number can represent positive-sign numbers between 0 and 32767, and negative-sign numbers between -1 and -32768.

Reserving the most significant bit as the sign of the signed number effectively halves the range of the available positive numbers (i.e., compared to unsigned numbers), with the tradeoff being that we gain an equal quantity of negative numbers instead.

To negate any signed number, every bit in the signed number must be inverted and then %1 must added to the result. Thus, negating %0000 0101 (which is the signed number +5) gives %1111 1011 (which is the signed number -5). As expected, performing the negation of this negative number gives us +5 again.

BITWISE LOGICAL OPERATORS

The BASIC statements **NOT**, **AND**, **OR** and **XOR** have functionality similar to that of the logic gates that they are named after.

The **NOT** statement must be given a 16-bit signed decimal integer as a parameter. It returns a 16-bit signed decimal integer as a result.

In the following example, all sixteen bits of the signed decimal number +0 are equal to 0. The **NOT** statement inverts all sixteen bits as per the NOT gate. This sets all sixteen bits. If we interpret the result as a signed decimal number, we obtain the answer of -1.

```
PRINT (NOT 0)
-1
```

As expected, repeating the **NOT** statement on the parameter of -1 gets us back to where we started, since all sixteen set bits become cleared.

```
PRINT (NOT -1)
 0
```

The **AND** statement must be given two 16-bit signed decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit signed decimal integer as a result, having changed each bit as per the AND gate.

In the following example, the number +253 is used as the first parameter. As a 16-bit signed decimal integer, this is equivalent to the following number in binary: %0000 0000 1111 1101. The **AND** statement uses a bit mask as the second parameter with a 16-bit signed decimal value of +239. In binary this is the number %0000 0000 1110 1110. If we use the AND logic gate table on corresponding pairs of bits, we obtain the 16-bit signed decimal integer +237 (which is %0000 0000 1110 1100 in binary).

```
PRINT (253 AND 239)
 237
```

We can see this process more clearly in the following table.

	%0000 0000 1111 1101
AND	%0000 0000 1110 1110
	%0000 0000 1110 1100

Notice that each bit in the top row passes through unchanged wherever there is a 1 in the mask bit below it. Otherwise the bit in that position gets cleared.

The **OR** statement must be given two 16-bit signed decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit signed decimal integer as a result, having changed each bit as per the OR gate.

In the following example, the number +240 is used as the first parameter. As a 16-bit signed decimal integer, this is equivalent to the following number in binary: %0000 0000 1111 0000. The **OR** statement uses a bit mask as the second parameter with a 16-bit signed decimal value of +19. In binary this is the number %0000 0000 0001 0011. If we use the OR logic gate table on corresponding pairs of bits, we obtain the 16-bit signed decimal integer +243 (which is %0000 0000 1111 0011 in binary).

```
PRINT (240 OR 19)
  243
```

We can see this process more clearly in the following table.

	%0000 0000 1111 0000
OR	%0000 0000 0001 0011
	%0000 0000 1111 0011

Notice that each bit in the top row passes through unchanged wherever there is a 0 in the mask bit below it. Otherwise the bit in that position gets set.

Next we look at the **XOR** statement. This statement must be given two 16-bit unsigned decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit unsigned decimal integer as a result, having changed each bit as per the XOR gate.

In the following example, the number 14091 is used as the first parameter. As a 16-bit unsigned decimal integer, this is equivalent to the following number in binary: %0011 0111 0000 1011. The **XOR** statement uses a bit mask as the second paramter with a 16-bit unsigned decimal value of 8653. In binary this is the number %0010 0001 1100 1101. If we use the XOR logic gate table on corresponding pairs of bits, we obtain the 16-bit unsigned decimal integer 5830 (which is %0001 0110 1100 0110 in binary).

```
PRINT (XOR(14091,8653))
5830
```

We can see this process more clearly in the following table.

```
        %0011 0111 0000 1011
    XOR %0010 0001 1100 1101
    ------------------------
        %0001 0110 1100 0110
    ========================
```

Notice that when the bits are equal the resulting bit is 0. Otherwise the resulting bit is 1.

