

andiBASIC

MANUAL

FOR
AMOS/ANDI & IM6/6E
SYSTEMS

General _____ **2**

Direct mode & program mode	2
Structure of an andiBASIC program	3
clear program code lines	3
logical end of a program	3
Executing an ANDIbasic program	4
execute ANDIbasic program	4
interrupt program execution	4
Initialize program RAM	4
list an ANDIbasic program	5
print characters to the screen	5
print characters to the logfile / ztrace	6

Data types, Variables, Constants _____ **7**

Data types	7
Integer numbers	7
Real numbers	7
Complex numbers	7
Byte numbers	7
Logical data	7
Strings	7
Variables	8
Dimensioning Variables	9
Conversion of Data types	10
convert the numerical value of a string to a number	11
converts the numerical argument(s) of a string to number(s)	11
convert numbers to strings	12
fill a complex constant to a complex variable	12
real part of a complex variable	12
imaginary part of a complex variable	12
ASCII to code conversion	12
code to ASCII conversion	13
hexadecimal to decimal conversion	13
decimal to hexadecimal conversion	13
String to string conversion	14

Mathematics _____ **16**

assignment	16
Mathematical rules	16
Mathematical Operations	17
Mathematical Functions	18
absolute value function	18
integer value function	18
5/4 rounding function	18

fraction value function	18
sign function	18
square root function	19
e^n function	19
Natural logarithm	19
10-based logarithm	19
Trigonometrical Functions	20
sine	20
cosine	20
tangent	20
cotangent	20
arc sine	20
arc cosine	20
arc tangent	20
area sine	20
area cosine	20
area tangent	20
area cotangent	20
area arc tangent	20
Functions with complex arguments	21
complex modulo	21
performs 5/4 rounding on real- and imaginary component	21
complex argument	21
exchanges real component vs. imaginary component	21
inverts the sign of the imaginary component	21
returns quadrant number	21
converts from polar coordinates to rectangular coordinates	21
converts from rectangular coordinates to polar coordinates	21
complex square root	21
complex exponential function	21
complex natural logarithm	21
complex sine	21
complex cosine	21
complex tangent	21
complex cotangent	21
complex arc sine	21
complex arc cosine	21
complex arc tangent	21
complex hyperbolic sine	21
complex hyperbolic cosine	21
complex hyperbolic tangent	21
complex hyperbolic cotangent	21
complex hyperbolic arc tangent	21
Comparison Operators	22
less than	22
greater than	22

equal to	22
not equal to	22
Logical Operators	23
Logical <i>and</i> operator	23
Logical <i>or</i> operator	23
Logical <i>xor</i> operator	23
Logical <i>not</i> operator	23
Mathematic operation on arrays	24
Copies between arrays with data conversion	24
Shifts/moves data within an array	24
Adds a single value or an array to another array	24
Subtracts a single value or an array from another array	24
Multiplies an array with a single value	24
Divides an array by a single value	24
Calculates amplitude and angle from a ADC-array	24
String handling	25
conversion of numeric expressions into strings	26
conversion of strings into numbers	26
returns the ASCII value of a character	26
converts ASCII code into its character	27
returns the length of a string	27
cuts string into pieces	27
returns the position of a sub-string	28
Other Functions and System variables	29
returns a random number	29
returns the current date, real time clock and timer	29
returns type and line number of a runtime error	30
Program and Memory Organization	31
cold-start of a program	31
warm-start of a program	31
direct start of a program from disk	31
interrupt a program & single step mode	32
continue after stop	32
switch off single step mode	32
comment basic statements	32
switch off single step and trace mode	32
pause program execution	33
list program code	33
list variables	34
list arrays	34
Search for a character string	34

delete program lines	34
initialize program RAM	35
clear variables	35
clear program code lines	35
clear stack	35
automatic line numbering	36
move program lines	36
copy program lines	37
renumber program lines	37
swap contents of variables	37
free memory	37
Text Handling	38
print characters to the screen	38
tabulator	39
blanks	39
cursor position	39
format text output	39
create formatted text	41
Jumps & Branches	42
Labels and line numbers	42
Jump Commands	43
obligatory jump	43
obligatory jump / return to/from subroutine	43
Local functions	45
one-line functions	45
multiple-line functions	46
Conditional Branches	48
conditional branch	48
conditional branch (extended version)	49
branch on branch table	49
branch on error number	51
branch on interrupt	51
branch on nmi	52
logical program end	52
stopping / continuing program execution	53
remark	53
Loops	54
count loop	54
conditional loop	55
conditional loop	56

Graphics **57**

Sets the plot parameters for Graphics-mode text	57
Sets the relative origin of coordinates	58
Sets the current plot position	58
Plots straight lines (vectors)	59
Plots rectangle Edges	60
Plots circle	60
Plots circular arc	61
Plots input window	61
Plots bar graph	61
Plots curves from a matrix	62
Reverses black and white	63
Clears the Screen Graphics	64
Switches the Micro-Cursor on/off	64
Gets mouse coordinates	64
Sets mouse coordinates	65
Defines a Graphic Window	65
Fills a Graphic Window	66
Clears a graphic window	66
Defines macro of plot-instructions	67
Outputs the Complete Screen to the Printer	68

System Level Links **69**

Reads a memory location	69
Writes to a memory location	69
Calls an assembler routine	70
Calls an assembler routine with data transfer	70
Pauses the execution of a program	71
Calls the system monitor	71

External Devices **72**

Introduction	72
Device Number <i>Dev</i>	72
Drive Number <i>Drv</i>	72
Logical Address <i>LAdr</i>	72
Secondary Address <i>SAdr</i>	73
File Name	73
Storage Instructions	73
Device Numbers and File Types	73
Opens a Logical Channel	74
Opens a COM-Port by a Logical Channel	75
Closes a Logical Channel	75
Saves/Loads on Logical Address	75
Input on a Logical Address	76
Input on a Logical Address / Keyboard	76

Multiple Input on a Logical Address	76
Output on a Logical Address	77
Show File List	77
Set Directory	78
Delete File	78
Concatenate Sequential Files	78
Saves/Loads Programs	79
Loads and Starts a Program	79
Saves/Loads a Program as an ASCII File	79
Opens a Logical Channel	80
Closes a Logical Channel	80
Renames a File	81
Copies files	81
Initializes Peripheral Devices	81
Disk Status (System Variable)	81
Signal Acquisition	83
Read Analog Input Channel	83
Reads an 8-Bit-Digital Input Channel	83
Sets an Analog Output Channel	83
Sets an 8-Bit-Digital Output Channel	83
Sets an 8-Bit-Digital andiBus Address	83
Measurement Scan (Transient Recording)	84
Generates analog signals from data-arrays	85
Averaging single variables or whole data records	87
Fast Fourier Transform	88
Appendix	91
ANDIbasic Statements	91
ANDIbasic System Variables	96
Disk Error Messages (ds, ds\$)	97
ANDIbasic Error Messages (Errtype)	98

General

Direct mode & program mode

You enter the **IM** environment by clicking (double-clicking) the **Thales** icon on the Windows desktop. The **IM** environment comes up with the start-up screen. Here you have the choice to go to the **andiBASIC** editor (key), the programming environment, or to start the **Thales** software (<d> key). The **Thales** software is described in the **Thales** manual. Here we focus on the **andiBASIC** editor.

When you entered **andiBASIC** mode by pressing the key the IM show the message:

```
*** andibasic 68k v xxx ***
yyyyy kbytes free

ready.
_
```

xxx indicates the **andiBASIC** version, yyyyy informs you of the free memory.

You are in the **andiBASIC** direct input mode now. The flashing cursor (_) indicates that the computer is ready to accept an input from the keyboard. Statements that you input will be executed, as soon as the **ENTER** key is pressed. If you write a number in front of a statement, this will be interpreted as a new line of an **andiBASIC** program. In this case, the statements will not be executed directly after pressing the **ENTER** key. You rather have to start the program with the **run** or **goto** statement. When executing a program, the IM no longer accepts the input of statements.



To execute a program, the statements **run**, **start** or **goto** must be used in direct mode (without a line number) and are terminated with the **ENTER** key. The statements **run**, **start** or **goto** can also be used in program code, so that a routine can start another one.

All **andiBASIC** instructions can be used either in direct or program mode, even if some of them make no sense in a program (e.g. **auto**, **cont**, **clp**, **delete**, **renumber** etc.). Some other statements make no sense in direct mode (e.g. **rem**, **data**, **end** etc.). Character strings that cannot be interpreted as an **andiBASIC** statement are interpreted as a string of ASCII characters. In the same way character strings in between double quotation marks ("...") always are interpreted as ASCII characters. In program mode all character following the **rem** statement are also interpreted as ASCII characters.

Structure of an andiBASIC program

At the beginning of an **andiBASIC** program line you always have to input a program line number. Program line numbers have to be in the range of 0 to 65535. If you exceed this range the error message:

```
line number too big error
```

will be displayed.

Lines may be input in any order, but must be confirmed by pressing the **ENTER** key. The **andiBASIC computer** sorts the lines automatically. A program code listing will show the lines in ascending sequence. If you input program code lines with the same line number several times, the program code line input at last (and confirmed with the **ENTER** key) will be memorized.

One program code line may contain more than one **andiBASIC** statement, separated by a **colon (:)**.

```
10 dima%(9):a=10:b=100
```

The amount of characters is limited by the length of a line on the screen. Two lines may be linked by writing through the end of a line to the beginning of the next line. You also may run the backspace key from the beginning of the second line to the end of the first line.

clp

clear program code lines

A line or a number of subsequent lines can be cleared using the **clp** statement. If the statement is used without a line number the complete program is cleared. With one line number, only this line is cleared. Defining two line numbers, all line numbers in between these numbers (including the indicated ones) are cleared.

Syntax	clp	clears entire program
	clp, <i>linenumber</i>	clears line number <i>linenumber</i>
	clp, <i>start-end</i>	clears line numbers <i>start</i> to <i>end</i>

You also can clear a program code line by typing its line number in an empty line and confirm with the **ENTER** key.

end

logical end of a program

The execution of a program is finished, if either its physical or its logical end is reached. The physical end is reached with the last program code line, the logical end is reached with the **end** statement. In both cases the operating system terminates the program execution and jumps back to the direct input mode. You can read

```
ready.
```

on the screen. The **end** statement is only needed, if the physical and the logical end of the program are not the same.

Executing an ANDIbasic program

The main **andiBASIC** statements for program handling are:

```
run
stop
new
list
```

run

execute ANDIbasic program

rU

The **run** statement executes the **andiBASIC** program in memory. The program can be started at any desired position. The line number to start at is defined behind the **run** statement. If no line number is defined the program will be executed from the beginning.



run clears all previously defined variables except common variables. If you want to keep all variables, start program execution with the **goto** statement.

Syntax

```
run zn
run
```

starts program execution at **andiBASIC** line number *zn*
starts program execution from the first **andiBASIC** line number

stop

interrupt program execution

The **stop** statement interrupts the execution of a running program and jumps to **andiBASIC** direct mode. The display shows:

```
break linenumber text
ready.
—
```

If a program was interrupted by the **stop** command **andiBASIC** enables the **Single-Step Mode**. Here you can trace through the logically following statements by pressing the **ENTER** key. You exit the **Single-Step Mode** with the **off** statement.

If you interrupt a program by pressing the **BREAK** key (**F12**) the **Single-Step Mode** is NOT activated. The IM finishes the current command and then jumps to direct mode.

The display shows:

```
break linenumber ... ready.
—
```

If you input **stop** in direct mode, the **Single-Step Mode** is activated.

After a program was interrupted by a **stop** statement or the **BREAK** key, it can be continued with the **cont** statement.

new

Initialize program RAM

The **new** command clears the program in memory and initializes all variables and pointers. If you use **new** in a program code the program erases itself as soon as it runs on the **new** command.

Syntax

```
new
```

list

list an ANDIbasic program

II

This command lists the code of the resident program to the screen. Optionally you can define **andiBASIC** line numbers to limit the listed block.

Syntax

<code>list</code>	lists the whole Program
<code>list start - end</code>	lines no. <i>start</i> through <i>end</i>
<code>list - end</code>	lists from the first line up to line no. <i>end</i>
<code>list start -</code>	lists from line no. <i>start</i> to the last line

In connection with the `cmd` command listing can be output on peripheral equipment (printer, floppy disk). Please see `cmd` command.

print

print characters to the screen

?

This statement prints text on the screen. The text may be defined by any sort of constants or variables (string or numerical) or by the result of an operation or function. The abbreviation of the `print` command is the question mark "?".

Syntax

<code>print expression</code>
<code>? expression</code>

expression can be a numeral or string. String constants must be enclosed in quotation marks.

Examples

```
print "stringconstant" ENTER
stringconstant
ready.
```

```
? 12 * 2 ENTER
24
ready.
```

```
A$="stringvariable" ENTER
print A$ ENTER
stringvariable
ready.
```

```
A=10 : B=5 ENTER
? A * B ENTER
50
ready.
```

The `print` statement accepts more than one argument. Valid separators are the semicolon (;) and the comma (,). Use the semi-colon if all arguments should be printed one after the other (without a separating space). Use the comma, if a 10 spaces tabulator should be used for each argument. This way a simple formatting of the output is achieved.

Examples

```
? 1;2;3;4 ENTER
1234
ready.
```

```
? 1,2,3,4 ENTER
1      2      3      4
ready.
```

```
A$="my name is":B$="Willy"
? A$,B$
my name is      Willy
ready.
```

Pressing the **HOME** key will position the cursor in the uppermost left corner of the text window (in case of a text window is defined). Pressing **HOME** a second time will open an existing text window an position the cursor in the uppermost left corner of the **Thales** window.

Pressing the **END** key once will clear the text from the screen, pressing it a second time will clear the graphics too.

These functions can also be used in a program. You have to use the **print** statement and input these keys in between quotation marks. The function is not executed immediately, but when the **print** statement is executed.

lprint

print characters to the logfile / ztrace

lpR

This statement prints text to the logfile. The text may be defined by any sort of constants or variables (string or numerical) or by the result of an operation or function. Z-Trace displays the logfile in realtime for diagnostics (Manual "Diagnostics and Troubleshooting")

Syntax **lprint** *expression*
 lpR *expression*

expression can be a numeral or string. String constants must be enclosed in quotation marks.

Examples

```
lprint "stringconstant"
ready.
```

```
A$="Testvalue: ":B=1.2345e-3
lpR A$;B
ready.
```

The **lprint** statement accepts more than one argument. Valid separators are the semicolon (;) and the comma (,). Use the semicolon if all arguments should be printed one after the other (without a separating space). Use the comma, if a 10 spaces tabulator should be used for each argument. This way a simple formatting of the output is achieved.

Examples

```
lprint 1;2;3,4,5
ready.
```

Data types, Variables, Constants

Data types

For a computer there are principle differences between numerical data (numbers) and strings (text). With numerical data it can carry out mathematical operations, with strings commonly not.

Integer numbers

The **IM6** handles two different integer number formats:

16-bit integer format is used with integer arrays. This is convenient, because 16-bit integers come from most of the measurement modules of the **IM** system. Their range is -32768 to $+32767$.

32-bit integer format is used with single integer variables. Their range is $-2.147 \cdot 10^9$ to $+2.147 \cdot 10^9$.

Real numbers

Real numbers contain all kind of real numbers in the range of -10^{308} to $+10^{308}$. The smallest fraction is $\pm 10^{-308}$. In the IM system they are displayed with an accuracy of 15 significant digits (double precision).

Real numbers can be printed in two different formats: the normal and the exponential format. The IM system changes the format automatically if needed. The exponential printout is used if the real number cannot be represented with 9 digits left of decimal point or if it is a smaller fraction than 0.01.

Complex numbers

Complex numbers in principal are a combination of 2 real numbers which contain the real and the imaginary parts. Each part has the same range as the real numbers. A complex number is displayed as two real numbers separated by a comma. The first number represents the real part, the second represents the imaginary part.

Byte numbers

Byte numbers are integer numbers which can be represented by one byte. They have a range of 0 to 255 ($= 2^8 - 1$). Byte numbers are used for the **peek()** and **poke** statements for example, which allow access to single bytes in memory. Byte numbers have no reserved variable types, but are usually stored within strings or as integer numbers.

Logical data

Logical data are produced by logical operations (**and**, **or**, **xor**, **not**) and comparison operations (**>**, **<**, **><**, **<>**, **=**). The result can only have the values of 0 (false) and -1 (true). Logical data have no reserved variable types, but are usually stored as integer or real numbers.

Strings

String constants are principally enclosed into quotation marks ("**text**"). They can contain all printable characters as well as the screen command codes (refer to chapter "command codes"). Spaces are not ignored but also displayed. The IM system can process strings of a length up to 65535 characters.

	real-variable (no add-on)
%	integer variable (including byte variables and logical variables)
&	complex variable
\$	string variable

Depending on the range of numerical variables or the current contents of string variables, the different types of variables occupy differing amounts of memory space.

integer array	6 byte + 4 byte/dimension + 2 byte/element
integer single	2 byte + 4 byte
real array	6 byte + 4 byte/dimension + 8 byte/element
real single	8 byte
complex array	6 byte + 4 byte/dimension + 16 byte/element
complex single	16 byte
string array	6 byte + 4 byte/dimension + 1 byte/character
string single	6 byte + 1 byte/character+(1 byte word alignment if necessary)

Dimensioning Variables

The IM system handles three different priority levels for all the variable types described above:

1. local variables
2. dim variables
3. com variables

LOCAL variables are defined for a single local subroutine such as functions and procedures and are valid only in this subroutine. As soon as the subroutine is left, all local variables lose their contents. Local variables are defined in between the brackets following the function name or as a list of variable names after a **local** statement.

DIM variables are valid throughout the whole execution of a program. Single variables and arrays of up to 11 elements and 11 dimensions can be used without defining them with the **dim** statement. Arrays with more than 11 elements or more than 11 dimensions must be defined before use with the **dim** statement. If such an array is not defined or you address more than the defined elements or dimensions, the IM system shows the error message **BAD SUBSCRIPT ERROR**.

Repeated dimensioning of a variable or array is possible. The contents is cleared then.

Variables and arrays may be cleared by using the **clr** statement. Other **andiBASIC** statements which clear variables are: **com**, **clr**, **run**, **new**.

If you run a program with the **run** statement, all variables will be cleared automatically. If you want to avoid this you can start a program with the **goto** (basic line number) statement. Make sure then that you jump to a program location where no dim statements are executed anymore.

COM variables are not only valid throughout the execution of one program but are also not cleared by **run**, **load**, **start**, **dload** and **dstart**. They can be used to transfer data from one program to another. com variables are only cleared by the **clr** and **new** statements. They must be defined using the **com** instruction **before** any other variables are assigned. Only one line (or a linked double-lines) can be used for the com definitions. Further **com** statements (with or without variable list) will erase dim and local variables. A following list of variables will be ignored.

Conversion of Data types

Most of the data types described can be converted to the other types. Numerical data need no explicit conversion function call: integer numbers can be converted to real variables, real numbers can be converted to complex numbers etc. by simply assigning them.

Examples

```
a%=10:b=a%:?b
10
```

ENTER

```
a=14.999:b%=a:?b%
14
```

ENTER

```
a%(0)=300000:?a%(0)
overflow error
```

ENTER

Converting complex to real numbers or complex to integer numbers, only the real part of the complex number is transferred to the real or integer variable, respectively. Vice versa, the value of the integer or real number is transferred to the real part of the complex variable, whereas the imaginary part is set to zero.

Internally, the IM system handles no integer constants. All numerical constants are handled as real (floating point) numbers. This means: as soon as a numerical constant appears in a mathematical calculation, the whole calculation will be carried out in the real domain. Integer mathematics is only possible, if nothing but integer variables are found. This must be considered when calculating with large numbers. Intermediate results, too, must be in the valid range of the data type used.

