MEGA65

BASIC 65 REFERENCE

MUSEUM OF ELECTRONIC GAMES & ART

MEGA65 TEAM

Assoc. Prof. Paul Gardner-Stephen
(highlander)
Founder
Software and virtual hardware architect
Spokesman and lead scientist

Martin Streit
(seriously)
Video and photo production
Tax and organization
Social media

Dan Sanderson
(dddaaannn)
Media and documentation
MEGA65.ROM

Dr. Edilbert Kirk
(Bit Shifter)
MEGA65.ROM
Manual and tools

Gábor Lénárt
(LGB)
Emulator

Farai Aschwanden
(Tayger)
Filehost and tools
Financial advisory

Falk Rehwagen
(bluewaysw)
GEOS

Robert Steffens
(kibo)
Network technology
Core bug hunting

Detlef Hastik
(deft)
Co-founder
General manager
Marketing and sales

Oliver Graf
(lydon)
Release management
VHDL and platform enhancements

Antti Lukats
(antti-brain)
Host hardware design and production

Dieter Penner
(doubleflash)
Host hardware support

Mirko H.
(sy2002)
Additional platforms and consulting

Gürçe Işıkyıldız
(gurce)
Tools and enhancements

Daniel England
(Mew Pokémon)
Additional code and tools

Hernán Di Pietro
(indiocolifa)
Additional emulation and tools

Roman Standzikowski
(FeralChild)
Open ROMs

Anton Schneider-Michallek
(adtbm)
Presentation and support

Reporting Errors and Omissions

This book is being continuously refined and improved upon by the MEGA65 community. The version of this edition is:

```
commit 06a35d2851767b1d9a560c8c33b3b7a1770f5e4a
date: Sun May 5 15:33:25 2024 +0930
```

We want this book to be the best that it possibly can. So if you see any errors, find anything that is missing, or would like more information, please report them using the MEGA65 User's Guide issue tracker:

https://github.com/mega65/mega65-user-guide/issues

You can also check there to see if anyone else has reported a similar problem, while you wait for this book to be updated.

Finally, you can always download the latest versions of our suite of books from these locations:

- https://mega65.org/mega65-book
- https://mega65.org/user-guide
- https://mega65.org/developer-guide
- https://mega65.org/basic65-ref
- https://mega65.org/chipset-ref
- https://mega65.org/docs

MEGA65 BASIC 65 REFERENCE

Published by
the MEGA Museum of Electronic Games & Art e.V., Germany.

WORK IN PROGRESS

May 5, 2024

Contents

CHAPTER 1

Introduction

- Welcome to the MEGA65!
- Other Books in this series
- Come Join Us!

WELCOME TO THE MEGA65!

Congratulations on your purchase of one of the most long-awaited computers in the history of computing! The MEGA65 is community designed, and based on the never-released Commodore® 65[1] computer; a computer designed in 1989 and intended for public release in 1990. Decades have passed, and we have endeavoured to invoke memories of an earlier time when computers were simple and friendly. They were not only simple to operate and understand, but friendly and approachable for new users.

These 1980s computers inspired many of their owners to pursue the exciting and rewarding technology careers they have today. Just imagine the exhilaration these early computing pioneers experienced, as they learned they could use their new computer to solve problems, write a letter, prepare taxes, invent new things, discover how the universe works, and perhaps even play an exciting game or two! We want to re-awaken that same level of excitement (which alas, is no longer found in modern computing), so we have created the **MEGA65**.

The MEGA65 team believes that owning a computer is like owning a home. You don't just use a home; you change things, big and small, to make it your own custom living space. After a while, when you settle in, you may decide to renovate or expand your home to make it more comfortable, or provide more utility. Think of the MEGA65 as your very own "computing home".

This guide will teach you how to do more than just hang pictures on a wall; it will show you how to build your dream home. While you read this user's guide, you will learn how to operate the MEGA65, write programs, add additional software, and extend hardware capabilities. What won't be immediately obvious is that along the journey, you will also learn about the history of computing as you explore the many facets of BASIC version 65 and operating system commands.

Computer graphics and music make computing more fun, and we designed the MEGA65 to be fun! In this user's guide, you will learn how to write programs using the MEGA65's built-in **graphics** and **sound** capabilities. But you don't need to be a programmer to have fun with the MEGA65. Because the MEGA65 includes a complete Commodore® 64™[2], it can also run thousands of existing games, utilities, and business software packages, as well as new programs being written today by Commodore computer enthusiasts. Excitement for the MEGA65 will grow as we all witness the programming marvels our MEGA65 community create, as they (and you!) discover and master the powerful capabilities of this modern Commodore computer recreation. Together, we can build a new "homebrew" community, teeming with software and projects that push the MEGA65's capabilities far beyond what anyone thought would be possible.

We welcome you on this journey! Thank you for becoming a part of the MEGA65 community of users, programmers, and enthusiasts!

---

[1] Commodore is a trademark of C= Holdings

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

OTHER BOOKS IN THIS SERIES

This book is one of several within the MEGA65 documentation suite. The series includes:

- **The MEGA65 User's Guide**
  Provides an introduction to the MEGA65, and a condensed BASIC 65 command reference
- **The MEGA65 BASIC 65 Reference**
  Comprehensive documentation of all BASIC 65 commands, functions and operators
- **The MEGA65 Chipset Reference**
  Detailed documentation about the MEGA65 and C65's custom chips
- **The MEGA65 Developer's Guide**
  Information for developers who wish to write programs for the MEGA65
- **The MEGA65 Complete Compendium**
  (Also known as **The MEGA65 Book**)
  All volumes in a single huge PDF for easy searching. 1200 pages and growing!

COME JOIN US!

Get involved, learn more about your MEGA65, and join us online at:

- https://mega65.org/chat
- https://mega65.org/forum

CHAPTER 2

BASIC 65 Command Reference

- **Commands, Functions, and Operators**
- **BASIC Command Reference**

COMMANDS, FUNCTIONS, AND OPERATORS

This appendix describes each of the commands, functions, and other callable elements of BASIC 65, which is an enhanced version of BASIC 10. Some of these can take one or more arguments, which are pieces of input that you provide as part of the command or function call, to help describe what you want to achieve. Some also require that you use special words.

Below is an example of how commands, functions, and operators (all of which are also known as keywords) will be described in this appendix.

KEY number**,** string

Here, **KEY** is a *keyword*. Keywords are special words that BASIC understands. In this manual, keywords are always written in **BOLD CAPITALS**, so that you can easily recognise them.

The "number" and "string" (in non-bold text) are examples of *arguments*. You replace these with values or algebraic phrases (*expressions*) that represent the data that controls the command's behavior.

Punctuation and other letters in bold text represent other characters that are typed as they appear. In this example, a comma must appear between the number argument and the string argument.

Here is an example of using the **KEY** command based on this pattern:

```
KEY 8,"LIST"+CHR$(13)
```

When you see square brackets around arguments, this indicates that the arguments are optional. You are not meant to type the square brackets. Consider this description of the **CIRCLE** command, which accepts optional arguments:

CIRCLE xc**,** yc**,** radius [**,** flags **,** start**,** stop]

The following examples of the **CIRCLE** command are both valid. They have different behavior based on their different arguments.

```
CIRCLE 100,150,30

CIRCLE 100,150,30,0,45,135
```

This arrangement of keywords, symbols, and arguments is called *syntax*. If you leave something out, or put the wrong thing in the wrong place, the computer will fail to understand the command and report a *syntax error*.

There is nothing to worry about if you get an error from the MEGA65. It is just the MEGA65's way of telling you that something isn't quite right, so that you can more easily find and fix the problem. For example, if you omit the comma in the **KEY** command, or replace it with a period, the MEGA65 will respond with a ?SYNTAX ERROR:

```
READY.
KEY 8"FISH"

?SYNTAX ERROR
READY.
KEY 8."FISH"

?SYNTAX ERROR
READY.
```

Expressions can be a number value such as `23.7`, a string value such as `"HELLO"`, or a more complex calculation that combines values, functions, and operators to describe a number or string value: `"LIST"+CHR$(13)`

It is important to use the correct type of expression when writing your programs. If you accidentally use the wrong type, the MEGA65 will display a `?TYPE MISMATCH ERROR`, to say that the type of expression you gave doesn't match what it expected. For example, the following command results in a `?TYPE MISMATCH ERROR`, because `"POTATO"` is a string expression, and a numeric expression is expected:

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

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

```
READY.
PRINT "HELLO"
HELLO

READY.
10 PRINT "HELLO"
RUN
HELLO
```

You can place a sequence of statements within a single line by separating them with colons, for example:

```
PRINT "HELLO" : PRINT "HOW ARE YOU?" : PRINT "HOW IS THE WEATHER?"
HELLO
HOW ARE YOU?
HOW IS THE WEATHER?
```

Direct Mode Commands

Some commands only work in **direct mode** (sometimes called "immediate mode"). This means that the command can't be part of a BASIC program, but can be entered directly to the screen. For example, the **RENUMBER** command only works in direct mode, because its function is to renumber the lines of a BASIC program.

In the two **PRINT** examples above, the first was entered in direct mode, whereas the second one was part of a program. The **PRINT** command works in both direct mode and in a program.

Command Syntax Descriptions

The following table describes the other symbols found in command syntax descriptions.

Symbol	Meaning
[ ]	Optional
...	The bracketed syntax can be repeated zero or more times
< | >	Include one of the choices
[|]	Optionally include one of the choices
{ , }	One or more of the arguments is required. The commas to the left of the last argument included are required. Trailing commas must be omitted. See **CURSOR** for an example.
[{ , }]	Similar to { , } but all arguments can be omitted

Fonts

Examples of text that appears on the screen, either typed by you or printed by the MEGA65, appear in the screen font: `"LIST"+CHR$(13)`

BASIC 65 Constants

Values that are typed directly into an expression or program are called *constants*. The values are “constant” because they do not change based on other aspects of the program state.

The following are types of constants that can appear in a BASIC 65 expression.

Type	Example	Example
Decimal Integer	32000	-55
Decimal Fixed Point	3.14	-7654.321
Decimal Floating Point	1.5E03	7.7E-02
Hex	$D020	$FF
Binary	%11010010	%101
String	"X"	"TEXT"

BASIC 65 Variables

A program manipulates data by storing values in the computer’s memory, referring to stored values, and updating them based on logic. In BASIC, elements of memory that store values are called *variables*. Each variable has a name, there are separate sets of variable names for each type of value.

For example, the variable AA can store a number value. The variable AA$ can store a string value. Commodore BASIC considers these to be separate variables, even though the names both begin with AA.

One way to store a value in a variable is with the assignment = operator. For example:

```
AA = 1.95

AA$ = "HELLO, "
```

Variable names must start with a letter, and contain only letters and numbers. They can be of any length, but Commodore BASIC only recognizes the first two letters of the name. SPEED and SP would be considered the same variable.

Variable names cannot contain any of the BASIC keywords. This makes using long names difficult: it is easy to use a keyword accidentally. For example, ENFORCEMENT is not a valid variable name, because **FOR** is a keyword. It is common to use short variable names to avoid these hazards.

A variable can be used within an expression with other constants, variables, functions, and operators. It is substituted with the value that it contains at that point in the program’s execution.

```
10 INPUT "WHAT IS YOUR NAME";NA$
20 MSG$ = "HELLO, "+NA$+"!"
20 PRINT MSG$
```

Unlike some programming languages, BASIC variables do not need to be declared before use. A variable has a default value of zero for number type variables, or the empty string ("") for string type variables.

A variable that stores a single value is also known as a *scalar variable*. The scalar variable types and their value ranges are as follows.

Type	Name Symbol	Range	Example
Byte	&	0 .. 255	`BY& = 23`
Integer	%	-32768 .. 32767	`I% = 5`
Real	none	-1E37 .. 1E37	`XY = 1/3`
String	$	length = 0 .. 255	`AB$ = "TEXT"`

A variable whose name is a single letter followed by the type symbol (or no symbol for real number variables) is a *fast variable*. BASIC 65 stores the variable in a way that makes it faster to access or update the value than variables with longer names. It otherwise behaves like any other variable. This is also true for functions defined by **DEF FN**.

BASIC 65 Arrays

In addition to scalar variables, Commodore BASIC also supports a type of variable that can store multiple values, called an *array*.

The following example stores three string values in an array, then uses a **FOR** loop to **PRINT** a message for each element:

```
10 DIM NA$(3)
20 NA$(0) = "DEFT"
30 NA$(1) = "GARDNERS"
40 NA$(2) = "LYDON"
50 FOR I=0 TO 3
60 PRINT "HELLO, ";NA$(I);"!"
70 NEXT I
```

Each value in an array is referenced by the name of the array variable and an integer index. For example, `AA(7)` refers to the element of the array named `AA()` with index 7. Indexes are “zero-based:” the first element in the array has an index of 0. The index can be a numeric expression, which can be a powerful way to operate on multiple elements of data.

All values in an array must be of the same type. The type is indicated in the name of the variable, similar to scalar variables. `AA()` is an array of real numbers, `AA$()` is an array of strings.

Array variable names are considered separate from scalar variable names. The scalar variable `AA` has no relationship to the array variable `AA()`.

BASIC needs to know the maximum size of the array before its first use, so that it can allocate the memory for the complete array. A program can declare an array's size using the **DIM** keyword, with the "dimensions" of the array. If an array variable is used without an explicit declaration, BASIC allocates a one-dimensional array of 10 elements, and the array cannot be re-dimensioned later (unless you **CLR** all variables).

An array can have multiple dimensions, each with its own index separated by a comma. The array must be declared with the maximum value for each dimension. Keep in mind that BASIC 65 allocates memory for the entire array, so large arrays may be constrained by available memory.

```
DIM BO$(3,3)
BO$(1,1) = "X"
BO$(0,0) = "O"
BO$(0,2) = "X"
BO$(1,0) = "O"
```

Screen Text and Colour Arrays

A BASIC 65 program can place text on the screen in several ways. The **PRINT** command displays a string at the current cursor location, which is especially useful for terminal-like output. The **CURSOR** command moves the cursor to a given position. A program can use these commands together to draw pictures or user interfaces.

A program can access individual characters on the screen using the special built-in arrays **T@&()** and **C@&()**. These arrays are two-dimensional with indexes corresponding to the column and row of each character on the screen, starting from (0,0) at the top left corner.

T@&(column**,** row**)** is the screen code of the character. Screen codes are not the same as PETSCII codes. See appendix 3 on page 277 for a list of screen codes.

C@&(column**,** row**)** is the colour code of the character. This is an entry number of the system palette. See appendix 6 on page 297 for the list of colours in the default system palette. Upper bits also set text attributes, such as blinking.

Like regular arrays, the screen and colour array entries can be assigned new values, or used in expressions to refer to their current values.

```
10 FOR X=10 TO 30
20 T@&(X,2)=1
30 C@&(X,2)=INT(RND(1)*16)
40 NEXT X
50 PRINT "COLOUR AT POSITION 15: ";C@&(15,2)
```

The dimensions of these arrays depend on the current text screen mode. In 80 $\times$ 25 text mode, the column is in the range 0 – 79, and the row is in the range 0 – 24. The MEGA65 also supports 80 $\times$ 50 and 40 $\times$ 25 text modes.

BASIC 65 Operators

An *operator* is a symbol or keyword that performs a function in an expression. It operates on one or two sub-expressions, called *operands*. The operator and its operands *evaluate to* the result of the operation.

For example, the * (asterisk) operator performs a multiplication of two number operands. The operator and its operands evaluate to the result of the multiplication.

```
A=6
PRINT A*7
```

The + (plus) operator has a different meaning depending on the type of the operands. If both operands are numbers, then the operator performs an addition of the numbers. If both operands are strings, then the operator evaluates to a new string that is the concatenation of the operands.

```
A=64
PRINT A+1

A$="MEGA"
PRINT A$+"65"
```

The - (minus) operator accepts either one operand or two operands. Given one number operand on the right-hand side, it evaluates to the negation of that number. Given two number operands, one on either side, it evaluates to the subtraction of the second operand from the first operand.

```
A=64
PRINT -A

PRINT A-16
```

The = symbol is used both as an assignment statement and as a relational operator. As an assignment, the = symbol is a statement that updates the value of a variable. The left-hand side must be a variable or array element reference, and its type must match the type of the expression on the right-hand side. The assignment is not an operator: it is not part of an expression.

```
AA=7
NA$="DEFT"
```

As a relational operator, the = symbol behaves as an expression. It evaluates the expressions on both sides of the operator, then tests whether the values are equal. If they are equal, the equality operator evaluates to -1, BASIC's representation of "true." If they are not equal, the operator evaluates to 0, or "false." The equality expression can be used with an **IF** statement to control program flow, or it can be used as part of a numeric expression. Both expressions must be of the same type.

```
100 IF X=99 THEN 130
110 X=X+1
120 GOTO 100
130 PRINT "DONE."
```

BASIC 65 knows the difference between assignment and equality based on context. Consider this line of code:

```
A = B = 10
```

BASIC 65 expects a statement, and notices a variable name followed by the = symbol. It concludes that this is a statement assigning a value to the number variable `A`. It then expects a number expression on the right-hand side of the assignment, and notices the = symbol is an operator in that expression. It concludes that the operation is an equality test, and proceeds to evaluate the expression and assign the result.

The operators **NOT**, **AND**, **OR** and **XOR** can be used either as logical operators or as boolean operators. A logical operator joins two conditional expressions as operands and evaluates to the logical comparison of their truth values.

```
IF X=99 OR Y<5 THEN 130

IF Y>10 AND Y<20 THEN 150
```

A boolean operator accepts two number operands and performs a calculation on the bits of the binary values.

```
A=17
PRINT A AND 20
```

Unlike other cases where operators have different behaviors based on how they are used, BASIC 65 does not need to determine whether these operators are behaving as logical operators or boolean operators. Because "true" and "false" are represented by carefully chosen numbers, the logical operators have the same behavior whether their operands are conditional expressions or numbers. A "true" conditional expression is the number -1, which internally is a binary number with all bits set. The logical expression "true and false" is equivalent to the binary boolean expression `%....0000` & `%....1111`. In this case, the **AND** operator evaluates to 0, which is "false."

Conditional expressions evaluating to numbers can be used in some clever programming tricks. Consider this example:

```
A = A - (B > 7)
```

This statement will increment the value in the `A` by 1 if the value in `B` is greater than 7. Otherwise it leaves it unchanged. If the sub-expression `B > 7` is true, then it evaluates to -1. `A - (-1)` is equivalent to `A + 1`. If the sub-expression is false, then it evaluates to 0, and `A - 0` is equivalent to `A`.

When multiple operators are used in a single expression, the order in which they are evaluated is specified by *precedence*. For example, in the statement `A * A - B * B`, both multiplications will be performed first, then the subtraction. As in algebra, you can use parentheses to change the order of execution. In the expression `A * (A - B) * B`, the subtraction is performed first.

The complete set of operators and their order of precedence are summarised in the sections that follow.

Assignment Statement

Symbol	Description	Examples
=	Assignment	`A = 42, A$ = "HELLO", A = B < 42`

Unary Mathematical Operators

Name	Symbol	Description	Example
Plus	+	Positive sign	`A = +42`
Minus	-	Negative sign	`B = -42`

Binary Mathematical Operators

Name	Symbol	Description	Example
Plus	+	Addition	`A = B + 42`
Minus	-	Subtraction	`B = A - 42`
Asterisk	*	Multiplication	`C = A * B`
Slash	/	Division	`D = B / 13`
Up Arrow	↑	Exponentiation	`E = 2 ↑ 10`
Left Shift	<<	Left Shift	`A = B << 2`
Right Shift	>>	Right Shift	`A = B >> 1`

NOTE: The ↑ character used for exponentiation is entered with ↑, which is next to RESTORE.

Relational Operators

Symbol	Description	Example
>	Greater Than	A > 42
>=	Greater Than or Equal To	B >= 42
<	Less Than	A < 42
<=	Less Than or Equal To	B <= 42
=	Equal	A = 42
<>	Not Equal	B <> 42

Logical Operators

Keyword	Description	Example
AND	And	A > 42 AND A < 84
OR	Or	A > 42 OR A = 0
XOR	Exclusive Or	A > 42 XOR B > 42
NOT	Negation	C = NOT A > B

Boolean Operators

Keyword	Description	Example
AND	And	A = B AND $FF
OR	Or	A = B OR $80
XOR	Exclusive Or	A = B XOR 1
NOT	Negation	A = NOT 22

String Operator

Name	Symbol	Description	Operand type	Example
Plus	+	Concatenates Strings	String	A$ = B$ + ".PRG"

Operator Precedence

Precedence	Operators
High	↑
	+ - (Unary Mathematical)
	* /
	+ - (Binary Mathematical)
	<< >> (Arithmetic Shifts)
	< <= > >= = <>
	NOT
	AND
Low	OR XOR

Keywords And Tokens Part 1

*	AC	COLLECT	F3	EXP	BD
+	AA	COLLISION	FE17	FAST	FE25
-	AB	COLOR	E7	FGOSUB	FE48
/	AD	CONCAT	FE13	FGOTO	FE47
<	B3	CONT	9A	FILTER	FE03
<<	FE52	COPY	F4	FIND	FE2B
=	B2	COS	BE	FN	A5
>	B1	CURSOR	FE41	FONT	FE46
>>	FE53	CUT	E4	FOR	81
ABS	B6	DATA	83	FOREGROUND	FE39
AND	AF	DCLEAR	FE15	FORMAT	FE37
APPEND	FE0E	DCLOSE	FE0F	FRE	B8
ASC	C6	DEC	D1	FREAD#	FE1C
ATN	C1	DEF	96	FREEZER	FE4A
AUTO	DC	DELETE	F7	FWRITE#	FE1E
BACKGROUND	FE3B	DIM	86	GCOPY	FE32
BACKUP	F6	DIR	EE	GET	A1
BANK	FE02	DISK	FE40	GO	CB
BEGIN	FE18	DLOAD	F0	GOSUB	8D
BEND	FE19	DMA	FE1F	GOTO	89
BIT	FE4E	DMODE	FE35	GRAPHIC	DE
BLOAD	FE11	DO	EB	HEADER	F1
BOOT	FE1B	DOPEN	FE0D	HELP	EA
BORDER	FE3C	DOT	FE4C	HEX$	D2
BOX	E1	DPAT	FE36	HIGHLIGHT	FE3D
BSAVE	FE10	DSAVE	EF	IF	8B
BUMP	CE03	DVERIFY	FE14	INFO	FE4D
BVERIFY	FE28	ECTORY	FE29	INPUT	85
CATALOG	FE0C	EDIT	FE45	INPUT#	84
CHANGE	FE2C	EDMA	FE21	INSTR	D4
CHAR	E0	ELLIPSE	FE30	INT	B5
CHDIR	FE4B	ELSE	D5	JOY	CF
CHR$	C7	END	80	KEY	F9
CIRCLE	E2	ENVELOPE	FE0A	LEFT$	C8
CLOSE	A0	ERASE	FE2A	LEN	C3
CLR	9C	ERR$	D3	LET	88
CMD	9D	EXIT	ED	LINE	E5

Keywords And Tokens Part 2

LIST	9B	PRINT	99	SOUND	DA
LOAD	93	PRINT#	98	SPC(	A6
LOADIFF	FE43	PUDEF	DD	SPEED	FE26
LOCK	FE50	RCOLOR	CD	SPRCOLOR	FE08
LOG	BC	RCURSOR	FE42	SPRITE	FE07
LOG10	CE08	READ	87	SPRSAV	FE16
LOOP	EC	RECORD	FE12	SQR	BA
LPEN	CE04	REM	8F	STEP	A9
MEM	FE23	RENAME	F5	STOP	90
MERGE	E6	RENUMBER	F8	STR$	C4
MID$	CA	RESTORE	8C	SYS	9E
MKDIR	FE51	RESUME	D6	TAB(	A3
MOD	CE0B	RETURN	8E	TAN	C0
MONITOR	FA	RGRAPHIC	CC	TEMPO	FE05
MOUNT	FE49	RIGHT$	C9	THEN	A7
MOUSE	FE3E	RMOUSE	FE3F	TO	A4
MOVSPR	FE06	RND	BB	TRAP	D7
NEW	A2	RPALETTE	CE0D	TROFF	D9
NEXT	82	RPEN	D0	TRON	D8
NOT	A8	RPLAY	CE0F	TYPE	FE27
OFF	FE24	RREG	FE09	UNLOCK	FE4F
ON	91	RSPCOLOR	CE07	UNTIL	FC
OPEN	9F	RSPEED	CE0E	USING	FB
OR	B0	RSPPOS	CE05	USR	B7
PAINT	DF	RSPRITE	CE06	VAL	C5
PALETTE	FE34	RUN	8A	VERIFY	95
PASTE	E3	RWINDOW	CE09	VIEWPORT	FE31
PEEK	C2	SAVE	94	VOL	DB
PEN	FE33	SAVEIFF	FE44	VSYNC	FE54
PIXEL	CE0C	SCNCLR	E8	WAIT	92
PLAY	FE04	SCRATCH	F2	WHILE	FD
POINTER	CE0A	SCREEN	FE2E	WINDOW	FE1A
POKE	97	SET	FE2D	WPEEK	CE10
POLYGON	FE2F	SGN	B4	WPOKE	FE1D
POS	B9	SIN	BF	XOR	E9
POT	CE02	SLEEP	FE0B	^	AE

Tokens And Keywords Part 1

80 END	A5 FN	CA MID$
81 FOR	A6 SPC(	CB GO
82 NEXT	A7 THEN	CC RGRAPHIC
83 DATA	A8 NOT	CD RCOLOR
84 INPUT#	A9 STEP	CF JOY
85 INPUT	AA +	D0 RPEN
86 DIM	AB -	D1 DEC
87 READ	AC *	D2 HEX$
88 LET	AD /	D3 ERR$
89 GOTO	AE ^	D4 INSTR
8A RUN	AF AND	D5 ELSE
8B IF	B0 OR	D6 RESUME
8C RESTORE	B1 >	D7 TRAP
8D GOSUB	B2 =	D8 TRON
8E RETURN	B3 <	D9 TROFF
8F REM	B4 SGN	DA SOUND
90 STOP	B5 INT	DB VOL
91 ON	B6 ABS	DC AUTO
92 WAIT	B7 USR	DD PUDEF
93 LOAD	B8 FRE	DE GRAPHIC
94 SAVE	B9 POS	DF PAINT
95 VERIFY	BA SQR	E0 CHAR
96 DEF	BB RND	E1 BOX
97 POKE	BC LOG	E2 CIRCLE
98 PRINT#	BD EXP	E3 PASTE
99 PRINT	BE COS	E4 CUT
9A CONT	BF SIN	E5 LINE
9B LIST	C0 TAN	E6 MERGE
9C CLR	C1 ATN	E7 COLOR
9D CMD	C2 PEEK	E8 SCNCLR
9E SYS	C3 LEN	E9 XOR
9F OPEN	C4 STR$	EA HELP
A0 CLOSE	C5 VAL	EB DO
A1 GET	C6 ASC	EC LOOP
A2 NEW	C7 CHR$	ED EXIT
A3 TAB(	C8 LEFT$	EE DIR
A4 TO	C9 RIGHT$	EF DSAVE

Tokens And Keywords Part 2

F0	DLOAD	FE09	RREG	FE2F	POLYGON
F1	HEADER	FE0A	ENVELOPE	FE30	ELLIPSE
F2	SCRATCH	FE0B	SLEEP	FE31	VIEWPORT
F3	COLLECT	FE0C	CATALOG	FE32	GCOPY
F4	COPY	FE0D	DOPEN	FE33	PEN
F5	RENAME	FE0E	APPEND	FE34	PALETTE
F6	BACKUP	FE0F	DCLOSE	FE35	DMODE
F7	DELETE	FE10	BSAVE	FE36	DPAT
F8	RENUMBER	FE11	BLOAD	FE37	FORMAT
F9	KEY	FE12	RECORD	FE39	FOREGROUND
FA	MONITOR	FE13	CONCAT	FE3B	BACKGROUND
FB	USING	FE14	DVERIFY	FE3C	BORDER
FC	UNTIL	FE15	DCLEAR	FE3D	HIGHLIGHT
FD	WHILE	FE16	SPRSAV	FE3E	MOUSE
CE02	POT	FE17	COLLISION	FE3F	RMOUSE
CE03	BUMP	FE18	BEGIN	FE40	DISK
CE04	LPEN	FE19	BEND	FE41	CURSOR
CE05	RSPPOS	FE1A	WINDOW	FE42	RCURSOR
CE06	RSPRITE	FE1B	BOOT	FE43	LOADIFF
CE07	RSPCOLOR	FE1C	FREAD#	FE44	SAVEIFF
CE08	LOG10	FE1D	WPOKE	FE45	EDIT
CE09	RWINDOW	FE1E	FWRITE#	FE46	FONT
CE0A	POINTER	FE1F	DMA	FE47	FGOTO
CE0B	MOD	FE21	EDMA	FE48	FGOSUB
CE0C	PIXEL	FE23	MEM	FE49	MOUNT
CE0D	RPALETTE	FE24	OFF	FE4A	FREEZER
CE0E	RSPEED	FE25	FAST	FE4B	CHDIR
CE0F	RPLAY	FE26	SPEED	FE4C	DOT
CE10	WPEEK	FE27	TYPE	FE4D	INFO
FE02	BANK	FE28	BVERIFY	FE4E	BIT
FE03	FILTER	FE29	ECTORY	FE4F	UNLOCK
FE04	PLAY	FE2A	ERASE	FE50	LOCK
FE05	TEMPO	FE2B	FIND	FE51	MKDIR
FE06	MOVSPR	FE2C	CHANGE	FE52	<<
FE07	SPRITE	FE2D	SET	FE53	>>
FE08	SPRCOLOR	FE2E	SCREEN	FE54	VSYNC

BASIC COMMAND REFERENCE

ABS

Token: $B6

Format: **ABS**(x)

Returns: The absolute value of the numeric argument **x**.

x numeric argument (integer or real expression)

Remarks: The result is of type real.