Much of the utility of these bitwise logical operators comes through combining them together into a compound statement. For example, the VIC II register to enable sprites is memory address 53269. There are eight sprites (numbered 0 to 7) with each bit corresponding to a sprite's status. Now suppose we want to turn off sprite 5 and turn on sprite 1, while leaving the statuses of the other sprites unchanged. We can do this with the following BASIC statement which combines an **AND** statement with an **OR** statement.

```
POKE 53269, (((PEEK(53269)) AND 223) OR 2)
```

The technique of using **PEEK** on a memory address and combining the result with bitwise logical operators, followed by a **POKE** to that same memory address is very common.

CONVERTING NUMBERS

The program below is written in BASIC. It does number conversion for you. Type it in and save it under the name "CONVERT.BAS".

To execute the program, type **RUN** and press the **RETURN** key.

The program presents you with a series of text menus. You may choose to convert a single decimal, binary or hexadecimal number. Alternatively, you may choose to convert a range of such numbers.

The program can convert numbers in the range 0 to 65535.

```
10 REM ****************************
20 REM *                          *
30 REM *  INTEGER BASE CONVERTER  *
40 REM *                          *
50 REM ****************************
60 POKE 0,65: BORDER 6: BACKGROUND 6: FOREGROUND 1
70 DIM P(15)
80 E$ = "STARTING INTEGER MUST BE LESS THAN OR EQUAL TO ENDING INTEGER"
90 FOR N = 0 TO 15
100 :  P(N) = 2 ↑ N
```

```
110 NEXT N
120 REM *** OUTPUT MAIN MENU ***
130 PRINT CHR$(147)
140 PRINT: PRINT "INTEGER BASE CONVERTER"
150 L = 22: GOSUB 1930: PRINT L$
160 PRINT: PRINT "SELECT AN OPTION (S, M OR Q):": PRINT
170 PRINT "[S]{SPACE*2}SINGLE INTEGER CONVERSION"
180 PRINT "[M]{SPACE*2}MULTIPLE INTEGER CONVERSION"
190 PRINT "[Q]{SPACE*2}QUIT PROGRAM"
200 GET M$
210 IF (M$="S") THEN GOSUB 260: GOTO 140
220 IF (M$="M") THEN GOSUB 380: GOTO 140
230 IF (M$="Q") THEN END
240 GOTO 200
250 REM *** OUTPUT SINGLE CONVERSION MENU ***
260 PRINT: PRINT "{SPACE*2}SELECT AN OPTION (D, B, H OR R):": PRINT
270 PRINT "{SPACE*2}[D]{SPACE*2}CONVERT A DECIMAL INTEGER"
280 PRINT "{SPACE*2}[B]{SPACE*2}CONVERT A BINARY INTEGER"
290 PRINT "{SPACE*2}[H]{SPACE*2}CONVERT A HEXADECIMAL INTEGER"