Examples

```
a%=2e5:b%=a%*1e6/1e6:print b%
2e+5
```

ENTER

```
a%=2e5 : b%=1e6 : c%=a%*b%/b%
overflow error
```

ENTER

In the first example, floating point mathematics is used for calculation automatically. Thus, the intermediate result ($b\% = a\% * 1e6 = 2e11$) remains within the permitted range (up to 10^{308} for real numbers). Example 2, however, calculates with integer mathematics, because only integer variables are used. Although the final result does not exceed the permitted range of values, the computer displays **OVERFLOW ERROR**. This happens because the intermediate result ($c\% = a\% * b\% = 2e11$) exceeds the permitted range of values (up to $2.147 * 10^9$ for integer numbers). Intentionally use of integer mathematics may speed up time consuming program loops. Nevertheless, most types of data may be converted into another data type format. The following conversions are possible:

val	string into number
cval	String into complex number
str\$	number into string
cmplx	real- and imaginary part into complex number
real	real part of a complex number into real number
imag	imaginary part of a complex number into real number
asc	character into ASCII value
chr\$	ASCII value into character
val	numerical string into numerical value

val ()

convert the numerical value of a string to a number

The `val ()` function converts the numerical value of a string into a number. If the string is beginning with a non numerical character, the function returns a zero. If a non-numerical character follows on some numerical characters, the function converts the numerical values left of the non-numerical character. As it is usual for the output of numerical data an empty space is left for a potential sign. With positive numbers it is suppressed, with negative ones it is displayed. The only non-numerical characters utilized by the VAL-instruction are plus and minus signs in front of a number, decimal point and the character "e" for recognizing exponential format.

Syntax `val ("string")`

Examples

```
print val (" +1234")     ENTER
1234
```

```
print val ("abc")       ENTER
0
```

```
print val ("a123")      ENTER
0
```

```
print val (" -12a34")   ENTER
-12
```

cval ()

converts the numerical argument(s) of a string to number(s)

The `cval ()` function has two applications. The more straightforward one converts two numerical arguments within a string, separated by a comma, into a complex variable. The rules for each argument are analogous to the single argument `val ()` function.

Syntax `cval ("real part string", "imaginary part string")`
 `cval ("multiple number string", Position)`

Examples

```
print cval (" +1234, -1") ENTER
1234, -1
```

The second application supports the analysis of several numerical arguments within a string, each underlying the rules analogous to the single argument `val ()` function. The parameter *Position* indicates, where the analysis has to start within the string. The function returns a complex value, where the real part belongs to the converted number string. The imaginary part is a pointer to the position of the next number start after non-numeric characters (separator). If the returned number is the last one within the list, the returned imaginary part equals the string length+1. `cval ()` can be used to extract a series of numbers from a string iteratively, for instance in table calculation.

```
print cval ("123,456", 1)     ENTER
123, 5
```

```
print cval ("123,456", 5)     ENTER
456, 8
```

str\$() **convert numbers to strings**

str\$() converts numbers to a string. The empty sign place is converted as an empty space for positive numbers or as a minus sign for negative numbers.

Syntax `str$(number)`

Examples

```
print str$(123)      ENTER
123
```

```
print str$(-1.234)      ENTER
-1.234
```

cmplx() **fill a complex constant to a complex variable**

The command converts two numbers to real and imaginary part of a complex number. With this data type complex mathematics can be carried out. `real` and `imag` must be numerical expressions.

Syntax `cmplx(real, imag)`

Example

```
a&=cmplx(3,4*10):print a&*3      ENTER
9, 120
```

real() **real part of a complex variable**
imag() **imaginary part of a complex variable**

Syntax `real(complex variable)` returns real part of a complex variable.
 `imag(complex variable)` returns imaginary part of a complex variable.

Example

```
C&=cmplx(5,7)      ENTER
print real(c&), imag(c&)      ENTER
5      7
```

asc() **ASCII to code conversion**

returns the ASCII-value of a character string. ASCII (=American standard code for information interchange) is the numerical code which contains all displayable characters and command codes. If the argument contains more than one character, the `asc` function only returns the value of the first character.

Syntax `asc("character")`

Examples

```
print asc("0")      ENTER
48
```

```
print asc("a")      ENTER
65
```

```
print asc("bcd")      ENTER
66
```

chr\$ () code to ASCII conversion

converts byte data to the corresponding ASCII-characters. This way, even non-displayable codes may be printed (e.g. `chr$(10)+chr$(13)` = LINE FEED + RETURN). Please refer to the table of ASCII-codes for the complete list of ASCII codes.

Syntax `chr$(number)`

Examples

```
print chr$(48)                    ENTER
0
```

```
print chr$(65)chr$(77)chr$(79)chr$(83)                    ENTER
amos
```

```
print chr$(27)chr$(28)                    ENTER
>screen is cleared<
```

```
Print chr$(27)"g"                    ENTER
>character set is changed from ASCII to German<
```

Conversion between Numerical Systems

hex () hexadecimal to decimal conversion
hex\$ () decimal to hexadecimal conversion

With the `hex` and `hex$()` statements you can convert data from the hexadecimal to the decimal numerical system and vice versa. A hexadecimal argument must be set in between quotation marks. A hexadecimal result string has a \$ at it's beginning.

Syntax `hex("string")` converts from hexadecimal to decimal
 `hex$(number)` converts from decimal to hexadecimal

Examples

```
print hex("a0")                    ENTER
160
```

```
print hex$(10)                    ENTER
$a
```

These functions may e.g. help handling memory-addresses. They can also be used together with memory addressing commands such as `peek`, `poke` or `usr`.

Example

```
poke hex("a0100"),hex("10")                    ENTER
ready.
```

Conversion between Character Code Systems

`tcode$()` String to string conversion

`tcode$()` provides a series of versatile functions to convert strings. The function executed depends on the arguments.

Syntax `tcode$("string", "image-string")` converts *string* to *image-string*.
 `tcode$(control, "string")` converts *string* according to the rule, selected by *control*.

Case the first argument is string type:
 The byte value *b* of each character within *string* is replaced by the *b*-th. Character of *image-string*. *image-string* is used as look-up-table: an value *b* of zero returns the first character of *image-string*, a value *b* of one returns the second character of *image-string* and so on. The following example assumes, that the image-string *image\$* contains all characters from ascii zero to 255 in ascending order, except for the first 64 characters, which are replaced by spaces (ascii 32).

Example

```
print tcode$("ABC0123abc?!$ABC", image$) ENTER
ABC    abc    ABC
```

If the image string is shorter than an occurring byte value within the source string, the corresponding character remains unchanged.

Example

```
print tcode$(chr$(0)+chr$(1)+"x", "ab") ENTER
abx
```

A typical application of `tcode$()` with two string arguments is converting strings to all-upper-case or all-lower-case character strings, or changing all non-numerical characters to spaces. In this kind of `tcode$()` the result string has always the same length as the source string.

In case the first argument is numeric:
 The value of *control* decides about the function performed.

Control	Function
1	Conversion of Windows- to Andibasic strings
0	Conversion of Andibasic- to Windows strings
-1	Deletes all control characters
-2	Deletes all control characters and spaces

Examples

```
print tcode$(0, "0123 abc ABC") ENTER
0123 ABC abc

print tcode$(-2, "0123 abc ABC") ENTER
0123abcABC
```

`tcode$()`-functions with a numeric control argument may change the string-length. Besides some language-specific characters, the string-constant representation of **andiBASIC** - and Windows-ASCII-code differ in a way, that upper case characters are swapped with lower case characters. The `tcode$()`-function for control-values of zero and one provide an easy way to change between both representations. The `tcode$()`-functions for negative control-values support the keyword-analysis within strings.

Examples

```
10 open1,65,1,"c:\thales\temp\ascii.txt,w"  
20  A$="Hello ASCII-World!"  
30  print#1,tcode$(0,A$)+hex$("0d0a"); :rem convert to ASCII-format  
40 close1 :rem and append CR + LF
```

Create ASCII formatted text files with the tcode\$()-function. For line formatted output a carriage return (CR) and a line feed (LF) is added.

Mathematics

let

assignment

In mathematics, a value, can be assigned to a variable using an equation.

Syntax `let A = expression`
 `A = expression`

In this case, the equal sign means that the quantity on the left side is equal to the quantity on the right side. The variable "A" now has the value of the expression. Mathematically, the equal sign is not correct if the left and the right side do not have the same value.

This is different in most programming languages: The equal sign merely symbolizes an assignment of the quantity on its right side to the one on its left side. For this reason, the statement `let` is introduced. In contrast to mathematics, the following line is allowed in Basic:

```
let a = a * 10
```

Left and the right side are not equal. First, the value of the right side is calculated: `a * 10`. Then, the result is assigned to the variable on the left side. That is why the same variables may appear on both sides of the equal sign. This way, the value of a variable is newly assigned. The instruction `let` can be omitted. It was taken up only for reasons of compatibility with other Basic dialects. In the case of the above example, it is also possible to write:

Example

```
a = a * 10
```

Mathematical rules

In *andiBASIC*, the same mathematical hierarchy is valid as in mathematics :

- 1. priority: **exponentiation (^)**
- 2. priority: **multiplication and division (* and /)**
- 3. priority: **addition and subtraction (+ and -)**

Two differences in the calculation signs should be pointed out :

The division sign in Basic is the slash "/".

The exponential sign (to the power of) is the upwards arrow "^".

Examples

```
print 10 - 3 * 2      ENTER  
4
```

```
print 2 + 5 ^ 2 * 3      ENTER  
77
```

This hierarchy can be broken in the same way as in mathematics: by the use of brackets. Only rounded brackets "(" and ")" are allowed. The IM system is able to handle up to 256 levels of brackets. For practical reasons (i.e. line length max. 160 characters) a maximum of 20 levels is allowed.

```
Print (10 - 3) * 2      ENTER  
14  
print (2 + 5) ^ (2 * 3) ENTER  
117649
```

The bracket levels are managed in the 4 kbyte *function stack*. Furthermore, the fn- (function) addresses are stored here. This means that the amount of bracket- and fn levels cannot exceed 256.

Mathematical Operations

The four fundamental operations addition, subtraction, multiplication and division are to be handled in the same way as in mathematics:

Examples

```
? 4 + 5
a = 200 - 12
print 6 * 7
b = 20 / 5
```

The exponential calculation too is to be handled similarly to the mathematical rules.

```
print 10 ^ 2
print 2 ^ 10
print 100 ^ 0.5
print 10 ^ -2
```

The examples show, that fractional and negative exponents are allowed. Fractional exponents are equivalent to a root function. The example (100 ^ 0.5) is equivalent to the square root of 100. Negative exponents are equivalent to 1 divided by the result with positive exponent.

```
print 100 ^ 0.5      ENTER
10
```

```
print 10 ^ -2       ENTER
0.01
```

```
Print 100 ^ -0.5    ENTER
0.1
```

Mathematical Functions

abs() absolute value function

abs() returns the absolute value of a real number. It ignores the sign if negative.

Syntax **abs**(*Numerical expression*)

Example

```
?abs(-5.25),abs(5.75),abs(1.25*5-100) ENTER
5.25 5.75 93.75
```

int() integer value function

int() converts real expressions into integer numbers. The next smaller integer is returned without rounding. The result is real number (no integer number!). Notice the result of a negative expression returning of the next smaller whole number.

Syntax **int**(*numerical expression*)

Example

```
?int(-5.25),int(5.75),int(1.25*5-100) ENTER
-6 5 -94
```

round() 5/4 rounding function

Round converts real expressions into the nearest 5/4 rounded number. The result is a real number (no integer number!).

Syntax **round**(*numerical expression*)

Example

```
?round(-5.4),round(5.4),round(-5.5),round(5.5) ENTER
-5 5 -6 6
```

frac() fraction value function

frac() strips real expressions from their integer share. The operand sign is saved.

Syntax **frac**(*numerical expression*)

Example

```
?frac(-5.25),frac(5.75),frac(100) ENTER
-.25 .75 0
```

sgn() sign function

sgn() returns the sign of a number and displays the result as 1 for a positive value and -1 for a negative value. If the argument value is zero, the return value is also zero.

Syntax **sgn**(*numerical expression*)

Example

```
?sgn(-5.25),sgn(5.75),sgn(1.25*5-100),sgn(2-2) ENTER
-1 1 -1 0
```

sqr ()

square root function

sqr () returns the square root of a numerical argument. A negative argument returns a complex value.

Syntax **sqr** (*numerical expression*)

Example

```
?sqr(16),sqr(1.5625),sqr(-4)      ENTER
4      1.25      0,-2
```

exp ()

eⁿ function

exp () calculates the function **eⁿ**, where **e** is the "Euler constant" (2.7182183) and **n** is a numerical expression. **n** can go to a maximum of 710. If it is higher the system shows an OVERFLOW ERROR.

Syntax **exp** (*numerical expression*)

Example

```
?exp(10)      ENTER
22026.4657948067
```

log ()

Natural logarithm

ld10 ()

10-based logarithm

log () returns the natural logarithm (base **e**), **ld10 ()** the logarithm with base 10. The argument must have a value unequal zero, otherwise the system will report an error message. A negative argument will cause automatic conversion to complex.

Syntax **log** (*numerical expression*)

Example

```
?log(10),ld10(10),log(-1)      ENTER
2.30258509299405      1      0, 3.14159625358979
```

Trigonometrical Functions

<code>sin()</code>	sine
<code>cos()</code>	cosine
<code>tan()</code>	tangent
<code>cot()</code>	cotangent

`sin()`, `cos()`, `tan()` and `cot()` calculate the trigonometric functions sine, cosine, tangent and cotangent. The argument has to be defined in radians. If the angle (degrees) is given, it first has to be converted to radians by the following formula:

$$\text{Radians} = \text{Degrees} * \pi / 180$$

Syntax

```

sin(numerical expression)
cos(numerical expression)
tan(numerical expression)
cot(numerical expression)
    
```

Example

```

?sin(2*pi),cos(pi-1)  ENTER
0      -.54030230586814
ready.
    
```

Under **Thales** you print π by pressing the “ β ” key on the computer keyboard. The value of π is 3.14159265...

<code>asin()</code>	arc sine
<code>acos()</code>	arc cosine
<code>atn()</code>	arc tangent

These functions calculate the **arc()** functions of the corresponding argument. The result is returned in radians. For converting it to degrees use the following formula:

$$\text{Degrees} = 180 / \text{Pi} * \text{Radians}$$

Syntax

```

asin(numerical expression)
acos(numerical expression)
atn(numerical expression)
    
```

<code>sinh()</code>	area sine
<code>cosh()</code>	area cosine
<code>tanh()</code>	area tangent
<code>coth()</code>	area cotangent
<code>atnh()</code>	area arc tangent

These functions calculate the hyperbolic sine, cosine, tangent and cotangent. The syntax corresponds to the trigonometrically fundamental functions. Here the argument must also be defined in radians. A conversion formula is described above.

Syntax

```

sinh(numerical expression)
cosh(numerical expression)
tanh(numerical expression)
coth(numerical expression)
atnh(numerical expression)
    
```

Functions with complex arguments

<code>cabs()</code>	complex modulo
<code>cround()</code>	performs 5/4 rounding on real- and imaginary component
<code>carg()</code>	complex argument
<code>cswap()</code>	exchanges real component vs. imaginary component
<code>conjg()</code>	inverts the sign of the imaginary component
<code>cquad()</code>	returns quadrant number
<code>cpow()</code>	
<code>cvec()</code>	converts from polar coordinates to rectangular coordinates
<code>cpol()</code>	converts from rectangular coordinates to polar coordinates
<code>csqr()</code>	complex square root
<code>cexp()</code>	complex exponential function
<code>clog()</code>	complex natural logarithm
<code>csin()</code>	complex sine
<code>ccos()</code>	complex cosine
<code>ctan()</code>	complex tangent
<code>ccot()</code>	complex cotangent
<code>casin()</code>	complex arc sine
<code>cacos()</code>	complex arc cosine
<code>catn()</code>	complex arc tangent
<code>csinh()</code>	complex hyperbolic sine
<code>ccosh()</code>	complex hyperbolic cosine
<code>ctanh()</code>	complex hyperbolic tangent
<code>ccoath()</code>	complex hyperbolic cotangent
<code>catnh()</code>	complex hyperbolic arc tangent

Comparison Operators

<	less than
>	greater than
=	equal to
<>	not equal to

As a computer must be able to make decisions, it must be able to compare data. This is done by 4 comparison operators:

<	less than
>	greater than
=	equal to
<> or ><	not equal to

In addition the following combinations are allowed:

=< or <=	less than or equal to
=> or >=	greater than or equal to

Syntax `expression1 OP expression2`

OP = comparison operator

Comparison operators often are user together with the `if` statement.

Example

```
10 input "a,b";a,b
20 if a > b then print "a is greater than b"
```

The result of a comparison is a logical value:

0 = false
-1 = true

This makes it possible to assign the result of a comparison to a variable for further calculations:

Example

```
b = 10 : c = 20
a = b < c : ?a
-1
ready.
```

ENTER



For complex expressions only the "=" -operator is allowed !

Logical Operators

and	Logical <i>and</i> operator
or	Logical <i>or</i> operator
xor	Logical <i>xor</i> operator
not	Logical <i>not</i> operator

and, **or**, **xor** and **not** execute the corresponding logical operations with all the bits of integer or real numeric arguments. If an argument format differs from 32-bit integer, an automatic conversion is involved.

The truth tables of the operations are as follows:

and	or	xor	not																																
<table border="1"><tr><td></td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>		1	0	0	0	0	1	1	0	<table border="1"><tr><td></td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>		1	0	0	1	0	1	1	1	<table border="1"><tr><td></td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr></table>		1	0	0	1	0	1	0	1	<table border="1"><tr><td></td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table>		1	0	0	1
	1	0																																	
0	0	0																																	
1	1	0																																	
	1	0																																	
0	1	0																																	
1	1	1																																	
	1	0																																	
0	1	0																																	
1	0	1																																	
	1	0																																	
0	1																																		

Syntax *expression1* **OP** *expression2* **OP** = logical operator
 not *expression*

Besides the decimal values, the binary pendants of the arguments are also shown in the following examples in order to demonstrate the functions clearly. Please remember, that each of the 32 bits is separately combined.

operator	decimal	binary
	-2	1111 1111 1111 1111 1111 1111 1111 1110
and	3	0000 0000 0000 0000 0000 0000 0000 0011
result	2	0000 0000 0000 0000 0000 0000 0000 0010

operator	decimal	binary
	-2	1111 1111 1111 1111 1111 1111 1111 1110
or	3	0000 0000 0000 0000 0000 0000 0000 0011
result	-1	1111 1111 1111 1111 1111 1111 1111 1111

operator	decimal	binary
	-2	1111 1111 1111 1111 1111 1111 1111 1110
xor	3	0000 0000 0000 0000 0000 0000 0000 0011
result	-3	1111 1111 1111 1111 1111 1111 1111 1101

operator	decimal	binary
not	-2	1111 1111 1111 1111 1111 1111 1111 1110
result	1	0000 0000 0000 0000 0000 0000 0000 0001

String handling

Strings are sequences of displayable characters as well as control characters. All characters can be represented in ASCII code. Two kinds of string representations exist: *string constants* and *string variables*.

String constants have to be enclosed in quotation marks. Please note, that the terminal switches to the "control mode" when opening a quote. In this case, most of screen edit functions (e.g. **Pos1** or **End**) are not carried out directly but are inserted into the string as control codes. They are visible as special signs. Each edit key has its own icon. The corresponding edit functions are executed when the string is printed.

Example

```
10 print"End" ENTER
run
```

The screen will be cleared only when the string constant is printed (when the program executes line 10 after the `run` statement), not when the key is pressed as it is the case in normal mode.

String variables are used for dynamic string management. The name of a string variable must end with a `$` sign:

variablename\$ (Refer to chapter E.II.2 "Variables").

The length of a string is limited to 65500 characters.

Assigning a string constant to a string variable has the following Syntax:

Syntax `variablename$ = "abcde..."`

To paste two strings use the `+` operator:

Syntax `string$ = string1$ + string2$ + string3$ + ...`

Examples

```
10 Name$ = "Dr." + " Herbert " + " Meier"
20 print Name$
run

Dr. Herbert Meier
```

```
10 for i=0 to 5
20 a$ = a$ + str$(i)
30 next
40 print a$
run

012345
```

```
10 print chr$(27) + "s0run" + chr$(13) + chr$(0)
20 rem loads the function key 1 with the run command
```

str\$()
conversion of numeric expressions into strings

`str$()` converts numeric expressions into strings.

Syntax `str$(number)`

number is a integer or real variable, constant or expression. The output format of a string-converted number is identical to the number format itself, except that no additional space is left after the number. The sign location in front of the number is kept. However, this space is filled only in the case of negative numbers, the same way as in the standard number format.

Example

```
10 a = -10.5 : b = -5.3 : c = 23
20 print a;b;c, str$(a);str$(b);str$(cos(c*3))
30 plot"da",100,100,str$(b+c)
```

As shown in this example, mathematical expressions and functions may also be used as arguments. This way, the result of the computation is converted. The `str$()` function is used to have numbers available where only string data are allowed (e.g. plot commands etc.).

val()
conversion of strings into numbers

`val()` is the reverse function of `str$()`, i.e. it converts strings into numerical values.

