

Basic scripting - Commands

List of all commands

Control Flow

DO/LOOP | FOR/NEXT | WHILE | GOTO | GOSUB | END | IF/THEN | SLEEP

Variables, Arrays and Operators

Operators | DIM | LET | DATA/READ/RESTORE

Math and String Functions

ABS | ASC | CHR\$ | INSTR | LCASE\$ | LEFT\$ | LEN | LTRIM\$ | MID\$ | RIGHT\$ | RTRIM\$ | STR\$ | UCASE\$ | VAL\$ | TAB | CSV\$

I/O Commands

AUXOPEN | INMODE | OUTMODE | INPUT | PORT | PRINT | PUT | GET | CLOSE | KEYS | SETLEDS | HSET

File Operations and NVRAM Access

OPEN | CLOSE | PUT | GET | KILL | SEEK | LOAD | SAVE | MOVE

Networking

BIND | CLOSE | CONNECT | LISTEN | RESOLV | UDPOPEN | PUT | GET | HSET

CAN Bus related

CANCSV\$ | GETCAN | PUTCAN | SETCAN | CANINFO

Informational

BYTESREAD | FREEMEM | CANINFO | KEYS | LOC | LOF | STATUS | TIME\$ | TIME | LASTERR | DATE\$ | MILLIS

Miscellaneous

EXEC | REM

Content

Operators	4
ABS	7
ASC	7
AUXOPEN	8
BIND	10
BYTESREAD	11
CANCSV\$	11
CANINFO	12
CHR\$	13
CLOSE	14
CONNECT	14
CSV\$	16
DATA - READ - RESTORE	16
DATE\$	17
DIM	17
DO ... LOOP ... UNTIL	18
END	19
FLOAT\$	20
FOR ... NEXT ... TO ... STEP	21
FREEMEM	21
GET	22
GETCAN	23
GETSCAN	26
GOSUB ... RETURN	27
GOTO	27
HEX\$	28
HSET	28
IF ... THEN ... ELSE	30
INMODE	30
INPUT	31
INSTR	32
KEYS	32
KILL	33
LASTERR	34
LCASE\$	36
LEFT\$	36
LEN	37
LET	37
LISTEN	38
LOAD	39
LOC	39
LOF	40
LTRIM\$	40
MEMCFG	41
MID\$	42
MILLIS	42

MOVE	43
OPEN	43
OUTMODE	44
PRINT	45
PUT	45
PUTCAN	47
PWM	48
REM	49
RESOLV	49
RIGHT\$	50
RTRIM\$	51
SAVE	51
SCAN	52
SCANNED	53
SEEK	53
SETCAN	54
SETLEDS	55
SLEEP	55
STATUS	56
STR\$	57
TIME\$	58
TIME	59
UCASE\$	59
UDPOPEN	59
VAL	60
WHILE	61

Operators

Bitwise Operators

Bitwise operators operate on 32 bit signed integer values.

The Avisaro Scripting Language supports these four bitwise operators:

AND

A bitwise AND takes two signed integers and performs the logical AND operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 AND the second bit is 1. Otherwise, the result is 0. In the Avisaro Scripting Language, one can use the keyword AND or the abbreviation & to perform an AND operation.

OR

A bitwise OR takes two signed integers and produces another one by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical inclusive OR operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 OR the second bit is 1 (or both), and otherwise the result is 0. In the Avisaro Scripting Language, one can use the keyword OR or the abbreviation | to perform an OR operation.

Exclusive OR

A bitwise exclusive OR takes two signed integers and performs the logical exclusive OR operation on each pair of corresponding bits. The result in each position is 1 if the two bits are different, and 0 if they are the same. In the Avisaro Scripting Language, one can use the keyword EOR or the abbreviation @ to perform an exclusive OR operation.

NOT

The bitwise NOT, or complement, is a unary operation which performs logical negation on each bit, forming the ones' complement of the given binary value. Digits which were 0 become 1, and vice versa. In the Avisaro Scripting Language, one can use the keyword NOT or the abbreviation ~ to perform a NOT operation.

Example

```
outmode -2
let a = 1
print a
let a = a EOR 1
print a
let a = a EOR 1
print a
let a = a OR 2
print a
let a = a AND 2
print a
let a = NOT 48
print a
```

Prints out six lines with the values 1,0,1,3,2 and -49

Arithmetic Operators

The scripting engine supports the usual arithmetic operators Multiplication, Division, Addition, Subtraction and Modulo. All these operators work with 32 bit signed integers, which is the only numeric data type of the scripting engine.

+ (Addition) Performs the standard mathematical process of putting two numbers together.

- (Subtraction) Performs the standard mathematical "inverse addition" of two numbers.

/ (Division) Performs the standard mathematical "inverse multiplication" of two numbers.

* (Multiplication) Performs the standard mathematical operation of adding together multiple copies of the same number.

% (Modulo) -- *Version 3.26 and above.* Gives the remainder of division of one number by another.

Example

```
outmode -2
let a = 13
let b = 4
```

```
print a
print "+"
print b
print "="
print a+b
```

```
print a
print "-"
print b
print "="
print a-b
```

```
print a
print "/"
print b
print "="
print a/b
```

```
print a
print "%"
print b
print "="
print a%b
```

Prints out the four lines:

```
13+4=17
13-4=9
13/4=3
13%4=1
```

Relational Operators

The scripting engine supports a number of relational (or comparison) operators in order to test the relation between two 32 bit signed integer numbers. Relational Operators are normally used in conditional control statements (IF/THEN/ELSE) and iterations (loops).