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: Performs a bit-wise 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 (losing precision). Logical operands are converted to 16-bit integer using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

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

Examples: Using **AND**

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

AND can be used in **IF** statements to require multiple conditions.

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

APPEND

Token: $FE $0E

Format: **APPEND#** channel**,** 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.

channel number, where:

- **1** <= **channel** <= **127** line terminator is CR.
- **128** <= **channel** <= **255** line terminator is CR LF.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **APPEND#** works similarly to **DOPEN#... ,W**, except that the file must already exist. The content of the file is retained, and all printed text is appended to the end. Trying to **APPEND** to a non-existing file reports a DOS error.

Examples: Open existing 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)

Returns: The PETSCII code of the first character of the string argument, as a number.

Remarks: **ASC** returns zero for an empty string. This is different to BASIC 2, which raised an error for `ASC("")`.

The inverse function to **ASC** is **CHR$**. Refer to the **CHR$** function on page 47 for more information.

The name was apparently chosen to be a mnemonic to “ASCII,” but the returned value is a PETSCII code.

Examples: Using **ASC**

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

ATN

Token: $C1

Format: **ATN**(numeric expression)

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**.

Examples: Using **ATN**

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

AUTO

Token: $DC

Format: **AUTO** [step]

Usage: Enables or disables automatic line numbering during BASIC program entry. After submitting a new program line to the BASIC editor with RETURN, 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 disables it.

Examples: Using **AUTO**

```
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.

colour the palette entry number, in the range 0 - 255

All colours within this range are customisable via the **PALETTE** command. On startup, the MEGA65 only has the first 32 colours configured. See appendix 6 on page 297 for the list of colours in the default system palette.

Example: Using **BACKGROUND**

```
BACKGROUND 3 : REM SELECT BACKGROUND COLOUR CYAN
```

BACKUP

Token: $F6

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

Usage: Copies one disk to another.

The first form of **BACKUP**, specifying units for source and target, can only be used for the drives connected to the internal FDC (Floppy Disk Controller). Units 8 and 9 are reserved for this controller. These can be either the internal floppy drive (unit 8) and another floppy drive (unit 9) attached to the same ribbon cable, or mounted D81 disk images. **BACKUP** can be used to copy from floppy to floppy, floppy to image, image to floppy and image to image, depending on image mounts and the existence of a second physical floppy drive.

The second form of **BACKUP**, specifying drives for source and target, is meant to be used for dual drive units connected to the IEC bus. For example: CBM 4040, 8050, 8250 via an IEEE-488 to IEC adapter. In this case, the backup is then done by the disk unit internally.

source unit or drive # of source disk.
target unit or drive # of target disk.

Remarks: The target disk will be formatted and an identical copy of the source disk will be written.
BACKUP cannot be used to backup from internal devices to IEC devices or vice versa.

Examples: Using **BACKUP**

```
BACKUP U8 TO U9      : REM BACKUP INTERNAL DRIVE 8 TO DRIVE 9
BACKUP U9 TO U8      : REM BACKUP DRIVE 9 TO INTERNAL DRIVE 8
BACKUP D0 TO D1, U10 : REM BACKUP ON DUAL DRIVE CONNECTED VIA IEC
```

BANK

Token: $FE $02

Format: **BANK** bank number

Usage: Selects the memory configuration for BASIC commands that use 16-bit addresses. These are **LOAD**, **LOADIFF**, **PEEK**, **POKE**, **SAVE**, **SYS**, and **WAIT**. Refer to the system memory map in *the MEGA65 Book*, System Memory Map (Appendix I) for more information.

Remarks: A value > 127 selects memory mapped I/O. The default value at system startup for the bank number is 128. This configuration has RAM from $0000 to $1FFF, the BASIC and KERNAL ROM, and I/O from $2000 to $FFFF.

Example: Using **BANK**

```
BANK 1   :REM SELECT MEMORY CONFIGURATION 1
```

BEGIN

Token: $FE $18

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

Usage: The beginning of a compound statement to be executed after **THEN** or **ELSE**. 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, as 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 end of a compound statement to be executed after **THEN** or **ELSE**. 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 the 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] [**,R**] [**,D** drive] [**,U** unit]

Usage: Loads a file of type **PRG** into RAM at address P. ("Binary load.")

BLOAD has two modes: The flat memory address mode can be used to load a program to any address in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address $8000000.

This mode is triggered by specifying an address at parameter P that is larger than $FFFF. The bank parameter is ignored in this mode.

For compatibility reasons with older BASIC versions, **BLOAD** accepts the syntax with a 16-bit address at P and a bank number at B as well. The attic RAM is out of range for this compatibility mode.

The optional parameter **R** (RAW MODE) does not interpret or use the first two bytes of the program file as the load address, which is otherwise the default behaviour. In RAW MODE every byte is read as data.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

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 overrides 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 such as the 1541, 1571, or 1581.

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

Remarks: **BLOAD** cannot cross bank boundaries.

BLOAD uses the load address from the file if no P parameter is given.

Examples: Using **BLOAD**

```
BLOAD "ML DATA", B0, U9
BLOAD "SPRITES"
BLOAD "ML ROUTINES", B1, P32768
BLOAD (FI$), B(BA%), P(PA), U(UN%)
BLOAD "CHUNK",P($8000000)         :REM LOAD TO ATTIC RAM
```

BOOT

Token: $FE $1B

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

Usage: Loads and runs a program or boot sector from a disk.

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 (512 bytes in total) from sector 0, track 1 and unit 8 to address $0400 in bank 0, and performs a `JSR $0400` afterwards (Jump To Subroutine).

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

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

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 overrides 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 such as the 1541, 1571, or 1581.

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

Examples: Using **BOOT**

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

BORDER

Token: $FE $3C

Format: **BORDER** colour

Usage: Sets the border colour of the screen.

colour the palette entry number, in the range 0 – 255

All colours within this range are customisable via the **PALETTE** command. See appendix 6 on page 297 for the list of colours in the default system palette.

Example: Using **BORDER**

```
10 BORDER  4 : REM SELECT BORDER COLOUR PURPLE
```

BOX

Token: $E1

Format: **BOX** x0,y0, x2,y2 [, solid]
BOX x0,y0, x1,y1, x2,y2, x3,y3 [, solid]

Usage: Bitmap graphics: draws a box.

The first form of **BOX** with two coordinate pairs and an optional **solid** parameter draws a simple rectangle, assuming that the coordinate pairs declare two diagonally opposite corners.

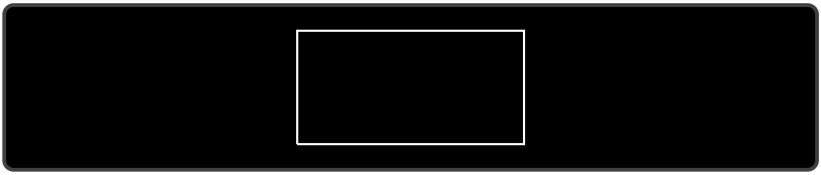
The second form with four coordinate pairs declares a path of four points, which will be connected with lines. The path is closed by connecting the last coordinate with the first.

The quadrangle is drawn using the current drawing context set with **SCREEN**, **PALETTE** and **PEN**. The quadrangle is filled if the parameter **solid** is not 0.

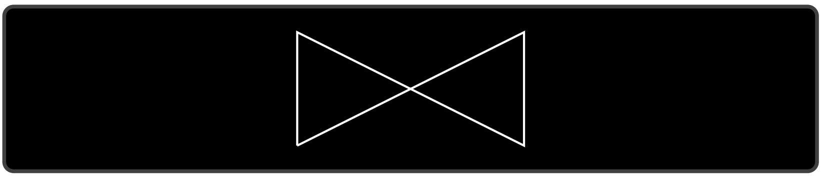
Remarks: **BOX** can be used with four coordinate pairs to draw any shape that can be defined with four points, not only rectangles. For example rhomboids, kites, trapezoids and parallelograms. It is also possible to draw bow tie shapes.

Examples: Using **BOX**

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



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



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

BSAVE

Token: $FE $10

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

Usage: Saves a memory range to a file of type **PRG**. ("Binary save.")

BSAVE has two modes: The flat memory address mode can be used to save a memory block in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address $8000000.

This mode is triggered by specifying addresses for the start and end parameter P, that are larger than $FFFF. The bank parameter is ignored in this mode. This flat memory mode allows saving ranges greater than 64K.

For compatibility reasons with older BASIC versions, **BSAVE** accepts the syntax with 16-bit addresses at P and a bank number at B as well. The attic RAM is out of range for this compatibility mode. This mode cannot cross bank boundaries, so start and end address must be in the same bank.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`. If the first character of the filename is an at sign '`@`', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

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

end address where the saving ends. **end-1** is the last address to be used for saving.

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

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: The length of the file is **end - start + 2**.
If the number after an argument letter is not a decimal number, it must be set in parenthesis, as shown in the third and fourth line of the examples.

The **PRG** file format that is used by **BSAVE** requires the load address to be written to the first two bytes. If the saving is done with a bank number that is not zero, or a start address greater than $FFFF, this information will not fit. For compatibility reasons, only the two low order bytes are written. Loading the file with the **BLOAD** command will then require the full 16-bit range of the load address as a parameter.

Examples: Using **BSAVE**

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

BUMP

Token: $CE $03

Format: **BUMP**(type)

Returns: A bitfield of sprites currently colliding with other sprites (type= 1) or screen data (type=2).

Each bit set in the returned value indicates that the sprite corresponding to that bit position was involved in a collision since the last call of **BUMP**. Calling **BUMP** resets the collision mask, so you will always get a summary of collisions encountered since the last call of **BUMP**.

Remarks: It's possible to detect multiple collisions, but you will need to evaluate the sprite coordinates to detect which sprites have collided.

Example: Using **BUMP**

```
10 S% = BUMP(1) : REM SPRITE-SPRITE COLLISION
20 IF (S% AND 6) = 6 THEN PRINT "SPRITE 1 & 2 COLLISION"
30 REM ---
40 S% = BUMP(2) : REM SPRITE-DATA COLLISION
50 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: Compares a memory range to a file of type **PRG**. ("Binary verify.")

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

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 that will be used.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 **BVERIFY** 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.

Examples: Using **BVERIFY**

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

CATALOG

Token: $FE $0C

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

Usage: Prints a file catalog/directory of the specified disk.

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

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

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

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **CATALOG** is a synonym of **DIRECTORY** and **DIR**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and **?** may be used. Adding **,T=** to the pattern string, with **T** specifying a filetype of **P**, **S**, **U** or **R** (for **P**RG, **S**EQ, **U**SR, **R**EL) filters the output to that filetype.

The shortcut symbol **$** can only be used in direct mode.

Examples: Using **CATALOG**

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

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

Below is an example showing how a directory looks with the **w**ide parameter:

```
DIR W
0 "BASIC EXAMPLES  "
1 "BEGIN"          P    1 "FREAD"          P    2 "PAINT.COR"      P
1 "BEND"           P    1 "FRE"            P    3 "PALETTE.COR"    P
1 "BUMP"           P    2 "GET#"           P    1 "PEEK"           P
1 "CHAR"           P    1 "GETKEY"         P    3 "PEN"            P
1 "CHR$"           P    1 "GET"            P    1 "PLAY"           P
4 "CIRCLE"         P    2 "GOSUB"          P    2 "POINTER"        P
1 "CLOSE"          P    2 "GOTO.COR"       P    1 "POKE"           P
1 "CLR"            P    2 "GRAPHIC"        P    1 "POS"            P
2 "COLLISION"      P    1 "HELP"           P    1 "POT"            P
1 "CURSOR"         P    1 "IF"             P    1 "PRINT#"         P
0 "DATA BASE"      R    2 "INPUT#"         P    1 "PRINT"          P
1 "DATA"           P    2 "INPUT"          P    1 "RCOLOR.COR"     P
1 "DEF FN"         P    2 "JOY"            P    1 "READ"           P
1 "DIM"            P    1 "LINE INPUT#"    P    1 "RECORD"         P
1 "DO"             P    3 "LINE"           P    1 "REM"            P
5 "ELLIPSE"        P    1 "LOOP"           P    1 "RESTORE"        P
1 "ELSE"           P    1 "MID$"           P    1 "RESUME"         P
1 "EL"             P    1 "MOD"            P    1 "RETURN"         P
1 "ENVELOPE"       P    1 "MOVSPR"         P    1 "REVERS"         S
2 "EXIT"           P    1 "NEXT"           P    3 "RGRAPHIC"       P
1 "FOR"            P    2 "ON"             P    1 "RMOUSE"         P
```

CHANGE

Token: $FE $2C

Format: **CHANGE /**findstring**/ TO /**replacestring**/** [**,** line range]
CHANGE "findstring**" TO "**replacestring**"** [**,** line range]

Usage: Edits the BASIC program that is currently in memory to replace all instances of one string with another.

An optional **line range** limits the search to this range, otherwise the entire BASIC program is searched. At each occurrence of the **findstring**, the line is listed and the user is prompted for an action:

- Y RETURN perform the replace and find the next string
- N RETURN do **not** perform the replace and find the next string
- * RETURN replace the current and all following matches
- RETURN exit the command, and don't replace the current match

Remarks: Almost any character that is not part of the string, including letters and punctuation, can be used instead of **/**.

However, using the double quote character finds text strings that are not tokenised, and therefore not part of a keyword.
For example, `CHANGE "LOOP" TO "OOPS"` will not find the BASIC keyword `LOOP`, because the keyword is stored as a token and not as text. However `CHANGE /LOOP/ TO /OOPS/` will find and replace it (possibly causing `SYNTAX ERROR`s).

Due to a limitation of the BASIC parser, **CHANGE** is unable to match the **REM** and **DATA** keywords. See **FIND**.

Can only be used in direct mode.

Examples: Using **CHANGE**

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

CHAR

Token: $E0

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

Usage: Bitmap graphics: displays text on a graphic screen.

column (in units of character positions) is the start position of the output horizontally. As each column unit is 8 pixels wide, a screen width of 320 has a column range of 0 – 39, while a screen width of 640 has a column range of 0 – 79.

row (in pixel units) is the start position of the output vertically. In contrast to the column parameter, its unit is in pixels (not character positions), with the top row having the value of 0.

height is a factor applied to the vertical size of the characters, where 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, where 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 to the default character set at $29800, which includes upper and lower case characters.

Three character sets (see also **FONT**) are available:

- **$29000** Font A (ASCII)
- **$3D000** Font B (Bold)
- **$2D000** Font C (CBM)

The first part of the font (upper case / graphics) is stored at $xx000 – $xx7FF.

The second part of the font (lower case / upper case) is stored at $xx800 – $xxFFF.

string is a string constant or expression which will be printed. This string may optionally contain one or more of the following control characters:

Expression	Keyboard Shortcut	Description
`CHR$(2)`	CTRL+B	Blank Cell
`CHR$(6)`	CTRL+F	Flip Character
`CHR$(9)`	CTRL+I	AND With Screen
`CHR$(15)`	CTRL+O	OR With Screen
`CHR$(24)`	CTRL+X	XOR With Screen
`CHR$(18)`	RVSON	Reverse
`CHR$(146)`	RVSOFF	Reverse Off
`CHR$(147)`	CLR	Clear Viewport
`CHR$(21)`	CTRL+U	Underline
`CHR$(25)+"-"`	CTRL+Y + "-"	Rotate Left
`CHR$(25)+"+"`	CTRL+Y + "+"	Rotate Right
`CHR$(26)`	CTRL+Z	Mirror
`CHR$(157)`	Cursor Left	Move Left
`CHR$(29)`	Cursor Right	Move Right
`CHR$(145)`	Cursor Up	Move Up
`CHR$(17)`	Cursor Down	Move Down

Notice that the start position of the string has different units in the horizontal and vertical directions. Horizontal is in columns and vertical is in pixels.

Refer to the **CHR$** function on page 47 for more information.

Examples: Using **CHAR**

```
10 SCREEN 640,400,2
20 CHAR 28,180,4,4,2,"MEGA65",$29000
30 GETKEY A$
40 SCREEN CLOSE
```

Will print the text "MEGA65" at the centre of a 640 x 400 graphic screen.

CHARDEF

Token: $E0 $96

Format: **CHARDEF** index, bit-matrix

Usage: Changes the appearance of a character.

index is the screen code of the character to change (@:0, A:1, B:2, ...). See appendix 3 on page 277 for a list of screen codes.

bit-matrix is a set of 8 byte values, which define the raster representation for the character from top row to bottom row. If more than 8 values are used as arguments, the values 9 – 16 are used for the character index+1, 17 – 24 for index+2, etc.

Remarks: The character bitmap changes are applied to the VIC character generator, which resides in RAM at the address $FF7E000.

All changes are volatile and the VIC character set can be restored by a reset or by using the **FONT** command.

Examples: Using **CHARDEF**

```
CHARDEF 1,$FF,$81,$81,$81,$81,$81,$81,$FF :REM CHANGE 'A' TO RECTANGLE
CHARDEF 9,$18,$18,$18,$18,$18,$18,$18,$00 :REM MAKE 'I' SANS SERIF
```

CHDIR

Token: $FE $4B

Format: **CHDIR** dirname [**,U** unit]

Usage: Changes the current working directory.

dirname the name of a directory. Either a quoted string such as "SOMEDIR", or a string expression in brackets such as (DR$).

Dependent on the **unit**, **CHDIR** is applied to different filesystems.

UNIT 12 is reserved for the SD-Card (FAT filesystem). This command can be used to navigate to subdirectories and mount disk images that are stored there. **CHDIR "..",U12** changes to the parent directory on **UNIT 12**.

For other units managed by CBDOS (typically 8 and 9), **CHDIR** is used to change into or out of subdirectories on floppy or disk image of type **D81**. Existing subdirectories are displayed as filetype **CBM** in the parent directory, they are created with the command **MKDIR**. **CHDIR "/",U unit** changes to the root directory.

Examples: Using **CHDIR**

```
CHDIR "ADVENTURES",U12  :REM ENTER ADVENTURES ON SD CARD
CHDIR "..",U12          :REM GO BACK TO PARENT DIRECTORY
CHDIR "RACING",U12      :REM ENTER SUBDIRECTORY RACING
0 "MEGA65          "    1D
800  "MEGA65 GAMES"     CBM
800  "MEGA65 TOOLS"     CBM
600  "BASIC PROGRAMS"   CBM
960  BLOCKS FREE.

CHDIR "MEGA65 GAMES",U8 :REM ENTER SUBDIRECTORY ON FLOPPY DISK
CHDIR "/",U8            :REM GO BACK TO ROOT DIRECTORY
```

CHR$

Token: $C1

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

Returns: A string containing one character of the given PETSCII value.

Remarks: The argument range is from 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**. The complete table of characters (and their PETSCII codes) is on page 281.

Example: Using **CHR$**

```
10 QUOTE$   = CHR$(34)
20 ESCAPE$  = CHR$(27)
30 PRINT QUOTE$;"MEGA65";QUOTE$ : REM PRINT "MEGA65"
40 PRINT ESCAPE$;"Q";        : REM CLEAR TO END OF LINE
```

CIRCLE

Token: $E2

Format: **CIRCLE** xc, yc, radius [, flags , start, stop]

Usage: Bitmap graphics: draws a circle.

This is a special case of **ELLIPSE**, using the same value for horizontal and vertical radius.

xc the x coordinate of the centre in pixels

yc the y coordinate of the centre in pixels

radius the radius of the circle in pixels

flags controls filling, arcs and the position of the 0 degree angle. Default setting (zero) is don't fill, draw legs and the 0 degree radian points to 3 o' clock.

Bit	Name	Value	Action if set
0	fill	1	Fill circle or arc with the current pen colour
1	legs	2	Suppress drawing of the legs of an arc
2	combs	4	Let the zero radian point to 12 o' clock

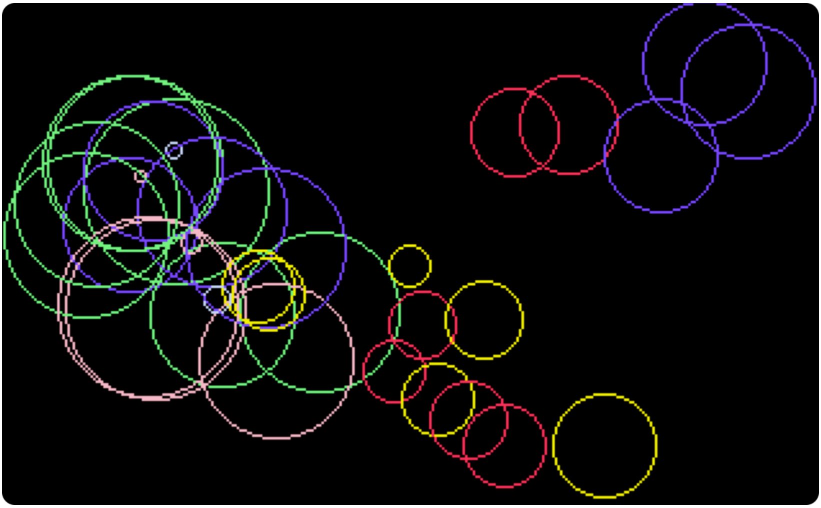
The units for the start- and stop-angle are degrees in the range of 0 to 360. The 0 radian starts at 3 o' clock and moves clockwise. Setting bit 2 of flags (value 4) moves the zero-radian to the 12 o' clock position.

start start angle for drawing an arc

stop stop angle for drawing an arc

Remarks: **CIRCLE** is used to draw circles on screens with an aspect ratio of 1:1 (for example: 320 x 200 or 640 x 400). Whilst using other resolutions (such as 640 x 200), the shape will be an ellipse instead.

The example program uses the random number function **RND** for circle colour, size and position. So it shows a different picture for each run.



Example: Using **CIRCLE**

```
100 REM CIRCLE (AFTER F.BOWEN)
110 BORDER 0                          :REM BLACK
120 SCREEN 320,200,4                  :REM SIMPLE SCREEN SETUP
130 PALETTE 0,0,0,0,0                 :REM BLACK
140 PALETTE 0,1,RND(.)*16,RND(.)*16,15  :REM RANDOM COLOURS
150 PALETTE 0,2,RND(.)*16,15,RND(.)*16
160 PALETTE 0,3,15,RND(.)*16,RND(.)*16
170 PALETTE 0,4,RND(.)*16,RND(.)*16,15
180 PALETTE 0,5,RND(.)*16,15,RND(.)*16
190 PALETTE 0,6,15,RND(.)*16,RND(.)*16
200 SCNCLR 0                          :REM CLEAR
210 FORI=0TO32                        :REM CIRCLE LOOP
220 PEN 0,RND(.)*6+1                  :REM RANDOM PEN
230 R=RND(.)*36+1                     :REM RADIUS
240 XC=R+RND(.)*320:IF(XC+R)>319THEN240:REM X CENTRE
250 YC=R+RND(.)*200:IF(YC+R)>199THEN250:REM Y CENTRE
260 CIRCLE XC,YC,R,.                  :REM DRAW
270 NEXT
280 GETKEY A$                         :REM WAIT FOR KEY
290 SCREEN CLOSE:BORDER 6
```

CLOSE

Token: $A0

Format: **CLOSE** channel

Usage: Closes an input or output channel.

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

Remarks: Closing files that have previously been opened before a program has completed is very important, especially for output files. **CLOSE** flushes output buffers and updates the directory information on disks. Failing to **CLOSE** can corrupt files and disks. BASIC does *not* automatically close channels nor files when a 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**
CLR variable

Usage: Clears BASIC variable memory.

After executing **CLR**, all variables and arrays will be undeclared. The run-time stack pointers and the table of open channels are also reset. **RUN** performs **CLR** automatically.

CLR variable clears (zeroes) the variable. **variable** can be a numeric variable or a string variable, but not an array.

Remarks: **CLR** should not be used inside loops or subroutines, as it destroys the return address. After **CLR**, all variables are unknown and will be initialised when they are next used.

Example: Using **CLR**

```
10 A=5: P$="MEGA65"
20 CLR
30 PRINT A;P$
RUN

 0
```

CLRBIT

Token: $9C $FE $4E

Format: **CLRBIT** address, bit number

Usage: Clears (resets) a single bit at the **address**.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

The **bit number** is a value in the range of 0 - 7.

Remarks: **CLRBIT** is a short version of using a bitwise **AND** to clear a bit, but you can only clear one bit at a time. Refer to **SETBIT** to set a bit instead.

Example: Using **CLRBIT**

```
10 BANK 128          :REM SELECT SYSTEM MAPPING
20 CLRBIT $D011,4    :REM DISABLE DISPLAY
30 CLRBIT $D016,3    :REM SWITCH TO 38 OR 76 COLUMN MODE
```

CMD

Token: $9D

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

Usage: Redirects the standard output from screen to a channel.

This enables you to print listings and directories to other output channels. It is also possible to redirect this output to a disk file, or a modem.

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

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

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

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

```
OPEN 1,4    :REM OPEN CHANNEL #1 TO PRINTER AT UNIT 4
CMD 1
LIST
PRINT#1
CLOSE 1
```

COLLECT

Token: $F3

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

Usage: Rebuilds the Block Availability Map (BAM) of a disk, deleting splat files (files which have been opened, but not properly closed) 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 such as the 1541, 1571, or 1581.

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

Remarks: While this command is useful for cleaning a disk from splat files, 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 and may be overwritten by further disk write operations.

Examples: Using **COLLECT**

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

COLLISION

Token: $FE $17

Format: **COLLISION** type [, line number]

Usage: Enables or disables a user-programmed interrupt handler for sprite collision.

With a handler enabled, a sprite collision of the given **type** interrupts the BASIC program and performs a **GOSUB** to **line number**. This handler must give control back with **RETURN**.

type the collision type for this interrupt handler:

Type	Description
1	Sprite - Sprite Collision
2	Sprite - Data - Collision
3	Light Pen

line number the line number of a subroutine which handles the sprite collision and ends with **RETURN**

A call without the line number argument disables the handler.

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

Info: **COLLISION** wasn't completed in BASIC 10. It is available in BASIC 65.

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
60 END
70 REM SPRITE <-> SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

COLOR

Token: $E7

Format: **COLOR** colour

Usage: Sets the foreground text colour for subsequent **PRINT** commands.

colour the palette entry number, in the range 0 - 31

See appendix 6 on page 297 for the list of colours in the default system palette.

Remarks: This is another name for **FOREGROUND**.

Example: Using **COLOR**

```
COLOR 2
PRINT "THIS IS RED"
COLOR 3
PRINT "THIS IS CYAN"
```

CONCAT

Token: $FE $13

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

Usage: Appends (concatenates) the contents of the file **appendfile** to the file **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 brackets, for example: `(FI$)`

targetfile is either a quoted string, for example: `"SAFE"` or a string expression in brackets, for example: `(FS$)`

If the disk unit has dual drives, it is possible to apply **CONCAT** to files which are stored on different disks. In this case, it is necessary to specify the drive# for both files. This is also necessary 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 such as the 1541, 1571, or 1581.

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

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

Examples: Using **CONCAT**

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

CONT

Token: $9A

Format: **CONT**

Usage: Resumes program execution after a break or stop caused by an **END** or **STOP** statement, or by pressing RUN STOP.

This is a useful debugging tool. The BASIC program may be stopped and variables can be examined, and even changed. The **CONT** statement resumes execution.

Remarks: **CONT** cannot be used if a program has stopped because of an error. Also, any editing of a program inhibits continuation. Stopping and continuation can spoil the screen output, and can also 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: $F4

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

Usage: Copies a file to another file, or one or more files from one disk to another.

source is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (FI$).

target is either a quoted string, e.g. "BACKUP" or a string expression in brackets, e.g. (FS$)

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

If none or one unit number is given, or the unit numbers before and after the TO token are equal, **COPY** is executed on the disk drive itself, and the source and target files will be on the same disk.

If the source unit (before TO) is different to the target unit (after TO), **COPY** executes a CPU-driven routine that reads the source files into a RAM buffer and writes to the target unit. In this case, the target file name cannot be chosen, it will be the same as the source filename. The extended unit-to-unit copy mode allows the copying of single files, pattern matching files or all files of a disk. Any combination of units is allowed, internal floppy, D81 disk images, IEC floppy drives such as the 1541, 1571, 1581, or CMD floppy and hard drives.

Remarks: The file types **PRG**, **SEQ** and **USR** can be copied. If source and target are on the same disk, the target filename must be different to the source file name.

COPY cannot copy **DEL** files, which are commonly used as titles or separators in disk directories. These do not conform to Commodore DOS rules and cannot be accessed by standard **OPEN** routines.

REL files cannot be copied from unit to unit.