Syntax `val(string)`

string is a string expression, a string variable or a string constant. *string* is examined from left to right, during conversion, for a non-numerical character. Up to that character, *string* is converted into a numerical value. 0 is transferred in the following cases:

""	empty String
" - e10"	missing mantissa
" "	blanks
" 0000.0"	zero
"z123456"	first character is not numerical

Therefore, if `val()` returns a 0, you can not define the contents of *string*. The content of *string* after a non-numerical character is not important for the conversion of `val()`. It is, however, not possible to determine up to which point `val()` has examined the string.

asc()
returns the ASCII value of a character

`asc()` converts the first character of a string into its ASCII value.

Syntax `asc(string)`

string is any given string constant, variable or expression that is not empty. Remember that only the first character is converted. If other characters than the first should be converted you have to select that character by means of the `mid$()` command, first. An empty string produces the value 0.

Examples

```
print asc("a")
65
print asc("abc")
65
print asc("")
0
```

```
10 input "String";String$
20 for i=1 to len(String$)
30 print i,mid$(String$,i,1),asc(mid$(String$,i))
40 next
50 goto 10
```

For any string input, the program above outputs the ASCII codes of all the characters. The length parameter can be omitted at the second mid\$ because ASC() only transfers the first character of the (MID\$-) strings anyway.

chr\$ () converts ASCII code into its character

chr\$ () is the reverse function of **asc\$ ()**, i.e. it converts an ASCII code into its string character.

Syntax **chr\$ (byte)**

byte is an integer number in the range of 0 to 255. All other values produce an

ILLEGAL QUANTITY ERROR.

Examples

```
?chr$(65)
a

r$=chr$(13)               :rem r$ is loaded with a carriage return

q$=chr$(34)               :rem q$ is loaded with a quotation mark
```

len () returns the length of a string

len () returns the length of a string, that means it returns the amount of (printable and not printable) characters in a string.

Syntax **len (string)**

string is a string constant, variable or expression. **len ()** returns values in the range of 0 to 65535. 0 is returned for an empty string.

left\$ () **mid\$ ()** **right\$ ()** cuts string into pieces

With the following instructions strings can be cut into pieces. It is possible to cut from the left end, from the right end or a piece in the middle.

Syntax	left\$ (string, count)	cuts from the left end
	right\$ (string, count)	cuts from the right end
	mid\$ (string, start, count)	cuts a piece in the middle

string is a string constant, variable or expression. *count* specifies the amount of elements to be cut from the corresponding end. **mid\$ ()** additionally requires the start position *start* from where the cutting should begin. If *number* is omitted the cut-string will continue to the right end of *string*.

Examples

```
print left$("abcdefghi",5)
abcde

print right$("abcdefghi",5)
efghi

print mid$("abcdefghi",2,3)
bcd

print mid$("abcdefghi",4)
defghi
```

instr()

returns the position of a sub-string

instr() is looking for a sequence of characters (**search string**) to be part of a **source string**. The **source string** is searched from left to right from the **starting position** on. If no start position is defined the source string is investigated from the left end. The position of the **search string** is returned. If the search string could not be found, 0 is returned. Even if a **starting position** is defined the returned position is to be counted from the leftmost end of the **source string**.

Syntax **instr**(*sourcestring*, *searchstring*, *startposition*)

sourcestring and *searchstring* are string constants, variables or expressions, *startposition* is a integer number.

Examples

```
print instr("abcdefg","cd")
3

print instr("abcdefg","h")
0

print instr("abc","abc")
1

print instr("abcdefabc","abc",4)
7
```

```
10 for i=65 to 90 : qstr$=qstr$+chr$(i) : next
20 get g$ : if g$="" goto 20 :rem this program returns the position
30 print instr(qstr$,g$) :rem of the typed letter in the alphabet
40 goto 20
```

Other Functions and System variables

rnd()

returns a random number

Random numbers are generated in *andiBASIC* with the `rnd()` function. It returns a random float number in the range of 0 to 1.

Syntax `rnd(ne)`

ne is a numerical constant, variable or expression which determines the algorithm generating the random number:

1. **ne > 0** : The algorithm returns the next value of a random sequence when called. This sequence is independent of the positive value of **ne**.
2. **ne < 0** : The returned value here is also derived from a random sequence. This sequence is however initialized when a negative argument is input, depending on its value.
3. **ne = 0** : The algorithm used here does not work with a fixed random sequence but rather makes use of a fast clock generator. This way, physical random numbers are transferred, apart from when the function is called several times within a continuously running program part.

rnd() only generates numbers between 0 and 1. With the following algorithm it is possible to generate random numbers in any required area of value:

RandomNumber = (end value - start value)***rnd(0)** + start value

Example

```
10 Number = int(5*rnd(0)+1)
20 print"Dice points : "Number
30 get g$: if g$="" then 30
40 goto 10
```

This program simulates a dice. It generates integer numbers from the range between 1 and 6.

dt\$
ti\$
ti

returns the current date, real time clock and timer

The IM system has got a battery buffered real time clock with date (calendar) and timer (stopwatch) functions. Date and time of day are saved even when the system is powered down. That means after switching on the computer the current date and clock time are present and can be called up.

dt\$ - Date

is set up of 6 digits in the format

ddmmyy (d=day, m=month, y=year)

The date does not compensate leap years. `dt$` can be read and set.

Syntax `printdt$` prints the current date on the screen
 `dt$="ddmmyy"` sets the date to September, 8th, 2002

ti\$ - Real Time Clock

returns 6 digits in the format

hhmmss (h=hours, m=minutes, s=seconds)

The hours are arranged in 24-hour standard.

Syntax	<code>printti\$</code>	returns the current Clock Time
	<code>ti\$="hhmmss"</code>	sets real time clock

ti - Timer

The timer function returns a floating point value counting the time from reset until the time of ti request. The time difference is represented in seconds. The float value has an resolution of 1 ms.

Syntax	<code>ti=0</code>	returns stopwatch time
	<code>printti</code>	sets back the stop time

The timer is able to trigger an interrupt. With its help multitasking programs can be set up easily.

Errtype
Errline

returns type and line number of an runtime error

The system variables **Errtype** and **Errline** return information about the type and program line of **andiBASIC** code errors. They can be requested after a program or a part of it was run. Also, if a program breaks, these information is of interest.

Errtype returns the error number,
Errline returns the program line number where the error has appeared.

This way, program parts can easily be scanned for errors. In the appendix there is a table of error numbers.

Example

```

10 Main program ::
100 on error goto Error-control
   :
1000 Error-control::
1010 print"Error Nr."Errtype" in line"Errline
1020 goto Main program
    
```

Program and Memory Organization

run
goto

cold-start of a program
warm-start of a program

rU
gO

A program resident in memory is executed by the direct-mode commands `run` and `goto`.

`run` performs a cold-start, i.e. previously assigned non-common variables (local and dim variables) as well as all pointers are cleared before execution. If the `run` command is followed by a line number the program is executed starting with this line number. If no line number is defined, execution starts with the first program line.

`goto` performs a warm-start, i.e. previously assigned variables and pointers are not cleared. The command must be followed by the line number the execution has to started.

Syntax `run line number` (*line number* is optional)
 `goto line number`

Example

```
run           :rem cold-start beginning with the first program line
run100       :rem cold-start beginning with program line 100
goto100      :rem warm-start beginning with program line 100
```

dstart

direct start of a program from disk

dsT

`dstart` loads a program from a storage device (floppy disk harddisk etc.) and runs it automatically. It has the same function as the combination `dload + run`.

Syntax `dstart"program name"` Loads a program from the device defined by the system variable **Floppy** and the drive defined by the system variable **Drive** and runs it automatically

`dstartdx,"program name" onuy` Loads a program from device number *y*, drive number *x* and runs it automatically

device number	device
0	IM floppy disk drive
2	IM harddisk drive
65	PC file system

The meaning of the drive number differs, depending on the device selected.

Device=0 (IM floppy)

drive #	drive
0	floppy drive 0
1	floppy drive 1
2	floppy drive 2
3	floppy drive 3

Device=2 (IM harddisk)

drive #	drive
0 ~ 7	subdirectories
8	main directory

Device=65 (PC harddisk)

drive #	drive
0	floppy A
1	floppy B
2	harddisk C
3	drive D etc.

Examples

```
dstartd2,"test1"onu65                    :rem loads the program test1 from the
c:\ partition of the PC harddisk
```

stop
cont
off

interrupt a program & single step mode continue after stop switch off single step mode

With the `stop` command as well as with the **BREAK** key **F12**, a program can be interrupted at any point. `stop` can be used in direct mode and in programs. This command will mainly be used during the testing of a program.

A `stop` command used in a program has two effects:

1. It interrupts the program at the point where the `stop` command was found.
2. It automatically activates the **single step mode**. In **single step mode** the program is executed step by step (statement by statement) by pressing the **ENTER** key. The **single step mode** is very convenient for program testing (debugging). The **single step mode** can be combined with the **trace mode**.

Please note that when hitting the **BREAK** key, the **single step mode** will not automatically be activated! To activate it in this case, input the `stop` command in direct mode.

Singles step mode and **trace mode** are switched off with `off` command.

A program interrupted by `stop` may be continued with the `cont` command. It will start at the current position, even if single steps were carried out. `cont` only is possible, if during **single step mode** no error message was displayed and no change in the program code was made. In this case the IM system displays "**CAN'T CONTINUE ERROR**". Then the program can only be restarted with `run` or `goto`.

Examples

```
10 rem
20 stop:A=10:B=20
30 end
```

```
run ENTER
```

```
break
20 stopIA=10:B=20
ready
```

```
- ENTER
```

```
20 stop:A=10IB=20
```

```
- ENTER
```

```
20 stop:A=10:B=20
```

```
- ENTER
```

```
30 end
```

I stands for the text cursor.

trace
off

comment basic statements switch off single step and trace mode

`trace` may be used alone or together with `stop`. It comments on the statement currently executed. In connection with the **single step mode** (`stop`, see above) the relevant program part is carried out step by step.

The **trace mode** can be finished with the `off` command. Like `stop`, trace can be used in both, direct and program mode.

Example

```
10 rem program part 1
20 trace
30 rem program part 2
40 a=3:b%=a*4:c$="abc":ifb%<10then40:elseifa=3then50
50 off:program part 3

run

transfer to 30
transfer to 40
a=3
b%=12
c$="abc"
false
true
transfer to 50
```

pause

pause program execution

paU

To delay a program at a particular point of the program for a defined period of time the **pause** command is used. The delay time is given in milliseconds as an argument.

Syntax `pausetime`

time is the delay time in milliseconds. The execution of the **pause** commands can be interrupted with the **BREAK** key.

list

list program code

ll

The **list** command lists the **andiBASIC** program code of the loaded program. If **list** is followed by line one or two numbers, the part of the program defined by the line numbers is listed. List may be used in both, direct and program modes.

With the **cmd** command, the listing can be deviated to another device than the monitor, e.g. a printer (see **cmd**).

Syntax

<code>list</code>	lists complete program code
<code>list start-end</code>	lists from line <i>start</i> to line <i>end</i> inclusive
<code>list start-</code>	lists from line <i>start</i> to the end of the program code
<code>list -end</code>	lists from the beginning to line <i>end</i>

The listed lines can be edited in direct-mode. To do this simply go with the cursor to the line to be edited, make your changes and press the **ENTER** key.

If a program has more than 30 lines, the listing is vertically scrolled through the Thales window.

The scrolling can be paused by pressing the keys **Ctrl** + **Q** shortly.

The scrolling is continued by pressing **any key**.

Pressing **Ctrl** + **Q** longer will cause a slow scrolling the time, the keys are pressed.

To interrupt the listing, press the **F12** key (**BREAK** key). The listing cannot be continued then.

vlist

list variables

vll

vlist outputs a list of one or more single variables. The variables are listed together with their current values. If it is used without an argument, all used variables are listed. If **vlist** is followed by a list of variable names, only these variables are listed. **vlist** may be used in both, direct and program mode.

Syntax

```
vlist
vlist v1,v2, ...
```

alist

list arrays

all

alist outputs a list of all used arrays. The arrays are listed together with the current values of each element. If it is used without an argument, all used arrays are listed. If **alist** is followed by a list of variable names, only these variables are listed. **alist** may be used in both, direct and program mode.

Syntax

```
alist
alist a1(),a2(), ...
```

search

Search for a character string

seA

search is a very useful tool for program editing. It scanning the **andiBASIC** text (statements, variable names, text strings, text remarks etc.) in the memory for a character string. This *search string* must follow the **search** command. Optionally, a text block for the scanning can be defined. If no block is defined, the complete program will be scanned.

Syntax

```
search, an-en, su$
```

an = beginning line
en = ending line
su\$ = searchstring

Examples

```
10 rem "text a$="
20 rem a$=
30 a$="abc"
40 rem END

search "a$=" :rem will list line 10
search a$= :rem will list lines 20 and 30
search,25-,a$ :rem will list line 30
search,-25,"a$=" :rem will list line 10
search,-25,a$= :rem no listing
```

delete

delete program lines

deL

With **delete** single Basic lines or program areas can be selectively cleared. **delete** is equivalent to the **clp** command.

Syntax

```
delete zn
delete an-en
```

zn = line number
an = beginning line
en = ending line

Examples

```
delete 100           :rem clears line 100
delete 50-80        :rem clears lines 50 to 80 incl.
delete -100         :rem clears lines 0 to 100 incl.
delete 100-         :rem clears lines 100 to the end
delete              :rem clears the whole program
delete 10-20,50-60  :rem clears lines 10 to 20, 50 to 60
```

new initialize program RAM

The **new** command clears the program in memory and initializes all variables and pointers. If you use **new** in a program code the program erases itself as soon as it runs on the **new** command.

Syntax **new**

clr clear variables

clr clears all variable types (local, dim, com). It can be used in both, direct and program modes. As an argument, a list of variables and arrays can follow. Without an argument, all variables will be cleared.

Syntax **clr**
clr *v1, v2, v3, . . . , vn*
v1, . . . , vn = variable and array list

Examples

```
clr a, b%, c$, d()  :rem clears the variables a, b%, c$ and the array d()
clr                :rem clears all variables used
```

clp clear program code lines

A line or a number of subsequent lines can be cleared using the **clp** statement. If the statement is used without a line number the complete program is cleared. With one line number, only this line is cleared. Defining two line numbers, all line numbers in between these numbers (including the indicated ones) are cleared.

Syntax **clp** clears entire program
clp, *linenumber* clears line number *linenumber*
clp, *start-end* clears line numbers *start* to *end*

You also can clear a program code line by typing its line number in an empty line and confirm with the **ENTER** key.

cls clear stack

cls initializes the stack pointer. All data on stack are erased with the effect that all subroutine-, **for...next**- and **fn** structures are cleared. This command is used mainly for program testing.

Syntax **cls**

cpy

copy program lines

cpy copies program lines to a location of the program defined in the argument. Here, same as with **move**, a **renumber** must follow. Other than with **move**, **cpy** does not erase the source line. If a **goto** destination line is moved, the argument of all **goto** commands which refer to this line are changed automatically throughout the entire program by the **renumber** command.

Syntax **cpy** *sourcelinez,destinationline*
 cpy *startline-endline,destinationline*

renumber

renumber program lines

renU

renumber changes the program line numbers and jump instructions (**goto**, **gosub**) in a program. Up to three arguments can be defined the: start line, the destination line and the intervals the auto-numbering should work with numbers. The default interval is 10. The destination line number must be higher than the start line number. If the first and/or the second parameter are not used, they can be replaced by a comma.

Syntax **renumber** *startline,destinationline,interval*

Examples

```
renumber           :rem entire program, interval 10
renumber1,200,5    :rem from line 1 to line 200, interval 5
renumber1,,1       :rem from line 1 to last line, interval 1
renumber,,1        :rem entire program, interval 1
renumber5          :rem from line 5 to last line, interval 10

100 goto 200       :rem 1. line
200 goto 100       :rem 2. line
renumber100,500,50 ENTER

list              ENTER

500 goto 550 : rem 1. line
550 goto 500 : rem 2. line
```

swap

swap contents of variables

swap exchanges the contents of two variables. This saves an additional variable and calculating time.

Syntax **swap** *variable1,variable2*

Example

```
A=10:B=20:swapA,B: ?A,B ENTER
20      10
```

fre(0)

free memory

The function **fre(0)** returns the amount of free memory space (in byte).

Syntax **fre(0)**

Text Handling

print

print characters to the screen

?

This statement prints text on the screen. The text may be defined by any sort of constants or variables (string or numerical) or by the result of an operation or function. The abbreviation of the `print` command is the question mark “?”.

Syntax `print expression`
 `? expression`

expression can be a numeral or a string. String constants must be enclosed in quotation marks.

Examples

```
print "stringconstant" ENTER
stringconstant
ready.
```

```
? 12 * 2 ENTER
24
ready.
```

```
A$="stringvariable" ENTER
print A$ ENTER
stringvariable
ready.
```

```
A=10 : B=5 ENTER
? A * B ENTER
50
ready.
```

The `print` statement accepts more than one argument. Valid separators are the semicolon (;) and the comma (,). Use the semi-colon if all arguments should be printed one after the other (without a separating space). Use the comma, if there should be 10 spaces between each argument. This way a simple formatting of the output is achieved.

```
? 1;2;3;4 ENTER
1234
ready.
```

```
? 1,2,3,4 ENTER
1    2        3        4
ready.
```

```
A$="my name is":B$="Willy" ENTER
? A$,B$ ENTER
my name is            Willy
ready.
```

Pressing the **HOME** key will position the cursor in the uppermost left corner of the text window (in case of a text window is defined). Pressing **HOME** a second time will open an existing text window an position the cursor in the uppermost left corner of the **Thales** window.

Pressing the **END** key once will clear the text from the screen, pressing it a second time will

clear the graphics too.

These functions can also be used in a program. You have to use the `print` statement and input these keys in between quotation marks. The function is not executed immediately, but when the `print` statement is executed.

`tab()` **tabulator**

With printing the `tab()` command you can move the text cursor (current writing position) in horizontal and vertical direction. As an argument you can input the horizontal and the vertical steps to go relative to the current writing position. The coordinates 0,0 refer to the upper left hand corner of the Thales window. The first argument defines the horizontal, the second the vertical steps. `tab()` can only be used together with `print`.

Syntax `print tab(horizontal,vertical)`

`spc()` **blanks**

`spc()` creates a certain amount of blanks. `spc()` can only be used together with `print`.

Syntax `print spc(number)`

`pos()` **cursor position**

By this function the current position of the text cursor is returned. `pos()` can be used in 3 different ways:

Syntax	<code>pos(0)</code>	returns the current cursor column
	<code>pos(1)</code>	returns the current cursor line
	<code>pos(-1)</code>	returns column and line

For all three cases, only one numeral is returned. For `pos(-1)` the following format is used:
line * 256 + column

Examples

```

printtab(10,20);pos(0)      ENTER
:
10
ready.

printtab(10,20);pos(-1)    ENTER
:
5130                       :rem 5130=20*256+10
ready.

```

`using` **format text output**

`using` allows to print text according to a mask. The `using` statement is followed by a string (constant or variable) which defines the mask. Then, separated by commas, the data to format must follow. These can be both, strings and numerals. The `using` command makes it comfortable to create tables etc. `using` has to be used together with `print`.

Syntax `printusing mask$,a1,a2,a3,...`

a1, a2, a3 = numeric or string data (constants or variables)

The mask string `mask$` defines the format the data should be printed. The following characters are used as substitutes, defining a mask:

#	fills the mask from the left	num + string
@	fills the mask from the right; fills numerals with leading or final zeros	num + string
+	enforces sign (+ and -)	num
-	enforces sign only for negative values	num
^	enforces exponential format for numerals	num
.	decimal point	num + string

The mask string may contain more than one mask division. A mask division ends where the substitute character changes to another substitute character or a non-substitute character. All data following the mask string and separated by commas, will be filled subsequently into the mask divisions. If there is more data than mask divisions, the rest of the data will be ignored. If there are more mask divisions than data, the mask characters are printed instead. A semi-colon in the data-list ends the `using` and prints the following data without using a mask.