Supported operators are:

= (equal to) Tests if two values are.

<> (not equal to) Tests if two values are not equal.

> (greater than) Test if the value of the left expression is greater than that of the right.

< (less than) Test if the value of the left expression is less than that of the right.

>= (greater or equal) Test if the value of the left expression is greater than or equal to that of the right.

<= (less or equal) Test if the value of the left expression is less than or equal to that of the right.

Example

```
outmode -2
let a = -5
while a <= 5
  print a
  a = a + 1
wend
end
```

Prints out all numbers between (and including) -5 ... 5

String Concatenation Operator

When using the plus sign + on string literals and variables, it produces a result string that contains both operands linked together.

Example

```
outmode -2
let a$ = "hello "
let b$ = "world"
let c$ = a$ + b$
print c$
end
```

Prints out the content of c\$ which becomes "hello world" after a\$ and b\$ are joined.

ABS

Description

The ABS function computes the absolute value of a 32 bit signed integer value. That is simply the value itself with sign discarded.

Example

```
outmode -2
let a = 3
let b = -3
print abs(a)
print abs(b)
end
```

This prints two times a '3', because both numbers lose their signs.

Remarks

ABS is the BASIC equivalent of $|x|$ in math notation.

ASC

Description

The *ASC* function converts the first character of a string into a numerical value, also known as ASCII value. All characters in the string other than the first one are ignored.

Example

```
outmode -2
print asc ("A")
print asc ("B")
print asc ("C")
```

This example prints the ASCII values of the first three capitals.

Remarks

To convert others than the first character, use the string extraction functions *MID\$*, *LEFT\$* and *RIGHT\$* before.

The inverse function of *ASC* is *CHR\$*. Please see also the *CHR\$* page.

AUXOPEN

Description

AUXOPEN is a generic function that can be used to enable I/O interfaces which normally are disabled. AUXOPEN takes six arguments, opens the interface and sets the LASTERR variable after invocation. The first argument is a pre-defined handle number that refers to the interface. The other five arguments are interface-specific configuration parameters.

Syntax

auxopen <handle number> <Baud Rate> <3. parameter> <4. parameter>

1. Handle number

- 4 : refers to RS232 Port 2 of the Avisaro 2.0
- 5 : opens the I2C interface in master mode
- 8 : opens the secondary CAN interface
- 10 : SPI (as Master) using the SPI lines 9,10,11, and 12

2. Baud Rate/Frequency

Defines the Baud Rate in bit/s or Hz.

3. Other parameter

The other parameter depends on the interface (handle number) selected:

Parameter	2. RS232	I2C Master	2. CAN	SPI Master on Pins 9,10,11,12	future use
1.	-4	-5	-8	-10	
2.	Baud rate bits/s	Baud rate bits/s	Baud rate bits/s	Clock frequency in Hz from 140 kHz up to 18 Mhz	
3.	Parity ("N","O","E")	Bus Address	Filter (lower ID)	Clock polarity, 0 or 1 0 = Clock is active high 1 = Clock is active low	
4.	# of Stop bits (1,2)	0 = master 1 = slave	Filter (upper ID)	Clock phase, 0 or 1 0 = Data is sampled on first clock edge 1 = Data is sampled on second clock edge	
5.	# of Data bits (6, 7, 8)	not used (0)	Mode (0, 1, 2)	Slave Select polarity, 0 or 1 0 = SS is active low 1 = SS is active high	
6.	Flow Cntrl ("N", "H", "S")	not used (0)	not used (0)	Slave Select mode, 0 or 1 0 = SS toggles once per frame 1 = SS toggles for every byte	

AUXOPEN - Using 2nd RS232

Details of the AUXOPEN command when used with RS232/485/422:

1. Handle number

-4 refers to RS232

2. Baud Rate

For all three interfaces, UART#3, IIC master or CAN2, this argument defines the baud rate in bits per second.

3. Parity, Slave Address or CAN Filter argument

This argument defines the type of parity that shall be used. One of the three ASCII values E, O, or N must be provided to set the parity type. E means even, O means odd and N means no parity. You can use the ASC function to convert a character into that value.

4. Stop Bits or CAN Filter argument

This argument defines the number of stop bits for the connection. Allowed values are 1 and 2.

5. Bits per Character or special CAN features

This argument defines character width for the connection. Allowed character widths are 5, 6, 7 and 8.

6. Flow control method

This argument defines the type of handshake for the connection. One of the ASCII values N, H, or S must be provided to set the handshake type. N means no handshake, H means hardware handshake (RTS/CTS) and S means software handshake (XON/XOFF). You can use the ASC function to convert a character into that value

AUXOPEN - Using 2nd CAN

1. Handle number

-8 opens the secondary CAN Interface

2. Baud Rate

For all three interfaces, UART#3, IIC master or CAN2, this argument defines the baud rate in bits per second.

3. Parity, Slave Address or CAN Filter argument

This is the first argument of the message acceptance filter (lower ID).

4. Stop Bits or CAN Filter argument

This is the second message acceptance filter value (upper ID).