300 PRINT "{SPACE*2}[R]{SPACE*2}RETURN TO TOP MENU"
310 GET M1$
320 IF (M1$="D") THEN GOSUB 500: GOTO 260
330 IF (M1$="B") THEN GOSUB 760: GOTO 260
340 IF (M1$="H") THEN GOSUB 810: GOTO 260
350 IF (M1$="R") THEN RETURN
360 GOTO 310
370 REM *** OUTPUT MULTIPLE CONVERSION MENU ***
380 PRINT: PRINT "{SPACE*2}SELECT AN OPTION (D, B, H OR R):": PRINT
390 PRINT "{SPACE*2}[D]{SPACE*2}CONVERT A RANGE OF DECIMAL INTEGERS"
400 PRINT "{SPACE*2}[B]{SPACE*2}CONVERT A RANGE OF BINARY INTEGERS"
410 PRINT "{SPACE*2}[H]{SPACE*2}CONVERT A RANGE OF HEXADECIMAL INTEGERS"
420 PRINT "{SPACE*2}[R]{SPACE*2}RETURN TO TOP MENU"
430 GET M2$
440 IF (M2$="D") THEN GOSUB 1280: GOTO 380
450 IF (M2$="B") THEN GOSUB 1670: GOTO 380
460 IF (M2$="H") THEN GOSUB 1800: GOTO 380
470 IF (M2$="R") THEN RETURN
480 GOTO 430
490 REM *** CONVERT SINGLE DECIMAL INTEGER ***
```

```
500 D$ = ""
510 PRINT: INPUT "ENTER DECIMAL INTEGER (UP TO 65535): ",D$
520 GOSUB 1030: REM VALIDATE DECIMAL INPUT
530 IF (V = 0) THEN GOTO 510
540 PRINT: PRINT " DEC";SPC(4);"BIN";SPC(19);"HEX"
550 L = 5: GOSUB 1930: L1$ = L$
560 L = 20: GOSUB 1930: L2$ = L$
570 PRINT SPC(1);L1$;SPC(2);L2$;SPC(2);L1$
580 FOREGROUND 7
590 B$ = ""
600 D1 = 0
610 IF (D < 256) THEN GOTO 660
620 D1 = INT(D / 256)
630 FOR N = 1 TO 8
640 :  IF ((D1 AND P(8 - N)) > 0) THEN B$ = B$ + "1": ELSE B$ = B$ + "0"
650 NEXT N
660 IF (D < 256) THEN B$ = "%" + B$: ELSE B$ = "%" + B$ + " "
670 D2 = D - 256*D1
680 FOR N = 1 TO 8
690 :  IF ((D2 AND P(8 - N)) > 0) THEN B$ = B$ + "1": ELSE B$ = B$ + "0"
700 NEXT N
710 H$ = HEX$(D)
720 IF (D < 256) THEN H$ = "{SPACE*2}$" + RIGHT$(H$,2): ELSE H$ = "$" + H$
730 IF (D < 256) THEN PRINT SPC(6 - LEN(D$)); D$;SPC(12) + MID$(B$,1,5) +
" " + MID$(B$,6,10); "{SPACE*2}" + H$: FOREGROUND 1: RETURN
740 PRINT SPC(6 - LEN(D$));D$;"{SPACE*2}" + MID$(B$,1,5) + " " + MID$(B$,6,4) +
MID$(B$,10,5) + " " + MID$(B$,15,4); "{SPACE*2}" + H$: FOREGROUND 1: RETURN
750 REM *** CONVERT SINGLE BINARY INTEGER ***
760 I$=""
770 PRINT: INPUT "ENTER BINARY INTEGER (UP TO 16 BITS): ",I$
780 GOSUB 1110: REM VALIDATE BINARY INPUT
790 IF (V = 0) THEN GOTO 760: ELSE GOTO 540
800 REM *** CONVERT SINGLE HEXADECIMAL INTEGER ***
810 H$=""
820 PRINT: INPUT "ENTER HEXADECIMAL INTEGER (UP TO 4 DIGITS): ",H$
830 GOSUB 1220: REM VALIDATE HEXADECIMAL INPUT
840 IF (V = 0) THEN GOTO 810: ELSE GOTO 540