Examples

```

printusing"####.#### KM",25.3 ENTER
25.3 KM
ready.

printusing"+####.@@@ KM",25.3 ENTER
+25.3000 KM
ready.

printusing"-@@@.#### KM",25.3 ENTER
0025.3 KM
ready.

printusing"#####@@", "ABC", "DEF" ENTER
ABC DEF
ready.

printusing"#.##",1.555,2,2.5 ENTER
1.56
ready.

A&=cplx(2,3.4):printusing"#.## #.##",A& ENTER
2.00 3.40
ready.

printusing"#.@@ DM -#.## %",5.2 ENTER
5.20 DM -#.## %
ready.

printusing"#.## DM",1.555;" net" ENTER
1.56 DM net
ready.

printusing"#.@@ DM =",1.3;:printusing" #.# %",5.5 ENTER
1.30 DM = 5.5 %
ready.

printusing"-#.###^^^^ joule in ###",15.3,"kWh ENTER
1.53E+01 joule in kWh
ready.

printusing"#.##^^^^ #.##^^^^ #.##^^^^",1e9,1e9,1e9 ENTER
1.00e+9 1.00+09 1.00+009
ready.

```

Numerical masks can be filled with strings (and vice versa), however, the special numerical substitute characters (+/-/.^) are printed directly and function as separators between two mask divisions.

pack

create formatted text

pack assigns the value of a filled mask to a numerical or string variable. If the destination variable is of numerical type, the data is rounded, if needed.

Syntax **pack** *var*, **using** *ma\$*, *a1*, *a2*, ...

var = destination variable (string or numerical)

ma\$ = masks

a1, *a2* = arguments

Examples

```

Zip=8640:Town$="Kronach"
packA$,using"@@@  #####",Zip,Town$
printA$
8640 Kronach
ready.

A=31415692:packB,using"#.###^ ^ ^",A
print A
3.142e+07
ready.

```

ENTER
ENTER

ENTER
ENTER

Jumps & Branches

Labels and line numbers

Computer programs normally are executed linearly, this means, in their physical order given by the line numbers. Anyway, there are commands, which allow to break this straight-forward principle: the jump commands.

As a destination a **line number** or a **label** is accepted. A label is set by a label name followed by two colons (::). A label name is following the same conventions as a variable name. No reserved **andiBASIC** keywords are allowed. Remember that capitals and lower case characters will lead to different label names (e.g. `label::` and `Label::`).

To call a label only the label name (without the colons) has to be used. Labels may be used at the beginning as well as in the course of a program line. A label at the beginning of a line does not substitute a line number.

Syntax `ln LABEL::` `ln = line number`

Example

```

10   rem
20   rem
30  gosub SUBROUTINE 1
40   rem
50  goto PROGRAM END
60   rem
70  SUBROUTINE_1::
80   rem
90  return
100  rem
110 PROGRAM_END::clr:end
    
```

In line 30 the program jumps to label `SUBROUTINE_1` (line 70). The subroutine will be executed and after reaching the `return` command in line 90 the program execution jumps back to line 40. Further examples can be found in the following chapters dealing with jump instructions.

Jump Commands

andiBASIC is able to interpret two types of jump commands:

Obligatory jumps	are carried out in any case	goto
Conditional jumps	are only carried out if a condition is met	gosub, fn...

Another differentiation can be made between jump commands that jump to another program line, where the program is continued then linearly (**goto**), and jump commands that jump to a subroutine which is inserted in the logical program execution (**gosub**). If you jump to a subroutine, the system remembers the statement where you jumped from and returns there as soon as a **return** command is reached.

goto obligatory jump **go**

An obligatory jump is executed by the **goto** command. The program is continued from the destination location. Line numbers as well as labels can be used to define jump destination.

Syntax **goto** *ln* *ln* = line number
 goto *LABEL* *LABEL* = label name

Examples

```

10 print "Program start"
20 goto 50
30 stop
50 print "Control transfer destination 1"
60 goto 30

run
Program start
Control transfer destination 1
break 30 stop !

```

```

10 print "Program start"
20 goto Destination1
30 finish:: stop
50 Destination1:: print "Control transfer destination 1"
60 goto finish

run
Program start
Control transfer destination 1
break 30 stop !

```

gosub/return obligatory jump / return to/from subroutine **goS / reT**

gosub uses exactly the same syntax than **goto**. A line number or a label defines the destination. The program is continued from the at the destination. The difference between **gosub** and **goto** is that the program remembers the starting point and returns to this point as soon as it runs on a **return** command. The program part between the **gosub** jump destination and the corresponding **return** command is called **subroutine**. Subroutines may be accessed from different points of the main program. They use global variables.

Syntax **gosub** *ln* *ln* = line number
 gosub *LABEL* *LABEL* = label name
 return

Examples

```

10 print "Main program"
20 gosub 100
30 gosub subroutine2

```

```
40 end
100 print "Subroutine 1"
110 return
200 subroutine2::
210 print "Subroutine 2"
220 return
```

andiBASIC can handle up to 256 subroutine levels. This means that you can call subroutines from a subroutine which was called from a subroutines etc.

```
10 print "Main program"
20 gosub 100
30 end
100 print "Subroutine level 1"
110 gosub Subroutine2
120 return
200 Subroutine2:: print "Subroutine level 2"
210 gosub Subroutine3
220 return
300 Subroutine3:: print "Subroutine level 3"
310 return
```

Local functions

A function is a special type of subroutine which may use global and local variables. It is defined by the statements `def fn`. In multiple-line functions the end of the function code is defined by the statement `fnend`. If a result has to be returned the statement `fnreturn` may be used optionally.

A function is called by the `fn` statement followed by the function name and one or more transfer data. The function name follows the same rules as a variable name. When called, a function returns the result of its mathematical or string expression in the same way a standard function such as `sin()`, `asc()` or `int()` does. The type of the return variable must fit the format of the returned value. In the same way, you must add the extension of this data type to the function name, e.g. for a function returning a string the function name must end with a "\$": `fnSTRINGCHAIN$(...)`.

The data to be transferred to the function may have all kinds of formats (integer, real, complex, string etc.), but on the other end, the types of the list of local variables, defined with the function definition (`def fn`), must correspond to the format of the transfer data: The first data will be transferred to the first variable, the second data in the second variable etc. So, take care of the correct order.

`fn`
`def fn`

one-line functions

The **one-line type** allows the definition of a mathematical operation. This has to be written after the function header (`def fn` + function name + list of local variables) separated by an equal character. One-line functions are limited in length to one *andiBASIC* program line (max. two screen lines). A function has access to its local variables as well as to all global variables. At least one local variable must be defined. If a function is called with less arguments than defined in the function, the ungiven variables could be use as local help-variables.

Syntax

```
def fn Funa (v1, v2, v3, ...) = ma
var = fn Funa (t1, t2, t3, ...)
```

Definition of a one-line function

Call of a (one-line) function

Funa = function name
v1...v3 = local variables
ma = mathematical expression
t1...t3 = transfer data
var = variable for return value

Examples

```
10 a=100 : b=200 : c=300           :rem global variables a, b, c
30 d = fnMULTIPLY(10,20,30) : print d
40 end
80 deffnMULTIPLY(a,b,c) = a*b*c   :rem local variables a, b, c

run                               ENTER
6000

ready.
```

In the above example, the value of 10 is transferred to the local variable a, the value of 20 to the local variable b and the value of 30 to the local variable c. The global variables a, b and c, defined in the main program, are kept, because the function uses local variables.

If string data is returned, the function name must end with a \$:

```
30 print fnDATE$(dt$)
40 end
80 deffnDATE$(b$)=left$(b$,2)+"."+mid$(b$,3,2)+"."+right$(b$,2)

run
07.08.2001

ready.
```

Besides the local variables all global variables may be used in a function:

```
10 x=100
20 print fnRandom(10,15),x,a,b
30 end
80 deffnRandom(a,b) = a*b + x

run
250 0 0

ready.
```

The contents of the (global) variables a and b is 0. The local variables a and b can not be printed in the global part of the program.

fn
def fn...fnreturn...fnend

multiple-line functions

Other than with the one-line functions you can use more than one **andiBASIC** code line with multiple-line functions. The function code starts with the **def fn** statement and ends with the **fnend** statement. The body of the function is located between these two statements. Local data may be transferred to the main program by using the **fnreturn** statement. The **def fn** statement is not followed by an equal character. All other conventions are same as for the one-line functions.

As with subroutines, you can call functions from a function. There are 256 function levels possible.

Syntax

```
def fn Funa (v1, v2, v3, ...)
[fnreturn retvar]
fnend
RetVar = fn Funa (t1, t2, t3, ...)
```

Definition of a one-line function
Returning local data
End of a function
Call of a (one-line) function
Funa = function name
v1...v3 = local variables
retvar = local return variable
t1...t3 = transfer data
RetVar = global return variable

Examples

```
10 deffn Calculation1 (a,b,c)
20 c = (a + b) * k
30 fnreturn c
40 print a,b
50 fnend

100 main_program::
110 input "x,y";x,y
120 k = 20
130 c = fn Calculation1 (x,y)
140 print c
150 goto main_program
```

An example for a two-level function routine:

```

10 deffn FUNCTION 2 (a,b)
20 b = a + 100
30 fnreturn b
40 fnend
100 deffn FUNCTION 1 (a,b,c)
110 c = a * b
130 d = fn FUNCTION 2 (c)
140 print c,d
150 e = a + d
160 fnreturn e
170 fnend
200 Main_Program::
210 input"a,b";a,b
220 print fn FUNCTION 1 (a,b)
240 goto Main_Program
    
```

With the **fnreturn** statement a value is returned to the calling routine:

```

10 a& = fn c&(10,20) : ?a&
20 a& = fn c&(10,20,1) : ?a&
100 deffn c&(a,b,c)
120 if c=0 then fnreturn cmplx(a,b)
130 else fnreturn 1/cmplx(a,b)
140 fnend
    
```

```

50 deffn Rounding%(a)=int(a+0.5)
60 deffn A$(a,b$)=b$+chr$(a)
    
```


if...trueif...falseif...endif conditional branch (extended version)

While with the `if...then...else` structure only one line of `then` and `else` statements are possible, the `if...trueif...falseif...endif` structure allow multiple-line procedures for all conditions.

After `trueif` and `falseif` may follow any number of program lines. The `trueif` block is ended by the `falseif` statement. If there is no `falseif` statement used, `endif` is ending the block. The `falseif` block is ended by `endif`.

Syntax

```

if condition
trueif
:   true block
[falseif
:   false block]
endif

```

Example

```

10 input"Number";a
20 if a=0 trueif
30 ?"Type in a number not equal 0"
40 goto10
50 falseif
60 c=1/a
70 printa
80 endif

```

SCRIPT Example

```

SCRIPT1
  input"Number";a
  if a < 100 trueif
    print"Number less than 100"
  falseif
    print"Number greater or equal 100"
  endif
  print"End of comparison"
SCRIPT_END

```

**on...goto
on...gosub**

branch on branch table

A byte expression defines, to which destination the program branches. The possible destination line numbers and/or labels are listed after the `goto/gosub` statement, separated by commas.

Syntax

```

on be goto de1,de2,de3,...
on be gosub de1,de2,de3,...

```

be = byte expression
de1...de3 = destinations

be (byte expression) may be a number, a constant or a variable (integer or real) in the range between 1 and the number of jump destinations. If *be* has the value of 1, the program branches to the first destination, if *be* is 2, it branches to the second etc. If *be* has a higher value than there are jump destinations, the statement following the `on...goto` is executed.

`on...gosub` is used in the same way. The only difference is, that the program branches to a subroutine. When the program returns from the subroutine, the program is continued with the statement following the `on...gosub` structure.

Examples

```

10 input"Jump destination";a
20 on a goto 100,200,300
30 print a" is no valid destination !" : goto 10
100 print"Destination 1" : goto 10
200 print"Destination 2" : goto 10
300 print"Destination 3" : goto 10
    
```

```

10 input"Destination";a
20 on a gosub Destination1,Destination2,Destination3 : goto 10
100 Destination1::
110 print"Destination 1" : return
200 Destination2::
210 print"Destination 2" : return
300 Destination3::
310 print"Destination 3" : return
    
```

The jump destinations must be listed in one **andiBASIC** line (max. linked double line) separated by commas. If more jump destinations are needed, the branch destinations may be divided to more than one **on...goto/gosub** structure.

Example

```

10 input"Destination";a
20 on a goto Destination1,Destination2,Destination3,Destination4
30 on a-4 goto Destination5,Destination6,destination7
40 on a-7 goto Destination8,Destination9
etc.
    
```

SCRIPT Example

```

SCRIPT1
  input"Number";a
  if a < 100 trueif
    ak100%=1
  falseif
    ak100%=2
  endif
  on ak100% goto SC DO 1,SC DO 2
SC1 CONT::
  print"End of comparison"
  goto SC1 END
SC DO 1::
  MESSAGE(0)
  MESSAGE$(fnSTR$(a))
  MESSAGE$("less than 100")
  MESSAGE(2000,2,2)
  goto SC1 CONT
SC DO 2::
  MESSAGE(0)
  MESSAGE$(fnSTR$(a))
  MESSAGE$("greater or equal 100")
  MESSAGE(2000,2,2)
  goto SC1_CONT
SC1 END::
SCRIPT END
    
```

on error...goto
on error...gosub **branch on error number**

When a program error occurs, the operating system returns an error number. This error number can be used to make the program respond to the error. If during program execution an error occurs, the system scans the program for the next `onerror` statement ahead and branches to the destination defined there. Knowing this fact, you can provide different parts of the program from breaking with different branch destinations.

The `onerror...goto/gosub...` structure is used in a different way than the `on...goto/gosub` structure: There is only one jump destination listed. Furthermore the error number can be specified on which the branch is to be executed. All other error numbers are ignored. If no error number is specified, the branch is executed on every error number.

Syntax `on error [en] goto de`
 `on error [en] gosub de`
de = destination ; *en* = error number

In addition there are two system variables which contain the error number and the number of the program line in which the error occurred.

Errtype contains the current error number
Errline contains the *andiBASIC* line number, in which the error appeared

By means of these variables, program parts can be created which are handling errors individually.

Example

```
10 Main program::
   :
   :
100 on error goto ErrorControl
   :
   :
1000 Error-control::
1010 on Errtype goto 1100,1200,1300,...
1100 print"Error Nr. ";Errtype;" in Line";Errline
1110 goto Main program
   :
   :
```

on irq...goto
on irq...gosub
setirq **branch on interrupt**

The 680x0-Processor is able to handle 7 *autovector interrupt levels*. An *andiBASIC* program can detect these interrupts and respond to them. To do so, first the interrupt level has to be chosen on which the program should respond. This is done by the `setirq` statement. As soon as the specified interrupt occurs, the program looks for the next `onirq...goto/gosub` statement and branches to the specified destination (line number or label). With this structure interrupt driven routines can be created. This is a very simple form of multitasking.

Syntax `setirq = il` *il* = interrupt level
 `onirq il goto dn` *dn* = destination
 `onirq il gosub dn`

Example

```
10 Main program::
20 ti=0
30 setirq=0
   :
   :
90 on irq gosub Clock Display
   :
   :
1000 Clock Display::
1010 print"<End>"ti$      :rem <End> means the <End>-key
1020 ti=0
1030 return
```

The interrupt driven subroutine Clock_Display displays the time in the upper left corner while the main program is running. If a data acquisition is running in the main program all interrupts should be turned off (masked).

on nmi goto
on nmi gosub

branch on nmi

The 680x0 processor also handles an NMI (non maskable interrupt). An NMI can be produced by hardware devices and the NMI switch on the OS040 card of the IM6/6e. When a NMI occurs, the **onnmigoto/gosub** structure branches to the specified destination of the program, so that e.g. the pressing of the NMI switch can be handled.

Syntax **onnmigoto dn** dn = destination (line number or label)

If a program cannot be interrupted by means of normal keyboard control (e.g., when running a transient recording, or during an I/O-operation) it may be helpful to have the possibility to break the operation without breaking the program. Pressing the NMI switch once will break a system routine (e.g. transient recording), pressing it twice will even break a handshake operation. This causes a *double nmi error* (**Errtype=58**). It can be trapped by an **on error** statement, as shown in the following example.

Example

```
100 on error 58 goto NMI ROUTINE
110 on NMI goto NMI ROUTINE
   :
200 NMI_ROUTINE:: cls:goto MENU
```

The example traps both normal and double NMI.

end

logical program end

A BASIC program has a physical and a logical end. The physical end is the end of the program code (last program line). The logical end is the point at which the program execution is ended and must not be same as the physical end. For example, if subroutines are placed after the main program body, the logical end is at the end of the main program. In such case, the logical end must be marked with the **end** statement. When **end** is reached program execution is finished and the system jumps to the direct mode. The program is still resident in memory, but you cannot resume the program. Anyway, it can be started again with **run**.

Syntax **end**

end can be omitted if the physical and the logical end are the same.

stop
cont**stopping / continuing program execution**

stop breaks a program, and jumps to the **ANDibasic direct mode** and activates the *single-step mode*. The system displays **break.** and the line where the program was interrupted. In *single-step mode* you can execute single instruction subsequently by pressing the **ENTER** key. All variables and pointers are saved so that you can resume the program from the point it stopped using the **cont** statement. This is possible only if the program code was not changed in the meanwhile.

Syntax **stop** (to be used in both program and direct mode)
 cont (to be used in direct mode)

rem**remark**

All characters following the **rem** statement in a basic line are ignored for program execution. The system continues execution in the next program line. With **rem** you can set comments in the basic text. Furthermore you can exclude **andiBASIC** code from execution temporarily.

Syntax **rem text**

Loops

Program loops are used to execute the same instructions a defined number of times or until a condition is reached. If the defined number of loops are executed or if the condition is reached, the program jumps to the instruction following the loop structure.

**for...to...step
next**

count loop

Count loops use a counting variable to count from the start value to the end value in the defined step width. They are used when a defined number of loop counts is needed or if the loop body is working with arrays.

The **for...next** loop automatically handles the counting variable, i.e. when entering a **for...next** loop it is loaded with the start value. With each loop turn it is automatically increased by the defined step width. If the step width is negative, the counting variable will be decreased. As a counting variable an integer or a real variable can be used. As step width positive and negative integer and real values are allowed. If the step width is omitted, it is automatically set to the value of +1.

A count loop can be divided into the loop head, the loop body and the loop end. The loop head is represented by the **for...to...step** structure and defines the counting variable, the start and the end value and the step width.

The loop end is defined by the **next** statement. Here the system checks whether the value of the counting variable has reached the end value. If this is true, the loop is left. If this is not true the system jumps back to the first instruction of the loop body. The loop body is the program code lying physically in between loop head and loop end.

Syntax

for <i>cv</i> = <i>sv</i> to <i>ev</i> [step <i>sm</i>]	<i>cv</i> = counting variable
:	(<i>loop body</i>)
next [<i>cv</i>]	<i>sv</i> = start value
	<i>ev</i> = end value
	<i>sm</i> = step width

Naming the counting variable after **next** can be omitted.

Example

```
10 for i = 1 to 10 step 2
20 print i,
30 next

run
1           3           5           7           9
ready.
```

The counting variable *i* is loaded with the start value of 1 in line 10. Then the loop body (line 20) is executed. The next statement checks if the counting variable has reached the end value. If not, it increases the counting variable by 2 (step width), if yes, the program is finished.

Example

```
10 for i = 10 to 1 step -2
20 print i,
30 next

run
10          8           6           4           2
ready.
```

In this example, the step value is negative. Therefore the start value must be higher than the end value.

You may create up to 256 loop levels. Each `next` statements are automatically associated with the `for...to` structure of the same level independent of the counting variable name stated after `next`. The loop levels will be passed from the innermost to the outermost. Of course the counting variables must be individual for each loop level.

Example

```
10 dim a(3,6)
20 for i = 0 to 3                               :rem outer loop header
30 for j = 0 to 6                               :rem inner loop header
40 a(i,j) = i * j                             :rem loop body
50 next                                        :rem inner loop end (j)
60 next                                        :rem outer loop end (i)
100 for i = 0 to 3 : for j = 0 to 6
110 print a(i,j),
120 next : print : next
```

SCRIPT Example

```
SCRIPT1
  dim a(3,6)
  for i = 0 to 3                               :rem outer loop header
    for j = 0 to 6                             :rem inner loop header
      a(i,j) = i * j                          :rem loop body
    next                                       :rem inner loop end (j)
  next                                        :rem outer loop end (i)
  for i = 0 to 3
    for j = 0 to 6
      print a(i,j),
    next
  print
next
SCRIPT_END
```

while...wend

conditional loop

The `while...wend` loop does not monotonically increment or decrement a counting variable, but checks if a condition is met. The loop head consists of the `while` statement followed by the condition (comparison). The loop end is marked by the `wend` statement. The loop is repeated as long as the condition is true. If the condition is false the program is continued with the statement following `wend`. The `while...wend` structure does not automatically handle a counting variable. The condition has to be changed by the loop body.