Examples: Using **COPY**

```
COPY U8 TO U9            :REM COPY ALL FILES
COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
COPY "*.TXT",U8 TO U9    :REM PATTERN COPY
COPY "M*",U9 TO U11      :REM PATTERN COPY
```

COS

Token: $BE

Format: **COS**(numeric expression)

Returns: The cosine of an angle.

The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: A value in units of **degrees** can be converted to **radians** by multiplying it with $\pi/180$.

Examples: Using **COS**

```
PRINT COS(0.7)
 0.76484219

X=60:PRINT COS(X * π / 180)
 0.5
```

CURSOR

Format: **CURSOR** <**ON** | **OFF**> [{**,** column**,** row**,** style}]
CURSOR column**,** row

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

ON or **OFF** displays or hides the cursor. When the cursor is **ON**, it will appear at the cursor position during **GETKEY**.

column and **row** specify the new position.

style sets a solid (1) or flashing (0) cursor.

Example: Using **CURSOR**

```
10  SCNCLR
20  CURSOR 1,2
30  PRINT "A"; : SLEEP 1
40  PRINT "B"; : SLEEP 1
50  PRINT "C"; : SLEEP 1
60  CURSOR 20,10
70  PRINT "D"; : SLEEP 1
80  CURSOR ,5              :REM MOVE THE CURSOR TO ROW 5 BUT DO NOT CHANGE THE COLUMN
90  PRINT "E"; : SLEEP 1
100 CURSOR 0               :REM MOVE THE CURSOR TO THE START OF THE ROW
110 PRINT "F"; : SLEEP 1
```

CUT

Token: $E4

Format: **CUT** x, y, width, height

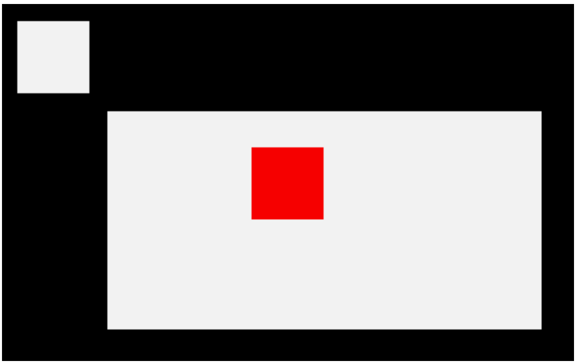
Usage: Bitmap graphics: copies the content of the specified rectangle with upper left position **x, y** and the **width** and **height** to a buffer, and fills the region afterwards with the colour of the currently selected pen.

The cut out can be inserted at any position with the command **PASTE**.

Remarks: The size of the rectangle is limited by the 1K size of the buffer. The memory requirement for a cut out region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using **CUT**

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1  :REM DRAW A WHITE BOX
30 PEN 2                :REM SELECT RED PEN
40 CUT  140,80,40,40    :REM CUT OUT A 40 * 40 REGION
50 PASTE 10,10,40,40    :REM PASTE IT TO NEW POSITION
60 GETKEY A$            :REM WAIT FOR KEYPRESS
70 SCREEN CLOSE
```



DATA

Token: $83

Format: **DATA** [constant [, constant ...]]

Usage: Defines constants which can be read by **READ** statements in a program.

Numbers and strings are allowed, but expressions are not. Items are separated by commas. Strings containing commas, colons or spaces must be placed in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility 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.

RESTORE may be used to set the data pointer to a specific line for subsequent reads.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

Example: Using **DATA**

```
1 REM 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 GAUSSLEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
60 END
80 DATA "MEGA65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252

RUN
PROGRAM:MEGA65  VERSION: 1.1
N-POINT GAUSSLEGENDRE FACTORS E1
 2  0.512
 3  0.3573
 4  0.276
 5  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 such as the 1541, 1571, or 1581.

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

The DOS of the disk drive will close all open files, clear all channels, free buffers and re-read the BAM. All open channels on the computer will also be closed.

Examples: Using **DCLEAR**

```
DCLEAR
DCLEAR U9
DCLEAR D0, U9
```

DCLOSE

Token: $FE $0F

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

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

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

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

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

Remarks: It is important to close all open files before a program ends. Otherwise buffers will not be freed and even worse, open files that have been written to may be incomplete (commonly called splat files), and no longer usable.

Examples: 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**)**

Returns: The decimal value of a hexadecimal string.

The argument range is “0” to “FFFFFFFF”. **DEC()** ignores everything after the first non-hex digit or the eighth character.

Remarks: Allowed digits in uppercase/graphics mode are 0 – 9 and A – F (0123456789ABCDEF) and in lowercase/uppercase mode are 0 – 9 and a – f (0123456789abcdef).

Example: Using **DEC**

```
PRINT DEC("D000")
53248

POKE DEC("600"),255
```

DECBIN

Token: $CE $11

Format: **DECBIN**(string expression)

Returns: The decimal value of a binary string.

The argument range is "0" to "11111111111111111111111111111111". **DECBIN()** ignores everything after the first non-binary digit or the 32nd character.

Example: Using **DECBIN**

```
PRINT DECBIN("1101000000000000")
53248
```

DEF FN

Token: $96

Format: **DEF FN** name(real variable) = [expression]

Usage: Defines a single statement user function with one argument of type real, that returns a real value when evaluated.

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 when the function is used.

Remarks: The function argument is not a real variable and will not overwrite a variable with that name. It only represents the argument value within the function definition.

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: The first form deletes a range of lines from the BASIC program. The second form deletes one or more files from a disk.

line range consists of the first and 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 brackets, for example: `(FS$)`

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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: **DELETE filename** is a synonym of **SCRATCH filename** and **ERASE filename**.

Examples: 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
DELETE "*=SEQ"  :REM DELETE ALL SEQUENTIAL FILES
DELETE "R*=PRG" :REM DELETE PROGRAM FILES STARTING WITH 'R'
```

DIM

Token: $86

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

Usage: Declares the shape, 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 as declared. The rules for variable names apply for array names as well. You can create byte arrays, 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: Byte arrays consume one byte per element, integer arrays two bytes, real arrays five bytes and string arrays three bytes for the string descriptor plus the length of the string itself.
If an array identifier is used without being previously declared, an implicit declaration of an one dimensional array with limit of 10 is performed.

Example: Using **DIM**

```
1 REM DIM
10 DIM A%(8)    : REM ARRAY OF 9 ELEMEMTS
20 DIM XX(2,3)  : REM ARRAY OF 3X4 = 12 ELEMENTS
30 FOR I=0 TO 8 : A%(I)=PEEK(256+I) : PRINT A%(I);: NEXT:PRINT
40 FOR I=0 TO 2 : FOR J=0 TO 3 : READ XX(I,J):PRINT XX(I,J);: NEXT J,I
50 END
60 DATA 1,-2,3,-4,5,-6,7,-8,9,-10,11,-12

RUN
 45  52  50  0  0  0  0  0  0
 1 -2  3 -4  5 -6  7 -8  9 -10  11 -12
```

DIR

Token: $EE (DIR) $FE $29 (ECTORY)

Format: **DIR** [filepattern] [**,W**] [**,P**] [**,R**] [**,D** drive] [**,U** unit]
DIRECTORY [filepattern] [**,W**] [**,P**] [**,R**] [**,D** drive] [**,U** unit]
$ [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]
DIR U12 [**,P**]

Usage: Prints a file directory/catalog of the specified disk.

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

The **P** (Pagination) parameter lists the directory one column wide, and pauses for each screenful of output. Press the Q key to interrupt the listing at the current page. Press any other key to display the next page.

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 brackets, e.g. (DI$)

The **U12** argument lists the contents of the SD card. It can be used with the **P** argument for a paginated display. It does not support other arguments.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **DIR** is a synonym of **CATALOG** and **DIRECTORY**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and **?** may be used. Adding **,T=** to the pattern string, with **T** specifying a filetype of **P**, **S**, **U** or **R** (for **P**RG, **S**EQ, **U**SR, **R**EL) filters the output to that filetype.

The shortcut symbol **$** can only be used in direct mode.

Examples: Using **DIR**

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

For a **DIR** listing with the **w**ide parameter, please refer to the example under **CATALOG** on page 41.

DISK

Token: $FE $40

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

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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.

Using **DISK** with no parameters prints the disk status.

The shortcut key @ can only be used in direct mode.

Examples: Using **DISK**

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

DLOAD

Token: $F0

Format: **DLOAD** filename [**,D** drive] [**,U** unit]
DLOAD "$[pattern=type]" [**,D** drive] [**,U** unit]
DLOAD "$$[pattern=type]" [**,D** drive] [**,U** unit]

Usage: The first form loads a file of type PRG into memory reserved for BASIC programs.

The second form loads a directory into memory, which can then be viewed with **LIST**. It is structured like a BASIC program, but file sizes are displayed instead of line numbers.

The third form is similar to the second one, but the files are numbered. This listing can be scrolled like a BASIC program with the keys **F9** or **F11**, edited, listed, saved or printed.

A filter can be applied by specifying a pattern or a pattern and a type. The asterisk matches the rest of the name, while the ? matches any single character. The type specifier can be a character of (P,S,U,R), that is Program, Sequential, User, or Relative file.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: The load address that is stored in the first two bytes of the PRG file is ignored. The program is always loaded into BASIC memory. This enables loading of BASIC programs that were saved on other computers with different memory configurations. After loading, the program is re-linked and ready to be **RUN** or edited.

It is possible to use **DLOAD** in a running program. This is called overlaying, or chaining. If you do this, then the newly 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 also be used by the newly loaded program.

Every **DLOAD**, of either a program or a directory listing, will replace a program that is currently in memory.

Examples: Using **DLOAD**

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

DLOAD "$"           :REM LOAD WHOLE DIRECTORY - WITH FILE SIZES
DLOAD "$$"          :REM LOAD WHOLE DIRECTORY - SCROLLABLE
DLOAD "$$X*=P"      :REM DIRECTOY WITH PRG FILES STARTING with 'X'
```

DMA

Token: $FE $1F

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

Usage: **DMA** ("Direct Memory Access") is obsolete, and has been replaced by **EDMA**.

command The lower two bits control the function: 0: copy, 1: mix, 2: swap, 3: fill. Note that only copy and fill are implemented in the MEGA65 DMAcontroller at the time of writing. Other DMAgic command bits can also be set, for example, to allow copying data in the reverse direction, or holding the source or destination address.

length number of bytes (in the range 0 to 65535). NOTE: Specifying a length of 0 will be interpreted as a length of 65536 (exactly 64 kilo-bytes).

source address 16-bit address of read area or fill byte

source bank bank number for source (ignored for fill mode)

target 16-bit address of write area

target bank bank number for target

sub sub command

Remarks: **DMA** has access to the lower 1MB address range organised in 16 banks of 64 K. To avoid this limitation, use **EDMA**, which has access to the full 256MB address range.

Examples: A sequence of **DMA** calls to demonstrate fast screen drawing operations

```
DMA 0, 80*25, 2048, 0, 0, 4  :REM SAVE SCREEN TO $00000 BANK 4
DMA 3, 80*25, 32, 0, 2048, 0 :REM FILL SCREEN WITH BLANKS
DMA 0, 80*25, 0, 4, 2048, 0  :REM RESTORE SCREEN FROM $00000 BANK 4
DMA 2, 80, 2048, 0, 2048+80, 0 :REM SWAP CONTENTS OF LINE 1 & 2 OF SCREEN
```

DMODE

Token: $FE $35

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

Usage: Bitmap graphics: sets "display mode" parameters of the graphics context, which is used by drawing commands.

Mode	Values
jam	0 - 1
complement	0 - 1
stencil	0 - 1
style	0 - 3
thick	1 - 8

DO

Token: $EB

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

Usage: **DO** and **LOOP** define the start of a BASIC loop.

Using **DO** and **LOOP** alone without any modifiers creates an infinite loop, which can only be exited by the **EXIT** statement. The loop can be controlled by adding **UNTIL** or **WHILE** after the **DO** or **LOOP**.

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

Examples: 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#** channel, filename [**,L** [reclen]] [**,W**] [**,D** drive] [**,U** unit]

Usage: Opens a file for reading or writing.

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 255** line terminator is CR LF.

L indicates, that the file is a relative file, which is opened for read/write, as well as random access.

The **reclen** record length is mandatory for creating relative files. For existing relative files, **reclen** is used as a safety check, if given.

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

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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 **PRG** files or ",U" for **USR** files.

If the first character of the filename is an at sign '`@`', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

Examples: Using **DOPEN**

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

DOT

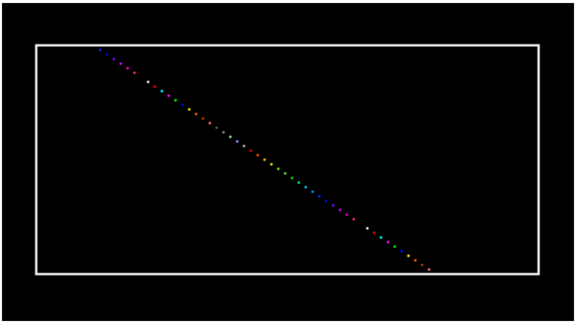
Token: $FE $4C

Format: **DOT** x, y [,colour]

Usage: Bitmap graphics: draws a pixel at screen coordinates x and y. The optional third parameter defines the colour to be used. If not specified, the current pen colour will be used.

Example: Using **DOT**:

```
10 SCREEN 320,200,5
20 BOX 50,50,270,150
30 VIEWPORT 50,50,220,100
40 FORI=0TO127
50 DOT I+I+I,I+I,I
60 NEXT
70 GETKEY A
80 SCREEN CLOSE
```



DPAT

Token: $FE $36

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

Usage: Bitmap graphics: sets the drawing pattern of the graphics context for drawing commands.

There a four predefined pattern types, that can be selected by specifying the type number (1, 2, 3, or 4) as a single parameter.

A value of zero for the type number indicates a user defined pattern. This pattern can be set by using a bit string that consists of either 8, 16, 24, or 32 bits. The number of used pattern bytes is given as the second parameter. It defines how many pattern bytes (1, 2, 3, or 4) follow.

- **Type** 0 - 4
- **Number** number of following pattern bytes (1 - 4)
- **Pattern** pattern bytes

DS

Format: **DS**

Usage: The status of the last disk operation.

This is a volatile variable. Each use triggers the reading of the disk status from the current disk device in usage.

DS is coupled to the string variable **DS$** which is updated at the same time.

Reading the disk status from a disk device automatically clears any error status on that device, so subsequent reads will return 0, if no other activity has since occurred.

Remarks: **DS** is a reserved system variable.

Example: Using **DS**

```
100 DOPEN#1,"DATA"
110 IF DS<>0 THEN PRINT"COULD NOT OPEN FILE DATA":STOP
```

DS$

Format: **DS$**

Usage: The status of the last disk operation in text form of the format: Code,Message,Track,Sector.

DS$ is coupled to the numeric variable **DS**. It is updated when **DS** is used. DS$ is set to `00,OK,00,00` if there was no error, otherwise it is set to a DOS error message (listed in the disk drive manuals).

Remarks: **DS$** is a reserved system variable.

Example: Using **DS$**

```
100 DOPEN#1,"DATA"
110 IF DS<>0 THEN PRINT DS$:STOP
```

DSAVE

Token: $EF

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

Usage: Saves the BASIC program in memory to a file of type **PRG**.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$). 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 not recommended to use this option on 1541 and 1571 drives, as they contain a "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 such as the 1541, 1571, or 1581.

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

Remarks: **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
```

DT$

Format: **DT$**

Usage: The current date, as a string.

The date value is updated from RTC (Real-Time Clock). The string **DT$** is formatted as: "DD-MON-YYYY", for example: "04-APR-2021".

Remarks: **DT$** is a reserved system variable. For more information on how to set the Real-Time Clock, refer to *the MEGA65 Book*, The Configuration Utility (section 4).

Example: Using **DT$**

```
100 PRINT "TODAY IS: ";DT$
```

DVERIFY

Token: $FE $14

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

Usage: Verifies that the BASIC program in memory is equivalent to a file of type **PRG**.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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. **DVERIFY** exits either with the message OK or with VERIFY ERROR.

Example: Using **DVERIFY**

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

EDIT

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

Usage: Enables or disables the text editing mode of the screen editor.

EDIT ON enables text editing mode. In this mode, you can create, edit, save, and load files of type SEQ as text files using the same line editor that you use to write BASIC programs. In this mode:

- The prompt appears as OK, instead of READY.
- The editor does no tokenising/parsing. All text entered after a linenumber remains pure text, BASIC keywords such as **FOR** and **GOTO** are not converted to BASIC tokens, as they are whilst in program mode.
- The line numbers are only used for text organisation, sorting, deleting, listing, etc.
- When the text is saved to file with **DSAVE**, a sequential file (type **SEQ**) is written, not a program (**PRG**) file. Line numbers are *not* written to the file.
- **DLOAD** in text mode can load only sequential files. Line numbers are automatically generated for editing purposes.
- Text mode applies to lines entered with line numbers only. Lines with no line number are executed as BASIC commands, as usual.

EDIT OFF disables text editing mode and returns to BASIC program editing mode. The MEGA65 starts in BASIC program editing mode.

Sequential files created with the text editor can be displayed (without loading them) on the screen by using **TYPE <filename>**.

Example: Using **EDIT**

```
ready.
edit on

ok.
100 This is a simple text editor.
dsave "example"

ok.
new

ok.
catalog

0 "demoempty      " 00 3d
1 "example"         seq
3159 blocks free

ok.
type "example"
This is a simple text editor.

ok.
dload "example"

loading

ok.
list

1000 This is a simple text editor.

ok.
```

EDMA

Token: $FE $21

Format: **EDMA** command**,** length**,** source**,** target

Usage: Copies or updates a large amount of memory quickly.

EDMA ("Extended Direct Memory Access") is the fastest method to manipulate memory areas using the DMA controller. Please refer to *the MEGA65 Book*, F018-Compatible Direct Memory Access (DMA) Controller (Appendix O) for more details on EDMA.

command 0: copy, 1: mix, 2: swap, 3: fill.

Because this two bits of the command share the same register with other bits you can for example use bit 5 to reverse loop operation. This is also working in overlapping memory regions for source and target. Please see the example below.

length number of bytes (in the range 0 to 65535). NOTE: Specifying a length of 0 will be interpreted as a length of 65536 (exactly 64 kilobytes).

source 28-bit address of read area or fill byte.

target 28-bit address of write area.

Remarks: **EDMA** can access the entire 256MB address range, using up to 28 bits for the addresses of the source and target.

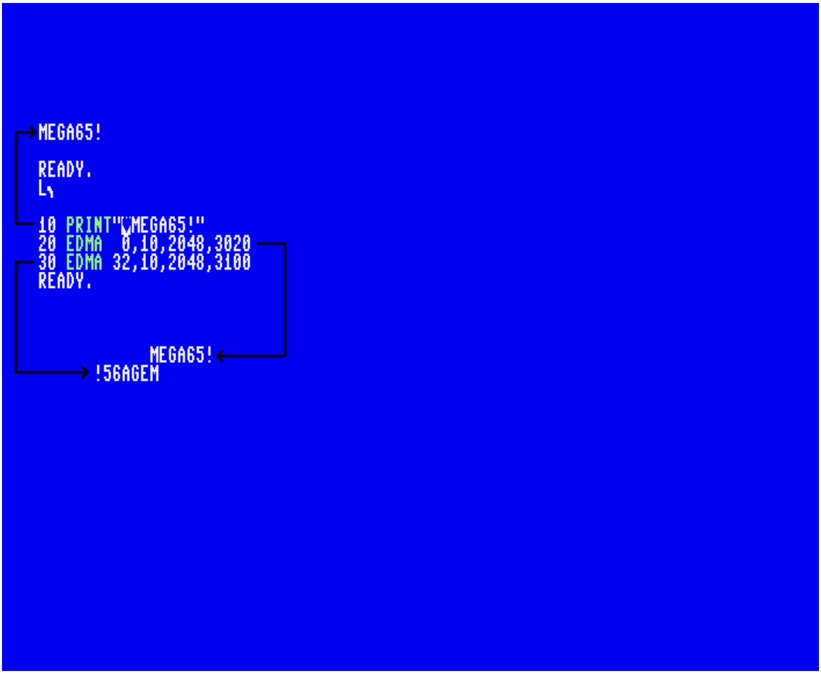
Examples: Using **EDMA**

```
EDMA 0, $800, $F700, $8000000 :REM COPY SCALAR VARIABLES TO ATTIC RAM
EDMA 3, 80*25, 32, 2048       :REM FILL SCREEN WITH BLANKS
EDMA 0, 80*25, 2048, $8000800 :REM COPY SCREEN TO ATTIC RAM
```

By adding 32 (bit 5) to the command parameter, the DMA operation can be performed in reverse order:

```
10 PRINT "♥MEGA65!"
20 EDMA  0,10,2048,3020  : REM 2048 IS BEGINNING OF SCREEN RAM
30 EDMA 32,10,2048,3100  : REM 3020 AND 3100 ARE THE LOWER PART OF THE SCREEN
```

Listing and output of the last example:



EL

Format: **EL**

Usage: The line number where the most recent BASIC error occurred, or the value -1 if there was no error.

Remarks: **EL** is a reserved system variable.

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

Example: Using **EL**

```
 10 TRAP 100
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

ELLIPSE

Token: $FE $30

Format: **ELLIPSE** xc, yc, xr, yr [, flags , start, stop]

Usage: Bitmap graphics: draws an ellipse.

xc is the x coordinate of the centre in pixels

yc is the y coordinate of the centre in pixels

xr is the x radius of the ellipse in pixels

yr is the y radius of the ellipse in pixels

flags control filling, arcs and orientation of the zero radian (combs flag named after **retroCombs**). Default setting (zero) is: Don't fill, draw legs, start drawing at 3 'o clock.

Bit	Name	Value	Action if set
0	fill	1	Fill ellipse or arc with the current pen colour
1	legs	2	Suppress drawing of the legs of an arc
2	combs	4	Drawing (0 degree) starts at 12 'o clock

The units for the start- and stop-angle are degrees in the range of 0 to 360. The 0 radian starts at 3 o' clock and moves clockwise. The combs-flag shifts the 0 radian and the start position to the 12 'o clock position.

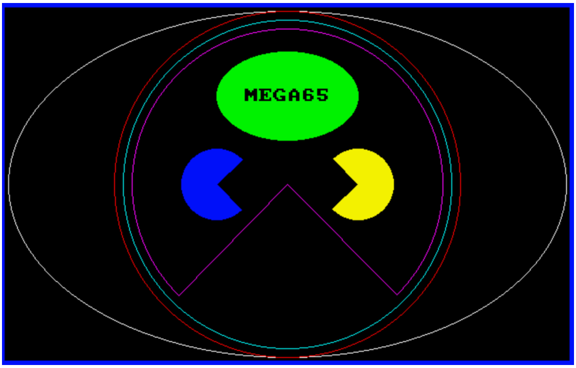
start start angle for drawing an elliptic arc.

stop stop angle for drawing an elliptic arc.

Remarks: **ELLIPSE** is used to draw ellipses on screens at various resolutions. If a full ellipse is to be drawn, start and stop should be either omissed or set both to zero (not 0 and 360). Drawing and filling of full ellipses is much faster, than using elliptic arcs.

Example: Using **ELLIPSE**

```
100 S%=2:D%=3:W%=320*S%:H%=200*S%  :REM SCREEN SETTINGS
110 CX%=W%/2:CY%=H%/2              :REM CENTRE AND RADII
120 RX%=W%/2:RY%=H%/2
130 SCREEN W%,H%,D%                :REM OPEN SCREEN
140 ELLIPSE CX%,CY%,CX%-4,CY%-4
150 PEN2:CIRCLE CX%,CY%,RY%-4,2
160 PEN3:CIRCLE CX%,CY%,RY%-14,2
170 PEN4:CIRCLE CX%,CY%,RY%-24,0,135,45
180 PEN5:ELLIPSE CX%,CY%/2,RX%/4,RY%/4,1
190 PEN6:CIRCLE 120*S%,CY%,40,1,45,315
200 PEN7:CIRCLE 200*S%,CY%,40,1,225,135
210 PEN0:CHAR 34,CY%/2-8,2,2,2,"MEGA65",$3D000
220 GETKEY A$                      :REM WAIT FOR ANY KEY
230 SCREEN CLOSE                   :REM CLOSE GRAPHICS SCREEN
```



ELSE

Token: $D5

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

Usage: **ELSE** is an optional part of an **IF** statement.

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

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

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

Remarks: There must be a colon before **ELSE**. There cannot be a colon or end-of-line after **ELSE**.

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

When the **true clause** does not use **BEGIN** and **BEND**, **ELSE** must be on the same line as **IF**.

Example: Using **ELSE**

```
100 REM ELSE
110 RED$=CHR$(28):BLACK$=CHR$(144):WHITE$=CHR$(5)
120 INPUT "ENTER A NUMBER";V
130 IF V<0 THENPRINT RED$;:ELSEPRINT BLACK$;
140 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
150 PRINT WHITE$
160 INPUT "END PROGRAM:(Y/N)";A$
170 IF A$="Y" THENEND
180 IF A$="N" THEN120:ELSE160
```

Using **ELSE** with **BEGIN** and **BEND**.

```
100 A = 0 : GOSUB 200
110 A = 1 : GOSUB 200
120 END
200 IF A = 0 THEN BEGIN
210 PRINT "HELLO"
220 BEND : ELSE BEGIN
230 PRINT "GOODBYE"
240 BEND
250 RETURN
```

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 nor close files. Variable definitions are still valid after **END**. The program may be continued with the **CONT** statement. After executing the last line of a 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: Sets the parameters for the synthesis of a musical instrument for use with **PLAY**.

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.

There are 10 slots for storing instrument parameters, preset with the following default 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
20 VOL 9,9
30 TEMPO 30
40 PLAY "T9O4Q CDEFGAB U3T8 CDEFGAB L","T5O3Q H CGEQG T7 HCGEQG L"
```

ER

Format: **ER**

Usage: The number of the most recent BASIC error that has occurred, or -1 if there was no error.

Remarks: **ER** is a reserved system variable.

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

Example: Using **ER**

```
 10 TRAP 100
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

ERASE

Token: $FE $2A

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

Usage: Erases (deletes) a disk file.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

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

Remarks: **ERASE filename** is a synonym of **SCRATCH filename** and **DELETE filename**.

In direct mode, the success and the number of erased files is printed. The second to last number from the message contains the number of successfully erased files.

Examples: Using **ERASE**

```
ERASE "DRM",U9 :REM ERASE FILE DRM ON UNIT 9
01, FILES SCRATCHED,01,00
ERASE "OLD*"   :REM ERASE ALL FILES BEGINNING WITH "OLD"
01, FILES SCRATCHED,04,00
ERASE "R*=PRG" :REM ERASE PROGRAM FILES STARTING WITH 'R'
01, FILES SCRATCHED,09,00
```

ERR$

Token: $D3

Format: **ERR$**(number)

Returns: The string description of a given BASIC error number.

number 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
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

EXIT

Token: $FD

Format: **EXIT**

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

Remarks: In nested loops, **EXIT** exits only the current loop, and continues execution in an outer loop (if there is one).

Example: Using **EXIT**

```
1 REM EXIT
10 OPEN 2,8,0,"$"           : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT 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,LO,HI            : REM FILE SIZE BYTES
45   S=LO + 256 * HI        : REM FILE SIZE
50   LINE INPUT#2, F$       : REM FILE NAME
55   PRINT S;F$             : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

EXP

Token: $BD

Format: **EXP(**numeric expression**)**

Returns: The value of the mathematical constant Euler's number (**2.71828183**) raised to the power of the argument.

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

Examples: Using **EXP**

```
PRINT EXP(1)
 2.71828183

PRINT EXP(0)
 1

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

FAST

Token: $FE $25

Format: **FAST** [speed]

Usage: Sets CPU clock speed to 1MHz, 3.5MHz or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.
- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **FAST** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

FAST is a synonym of **SPEED**.

FAST has no effect if `POKE 0,65` has previously been used to set the CPU to 40MHz.

Example: Using **FAST**

```
10 FAST     :REM SET SPEED TO MAXIMUM (40 MHZ)
20 FAST 1   :REM SET SPEED TO 1 MHZ
30 FAST 3   :REM SET SPEED TO 3.5 MHZ
40 FAST 3.5 :REM SET SPEED TO 3.5 MHZ
```

FGOSUB

Token: $FE $48

Format: **FGOSUB** numeric expression

Usage: Evaluates the given numeric expression, then calls (**GOSUB**s) the subroutine at the resulting line number.

Warning: Take care when using **RENUMBER** to change the line numbers of your program that any **FGOSUB** statements still use the intended numbers.

Example: Using **FGOSUB**:

```
 10 INPUT "WHICH SUBROUTINE TO EXECUTE 100,200,300";LI
 20 FGOSUB LI     :REM HOPEFULLY THIS LINE # EXISTS
 30 GOTO 10       :REM REPEAT
100 PRINT "AT LINE 100":RETURN
200 PRINT "AT LINE 200":RETURN
300 PRINT "AT LINE 300":RETURN
```

FGOTO

Token: $FE $47

Format: **FGOTO** numeric expression

Usage: Evaluates the given numeric expression, then jumps (**GO**es**TO**) to the resulting line number.