850 REM *** VALIDATE DECIMAL INPUT STRING ***
860 FOR N = 1 TO LEN(D$)
```

```
870 :  M = ASC(MID$(D$,N,1)) - ASC("0")
880 :  IF ((M < 0) OR (M > 9)) THEN V = 0
890 NEXT N: RETURN
900 REM *** VALIDATE BINARY INPUT STRING ***
910 FOR N = 1 TO LEN(I$)
920 :  M = ASC(MID$(I$,N,1)) - ASC("0")
930 :  IF ((M < 0) OR (M > 1)) THEN V = 0
940 NEXT N: RETURN
950 REM *** VALIDATE HEXADECIMAL INPUT STRING ***
960 FOR N = 1 TO LEN(H$)
970 :  M = ASC(MID$(H$,N,1)) - ASC("0")
980 :  IF (NOT (((M >= 0) AND (M <= 9)) OR
((M >= 17) AND (M <= 22)))) THEN V = 0
990 NEXT N: RETURN
1000 REM *** OUTPUT ERROR MESSAGE ***
1010 FOREGROUND 2: PRINT: PRINT A$: FOREGROUND 1: RETURN
1020 REM *** VALIDATE DECIMAL INPUT ***
1030 V = 1: GOSUB 860: REM VALIDATE DECIMAL INPUT STRING
1040 IF (V = 0) THEN A$ = "INVALID DECIMAL NUMBER": GOSUB 1010
1050 IF (V = 1) THEN BEGIN
1060 :  D = VAL(D$)
1070 :  IF ((D < 0) OR (D > 65535)) THEN A$ = "DECIMAL NUMBER OUT OF RANGE":
GOSUB 1010: V = 0
1080 BEND
1090 RETURN
1100 REM *** VALIDATE BINARY INPUT ***
1110 V = 1: GOSUB 910: REM VALIDATE BINARY INPUT STRING
1120 IF (V = 0) THEN A$ = "INVALID BINARY NUMBER": GOSUB 1010: RETURN
1130 IF (LEN(I$) > 16) THEN A$ = "BINARY NUMBER OUT OF RANGE":
GOSUB 1010: V = 0 : RETURN
1140 IF (V = 1) THEN BEGIN
1150 :  I = 0
1160 :  FOR N = 1 TO LEN(I$)
1170 :    I = I + VAL(MID$(I$,N,1)) * P(LEN(I$) - N)
1180 :  NEXT N
1190 BEND
1200 D$ = STR$(I): D = I: RETURN
1210 REM *** VALIDATE HEXADECIMAL INPUT ***
1220 V = 1: GOSUB 960: REM VALIDATE HEXADECIMAL INPUT STRING
```

```
1230 IF (V = 0) THEN A$ = "INVALID HEXADECIMAL NUMBER": GOSUB 1010: RETURN
1240 IF (LEN(H$) > 4) THEN A$ = "HEXADECIMAL NUMBER OUT OF RANGE":
GOSUB 1010: V = 0: RETURN
1250 D = DEC(H$): D$ = STR$(D): H = D: RETURN
1260 RETURN
1270 REM *** CONVERT MULTIPLE DECIMAL INTEGERS ***
1280 DB$=""
1290 PRINT: INPUT "ENTER STARTING DECIMAL INTEGER (UP TO 65535): ", DB$
1300 D$=DB$: GOSUB 1030: D$="": REM VALIDATE DECIMAL INPUT
1310 IF (V = 0) THEN GOTO 1290
1320 DE$=""
1330 PRINT: INPUT "ENTER ENDING DECIMAL INTEGER (UP TO 65535): ", DE$
1340 D$=DE$: GOSUB 1030: D$="": REM VALIDATE DECIMAL INPUT
1350 IF (V = 0) THEN GOTO 1330
1360 DB=VAL(DB$): DE=VAL(DE$)
1370 IF (DE < DB) THEN A$ = E$: GOSUB 1010: GOTO 1280
1380 SC = 1: SM = INT(((DE - DB) / 36) + 1)
1390 D = DB
1400 FOR J = SC TO SM