Syntax

```
while condition
:                               (loop body)
wend
```

Examples

```
10 Number1 = 10000 : Number2 = 3
20 while Number2 < Number1
30 Number2 = Number2 * Number2
40 wend
50 print Number2
```

You may create up to 256 levels of `while...wend` loops.

SCRIPT Example

```

SCRIPT1
  Number1 = 10000 : Number2 = 3
  while Number2 < Number1
    Number2 = Number2 * Number2
  wend
  print Number2
SCRIPT_END  Number1 = 10000 : Number2 = 3

```

repeat...until conditional loop

Whereas a **while...wend** loop repeats the loop body as long as the condition is true, the **repeat...until** loop repeats the loop body as long as a condition is not true. Other than with the **while...wend** loop, here the condition is checked at the end of the loop. The loop head is represented by the **repeat** statement, the loop end by the **until** statement followed by the condition (comparison). Apart from this, the **repeat...until** loop follows the same rules than the **while...wend** loop.

Syntax **repeat**
 :
 until *condition* (*loop body*)

Example

```

10 Number1 = 10000 : Number2 = 3
20 repeat
30 Number2 = Number2 * Number2
40 until Number2 > Number1
50 print Number2

```

You may create up to 256 levels of **repeat...until** loops.

SCRIPT Example

```

SCRIPT1
  Number1 = 10000 : Number2 = 3
  repeat
    Number2 = Number2 * Number2
  until Number2 > Number1
  print Number2
SCRIPT_END

```

Graphics

The Thales window has a graphic resolution of 640 x 512 pixel. In Text mode (Basic input mode) 32 lines and 80 columns can be displayed. The font and font size is fixed. Text and graphics are completely independent. Text mode is only used in the Basic input mode. In programs you should use Graphics-mode text, instead. Here you have several fonts available and font size is scaleable.

The virtual screen in Graphics-mode addresses +32767 to -32768 pixels in both, x- and y-direction. The real screen, showing 640 x 512 pixels, can be positioned somewhere in the virtual screen.

The set of graphics commands includes special macros for displaying scientific data such as curves, bar graphs etc. The plots are handled vector-oriented, not pixel-oriented. This grants smaller file sizes and a far more comfortable editing features.

Nearly all graphics commands have a similar syntax: The leading command is *plot* followed by the specification of the command. The specification often is a two-letter-code written in quotation marks ("). After that, parameters may follow, separated by a comma (.). Example: `plot"da",10,20`. Some special graphics commands – e.g. macro-commands – show a slightly different syntax, having the specification directly attached to the basic command `plot`. Example: `plotmata%(),10,5`.

`plot"sy"`

Sets the plot parameters for Graphics-mode text

This command sets the parameters for text written with the `plot"da"` and `plot"dr"` commands. You may define the font, character size and orientation of the text. character size can be defined with `plot"sy"` (sy= set symbols).

Syntax `plot"sy",difo,cs`

difo = *di* + *fo* = writing direction and font type
cs = character size

The writing direction (*di*) is defined as follows:

<i>di</i> value	direction
0	left to right
1	down to up
2	right to left
3	up to down

An IM font type (*fo*) is defined as follows:

<i>fo</i> value	font type
0	CG Times
4	Univers
8	Courier
12	Symbols
16	Helvetica
20	Times Roman

A Windows font type (*wifo*) is defined as follows:

<i>wifo</i> value	font type
0	Times New Roman
4	Univers
8	Courier
12	
16	Arial
20	Times Roman

The *difo* value is calculated as follows:

$$difo = di \text{ and } (4096 + fo) \text{ and } (128 + wifo) \text{ and } (32 * ASCII)$$

For the values for *fo* and *wifo* refer to the tables above. To select ASCII character set, ASCII = 1, for German character set, ASCII = 0.

The `plot"sy"` command is mainly used for naming the axis of a graph.

cs sets the character size. Values from 0 to 31 are allowed. *cs* = 0 sets the default character size. Values greater than 31 cause further enlargement of the lettering. Negative values between -31 to -1 produce the same character size as 1 to 31.

Examples

```
1 rem plots the word "text" in different sizes
10 for i=0 to 31
20 plot"sy",0,i
30 plot"da",10,100,"text"
40 next
```

```
1 rem plots the word "text" in a vertical direction (1)
2 rem using the Arial font of the PC (4096+128+16) and the ASCII
   character set (32)
3 rem in font size 3
20 plot"sy",(1 and (4096 + 128 + 16) and 32),3
```

`plot"sk"`

Sets the relative origin of coordinates

The absolute origin ($x_{a0} = 0$, $y_{a0} = 0$) of coordination (AC) lies in the bottom left corner of the screen. It cannot be moved. The relative origin of coordinates (RC) is set relative to that. If no `plot"sk"` is used, the RC is identical with the AC. All plot commands using coordinates refer to the RC, not to the AC.

Syntax

`plot"sk", x_{r0} , y_{r0}`

X_{r0} = x coordinate (pixel)

Y_{r0} = y coordinate (pixel)

`plot"da"/"dr"`

Sets the current plot position

This command sets the current plot position to the specified coordinates. The `plot"da"` command (da = direct absolute) sets the position relative to the RC set with the `plot"sk"` command. The `plot"dr"` command (dr = direct relative) sets the position relative to the previous plot position.

With a string expression following the coordinates you can plot a graphics text beginning at the current position. Font type and font size are defined with the `plot"sy"` command.

Syntax

`plot"da", x , y , $t\$$`

`plot"dr", x , y , $t\$$`

x = x coordinate

y = y coordinate

$t\$$ = string expression

plot"va"/"vr" Plots straight lines (vectors)

Using one of these instructions, a straight line is plotted from the current plot position to the coordinates defined in the `plot"va"/"vr"` command. With the `plot"va"` command (*va* = vector absolute) you define the coordinates of the end point relative to the RC (defined with the `plot"sk"` command). With the `plot"vr"` command (*vr* = vector relative) you define the coordinates relative to the current plot position. After the execution of the command the end point of the line is set as the new current plot position so that you are able to directly append a new line beginning at this coordinates. The third parameter of the command defines the plot mode, thickness, and type of the line.

Syntax

```
plot"va",x,y,l
plot"vr",x,y,l
```

x = x coordinate
y = y coordinate
l = line type

The coordinates *x* and *y* are set as pixel coordinates. In the case of `plot"va"` they are relative to RC, in the case of `plot"vr"` they are relative to the current plot position.

l is an 8-bit-integer value. The meaning of the bits are as follows:

l = `mmnnntttbin`

mm = plot mode

mm	operation	line
00	AND	black
01	only <code>nnnttt_{bin}</code> is used	
10	OR	normal
11	XOR	reverse

The plot mode is kept until a new plot mode is defined. The default is `10bin = 128d` = normal plot.

nnn = thickness of the line

Values of 0 – 7 result in lines of 1 to 15 pixel thickness

ttt = line type

ttt	line type
0	solid
1	narrow dotted
2	dashed
3	dash-dotted
4	wide dotted
5	solid
6	solid
7	solid

l can only be used with `plot"va"/"vr"` instructions, but not `plot"da"/"dr"` instructions. That's why a graphic text (`plot"da",5,5,"text"`) cannot be deleted by `plot"vr",0,0,127`.

Example

```
1 rem the following instructions plot an x- and y-axis with scaling
  and naming
10 plot"sy",0,2 :rem selects CG Times and font size 2
20 plot"sk",320,255 :rem sets the RC to xr0=320 and yr0=255
30 plot"da",-300,0 :rem sets the current plot position to x=-300/y=0
40 plot"vr",600,0 :rem plots a horizontal line of 600 pixel length
50 plot"da",0,-250 :rem sets the current plot position to x=0/y=-250
```

```

60 plot"vr" 0 500 .rem plots a vertical line of 500 pixel length
70 for i=-300 to 300 step30 :rem loop start
80 plot"da",i,0 : plot"vr",0,-4 :rem plots x-scale marks
90 next :rem loop end
100 for i=-250 to 250 step25 :rem loop start
110 plot"da",0,i : plot"vr",-4,0 :rem plots y-scale marks
120 next :rem loop end
130 plot"da",200,-20,"x Axis":rem plots the naming of the x-axis
140 plot"sy",1,2 :rem sets vertical text plot direction
150 plot"da",-10,150,"y Axis":rem plots the naming of the y-axis

```

plot"ea"/"er" Plots rectangle Edges

Using one of these instructions, a rectangle edge is plotted from the current plot position to the coordinates defined in the `plot"ea"/"er"` command. With the `plot"ea"` command (*ea* = edge absolute) you define the coordinates of the end point relative to the RC (defined with the `plot"sk"` command). With the `plot"er"` command (*er* = edge relative) you define the coordinates relative to the current plot position. After the execution of the command the end point of the edge is set as the new current plot position. A screen window created by `plotdim` or the `window` command will limit the plotted edge.

The line type of the edge is preselected by the last vector-plot command (`plot"va"/"vr"`).

Syntax

```

plot"ea",x,y
plot"er",x,y

x = x coordinate
y = y coordinate

```

The coordinates *x* and *y* are set as pixel coordinates. In the case of `plot"ea"` they are relative to RC, in the case of `plot"er"` they are relative to the current plot position.

Example

```

10 plotnew
20 plot"ea",639,511 :rem plots an edge on complete screen
30 plotnew
40 plot"da",320,256 :rem sets plot position to center
50 plot"er",-200,-100 :rem plots an edge from center position
to the left down position x=120, y=156
60 plot"vr",0,0,0 :rem set new linetype (delete)
70 plot"er",200,100 :rem delete the last edge

```

plot"ci" Plots circle

Using this instruction, a circle is plotted around the current plot position (circle center) with the radius defined in the `plot"ci"` command. A screen window created by `plotdim` or the `window` command will limit the plotted circle. The line type of the circle is preselected by the last vector-plot command (`plot"va"/"vr"`).

Syntax

```

plot"ci",r

r = radius

```

Example

```

10 plotnew
20 plot"ci",200 :rem plots a quarter circle with radius
200 around the lower left screen-edge
30 plotnew
40 plot"da",320,256 :rem sets plot position to center
50 plot"ci",100 :rem plots a circle with radius 100 in
the center of the screen

```

plot"ac"

Plots circular arc

Using this instruction, a circular arc is plotted around the current plot position (arc center) with the start-angle, end-angle and radius defined in the `plot"ac"` command. The resolution of the start- and end-angle is 0.25 degrees (DEG). The line type of the circle is preselected by the last vector-plot command (`plot"va"/"vr"`).

Syntax `plot"ac", sa, ea, r`

sa = start angle in 0.25 DEG
ea = end angle in 0.25 DEG
r = radius

Example

```
10 plotnew
20 plot"ac",30*4,60*4,100 :rem plots a circular arc with
                           radius=100 around the screen-origin from
                           30° to 60°

30 plotnew
40 plot"da",320,256 :rem sets plot position to center
50 plot"ac",270*4,0,100 :rem plots a circular arc with radius
                           100 around the center of the screen from
                           270° to 360° (4th quadrant)
```

plot"iw"

Plots input window

`plotdiml`, `plotdimu` defines a window at the current plot position. The position must be set before. Using `plot"iw"`, a window can be defined without using the current plot position. The edges are defined by absolute coordinates.

Syntax `plot"iw", xll, yll, xur, yur`

xll = lower left x-coordinate
yll = lower left y-coordinate
xur = upper right x-coordinate
yur = upper right y-coordinate

Example

```
10 plot"iw",200,200,400,400 :rem defines a window from position
                           200,200 to 400,400
```

plot"ba"

Plots bar graph

A commonly used scientific and technical method of plotting measured and analyzed data is the bar graph. The `plot"ba"` allows to very easily plot bar graphs.

Syntax `plot"ba", w, h`

w = bar width
h = bar length

`plot"ba"` plots a bar of width *w* and height *h* into the defined graphic window. The area above/below the bar (from *h* to the graphic window border with width *w*) is cleared. Negative values of *h* are plotted from the baseline downwards.

Example

```

10 plot"da",10,255           :rem defines base-line
20 for i=0 to 49             :rem loop defining the number of bars
30 plot"dr",11,0            :rem step to the next bar
40 plot"ba",10,rnd(0)*200-100 :rem plots a random long bar
50 next                      :rem loop end
60 pause2000 : goto10       :rem after a pause plot next set of
                             random bars

```

The program above plots bar graphs with positive and negative random num lengths.

plotmat **Plots curves from a matrix**

One of the most powerful plot commands of *andiBASIC* is `plotmat`. It plots a curve from a numerical matrix. The matrix may contain e.g. measurement data or analyzed data. The points of the curve have the following coordinates:

- first matrix index => x-coordinate
- value of the element => y-coordinate
- second matrix index => curve

The first matrix index represents e.g. the time base of measurement data. The value of this element represents the measured data for that time. A one-dimensional matrix (array) contains the data of one curve only, whereas a two-dimensional matrix contains more than one curve. The number of curves is defined by the second dimension. Please remember that the counting of indices with matrices begins with "0", not with "1".

Example

A(5,2) = two-dimensional matrix with three tracks (second index 0-2) and six elements for each track (first index 0-5).

	Track 0	Track 1	Track 2
Element 0	A(0,0)=100	A(0,1)=250	A(0,2)=550
Element 1	A(1,0)=110	A(1,1)=240	A(1,2)=551
Element 2	A(2,0)=120	A(2,1)=230	A(2,2)=552
Element 3	A(3,0)=130	A(3,1)=220	A(3,2)=553
Element 4	A(4,0)=140	A(4,1)=210	A(4,2)=554
Element 5	A(5,0)=150	A(5,1)=200	A(5,2)=555

Syntax `plotmatA() - (), xf, yf, m`

- A() = numerical array
- xf = x factor
- yf = y factor
- m = mask

A() may be an integer- or a real matrix. The following brackets remain empty as a matrix descriptor. Optionally x or y factors can be programmed which zoom the curve in x or y direction respectively. The mask (m) allows to plot one bit of e.g. the recoding of a 8-bit digital channel. *xf*, *yf* and *m* may be omitted.

Example

```

10 dim a(639)
20 for i=0 to 639
30 a(i)=sin(i/50)*200
40 next
50 plot"sk",0,255
60 for i=.5 to 1 step.02
70 plotmata(),i,i
80 next
    
```

The above program plots the one-dimensional array a(), filled in lines 20 to 40 with a sine function, in various scales. `plotmat` refers to the relative origin of coordinates defined by `plot"sk"`. Since a sine function provides both, positive and negative values, the coordinate-base must be moved upwards to 0,255 (line 50).

In the same way two-dimensional arrays may also be plotted. Each track is drawn as a individual curve. **andiBASIC** does automatically detect how many tracks a matrix provides or how many tracks are to be plotted.

Example

```

10 dima(639,2):plot"sk",0,255
20 for i=1 to 639
30 a(i,0)=200/exp(i/200)
40 a(i,1)=-200/exp(i/200)
50 a(i,2)=200*sin(i/17)/exp(i/200)
60 next
70 plot"vr",639,0:plotmata()
    
```

If only one track has to be plotted from a two-dimensional matrix this track first has to be copied to a one-dimensional array. This is done using either a `for...next` loop or the `matcopy` instruction (option 2).

plotinv
Reverses black and white

With `plotinv` the total graphic window can be inverted i.e. white dots turn to black and vice versa.

Syntax `plotinv`

Please insert the following lines into the previous program :

Example

```

10 dima(639,2):plot"sk",0,255
20 for i=1 to 639
30 a(i,0)=200/exp(i/200)
40 a(i,1)=-200/exp(i/200)
50 a(i,2)=200*sin(i/17)/exp(i/200)
60 next
70 plot"vr",639,0:plotmata()
80 for I=0 to 9
82 plotinv
84 pause1000:next
    
```

The whole screen graphics are inverted over and over again (10 times).

plotnew

Clears the Screen Graphics

With this instruction the whole screen graphics can be cleared. Graphic windows are opened and the RC (relative origin of coordinates) is put in the place of the AC (absolute origin of coordinates).

Syntax `plotnew`

With `plotnew` the complete graphics are initiated and the graphic memory of the selected page is cleared.

! Please note that the order of the graphic pages is kept even after a `plotnew`.

ploton/off

Switches the Micro-Cursor on/off

As in normal mode a cursor is also at hand in graphic mode, the so-called microcursor. It is, however, smaller and consists of merely one pixel. It gives the current position of the coordinates and is altered by the following PLOT instruction:

```
plot"sk"
plot"da"/"dr"
plot"va"/"vr"
plotmat
```

`plotoff` only switches of the blinking micro-cursor. `plotnew` additionally clears the screen.

Syntax `ploton`
`plotoff`

Example

```
10 plotnew:ploton
20 cu$=chr$(27)+chr$(12):cd$=chr$(27)+chr$(11)
22 cr$=chr$(16):cl$=chr$(8)
30 getg$
40 oninstr(cu$+cd$+cr$+cl$,g$)goto30,50,50,60,60,70,80
50 plot"vr",0,1:goto30
60 plot"vr",0,-1:goto30
70 plot"vr",1,0:goto30
80 plot"vr",-1,0:goto30
```

(cu\$=Cursor up, cd\$=Cursor down, cr\$=Cursor right, cl\$=Cursor left -keys)

The above program allows to "draw" lines using the cursor control keys. The code for "cursor up" and "cursor down" consists of two characters (see line 20). Consequently, the appropriate line numbers after `goto` must appear twice.

cpos ()

Gets mouse coordinates

The `cpos ()` function returns the current mouse position from the terminal in a complex variable. The x-position is given by the real-part and the y-position by the imaginary-part of the variable. `cpos (0)` regards the mouse position to the origin of the whole screen (lower left edge). If the coordinate origin is set by `plot"sk"` to a different position on the screen, the function `cpos (1)` returns the mouse coordinates relative to the user-defined coordinate-origin.

Syntax `cpos (0)` get mouse position regarding to screen origin
`cpos (1)` get mouse position regarding to last `plot"sk"`

Example

```

10 plotnew
20 printcpos(0)           :rem prints the mouse position
30 plot"da", cpos(0), 40 :rem plots a crosshair at mouse position
40 plotnew
50 plot"sm", 200, 200    :rem sets mouse position 200/200 to last
                        :rem coordinate-origin
60 plot"sk", 100, 100   :rem sets coordinate-origin to 100/100
70 printcpos(0)        :rem prints mouse position 200, 200
                        :rem (absolute to screen-origin)
80 printcpos(1)        :rem prints mouse position 100, 100
                        :rem (relative to last plot"sk"-position)
90 plotnew
100 printcpos(1)        :rem prints mouse position 200, 200
                        :rem because plotnew sets coordinate-origin
                        :rem to 0, 0

```

plot"sm"

Sets mouse coordinates

The `plot"sm"` allows to set the mouse position to given coordinates relative to the current coordinate-origin. The coordinate-origin is set by `plot"sk"` or will be the screen-origin after `plotnew`.

Syntax

```

plot"sm", xpos, ypos
plot"sm", pos&

```

xpos = x-position relative to the last coordinate-origin
(set by `plot"sk"`)

ypos = y-position relative to the last coordinate-origin
(set by `plot"sk"`)

pos& = mouse position relative to coordinate-origin
(real-part = x-position, imaginary-part = y-position)

Example

```

10 plotnew
20 plot"sm", 200, 200    :rem sets mouse position 200/200
30 printcpos(1)        :rem prints mouse position 200, 200
40 plot"sk", 100, 100  :rem sets coordinate-origin to 100/100
50 plot"sm", 200, 200  :rem sets mouse position 200/200
                        :rem relative to last coordinate-origin
60 printcpos(1)        :rem prints mouse position 200, 200
                        :rem (relative to last plot"sk"-position)
70 printcpos(0)        :rem prints mouse position 100, 100
                        :rem (absolute to screen-origin)
80 plotnew
90 mouse&=cmplx(320, 256) :rem complex variable for mouse position
100 plot"sm", mouse&   :rem sets mouse position to center

```

plotdim1/plotdimu

Defines a Graphic Window

With no graphic window defined the full screen is available for the plot commands. With `plotdim1` and `plotdimu` you set a graphics window and define the area where plotting is allowed. The procedure of defining a graphics window is easy: Jump to the coordinates of the lower left edge of the window (`plot"da"/"dr"` or `plot"va"/"vr"`) followed by a `plotdim1`. Then jump to the coordinates of the upper right edge followed by `plotdimu`.

Syntax

```

plotdim1
plotdimu

```

Please erase the line 100 from the second example in chapter E.XII.5 and append the following lines to the end:

Example

```
10 plot"da",200,200:plotdiml
20 plot"dr",100,100:plotdimu
22 for i=0 to 9
30 plotclr,192
40 pause200:next
```

In the program above graphic windows are defined which are then inverted. For this the `plotclr` instruction is used. This instruction is described in the next chapter.

plotclr **Fills a Graphic Window**

`plotclr` fills the current screen window either white, black or reverse according to the argument.