Warning: Take care when using **RENUMBER** to change the line numbers of your program that any **FGOTO** statements still use the intended numbers.

Example: Using **FGOTO**:

```
 10 INPUT "WHICH LINE # TO EXECUTE 100,200,300";LI
 20 FGOTO LI     :REM HOPEFULLY THIS LINE # EXISTS
 30 END
100 PRINT "AT LINE 100":END
200 PRINT "AT LINE 200":END
300 PRINT "AT LINE 300":END
```

FILTER

Token: $FE $03

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

Usage: Sets the parameters for a SID sound filter.

sid 1: right SID, 2: left SID

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 parameters 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**

```
10 PLAY "T7X103P9C"
15 SLEEP 0.02
20 PRINT "LOW PASS SWEEP" :L=1:B=0:H=0:GOSUB 100
30 PRINT "BAND PASS SWEEP":L=0:B=1:H=0:GOSUB 100
40 PRINT "HIGH PASS SWEEP":L=0:B=0:H=1:GOSUB 100
50 GOTO 20
100 REM *** SWEEP ***
110 FOR F = 50 TO 1950 STEP 50
120 IF F >= 1000 THEN FF = 2000-F : ELSE FF = F
130 FILTER 1,FF,L,B,H,15
140 PLAY "X1"
150 SLEEP 0.02
160 NEXT F
170 RETURN
```

FIND

Token: $FE $2B

Format: **FIND /**string**/** [**,** line range]
FIND "string**"** [**,** line range]

Usage: Searches the BASIC program that is currently in memory for all instances of a string.

It searches a given line range (if specified), otherwise the entire BASIC program is searched.

At each occurrence of the "find string" the line is listed with the string highlighted.

NO SCROLL can be used to pause the output.

Remarks: Almost any character that is not part of the string, including letters and punctuation, can be used instead of the slash **/**.

Using double quotes **"** as a delimiter has a special effect: The search text is not tokenised. **FIND "FOR"** will search for the three letters F, O, and R, not the BASIC keyword **FOR**. Therefore, it can find the word **FOR** in string constants or REM statements, but not in program code.

On the other hand, **FIND /FOR/** will find all occurrences of the BASIC keyword, but not the text "FOR" in strings.

Partial keywords cannot be searched. For example, **FIND /LOO/** will not find the keyword **LOOP**.

Due to how BASIC is parsed, finding the **REM** and **DATA** keywords requires using the colon as the delimiter: **FIND :REM TODO:** This does not work with the **CHANGE** command.

FIND is an editor command that can only be used in direct mode.

Example: Using **FIND**

```
READY.
LIST

10 REM PARROT COLOUR SCHEME
20 FONT B         :REM SERIF
30 FOREGROUND 5   :REM GREEN
40 BACKGROUND 0   :REM BLACK
50 HIGHLIGHT 4,0 :REM SYSTEM   PURPLE
60 HIGHLIGHT14,1 :REM REM      BLUE
70 HIGHLIGHT 7,2 :REM KEYWORD  YELLOW

READY.
FIND /OLO/

10 REM PARROT COLOUR SCHEME

READY.
FIND /HIGHLIGHT/

50 HIGHLIGHT 4,0 :REM SYSTEM   PURPLE
60 HIGHLIGHT14,1 :REM REM      BLUE
70 HIGHLIGHT 7,2 :REM KEYWORD  YELLOW

READY.
█
```

FN

Token: $A5

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

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

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
```

FONT

Token: $FE $46

Format: **FONT** <**A** | **B** | **C**>

Usage: Updates all characters to the given built-in font.

FONT A is the PETSCII font with several lowercase characters replaced with ASCII punctuation.

FONT B is an alternate appearance of **FONT A**.

FONT C is the PETSCII font. This is the default when the MEGA65 is first switched on.

This resets any changes made by the **CHARDEF** command.

The ASCII symbols of fonts **A** and **B** are typed by pressing the keys in the table below, some of which also require the holding down of the MEGA key. The codes for uppercase and lowercase are swapped compared to ASCII.

Code	Key	PETSCII	ASCII
$5C	Pound	£	\ (backslash)
$5E	Up Arrow (next to RESTORE)	↑	^ (caret)
$5F	Left Arrow (next to 1)	←	_ (underscore)
$7B	MEGA + Colon	┼	{ (open brace)
$7C	MEGA + Dot	▒	| (pipe)
$7D	MEGA + Semicolon	│	} (close brace)
$7E	MEGA + Comma	π	~ (tilde)

Remarks: The additional ASCII characters provided by FONT A and B are only available while using the lowercase character set.

Examples: Using **FONT**

```
FONT A :REM ASCII - ENABLE {|}_~^
FONT B :REM LIKE A, WITH A SERIF FONT
FONT C :REM COMMODORE FONT (DEFAULT)
```

FOR

Token: $81

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

Usage: **FOR** statements start a BASIC loop with an index variable.

index 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.

start is used to initialise the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the index variable at the end of an iteration. 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 than or equal to **start**, whereas for negative increments **end** must be less than or equal to **start**.

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

Examples: 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 text colour for subsequent **PRINT** commands.

colour the palette entry number, in the range 0 – 31

See appendix 6 on page 297 for the list of colours in the default system palette.

Remarks: This is another name for **COLOR**.

Example: Using **FOREGROUND**

FORMAT

Token: $FE $37

Format: **FORMAT** diskname [**,I** id] [**,D** drive] [**,U** unit]

Usage: Formats a disk. *This erases all data on the disk.*

I The disk ID.

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

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **FORMAT** is another name for the **HEADER** command.

For new floppy disks which have not already been formatted in MEGA65 (1581) format, it is necessary to specify the disk ID with the **I** parameter. This switches the format command to low level format, which writes sector IDs and erases all contents. This takes some time, as every block on the floppy disk will be written.

If the **I** parameter is omitted, a quick format will be performed. This is only possible if the disk has already been formatted as a MEGA65 or 1581 floppy disk. A quick format writes the new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, and blocks are not overwritten, so contents may be recovered with **ERASE R**. You can read more about **ERASE** on page 102.

Examples: Using **FORMAT**

```
FORMAT "ADVENTURE",IDK : FORMAT DISK WITH NAME ADVENTURE AND ID DK
FORMAT "ZORK-I",U9     : FORMAT DISK IN UNIT 9 WITH NAME ZORK-I
FORMAT "DUNGEON",D1,U10: FORMAT DISK IN DRIVE 1 UNIT 10 WITH NAME DUNGEON
```

FRE

Token: $B8

Format: **FRE**(bank)

Returns: The number of free bytes for banks 0 or 1, or the ROM version if the argument is negative.

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. **FRE(1)** also triggers "garbage collection", which is a process that collects strings in use at the top of the bank, thereby defragmenting string memory.

FRE(-1) returns the ROM version, a six-digit number of the form 92XXXX.

Example: Using **FRE**:

```
10 PM = FRE(0)
20 VM = FRE(1)
30 RV = FRE(-1)
40 PRINT PM;" FREE FOR PROGRAM"
50 PRINT VM;" FREE FOR VARIABLES"
60 PRINT RV;" ROM VERSION"
```

FREAD

Token: $FE $1C

Format: **FREAD#** channel**,** pointer**,** size

Usage: Reads **size** bytes from **channel** to memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**

FREAD can be used to read data from disk directly into a variable. It is recommended to use the **POINTER** statement for the pointer argument, and to compute the size parameter by multiplying the number of elements with the item size.

Type	**Item Size**
Byte Array	1
Integer Array	2
Real Array	5

Keep in mind that the **POINTER** function with a string argument does *not* return the string address, but the string descriptor. It is not recommended to use **FREAD** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

To read into an array, ensure that you always specify an array index so that **POINTER** returns the address of an element. The start address of array `XY()` is `POINTER(XY(0))`. `POINTER(XY)` returns the address of the scalar variable `XY`.

Example: Using **FREAD**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FORI=0TON-1:PRINTCHR$(B&(I));:NEXT
160 FORI=0TON-1:C&(I)=B&(N-1-I):NEXT
170 DOPEN#2,"REVERS",W
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

FREEZER

Token: $FE $4A

Format: **FREEZER**

Usage: Invokes the Freezer menu.

Remarks: Entering the **FREEZER** command is an alternative to holding and releasing the RESTORE key.

Examples: Using **FREEZER**

```
FREEZER :REM CALL FREEZER MENU
```

FWRITE

Token: $FE $1E

Format: **FWRITE#** channel**,** pointer**,** size

Usage: Writes **size** bytes to **channel** from memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

FWRITE can be used to write the value of a variable to a file. It is recommended to use the **POINTER** statement for the pointer argument and compute the size parameter by multiplying the number of elements with the item size.

Refer to the **FREAD** item size table on page 118 for the item sizes.

Keep in mind that the **POINTER** function with a string argument does *not* return the string address, but the string descriptor. It is not recommended to use **FWRITE** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

To write an array, ensure that you always specify an array index so that **POINTER** returns the address of an element. The start address of array `XY()` is `POINTER(XY(0))`. `POINTER(XY)` returns the address of the scalar variable `XY`.

Example: Using **FWRITE**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FORI=0TON-1:PRINTCHR$(B&(I));:NEXT
160 FORI=0TON-1:C&(I)=B&(N-1-I):NEXT
170 DOPEN#2,"REVERS",W
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

GCOPY

Token: $FE $32

Format: **GCOPY** x, y, width, height

Usage: Bitmap graphics: copies the content of the specified rectangle with upper left position **x, y** and the **width** and **height** to a buffer.

The copied region can be inserted at any position with the command **PASTE**.

Remarks: The size of the rectangle is limited by the 1K size of the buffer. The memory requirement for a region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using **GCOPY** (see also **CUT**).

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1  :REM DRAW A WHITE BOX
30 GCOPY 140,80,40,40   :REM COPY A 40 * 40 REGION
40 PASTE 10,10,40,40    :REM PASTE IT TO NEW POSITION
50 GETKEY A$            :REM WAIT FOR KEYPRESS
60 SCREEN CLOSE
```

GET

Token: $A1

Format: **GET** variable

Usage: Gets the next character, or byte value of the next character, from the keyboard queue.

If the variable being set to the character is of type string and the queue is empty, an empty string is assigned to it, otherwise a one character string is created and assigned instead. If the variable is of type numeric, the byte value of the key is assigned to it, otherwise zero will be assigned if the queue is empty. **GET** does not wait for keyboard input, so it's useful to check for key presses at regular intervals or in loops.

Remarks: **GETKEY** is similar, but waits until a key has been pressed.

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**,** variable [**,** variable ...]

Usage: Reads a single byte from the channel argument and assigns single character strings to string variables, or an 8-bit binary value to numeric variables.

This is useful for reading characters (or bytes) from an input stream one byte at a time.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

Remarks: All values from 0 to 255 are valid, so **GET#** can also be used to read binary data.

Example: Using **GET#** to read a disk directory:

```
1 REM GET#
10 OPEN 2,8,0,"$"            : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT 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,LO,HI             : REM FILE SIZE BYTES
45   S=LO + 256 * HI         : REM FILE SIZE
50   LINE INPUT#2, F$        : REM FILE NAME
55   PRINT S;F$              : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

GETKEY

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

Format: **GETKEY** variable

Usage: Gets the next character, or byte value of the next character, from the keyboard queue. If the queue is empty, the program will wait until a key has been pressed.

After a key has been pressed, the variable will be set and program execution will continue. When used with a string variable, a one character string is created and assigned. Otherwise if the variable is of type numeric, the byte value is assigned.

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 MEGA65 to C64-compatible mode.

If you're in direct mode, a security prompt `ARE YOU SURE?` is displayed, which must be responded with `Y` to continue.

You can switch back to MEGA65 mode with this command: `SYS58552`

Example: Using **GO64**:

```
GO64
ARE YOU SURE?
```

GOSUB

Token: $8D

Format: **GOSUB** line

Usage: **GOSUB** (GOto SUBroutine) continues program execution at the given BASIC line number, saving the current BASIC program counter and line number on the run-time stack. This enables the resumption of execution after the **GOSUB** statement, once a **RETURN** statement in the called subroutine is executed. Calls to subroutines via **GOSUB** may be nested, but the subroutines must always end with **RETURN**, otherwise a stack overflow may occur.

Remarks: Unlike other programming languages, BASIC 65 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 fewer digits to decode. The subroutines will also be found faster, since the search for subroutines often starts at the beginning 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 REM *** 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.

Remarks: If the target **line** number is higher than the current line number, the search starts from the current line, proceeding to higher line numbers. If the target **line** number is lower, the search starts at the first **line** number of the program. It is possible to optimise the run-time speed of the program by grouping often used targets at the start (with lower line numbers).

GOTO (written as a single word) executes faster than **GO TO**.

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 GOSUB 30: IF DD THEN DCLOSE#2:GOSUB 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GRAPHIC

Token: $DE

Format: **GRAPHIC CLR**

Usage: Bitmap graphics: initialises the BASIC bitmap graphics system. It clears the graphics memory and screen, and sets all parameters of the graphics context to their default values.

Once the graphics system has been cleared, commands such as **LINE**, **PALETTE**, **PEN**, **SCNCLR**, and **SCREEN** can be used to set graphics system parameters.

Example: Using **GRAPHIC**:

```
100 REM GRAPHIC
110 GRAPHIC CLR        : REM INITIALISE
120 SCREEN DEF 1,1,1,2 : REM 640 X 400 X 2
130 SCREEN OPEN 1      : REM OPEN IT
140 SCREEN SET 1,1     : REM VIEW IT
150 PALETTE 1,0,0, 0,0 : REM BLACK
160 PALETTE 1,1,0,15,0 : REM GREEN
170 SCNCLR 0           : REM FILL SCREEN WITH BLACK
180 PEN 0,1            : REM SELECT PEN
190 LINE 50,50,590,350 : REM DRAW LINE
200 GETKEY A$          : REM WAIT FOR KEYPRESS
210 SCREEN CLOSE 1     : REM CLOSE SCREEN AND RESTORE PALETTE
```

HEADER

Token: $F1

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

Usage: Formats a disk. *This erases all data on the disk.*

I The disk ID.

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

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **HEADER** is another name for the **FORMAT** command.

For new floppy disks which have not already been formatted in MEGA65 (1581) format, it is necessary to specify the disk ID with the **I** parameter. This switches the format command to low level format, which writes sector IDs and erases all contents. This takes some time, as every block on the floppy disk will be written.

If the **I** parameter is omitted, a quick format will be performed. This is only possible if the disk has already been formatted as a MEGA65 or 1581 floppy disk. A quick format writes the new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, and blocks are not overwritten, so contents may be recovered with **ERASE R**. You can read more about **ERASE** on page 102.

Examples: Using **HEADER**

```
HEADER "ADVENTURE",IDK : FORMAT DISK WITH NAME ADVENTURE AND ID DK
HEADER "ZORK-I",U9     : FORMAT DISK IN UNIT 9 WITH NAME ZORK-I
HEADER "DUNGEON",D1,U10: FORMAT DISK IN DRIVE 1 UNIT 10 WITH NAME DUNGEON
```

HELP

Token: $EA

Format: **HELP**

Usage: Displays information about where an error occurred in a BASIC program.

When the BASIC program stops due to an error, **HELP** can be used to gain further information. The interpreted line is listed, with the erroneous statement highlighted or underlined.

Remarks: Displays BASIC errors. For errors related to disk I/O, the disk status variable **DS** or the disk status string **DS$** should be used instead.

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
```

HEX$

Token: $D2

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

Returns: A four character hexadecimal representation of the argument.

The argument must be in the range of 0 – 65535, corresponding to the hex numbers $0000-$FFFF.

Remarks: If real numbers are used as arguments, the fractional part will be ignored. In other words, real numbers will not be rounded.

Example: Using **HEX$**:

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

HIGHLIGHT

Token: $FE $3D

Format: **HIGHLIGHT** colour [, mode]

Usage: Sets the colours used for code highlighting.

Different colours can be set for system messages, **REM** statements and BASIC 65 keywords.

colour is one of the first 16 colours in the current palette. See appendix 6 on page 297 for the list of colours in the default system palette.

mode indicates what the colour will be used for.

- **0** system messages (the default mode)
- **1 REM** statements
- **2** BASIC keywords

Remarks: The system messages colour is used when displaying error messages, and in the output of **CHANGE**, **FIND**, and **HELP**. The colours for **REM** statements and BASIC keywords are used by **LIST**.

Example: Using **HIGHLIGHT** to change the colour of BASIC keywords to red.

```
LIST

10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"

READY.
HIGHLIGHT 8,2

READY.
LIST

10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"

READY.
█
```

IF

Token: $8B

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

Usage: Starts a conditional execution statement.

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

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

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

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

Example: Using **IF**

```
1 REM IF
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

IMPORT

Token: $DD

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

Usage: Loads BASIC code in text format from a file of type **SEQ** into memory reserved for BASIC programs.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: The program is loaded into BASIC memory and converted from text to the tokenised form of **PRG** files. This enables loading of BASIC programs that were saved as plain text files as program listing.

After loading, the program is re-linked and ready to be **RUN** or edited. It is possible to use **IMPORT** for merging a program text file from disk to a program already in memory. Each line read from the file is processed in the same way, as if typed from the user with the screen editor.

There is no **EXPORT** counterpart, because this function is already available. The sequence `DOPEN#1,"LISTING",W:CMD 1:LIST:DCLOSE#1` converts the program in memory to text and writes it to the file, that is named in the **DOPEN** statement.

Examples: Using **IMPORT**

```
IMPORT "APOCALYPSE"
IMPORT "MEGA TOOLS",U9
IMPORT (FI$),U(UN%)
```

INFO

Token: $FE $4D

Format: **INFO**

Usage: Displays information about the runtime environment.

Remarks: The **INFO** command displays information about the BASIC runtime environment, including:

- The video mode (PAL, NTSC)
- The version of the ROM
- The CPU speed
- The current **MEM** setting
- Memory used and memory available for program text and variables

Examples: Using **INFO**

```
INFO
```

INPUT

Token: $85

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

Usage: Prompts the user for keyboard input, printing an optional prompt string and question mark to the screen.

prompt optional string expression to be printed as the prompt

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 instead.

variable list list of one or more variables that receive the input

The input will be processed after the user presses RETURN.

Remarks: The user must take care to enter the correct type of input, so it matches the **variable list** types. Also, the number of input items must match the number of variables. A surplus of input items will be ignored, whereas too few input items trigger another request for input with the prompt ??. Typing non numeric characters for integer or real variables will produce a `TYPE MISMATCH ERROR`. Strings for string variables must be in double quotes (") if they contain spaces or commas. Many programs that need a safe input routine use **LINE INPUT** and a custom parser, in order to avoid program errors by wrong user input.

Example: Using **INPUT**:

```
 10 DIM N$(100),A%(100),S$(100):
 20 DO
 30 INPUT "NAME, AGE, GENDER";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 "GENDER?":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#** channel**,** variable [**,** variable ...]

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

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

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.
LINE INPUT# may be used to read a whole record into a single string variable.

Sequential files, that can be read by **INPUT#** can be generated by programs with **PRINT#** or with the editor of the MEGA65. For example:

```
EDIT ON

10 "CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"
20 "JACK TRAMIEL",1928,"FOUNDER OF CBM"
30 "BILL MENSCH",1945,"HARDWARE"

DSAVE "CBM-PEOPLE"
EDIT OFF
```

Example: Using **INPUT#**:

```
 10 DIM N$(100),B%(100),S$(100):
 20 DOPEN#2,"CBM-PEOPLE":REM OPEN SEQ FILE
 25 IF DS THEN PRINT DS$:STOP:REM OPEN ERROR
 30 FOR I=0 TO 100
 40 INPUT#2,N$(I),B%(I),S$(I)
 50 IF ST AND 64 THEN 80:REM END OF FILE
 60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
 70 NEXT I
 80 DCLOSE#2
110 PRINT "READ";I+1;" RECORDS"
120 FOR J=0 TO I:PRINT N$(J):NEXT J

RUN
READ 3  RECORDS
CHUCK PEDDLE
JACK TRAMIEL
BILL MENSCH

TYPE "CBM-PEOPLE"
"CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"
"JACK TRAMIEL",1928,"FOUNDER OF CBM"
"BILL MENSCH",1945,"HARDWARE"
```

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**.

An enhanced version of string search using pattern matching is used if the first character of the search string is a pound sign '£'. The pound sign is not part of the search but enables the use of the **'.'** (dot) as a wildcard character, which matches any character. The second special pattern character is the **'*'** (asterisk) character. The asterisk in the search string indicates that the preceding character may never appear, appear once, or repeatedly in order to be considered as a match.

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.

Examples: Using **INSTR**:

```
I = INSTR("ABCDEF","CD")      : REM I = 3
I = INSTR("ABCDEF","XY")      : REM I = 0
I = INSTR("RAIIIN","£A*IN")   : REM I = 5
I = INSTR("ABCDEF","£C.E")    : REM I = 3
I = INSTR(A$+B$,C$)
```

INT

Token: $B5

Format: **INT(**numeric expression**)**

Returns: The integer part of a number.

This function is **NOT** limited to the typical 16-bit integer range (-32768 to 32767), as it uses real arithmetic. The allowed range is therefore determined by the size of the real mantissa which is 32-bits wide (-2147483648 to 2147483647).

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

Examples: Using **INT**:

```
X  = INT(1.9)       :REM X = 1
X  = INT(-3.1)      :REM X = -3
X  = INT(100000.5)  :REM X = 100000
N% = INT(100000.5)  :REM ?ILLEGAL QUANTITY ERROR
```

JOY

Token: $CF

Format: **JOY**(port)

Returns: The state of the joystick for the selected controller 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
KEY <**ON** | **OFF**>
KEY <**LOAD** | **SAVE**> filename
KEY number, string

Usage: Manages the function key macros in the BASIC editor.

Each function key can be assigned a string that is typed when pressed. The function keys have default assignments on boot, and can be changed by the **KEY** command.

KEY : list current assignments.

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

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

KEY LOAD filename : loads key definitions from file.

KEY SAVE filename : saves key definitions to file.

KEY number, string : assigns the string to the key with the given number.

number can be any value within this range:

- **1 - 14**: corresponds to keys ranging from F1 to F14
- **15**: corresponds to HELP
- **16**: corresponds to SHIFT RUN STOP

Default assignments:

```
KEY
KEY 1,CHR$(27)+"X"
KEY 2,CHR$(27)+"@"
KEY 3,"DIR"+CHR$(13)
KEY 4,"DIR "+CHR$(34)+"*=PRG"+CHR$(34)+CHR$(13)
KEY 5,"U"
KEY 6,"KEY6"+CHR$(141)
KEY 7,"W"
KEY 8,"MONITOR"+CHR$(13)
KEY 9,"P"
KEY 10,"KEY10"+CHR$(141)
KEY 11,"V"
KEY 12,"KEY12"+CHR$(141)
KEY 13,CHR$(27)+"O"
KEY 14,"U"+CHR$(27)+"O"
KEY 15,"HELP"+CHR$(13)
KEY 16,"RUN "+CHR$(34)+"*"+CHR$(34)+CHR$(13)
```

Remarks: The sum of the lengths of all assigned strings must not exceed 240 characters. Special characters such as RETURN or QUOTE are entered using their codes with the **CHR$** function. Refer to **CHR$** on page 47 for more information.

Examples: 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
KEY SAVE "MY KEY SET"    :REM SAVE CURRENT DEFINITIONS TO FILE
KEY LOAD "ELEVEN-SET"    :REM LOAD DEFINITIONS FROM FILE
```

LEFT$

Token: $C8

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

Returns: A string containing the first **n** characters from the argument **string**.

If the length of **string** is equal to or less than **n**, the resulting string will be identical to the argument string.

string a string expression

n a numeric expression (0 - 255)

Remarks: Empty strings and zero length strings are legal values.

Example: Using **LEFT$**:

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

LEN

Token: $C3

Format: **LEN**(string)

Returns: The length of a string.

string a string expression

Remarks: Commodore BASIC strings can contain any character, including the null character. Internally, the length of a string is stored in a string descriptor.

Example: Using **LEN**:

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

LET

Token: $88

Format: [**LET**] variable = expression

Usage: Assigns values (or results of expressions) to variables.

Remarks: The **LET** statement is obsolete and not required. Assignment to variables can be done without using **LET**, but it has been left in BASIC 65 for backwards compatibility.

Examples: Using **LET**:

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

LINE

Token: $E5

Format: **LINE** xbeg, ybeg [**,** xnext1**,** ynext1 ...]

Usage: Bitmap graphics: draws a line or series of lines.

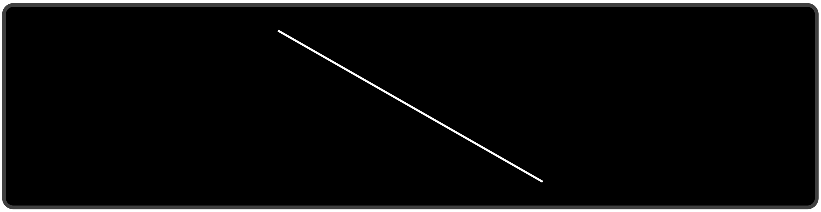
If only one coordinate pair is given, **LINE** draws a dot.

If more than one pair is defined, a line is drawn on the current graphics screen from the coordinate (xbeg/ybeg) to the next coordinate pair(s).

All currently defined modes and values of the graphics context are used.

Example: Using **LINE**:

```
1 REM SCREEN EXAMPLE 1
10 SCREEN 320,200,2      :REM SCREEN #0 320 X 200 X 2
20 PEN 1                 :REM DRAWING PEN COLOUR 1 (WHITE)
30 LINE 25,25,295,175    :REM DRAW LINE
40 GETKEY A$             :REM WAIT FOR KEYPRESS
50 SCREEN CLOSE          :REM CLOSE SCREEN AND RESTORE PALETTE
```



LINE INPUT

Token: $E5 $85

Format: **LINE INPUT** [prompt <**,** | **;**>] string variable [**,** string variable ...]

Usage: Prompts the user for keyboard input, printing an optional prompt string and question mark to the screen.

prompt optional string expression to be printed as the prompt

If the separator between **prompt** and the first **string variable** 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 instead.

string variable one or more string variables that accept one line of input each

Remarks: This differs from **INPUT** in how the input is parsed. **LINE INPUT** accepts every character entered on a line as a single string value. Only the RETURN key does not produce a character.

If the variable list has more than one variable, **LINE INPUT** will use the entire first line for the first variable, and present the ?? prompt for each subsequent variable.

LINE INPUT only works with string variables. If a non-string variable is used, **LINE INPUT** throws produces a `TYPE MISMATCH ERROR` after data has been entered.

Example: Using **LINE INPUT**:

```
10 LINE INPUT "ENTER A PHRASE: ",PH$
20 PRINT "THE PHRASE YOU ENTERED:";CHR$(13);"  ";PH$
RUN
ENTER A PHRASE: YOU SAY "POTATO," I SAY "POTATO."
THE PHRASE YOU ENTERED:
  YOU SAY "POTATO," I SAY "POTATO."
```

LINE INPUT#

Token: $E5 $84

Format: **LINE INPUT#** channel**,** variable [**,** variable ...]

Usage: Reads one record per variable from an input device, (such as a disk drive) and assigns the read data to the variable.

The records must be terminated by a **RETURN** character, which will not be copied to the string variable. Therefore, an empty line consisting of only the **RETURN** character will result in an empty string being assigned.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

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

Remarks: Only string variables or string array elements can be used in the variable list. Unlike other INPUT commands, **LINE INPUT#** does not interpret or remove quote characters in the input. They are accepted as data, as all other characters.

Records must not be longer than the input buffer, which is 160 characters.

Example: Using **LINE INPUT#**:

```
 10 DIM N$(100)
 20 DOPEN#2,"DATA"
 30 FOR I=0 TO 100
 40 LINE INPUT#2,N$(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"
```

LIST

Token: $9B

Format: **LIST** [**P**] [line range]

Usage: Lists a range of lines from the BASIC program in memory.

Given a single line number, **LIST** lists that line.

Given a range of line numbers, **LIST** lists all lines in that range. A range can be two numbers separated by a hyphen (**-**), or it can omit the beginning or end of the range to imply the beginning or end of the program. (See examples below.)

Format: **LIST** [**P**] filename [**,U** unit]

Usage: Lists a range of lines from a BASIC program directly from a file.

Remarks: The optional parameter **P** enables page mode. After listing a screenful of lines, the listing will stop and display the prompt `[MORE]` at the bottom of the screen. Pressing **Q** quits page mode, while any other key continues to the next page.

LIST output can be redirected to other devices via **CMD**.

Another way to display a program listing from memory on the screen is to use the keys **F9** and **F11**, or **Ctrl** **P** and **Ctrl** **V**, to scroll a BASIC listing on screen up or down.

Examples: 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
LIST "DEMO"   :REM LIST FILE "DEMO"
LIST P        :REM LIST PROGRAM IN PAGE MODE
LIST P "MURX" :REM LIST FILE "MURX" IN PAGE MODE
```

LOAD

Token: $93

Format:
LOAD filename [, unit [, flag]]
LOAD "$[pattern=type]" [**, unit**]
LOAD "$$[pattern=type]" [**, unit**]
/ filename [, unit [, flag]]

Usage: The first form loads a file of type **PRG** into memory reserved for BASIC programs.