5. Bits per Character or special CAN features

This argument can 0, 1 or 2. 0 means normal operation. When 1, the CAN interface uses a proprietary flow control mechanism by sending out certain messages when its input FIFO becomes too full. When 2, the CAN interface operates in "sniffing" mode, that is, it only receives but does not send anything.

6. Flow control method

This value is unused and shall be zero.

AUXOPEN - Using I2C Master or Slave

Details of the AUXOPEN command when used with I2C:

1. Handle number

-5 opens the IIC interface .

2. Baud Rate

This argument defines the baud rate in bits per second. If I2C is configured as slave, enter a 0.

3. Parity, Slave Address or CAN Filter argument

This is the address of the slave where the master should talk to.

4. Stop Bits or CAN Filter argument

This value is not used and shall be zero.

5. Bits per Character or special CAN features
This value is not used and shall be zero.
6. Flow control method
This value is unused and shall be zero.

Example (RS232)

```
outmode -2
let a$ = "hello"
auxopen -4, 115200, ASC("N"), 1, 8, ASC("N")
if LASTERR = 0 then
    print "AUX UART open!"
    put -4, a$
else
    print "failed to open AUX UART"
end if
```

Remarks

Because I/O lines of the Avisaro module are shared among multiple interfaces, enabling an auxiliary port will (in many cases) disable other functionality. Please see the hardware manual for functions that interfere with others. Once activated, an auxiliary port can not be disabled programmatically. If called more than once, `AUXOPEN` opens den auxiliary port again and again. This is roughly the same as resetting the port.

Note on SPI master: SPI master uses 8-bit frames and works like a bi-directional synchronous shift register. Bytes are actively clocked in and out when an output command (such as PUT) is executed. Incoming bytes are stored into a FIFO buffer. This buffer can be read with an input command (such as GET). If e.g. 200 bytes are clocked out, simultaneous incoming 200 bytes are buffered.

BIND

Description

The `BIND` command can be used to bound TCP and UDP sockets explicitly to a specific network interface. This is only useful on configurations that have more than one network interface enabled. `BIND` requires two arguments. The first one is the socket handle and the last one is a number which specifies the network interface. These values are currently defined:

1 Is the Ethernet interface

0 Is the WLAN interface

-1 means "all interfaces". This is the default value for new sockets

Example

The following example puts a socket into listen state and binds it to exclusively to the WLAN interface. This prohibits ethernet clients from connecting to that socket.

```
listen 101,123,0
bind 101, 0
```

Remarks

BIND sets LASTERR to ERR_OK (0) on success or any other value on failure. *BIND* can be used regardless of socket state. A socket can be bound or unbound even it is connected or closed (which makes very less sense). The default binding (-1 = unbound) enables listening sockets to listen on both, and also connecting sockets to send their requests over both interfaces. Same applies to UDP sockets, which work on both interfaces in parallel if binding is removed by calling *BIND* with -1. A socket loses its binding when it is closed. Please see also the [CONNECT](#), [LISTEN](#) and [UDPLISTEN](#) pages.

BYTESREAD

Description

BYTESREAD is a read-only pseudo variable. It primarily exists to let the program know how many bytes just were transferred. A script can only read this variable. Write access is prohibited and treated as error.

Example

The following program reads data from the current I/O protocol until one second elapses. Then it prints out how many bytes it has stored in the variable a\$.

```
outmode -2
inmode -3
let a$ = ""
while 1=1
  sleep (1000)
  input a$
  if BYTESREAD < > 0 then
    print BYTESREAD
  end if
wend
```

Remarks

Why and when *BYTESREAD* changes its value, depends on certain commands and functions of the scripting language.

CANCSV\$

Description

The *CANCSV\$* pseudo variable exists to view the last received CAN frame as comma-separated string. Such CSV strings provide a convenient way for logging CAN messages in the human-readable and exchangeable CSV format. *CANCSV\$* uses the last received CAN frame as input. Therefore, accessing *CANCSV\$* without a previous successful call to *GETCAN* should be avoided. Internally, *CANCSV\$* generates CSV strings of variable length, depending on the number of data bytes. A *CANCSV\$* generated string is build up like this:

1. Millisecond timestamp.
2. Frame type: 11 for standard, 29 for extended
3. Message ID
4. RTR bit, 0 or 1
5. Number of data bytes, 0 to 8.
6. Zero or up to eight data bytes, only the commas for missing bytes

Thus, CSV strings generated by `CANCSV$` always have 12 commas. An example might look like this:

```
144661,29,123,0,5,11,22,33,44,55,,,
```

144661 is the millisecond timestamp.

It's an extended frame.

The Message ID is 123.

There's no RTR bit.

There are 5 data bytes.

The data bytes are 11, 22, 33, 44 and 55.