Syntax `plotclr, ar`

ar = argument

Valid arguments are:

ar	result
0	black
128	white
192	reverse

The example in the previous chapter shows the effect of `plotclr,192`. Change line 30 to `plotclr,0` and `plotclr,128` and watch the differences.

plot"ca"/"cr" **Clears a graphic window**

`plotclr,plot"ba"` deletes, sets or inverts a part of the screen only when the graphic window is limit by a window (instruction) before. The `plot"ca"/"cr"` instruction can delete or set an arbitrary graphic window without being limited by a window before. However set Windows are considered. The `plot"ca"/"cr"` instruction deletes a graphic window defined from the current plot position to the end-position indicated by the instruction-arguments. The end-position is defined by absolute or relative coordinates. The delete mode is specified by the third parameter similar to the linetype-argument of the `plot"va"/"vr"` instruction. The default value is 0 for solid black.

Syntax `plot"ca", xa, ya, mode`
`plot"cr", xr, yr, mode`

- xa* = x-coordinate of end-position (absolute)
- ya* = y-coordinate of end-position (absolute)
- xr* = x-coordinate of end-position (relative)
- yr* = y-coordinate of end-position (relative)
- mode* = delete mode

mode is (equal to *l* for the `plot"va"/"vr"` instruction) an 8-bit-integer value.

The meaning of the bits are as follows:

mode = *mmnnnttt*_{bin}

mm = plot mode

mm	operation	line
00	AND	black
01	only <i>nnnttt</i> _{bin} is used	
10	OR	normal
11	XOR	reverse

The plot mode is kept until a new plot mode is defined. The default is $10_{bin} = 128_d =$ normal plot.

nnn = thickness of the line

Values of 0 – 7 result in lines of 1 to 15 pixel thickness

ttt = line type

ttt	line type
0	solid
1	50% solid
2	narrow shaded 45°
3	narrow shaded 135°
4	wide shaded 45°
5	wide shaded 135°
6	solid
7	solid

Example

```

10 plotnew
20 plotinv           :rem inverts the hole screen
30 plot"ca", 320, 256 :rem deletes the lower left quarter of
                       the screen
40 plot"da", 639, 511 :rem sets current position to upper
                       right edge of the screen
50 plot"cr", -100, -100 :rem deletes a square of 100 pixels in
                          the upper right edge of the screen
60 plotnew
70 plot"iw", 320, 256 :rem sets a window in the lower left
                       quarter of the screen
80 plot"da", 200, 200 :rem sets current position to 200/200
90 plot"ca", 400, 400, 192 :rem inverts only the screen range from
                              200/200 to 320/256 (limited by the
                              window)

```

\$1b0614/\$1b0615 Defines macro of plot-instructions

Fast plot instructions or repeated plots are more limited by the communication time of the serial interface, not by the cpu. For those applications (i.e. moving parts of a plot on the screen) the macro-function is favored.

The operational sequence in principle is:

1. Macro start The opening string (**\$1b0614**) indicates the terminal, that the following plot-instructions are stored in the macro memory and not being processed.
2. Macro definition The opening string is followed by the plot-instructions which describes the plot-design. All plot-instructions except **plotnew** may be used. All instructions are only stored in the macro memory. **plotnew** deletes the current macro-definition.
3. Macro end The closing string (**\$1b0615**) indicates the terminal the end of the macro-definition. The macro is now defined and available. All following plot-instructions are processed normally.
4. Macro execution The macro-function can be combined with other instructions and processed arbitrarily often by the execution string (**\$1b0616**).

The following example will move a framed text from lower left to upper right of the screen. For a clearer program list the command codes for line type and macro-function are replaced by variables.

Example

```

10 Invert$=hex$("1b060fc0") :rem linetype solid inverted
20 Begin$=hex$("1b0614") :rem opening string for macro
30 End$=hex$("1b0615") :rem closing string for macro
40 Macro$=hex$("1b0616") :rem execution string for macro
45:
50 plotnew:printInvert$; :rem initialization and linetype
60 plot"sy",0,3 :rem text size 3
65:
70 printBegin$; :rem start of macro-defintion
80 text$="plot with macros" :rem macro-code (not processed)
90 plot"da",0,0 :rem macro-code (not processed)
100 plot"er",380,26 :rem macro-code (not processed)
110 plot"dr",4,4,text$ :rem macro-code (not processed)
120 printEnd$; :rem end of macro-definition
125:
130 forxy=0to400step4 :rem begin of plotting loop
140 plot"sk",xy,xy: :rem shift point of reference
150 printMacro$; :rem execute macro
160 pause 5 :rem short pause for better view
170 printMacro$; :rem deletes plot by inverting
180 next :rem next plotting run

```

plothardcopy

Outputs the Complete Screen to the Printer

Using **plothardcopy** a complete graphic screen can be output to the PC printer. This helps the programmer enormously since graphics produced on the screen do not have to be reprogrammed for the printer. Thus the User can see the graphics on the screen first before having them printed. There are many printers which **andiBASIC** -68K will support (see appendix). The device address of the IEEE-Bus-printer is pre-adapted to 4 ; devices adapted in another way must be re-adapted to 4 or the content of the system variable Printer has to be set equal to the device address.

Syntax

```

plothardcopy
plothardcopy" S$"

```

- | | |
|-----|------------------------------------|
| S\$ | <i>Printtype</i> |
| b | big hardcopy on NEC P6 |
| 0 | small hardcopy on NEC P6 |
| 1 | small hardcopy with text on NEC P6 |
| 2 | big hardcopy on NEC P6 |
| 3 | big hardcopy with text on NEC P6 |
| 4 | small hardcopy on NEC 8023 |
| 5 | small hardcopy with text on 8023 |
| 6 | big hardcopy on NEC 8023 |
| 7 | big hardcopy on with text NEC 8023 |

System Level Links

peek

Reads a memory location

With the **peek** instruction the contents of single memory locations can be loaded into **andiBASIC**. You may choose between Bytes (8 Bit), Words (16 Bit), Long-Words (32 Bit) and Strings. Accordingly, a 8, 16 or 32 Bit value is transferred. For strings, the number of bytes is defined by the number of string elements.

Syntax	peek (<i>adr</i>)	Byte
	deek (<i>adr</i>)	Word
	leek (<i>adr</i>)	Long-Word
	steek (<i>adr</i> , <i>len</i>)	String

adr = Memory address
len = Length of the string (byte)

Words consist of 2, Long-Words of 4 successive bytes. *adr* is the address of the first byte. The same is true for strings. The content of the first byte (basis address) represents the high-byte[^], the following byte(s) are the lower-bytes. The **peek** () function converts these bytes into one decimal number. Because of the memory structure it makes no sense to **deek** or **leek** a Word or Long-Word with a odd-numbered basis address. Thus, the **deek** () and **leek** () functions automatically check if the defined address *adr* is even-numbered.

Examples

```
printpeek(0)
a=deek(2)
b=leek(4)
printsteek(8,16)
```

poke

Writes to a memory location

The **poke** instruction writes a value into a memory location. In the same way as with peek you may choose between Byte, Word, Longword and String.

Syntax	poke <i>adr</i> , <i>value</i>
	doke <i>adr</i> , <i>value</i>
	loke <i>adr</i> , <i>value</i>
	stoke <i>adr</i> , <i>string</i>

adr = address
value = value to be written to the memory location
string = string to be written to the memory location

For the same reasons as with **deek** () and **leek** () only even-numbered addresses can be used with **doke** and **loke**.

Examples

```
10 adr=hex("150000"):pokeadr,123456789
20 a=leek(adr)
30 b=peek(adr+3):b=2^8b+peek(adr+2)
40 b=b+2^16*peek(adr+1):b=b+2^32*peek(adr)
50 printa,b
```

sys
Calls an assembler routine

Using `sys` control can be transferred from the Basic level to a machine subprogram. This is then executed up until the machine language command `rts` (return from subroutine) and then returns to the Basic statement (Basic level) following the `sys`. Therefore `sys ... rts` form a main/subprogram structure similarly to `gosub ... return` with the difference that the main program is a Basic program and the subprogram a machine program. However, the return address of `sys` is also determined by the stack. Accordingly, control can be transferred from the machine sub-routine to deeper subprogram levels. `rts` will jump to the next highest level.

Syntax `sysadr`

adr = destination address

The destination address must be a valid address at hand in the RAM memory. *adr* may be hexadecimal (in quotation marks) or decimal (without quotation marks).

Examples

```
sys 110000      jumps into dec. 110000
sys "54000"    jumps into hex. 54000 (= Basic cold-start !!)
sys "54004"    jumps into hex. 54004 (= Basic warm-start !!)
```

usr / setusr
Calls an assembler routine with data transfer

`usr` calls a assembler routine in the same way as `sys` does. Additionally, one or more data can be transferred from the calling *andiBASIC* level to the machine level and one data is returned. The destination address is defined by the `setusr` command. Since one must be able to get from a Basic frame program to several MSPgs it is practical to be able to define several `USR` functions. This is possible in *andiBASIC*. `usr` as well as `setusr` is followed by a routine name. The number of `usr` functions (with appropriate addresses) is not limited.

Syntax `setusrname=adr`
`var = usrname(value)`

name = name of the *usr* routine

var = numerical variable (integer or float) keeping the result

adr = memory address of the *usr* routine

value = data transferred to the *usr* routine

Whereas the `setusr` function defines the name and the memory location of the `usr` routine, the `usr` function executes the routine and transfers the returned value (result of the use routine) into the variable `var`.

Example

```
setusrSubRoutine1=hex("190000")
a=usrSubRoutine1(20*13.7564)
printa
```

The address of the `usr` function must be set as a decimal value. Therefore, in the example the `hex` function must be used to define the memory location (`hex`) \$190000. The result of the mathematical expression (20 * 13.7564) is transferred to the `usr` routine. The result of the `usr` routine can be found in the variable "a" after returning to *andiBasic* level.

The writing of a machine program is not be the subject of this *andiBasic* manual.

wait**Pauses the execution of a program**

wait interrupts a Basic program up to the point when a fixed value is encountered at a fixed memory location.

Syntax `waittime`

time = time to wait [ms]

A *wait* command can only be left with double NMI or RESET.

moni**Calls the system monitor**

The command **moni** calls up the machine language monitor. The functions of **ANDIMON** are described in a separate chapter (Monitor instructions).

External Devices

Introduction

External devices connected to the IM system are addressed by a couple of numbers: The *device number* defines the device, the *drive number* selects the drive or partition, the *logical address* represents a virtual channel on which data are to be exchanged whereas the *secondary address* executes special functions.

Before data can be transferred from or to a device, you have to open a (virtual) logical channel to that device. With the `open` command you assign a device number and a secondary address to a logical address. After opening you are able to transfer the data through this logical channel, referring to the *LAdr* only. After the data transfer is completed you have to close the logical channel.

Device Number *Dev*

Each device connected to the IM (internal hard disk, PC file system, PC printer, etc.) must have an individual and unique device number (*Dev*). *Dev* is a number between 0 and 255 so that a maximum of 255 devices can be handled. The default Device Number is set globally with the *system variable* `Floppy`. *Dev* numbers must also be defined in the open command for file operations.

The following *Dev* numbers are reserved:

Dev	Device
0	terminal (Screen + Keyboard)
1	Internal IM floppy disc controller
2	Internal hard drive
3	terminal (Screen + Keyboard)
4 - 15	IEEE-488 devices
16 - 31	RS-232 devices
65	PC file system
66	PC printer
67	PC Com1 (RS232)
68	PC Com2 (RS232)

Drive Number *Drv*

A device may contain different "drives". This may be the different partitions of the PC file system or of the internal HD or the drives of double floppy-disc drive of older IM systems. These partitions/drives are addressed by the *Drv* number. Please note that the *Drv* numbers start with "0". The *Drv* may be defined either in the open string for file operations or globally by the system variable *Drive*.

The following numbers are used for the PC file system:

Drv	Partition
0	A (first floppy disk drive)
1	B (second floppy disk drive)
2	C (HD)
3	D
...	...

Logical Address *LAdr*

The *LAdr* is a number defining a virtual (logical) channel on which data can be transferred. After you

have opened a *LAdr* on a specific device you are able to handle even complex transfers using only the *LAdr*. *LAdr* is a number between 0 and 255 so that a maximum of 255 logical channels can be open at the same time.

Secondary Address *SAdr*

The *SAdr* is set with the open command optionally. It may select special operation modes for the device (e.g. printer). For standard data transfers the *SAdr* is not needed and may be omitted.

andiBASIC provides special commands for special devices. For example: print sends data exclusively to the monitor or a printer, get picks a character from the keyboard or a storage device, save sends data to a storage device.

data files (numbers, text, variables) differ fundamentally from program files. Since saving and loading programs are the most frequent operations, a set of special statements is available: asave, dsave, aload, dload. No logical address has to be opened for program-load/save operations. Note that a logical address has to be opened only for statements with "#" (save#, load#, print#).

File Name

Data or program files are identified on a storage device by their file names. This name will appear in the directory list and with this name you can load or delete the file. The file name is defined with the saving procedure. It may consist of all printable characters except the characters "\, ?, *". The file name can contain further optional parameters such as the drive number, the file type (program, data, etc.) and the file access mode (read, write).

Storage Instructions

Device Numbers and File Types

As these operations are often used and thus ought to be as simple and efficient as possible, a special set of commands is already available. All these storage commands automatically address floppy disc or hard drives installed in the IM6/6e as well as the PC file system. On the PC file system all drive types are supported. Direct burning of CDs and DVDs is not possible from the Thales environment. Thales supports up to 12 partitions of the PC file system. If there are more than 12 partitions, the first 12 are accepted and the rest is ignored.

As stated above, the default device number is set globally by the system variable **Floppy**:

Floppy=65	=> PC file system is addressed
Floppy=2	=> internal IM hard drive is addressed

For a list of reserved device numbers refer to the table in the previous chapter. The default value set in recent Thales versions at start-up is 65 (PC file system). Older Thales versions set the start-up default to 2 (internal IM hard drive).

In the same way the Drive is assigned globally by the system variable **Drive**.

Floppy=65	=> PC file system is addressed
Drive=2	=> addresses partition C: on the PC hard drive

In principle, on mass storage devices (floppy-disc and hard disc) may have two different types of files available: program files and data files. Program files are designated as runtime-files, data as sequential files. Accordingly they are labeled in the disc (hard disc) directory with rtm (runtime) or seq (sequential), respectively. Both types have different structures and are handled in a different way for loading and saving. For example, loaded sequential files cannot be run or loaded with the start command. To avoid trouble it is essential to provide different commands or different syntax for these file types. **andiBASIC** distinguishes between commands handling runtime files and those treating data

(sequential) files. This difference in handling was also discussed in the previous chapter:

save does only save programs
save# does only save data

open **oP**
Opens a Logical Channel

Whereas **dopen** opens a file on the hard disks and floppy disks, **open** refers to any peripheral device (hard disk, printer, monitor, RS232, etc.).

Syntax `openLAdr, Dev, SAdr, "String"`

LAdr = Logical Address (number between 0 and 255)
Dev = Device Number
 0 = terminal (screen and keyboard)
 1 = internal IM floppy disc controller
 2 = internal IM harddisc
 3 = terminal (screen and keyboard)
 4-15 = IEEE-488 devices (IM)
 16-31 = RS-232 devices (IM)
 65 = PC Filesystem
 66 = PC printer
 67 = PC Com1 interface
 68 = PC Com2 interface

SAdr = Secondary Address
String = [DirOW]Drive:\Path\Filename[.Ext],[Type],[WEna]

[DirOW] = \$ for reading directory
 @ overwrite existing file

Drive = C (PC 1st Partition)
 D (PC 2nd Partition)
 and so on...

Path = Path to File
 Filename = Name of File
 [.Ext] = Extension of File
 [Type] = Filetype
 b = Binary data (*.bin)
 p = Programm (*.rtm)
 s = Sequential/ Measured Data

[WEna] = w = write access
 r = read access (preset)

Examples

```

10 rem Opens PC-Harddrive
20 open1,65,3,"$c:\flink,r"           :rem opens directory c:\flink
30 fori=1to10                         :rem loop
40 input#1,f$:printf$:next           :rem reads one line of the
                                       directory and print it on screen
60 close1                             :rem close logical channel 1
70 :
100 open1,65,3,"c:\thales\test.txt,w" :rem creates file test.txt
110 a$="File-Write Demo"
120 b%=-123
130 dimc(99,2)
140 c(33,1)=331
150 save#1,a$,b%,c() : close1         :rem saves text to file
160 a$="" :b%=0:c(33,1)=0
170 open1,65,3,"c:\thales\test.txt"   :rem opens test.txt for
                                       reading
180 load#1,a$,b%,c() : close1         :rem reads first text
190 a$=a$+" for Overwriting"
200 open1,65,3,"@c:\thales\test.txt,w" :rem opens existing file
                                       test.txt for writing
210 save#1,a$,b%,c() : close1         :rem overwrites test.txt

```

open

Opens a COM-Port by a Logical Channel

oP

Whereas **dopen** opens a file on the hard disks and floppy disks, **open** refers to any peripheral device (hard disk, printer, monitor, RS232, etc.).

Syntax `openLAdr"COM-Setup"`

LAdr = Logical Address of opened device (number between 0 an 255)
COM-Setup: `baud = 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400` (baud rate)
`parity = n` (no parity-bit), even (even parity-bit), odd (odd parity-bit)
`data = 5, 6, 7, 8` (number of databits)
`stop = 1, 2` (number of stopbits)
`[rts] = off, on` (request to send)
`[octs] = off, on` (output clear to send)

Examples

```
10 rem Opens PC Com1 with 38k4 baudrate, no parity, 8 databits and 2
stopbits
20 open1,67,1,"baud=38400 parity=n data=8 stop=2"
30 Timeout=100 :rem timeout setting in ms
40 print#1 chr$(129) :rem output character on Com1
50 get#1,a$ :rem input character from Com1
60 close1 :close PC Com1
70 :
100 rem Opens PC Com1 with 38k4 baudrate, no parity, 8 databits, 2
stopbits and hardware-handshake
110 open1,67,1,"baud=38400 parity=n data=8 stop=2 octs=on rts=on"
120 Timeout=1
130 print#1 chr$(129)
140 get#1,a$
150 close1
```

close

Closes a Logical Channel

Whereas **open** opens any peripheral device (hard disk, printer, monitor, RS232, etc.) in a logical address, **close** closes these logical address.

Syntax `close[LAdr]`

LAdr = Logical Address of opened device (number between 0 an 255)

For Example look at **open**

save#/load#

Saves/Loads on Logical Address

save# writes data from memory to a device which is defined by a logical address.

Syntax `save#LAdr,Var1,Var2,...` writes Var1,Var2,... to opened file

LAdr = Logical Address of opened device (number between 0 an 255)
Var1,Var2,... = all kinds of variables and arrays

With `load#` you load previously saved data (**andiBASIC** program, data,...) from a logical addressed device into the memory. Consider the correct sequence of the data types.

Syntax `load#LAdr, Var1, Var2, ...` reads data from file into Var1, Var2,...

LAdr = Logical Address of opened device (number between 0 an 255)
Var1, Var2, ... = all kinds of variables and arrays even with mixed data types

For Example look at `open`

`input#` Input on a Logical Address

`input#` reads data from a device which is defined by a logical address.

Syntax `input#LAdr, Var` reads a data-line from file into Var

LAdr = Logical Address of opened device (number between 0 an 255)
Var = variable of matching data type

For Example look at `open`

`get#/get` Input on a Logical Address / Keyboard

`get#` reads data from a interface device which is defined by a logical address into memory.
`get` reads a character from the keyboard into memory.

Syntax `get#LAdr, Var` reads data from interface device to Var
`getVar` reads a character from keyboard to Var

LAdr = Logical Address of opened device (number between 0 an 255)
Var = variable of matching data type

Examples

```
10 whileA$=""                      :rem loops till keystroke
20 getA$                            :rem reads character from keyboard
30 wend
40 printA$
```

For `get#` examples look at `open`

`get#count ()` Multiple Input on a Logical Address `getcount ()`

`get#count ()` reads a number of data from a interface device which is defined by a logical address into memory.

`getcount ()` reads a number of characters from the keyboard into memory.