The second form loads a directory into memory, which can then be viewed with **LIST**. It is structured like a BASIC program, but file sizes are displayed instead of line numbers.

The third form is similar to the second one, but the files are numbered. This listing can be scrolled like a BASIC program with the keys **F9** or **F11**, edited, listed, saved or printed.

A filter can be applied by specifying a pattern or a pattern and a type. The asterisk matches the rest of the name, while the ? matches any single character. The type specifier can be a character of (P,S,U,R), that is Program, Sequential, User, or Relative file.

A common use of the shortcut symbol **/** is to quickly load **PRG** files. To do this:

1. Print a disk directory using either **DIR**, or **CATALOG**.
2. Move the cursor to the desired line.
3. type **/** in the first column of the line, and press **RETURN**.

After pressing **RETURN**, the listed file on the line with the leading **/** will be loaded. Characters before and after the file name double quotes (") will be ignored. This applies to **PRG** files only.

filename is either a quoted string, e.g. `"PROG"`, or a string expression.

The unit number is optional. If not present, the default disk device is assumed.

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

LOAD "*" can be used to load the first **PRG** from the given **unit**.

LOAD "$" can be be used to load the list of files from the given **unit**. When using **LOAD "$"**, **LIST** can be used to print the listing to screen.

LOAD is implemented in BASIC 65 to keep it backwards compatible with BASIC V2.

The shortcut symbol **/** can only be used in direct mode.

By default the C64 uses **unit** 1, which is assigned to datasette tape recorders connected to the cassette port. However the MEGA65 uses **unit** 8 by default, which is assigned to the internal disk drive. This means you don't need to add `,8` to **LOAD** commands that use it.

Examples: Using **LOAD**

```
LOAD "APOCALYPSE"   :REM LOAD A FILE CALLED APOCALYPSE TO BASIC MEMORY
LOAD "MEGA TOOLS",9 :REM LOAD A FILE CALLED "MEGA TOOLS" FROM UNIT 9 TO BASIC MEMORY
LOAD "*",8,1        :LOAD THE FIRST FILE ON UNIT 8 TO RAM AS SPECIFIED IN THE FILE

LOAD "$"            :REM LOAD WHOLE DIRECTORY - WITH FILE SIZES
LOAD "$$"           :REM LOAD WHOLE DIRECTORY - SCROLLABLE
LOAD "$$X*=P"       :REM DIRECTORY, WITH PRG FILES STARTING with 'X'
```

LOADIFF

Token: $FE $43

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

Usage: Bitmap graphics: loads an IFF file into graphics memory.

The IFF (Interchange File Format) is supported by many different applications and operating systems. **LOADIFF** assumes that files contain bitplane graphics which match the currently active graphics screen for resolution and colour depth.

Supported resolutions are:

Width	Height	Bitplanes	Colours	Memory
320	200	max. 8	max. 256	max. 64 K
640	200	max. 8	max. 256	max. 128 K
320	400	max. 8	max. 256	max. 128 K
640	400	max. 4	max. 16	max. 128 K

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as AMIGA OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netbpm** package.

To use **convert** and **ppmtoilbm** for converting a JPG file to an IFF file on Linux:

```
convert <myImage.jpg> <myImage.ppm>
ppmtoilbm -aga <myImage.ppm> > <myImage.iff>
```

Example: Using **LOADIFF**

```
100 BANK128:SCNCLR
110 REM DISPLAY PICTURES IN 320 X 200 X 7 RESOLUTION
120 GRAPHIC CLR:SCREEN DEF 0,0,0,7:SCREEN OPEN 0:SCREEN SET 0,0
130 FORI=1TO7: READF$
140 LOADIFF(F$+".IFF"):SLEEP 4:NEXT
150 DATA ALIEN,BEAKER,JOKER,PICARD,PULP,TROOPER,RIPLEY
160 SCREEN CLOSE 0
170 PALETTE RESTORE
```

LOCK

Token: $FE $50

Format: **LOCK** filename/pattern [**,D** drive] [**,U** unit]

Usage: Locks a file on disk, preventing it from being updated or deleted.

The specified file or a set of files, that matches the pattern, is locked and cannot be deleted with the commands **DELETE**, **ERASE** or **SCRATCH**.

The command **UNLOCK** removes the lock.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: In direct mode the number of locked files is printed. The second to last number from the message contains the number of locked files,

Examples: Using **LOCK**

```
LOCK "DRM",U9 :REM LOCK FILE DRM ON UNIT 9
03,FILES LOCKED,01,00
LOCK "BS*"    :REM LOCK ALL FILES BEGINNING WITH "BS"
03,FILES LOCKED,04,00
```

LOG

Token: $BC

Format: **LOG(**numeric expression**)**

Returns: The natural logarithm of a number.

The natural logarithm uses Euler's number (**2.71828183**) as base, not base 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). **LOG10()** provides this feature as a function.

Example: Using **LOG**

```
PRINT LOG(1)
 0

PRINT LOG(0)
 ?ILLEGAL QUANTITY ERROR

PRINT LOG(4)
 1.38629436

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

LOG10

Token: $CE $08

Format: **LOG10**(numeric expression)

Returns: The decimal logarithm of the argument.

The decimal logarithm uses 10 as base.

Example: Using **LOG10**

```
PRINT LOG10(1)
 0

PRINT LOG10(0)
 ?ILLEGAL QUANTITY ERROR

PRINT LOG10(5)
 0.69897

PRINT LOG10(100);LOG10(10);LOG10(1);LOG10(0.1);LOG10(0.01)
 2  1  0 -1 -2
```

LOOP

Token: $EC

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

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

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

Examples: 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)

Returns: The state of a light pen peripheral.

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

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

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

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

Example: Using **LPEN**

```
PRINT LPEN(0),LPEN(1)   :REM PRINT LIGHT PEN COORDINATES
```

MEM

Token: $FE $23

Format: **MEM mask4,mask5**

Usage: Reserves memory in banks 4 or 5 such that the bitmap graphics system will not use it.

mask4 and **mask5** are byte values, that are interpreted as mask of 8 bits. Each bit set to 1 reserves an 8K segment of memory in bank 4 for the first argument and in bank 5 for the second argument.

bit	memory segment
0	$0000 - $1FFF
1	$2000 - $3FFF
2	$4000 - $5FFF
3	$6000 - $7FFF
4	$8000 - $9FFF
5	$A000 - $BFFF
6	$C000 - $DFFF
7	$E000 - $FFFF

Remarks: After reserving memory with **MEM** the graphics library will not use the reserved areas, so it can be used for other purposes. Access to bank 4 and 5 is possible with the commands **PEEK, WPEEK, POKE, WPOKE** and **EDMA**.

If a graphics screen cannot be opened, because the remaining memory is not sufficient, the program stops with a ?OUT OF MEMORY ERROR.

Some direct mode commands like **RENUMBER** use memory in banks 4 and 5 and do not honour **MEM** reservations. Such reservations are only guaranteed during program execution.

When 80 × 50 text mode is enabled, segment 0 is reserved automatically and used for screen data. It always uses segment 0, even if it was previously reserved with **MEM** or a graphic screen. If your program uses 80 × 50 text mode and also reserves a region with **MEM**, be sure to set region 0 as reserved, and do not use it for other purposes.

Example: Using **MEM**

```
10 MEM 1,3        :REM RESERVE $40000 - $41FFF AND $50000 - $53FFF
20 SCREEN 320,200 :REM SCREEN WILL NOT USE RESERVED SEGMENTS
40 EDMA 3,$2000,0,$4000:REM FILL SEGMENT WITH ZEROES
```

MERGE

Token: $E6

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

Usage: Loads a BASIC program file from disk and appends it to the program in memory.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: The load address that is stored in the first two bytes of the file is ignored. The loaded program does not replace a program in memory (which is what **DLOAD** does), but is appended to a program in memory. After loading, the program is re-linked and ready to run or edit.

It is the user's responsibility to ensure that there are no line number conflicts among the program in memory and the merged program. The first line number of the merged program must be greater than the last line number of the program in memory.

Example: Using **MERGE**

```
DLOAD "MAIN PROGRAM"
MERGE "LIBRARY"
```

MID$

Token: $CA

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

Usage: As a function, the substring of a string. As a statement, replaces a sub-string of a string variable with another string.

string a string expression.

index start index (1 - 255).

n length of sub-string (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
```

MKDIR

Token: $FE $51

Format: **MKDIR** dirname **,L** size [**,U** unit]

Usage: Makes (creates) a subdirectory on a floppy or D81 disk image.

dirname the name of a directory. Either a quoted string such as "SOMEDIR", or a string expression in brackets such as (DR$).

MKDIR can only be used on units managed by CBDOS. These are the internal floppy disk drive and SD-Card images of **D81** type. The command cannot be used on external drives connected to the serial IEC bus.

The **size** parameter specifies the number of tracks, to be reserved for the subdirectory, with one track = 40 sectors at 256 byte. The first track of the reserved range is used as directory track for the subdirectory.

The minimum size is 3 tracks, the maximm 38 tracks. There must be a contiguous region of empty tracks on the floppy (D81 image), that is large enough for the creation of the subdirectory. The error message DISK FULL is reported if there isn't such a region.

Several subdirectories may be created as long as there are enough empty tracks.

After successful creation of the subdirectory an automatic **CHDIR** into this subdirectory is performed.

CHDIR "/" changes back to the root directory.

Examples: Using **MKDIR**

```
MKDIR "SUBDIR",L5   :REM MAKE SUBDIRECTORY WITH 5 TRACKS
DIR
0 "SUBDIR          "    1D
160 BLOCKS FREE.
```

MOD

Token: $NN

Format: **MOD(**dividend**,** divisor**)**

Returns: The remainder of a division operation.

Remarks: In other programming languages such as C, this function is implemented as an operator (%). In BASIC 65 it is implemented as a function.

Example: Using **MOD**:

```
FOR I = 0 TO 8: PRINT MOD(I,4);: NEXT I
 0  1  2  3  0  1  2  3  0
```

MONITOR

Token: $FA

Format: **MONITOR**

Usage: Invokes the machine language monitor.

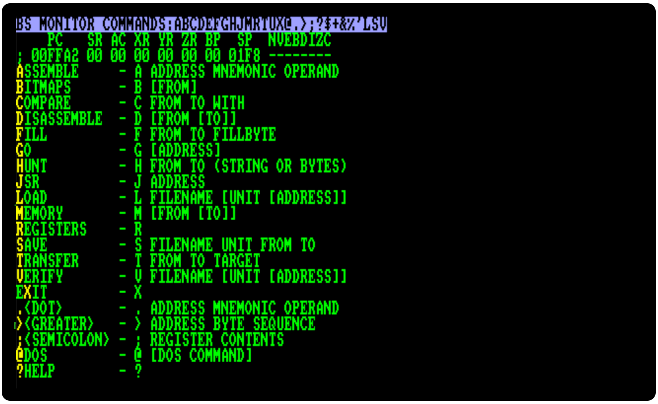
Remarks: Using the **MONITOR** requires knowledge of the CSG4510 / 6502 / 6510 CPU, the assembly language they use, and their architectures. More information on the **MONITOR** is available in *the MEGA65 Book*, Machine Language Monitor (Appendix N).

To exit the monitor press **X**.

Help text can be displayed with either **?** or **H**.

Example: Using **MONITOR**

```
MONITOR
```



MOUNT

Token: $FE $49

Format: **MOUNT** filename [**,U** unit]

Usage: Mounts a floppy image file of type **D81** from SD-Card to unit 8 (default) or unit 9.

If no argument is given, **MOUNT** assigns the real floppy drive of the MEGA65 to unit 8.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

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

Remarks: **MOUNT** can be used either in direct mode or in a program. It searches the file on the SD-card and mounts it, as requested, on unit 8 or 9. After mounting the floppy image can be used as usual with all DOS commands.

Examples: Using **MOUNT**

```
MOUNT "APOCALYPSE.D81"  ;REM MOUNT IMAGE TO UNIT 8
MOUNT "BASIC.D81",U9    :REM MOUNT IMAGE TO UNIT 9
MOUNT (FI$),U(UN%)      :REM MOUNT WITH VARIABLE ARGUMENTS
MOUNT                   :REM SELECT REAL FLOPPY DRIVE
```

MOUSE

Token: $FE $3E

Format: **MOUSE ON** [{**,** port**,** sprite**,** hotspot**,** 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 or 2 (default 2).

sprite sprite number for mouse pointer (default 0).

hostpot location of the "hot spot" that determines the position and click target (x,y) (default 0,0).

pos initial mouse position (x,y). If not specified, uses the last known position of the sprite.

MOUSE OFF disables the mouse driver and hides the associated sprite.

Remarks: The "hot spot" of the mouse specifies where in the mouse sprite image is considered the click target, such as the top of an arrow or the center of a target reticle. The hot spot is always kept within the screen border. The default **hotspot** is 0,0, representing the top left corner of the sprite.

When the system boots, sprite 0 is initialised to a picture of a mouse pointer, with the hot spot at 0,0.

Use **RMOUSE** to test the location and button status of the mouse. This returns the coordinates of the top-left corner of the sprite, not the coordinates of the hot spot. To get the coordinates of the hot spot, add the hot spot location to the sprite coordinates.

pos can be an absolute coordinate, or a relative coordinate to the current mouse position, similar to **MOVSPR**.

Examples: 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

MOUSE ON,1,0,2,4 :REM SET THE HOT SPOT TO (2,4)
RMOUSE X,Y,B     :REM FETCH MOUSE SPRITE COORDINATES
X=X+2 : Y=Y+4    :REM CALCULATE THE COORDINATES OF THE HOT SPOT

REM SET THE INITIAL POSITION TO X=300 Y=75
MOUSE ON,1,0,0,0,300,75
```

MOVSPR

Token: $FE $06

Format: **MOVSPR** number**,** position

Usage: Moves a sprite to a location on screen.

Each **position** argument consists of two 16-bit values, which specify either an absolute coordinate, a relative coordinate, an angle, or a speed. The value type is determined by a prefix:

- **+value** relative coordinate: positive offset.
- **-value** relative coordinate: negative offset.
- **#value** speed.

If no prefix is given, the absolute coordinate or angle is used.

Therefore, the position argument can be used to either:

- set the sprite to an absolute position on screen.
- specify a displacement relative from the current position.
- trigger a relative movement from a specified position.
- describe movement with an angle and speed starting from the current position.

MOVSPR number, position is used to set the sprite immediately to the position or, in the case of an angle#speed argument, describe its further movement.

Format: **MOVSPR** number**,** start-position **TO** end-position**,** speed

Usage: Places the sprite at the start position, defines the destination position, and the speed of movement.

The sprite is placed at the start position, and will move in a straight line to the destination at the given speed. Coordinates must be absolute or relative. The movement is controlled by the BASIC interrupt handler and happens concurrently with the program execution.

number sprite number (0 – 7).

position x,y | xrel,y | x,yrel | xrel,yrel | angle#speed.

x absolute screen coordinate pixel.

y absolute screen coordinate pixel.

xrel relative screen coordinate pixel.

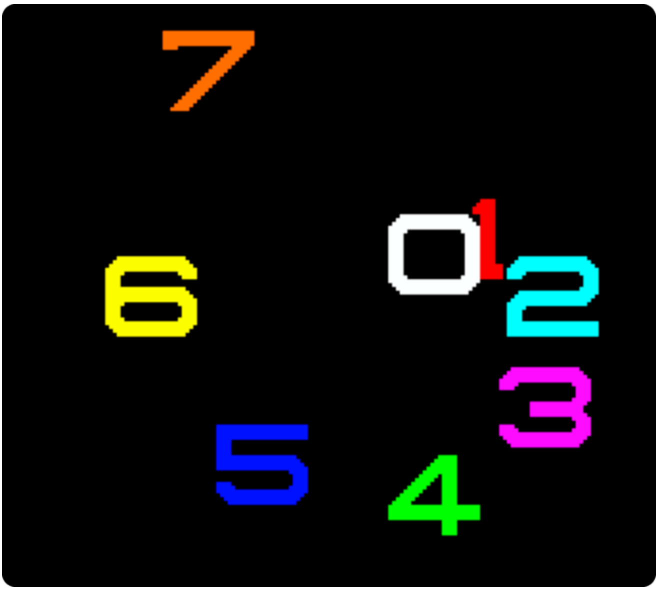
yrel relative screen coordinate pixel.

angle compass direction for sprite movement [degrees]. 0: up, 90: right, 180: down, 270: left, 45 upper right, etc.

speed speed of movement, configured as a floating point number in the range of 0.0 – 127.0, in pixels per frame. PAL has 50 frames per second whereas NTSC has 60 frames per second. A speed value of 1.0 will move the sprite 50 pixels per second in PAL mode.

Example: Using **MOVSPR**:

```
100 CLR:SCNCLR:SPRITECLR
110 BLOAD "DEMOSPRITES1",B0,P1536
130 FORI=0TO7: C=I+1:SP=0.07*(I+1)
140 MOVSPRI, 160,120
145 MOVSPRI,45*I#SP
150 SPRITEI,1,C,,0,0
160 NEXT
170 SLEEP 3
180 FORI=0TO7:MOVSPR I,0#0:NEXT
```



NEW

Token: $A2

Format: **NEW**
NEW RESTORE

Usage: Erases the BASIC program in memory, and resets all BASIC parameters to their default values.

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

Examples: 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: Marks the end of the BASIC loop associated with the given index variable. When a BASIC loop is declared with **FOR**, it must end with **NEXT**.

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.

start value to initialise the index with.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the index variable at the end of every iteration. 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 it is omitted, the variable for the current loop is assumed. Several consecutive **NEXT** statements may be combined by specifying the indexes in a comma separated list. The statements `NEXT I:NEXT J:NEXT K` and `NEXT I,J,K` are 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 bit-wise logical NOT operation on a 16-bit value.

Integer operands are used as they are, whereas real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to a 16-bit integer, using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
NOT 0	1
NOT 1	0

Remarks: The result is of type integer.

Examples: Using **NOT**

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

In most cases, **NOT** is 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, such as **KEY** and **MOUSE**.

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

Examples: Using **OFF**

```
KEY OFF   :REM DISABLE FUNCTION KEY STRINGS
MOUSE OFF :REM DISABLE MOUSE DRIVER
```

ON

Token: $91

Format: **ON** expression **GOSUB** line number [**,** line number ...]
ON expression **GOTO** line number [**,** line number ...]
keyword **ON**

Usage: Performs **GOSUB** or **GOTO** to a line number selected by a number expression.

Depending on the result of the expression, the target for **GOSUB** and **GOTO** is chosen from the table of line addresses at the end of the statement.

When used as a secondary keyword, **ON** is used in combination with primary keywords, such as **KEY** and **MOUSE**.

expression is a positive numeric value. Real values are converted to integer (losing precision). Logical operands are converted to a 16-bit integer, using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Remarks: Negative values for **expression** will stop the program with an error message. The **line number list** specifies the targets for values of 1, 2, 3, etc.
An expression result of zero, or a result that is greater than the number of target lines will not do anything, and the program will continue execution with the next statement.

Example: Using **ON**

```
 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** channel, first address [, secondary address [, filename]]

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

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 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 possible:

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 reserved 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 **DOPEN#**, since the **filename** for **OPEN** includes all file attributes, for example: `"0:DATA,S,W"`.

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

If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

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 bit-wise 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 (losing precision). Logical operands are converted to a 16-bit integer using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0), for FALSE.

Expression	Result
0 OR 0	0
0 OR 1	1
1 OR 0	1
1 OR 1	1

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

Example: Using **OR**

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

In most cases, **OR** is used in **IF** statements.

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

PAINT

Token: $DF

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

Usage: Bitmap graphics: performs a flood fill of an enclosed graphics area using the current pen colour.

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

mode specifies the paint mode:

- **0** The colour of pixel (x,y) defines the colour, which is replaced by the pen colour.
- **1** The **region border colour** defines the region to be painted with the pen colour.
- **2** Paint the region connected to pixel (x,y).

region border colour defines the colour index for mode 1.

Example: Using **PAINT**

```
10 SCREEN 320,200,2     :REM OPEN SCREEN
20 PALETTE 0,1,10,15,10 :REM COLOUR 1 TO LIGHT GREEN
30 PEN 1                :REM SET DRAWING PEN (PEN 0) TO LIGHT GREEN (1)
40 LINE 160,0,240,100   :REM 1ST. LINE
50 LINE 240,100,80,100  :REM 2ND. LINE
60 LINE 80,100,160,0    :REM 3RD. LINE
70 PAINT 160,10         :REM FILL TRIANGLE WITH PEN COLOUR
80 GETKEY A$            :REM WAIT FOR KEY
90 SCREEN CLOSE         :REM END GRAPHICS
```

PALETTE

Token: $FE $34

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

Usage: **PALETTE** 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 – 3).

COLOR keyword for changing system palette.

colour index to palette entry (0 – 255). PALETTE can define colours beyond the default system palette entries 0 – 31.

red red intensity (0 – 15).

green green intensity (0 – 15).

blue blue intensity (0 – 15).

Example: Using **PALETTE**

```
 10 REM CHANGE SYSTEM COLOUR INDEX
 20 REM --- INDEX  9 (BROWN) TO (DARK BLUE)
 30 PALETTE COLOR 9,0,0,7
```

```
 10 GRAPHIC CLR            :REM INITIALISE
 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 2                  :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
100 LINE 160,0,240,100     :REM 1ST. LINE
110 LINE 240,100,80,100    :REM 2ND. LINE
120 LINE 80,100,160,0      :REM 3RD. LINE
130 PAINT 160,10,0,2       :REM FILL TRIANGLE WITH BLUE (2)
140 GETKEY K$              :REM WAIT FOR KEY
150 SCREEN CLOSE 1         :REM END GRAPHICS
```

PASTE

Token: $E3

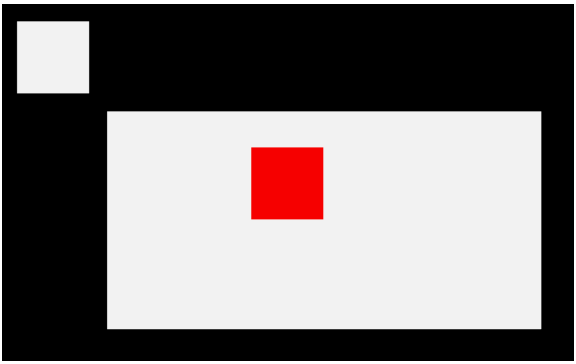
Format: **PASTE** x, y, width, height

Usage: Bitmap graphics: pastes the content of the **CUT** / **GCOPY** buffer onto the screen. The arguments upper left position **x, y** and the **width** and **height** specify the paste position on the screen.

Remarks: The size of the rectangle is limited by the 1K size of the buffer. The memory requirement for region is width * height * number of bitplanes / 8. It must not equal or exceed 1024 byte. For a 4-bitplane screen for example, a 45 x 45 region needs 1012.5 byte.

Example: Using **PASTE**

```
10 SCREEN 320,200,2
20 BOX 60,60,300,180,1  :REM DRAW A WHITE BOX
30 PEN 2                :REM SELECT RED PEN
40 CUT  140,80,40,40    :REM CUT OUT A 40 * 40 REGION
50 PASTE 10,10,40,40    :REM PASTE IT TO NEW POSITION
60 GETKEY A$            :REM WAIT FOR KEYPRESS
70 SCREEN CLOSE
```



PEEK

Token: $C2

Format: **PEEK**(address)

Returns: The byte value stored in memory at **address**, as an unsigned 8-bit number.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0 – 127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **PEEK**

```
10 BANK 128              :REM SELECT SYSTEM BANK
20 L = PEEK($02F8)       :REM USR JUMP TARGET LOW
30 H = PEEK($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: Bitmap graphics: sets the colour of the graphic pen for the current screen.

pen pen number (0 – 2):

- **0** drawing pen (default, if only single parameter provided).
- **1** off bits in jam2 mode.
- **2** currently unused.

colour palette index, from the palette of the current screen

See appendix 6 on page 297 for the list of colours in the default system palette.

Remarks: The colour selected by **PEN** will be used by all graphic/drawing commands that follow it. If you intend to set the drawing **pen** 0 to a colour, you can omit the first parameter, and only provide the **colour** parameter.

Example: Using **PEN**

```
 10 GRAPHIC CLR             :REM INITIALISE
 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 1                   :REM SET DRAWING PEN (PEN 0) TO RED (1)
100 LINE 160,0,240,100      :REM DRAW RED LINE
110 PEN 2                   :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
120 LINE 240,100,80,100     :REM DRAW BLUE LINE
130 PEN 3                   :REM SET DRAWING PEN (PEN 0) TO GREEN (3)
140 LINE 80,100,160,0       :REM DRAW GREEN LINE
150 GETKEY K$               :REM WAIT FOR KEY
160 SCREEN CLOSE 1          :REM END GRAPHICS
```

PIXEL

Token: $CE $0C

Format: **PIXEL**(x, y)

Returns: Bitmap graphics: the colour of a pixel at the given position.

x absolute screen coordinate.

y absolute screen coordinate.

PLAY

Token: $FE $04

Format: **PLAY** [{string1, string2, string3, string4, string5, string6}]

Usage: Starts playing a sequence of musical notes, or stops a currently playing sequence.

PLAY without any arguments will cause all voices to be silenced, and all of the music system's variables to be reset (such as **TEMPO**).

PLAY accepts up to six comma-separated string arguments, where each string describes the sequence of notes and directives to be played on a specific voice on the two available SID chips, allowing for up to 6-channel polyphony.

PLAY uses SID1 (for voices 1 to 3) and SID3 (for voices 4 to 6) of the 4 SID chips of the system. By default, SID1 and SID2 are slightly right-biased and SID3 and SID4 are slightly left-biased in the stereo mix.