Example

This little example reads all received CAN frames and prints them out as CSV strings:

```
dim a(28)
do
  getcan a
  if LASTERR = 0 then
    let c$ = CANCSV$
    print c$
  end if
loop
```

Remarks

Please see also the [CSV\\$](#) and [PUTCAN/GETCAN](#) pages

CANINFO

Description

The CANINFO function is a convenient function to extract information from the most recent GETCAN call. After GETCAN put a CAN message into an array, CANINFO can be called to read parts of that array, which would otherwise be a hard job. CANINFO needs a single numeric argument that tells him what to do. Here is a list of all CANINFO arguments:

1 Get the Message ID. CANINFO will give the message ID.

2 Get the frame type. CANINFO returns 0 for standard and 1 for extended frames.

3 Get the RTR bit. CANINFO returns 1 if the RTR bit is set, otherwise 0.

4 Get the number of data bytes. CANINFO will return a number between 0 and 8 inclusive.

5 Get the RTC second timestamp. CANINFO will give the value of the RTC field.

6 Get the millisecond timestamp. CANINFO will return the value of the millisecond timestamp field.

7 Get the arbitrary header value. CANINFO will then return what's stored in the header field.

Example

This example never-ending scans the CAN bus and prints all IDs of incoming messages:

```
dim a(28)
do
  getcan a
  if LASTERR = 0 then
    print "message received from: ";
    print caninfo(1)
  end if
loop
```

Remarks

Never call CANINFO before a frame was read in with GETCAN. Doing so can cause unpredictable behaviour. See also the

CHR\$

Description

CHR\$ is a string function that generates strings from numeric values (a.k.a ASCII values). These string have always a length of 1, since an ASCII value represents exactly one character.

Example

This little example prints the string "ABC" that is constructed from ASCII values

```
outmode -2
let a$ = chr$(65) + chr$(66) + chr$(67)
print a$
```

Remarks

Greater arguments than 255 are allowed, but all bits above bit 7 are truncated. The inverse function of *CHR\$* is *ASC*. Please see also the *ASC* page.

CLOSE

Description

The *CLOSE* command can be used to close files, TCP network connections and UDP channels. *CLOSE* can be called with either zero or one argument. If no argument given, *CLOSE* closes all open files but keep network connections open. If an argument is given, it must be a handle that refers to an open file, TCP connection or UDP channel.

Example

This example opens an existing file, reads some data, prints it and closes the file after that.

```
outmode -2
open "R", 1, "test.txt"
if LASTERR = 0 then
  get 1, a$
  print a$
  close 1
  if LASTERR <> 0 then
    print "close failed !!!"
  end if
else
  print "could not open!"
end if
```

Remarks

CLOSE changes the pseudo-variable *LASTERR*. If everything works as expected, *LASTERR* will be 0 (ERR_OK). Any other value indicates an error. Please see the *LASTERR* page and also the page describing the *OPEN* command.

CONNECT

Description

CONNECT actively opens a TCP connection. It needs four arguments which are described below:

1. A handle number

This can be any number in the range from 100..199. Those values are reserved for TCP connections. If *CONNECT* succeeds, the given handle must be used in subsequent invocations of data transfer commands on this connection.

2. The remote IP address

This is a 32 bit signed integer number that is the IP address of the remote station. To calculate that number from standard dotted notation, use the *RESOLV* function.

3. The remote port number

This is the port where the remote service is listening. Although this is also a 32 bit number, only the lower 16 bits (0...65535) are used.

4. A TX delay value

This value is only used when the connection is used for "streaming mode". While streaming, data is collected until TX delay time-out or the transmit buffer is full.

CONNECT sets *LASTERR* accordingly to:

0 (ERR_OK)

if everything works

4 (ERR_ARGUMENT)

if one of the arguments was wrong

38 (ERR_NOCONN)

if, for any reason, a connection could not be established

39 (ERR_NET_DOWN)

if the network interface is not active

32 (ERR_FILE_OPEN)

if the given handle is already in use

26 (ERR_FILE_EXHAUSTED)

if the system has too few resources

Example

This little program tries to connect to the Avisaro web site on port 80 (HTTP). It uses predefined handle #101 and waits in a loop until the connection is established. Then it prints a message and closes the connection.

```
outmode -2
connect 101, resolve ("http://www.avisaro.com"), 80, 0
print LASTERR
REM wait for connection
while status(101) <> 9
wend
print "CONNECTED!!!"
close 101
```

Remarks

CONNECT is only functional on modules that have a kind of network interface (WLAN or Ethernet). Any open socket must later be closed, using the *CLOSE* command, to free up resources. This must also be done if the remote station already has finished the connection. If the script ends, open sockets are *not* closed automatically.

CSV\$

Description

The *CSV\$* string function can be used, to generated a comma separated string from values, that are elements of a byte array. CSV is a widely accepted text format that is used e.g. for data interchange between different systems. *CSV\$* requires three arguments. The first argument is the name of a byte array that contains contiguous bytes which should be converted to CSV. The second argument is the starting offset where conversion should begin, and the last argument is the last offset, where CSV conversion should end. Both offsets are zero-based.

Example

This sample program creates an array of 100 bytes, sets element #2 to 123 and element #99 to 231. Then it uses *CSV\$* twice to convert different parts into CSV and prints the results.

```
outmode -2
dim a(100)
let a(2) = 123
let c$ = csv$ (a, 0, 4)
print c$
let a(99) = 231
let c$ = csv$ (a, 80, 99)
print c$
```

Remarks

Please keep in mind that strings in the Avisaro Scripting Language must not exceed 255 bytes.

DATA - READ - RESTORE

Description

DATA can be used to store numeric and string constants that can be fetched with the *READ* command.

READ gets those constants one-by-one and puts them into variables. Then, the internal read pointer is advanced. After reading the last item, subsequent *READs* will always read 0 until the internal read pointer is reset with *RESTORE*.

After *RESTORE*, the next *READ* command will read the first item. There are more than one *DATA* statement possible. Multiple *DATA* statements work like a single, big one. That is, all items are concatenated.