Syntax `get#LAdr, count (n) Var` reads n data from device LAdr to Var
`getcount (n) Var` reads n character from keyboard,
program waits until input is finished

LAdr = Logical Address of opened device (number between 0 an 255)
n = number of data / character
Var = variable of matching data type

Examples

```

open3,65,1,"c:\thales\temp\gettest.txt,w"      :rem create text file
  print#3,"getcounttest"                       :rem
close3                                          :rem
open2,65,1,"c:\thales\temp\gettest.txt"       :rem open text file
  get#2,count(8)A$                             :rem reads 8 characters from file
close2
printA$                                        :rem A$="getcount"
getcount(3)A$:printA$                          :rem reads three characters from keyboard
                                          :rem 1,2,3 => A$="123"
getcount(4)B%:printB%                          :rem reads a 4-place integer from keyboard
                                          :rem -,0,4,2 => B%= -42
getcount(5)C:printC                            :rem reads a 5-place real from keyboard
                                          :rem 1,.,2,3,4 => C= 1.234
                                          :rem 1,.,2,e,3 => C= 1200
  
```

print# **Output on a Logical Address** **?#**

print# writes data from memory to a interface device which is defined by a logical address.

Syntax **print#***LAdr, Var* writes *Var* to a interface device

LAdr = Logical Address of opened device (number between 0 an 255)
Var = variable

For Example look at open

directory **Show File List** **diR**

This command prints a list of the names of all or of specific files stored in a partition.

Syntax **directory** [*dDrv*] [, "*String*"] lists all files in the assigned partition
diR [*dDrv*] [, "*String*"] lists all files in the assigned partition
directory ["*String*"] [, *dDrv*] lists all files in the assigned partition

Drv = Drive number
String = Searchstring

The parameters *Drv* and *String* are optional. If no drive number is defined, the default drive assigned with the system variable *Drive* is taken. If no search string is defined, all files in the partition are listed. A search string may restrict the output to file names starting with specific characters. As a joker for one character use the question mark "?", as a joker for more than one character use the asterisk "*".

Examples

```

Floppy=65:diRd2      :rem lists the complete contents of PC HD c:\
Floppy=2:diRd8       :rem lists the complete contents of IM HD root
Floppy=65:diRd0,"im*" :rem lists all files on PC HD a:\ which begin
                    :rem with the characters „im“
Floppy=65:dird0,"im" :rem lists the file on PC HD a:\ with the file
                    :rem name „im“
Floppy=65:diRd2,"i?m*" :rem lists all files on PC HD c:\ which have
                    :rem an "i" in the first and an "m" in the
                    :rem third position
  
```

dstate

Set Directory

This command set the directory for d2.

Syntax `dstate "a:\ [Path$] "` set d2 to c:\path

Path = Pathstring (for example thales\temp\)

If no path string is defined, the directory d2 is set to c:\ root directory.

Examples

```
dstate"a:\thales\temp\" :rem set d2 to c:\thales\temp
diRd2                  :rem print file list of c:\thales\temp

dstate"a:\flink\"      :rem set d2 to c:\flink
diRd2                  :rem print file list of c:\flink

dstate"a:\"             :rem set d2 to c:\
diRd2                  :rem print file list of c:\
```

scratch

Delete File

scR

This command deletes a file from the selected directory (d2). Before the file will be deleted the user have to confirm "are you sure?" with "y". All other inputs will cancel the scratch command.

Syntax `scratch d [Drv] , " [File$] "` delete *File\$* in directory *Drv*

Drv = drive number

File\$ = file name ("filename.extension")

If no path string is defined, the directory d2 is set to c:\ root directory.

Examples

```
dstate"a:\thales\temp\" :rem set d2 to c:\thales\temp
scratchd2,"testfile.txt" ENTER
are you sure?           :rem confirm with  ENTER
01 files scratched
```

concat

Concatenate Sequential Files

This command attach a sequential file (A) to the end of an other sequential file (B). After the concat command the destination file (B) contains both data. The attached file (A) will not be changed.

Syntax `concatd2, [Attach$] to d2, " [Dest$] "` append file *Attach\$* to *Dest\$*

Attach\$ = file to be attached ("path\filename.extension")

Dest\$ = destination file ("path\filename.extension")

Examples

```
open1,65,1,"c:\thales\temp\attach.txt,w"
fori=10to19:print#2,"Line number "+str$(i):nexti
close1
open1,65,1,"c:\thales\temp\dest.txt,w"
fori=0to9:print#2,"Line number "+str$(i):nexti
close1
concatd2,"\thales\temp\attach.txt" to d2,"\thales\temp\dest.txt"
```

dsave/dload

Saves/Loads Programs

dIO/dsA

dsave writes a **andiBASIC** program from memory to a mass storage device.

Syntax	dsave [dDrv ,] " <i>File Name</i> , s,w "	writes a data file
	dsave [dDrv ,] " <i>File Name</i> , p,w "	writes a program file
	dsave " <i>File Name</i> , s,w " [, dDrv]	

The **File Name** is obligatory and must consist of 1 to 16 printable characters except the characters "\, ?, *". The **Drv** number is optional. If it is omitted, the recently assigned drive number is taken. If both, file name and drive number are defined, they are separated by a comma.

With **dload** you load a previously save **andiBASIC** program from a mass storage device into the memory. Previously memory-resident programs will be erased.

Syntax	dload [dDrv ,] " <i>File Name</i> "
	dload " <i>File Name</i> " [, dDvnr]

For **File Name** and **Drv** the same is true as it is for **dsave**.

dstart

Loads and Starts a Program

dsT

dstart loads an **andiBASIC** program from a mass storage device into memory and automatically starts it. The syntax corresponds to that of **dload**.

Syntax	dstart [dDrv ,] " <i>File Name</i> "
	dstart " <i>File Name</i> " [, dDvnr]

For **File Name** and **Drv** the same is true as it is for **dsave**.

asave/aload

Saves/Loads a Program as an ASCII File

aIO/asA

andiBASIC supports two different program data formats on a mass storage device: the completer code and the ASCII code. All in/output and floppy disc commands apart from the "A commands" **asave**, **aload** and **amerge** work with Completer code data. With **asave**, on the other hand, programs can be stored as ASCII files and reloaded with **aload**. Both data formats have advantages and disadvantages:

Completer code programs are stored and loaded faster. It may be, however, that when changing the operating system or the DOS (e.g. updates) these programs will not be able to be read correctly. ASCII code programs are certainly loaded more slowly but can still be read after changing the operating system or the DOS. Therefore we recommend that security copies of programs are fundamentally saved as ASCII files. Due to the way they are organized ASCII files take up less memory space on the disc than Completer code files.

Syntax	asave [dDvnr ,] " <i>File Name</i> "
	asave " <i>File Name</i> " [, dDvnr]

ASCII files can be reloaded with **aload**, when required.

Syntax	aload [dDvnr ,] " <i>File Name</i> "
	aload " <i>File Name</i> " [, dDvnr]

If your try to load an "d-file" with **aload** the data coming from the disc will be interpreted wrongly and at best results in senseless data in the memory. It can also happen that the **andiBASIC** continuously attempts to load data. This can be stopped by pressing the NMI or RESET key.

If an attempt is made to load an ASCII code, program with `dload`, the following is displayed: no program file forced load ? This message must be confirmed with the RETURN key (empty input !!) otherwise the computer may jump into monitor. It is, however, not at all possible to load ASCII code programs with `dload`.

dopen#

Opens a Logical Channel

doP

Whereas `open` opens a file on any peripheral device (hard disk, printer, monitor, etc.), `dopen` refers specially to the hard disks and floppy disks. Therefore, with "D commands" the device address does not need to be specified.

Syntax `dopen#LAdr, "File Name" [, Acc, dDrv, uDev]`

- LAdr* = Logical Address
- Acc* = Access (*r* = read, *w* = write)
- Drv* = Drive number
- Dev* = Device number

If **Acc** is omitted the channel is opened for reading, if **Drv** is omitted, the drive specified with **Drive** is used and if **Dev** is omitted, the device specified with **Floppy** is used.

Examples

```
dopen#1,"test",w,d2,u65 :rem opens a logical channel for writing (w)
                        on partition c (d2) of the PC hard drive (U65)
                        named "test"
-----
Floppy=65:Drive=2
dopen#1,"test",w      :rem same function as above
```

dclose#
dclose

Closes a Logical Channel

dcL

`dclose` closes a logical channel previously opened with the `dopen` command. If an *LAdr* is specified only this specific channel is closed. Without a *LAdr* all open logical channels are closed. Then, all *LAdrs* are free for use.

Furthermore, `dclose` updates the "file allocation table" (FAT) of the hard disc. Thus, a file and its complete contents is only available after its logical channel had been closed. Therefore, opened files should be closed immediately after writing or reading. Up to 16 files can be opened simultaneously.

Syntax `dclose#LAdr, uDev` closes the logical channel *LAdr*
`dclose` closes all open logical channels

- LAdr* = Logical Address
- Dev* = Device number

! When writing to a removable medium, close all channels before you remove the medium. Otherwise all data on the medium may be corrupted.

rename

Renames a File

reN

With this command the name of files stored on hard disc may be changed. The file to be renamed has to be closed regularly, before.

If the new name already exist in the directory the message **"FILE EXISTS"** is displayed.

Syntax `rename [dDrv, uDev,] "OLD NAME"to"NEW NAME"`

Example

```
renamed2,u65,"test"to"letstry"      :rem changes the file name from
                                     "test" to "letstry"
```

copy

Copies files

coP

The **copy** command copies data from one hard disk location to another one. If the destination file name already exists in this folder the message **"FILE EXISTS"** is displayed. Files can only be copied on disc drives with the same **Dev**.

Syntax `copydDrv1, "FILENAME 1"todDrv2"FILENAME 2", uDev`
Drv1 = Source drive
Drv2 = Destination drive
Dev = Device number

Examples

```
copyd2,"test"tod3,"letstry",u65     :rem copies the file "test" on the
                                     c partition of the PC hard drive to the
                                     d partition using the name "letstry"
-----
copy"test"tod3                       :rem copies the file "test" from the
                                     active partition to the d partition
-----
copyd2tod3                           :rem copies all files of the c partition
                                     of the PC hard drive to the d partition
```

init

Initializes Peripheral Devices

The **init** command initializes the internal mass storage interfaces of the IM. If the PC hard drive is used the **init** command has no effect.

Syntax `initDev`
Dev = Device number

ds / ds\$

Disk Status (System Variable)

If an error occurs during the program run caused by disk handling, you may want to react on the error specifically. To be able to do that, an error number as well as the error text is automatically loaded to 2 system variables by **anbiBASIC**.

ds holds the error number
ds\$ holds the error text

For a list of disk error numbers and disk error texts please refer to the appendix of this manual.

Example

```
100 dopen#1,"test file"  
110 input#1,a%,b,c$:if ds goto DISKERROR  
120 dclose#1  
200 DISKERROR::  
210 print"Disk Error: "ds$  
220 print"Press any key to continue"  
230 dclose  
240 getg$:ifg$=""goto240:else goto100
```

Signal Acquisition

ina()

Read Analog Input Channel

ina() reads an analog channel. The channel (0 to 128) is specified with the command. The value may be printed or read into a variable.

Syntax **ina(ch)**
ch = channel to read

ind()

Reads an 8-Bit-Digital Input Channel

ind() reads a 8-Bit value from a digital input card (DIO). The channel (= card number; 0 to 128) is specified with the command. The value may be printed or read into a variable.

Syntax **ind(ch)**
ch = channel to read

outa()

Sets an Analog Output Channel

outa() sets an analog output channel to the voltage specified with the command.

Syntax **outa(ch, val)**
ch = channel to read
val = output value

outd()

Sets an 8-Bit-Digital Output Channel

outd() sets an 8-bit digital output channel.

Syntax **outd(ch, val)**
ch = channel to read
val = output value

val represents the value of the 8 bits of the digital channel to be set. It calculates as follows:
 $128 \cdot \text{bit7} + 64 \cdot \text{bit6} + 32 \cdot \text{bit5} + 16 \cdot \text{bit4} + 8 \cdot \text{bit3} + 4 \cdot \text{bit2} + 2 \cdot \text{bit1} + \text{bit0}$
 bit0 to bit7 are digital values of 0 or 1.

outk()

Sets an 8-Bit-Digital andiBus Address

outk() sets an 8-bit digital-address of the andiBus.

Syntax **outk(ad, val)**
ad = bus address (decimal)
val = output value

val represents the value of the 8 bits of the digital channel to be set. It calculates as follows:
 $128 \cdot \text{bit7} + 64 \cdot \text{bit6} + 32 \cdot \text{bit5} + 16 \cdot \text{bit4} + 8 \cdot \text{bit3} + 4 \cdot \text{bit2} + 2 \cdot \text{bit1} + \text{bit0}$
 bit0 to bit7 are digital values of 0 or 1.

`tcin` is a very powerful macro-command sampling data from an analog or digital input at a fixed sampling rate into a pre-dimensioned array. The array must be of integer type (%-ending) and thus handles 16-bit data per element. If you define a two-dimensional integer array, multi-channel sampling is performed automatically. The number of channels is defined by the value of the second dimension. The number of samples per channel is defined by the value of the first dimension. Arrays with more than two dimensions are not accepted for `tcin`. The sampling rate, the trigger conditions and the (starting) channel are defined by separate system variables.

The following programming steps have to be done to set up `tcin` properly:

1. Defining a sampling array: e.g. `dim%(249,2)` means 3 channels with 250 samples each)
2. Setting the sampling rate: e.g. `Timebase=500` means a sampling rate of 20 kHz
3. Setting the starting channel: e.g. `Channel=2` means the first channel to sample is channel 2
4. Setting the trigger conditions: e.g. `Trigger=100` means 100 pre-trigger samples are kept
5. `tcina%()`: starts the sampling (eventually waiting for an external trigger signal at the trigger input of the AD-converter)

Defining a sampling array

Use the `dim` command described above to define a sampling array.

Examples

<code>100 dim%(999)</code>	<code>:rem 1 channel, 1000 samples</code>
<code>110 dimdata%(499,3)</code>	<code>:rem 4 channels, 500 samples each</code>

The number of total samples is limited by the memory available. The number of total samples is calculated as:

$$N = (\textit{dimension1} + 1) * (\textit{dimension2} + 1)$$

The free memory available can be found out with the command

`N1=fre(0)`

The number of channels (*dimension2*) is limited to 256.

Setting the time base

With the system variable `Timebase` the sampling rate is set in the unit of μs . The sampling rate is limited by the hardware specifications of the AD converter. The IM systems use a 2-channel parallel-AD-converter with a maximum rate of 500 kHz. Thus, the minimum `Timebase` setting is 2 (μs).

Setting the trigger conditions

`tcin` allows to keep a certain amount of samples which were measured before an external trigger signal at the trigger input of the AD-converter was detected. The amount of pre-trigger sampled is set with the system variable `Trigger`. The maximum amount of pre-trigger samples is the amount of total samples. If set to zero (`Trigger=0`) external triggering is disabled and the measurement starts as soon as the `tcin` command is executed.

If pre-triggering is enabled sampling starts as soon as the `tcin` command is executed for the amount of pre-trigger samples without accepting a trigger event. Only after that time trigger-events are processed.

Setting the starting channel

When setting up a one-channel measurement, the system variable `Channel` (note the capital C !) defines the channel the samples have to be taken from.

When setting up a multi-channel measurement, `Channel` defines the channel, the multi-channel measurement is starting with. The second channel is (`Channel + 1`), etc.

Examples

```

10 dima%(99)                :rem 1 channel, 100 samples
20 dimb%(999,4)             :rem 5 channels, 1000 samples each
30 dimc%(999,1)             :rem 2 channels, 1000 samples each
40 input"KANAL",Channel:tcina%() :rem samples Channel, external
                                clock is possible
50 Timebase=1e5:Channel=0:tcina%() :rem samples channel 0 with 10 Hz
                                sampling rate (duration: 10 sec),
                                no external clock possible

60 Channel=2:Trigger=400
70 Timebase=1000:tcinb%()    :rem recurring sampling of channels
                                2-6 to b%(SAMPLES,0-4), timebase
                                1ms = 1 kHz sample rate, enable
                                trigger and keep 400 pre-trigger
                                samples, after 600 post-trigger
                                samples the array b%() is being
                                sorted to fit the trigger event to
                                the center of the array

80 Timebase=195:tcina%()    :rem a%() is sampled with software-
                                timebase
90 Timebase=200:tcinb%()    :rem b%() is sampled with crystal-
                                timebase

100 Timebase=2:Trigger=100
110 tcina%()                :rem pipelineing-tcin, 500kHz, max.
                                64k samples, trigger-prehistory
                                enabled, at pipelineing-tcin the
                                starting channel is always 0

120:
130 rem new example
140:
200 dima%(999,1)            :rem 2 channels, 1000 samples each
210 Timebase=100           :rem 10 kHz sampling rate
220 Channel=2              :rem sample channel 2 & 3
230 Trigger=100           :rem enable trigger and keep 100
                                pre-trigger samples
240 tcina%()              :rem execute sampling

```

Examples for improper use of tcin and its control variables:

```

dima%(999,3,1)             Three-dimensional measurement arrays produce a
                                "bad subscript error" message when executing tcin

dim B(99),E&(99,1)         Real, complex and string arrays produce a
dim c$(999,1)              "type mismatch error" message when executing tcin.

dim A%(99):Trigger=200     "illegal quantity error" since the Trigger value is
                                higher than the total number of samples.

dim A%(9,99):Timebase=20   "illegal quantity error" since 100 channels cannot be
                                sampled within 20 µs.

```

tcout

Generates analog signals from data-arrays

tcout is a very powerful macro-command outputting data from an array to one or more DA converter (output-channels) at a fixed clock rate. The array must be of integer type (%-ending) and thus handles 16-bit data per element. If you use a two-dimensional integer array, multi-channel outputting is performed automatically. The number of channels is defined by the value of the second dimension. The number of samples per channel is defined by the value of the first dimension. Arrays with more than two dimensions are not accepted for tcout. The clock rate, the output-mode and the (starting) channel are defined by separate system variables.

The following programming steps have to be done to set up `tcout` properly:

1. Defining a sample array: e.g. `dima%(249,2)` means 3 channels with 250 samples each)
2. Filling the array with previous sampled signals (`tcin`) or calculated data
3. Setting the clock rate: e.g. `Timebase=500` means a clock rate of 20 kHz, or 500µs for each data point
4. Setting the starting channel: e.g. `Channel=2` means first output-channel is channel 2
5. Setting output mode with system variable `Trigger`:
 - `Trigger=-1` continuously output of samples, break condition (<break> button) is checked every end of output-cycle (end of sample array). This will produce a short timedelay of 2 µs at every end of scan. Pressing <break> will not stop program execution.
 - `Trigger=0` single output of sample-data
 - `Trigger=N-1` N times output of sample-data
6. `tcouta%()`: starts the output with the current parameters

Defining a sample array

Use the `dim` command described above to define a sampling array.

Examples

```
100 dima%(999)           :rem 1 channel, 1000 samples
110 dimdata%(499,3)     :rem 4 channels, 500 samples each
```

The number of total samples is limited by the memory available. The number of total samples is calculated as:

$$N = (\text{dimension1} + 1) * (\text{dimension2} + 1)$$

The free memory available can be found out with the command

`N1=fre(0)`

The number of channels (*dimension2*) is limited to 256.

Setting the time base

With the system variable `Timebase` the clock rate is set in the unit of µs. The clock rate is limited by the hardware specifications of the DA converter. The IM systems use a 2-channel parallel-DA-converter with a maximum rate of 500 kHz. Thus, the minimum `Timebase` setting is 2 (µs).

Setting the trigger conditions

`tcout` allows to setup the number of output-scans with the system variable `Trigger`.

Trigger	output mode
-1	continuously
0	single scan
N-1	output-scan N times

Setting the starting channel

When setting up a one-channel output, the system variable `Channel` (note the capital C !) defines the channel the samples have to be converted in an analog signal.

When setting up a multi-channel output, `Channel` defines the channel, the multi-channel output is starting with. The second channel is (`Channel + 1`), etc.