```
PLAY "CEG"
PLAY "C","E","G"
```

Within a **PLAY** string, a musical note is a character (A, B, C, D, E, F, or G), which may be preceded by an optional modifier.

Possible modifiers are:

Character	Effect
#	Sharp
$	Flat
.	Dotted
W	Whole Note
H	Half Note
Q	Quarter Note
I	Eighth Note
S	Sixteenth Note
R	Pause (rest)

Notice that the dot (.) modifier appears before the note name, not after it as in traditional sheet music.

Directives consist of a letter, followed by a digit. Directives apply to all future notes, until the parameter is changed by another directive.

Character	Directive	Argument Range
O	Octave	0 – 6
T	Instrument Envelope	0 – 9
V	Volume	0 – 9
X	Filter	0 – 1
M	Modulation	0 – 9
P	Portamento	0 – 9
L	Loop	N/A

An octave is a range of notes from C to B. The default octave is 4, representing the "middle" octave.

Instrument envelopes describe the nature of the sound. See **ENVELOPE** for a list of default envelope styles, and information on how to adjust the ten envelopes.

The modulation directive adds a pitch-based vibrato your note by the magnitude you specify (1 – 9). A value of 0 disables it.

Similarly, the portamento directive slides between consecutive notes at the speed you specify (1 – 9). A value of 0 disables it. Note that the gate-off behaviour of notes is disabled while portamento is enabled. To re-enable the gate-off behavior, you must disable portamento (P0).

If a string ends with the **L** directive, the pattern loops back to the beginning of the string upon completion.

You can omit a string for a given voice to allow an already playing pattern in that voice to continue, using empty arguments:

```
PLAY "O4EDCDEEERL",,,"O2CGEGCGEGL"
```

An example using voice 2 and voice 5:

```
PLAY ,"O5T2IGAGFEDCEGO6.QCL",,,"O3T2.QG.B O4ICO3GE.QCL"
```

RPLAY(voice) tests whether music is playing on the given voice, and returns 1 if it is playing or 0 if it is not.

One caveat to be aware of is that BASIC strings have a maximum length of 255 bytes. If your melody needs to exceed this length, consider breaking up your melody into several strings, then use **RPLAY(voice)** to assess when your first string has finished and then play the next string.

Instrument envelope slots may be modified by using the **ENVELOPE** statement. The default settings for the envelopes are on page 100.

Remarks: The **PLAY** statement makes use of an interrupt driven routine that starts parsing the string and playing the melody. Program execution continues with the next statement, and will not block until the melody has finished. This is different to the Commodore 128, which stops program execution during playback.

The 6 voice channels used by the **PLAY** command (on SID1+SID3) are distinct to the 6 channels used by the **SOUND** command (on SID2+SID4). Sound effects will not interrupt music, and vice versa.

Example: Using **PLAY**

```
 5 REM *** SIMPLE LOOPING EXAMPLE ***
10 ENVELOPE 9,10,5,10,5,0,300
20 VOL 8,8
30 TEMPO 30
40 PLAY "O5T9HCIDCDEHCG IGAGFEFDEWCL", "O2T0QCGEGCGEG DBGB CGEGL"
```

```
 5 REM *** MODULATION + PORTAMENTO EXAMPLE ***
10 TEMPO 20
20 M$ = "M5 T2O5P0QD P5FP0RP5QG .AI#AQA HGQE.C IDQE HFQD .DI#CQD HEQ#CQO4HA"
30 M$ = M$ + "O5QDHFQG.AI#AQA HGQE.C IDQEFED#CO4BO5#C DO4AFD P0R L"
40 B$ = "T0QRO2H.D.F.CO1.A.#A.G.A QAIO2AGFE H.D.F.CO1.A.#A.AO2 .D DL"
50 PLAY M$,B$
```

POINTER

Token: $CE $0A

Format: **POINTER**(variable)

Returns: The current address of a variable or an array element as a 32-bit pointer.

For string variables, it is the address of the string descriptor, not the string itself. The string descriptor consists of three bytes: length, string address low, string address high. The string address is an offset in bank 1.

For number-type scalar variables, it is the address of the value. The format depends on the type. A byte variable (`A&`) is one byte, in a "two's complement" signed integer format. An integer variable (`A%`) is two bytes, with the least significant byte first. A real variable (`A`) is five bytes, in a compact floating point number format.

To get the address of an array, use **POINTER** with the first element of the array (index `0` in each dimension). Array elements are stored consecutively, in the format of the scalar record, with the left-most index using the shortest stride. For example, an array dimensioned as `DIM A%(3,2)` starts at address `POINTER(A%(0,0))`, has two-byte records, and is ordered as:

(0,0) (1,0) (2,0) (3,0) (0,1) (1,1) (2,1) (3,1) ...

Remarks: The address values of arrays and their elements are constant while the program is executing.
However, the addresses of strings (not their descriptors) may change at any time due to "garbage collection."

Example: Using **POINTER**

```
10 BANK 0                :REM SCALARS ARE IN BANK 0
20 H$="HELLO"            :REM ASSIGN STRING TO H$
30 P=POINTER(H$)         :REM GET DESCRIPTOR ADDRESS
40 PRINT "DESCRIPTOR AT: $";HEX$(P)
50 L=PEEK(P):SP=WPEEK(P+1) :REM LENGTH & STRING POINTER
60 PRINT "LENGTH = ";L     :REM PRINT LENGTH
70 BANK 1                :REM STRINGS ARE IN BANK 1
80 FOR I%=0 TOL-1:PRINT PEEK(SP+I%);:NEXT:PRINT
90 FOR I%=0 TOL-1:PRINT CHR$(PEEK(SP+I%));:NEXT:PRINT

RUN
DESCRIPTOR AT: $FD75
LENGTH =  5
 72  69  76  76  79
HELLO
```

POKE

Token: $97

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

Returns: Writes one or more bytes into memory or memory mapped I/O, starting at **address**.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

If **value** is in the range of 0 - 255, this is poked into memory, otherwise the low byte of value is used. So a command like `POKE AD,V AND 255` can be written as `POKE AD,V` because **POKE** uses the low byte anyway.

Remarks: The address is incremented for each data byte, so a memory range can be written to with a single **POKE**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **POKE**

```
10 BANK 128        :REM SELECT SYSTEM BANK
20 POKE $02F8,0,24 :REM SET USR VECTOR TO $1800
```

POLYGON

Token: $FE $2F

Format: **POLYGON** x, y, xrad, yrad, sides [{, drawsides, subtend, angle, solid}]

Usage: Bitmap graphics: 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.

sides number of polygon sides.

drawsides sides to draw.

subtend draw line from centre to start (1).

angle start angle.

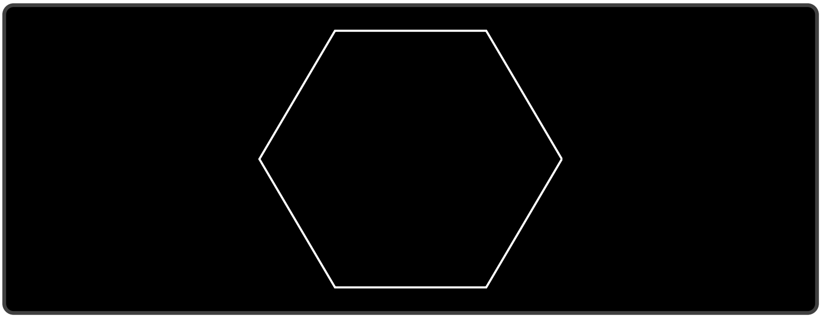
solid fill (1) or outline (0).

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

Example: Using **POLYGON**

```
100 SCREEN 320,200,1          :REM OPEN 320 x 200 SCREEN
110 POLYGON 160,100,40,40,6   :REM DRAW HONEYCOMB
120 GETKEY A$                 :REM WAIT FOR KEY
130 SCREEN CLOSE              :REM CLOSE GRAPHICS SCREEN
```

Results in:



POS

Token: $B9

Format: **POS**(dummy)

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)

Returns: The position of a paddle peripheral.

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 anticlockwise limit.

A value greater than 255 indicates that the fire button is also being pressed.

Remarks: Analogue paddles are noisy and inexact. The range may be less than 0 – 255 and there could be some jitter in the values returned from **POT**.

Paddles made for Atari game consoles return different values from paddles made for Commodore computers. Commodore paddles provide more accurate values in the 0 – 255 range.

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   : REM PADDLE #1 VALUE
```

PRINT

Token: $99

Format: **PRINT** arguments

Usage: Prints a series of values formatted to the current output stream, typically the screen.

Values are formatted based on their type. For more control over formatting, see **PRINT USING**.

The following expressions and characters can appear in the argument list:

- **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 in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** the string may consist of printable characters and control codes. Printable characters are printed at the cursor position. Control codes are executed.
- ; (semicolon) separates arguments of the list. It does not print any characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.
- , (comma) moves the cursor to the next tab position.

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

CMD can be used to redirect printed characters to a device other than the screen.

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: Prints a series of values formatted to the device assigned to **channel**.

Values are formatted based on their type. For more control over formatting, see **PRINT# USING**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

The following argument types are evaluated:

- **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 in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** may consist of printable characters and control codes. Printable characters are printed at the cursor position, while control codes are executed.
- ; (semicolon) separates arguments of the list. It does not print any characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.
- , (comma) moves the cursor to the next tab position.

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

Example: Using **PRINT#** to write a file to drive 8:

```
10 DOPEN#2,"TABLE",W,U8
20 FOR I=1 TO 10    : REM START LOOP
30 PRINT#2,I,I*I,SQR(I)
40 NEXT
50 DCLOSE#2
```

You can confirm that the file **'TABLE'** has been written by typing `DIR "TA*"`, and then view the contents of the file by typing `TYPE "TABLE"`.

PRINT USING

Token: $98 $FB or $99 $FB

Format: **PRINT**[**#** channel**,**] **USING** format**;** argument

Usage: Prints a series of values formatted using a pattern to the current output stream (typically the screen) or an output channel.

The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can be done in either CBM style, providing a pattern such as `###.##` or in C style using a <width.precision> specifier, such as `%3D %7.2F %4X` .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (`###`), asterisks will be used instead.

Remarks: The format string is applied for one argument only, but it is possible to append more with **USING format;argument** sequences.

argument 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. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Examples: Using **PRINT# USING**

```
PRINT USING "##.##";π, USING " [%6.4F] ";SQR(2)
 3.14 [1.4142]

PRINT USING " < # # # > ";12*31
 < 3 7 2 >

PRINT USING "###"; "ABCDE"
ABC

PRINT USING ">###"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="###,###,###.#":PRINT USING A$;1E8/3
 33,333,333.3
```

RCOLOR

Token: $CD

Format: **RCOLOR(**colour source**)**

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

Colour sources are:

- **0** background colour (VIC $D021).
- **1** text colour ($F1).
- **2** highlight colour ($2D8).
- **3** border colour (VIC $D020).

Example: Using **RCOLOR**

```
10 C = RCOLOR(3)  : REM C = colour index of border colour
```

RCURSOR

Token: $FE $42

Format: **RCURSOR** {colvar**,** rowvar}

Usage: Reads the current cursor column and row into variables.

Remarks: The row and column values start at zero, where the left-most column is zero, and the top row is zero.

Example: Using **RCURSOR**

```
100 CURSOR ON,20,10
110 PRINT "[HERE]";
120 RCURSOR X,Y
130 PRINT " COL:";X;" ROW:";Y

RUN

                    [HERE] COL: 26  ROW: 10
```

READ

Token: $87

Format: **READ** variable [, variable ...]

Usage: Reads values from **DATA** statements into variables.

variable list Any legal variables.

All types of constants (integer, real, and strings) can be read, but not expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility 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.

RESTORE may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

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 "MEGA65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

RECORD

Token: $FE $12

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

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

channel number, which was given to a previous call of commands such as **DOPEN**, or **OPEN**.

record target record (1 – 65535).

byte byte position in record.

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

RECORD positions the file pointer to the specified record number. If this record number does not exist and there is enough space on the disk which **RECORD** is writing to, the file is expanded to the requested record count by adding empty records. When this occurs, the disk status will give the message `RECORD NOT PRESENT`, but this is not an error!

After a call of **INPUT#** or **PRINT#**, the file pointer will proceed to the next record position.

Remarks: The Commodore disk drives have a bug in their DOS, which can destroy data by using relative files. A recommended workaround is to use the command **RECORD** twice, before and after the I/O operation.

Example: Using **RECORD**

```
100 DOPEN#2,"DATA BASE",L240 :REM OPEN OR CREATE
110 FOR I%=1 TO 20           :REM WRITE LOOP
120 PRINT#2,"RECORD #";I%    :REM WRITE RECORD
130 NEXT I%                  :REM END LOOP
140 DCLOSE#2                 :REM CLOSE FILE
150                          :REM NOW TESTING
160 DOPEN#2,"DATA BASE",L240 :REM REOPEN
170 FOR I%=20 TO 2 STEP -2   :REM READ FILE BACKWARDS
180 RECORD#2,I%              :REM POSITION TO RECORD
190 INPUT#2,A$               :REM READ RECORD
200 PRINT A$;:IF I% AND 2 THEN PRINT
210 NEXT I%                  :REM LOOP
220 DCLOSE#2                 :REM CLOSE FILE

RUN
RECORD # 20 RECORD # 18
RECORD # 16 RECORD # 14
RECORD # 12 RECORD # 10
RECORD # 8 RECORD # 6
RECORD # 4 RECORD # 2
```

REM

Token: $8F

Format: **REM**

Usage: Ignores all subsequent characters on a line of BASIC code, as a code comment.

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 brackets, e.g. `(FI$)`.

new is either a quoted string, e.g. `"BACKUP"` or a string expression in brackets, e.g. `(FS$)`

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **RENAME** is executed in the DOS of the disk drive. It can rename all regular file types (**PRG**, **REL**), **SEQ**, **USR**. The old file must exist, and the new file must not exist. Only single files can be renamed, wildcard characters such as '*' 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**,** range}]

Usage: Renumbers lines of a BASIC program.

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

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

range line range to renumber. The default values are from first to last line.

RENUMBER executes in either space conserving mode or optimisation mode. Optimisation mode removes space characters before line numbers, thereby reducing code size and decreasing execution time, while the space conserving leaves spaces untouched. Optimisation mode is triggered by typing the first argument, (the **new** starting number), adjacent to the keyword **RENUMBER** with no space in between.

RENUMBER changes all line numbers in the chosen range and also changes all references in statements that use **GOSUB**, **GOTO**, **RESTORE**, **RUN**, **TRAP**, etc.

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

RENUMBER may be called with 0 – 3 parameters. Unspecified parameters use their default values.

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

RENUMBER can only be used in direct mode.

This command temporarily uses memory in banks 4 and 5, and may overwrite anything stored there.

Examples: Using **RENUMBER**

```
RENUMBER                :REM SPACE CONSERVING, NUMBERS WILL BE 10,20,30,...
RENUMBER 100,5          :REM SPACE CONSERVING, NUMBERS WILL BE 100,105,110,115,...
RENUMBER601,1,500       :REM OPTIMISATION, RENUMBER STARTING AT 500 TO 601,602,...
RENUMBER 100,5,120-180  :REM SPACE CONSERVING RENUMBER LINES 120-180 TO 100,105,...

10 GOTO  20
20 GOTO 10
RENUMBER 100,10         :REM SPACE CONSERVING
100 GOTO  110
110 GOTO 100
RENUMBER100,10          :REM OPTIMISATION
100 GOTO110
110 GOTO100
```

RESTORE

Token: $8C

Format: **RESTORE** [line]

Usage: Sets the internal pointer for **READ** from **DATA** statements.

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

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

Example: Using **RESTORE**

```
10 DATA 3,1,4,1,5,9,2,6
20 DATA "MEGA65"
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: Resumes normal program execution in a **TRAP** routine, after handling an error.

RESUME with no parameters attempts to re-execute the statement that caused the error. The **TRAP** routine should have examined and corrected the issue where the error occurred.

line line number to resume program execution at.

NEXT resumes execution following the statement that caused the error. This could be the next statement on the same line (separated with a colon ':'), or the statement on the next line.

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 that 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 from to call the handler.

Example: Using **RETURN**

```
 10 SCNCLR          :REM CLEAR SCREEN
 20 FOR I=1 TO 20   :REM DEFINE LOOP
 30 GOSUB 100       :REM CALL SUBROUTINE
 40 NEXT I          :REM LOOP
 50 END             :REM END OF PROGRAM
100 CURSOR ON,I,I,0 :REM ACTIVATE AND POSITION CURSOR
110 PRINT "X";      :REM PRINT X
120 SLEEP 0.5       :REM WAIT 0.5 SECONDS
130 CURSOR OFF      :REM SWITCH BLINKING CURSOR OFF
140 RETURN          :REM RETURN TO CALLER
```

RGRAPHIC

Token: $CC

Format: **RGRAPHIC**(screen, parameter)

Returns: Bitmap graphics: the status of a given graphic screen parameter.

Parameter	Description
0	Open (1), Closed (0), or Invalid (>1)
1	Width (0=320, 1=640)
2	Height (0=200, 1=400)
3	Depth (1 – 8 Bitplanes)
4	Bitplanes Used (Bitmask)
5	Bank 4 Blocks Used (Bitmask)
6	Bank 5 Blocks Used (Bitmask)
7	Drawscreen # (0 – 3)
8	Viewscreen # (0 – 3)
9	Drawmodes (Bitmask)
10	pattern type (bitmask)

Example: Using **RGRAPHIC**

```
 10 GRAPHIC CLR          :REM INITIALISE
 20 SCREEN DEF 0,1,0,4  :REM SCREEN 0:640 X 200 X 4
 30 SCREEN OPEN 0       :REM OPEN
 40 SCREEN SET 0,0      :REM DRAW = VIEW = 0
 50 SCNCLR 0            :REM CLEAR
 60 PEN 0,1             :REM SELECT COLOUR
 70 LINE 0,0,639,199    :REM DRAW LINE
 80 FOR I=0 TO 10:A(I)=RGRAPHIC(0,I) :NEXT
 90 SCREEN CLOSE 0
100 FOR I=0 TO  6:PRINT I;A(I):NEXT :REM PRINT INFO

RUN
 0  1
 1  1
 2  0
 3  4
 4  15
 5  15
 6  15
```

RIGHT$

Token: $C9

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

Returns: A string containing the last **n** characters from **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** x variable**,** y variable**,** button variable

Usage: Reads mouse position and button status.

x variable numeric variable where the x-position will be stored.

y variable numeric variable where the y-position will be stored.

button variable numeric variable receiving button status.
left button sets bit 7, while right button sets bit 0.

Coordinates are reported to be compatible with sprite coordinates, limited to the visible screen inside the border. In the top-left corner, X=24 and Y=50.

Value	Status
0	No Button
1	Right Button
128	Left Button
129	Both Buttons

RMOUSE places -1 into all variables if the mouse is not connected or disabled.

Remarks: 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)

Returns: A pseudo-random number.

This is called a "pseudo-random" number as computers cannot generate numbers that are truly random. Pseudo-random numbers are derived mathematically from another number called a "seed" that generates reproducible sequences. **type** determines which seed is used:

- **type = 0** use system clock.
- **type < 0** use the value of **type** as seed.
- **type > 0** derive a new random number from previous one.

Remarks: Seeded random number sequences produce the same sequence for identical seeds.

The algorithm is initially seeded from the Real-Time Clock and other factors during boot, so `RND(1)` is unlikely to return the same sequence twice. This is unlike the Commodore 64, which always used the same initial seed. If **RND()** is ever called with a negative value, that value is used as a new seed, and sequences generated by `RND(1)` become predictable. Use `RND(0)` to re-seed with an unpredictable value.

Each call to `RND(0)` generates a new seed based on the system clock and other factors. Calling `RND(0)` repeatedly tends to produce a better distribution of values than on a Commodore 64 due to the precision of the sources of the seed.

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
```

RPALETTE

Token: $CE $0D

Format: **RPALETTE(**screen**,** index**,** rgb**)**

Returns: The red, green, or blue value of a palette colour index.

screen screen number (0 – 3), or a negative value to select one of the four system palettes: -1 for system palette 0 (the default system palette), -2 for system palette 1, -3 for palette 2, or -4 for palette 3.

index palette colour index.

rgb (0: red, 1: green, 2:blue).

Example: Using **RPALETTE**

```
 10 SCREEN 320,200,4    :REM DEFINE AND OPEN SCREEN
 20 R = RPALETTE(0,3,0) :REM GET RED
 30 G = RPALETTE(0,3,1) :REM GET GREEN
 40 B = RPALETTE(0,3,2) :REM GET BLUE
 50 SCREEN CLOSE        :REM CLOSE SCREEN
 60 PRINT "PALETTE INDEX 3 RGB =";R;G;B

RUN
PALETTE INDEX 3 RGB = 0  15  15
```

RPEN

Token: $D0

Format: **RPEN**(n)

Returns: The colour index of pen **n**.

n pen number (0 – 2), where:

- 0 draw pen
- 1 erase pen
- 2 outline pen

Example: Using **RPEN**

```
 10 GRAPHIC CLR          :REM INITIALISE
 20 SCREEN DEF 0,1,0,4  :REM SCREEN 0:640 X 200 X 4
 30 SCREEN OPEN 0       :REM OPEN
 40 SCREEN SET 0,0      :REM DRAW = VIEW = 0
 50 SCNCLR 0            :REM CLEAR
 60 PEN 0,1             :REM SELECT COLOUR
 70 X = RPEN(0)
 80 Y = RPEN(1)
 90 C = RPEN(2)
100 SCREEN CLOSE 0
110 PRINT "DRAW PEN COLOUR = ";X
RUN
DRAW PEN COLOUR =  1
```

RPLAY

Token: $FE $0F

Format: **RPLAY**(voice)

Returns: Tests whether music is playing on the given voice channel.

voice the voice channel to assess, ranging from 1 to 6.

Returns 1 if music is playing on the channel, otherwise 0.

Example: Using **RPLAY**:

```
10 PLAY "O4ICDEFGABO5CR","O2QCGEGCO1GCR"
30 IF RPLAY(1) OR RPLAY(2) THEN GOTO 30: REM WAIT FOR END OF SONG
```

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 gets accumulator value.

xreg gets X register value.

yreg gets Y register value.

zreg gets Z register value.

sreg gets status register value.

Remarks: The register values after a **SYS** call are stored in system memory. This is how **RREG** is able to retrieve them.

Example: Using **RREG**:

```
10 POKE $1800, $18, $8A, $65, $06, $60
20 REM        CLC  TXA  ADC   06  RTS
30 SYS $1800, 77, 11  : REM A=77 X=11
40 RREG AC,X,Y,Z,S
50 PRINT "REGISTER:";AC;X;Y;Z;S
```

RSPCOLOR

Token: $CE $07

Format: **RSPCOLOR**(n)

Returns: The colour setting of a multi-colour sprite colour.

n sprite multi-colour number:

- **1** get multi-colour # 1.
- **2** get multi-colour # 2.

Remarks: Refer to **SPRITE** and **SPRCOLOR** for more information.

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
```

RSPEED

Token: $CE $0E

Format: **RSPEED**(n)

Returns: The current CPU clock in MHz.

n numeric dummy argument, which is ignored.

Remarks: **RSPEED(n)** will not return the correct value if `POKE 0,65` has previously been used to enable the highest speed (40MHz).

Refer to the **SPEED** command for more information.

Example: Using **RSPEED**:

```
10 X=RSPEED(0)      :REM GET CLOCK
20 IF X=1  THEN PRINT "1 MHZ"   :GOTO 50
30 IF X=3  THEN PRINT "3.5 MHZ" :GOTO 50
40 IF X=40 THEN PRINT "40 MHZ"
50 END
```

RSPPOS

Token: $CE $05

Format: **RSPPOS**(sprite, n)

Returns: A sprite's position or speed.

sprite sprite number.

n sprite parameter to retrieve:

- **0** X position.
- **1** Y position.
- **2** speed.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

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**)**

Returns: A sprite parameter.

sprite sprite number (0 – 7).

n the sprite parameter to return (0 – 5):

- **0** turned on (0 or 1) A 0 means the sprite is off.
- **1** foreground colour (0 – 15).
- **2** background priority (0 or 1).
- **3** x-expanded (0 or 1). 0 means it's not expanded.
- **4** y-expanded (0 or 1). 0 means it's not expanded.
- **5** multi-colour (0 or 1). 0 means it's not multi-colour.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

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
40 BP = RSPRITE(1,2)   :REM SPRITE 1 BACKGROUND PRIORITY
50 XE = RSPRITE(1,3)   :REM SPRITE 1 X EXPANDED ?
60 YE = RSPRITE(1,4)   :REM SPRITE 1 Y EXPANDED ?
70 MC = RSPRITE(1,5)   :REM SPRITE 1 MULTI-COLOUR ?
```

RUN

Token: $8A

Format: **RUN** [line number]
RUN filename [**,D** drive] [**,U** unit] ↑ filename

Usage: Runs the BASIC program in memory, or loads and runs a program from disk.

If a filename is given, the program file is loaded into memory and run, otherwise the program that is currently in memory will be used instead.

The ↑ can be used as shortcut, if used in direct mode at the leftmost column. It can be used to load and run a program from a dir listing by moving the cursor to the row with the filename, typing the ↑ at the start of the row and pressing return. Characters before and after the quoted filename, will be ignored (like the PRG for example).

line number an existing line number of the program in memory to run from.

filename either a quoted string, e.g. `"PROG"` or a string expression in brackets, 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 such as the 1541, 1571, or 1581.

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

RUN first resets all internal pointers to their default values. Therefore, there will be no variables, arrays or strings defined. The run-time stack is also reset, and the table of open files is cleared.

Remarks: To start or continue program execution without resetting everything, use **GOTO** instead.

Examples: Using **RUN**

```
RUN "FLIGHTSIM" :REM LOAD AND RUN PROGRAM FLIGHTSIM
RUN 1000        :REM RUN PROGRAM IN MEMORY, START AT LINE# 1000
RUN             :REM RUN PROGRAM IN MEMORY
```

RWINDOW

Token: $CE $09

Format: **RWINDOW**(n)

Returns: A parameter of the current text window.

n the screen parameter to retrieve:

- **0** width of current text window.
- **1** height of current text window.
- **2** number of columns on screen (40 or 80).

Remarks: Older versions of **RWINDOW** reported the width - 1 and the height - 1 for arguments 0 and 1.

Refer to the **WINDOW** command for more information.

Example: Using **RWINDOW**:

```
10 W = RWINDOW(2)          :REM GET SCREEN WIDTH
20 IF W=80 THEN BEGIN      :REM IS 80 COLUMNS MODE ACTIVE?
30   PRINT CHR$(27)+"X";   :REM YES, SWITCH TO 40COLUMNS
40 BEND
```

SAVE

Token: $94

Format: **SAVE** filename [, unit]
← filename [, unit]

Usage: Saves a BASIC program to a file of type **PRG**.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

The maximum length of the filename is 16 characters, not counting the optional save and replace character '@' and the in-file drive definition. If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS. The filename may be preceded 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 from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **SAVE** is obsolete, implemented only for backwards compatibility. **DSAVE** should be used instead. The shortcut symbol ← is next to **1**. Can only be used in direct mode.

Examples: Using **SAVE**

```
SAVE "ADVENTURE"
SAVE "ZORK-I",8
SAVE "1:DUNGEON",9
```

SAVEIFF

Token: $FE $44

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

Usage: Bitmap graphics: saves the current graphics screen to a disk file in **IFF** format.

The IFF (Interchange File Format) is supported by many different applications and operating systems. **SAVEIFF** saves the image, the palette and resolution parameters.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$). 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 not recommended to use this option on 1541 and 1571 drives, as they contain a "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 such as the 1541, 1571, or 1581.

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

Remarks: Files saved with **SAVEIFF** can be loaded with **LOADIFF**. Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as Amiga OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netbpm** package.

Example: Using **SAVEIFF**

```
10 SCREEN 320,200,2          :REM SCREEN #0 320 X 200 X 2
20 PEN 1                     :REM DRAWING PEN COLOUR 1 (WHITE)
30 LINE 25,25,295,175        :REM DRAW LINE
40 SAVEIFF "LINE-EXAMPLE",U8 :REM SAVE CURRENT VIEW TO FILE
50 SCREEN CLOSE              :REM CLOSE SCREEN AND RESTORE PALETTE
```

SCNCLR

Token: $E8

Format: **SCNCLR** [colour]

Usage: Clears a text window or bitmap graphics 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 with the given **colour**.

Example: Using **SCNCLR**:

```
1 REM SCREEN EXAMPLE 2
10 GRAPHIC CLR          :REM INITIALISE
20 SCREEN DEF 1,0,0,2   :REM SCREEN #1 320 X 200 X 2
30 SCREEN OPEN 1        :REM OPEN SCREEN 1
40 SCREEN SET 1,1       :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0         :REM CLEAR SCREEN
60 PALETTE 1,1,15,15,15 :REM DEFINE COLOUR 1 AS WHITE
70 PEN 0,1              :REM DRAWING PEN
80 LINE 25,25,295,175   :REM DRAW LINE
90 SLEEP 10             :REM WAIT FOR 10 SECONDS
100 SCREEN CLOSE 1      :REM CLOSE SCREEN AND RESTORE PALETTE
```

SCRATCH

Token: $F2

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

Usage: Erases ("scratches") a disk file.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

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

Remarks: **SCRATCH filename** is a synonym of **ERASE filename** and **DELETE filename**.

In direct mode the success and the number of erased files is printed. The second to last number from the message contains the number of successfully erased files,

Examples: Using **SCRATCH**

```
SCRATCH "DRM",U9 :REM ERASE FILE DRM ON UNIT 9
01, FILES SCRATCHED,01,00
SCRATCH "OLD*"   :REM ERASE ALL FILES BEGINNING WITH "OLD"
01, FILES SCRATCHED,04,00
SCRATCH "R*=PRG" :REM ERASE PROGRAM FILES STARTING WITH 'R'
01, FILES SCRATCHED,09,00
```

SCREEN

Token: $FE $2E

Format:
SCREEN [screen**,**] width**,** height**,** depth
SCREEN CLR colour
SCREEN DEF width flag**,** height flag**,** depth
SCREEN SET drawscreen**,** viewscreen
SCREEN OPEN [screen]
SCREEN CLOSE [screen]

Usage: Bitmap graphics: manages a graphics screen.

There are two approaches available when preparing the screen for the drawing of graphics: a simplified approach, and a detailed approach.

Simplified approach:

The first version of **SCREEN** (which has pixel units for width and height) is the easiest way to start a graphics screen, and is the preferred method if only a single screen is needed (i.e., a second screen isn't needed for double buffering). This does all of the preparatory work for you, and will call commands such as **GRAPHIC CLR**, **SCREEN CLR**, **SCREEN DEF**, **SCREEN OPEN** and, **SCREEN SET** on your behalf. It takes the following parameters:

SCREEN [screen,] width, height, depth

- **screen** the screen number (0 - 3) is optional. If no screen number is given, screen 0 is used. To keep this approach as simple as possible, it is suggested to use the default screen 0.
- **width** 320 or 640 (default 320)
- **height** 200 or 400 (default = 200)
- **depth** 1..8 (default = 8), colours = 2 ^depth.

The argument parser is error tolerant and uses default values for width (320) and height (200) if the parsed argument is not valid.

This version of **SCREEN** starts with a predefined palette and sets the background to black, and the pen to white, so drawing can start immediately using the default values.

On the other hand, the detailed approach will require the setting of palette colours and pen colour before any drawing can be done.

The **colour** value must be in the range of 0 to 15. See appendix 6 on page 297 for the list of colours in the default system palette.

When you are finished with your graphics screen, simply call **SCREEN CLOSE** [screen] to return to the text screen.

Detailed approach:

The other versions of **SCREEN** perform special actions, used for advanced graphics programs that open multiple screens, or require "double buffering". If you have chosen the simplified approach, you will not require any of these versions below, apart from **SCREEN CLOSE**.

SCREEN CLR colour (or **SCNCLR** colour)
Clears the active graphics screen by filling it with **colour**.

SCREEN DEF screen, width flag, height flag, depth
Defines resolution parameters for the chosen screen. The width flag and height flag indicate whether high resolution (1) or low resolution (0) is chosen.

- **screen** screen number 0 – 3
- **width flag** 0 – 1 (0:320, 1:640 pixel)
- **height flag** 0 – 1 (0:200, 1:400 pixel)
- **depth** 1 – 8 (2 – 256 colours)

Note that the width and height values here are **flags**, and **not pixel units**.

SCREEN SET drawscreen, viewscreen
Sets screen numbers (0 – 3) for the drawing and the viewing screen, i.e., while one screen is being viewed, you can draw on a separate screen and then later flip between them. This is what's known as double buffering.