Example

```
OUTMODE -2
READ a,b,c$,d,e,f,g
PRINT a
PRINT b
PRINT c$
PRINT d
PRINT e
PRINT f
PRINT g
RESTORE
```

```
READ b
PRINT b
DATA 10,20,"just a string",30,40
```

The READ statement fills 7 Variables from 5 stored items. The first 5 variables are assigned to the items found in the DATA line in the same order.

Next two variables (f and g) become 0, because there are no more items in the list. After RESTORE executes, the following READ statement reads the first DATA item again.

Remarks

See also: MEMCFG command to resize memory areas.

Attention:

Due to a bug in the scripting engine, DATA-READ-RESTORE is not fully functional before firmware version 3.24. The module might crash if a program reads more items than the sum of all DATA statement contains. This problem does not exist in version 3.24 and above.

DATE\$

Description

DATE\$ is a pseudo-variable that can be queried to get the current date as string. The returned string contains year, month and day separated by slashes:

YYYY:MM:DD

DATE\$ can only be read. Any write attempt is prohibited and ignored or rejected by the compiler.

Example

Simply prints the current date:

```
outmode -2
print DATE$
end
```

Remarks

There's no function to set the date from a BASIC program. Use the command line interface if you want to set a new time. See also the [TIME\\$](#) page.

DIM

Description

DIM is the standard BASIC keyword to create arrays. Arrays are an ordered set of one ore more variables of the same type. In the Avisaro Scripting Language, the default element type is byte, which is an unsigned 8 bit value. Thus, a DIM instruction without extension creates byte arrays. If the DIM statement is extended with *AS INTEGER*, an array is created which elements are 32 bit signed integers.

Example

The following example creates one byte- and one integer array, sets some value and prints them out.

```
outmode -2
dim a(10)
dim b(10) as integer
let a(0) = 255
let a(1) = 256
let b(0) = 255
let b(1) = 256
print a(0)
print a(1)
print b(0)
print b(1)
```

Because *a* is a byte array, the instruction *let a(1) = 256* wraps around to zero. This differs from *let b(1) = 256*, since *b* is a 32 bit signed integer array that can hold much bigger values.

Remarks

Please note that memory is limited when using arrays. See also the [MEMCFG](#) command.

DO ... LOOP ... UNTIL

Description

The *DO...LOOP* statement can be used in two ways. You have the option to form uncontrolled (endless) loops if the *LOOP* statement has no following *UNTIL*. In this case, the enclosing block is executed over and over. To get out of an endless loop use the [GOTO](#) statement. If there's an *UNTIL* immediately after the *LOOP* statement, you have a condition-controlled loop that runs at least once, because the condition is checked at the end.

Example

The following program is an endless loop. It prints out the string "hello" multiple times, until the program is stopped by an external event or the module is powered off.

```
outmode -2
do
    print "hello"
loop
```

In the next example there's a condition at the end, that causes the loop to run only ten times.

```
outmode -2
let a = 0
do
    print "hello"
    let a = a + 1
loop until a = 10
```

Remarks

In some cases, head-controlled loops are more suitable than *DO...WHILE* loops. Please see also the [WHILE](#) and [FOR](#) pages.

END

Description

END is the standard BASIC instruction to terminate programs. Also in the Avisaro Scripting Language, *END* causes the program to stop immediately.

Example

This program demonstrates the effect of *END*:

```
print "Hello World"
end
print "This line will never be printed"
```

Remarks

The task that runs the BASIC VM inside an RTOS is terminated when a program ends. *END* has the same effect as the program has executed its last line.

EXEC

Description

The EXEC command can be used to send strings to the Command Machine for execution. This enables a BASIC program to do things that are not possible by using the Scripting Language only. The EXEC command needs a single string that is submitted to the command machine during runtime. The Command Machine doesn't distinguish between 'normal' commands and those that came from a BASIC program. Therefore, textual output of the command is always send over the current I/O interface.

Example

This program instructs the Command Machine to show the top level directory of the current mass storage device:

```
outmode -2
exec "dir"
```

Remarks

Prior to firmware version 3.26, there is a conflict with the 'inmode' command:

If inmode is set to -2 or -3 all inputs are redirected to BASIC. Thus, a command issued via exec command is not executed, but also redirected. Example: inmode -2 ; exec "dir logs" does not work, since the "dir logs" text is send to Basic. Workaround: place an inmode 0 command before the exec command and after the exec set the original inmode.

FLOAT\$

Description

FLOAT\$ is a string function that can be used to convert standard IEEE 754 single precision (32 bits) floating point numbers into human-readable form. Any four consecutive bytes of a source array (which must be a byte array) can be converted. *FLOAT\$* requires four arguments as shown below:

1. Name of the source array.
2. Zero-based offset to the first byte of the floating point number.
3. Precision. Length of the fractional portion after the decimal point.
4. Mode. Set to 0 if source bytes are in little endian order, set to 1 if they are big endian.

Example

This program shows *FLOAT\$* in action:

```
outmode -2

REM contains 12.34567 in normal and reverse order
dim a(8)
a(0) = 221
a(1) = 135
a(2) = 69
a(3) = 65
a(4) = 65
a(5) = 69
a(6) = 135
a(7) = 221

REM little endian offset 0
let f$ = float$ (a, 0, 6, 0)
print f$

REM big endian offset 4
let f$ = float$ (a, 4, 6, 1)
print f$
```

Remarks

FLOAT\$ new since version 4.58. *FLOAT\$* does not support exponential notation. If conversion cannot be performed e.g. if some argument is invalid, *FLOAT\$* sets *LASTERR* to *ERR_REJECTED* (12) or *ERR_ARGUMENT* (4). On success, *LASTERR* is set to *ERR_OK* (0).