1410 :  PRINT CHR$(147) + "RANGE: " + DB$ + " TO " + DE$ + "{SPACE*10}SCREEN: "
+ STR$(J) + " OF " + STR$(SM)
1420 :  PRINT: PRINT "DEC";SPC(4);"BIN";SPC(19);"HEX";SPC(8);"DEC";SPC(4);
"BIN";SPC(19);"HEX"
1430 L = 5: GOSUB 1930: L1$ = L$
1440 L = 20: GOSUB 1930: L2$ = L$
1450 :    PRINT SPC(1);L1$;SPC(2);L2$;SPC(2);L1$;SPC(6);L1$;SPC(2);
L2$;SPC(2);L1$
1460 :  FOR K = 0 TO 17
1470 :      FOREGROUND (7 + MOD(K,2))
1480 :      D$ = STR$(D): GOSUB 590: D = D + 1
1490 :      IF (D > DE) THEN GOTO 1630
1500 :  NEXT K
1510 :  PRINT CHR$(19): PRINT: PRINT: PRINT
1520 :  FOR K = 0 TO 17
1530 :      FOREGROUND (7 + MOD(K,2))
1540 :      D$ = STR$(D): PRINT TAB(40): GOSUB 590: D = D + 1
1550 :      IF (D > DE) THEN GOTO 1630
1560 :  NEXT K
1570 :  FOREGROUND 1: PRINT: PRINT SPC(19);
```

```
"PRESS X TO EXIT OR SPACEBAR TO CONTINUE..."
1580 :  GET B$
1590 :  IF B$="X" THEN RETURN
1600 :  IF B$=" " THEN GOTO 1620
1610 :  GOTO 1580
1620 NEXT J
1630 PRINT CHR$(19): FOR I = 1 TO 22: PRINT: NEXT I
1640 PRINT SPC(20); "COMPLETE. PRESS SPACEBAR TO CONTINUE..."
1650 GET B$: IF B$<>" " THEN GOTO 1650: ELSE RETURN
1660 REM *** CONVERT MULTIPLE BINARY INTEGERS ***
1670 IB$=""
1680 PRINT: INPUT "ENTER STARTING BINARY INTEGER (UP TO 16 BITS): ", IB$
1690 I$=IB$: GOSUB 1110: I$="": REM VALIDATE BINARY INPUT
1700 IF (V = 0) THEN GOTO 1680
1710 IB = I
1720 IE$=""
1730 PRINT: INPUT "ENTER ENDING BINARY INTEGER (UP TO 16 BITS): ", IE$
1740 I$=IE$: GOSUB 1110: I$="": REM VALIDATE BINARY INPUT
1750 IF (V = 0) THEN GOTO 1730
1760 IE = I
1770 IF (IE < IB) THEN A$ = E$: GOSUB 1010: GOTO 1670
1780 DB = IB: DE = IE: DB$ = STR$(IB): DE$ = STR$(IE): GOTO 1380
1790 REM *** CONVERT MULTIPLE HEXADECIMAL INTEGERS ***
1800 HB$=""
1810 PRINT: INPUT "ENTER STARTING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HB$
1820 H$=HB$: GOSUB 1220: H$="": REM VALIDATE HEXADECIMAL INPUT
1830 IF (V = 0) THEN GOTO 1810
1840 HB = H
1850 HE$=""
1860 PRINT: INPUT "ENTER ENDING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HE$
1870 H$=HE$: GOSUB 1220: H$="": REM VALIDATE HEXADECIMAL INPUT
1880 IF (V = 0) THEN GOTO 1860
1890 HE = H
1900 IF (HE < HB) THEN A$ = E$: GOSUB 1010: GOTO 1800
1910 DB = HB: DE = HE: DB$ = STR$(HB): DE$ = STR$(HE): GOTO 1380
1920 REM *** MAKE LINES ***
1930 L$=""
1940 FOR K = 1 TO L: L$ = L$ + "-": NEXT K
1950 RETURN
```