SCREEN OPEN screen
Allocates resources and initialises the graphics context for the selected **screen** (0 – 3). An optional variable name as a further argument, gets the result of the command that can be tested afterwards for success.

SCREEN CLOSE [screen]
Closes **screen** (0 – 3) and frees resources. If no value is given, it will default to 0. Also note that upon closing a screen, **PALETTE RESTORE** is automatically performed for you.

Examples: Using **SCREEN**:

```
 5 REM *** SIMPLIFIED APPROACH ***
10 SCREEN 320,200,2    :REM SCREEN #0: 320 X 200 X 2
20 PEN 1               :REM DRAWING PEN COLOUR = 1 (WHITE)
30 LINE 25,25,295,175  :REM DRAW LINE
40 GETKEY A$           :REM WAIT KEYPRESS
50 SCREEN CLOSE        :REM CLOSE SCREEN 0 (RESTORE PALETTE)
```

```
  5 REM *** DETAILED APPROACH ***
 10 GRAPHIC CLR         :REM INITIALISE
 20 SCREEN DEF 1,0,0,2  :REM SCREEN #1: 320 X 200 X 2
 30 SCREEN OPEN 1       :REM OPEN SCREEN 1
 40 SCREEN SET 1,1      :REM USE SCREEN 1 FOR RENDERING AND VIEWING
 50 SCREEN CLR 0        :REM CLEAR SCREEN
 60 PALETTE 1,1,15,15,15:REM DEFINE COLOUR 1 AS WHITE
 70 PEN 0,1             :REM DRAWING PEN
 80 LINE 25,25,295,175  :REM DRAW LINE
 90 SLEEP 10            :REM WAIT 10 SECONDS
100 SCREEN CLOSE 1      :REM CLOSE SCREEN 1 (RESTORE PALETTE)
```

SET

Token: $FE $2D

Format:
SET DEF unit
SET DISK old **TO** new
SET VERIFY <**ON** | **OFF**>

Usage: **SET DEF** redefines the default unit for disk access, which is initialised to 8 by the DOS. Commands that do not explicitly specify a unit will use this default unit.

SET DISK is used to change the unit number of a disk drive temporarily.

SET VERIFY enables or disables the DOS verify-after-write mode for 3.5 drives.

Remarks: These settings are valid until a reset or shutdown.

Examples: 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)
SET VERIFY ON   :REM ACTIVATE VERIFY-AFTER-WTITE MODE
```

SETBIT

Token: $FE $2D $FE $4E

Format: **SETBIT** address**,** bit number

Usage: Sets a single bit at the **address**.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

The **bit number** is a value in the range of 0 - 7.

A bank value > 127 is used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **SETBIT**

```
10 BANK 128         :REM SELECT SYSTEM MAPPING
20 SETBIT $D011,6   :REM ENABLE EXTENDED BACKGROUND MODE
30 SETBIT $D01B,0   :REM SET BACKGROUND PRIORITY FOR SPRITE 0
```

SGN

Token: $B4

Format: **SGN**(numeric expression)

Returns: The sign of a numeric expression, as a number.

- **-1** negative argument.
- **0** zero.
- **1** 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: $BF

Format: **SIN**(numeric expression)

Returns: The sine of an angle.

The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: A value in units of degrees can be converted to radians by multiplying it with $\pi/180$.

Examples: Using **SIN**

```
PRINT SIN(0.7)
 .644217687

X=30:PRINT SIN(X * π / 180)
 .5
```

SLEEP

Token: $FE $0B

Format: **SLEEP** seconds

Usage: Pauses execution for the given duration.

The argument is a positive floating point number of seconds. The precision is 1 microsecond.

Remarks: Pressing RUN STOP interrupts the sleep.

Example: Using **SLEEP**

```
20 SLEEP 10     :REM WAIT 10 SECONDS
40 SLEEP 0.0005 :REM SLEEP 500 MICRO SECONDS
50 SLEEP 0.01   :REM SLEEP  10 MILLI SECONDS
60 SLEEP DD     :REM TAKE SLEEP TIME FROM VARIABLE DD
70 SLEEP 600    :REM SLEEP 10 MINUTES
```

SOUND

Token: $DA

Format: **SOUND** voice, freq, dur [{, dir, min, sweep, wave , pulse}]
SOUND CLR

Usage: **SOUND** plays a sound effect.

voice voice number (1 – 6).

freq frequency (0 – 65535).

dur duration in jiffies (0 – 32767). The duration of a jiffy depends on the display standard. There are 50 jiffies per second with PAL, 60 per second with NTSC.

dir direction (0:up, 1:down, 2:oscillate).

min minimum frequency (0 – 65535).

sweep sweep range (0 – 65535).

wave waveform (0:triangle, 1:sawtooth, 2:square, 3:noise).

pulse pulse width (0 – 4095).

SOUND CLR silences all sound from **SOUND** and **PLAY**, and resets the sound system and all parameters.

Remarks: **SOUND** starts playing the sound effect and immediately continues with the execution of the next BASIC statement while the sound effect is played. This enables the showing of graphics or text and playing sounds simultaneously.

SOUND uses SID2 (for voices 1 to 3) and SID4 (for voices 4 to 6) of the 4 SID chips of the system. By default, SID1 and SID2 are slightly right-biased and SID3 and SID4 are slightly left-biased in the stereo mix.

The 6 voice channels used by the **SOUND** command (on SID2+SID4) are distinct to the 6 channels used by the **PLAY** command (on SID1+SID3). Sound effects will not interrupt music, and vice versa.

Examples: Using **SOUND**

```
IF PEEK($D06F) AND $80 THEN J = 60: ELSE J = 50 :REM J IS JIFFIES PER SECOND
SOUND 1, 7382, J                 :REM PLAY SQUARE WAVE ON VOICE 1 FOR 1 SECOND
SOUND 2,  800, J*60              :REM PLAY SQUARE WAVE ON VOICE 2 FOR 1 MINUTE
SOUND 3, 4000, 120, 2, 2000, 400, 1 :REM PLAY SWEEPING SAWTOOTH WAVE ON VOICE 3

SOUND CLR   :REM SILENCE SOUND, RESET PARAMETERS
```

SPC

Token: $A6

Format: **SPC**(columns)

Returns: As an argument to **PRINT**, a string of cursor-right PETSCII codes, suitable for printing to advance the cursor the given number of columns.

Printing this is similar to pressing → **<column>** times.

This is not a real function and does not generate a string. It can only be used as an argument to **PRINT**.

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
```

SPEED

Token: $FE $26

Format: **SPEED** [speed]

Usage: Sets the CPU clock speed to 1MHz, 3.5MHz, or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.
- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **SPEED** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

SPEED is a synonym of **FAST**.

SPEED has no effect if `POKE 0,65` has previously been used to set the CPU to 40MHz.

Example: Using **SPEED**

```
10 SPEED     :REM SET SPEED TO MAXIMUM (40 MHZ)
20 SPEED 1   :REM SET SPEED TO 1 MHZ
30 SPEED 3   :REM SET SPEED TO 3.5 MHZ
40 SPEED 3.5 :REM SET SPEED TO 3.5 MHZ
```

SPRCOLOR

Token: $FE $08

Format: **SPRCOLOR** [{mc1, mc2}]

Usage: Sets multi-colour sprite colours.

SPRITE, which sets the attributes of a sprite, only sets the foreground colour. For setting the additional two colours of multi-colour sprites, use **SPRCOLOR** instead.

Remarks: The colours used with **SPRCOLOR** will affect all sprites. Refer to the **SPRITE** command for more information.

The final argument to **SPRITE** enables multi-colour mode for the 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 CLR**
SPRITE LOAD filename [**,D** drive] [**,U** unit]
SPRITE SAVE filename [**,D** drive] [**,U** unit]
SPRITE num [{**,** switch**,** colour**,** prio**,** expx**,** expy**,** mode}]

Usage: **SPRITE CLR** clears all sprite data and sets all pointers and attributes to their default values.

SPRITE LOAD loads sprite data from **filename** to sprite memory.

SPRITE SAVE saves sprite data from sprite memory to **filename**.

filename the name of a file. Either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

The last form switches a sprite on or off and sets its attributes:

num sprite number

switch 1: on, 0: off

colour sprite foreground colour

prio 0: sprite in front of text, 1: sprite behind text

expx 1: sprite X expansion

expy 1: sprite Y expansion

mode 1: multi-colour sprite

Remarks: **SPRCOLOR** must be used to set additional colours for multi-colour sprites (mode = 1).

Example: Using **SPRITE**:

```
2290 CLR:SCNCLR:SPRITE CLR
2300 SPRITE LOAD "DEMOSPRITES1"
2320 FORI=0TO7: C=I: IFC=6THENC=8
2330 MOVSPR I, 60+30*I,0 TO 60+30*I,65+20*I, 3:SPRITE I,1,C,,1,1:NEXT: SLEEP3
2340 FORI=0TO7: SPRITE I,,,,0,0 :NEXT: SLEEP3: SPRITE CLR
2350 FORI=0TO7: MOVSPR I,45*I#5 :NEXT: FORI=0TO7: SPRITE I,1: NEXT
2360 FORI=0TO7:X=60+30*I:Y=65+20*I:DO
2370 LOOPUNTIL(X=RSPPOS(I,.))AND(Y=RSPPOS(I,1)):MOVSPRI,.#.:NEXT
```

SPRSAV

Token: $FE $16

Format: **SPRSAV** source, destination

Usage: Copies sprite data between two sprites, or between a sprite and a string variable.

source sprite number or string variable.

destination sprite number or string variable.

Remarks: Source and destination can either be a sprite number or a string variable,

SPRSAV can be used with the basic form of sprites (C64 compatible) only. These sprites occupy 64 bytes of memory, and create strings of length 64, if the destination parameter is a string variable.

Extended sprites and variable height sprites cannot be used with **SPRSAV**.

A string array of sprite data can be used to store many shapes and copy them fast to the sprite memory with the command **SPRSAV**.

It's also a convenient method to read or write shapes of single sprites from or to a disk file.

Example: Using **SPRSAV**:

```
10 SPRITE LOAD "SPRITEDATA" :REM LOAD DATA FOR 8 SPRITES
20 SPRITE 1,1               :REM TURN SPRITE 1 ON
30 SPRSAV 1,2               :REM COPY SPRITE 1 DATA TO SPRITE 2
40 SPRITE 2,1               :REM TURN SPRITE 2 ON
50 SPRSAV 1,A$              :REM SAVE SPRITE 1 DATA IN STRING A$
```

SQR

Token: $BA

Format: **SQR**(numeric expression)

Returns: The square root of a numeric expression.

Remarks: The argument must not be negative.

Example: Using **SQR**

```
PRINT SQR(2)
 1.41421356
```

ST

Format: **ST**

Usage: The status of the last I/O operation.

If **ST** is zero, there was no error, otherwise it is set to a device dependent error code.

Remarks: **ST** is a reserved system variable.

Example: Using **ST**

```
100 MX=100:DIM T$(MX)        :REM DATA ARRAY
110 DOPEN#1,"DATA"           :REM OPEN FILE
120 IF DS THEN PRINT"COULD NOT OPEN":STOP
130 LINE INPUT#1,T$(N):N=N+1  :REM READ ONE RECORD
140 IF N>MX THEN PRINT "TOO MANY DATA":GOTO 160
150 IF ST=0 THEN 130         :REM ST = 64 FOR END-OF-FILE
160 DCLOSE#1
170 PRINT "READ";N;" RECORDS"
```

STEP

Token: $A9

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

Usage: **STEP** is an optional part of a **FOR** loop.

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

start initial value of the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the **index** at the end of a loop iteration. 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 than or equal to **start**. For negative increments, **end** must be less than or equal to **start**.

It is bad programming practice to change the value of the **index** variable inside the loop or to jump into or out of a 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 will be displayed showing the line number where the program stopped. The READY. prompt appears and the computer goes into direct mode, waiting for keyboard input.

Remarks: All variable definitions are still valid after **STOP**. They may be inspected or altered, and the program may be continued with **CONT**. However, any editing of the program source will disallow any further continuation.

Program execution can be resumed with **CONT**.

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)

Returns: A string of the formatted value of the argument, as if it were **PRINT**ed to the string.

Example: Using **STR$**:

```
A$ = "THE VALUE OF PI IS " + STR$(π)
PRINT A$
THE VALUE OF PI IS 3.14159265
```

STRBIN$

Token: $C2 $12

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

Returns: The number value as a string of its binary representation.

Example: Using **STRBIN$**:

```
PRINT STRBIN$(245)
11110101
```

SYS

Token: $9E

Format: **SYS** address [{, areg, xreg, yreg, zreg, sreg}]

Usage: Calls a machine language subroutine.

address start address of the subroutine. This can be a ROM-resident KERNAL routine or any other routine which has previously been loaded or **POKE**d to RAM.

areg CPU accumulator value.

xreg CPU X register value.

yreg CPU Y register value.

zreg CPU Z register value.

sreg Status register value.

SYS loads the arguments (if any) into registers, then calls the subroutine. The called routine must exit with an **RTS** instruction. After the subroutine has returned, it saves the new register contents, then returns control to the BASIC program.

If the address value is 16 bit ($0000 – $FFFF), the **BANK** value is used to determine the actual address. If the address is higher than $FFFF, it is interpreted as a linear 24 bit address and the value of **BANK** is ignored.

Unlike other BASIC commands that access memory, there are restrictions on which addresses **SYS** can access:

- **SYS** can only access banks 0 – 5, and cannot access Attic RAM or upper memory, even when using long addresses.
- Only offsets $2000 – $7FFF within a given bank actually refer to the memory of that bank.
- **SYS** can only access offsets $0000 – $1FFF in bank 0.
- Accessing offsets $8000 – $FFFF always accesses memory as if **BANK** is set to 128 (including ROM and I/O register mappings), even when **BANK** is set to a different bank or when using long addresses.

Remarks: The register values after a **SYS** call are stored in system memory. **RREG** can be used to retrieve these values.

Despite the unusual restrictions on addresses, the **SYS** command is a powerful way to combine BASIC and machine language code. For short routines, memory in bank 0 offsets $1800 – $1EFF are available for program use. If care is taken to avoid overwriting the end of the BASIC program, machine language routines can be loaded elsewhere in bank 0 up to offset $BFFF.

Using **SYS** properly (i.e. without corrupting the system) requires some technical skill, which is out of scope of the User's Guide. For more information and examples, see *the MEGA65 Book*, Programming with Memory (chapter 13).

Example: Using **SYS**:

```
10 REM DEMO FOR SYS:CHANGING THE BORDER COLOUR
20 BANK 0
30 POKE $4000,$EE,$20,$D0,$60 :REM INC $D020:RTS
40 SYS $4000                  :REM CALL SUBROUTINE AT $4000 / BANK $00
50 GETKEY A$:IF A$ <> "Q" THEN 40
```

TAB

Token: $A3

Format: **TAB**(column)

Returns: Positions the cursor at **column**.

This is only done if the target column is *right* of the current cursor column, otherwise the cursor will not move. The column count starts with 0 being the left-most column.

Remarks: This function shouldn't be confused with TAB, which advances the cursor to the next tab-stop.

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)

Returns: The tangent of an angle.

The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: A value in units of degrees can be converted to radians by multiplying it 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 **PLAY**.

speed 1 – 255

The duration (in seconds) of a whole note is computed with $duration = 24/speed$.

Example: Using **TEMPO**

```
10 VOL 8,8
20 FOR T = 24 TO 18 STEP -2
30 TEMPO T
40 PLAY "T0M3O4QGAGFED","T2O4M5P0H.DP5GB","T5O3IGAGAGAABABAB"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT T
70 PLAY "T0O5QCO4GEH.C","T2O5IEFEDEDCEGO6P8CP0R","T5O3ICDCDEFEDCO4C"
```

THEN

Token: $A7

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

Usage: **THEN** is part of an **IF** statement.

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

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

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

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

Example: Using **THEN**

```
1 REM THEN
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

TI

Format: **TI**

Usage: A high precision timer variable with a resolution of 1 micro second.

It is started or reset with **CLR TI**, and can be accessed in the same way as any other variable in expressions.

Remarks: **TI** is a reserved system variable. The value in **TI** is the number of seconds (to 6 decimal places) since it was last cleared or started.

Example: Using **TI**

```
100 CLR TI               :REM START TIMER
110 FOR I%=1 TO 10000:NEXT :REM DO SOMETHING
120 ET = TI              :REM STORE ELAPSED TIME IN ET
130 PRINT "EXECUTION TIME:";ET;" SECONDS"
```

TIS

Format: **TI$**

Usage: The current time of day, as a string.

The time value is updated from the RTC (Real-Time Clock). The string **TI$** is formatted as: "hh:mm:ss".

TI$ is a read-only variable, which reads the registers of the RTC and formats the values to a string. This differs from other Commodore computers that do not have an RTC.

Remarks: **TI$** is a reserved system variable.

It is possible to access the RTC registers directly via **PEEK**. The start address of the registers is at $FFD7110.

For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page *the MEGA65 Book*, The Configuration Utility (section 4).

```
100 REM ****** READ RTC ****** ALL VALUES ARE BCD ENCODED
110 RT = $FFD7110           :REM ADDRESS OF RTC
120 FOR I=0 TO 5            :REM SS,MM,HH,DD,MO,YY
130 T(I)=PEEK(RT+I)         :REM READ REGISTERS
140 NEXT I                  :REM USE ONLY LAST TWO DIGITS
150 T(2) = T(2) AND 127     :REM REMOVE 24H MODE FLAG
160 T(5) = T(5) + $2000     :REM ADD YEAR 2000
170 FOR I=2 TO 0 STEP -1    :REM TIME INFO
180 PRINT USING ">## ";HEX$(T(I));
190 NEXT I
RUN
12 52 36
```

Example: Using **TI$**

```
PRINT DT$;TI$
05-APR-2021 15:10:00
```

TO

Token: $A4

Format: keyword **TO**

Usage: **TO** is a secondary keyword used in combination with primary keywords, such as **BACKUP**, **BSAVE**, **CHANGE**, **CONCAT**, **COPY**, **FOR**, **GO**, **RENAME**, and **SET DISK**

Remarks: **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: Registers (or clears) a BASIC error handler subroutine.

With an error handler registered, when a BASIC program encounters an error, it calls the subroutine instead of exiting the program. During the subroutine, the system variable **ER** contains the error number. The **TRAP** error handler can then decide whether to **STOP** or **RESUME** execution.

TRAP with no argument disables the error handler, and errors will then 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**).

When trace mode is active, each line number is printed before it is executed. **TROFF** turns off trace mode.

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.

When trace mode is active, each line number is printed before it is executed. **TRON** turns on trace mode.

This is useful for debugging the control flow of a BASIC program. To use it, add **TRON** and **TROFF** statements to the program around the lines that need debugging.

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** [**P**] filename [**,D** drive] [**,U** unit]

Usage: Prints the contents of a file containing text encoded as PETSCII.

If the **P** flag is specified, the listing will pause for each screenful of text. Pressing **Q** quits page mode, while any other key continues to the next page.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: **TYPE** cannot be used to print BASIC programs. Use **LIST** for programs instead. **TYPE** can only process **SEQ** or **USR** files containing records of PETSCII text, delimited by the CR character. (The CR (carriage return) character can be written to a file using **CHR$(13)**.)

See the **EDIT** command for a way to create and modify text files interactively with the MEGA65.

Example: Using **TYPE**

```
TYPE "README"
TYPE "README 1ST",U9

TYPE P "MOBYDICK"
```

UNLOCK

Token: $FE $4F

Format: **UNLOCK** filename/pattern [**,D** drive] [**,U** unit]

Usage: Unlocks a locked file on disk.

The specified file or a set of files, that matches the pattern, is unlocked and no more protected. It can be deleted afterwards with the commands **DELETE**, **ERASE** or **SCRATCH**

The **LOCK** command locks a file.

filename the name of a file. Either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

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

Remarks: Unlocking a file that is already unlocked has no effect.

In direct mode the number of unlocked files is printed. The second to last number from the message contains the number of unlocked files,

Examples: Using **UNLOCK**

```
UNLOCK "SNOOPY",U9 :REM UNLOCK FILE SNOOPY ON UNIT 9
03,FILES UNLOCKED,01,00
UNLOCK "BS*"    :REM UNLOCK ALL FILES BEGINNING WITH "BS"
03,FILES UNLOCKED,04,00
```

UNTIL

Token: $FC

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

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

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

Examples: 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**;** argument

Usage: Parses the **format** string and evaluates the argument. The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can be done in either CBM style, providing a pattern such as `###.##` or in C style using a <width.precision> specifier, such as `%3D %7.2F %4X` .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (`###`), asterisks will be used instead.

Remarks: The format string is only applied for one argument, but it is possible to append more than one **USING format;argument** sequences.

argument 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. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Example: **USING** with a corresponding **PRINT#**

```
PRINT USING "##.##";π, USING " [%6.4F] ";SQR(2)
 3.14 [1.4142]

PRINT USING " < # # # > ";12*31
 < 3 7 2 >

PRINT USING "###"; "ABCDE"
ABC

PRINT USING ">###"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="###,###,###.#":PRINT USING A$;1E8/3
 33,333,333.3
```

USR

Token: $B7

Format: **USR**(numeric expression)

Usage: Invokes an assembly language routine whose memory address is stored at $02F8 – $02F9.

The result of the **numeric expression** is written to floating point accumulator 1.

After executing the assembly routine, BASIC returns the contents of the floating point accumulator 1.

Remarks: Banks 0 – 127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

The floating point accumulator is a facility of the KERNAL that is outside the scope of the User's Guide.

Example: Using **USR**

```
10 WPOKE $2F8, $7F33  : REM NEGATE ROUTINE
20 PRINT USR(π)
30 PRINT USR(-5)
```

VAL

Token: $C5

Format: **VAL**(string expression)

Returns: The decimal floating point value represented by a string.

Remarks: **VAL** parses characters from the beginning of the string that resemble a BASIC decimal number, including a leading negative sign, digits, a decimal point, and an exponent. If it encounters an invalid character, it stops parsing and returns the result up to that point in the string.

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: **VERIFY** with no **binflag** compares a BASIC program in memory with a disk file of type **PRG**. It does the same as **DVERIFY**, but the syntax is different.

VERIFY with **binflag** compares a binary file in memory with a disk file of type **PRG**. It does the same as **BVERIFY**, but the syntax is different.

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

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. 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. **VERIFY** exits with either the message OK or with VERIFY ERROR.

VERIFY is obsolete in BASIC 65. It is only here for backwards compatibility. It is recommended to use **DVERIFY** and **BVERIFY** instead.

Examples: Using **VERIFY**

```
VERIFY "ADVENTURE"
VERIFY "ZORK-I",9
VERIFY "1:DUNGEON",10
```

VIEWPORT

Token: $FE $31

Format: **VIEWPORT CLR**
VIEWPORT DEF x, y, width, height

Usage: Bitmap graphics: manages the viewport of a screen.

VIEWPORT DEF defines a clipping region with the origin (upper left position) set to **x, y** and the **width** and **height**. All following graphics commands are limited to the **VIEWPORT** region.

VIEWPORT CLR fills the clipping region with the colour of the drawing pen.

Remarks: The clipping region can be reset to full screen by the command `VIEWPORT DEF 0,0,WIDTH,HEIGHT` using the same values for WIDHTH and HEIGHT as in the SCREEN command.

Example: Using **VIEWPORT**

```
10 SCREEN 320,200,2
20 VIEWPORT DEF 20,30,100,120 :REM REGION 20->119, 30->149
30 PEN 1                      :REM SELECT COLOUR 1
40 VIEWPORT CLR               :REM FILL REGION WITH COLOUR OF PEN
50 GETKEY A$                  :REM WAIT FOR KEYPRESS
60 SCREEN CLOSE
```

VOL

Token: $DB

Format: **VOL** right, left

Usage: Sets the volume for sound output with **SOUND** or **PLAY**.

right is the volume for SIDs 1 and 2, and **left** is the volume for SIDs 3 and 4. The value ranges from 0 (off) to 15 (loudest).

Remarks: The terms ”right” and ”left” refer to the default pan settings for the MEGA65 SID chips in the audio mixer. The actual volume and pan position for each pair of SIDs depends on the audio mixer settings. You can adjust the audio settings in the Freezer.

Example: Using **VOL**

```
10 TEMPO 22
20 FOR V = 2 TO 12 STEP 2
30 VOL V,16-V
40 PLAY "T0M3O4QGAGFED","T2O4M5P0H.DP5GB","T5O3IGAGAGAABABAB","G"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT V
70 PLAY "T0O5QCO4GEH.C","T2O5IEFEDEDCEGO6P9CP0R","T5O3ICDCDEFEDCO4C","C"
```

VSYNC

Token: $FE $54

Format: **VSYNC** raster line

Usage: Waits until the selected raster line is active.

raster line (0 - 311) for PAL, (0 - 262) for NTSC mode.

This pauses execution of the BASIC program until the screen update reaches the given vertical pixel coordinate. This is a very brief pause: the screen updates 50 times per second in PAL mode, and 60 times per second in NTSC mode. This is useful to change graphics parameters at specific points in the screen update, and to synchronize BASIC program logic with the screen refresh rate.

Example: Using **VSYNC**

```
10 IF FRE(-1)<920364 THEN PRINT"UPDATE ROM":END
20 BORDER 3     :REM CHANGE BORDER COLOUR TO CYAN
30 VSYNC 100    :REM WAIT UNTIL RASTER LINE 100
40 BORDER 7     :REM CHANGE BORDER COLOUR TO YELLOW
50 VSYNC 260    :REM WAIT UNTIL RASTER LINE 260
60 GOTO 20      :REM LOOP
```

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 non-zero, otherwise reading is repeated. This may hang the computer indefinitely if the condition is never met.

Remarks: **WAIT** 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 specific raster line is about to be drawn to the screen.

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 expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

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

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

Examples: 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

By default, text updates occur on the entire available text screen. **WINDOW** narrows the update region to a rectangle of the available screen space.

Remarks: The row values range from 0 to 24. The column values range from 0 to either 39 or 79. This depends on the screen mode.

There can be only one window on the screen. Pressing CLR HOME twice or **PRINT**ing **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 CENTRED WINDOW
```

WPEEK

Token: $CE $10

Format: **WPEEK(**address**)**

Returns: The 16-bit word value stored in memory at **address** (low byte) and **address** + 1 (high byte), as an unsigned 16-bit number.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0 – 127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **WPEEK**

```
20 UA = WPEEK($02F8)   :REM USR JUMP TARGET
50 PRINT "USR FUNCTION CALL ADDRESS";UA
```

WPOKE

Token: $FE $1D

Format: **WPOKE** address**,** word [**,** word ...]

Returns: Writes one or more 16-bit words into memory or memory mapped I/O, starting at **address**.

If the address is in the range of $0000 to $FFFF (0 - 65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

word a value from 0 - 65535. The first word is stored at address (low byte) and address+1 (high byte). The second word is stored at address+2 (low byte) and address+3 (high byte), etc. If a value is larger than 65535, only the lower two bytes are used.

Remarks: The address is increased by two for each data word, so a memory range can be written to with a single **WPOKE**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **WPOKE**

```
10 BANK 128              :REM SELECT SYSTEM BANK
20 WPOKE $02F8,$1800     :REM SET USR VECTOR TO $1800
```

XOR

Token: $E9

Format: operand **XOR** operand

Usage: Performs a bit-wise logical Exclusive OR operation on two 16-bit values.

Integer operands are used as they are. Real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to 16-bit integer using $FFFF, (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

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

Example: Using **XOR**

```
FOR I = 0 TO 8: PRINT I XOR 5;: NEXT I
 5  4  7  6  1  0  3  2  13
```

CHAPTER 3

Screen Codes

SCREEN CODES

A text character is represented in screen memory by a screen code. There are 256 possible screen codes, each referring to an image in the current character set.

A complete character set contains two groups of 256 images, one for the uppercase mode and one for the lowercase mode, for a total of 512 images. Only one mode can be displayed at a time. The built-in character sets use the first 128 characters of each group for normal characters and the next 128 for reversed versions of the same characters.

In BASIC, the **T@&()** special array provides access to the characters on the screen using column and row indexes. The values in this special array are screen codes. The **FONT** command changes between the built-in character sets. The **CHARDEF** command changes the image associated with a screen code.

Note: Screen codes are different to PETSCII codes. PETSCII codes are used to store, transmit, and receive textual data, and control the way strings are printed to the screen. When a PETSCII character is printed to the screen, the corresponding screen code is written to screen memory. For a list of PETSCII codes, see appendix 4 on page 281.

The following table lists the screen codes. When a code produces a different character based on the mode, the character is listed as "uppercase / lowercase."