FOR ... NEXT ... TO ... STEP

Description

The FOR...NEXT loop is a standard BASIC head-controlled loop. When the number of repetitions is known in advance, a FOR...NEXT loop should generally be preferred. FOR...NEXT increments a controlling variable by one, until the value given by TO is exceeded. If the optional keyword STEP exists, the value after STEP is added to the controlling variable instead of one. If this value is less than zero, the controlling variable is counted down.

Example

The following program shows a simple FOR..NEXT loop that prints out the numbers from 0 to 10

```
outmode -2
for i=0 to 10
  print i
next
```

The next program shows the opposite, that is, counting down from 10 to 0

```
outmode -2
for i=10 to 0 step -1
  print i
next
```

Remarks

In some situations when FOR...NEXT does not fit you needs, you can use [DO...LOOP](#) or [WHILE...WEND](#) constructs.

FREEMEM

Description

The Scripting Language's Virtual Machine manages a heap to store temporary and static objects. With *FREEMEM*, a pseudo variable, the program can ask the VM how many bytes are available to generate new objects like strings, arrays and so on. *FREEMEM* is read-only. An attempt to write to *FREEMEM* will be rejected by the compiler.

Example

The following program prints out how many bytes are allocated by both of two 1000-bytes arrays:

```
outmode -2
let a = FREEMEM
dim b(1000)
let a = a - FREEMEM
print a
let a = FREEMEM
dim c(1000)
let a = a - FREEMEM
print a
```

Remarks

Please notice that the VM always needs some heap memory for itself. If you create big arrays that occupy all heap space, the program might terminate with an out-of-memory error. See also the [MEMCFG](#) command.

GET

Description

The *GET* command can be used to read data from a file, network connection or I/O interface. *GET* needs two arguments. The first argument is a handle number that designates the source and the second one is the target variable, which should be filled by *GET*. The maximum number of bytes that will be read with one call depends on the size of the target variable. For a 32 bit signed integer, *GET* reads up to four bytes. For a 1000 bytes byte-array, *get* will read up to 1000 bytes.

GET knows the following sources:

- (0...100)** File handles. Any file that is open for reading.
- (101...200)** TCP connections. Any established TCP connections.
- (201...300)** UDP channels. Any UDP channel that is open.
- (-3)** The current selected I/O interface. New since version 3.36
- (-4)** The auxiliary RS232 port.
- (-5)** The auxiliary IIC port using stop conditions.
- (-6)** The auxiliary IIC port but without using stop conditions.
- (-7)** The primary CAN interface (complete frames). The module must have CAN as I/O protocol enabled.
- (-8)** The auxiliary CAN interface. The program must enable this port with [AUXOPEN](#)
- (-100...-105)** An edit control of the web page.
- (-201...-224)** Digital I/O lines. New since version 3.48
- (-301...-324)** Analog I/O lines. New since version 3.51

GET can use the following targets:

A complete array, either byte- or integer-based

A single element of a byte- or integer array

A string variable

A 32 bit signed integer variable

Example

The following example opens an UDP channel. Incoming data is transferred into the array *a* which then is printed to the console:

```
DIM a(500)
LET x = RESOLV ("255.255.255.255")
UDPOPEN 201, x, 25, 25, 5, 0
DO
  GET 201, a
  IF BYTESREAD <> 0 THEN
    FOR n=0 TO BYTESREAD
      PRINT a(n)
    NEXT
  END IF
LOOP
```

Remarks

GET sets the *LASTERR* and *BYTESREAD* pseudo variables to let the program know what happened. If *GET* encounters an error, *LASTERR* will become an error value other than 0 (ERR_OK). After *GET* returns and *LASTERR* is 0, *BYTESREAD* contains the number of bytes that were actually transferred. This can be a value in the range from 0 (nothing transferred) to the size of the target variable (all requested bytes have arrived).

Handles -4, -5 and -6 are available only when the according I/O interface was activated with *AUXOPEN*. For handles -201...-224, pin functions must be properly set by using the command interface's PORT command.

Digital and Analog I/O ports can only be read into signed integer variables.
See also the *PUT* command.

GETCAN

Description

GETCAN enables a program to read CAN (Controller Area Network) frames from the CAN interface into byte arrays. The CAN interface must be enabled for this to work. The array must have at least 28 bytes because CAN frames handled by the module have an extended header that contains e.g. time stamps. *GETCAN* needs a single argument which is the name of the target array. If this argument is not an array or an array which is smaller than 28 bytes, *GETCAN* rejects the call and sets *LASTERR* to ERR_ARGUMENT(4).

A CAN frame read by *GETCAN* consists of:

4 bytes arbitrary header This field is not affected by the *GETCAN* command but must be there for compatibility reasons.

4 bytes frame information

This field contains information that describes the CAN frame e.g. its type and if it has the RTR bit set.

4 bytes message ID

This field contains the message ID of the frame

8 bytes payload

This field contains the data bytes of the CAN frame

4 bytes Timestamp #1

This field contains the second timestamp based on the RTC of the module.

4 bytes Timestamp #2

This field contains the value of the free running missecond counter of the module.