Bibliography

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158

[1] L. Soares and M. Stumm, “Flexsc: Flexible system call scheduling with exception-less system calls.” in *Osdi*, vol. 10, 2010, pp. 1–8.

[2] X. S. me, “”vic-ii for beginners: Screen modes, cheaper by the dozen,” XXX Set me. [Online]. Available: http://dustlayer.com/vic-ii/2013/4/26/vic-ii-for-beginners-screen-modes-cheaper-by-the-dozen

INDEX

1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102
103 104 105 106 107 108 109
110 111 112 113 114 115 116
117 118 119 120 121 122 123
124 125 126 127 128 129 130
131 132 133 134 135 136 137
138 139 140 141 142 143 144
145 146 147 148 149 150 151
152 153 154 155 156 157 158

PART V

ELEMENT CATALOGUE

GRAPHIC SYMBOLS FONT

Graphic chars and MEGA logo using the **graphicsymbol** macro:

Graphic chars using the **symbolfont** font definition:

The MEGA logo in default black using the **megasymbol** macro:

for tables and symbol usage.

The MEGA logo using a passed in colour:

Special multiline keys: RUN STOP CLR HOME NO SCRL HELP INST DEL SHIFT LOCK ESC ALT

KEYBOARD KEYS

MEGA key looks like this.

NORMAL SHIFT

BIG SHIFT

Text to the left RUN STOP and text to the right.

SHIFT CTRL 9 RETURN

* ← ↑ → ↓

SCREEN OUTPUT

```
10 INPUT A$
20 PRINT "YOU TYPED: ";A$
30 PRINT
40 GOTO 10
RUN
? MEGA 65
YOU TYPED: MEGA 65
```

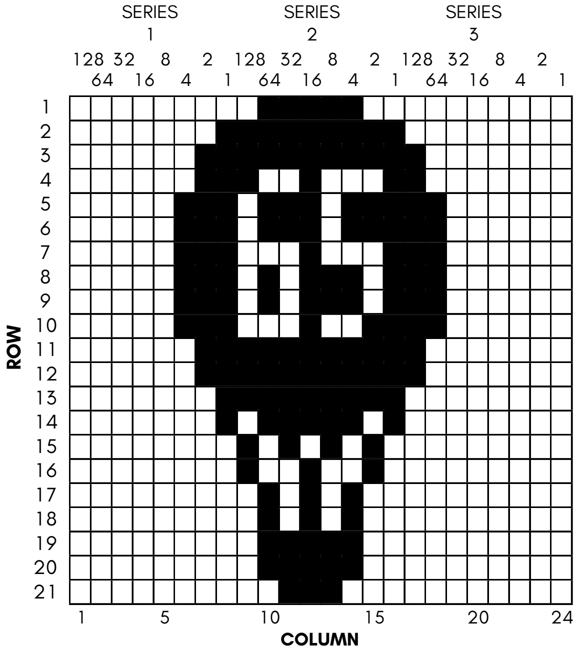
```
10 OPEN 1,8,0,"$0:*,P,R
20 : IF DS THEN PRINT DS$: GOTO 100
30 GET#1,X$,X$
40 DO
50 : GET#1,X$,X$: IF ST THEN EXIT
60 : GET#1,BL$,BH$
70 : LINE INPUT#1, F$
80 : PRINT LEFT$(F$,18)
90 LOOP
100 CLOSE 1

RUN
```

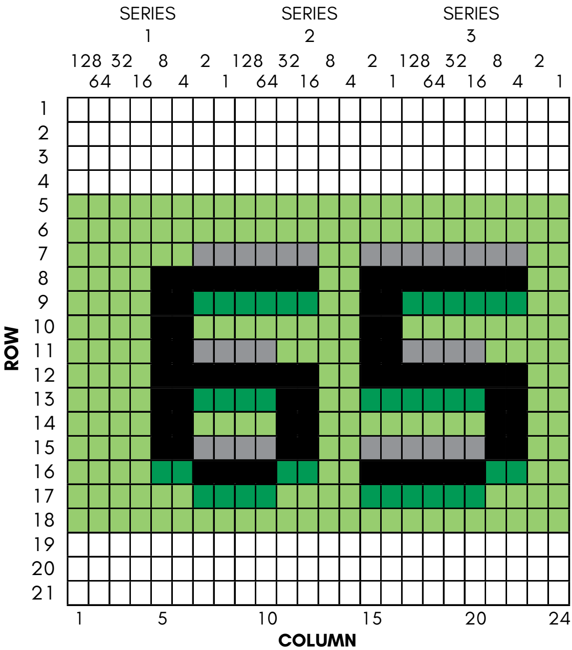
Use the "screentext" macro to perform inline screen text: `?SYNTAX ERROR`

SPRITE GRIDS

Balloon Sprite Demo



Multicolor Sprite


SERIES 1
SERIES 2
SERIES 3
128 32 8 2 128 32 8 2 128 32 8 2
64 16 4 1 64 16 4 1 64 16 4 1
ROW
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
1 5 10 15 20 24
COLUMN