Examples

```
10 dims%(999)           :rem 1 channel, 1000 samples
20 for w = 0 to 999     :rem Loop
30 s%(w)=2000*sin(w*2*PI/1000) :rem s%() is filled with calculated
                           sin-function, 2000 mV peak-peak
40 next                 :rem continue loop
```

```

50 Timebase=10          :rem 10us each sample = 100 kHz
60 Trigger=-1          :rem continuous output mode
70 Channel=1           :rem output channel 1
80 tcouts%()          :rem starts output of sinusoidal
                       signal (frequency = 100 Hz, Vpp = 2
                       V) on channel 1, output could be
                       stopped with <break>
90 dimt%(999,2)        :rem 3 channels each 1000 samples
100 for w = 0 to 999   :rem loop
110 t%(w,0)=2000*sin(w*2*PI/1000) :sin-func. for first channel
120 t%(w,1)=2000*cos(w*2*PI/1000) :cos-func. for second channel
130 t%(w,2)=4*w-2000   :triangle-func. for third channel
140 next               :rem continue loop
150 Timebase=100       :rem 100us each sample = 10 kHz
160 Trigger=999        :rem 1000 outputs
170 Channel=4          :rem output-channels starting at 4
180 tcoutt%()          :rem output t%() on analog channels
                       4 to 6 (and/or 8-Channel-DAC) with
                       10 kHz clock rate (function
                       frequency = 10 Hz)

```

average

Averaging single variables or whole data records

average is normally used for measuring signals which are overlaid by noise. Noise and non-periodical disturbances can be eliminated by over sampling the measured data. This means that the 12-Bit-ANDI-measured-data is sampled several times in a real-array (1 Byte exponent and 4 Bytes mantissa). Each single measurement data is added and subsequently normalized. Before averaging the value of the average-variable (or array) must be set Zero. For higher sample rate the accumulation is done in a faster fixed-point-addition. Therefore it is necessary to normalize the variable-array after measurement (convert into BASIC-conformal floating-point-format).

These three steps (clear variable, accumulation of data, normalizing data) can be processed automatically (AVERAGE1) or step by step (AVERAGE2). AVERAGE1 should be preferred, when the total number of scans is already fixed or the scans should be performed very fast. There is only a short delay of 60µs between the single scans. AVERAGE1 could also be used to accumulate a single variable by averaging an array of dimension zero (**dima(x,0)**). Before every accumulation (array-variable with index 0) **average** tests the logic level at trigger-input (pin13 of analog-in socket) and waits for high condition. After trigger-condition occurred the data is accumulated till the end of the array (last index). With TTL-level driven trigger-pin (pin13) every single scan could be triggered manually. The analog input-channel and the timebase (scanrate) is setting up equal to **tcin**.

The following programming steps have to be done to set up **average** properly:

1. Defining an averaging array: e.g. **dima(249,2)** means 3 channels with 250 samples each
2. Setting the scan rate: e.g. **Timebase=500** means a sample rate of 20 kHz, or 500µs for each data point
3. Setting the starting channel: e.g. **Channel=2** means first input-channel is channel 2
4. Sampling and averaging data with **average**

Defining an averaging array

Use the **dim** command described above to define an averaging array.

Examples

```

100 dima(999)          :rem 1 channel, 1000 samples
110 dimdata(499,3)     :rem 4 channels, 500 samples each

```

The number of total samples is limited by the memory available. The number of total samples is calculated as:

$$N = (\text{dimension1} + 1) * (\text{dimension2} + 1)$$

The free memory available can be found out with the command

N1=fre(0)

The number of channels (*dimension2*) is limited to 256.

Setting the time base

With the system variable **Timebase** the sample rate is set in the unit of μs . The sampling rate is limited by the hardware specifications of the AD converter. The IM systems use a 2-channel parallel-AD-converter with a maximum rate of 500 kHz. Thus, the minimum **Timebase** setting is 190 (μs).

Setting the starting channel

When setting up a one-channel average, the system variable **Channel** (note the capital C !) defines the channel the samples have to be taken from.

When setting up a multi-channel average, **Channel** defines the channel, the multi-channel average is starting with. The second channel is (**Channel** + 1), etc.

Examples

```

10 dima(999)                :rem 1 channel, 1000 samples
20 Timebase=190             :rem sample rate 5.263kHz
30 Channel=2                :rem input channel 2
40 averagea(),100          :rem sets all a()=0
                           :rem accumulates 100-times
                           :rem normalize a()
                           :rem total averaging time = 19sec.
                           :rem (190us*1000*100)
    
```

```

10 dima(3,99)               :rem 4 channels, 100 samples each
20 Timebase=1000           :rem sample rate 1kHz
30 Channel=1                :rem input channels 1-5
40 averagea(),100          :rem averaging channels 1-5 100-
                           times
    
```

```

10 dima(3,0)                :rem 4 channels, single variable
20 Timebase=1000           :rem sample rate 1kHz
30 Channel=1                :rem input channels 1-5
40 averagea(),100          :rem averaging variables at
                           channels 1-5 100-times
    
```

```

10 dima(999)                :rem 1 channel, 1000 samples each
20 Timebase=1000           :rem sample rate 1kHz
30 Channel=0                :rem input channel 0
40 averagea(),0            :rem sets a()=0
                           :rem manual averaging (Average2)
50 if IND[]15 then 50      :rem waits till every 4 LSB of
                           digital-input are low
60 averagea()              :rem tests trigger and accumulates
                           1-times
70 input"One more scan";A$
80 if A$ = "Y" then 50     :rem possibly repeat accumulation
90 average END             :rem normalize a()
                           :rem end of manual averaging
                           (Average2)
    
```

fft

Fast Fourier Transform

The *Fast Fourier Transform* (FFT) is a runtime optimized version of the *Discrete Fourier Transform* (DFT). It transforms time-domain data into the frequency domain. Whereas the archetype, the *Continuous Fourier Transform* deals with infinite, continuous signals, the DFT and the FFT are able to calculate with discontinuous data samples e.g. from digitalized signals within finite integration limits.


```

100 dimf(255)           :rem reserves 256 real variables (for
                        :rem transformation)
110 dima%(9999)        :rem reserves 10000 integer variables
                        :rem a%()
120 timebase=10        :rem Timebase 10 microseconds (100kHz)
130 channel=2          :rem Pre-selection of analog channel 2
140 tcin a%()          :rem performs measurement
150 matcopy a%(200) to f() :rem copies a%() to f() beginning with
                        :rem test point 200
160 fft"hp",f()        :rem FFT with hanning window, result in
                        :rem polar coordinates: intensity and phase
170 print f(0),f(2)    :rem prints DC voltage proportion,
                        :rem intensity an the phase of the
                        :rem fundamental wave. The fundamental wave
                        :rem has a frequency of 1/(256*10 us), that's
                        :rem about 400Hz
180 print f(254),f(255) :rem prints intensity and phase (in
                        :rem rad!) of the last registrable harmonic:
                        :rem frequency = 128/(256*10 us) = 50kHz
    
```

```

100 dimc&(amp;511)       :rem reserves 512 complex-variables c&()
110 call Synthesis     :rem symbolizes synthesis of the
                        :rem function to be transformed
120 fftc&()            :rem direct (forwards) transformation of
                        :rem complex into complex
    
```

Please note that the FFT "has no idea" about the formation of the data to be transformed. Herein, only the order of the DC voltage proportion to the fundamental wave and the harmonic waves -Waves with a frequency that is an integral multiple of the frequency of the fundamental wave-. Only when considering the sampling rate -which can be fixed with timebase for example- that it is possible to know the factual frequencies present.

A further block of keys are the 5 function-keys, which may be used with or without shift, which thus forms 10 functions. All 10 functions can be occupied by the user. For this the following statement sequence is required :

```

print chr$(27) "sx statement sequence"chr$(0)
x = function number (0-9)
    
```

The statement sequence may be up to 80 characters long. It can alternately consist of text (in quotation marks !) and chr\$-instruction (without quotation marks).

The function keys are originally constructed with regularly needed Basic commands. Please read the list "function key assignment" in the appendix.

Appendix

ANDIbasic Statements

Command	Description	Page	SCRIPT & USR relevant
abs()	Absolute value function		*
acos()	Arc-cosine function		*
alist	Lists the values of arrays		
aload	Loads and converts an ASCII-program file		
amerge	Merges an ASCII-saved program file to an already loaded program		
and	Logical AND operator		*
append	Reopens disc files and appends additional values		
asave	Saves a program as an ASCII-file		
asc()	Converts string into ASCII-code		*
asin()	Arc-sine function		*
astart	Loads and starts an ASCII-saved program file		
atn()	Arc-tangent function		*
auto	Automatic numbering of program lines		
average	Signal averaging with EuroAndi		
backup	(reserved)		
begin	(reserved)		
bend	(reserved)		
break	Breaks program execution (without single step mode)		
bsave	(reserved)		
call	(reserved)		
case	(reserved)		
cabs()	Complex absolute value function		*
cacos()	Complex inverse cosine function		*
carg()	Complex argument value function		*
casin()	Complex inverse sine function		*
catalog	Lists disc files		
catn()	Complex inverse tangent function		*
catnh()	Complex inverse area tangent function		*
ccos()	Complex cosine function		*
ccosh()	Complex area cosine function		*
ccot()	Complex cotangent function		*
ccoth()	Complex area cotangent function		*
cexp()	Complex exponentiation function		*
chr\$()	Converts ASCII-code number to string		*
clist	Listing program lines and marks ANDIbasic commands		
clog()	Complex natural logarithm function		*
close	Closes open files		*
clp	Erases part of a program		
clr	Erases a variable		*
cls	Clears stack		*
cmd	Directs screen output to another devices		
collect	(reserved)		
com	Defines variables of first priority level		
cmplx()	Converts real- and imaginary parts to complex expression		*
concat	(reserved)		
conjg()	Complex conjugation function		*
cont	Continues execution of a stopped program		
copy	Copies files		
correlate()	Auto- & cross-correlation function		*

Command	Description	Page	SCRIPT & USR relevant
cos()	Cosine function		*
cosh()	Area cosine function		*
cot()	Cotangent function		*
coth()	Area cotangent function		*
count()	Reads data from the <i>Counter</i> module		
cpol()	Complex conversion function: rectangular to polar data		*
cquad	Returns the quadrant of complex coordinates		*
csin()	Complex sine function		*
csinh()	Complex area sine function		*
cswap()	Swaps real and imaginary part of a complex variable		*
ctan()	Complex tangent function		*
csqr()	Complex square root function		*
ctanh()	Complex area tangent function		*
cpy	Basic-lines copying		
cvec()	Complex conversion function: polar to rectangular data		*
data	Defines data lines		*
dclose	Closes open program files		
dec	Decrements variable contents by one		*
tdeffn	Defines a function		
delete	Deletes ANDIbasic lines		
determ()	Determines a matrix		*
diag()	Matrix diagonal assignment		*
dim	Defines variables of second priority level		*
directory	Lists disc files		
dload	Loads a program file from disc		
dopen	Opens a program file on disc		
dsave	Writes a program file to disc		
dstart	Loads and automatically runs a program file from disc		
dstate	Set the path of directory d2		
dump	Outputs hexadecimal data list from memory		
else	"if not" condition		*
end	Defines the logical end of a program		*
endif	Defines the end of an if-then-else-trueif-falseif block		*
endproc	(reserved)		
equiv	(reserved)		
erf()	Gaussian error integral function		*
exitif	(reserved)		
exp()	Exponential function		*
extremes	Check numeric arrays for minima and maxima values		*
falseif	"if not" condition		*
fetch	(reserved)		
fft	Fast-Fourier-Transform function		*
fn	Calls a function		
fnend	Defines the end of a function		
fnreturn	Returns the results of a function		
for...to	Program loop		*
frac()	Numeric fractional part function		*
fre(0)	Returns the free memory		*
get	Reads single characters from keyboard		
get#	Reads single characters from a peripheral device		
getcount()	Reads definable number of characters from keyboard		
get#count()	Reads definable number of characters from peripheral device		
gosub	Jumps to a subroutine		*
goto	Jumps to another program location		*
hardcopy	Screen dump to a printer (text only)		
header	Formats a disc or harddisc		

Command	Description	Page	SCRIPT & USR relevant
hex("")	Converts a hex number to a real number		*
hex\$()	Converts a decimal number to a hex number		*
if...then	Sets a condition		*
imag	Converts a complex imaginary part in a real expression		*
ina()	Reads an analog channel		
incr	Increments variable contents by one		*
ind()	Reads a digital channel		
init	Initializes disc drives		
ink()	Reads an ANDI channel		
input	Reads a string from keyboard		*
input#	Reads a string from peripheral device		*
instr()	Checks a string for a sequence of characters		*
int()	Converts real expression to integer expression		*
inverf()	Inverse of Gaussian error integral function		*
ld10()	Decimal logarithm function		*
left\$()	Returns the left part of a string		*
len()	Returns the number of characters in a string		*
let	Assigns a value to a variable		
list	Lists the program code on the screen		
load	(reserved)		
load#	Loads a file from disc		*
local	(reserved)		
log()	Natural logarithm function		*
loop	Defines end of "do – loop" block		*
mat	Matrix/array command header		*
mid\$()	Returns a definable part of a string		*
mod	Numeric modulo operator		*
moni	Jumps to ANDIMON system monitor		
move	Moves ANDIbasic lines		
new	Erases a program in memory		
next	Defines the end of a for-next loop		*
norm()	Norm function of a matrix		*
not	Logical inverse		*
oct()	Converts octadecimial string to numeric value		
oct\$()	Converts numeric value to octadecimial string		
off	Switches off trace and single step mode		
on ... goto	Conditional jump list		*
on ... gosub	Conditional jump list to subroutines		*
on error	Jumps on error condition		*
on irq	Jumps on IRQ condition		
on nmi	Jumps on NMI condition		
or	Logical OR		*
open	Opens a file on disc		*
outa	Outputs data to an analog output channel		
outc	Sets clock condition		
outd	Outputs data to a digital output channel		
outk	Outputs to an ANDI channel		
overlay	Overlays program code to a program in memory		
pack	Formats data for output (with <i>using</i>)		*
pause	Pauses the program for a definable period of time		*
peek()	Reads data from a memory location		
plotclr	Erases/fills screen window		
plot"da"	Sets the plot vector to absolute coordinates		
plotdiml	Defines the lower left corner of a screen window		
plotdimu	Defines the upper right corner of a screen window		
plot"dr"	Sets the plot vector to relative coordinates		

Command	Description	Page	SCRIPT & USR relevant
plotinv	Inverts colors of screen graphics		
plotmat	Plots curve from a numerical array		
plotnew	Clears the screen from graphics, initializes graphics		
plotoff	Microcursor off		
ploton	Mircorcursor on		
plot"sk"	Defines origin of plot coordinates		
plot"sy"	Defines plot parameters		
plot"va"	Plots vector to absolute coordinates		
plot"vr"	Plots vector to relative coordinates		
plothardcopy	Screen dump to a printer (graphics only)		
poke	Writes a byte to a memory location		
pos()	Returns cursor coordinates		
print	Prints to screen		*
print#	Prints to peripheral device		
procedure	(reserved)		
psave	Saves program as protected file		
read	Reads <i>data</i> line		*
real()	Converts complex to real		*
record	Sets the pointer for relative data files		
rem	Defines remark text		*
rename	Renames disc file		
renumber	Renumsbers ANDIbasic program lines		
repeat...until	Program loop until condition is reached		*
reset	(reserved)		
restore	Sets the <i>data</i> line pointer		*
return	Defines the logical end of a subroutine		*
right\$()	Returns the left part of a string		*
rnd()	Random number function		*
round()	Returns 5/4 rounded numeric value		*
run	Starts program execution		*
save	(reserved)		
save#	Writes data to disc		*
scratch	Erases files on disc		*
search	Searches for ANDIbasic text		
setusr	Sets <i>usr</i> vector		
setirq	Sets IRQ-level		
sgn()	Sign function		*
sin()	Sine function		*
sinh()	Area sine function		*
spc()	Creates spaces		
sqr()	Square root function		*
start	(reserved)		
step	Defines the step width of a <i>for...next</i> loop		*
stop	Interrupts the execution of a program for single step mode		
str\$()	Converts numerical value to string		*
swap	Swaps the values of two variables		*
switch	(reserved)		
sys	Calls a routine written in machine language		
system	Jumps to System-Menu		
tab()	Tabulator function (horizontal and vertical)		*
tan()	Tangent function		*
tanh()	Area tangent function		*
tcin	Reads arrays of data from analog and digital input channels		
tcout	Outputs a function to analog and digital output channels		
trace	Activates trace mode		
using	Applies a format mask		*

Command	Description	Page	SCRIPT & USR relevant
usr()	Calls a function written in machine language		
usr...()	USR-Functions names		
val()	Returns the numerical equivalent of a string		*
verify	Compares the program in memory with the copy on disc		
vlist	Lists variables names their contents		
wait	Waits for a certain value in a memory location		
while...wend	Conditional loop		*
xor	Logical exclusive OR		*

ANDIbasic System Variables

Variable	Description	Page	SCRIPT & USR relevant
ds	Disc status (error number)		*
ds\$	Disc status (text)		*
dt\$	Date (ddmmyy)		*
Errline	Error line		
Errtype	Error type		
Floppy	Device number for disc commands		*
Printer	Device number for print commands		
Plotter	Device number for plot commands		
st	Device status		
ti	Stop-watch		
ti\$	Time (hhmmss)		*
Timeout	bus timeout definition		

Disk Error Messages (ds, ds\$)

ds	ds\$	avail. for PC	avail. for IM
00	ok		
01	read error		
02	hard disk not formatted		
03	subdirectory not found		
04	write protected file		
05	warning data correction attempted		
06	illegal access		
07	syntax error		
08	catalog full		
09	disk full		
10	bad sector allocation table		
11	file not found		
12	no more data		
13	file locked		
14	file exists		
15	subdirectory exists		
16	-		
17	file(s) scratched		
18	-		
19	-		
20	file(s) copied		
21	no read file		
22	internal diagnostic error		
23	channel yet in use		
24	drive not available		
25	no drive available		
26	write failure		
27	file damaged		
28	file not open		
29	format requested		
30	rootdirectory damaged		
31	subdirectory not empty		
32	subdirectory scratched		
33	subdirectory in use		
34	subdirectory locked		
35	illegal block address		
36	too much data		
37	warning, retries on maindirectory		
38	warning, bad sector allocation table		
39	unit unusable, format requested		
40	unknown subcommand		
41	problems while writing fat		
42	not a write file		
43	warning, retry attempted		
44	too much subdirectories		
45	not a random access file		
46	scsi-subsystem not available		
47	spindle motor stopped by software		
48	cannot read group		
49	-		
50	-		
51	device & partition selected		
62	file not found		
63	file exists		
72	disk full		
74	subdirectory not found		

ANDIbasic Error Messages (Errtype)

#	Message	Possible reason
01	too many files	More than 255 file channels open
02	file open	Not relevant from sys 7.5 on, reopen allowed
03	file not open	File channel requested was not opened
04	file not found	File requested not found on selected I/O device
05	load error	
06	verify error	File in memory not identical with file on selected I/O device
07	device not present	Selected I/O device not found
08	not input file	It was tried to read from a write channel
09	not output file	(reserved)
10	next without for	for-next loop not initiated with for statement
11	syntax error	Violation of the ANDIbasic syntax
12	return without gosub	Program execution found return without finding gosub before
13	out of data	read statement ran out of data
14	illegal quantity	Number out of valid range for this function or variable
15	overflow	Violation of a data format (e.g. integer numbers: 16 bit)
16	out of memory	Memory insufficient
17	undef'd statement	Run, goto or gosub to a not defined label or line number
18	bad subscript	Array index out of defined range
19	redim'd array	Trying to dim an already dim'd array a second time
20	division by zero	Trying to divide by zero
21	illegal direct	Trying to use a program-only statement in direct input mode
22	type mismatch	Trying to calculate with different types of data
23	string too long	String cannot be longer than 65535 characters, input string (input#) cannot be longer than 1024 byte
24	file data	File data do not fit variables to be loaded to
25	formula too complex	Overrun of function stack
26	can't continue	cont statement cannot be executed because of an error, a change in program code or after running into end statement
27	undef'd function	Trying to call an undefined function
28	undim'd array	Trying to use an array which was not dim'd before
29	illegal com	Trying to define com variables after dim definitions
30	else without if	Program execution found else without finding if before
31	bad mask	Invalid using mask (e.g. for exponents)
32	linenumber too big	Line numbers cannot be bigger than 63999
33	until without repeat	Program execution found until without finding repeat before
34	wend without while	Program execution found wend without finding while before
35	protected names	Trying to use a protected word (ANDIbasic statement or system variable) as a variable or label
36	? extra ignored	Comma in input works as a separator
44	timeout	Waiting too long for the answer of an I/O device
54	no program file	Check sum error in file to load (e.g. when trying to load an asave 'd program with dload)
55	no ascii source file	Trying to load a dsave 'd program with aload
56	list impossible	Trying to list a protected file
57	double nmi	When interrupting program execution by pushing NMI switch twice
59	system error	CPU reports bus, address or op-code error
60	missing do	(reserved)
61	matrix	Matrix operands do not conform
62	singular matrix	Matrix is singular