Example

This example repeatedly tries to read CAN frames. The first time it catches one, it prints a message and exits.

```
DIM a(28)
DO
  GETCAN a
  IF LASTERR = 0 THEN
    PRINT "frame received"
  END IF
LOOP
```

Remarks

LASTERR becomes *ERR_OK(0)* if a frame has been successfully fetched from FIFO buffer. If there's no frame *LASTERR* becomes *ERR_NO_DATA(8)*. If *LASTERR* is 0, that is when *GETCAN* has fetched a frame, *BYTESREAD* is set to the number of payload bytes in that frame. Regarding to this, if both, *LASTERR* and *BYTESREAD* are zero, *GETCAN* has fetched a frame with no payload. There are convenient functions like *CANINFO* and *SETCAN* to dissect and construct CAN frames. Please read also their manual pages.

CAN Data Format

CAN messages are mapped into a 28 byte long array:

Byte	Bits	Description
1	7..0	User definable header bytes. Usually all '0'.
2	15...8	
3	23..16	
4	31..23	
5	7..0	Frame descriptor: reserved bits, all '0'
6	7..0	Frame descriptor: reserved bits, all '0'
7	7..4	Frame descriptor: reserved bits
	3..0	Frame descriptor: CAN data length
8	7	Frame descriptor: Frame type (0 = standard, 1 = extended)
	6	Frame descriptor: RTR bit ('1' = RTR bit is set)
	5..0	Frame descriptor: reserved bits
9	7...0	Message ID
10	15...8	Standard message: valid bits 10..0
11	23..16	Extended message: valid bits 28..0
12	31..23	
13	7..0	CAN data bytes in the order 1 ...8
14	7..0	Valid number of bytes defined by CAN data length
15	7..0	
16	7..0	
17	7..0	
18	7..0	
19	7..0	
20	7..0	
21	7...0	
22	15...8	Millisecond time stamp. Value in ms since module power on.
23	23..16	
24	31..23	
25	7...0	
26	15...8	
27	23..16	
28	31..23	

GETSCAN

Description

The *GETSCAN* command can be used to pull one record off the result set generated by a previous [WLAN SCAN](#). *GETSCAN* needs one argument that must be a string variable which should receive the record.

GETSCAN must be called repeatedly to read out all records. After the last record has been read, *GETSCAN* automatically frees the result set.

After a successful *GETSCAN* call, the string variable contains space-separated information in the following order:

Position 0...11: These characters are the BSSID

Position 13...14: This is a two-digits decimal number that is the RSSI

Position 16: Can be either B or I. B stands for BSS, which is a net that normally needs an access point. I stands for IBSS, which means an ad-hoc network.

Position 18: Can be either 0,1,2 or 3. 0 means no encryption, 1 stands for WEP encryption, 2 is WPA and 3 is WPA2 encryption.

Position 20...53: These characters contain the SSID which is a variable-length string from up to 32 characters. If the station doesn't broadcast its SSID, the string [no name] is inserted.

Example

```
let a$ = ""
getscan a$
print a$
```

This prints a string, e.g:

```
000c419d2f64 43 B 1 Toshi ba_AP
```

which contains a full scan record.

Remarks

This command is new since Version 3.49.

There's no chance to read a record twice because *GETSCAN* advances its internal read pointer after each call and throws the result set away when the last record was fetched. *LASTERR* becomes `ERR_NO_DATA` (8) if *GETSCAN* is called and there's no result set available.

See also

[SCAN](#) (Search for WLAN networks), [SCANNED](#) (Returns number of found WLAN networks), [GETSCAN](#) (Read results from scan command), [STATUS\(-4\)](#) (Returns status of WLAN connection)

GOSUB ... RETURN

Description

GOSUB is the standard BASIC statement to call subroutines. Subroutines are parts of the program, that can be called to perform specific tasks. The *GOSUB* statement must be followed by a label name, which refers to the beginning of the subroutine. All subroutines must begin with a label and must have at least one *RETURN* statement. *RETURN* causes the subroutine to exit and program flow continues right after the *GOSUB* call.

Example

The following program prints twice the string "hello from subroutine".

```
outmode -2
print "hello from main program"
gosub sub
print "hello again"
gosub sub
end

sub:
print "hello from subroutine"
return
```

Remarks

Subroutines can be nested, that means, subroutines can call other subroutines.

GOTO

Description

The *GOTO* statement is BASIC's unconditional jump instruction. *GOTO*, followed by a label name, jumps to that label and execution of the program continues from the first statement after the label.

Example

This little example demonstrates unconditional jumps with *GOTO* statements

```
outmode -2
goto lab1
lab2:
print "world"
end
lab1:
print "hello"
goto lab2
```

Remarks

In many programming languages, although they have *GOTO*, unconditional jumps are considered harmful. This might be true for very large programs, but the Avisaro Scripting Language is for relatively small programs. So, it's unlikely that you get into trouble by using *GOTO*.

HEX\$

Description

HEX\$ is a function that generates strings which are a hexadecimal (a radix16 numeral system) representation of the input value.

HEX\$ requires two arguments. The first one is a number (constant or variable), which should be converted into a hex string. The other argument is the number of hexadecimal digits, that is, the length of the output string. This number can be any value between 1 and 8 (inclusive).