0 @	**15** O / o	**30** ↑	**45** -
1 A / a	**16** P / p	**31** ←	**46** .
2 B / b	**17** Q / q	**32** space	**47** /
3 C / c	**18** R / r	**33** !	**48** 0
4 D / d	**19** S / s	**34** "	**49** 1
5 E / e	**20** T / t	**35** #	**50** 2
6 F / f	**21** U / u	**36** $	**51** 3
7 G / g	**22** V / v	**37** %	**52** 4
8 H / h	**23** W / w	**38** &	**53** 5
9 I / i	**24** X / x	**39** '	**54** 6
10 J / j	**25** Y / y	**40** (	**55** 7
11 K / k	**26** Z / z	**41**)	**56** 8
12 L / l	**27** [	**42** *	**57** 9
13 M / m	**28** £	**43** +	**58** :
14 N / n	**29**]	**44** ,	**59** ;

60	<
61	=
62	>
63	?
64	
65	/ A
66	/ B
67	/ C
68	/ D
69	/ E
70	/ F
71	/ G
72	/ H
73	/ I
74	/ J
75	/ K
76	/ L
77	/ M
78	/ N
79	/ O
80	/ P
81	/ Q
82	/ R
83	/ S
84	/ T
85	/ U
86	/ V
87	/ W
88	/ X
89	/ Y
90	/ Z
91	
92	
93	
94	π /
95	/
96	space
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	

Note: In the built-in character sets, codes 128-255 are reversed versions of 0-127.

CHAPTER 4

PETSCII Codes

- PETSCII Codes and CHR$

PETSCII CODES AND CHR$

In BASIC, `PRINT CHR$(X)` can be used to print a character from a PETSCII code. Below is the full table of PETSCII codes you can print by index. For example, while in the default uppercase/graphics mode, by using index 65 from the table below as: `PRINT CHR$(65)` you will print the letter `A`. You can read more about **CHR$** on page 47.

You can also do the reverse with the ASC statement. For example: `PRINT ASC("A")` will output `65`, which matches the code in the table.

<u>NOTE</u>: Function key (F1-F14 + HELP) values in this table are not intended to be printed via `CHR$()`, but rather to allow function-key input to be assessed in BASIC programs via the GET / GETKEY commands.

0	
1	ALTERNATE PALETTE
2	UNDERLINE ON
3	
4	DEFAULT PALETTE
5	WHITE
6	
7	BELL
8	
9	TAB
10	LINEFEED
11	DISABLE SHIFT
12	ENABLE SHIFT
13	RETURN
14	LOWER CASE
15	BLINK/FLASH ON
16	F9
17	↓
18	RVS ON
19	CLR HOME
20	INST DEL
21	F10 / BACK WORD
22	F11
23	F12 / NEXT WORD
24	SET/CLEAR TAB
25	F13
26	F14 / BACK TAB
27	ESCAPE
28	RED
29	→
30	GREEN
31	BLUE
32	SPACE
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V

87	W
88	X
89	Y
90	Z
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	π
127	
128	
129	ORANGE
130	UNDERLINE OFF
131	SHIFT RUN STOP
132	HELP
133	F1
134	F3
135	F5
136	F7
137	F2
138	F4
139	F6
140	F8
141	SHIFT RETURN
142	UPPERCASE
143	BLINK/FLASH OFF
144	BLACK
145	↑
146	RVS OFF
147	SHIFT CLR HOME
148	SHIFT INST DEL
149	BROWN
150	LT. RED (PINK)
151	DK. GREY
152	GREY
153	LT. GREEN
154	LT. BLUE
155	LT. GREY
156	PURPLE
157	←
158	YELLOW
159	CYAN
160	SPACE
161	
162	
163	
164	
165	
166	
167	
168	
169	
170	
171	
172	
173	
174	
175	
176	
177	
178	
179	
180	
181	
182	
183	
184	
185	
186	
187	
188	
189	
190	
191	

Note 1: Codes from 192 to 223 are equal to 96 to 127. Codes from 224 to 254 are equal to 160 to 190, and code 255 is equal to 126.

Note 2: While using lowercase/uppercase mode (by pressing + SHIFT), be aware that:

- The uppercase letters in region 65-90 of the above table are replaced with lowercase letters.
- The graphical characters in region 97-122 of the above table are replaced with uppercase letters.
- PETSCII's lowercase (65-90) and uppercase (97-122) letters are in ASCII's uppercase (65-90) and lowercase (97-122) letter regions.

CHAPTER 5

Screen Editor Keys

- Screen Editor Keys
- Control codes
- Shifted codes
- Escape Sequences

SCREEN EDITOR KEYS

The following key combinations perform actions in the MEGA65 screen editor.

In some cases, a program can print the equivalent PETSCII codes to perform the same actions. For example, CTRL + G, which plays a bell sound, can be printed by a program as `CHR$(7)`. To print an ESC sequence, use `CHR$(27)` to represent the ESC key, followed by the next key in the sequence.

CONTROL CODES

Keyboard Control	Function
Colours	
CTRL + 1 to 8	Choose from the first range of colours. See appendix 6 on page 297 for the list of colours in the system palette.
MEGA + 1 to 8	Choose from the second range of colours.
CTRL + E	Restores the colour of the cursor back to the default (white).
CTRL + D	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or MEGA and keys 1 to 8 (for the remaining 8 colours).
CTRL + A	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or MEGA and keys 1 to 8 (for the remaining 8 colours).
Tabs	

Keyboard Control	Function
CTRL + Z	Tabs the cursor to the left. If there are no tab positions remaining, the cursor will remain at the start of the line.
CTRL + I	Tabs the cursor to the right. If there are no tab positions remaining, the cursor will remain at the end of the line.
CTRL + X	Sets or clears the current screen column as a tab position. Use CTRL + Z and I to jump back and forth to all positions set with X.
Movement	
CTRL + Q	Moves the cursor down one line at a time. Equivalent to ↓.
CTRL + J	Moves the cursor down a position. If you are on a long line of BASIC code that has extended to two lines, then the cursor will move down two rows to be on the next line.
CTRL +]	Equivalent to →.
CTRL + T	Backspace the character immediately to the left and to shift all rightmost characters one position to the left. This is equivalent to INST DEL.
CTRL + M	Performs a carriage return, equivalent to RETURN.
Word movement	
CTRL + U	Moves the cursor backward to the start of the previous word. If there is no previous word on the current line, it moves to the first column of the current line, then to the previous line, until a line with a word is encountered.

Keyboard Control	Function
CTRL + W	Advances the cursor forward to the start of the next word. If there is no next word on the current line, it moves to the first column of the next line, until a line with a word is encountered.
Scrolling	
CTRL + P	Scroll BASIC listing down one line. Equivalent to F9.
CTRL + V	Scroll BASIC listing up one line. Equivalent to F11.
CTRL + S	Equivalent to NO SCROLL.
Formatting	
CTRL + B	Enables underline text mode. You can disable underline mode by pressing ESC, then O.
CTRL + O	Enables flashing text mode. You can disable flashing mode by pressing ESC, then O.
Casing	
CTRL + N	Changes the text case mode from uppercase to lowercase.
CTRL + K	Locks the uppercase/lowercase mode switch usually performed with M + SHIFT.
CTRL + L	Enables the uppercase/lowercase mode switch that is performed with the M + SHIFT.
Miscellaneous	
CTRL + G	Produces a bell tone.
CTRL + [	Equivalent to pressing ESC.
CTRL + *	Enters the Matrix Mode Debugger.

SHIFTED CODES

Keyboard Control	Function
SHIFT + INST DEL	Insert a character at 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, briefly press and release ESC, then press one of the following keys to perform the sequence.

Key	Sequence
Editor behaviour	
ESC X	Clears the screen and toggles between 40 × 25 and 80 × 25 text modes.
ESC 4	Clears the screen and switches to 40 × 25 text mode.
ESC 8	Clears the screen and switches to 80 × 25 text mode.
ESC 5	Switches to 80 × 50 text mode.
Note that some programs expect to be started in 80 × 25 mode, and may not behave correctly when started in 80 × 50 mode.	

Key	Sequence
ESC @	Clears a region of the screen, starting from the current cursor position, to the end of the screen.
ESC O	Cancels the quote, reverse, underline, and flash modes.
Scrolling	
ESC V	Scrolls the entire screen up one line.
ESC W	Scrolls the entire screen down one line.
ESC L	Enables scrolling when ↓ is pressed at the bottom of the screen.
ESC M	Disables scrolling. When pressing ↓ at the bottom of the screen, the cursor will move to the top of the screen. However, when pressing ↑ at the top of the screen, the cursor will remain on the first line.
ESC N	Enables "line pushing:" typing or printing in the rightmost column pushes subsequent lines down by one.
ESC R	Disables "line pushing:" typing or printing in the rightmost column moves the cursor to the beginning of the next line, but does not push any lines. Disable both line pushing (ESC R) and scrolling (ESC M) to allow **PRINT**ing in the rightmost column without disturbing the rest of the display.
Insertion and deletion	
ESC I	Inserts an empty line at the current cursor position and moves all subsequent lines down one position.

Key	Sequence
ESC D	Deletes the current line and moves lines below the cursor up one position.
ESC P	Erases all characters from the cursor to the start of the current line.
ESC Q	Erases all characters from the cursor to the end of the current line.
Movement	
ESC J	Moves the cursor to the start of the current line.
ESC K	Moves the cursor to the last non-whitespace character on the current line.
ESC ↑	Saves the current cursor position. Use ESC ← (next to 1) to move it back to the saved position. Note that the ↑ used here is next to RESTORE.
ESC ←	Restores the cursor position to the position stored via a prior a press of the ESC ↑ (next to RESTORE) key sequence. Note that the ← used here is next to 1.
ESC HOME	Restores the cursor position to the position stored via a prior a press of HOME.
Windowing	
ESC T	Sets the top-left corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see ESC B. Windowed mode can be disabled by pressing CLR HOME twice.

Key	Sequence
ESC **B**	Sets the bottom right corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see **ESC** **T**. Windowed mode can be disabled by pressing **CLR HOME** twice.
Cursor behaviour	
ESC **A**	Enables auto-insert mode. Any keys pressed will be inserted at the current cursor position, shifting all characters on the current line after the cursor to the right by one position.
ESC **C**	Disables auto-insert mode, reverting back to overwrite mode.
ESC **E**	Sets the cursor to non-flashing mode.
ESC **F**	Sets the cursor to regular flashing mode.
Bell behaviour	
ESC **G**	Enables the bell which can be sounded using **CTRL** and **G**.
ESC **H**	Disable the bell so that pressing **CTRL** and **G** will have no effect.
Colours	
ESC **U**	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with **CTRL** and keys **1** to **8** (for the first 8 colours), or **M** and keys **1** to **8** (for the remaining colours).

Key	Sequence
ESC S	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or M and keys 1 to 8 (for the remaining colours).
Tabs	
ESC Y	Set the default tab stops (every 8 spaces) for the entire screen.
ESC Z	Clears all tab stops. Any tabbing with CTRL and I will move the cursor to the end of the line.

CHAPTER 6

System Palette

- System Palette

SYSTEM PALETTE

The following table describes the system colour palette as it is defined by default.

Colour palette indexes are used as values in the **C@&()** special array, and as arguments for BASIC commands such as **BACKGROUND**, **BORDER**, **COLOR**, **FOREGROUND**, **HIGHLIGHT**, **PEN**, and **SCREEN CLR**.

Index	Red	Green	Blue	Colour
0	0	0	0	Black
1	15	15	15	White
2	15	0	0	Red
3	0	15	15	Cyan
4	15	0	15	Purple
5	0	15	0	Green
6	0	0	15	Blue
7	15	15	0	Yellow
8	15	6	0	Orange
9	10	4	0	Brown
10	15	7	7	Light Red (Pink)
11	5	5	5	Dark Grey
12	8	8	8	Medium Grey
13	9	15	9	Light Green
14	9	9	15	Light Blue
15	11	11	11	Light Grey
16	14	0	0	Guru Meditation
17	15	5	0	Rambutan
18	15	11	0	Carrot
19	14	14	0	Lemon Tart
20	7	15	0	Pandan
21	6	14	6	Seasick Green
22	0	14	3	Soylent Green
23	0	15	9	Slimer Green
24	0	13	13	The Other Cyan
25	0	9	15	Sea Sky
26	0	3	15	Smurf Blue
27	0	0	14	Screen of Death
28	7	0	15	Plum Sauce
29	12	0	15	Sour Grape
30	15	0	11	Bubblegum
31	15	3	6	Hot Tamales

CHAPTER 7

Supporters & Donors

- Organisations
- Contributors
- Supporters

The MEGA65 would not have been possible to create without the generous support of many organisations and individuals.

We are still compiling these lists, so apologies if we haven't included you yet. If you know anyone we have left out, please let us know, so that we can recognise the contribution of everyone who has made the MEGA65 possible, and into the great retro-computing project that it has become.

ORGANISATIONS

The MEGA Museum of Electronic Games & Art e.V. Germany
EVERYTHING

Trenz Electronic, Germany
MOTHERBOARD MANUFACTURING SALES

Hintsteiner, Austria
CASE

GMK, Germany
KEYBOARD

KEVAG Telekom, Germany
WEB HOSTING

CONTRIBUTORS

Andreas Liebeskind
(libi in paradize)
CFO MEGA eV

Thomas Hertzler
(grumpyninja)
USA spokesman

Russell Peake
(rdpeake)
Bug herding

Alexander Nik Petra
(n0d)
Early case design

Ralph Egas
(0-limits)
Business advisor

Lucas Moss
MEGAphone PCB design

Daren Klamer
(Impakt)
Manual proof-reading

Daniël Mantione
(dmantione)
C64 hardware guru

Dr. Canan Hastik
(indica)
Chairwoman MEGA eV

Simon Jameson
(Shallan)
Platform enhancements

Stephan Kleinert
(ubik)
Destroyer of BASIC 10

Wayne Johnson
(sausage)
Manual additions

L. Kleiss
(LAK132)
MegaWAT presentation software

Maurice van Gils
(Maurice)
BASIC 65 example programs

Andrew Owen
(Cheveron)
Keyboard, Sinclair support

Adam Barnes
(amb5l)
HDMI expert and board revision

Wayne Rittimann, Jr.
(johnwayner)
Bug squashing on all levels

SUPPORTERS

3c74ce64
8-Bit Classics
@11110110100
Aaron Smith
Achim Mrotzek
Adolf Nefischer
Adrian Esdaile
Adrien Guichard
Ahmed Kablaoui
Alan Bastian Witkowski
Alan Field
Alastair Paulin-Campbell
Alberto Mercuri
Alexander Haering
Alexander Kaufmann
Alexander Niedermeier
Alexander Soppart
Alfonso Ardire
Amiga On The Lake
André Kudra
André Simeit
André Wösten
Andrea Farolfi
Andrea Minutello
Andreas Behr
Andreas Freier
Andreas Grabski
Andreas Millinger
Andreas Nopper
Andreas Ochs
Andreas Wendel Manufaktur
Andreas Zschunke
Andrew Bingham
Andrew Dixon
Andrew Mondt
Andrzej Hłuchyj
Andrzej Sawiniec
Andrzej Śliwa
Anthony W. Leal
Arkadiusz Bronowicki
Arkadiusz Kwasny
Arnaud Léandre
Arne Drews
Arne Neumann
Arne Richard Tyarks
Axel Klahr
Balaz Ondrej
Barry Thompson
Bartol Filipovic
Benjamin Maas
Bernard Alaiz
Bernhard Zorn
Bieno Marti-Braitmaier
Bigby
Bill LaGrue
Bjoerg Stojalowski
Björn Johannesson
Bjørn Melbøe
Bo Goeran Kvamme
Boerge Noest
Bolko Beutner
Brett Hallen
Brian Gajewski
Brian Green
Brian Juul Nielsen
Brian Reiter
Bryan Pope
Burkhard Franke
Byron Goodman
Cameron Roberton (KONG)
Carl Angervall
Carl Danowski
Carl Stock
Carl Wall
Carlo Pastore
Carlos Silva
Carsten Sørensen
Cenk Miroglu Miroglu
Chang sik Park
Charles A. Hutchins Jr.
Chris Guthrey
Chris Hooper
Chris Stringer
Christian Boettcher
Christian Eick
Christian Gleinser
Christian Gräfe
Christian Heffner
Christian Kersting
Christian Schiller
Christian Streck
Christian Weyer
Christian Wyk
Christoph Haug
Christoph Huck
Christoph Pross
Christopher Christopher
Christopher Kalk
Christopher Kohlert
Christopher Nelson
Christopher Taylor
Christopher Whillock
Claudio Piccinini
Claus Skrepek
Collen Blijenberg
Constantine Lignos
Crnjaninja
Daniel Auger
Daniel Julien
Daniel Lobitz
Daniel O'Connor
Daniel Teicher
Daniel Tootill
Daniel Wedin
Daniele Benetti
Daniele Gaetano Capursi
Dariusz Szczesniak
Darrell Westbury
David Asenjo Raposo
David Dillard
David Gorgon
David Norwood
David Raulo
David Ross
de voughn accooe
Dean Scully
Dennis Jeschke
Dennis Schaffers
Dennis Schierholz

Dennis Schneck
denti
Dick van Ginkel
Diego Barzon
Dierk Schneider
Dietmar Krueger
Dietmar Schinnerl
Dirk Becker
Dirk Wouters
Domingo Fivoli
DonChaos
Donn Lasher
Douglas Johnson
Dr. Leopold Winter
Dusan Sobotka
Earl Woodman
Ed Reilly
Edoardo Auteri
Eduardo Gallardo
Eduardo Luis Arana
Eirik Juliussen Olsen
Emilio Monelli
EP Technical Services
Epic Sound
Erasmus Kuhlmann
ergoGnomik
Eric Hilaire
Eric Hildebrandt
Eric Hill
Eric Jutrzenka
Erwin Reichel
Espen Skog
Evangelos Mpouras
Ewan Curtis
Fabio Zanicotti
Fabrizio Di Dio
Fabrizio Lodi
FARA Gießen GmbH
FeralChild
First Choice Auto's
Florian Rienhardt
Forum64. de
Francesco Baldassarri
Frank Fechner
Frank Glaush
Frank Gulasch
Frank Haaland
Frank Hempel
Frank Koschel
Frank Linhares
Frank Sleeuwaert
Frank Wolf
FranticFreddie
Fredrik Ramsberg
Fridun Nazaradeh
Friedel Kropp
Garrick West
Gary Lake-Schaal
Gary Pearson
Gavin Jones
Geir Sigmund Straume
Gerd Mitlaender
Giampietro Albiero
Giancarlo Valente
Gianluca Girelli
Giovanni Medina
Glen Fraser
Glen R Perye III
Glenn Main
Gordon Rimac
GRANT BYERS
Grant Louth
Gregor Bubek
Gregor Gramlich
Guido Ling
Guido von Gösseln
Guillaume Serge
Gunnar Hemmerling
Günter Hummel
Guy Simmons
Guybrush Threepwood
Hakan Blomqvist
Hans Pronk
Hans-Jörg Nett
Hans-Martin Zedlitz
Harald Dosch
Harri Salokorpi
Harry Culpan
Harry Venema
Heath Gallimore
Heinz Roesner
Heinz Stampfli
Helge Förster
Hendrik Fensch
Henning Harperath
Henri Parfait
Henrik Kühn
Holger Burmester
Holger Sturk
Howard Knibbs
Hubert de Hollain
Huberto Kusters
Hugo Maria Gerardus v.d. Aa
Humberto Castaneda
Ian Cross
IDE64 Staff
Igor Ianov
Igor Kurtes
Immo Beutler
Ingo Katte
Ingo Keck
Insanely Interested Publishing
IT-Dienstleistungen Obsieger
Ivan Elwood
Jaap HUIJSMAN
Jace Courville
Jack Wattenhofer
Jakob Schönpflug
Jakub Tyszko
James Hart
James Marshburn
James McClanahan
James Sutcliffe
Jan Bitruff
Jan Hildebrandt
Jan Iemhoff
Jan Kösters
Jan Peter Borsje
Jan Schulze
Jan Stoltenberg-Lerche
Janne Tompuri
Jannis Schulte
Jari Loukasmäki
Jason Smith
Javier Gonzalez Gonzalez
Jean-Paul Lauque
Jeffrey van der Schilden
Jens Schneider

Jens-Uwe Wessling
Jesse DiSimone
Jett Adams
Johan Arneklev
Johan Berntsson
Johan Svensson
Johannes Fitz
John Cook
John Deane
John Dupuis
John Nagi
John Rorland
John Sargeant
John Traeholt
Jon Sandelin
Jonas Bernemann
Jonathan Prosise
Joost Honig
Jordi Pakey-Rodriguez
Jöre Weber
Jörg Jungermann
Jörg Schaeffer
Jörg Weese
Josef Hesse
Josef Soucek
Josef Stohwasser
Joseph Clifford
Joseph Gerth
Jovan Crnjanin
Juan Pablo Schisano
Juan S. Cardona Iguina
JudgeBeeb
Juliussen Olsen
Juna Luis Fernandez Garcia
Jürgen Endras
Jürgen Herm Stapelberg
Jyrki Laurila
Kai Pernau
Kalle Pöyhönen
Karl Lamford
Karl-Heinz Blum
Karsten Engstler
Karsten Westebbe
katarakt
Keith McComb
Kenneth Dyke
Kenneth Joensson
Kevin Edwards
Kevin Thomasson
Kim Jorgensen
Kim Rene Jensen
Kimmo Hamalainen
Konrad Buryło
Kosmas Einbrodt
Kurt Klemm
Lachlan Glaskin
Large bits collider
Lars Becker
Lars Edelmann
Lars Slivsgaard
Lasse Lambrecht
Lau Olivier
Lee Chatt
Loan Leray
Lorenzo Quadri
Lorenzo Travagli
Lorin Millsap
Lothar James Foss
Lothar Serra Mari
Luca Papinutti
Ludek Smetana
Lukas Burger
Lutz-Peter Buchholz
Luuk Spaetgens
Mad Web Skills
MaDCz
Magnus Wiklander
Maik Diekmann
Malte Mundt
Manfred Wittemann
Manuel Beckmann
Manzano Mérida
Marc "3D-vice" Schmitt
Marc Bartel
Marc Jensen
Marc Schmidt
Marc Theunissen
Marc Tutor
Marc Wink
Marcel Buchtmann
Marcel Kante
Marco Beckers
Marco Cappellari
Marco Rivela
Marco van de Water
Marcus Gerards
Marcus Herbert
Marcus Linkert
Marek Pernicky
Mario Esposito
Mario Fetka
Mario Teschke
Mariusz Tymków
Mark Adams
Mark Anderson
Mark Green
Mark Hucker
Mark Leitiger
Mark Spezzano
Mark Watkin
Marko Rizvic
Markus Bieler
Markus Bonet
Markus Dauberschmidt
Markus Fehr
Markus Fuchs
Markus Guenther-Hirn
Markus Liukka
Markus Merz
Markus Roesgen
Markus Uttenweiler
Martin Bauhuber
Martin Benke
Martin Gendera
Martin Groß
Martin Gutenbrunner
Martin Johansen
Martin Marbach
Martin Sonnleitner
Martin Steffen
Marvin Hardy
Massimo Villani
Mathias Dellacherie
Mathieu Chouinard
Matthew Adams
Matthew Browne
Matthew Carnevale
Matthew Palmer

Matthew Santos
Matthias Barthel
Matthias Dolenc
Matthias Fischer
Matthias Frey
Matthias Grandis
Matthias Guth
Matthias Lampe
Matthias Meier
Matthias Mueller
Matthias Nofer
Matthias Schonder
Maurice Al-Khaliedy
Max Ihlenfeldt
Meeso Kim
Michael Dailly
Michael Dötsch
Michael Dreßel
Michael Fichtner
Michael Fong
Michael Geoffrey Stone
Michael Gertner
Michael Grün
Michael Habel
Michael Härtig
Michael Haynes
Michael J Burkett
Michael Jensen
Michael Jurisch
Michael Kappelgaard
Michael Kleinschmidt
Michael Lorenz
Michael Mayerhofer
Michael Nurney
Michael Rasmussen
Michael Richmond
Michael Sachse
Michael Sarbak
Michael Schneider
Michael Scholz
Michael Timm
Michael Traynor
Michael Whipp
Michal Ursiny
Michele Chiti
Michele Perini
Michele Porcu
Miguel Angel Rodriguez Jodar
Mikael Lund
Mike Betz
Mike Kastrantas
Mike Pikowski
Mikko Hämäläinen
Mikko Suontausta
Mirko Roller
Miroslav Karkus
Morgan Antonsson
Moritz
Morten Nielsen
MUBIQUO APPS,SL
Myles Cameron-Smith
Neil Moore
Nelson
neoman
Nicholas Melnick
Nikolaj Brinch Jørgensen
Nils Andreas
Nils Eilers
Nils Hammerich
Nils77
Norah Smith
Norman King
Normen Zoch
Olaf Grunert
Ole Eitels
Oliver Boerner
Oliver Brüggmann
Oliver Graf
Oliver Smith
Olivier Bori
ONEPSI LLC
oRdYNe
Osaühing Trioflex
OSHA-PROS USA
Padawer
Patrick Becher
Patrick Bürckstümmer
Patrick de Zoete
Patrick Toal
Patrick Vogt
Paul Alexander Warren
Paul Gerhardt (KONG)
Paul Jackson
Paul Johnson
Paul Kuhnast (mindrail)
Paul Massay
Paul Westlake
Paul Woegerer
Pauline Brasch
Paulo Apolonia
Pete Collin
Pete of Retrohax.net
Peter Eliades
Peter Gries
Peter Habura
Peter Herklotz
Peter Huyoff
Peter Knörzer
Peter Leswell
Peter Weile
Petri Alvinen
Philip Marien
Philip Timmermann
Philipp Rudin
Pierre Kressmann
Pieter Labie
Piotr Kmiecik
Power-on.at
Przemysław Safonow
Que Labs
R Welbourn
R-Flux
Rafał Michno
Rainer Kappler
Rainer Kopp
Rainer Weninger
Ralf Griewel
Ralf Pöscha
Ralf Reinhardt
Ralf Schenden
Ralf Smolarek
Ralf Zenker
Ralph Bauer
Ralph Wernecke
Rédl Károly
Reiner Lanowski
Remi Veilleux
Riccardo Bianchi

Richard Englert
Richard Good
Richard Menedetter
Richard Sopuch
Rick Reynolds
Rico Gruninger
Rob Dean
Robert Bernardo
Robert Eaglestone
Robert Grasböck
Robert Miles
Robert Schwan
Robert Shively
Robert Tangmar
Robert Trangmar
Rodney Xerri
Roger Olsen
Roger Pugh
Roland Attila Kett
Roland Evers
Roland Schatz
Rolf Hass
Ronald Cooper
Ronald Hunn
Ronny Hamida
Ronny Preiß
Roy van Zundert
Rüdiger Wohlfromm
Ruediger Schlenter
Rutger WIllemsen
Sampo Peltonen
Sarmad Gilani
SAS74
Sascha Hesse
Scott Halman
Scott Hollier
Scott Robison
Sebastian Baranski
Sebastian Bölling
Sebastian Felzmann
Sebastian Lipp
Sebastian Rakel
Şemseddin Moldibi
Seth Morabito
Shawn McKee
Siegfried Hartmann
Zytex Online Store
Sigurbjorn Larusson
Sigurdur Finnsson
Simon Lawrence
Simon Wolf
spreen.digital
Stefan Haberl
Stefan Kramperth
Stefan Richter
Stefan Schultze
Stefan Sonnek
Stefan Theil
Stefan Vrampe
Stefano Canali
Stefano Mozzi
Steffen Reiersen
Stephan Bielmann
Stephen Jones
Stephen Kew
Steve Gray
Steve Kurlin
Steve Lemieux
Steven Combs
Stewart Dunn
Stuart Marsh
Sven Neumann
Sven Stache
Sven Sternberger
Sven Wiegand
Szabolcs Bence
Tantrumedia Limited
Techvana Operations Ltd.
Teddy Turmeaux
Teemu Korvenpää
The Games Foundation
Thierry Supplisson
Thieu-Duy Thai
Thomas Bierschenk
Thomas Edmister
Thomas Frauenknecht
Thomas Gitzen
Thomas Gruber
Thomas Haidler
Thomas Jager
Thomas Karlsen
Thomas Laskowski
Thomas Marschall
Thomas Niemann
Thomas Scheelen
Thomas Schilling
Thomas Tahsin-Bey
Thomas Walter
Thomas Wirtzmann
Thorsten Knoll
Thorsten Nolte
Tim Krome
Tim Waite
Timo Weirich
Timothy Blanks
Timothy Henson
Timothy Prater
Tobias Butter
Tobias Heim
Tobias Köck
Tobias Lüthi
Tommi Vasarainen
Toni Ammer
Tore Olsen
Torleif Strand
Torsten Schröder
Tuan Nguyen
Uffe Jakobsen
Ulrich Hintermeier
Ulrich Nieland
Ulrik Kruse
Urban Lindeskog
Ursula Förstle
Uwe Anfang
Uwe Boschanski
Vedran Vrbanc
Verm Project
Wayne Rittimann
Wayne Sander
Wayne Steele
Who Knows
Winfried Falkenhahn
Wolfgang Becker
Wolfgang Stabla
Worblehat
www.patop69.net
Yan B
Zoltan Markus
Zsolt Zsila

Bibliography

[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] N. Montfort, P. Baudoin, J. Bell, I. Bogost, J. Douglass, M. C. Marino, M. Mateas, C. Reas, M. Sample, and N. Vawter, *10 PRINT CHR $(205.5+ RND (1));: GOTO 10*. MIT Press, 2012.

[3] Actraiser, "Vic-ii for beginners: Screen modes, cheaper by the dozen," 2013. [Online]. Available: http://dustlayer.com/vic-ii/2013/4/26/vic-ii-for-beginners-screen-modes-cheaper-by-the-dozen

INDEX