If the second argument demands more digits than the raw conversion would produce, the output is filled up with leading zeros. On the other hand, if the second argument demands fewer digits than the conversion would produce, the output is truncated to keep only the low-order bytes. If the second argument is out of range, the output is forced to be always 8 digits in length.

Example

This little example prints the hexadecimal value of the number 255 as a four-digit hex string. The output will be 00ff

```
outmode -2
let a = 255
print hex$(a, 4)
end
```

Remarks

The output contains only hex digits. There's no leading "0x" as there is in C-style programming languages.

HSET

Description

The *HSET* command exists to change various settings of open handles (e.g. files, sockets) at runtime. *HSET* requires three arguments. The first one is a handle number that must refer to an open resource, a file or a socket. The third argument is called the "option number" that is, which aspect of the given handle *HSET* should modify. The third and last argument is the new value to be set. These options are currently defined:

0 - Set maximum number of packets that socket's receive list can accumulate.

1 - Set maximum number of packets that socket's transmit list can accumulate.

In both cases, the third argument is the number of packets for this option. This value can be any number from 0 to 2147483647. Although, there are fewer packet buffers in the system (see *SSTAT* command output), any greater value permits a socket to consume all packet buffers. On the other hand, a value of 0 completely disables the specified direction.

The new values come into effect immediately if the specific list currently holds exactly the same or fewer packets. If any socket list has more packets before a new value is applied, no more packets are added to the list until it reaches the defined level.

What happens with receive functions if RX list maximum is set to zero:

- A program reading from that UDP socket will not receive anything.
- Incoming packets are silently thrown away.
- A program reading from that TCP socket will not receive anything.
- Incoming packets are thrown away but acknowledged to keep the sender happy.
- *BYTESREAD* will always be zero.
- *LASTERR* will always be *ERR_NO_DATA*.

What happens with transmit functions if TX list maximum is set to zero:

- An attempt trying to transmitting over that UDP socket will fail.
- An attempt trying to transmitting over that TCP socket will fail.
- *LASTERR* will always be *ERR_FR_DENIED*.

What happens with receive functions if RX list reaches maximum:

- Received UDP packets are thrown away until the program frees space by reading packets from the socket.
- Received TCP packets are thrown away and not acknowledged until the program frees space by reading packets from the socket.
- The remote sender must (and will) retransmit lost packets.

What happens with transmit functions if TX list reaches maximum:

- The program will not be able to send over that UDP socket and *LASTERR* will be *ERR_FR_DENIED* until the network task has sent at least one packet.
- The program will not be able to send over that TCP socket and *LASTERR* will be *ERR_FR_DENIED* until the network task has sent at least one packet and got an ACK from the remote station.

Example

Set max. sizes of RX list on socket handle #198 to 0 and TX list to 3

```
HSET 198, 0, 0
HSET 198, 1, 3
```

Remarks

HSET is a generic command that will be extended in the future. Currently, *HSET* is only able to modify the number of packet buffers that a socket is allowed to keep in its receive and transmit lists.

IF ... THEN ... ELSE

Description

For condition testing, the Scripting Language defines the *IF ... THEN* statement. If the condition evaluates to true, code that follows the *IF...THEN* statement is executed, otherwise not. If there is an optional *ELSE*, code after that *ELSE* is executed when the condition evaluates to false. There must be an *END IF* at the end of every *IF...THEN* block.

Example

The following program prints the relationship to 5 of the numbers from 0 to 10:

```
outmode -2
for n = 0 to 10
  print n;
  if n < 5 then
    print " is less than 5"
  else
    print " is greater or equal to 5"
  end if
next
```

Remarks

IF...THEN can be nested. That is, code inside *IF...THEN* and *ELSE* blocks can contain other *IF...THEN* and *ELSE* blocks.

INMODE

Description

The *INMODE* command selects the input mode of the scripting language. The input mode tells the *INPUT* command where to get data from.

There currently defined input modes are:

-1: No input. The *INPUT* command is completely disabled

-2: Synchronous input from the current I/O interface. That means, *INPUT* blocks until all requested characters or a carriage return are received. e.g. for a string variable, *INPUT* returns when 256 characters or an CR arrived.

-3: Asynchronous input from the current I/O interface. In this case, *INPUT* doesn't wait. It returns until the requested number of characters have been read or there's no more data, whichever comes first.

Any valid and open file handle: *INPUT* then reads data from an open file, until the requested number of characters have been read, or the read pointer is at the end, whichever comes first.

Example

The following example repeats all input until the user enters "stop":

```
outmode -2
inmode -2
do
  input c$
  if c$ = "stop" then
    end
  end if
  print "you entered ";
  print c$
loop
```

Remarks

If *INMODE* is -2 or -3, the Command Execution Machine will be suspended until the BASIC program ends or *INMODE* switches to another mode. This must be done because the CMD machine would otherwise suck off all input. See also the OUTM

INPUT

Description

The INPUT command reads data from either the current I/O interface or a file into variables, depending on the mode (see INMODE). INPUT can handle strings and 32 bit signed integer variables. If input succeeds, the pseudo variable LASTERR is set to 0 (ERR_OK) and BYTESREAD contains the number of bytes actually read.

Example

The following example repeats all input until the user enters "stop":

```
outmode -2
inmode -2
do
  input c$
  if c$ = "stop" then
    end
  end if
  print "you entered ";
  print c$
loop
```

Remarks

If there's no active input channel, that is, INMODE was called with -1, LASTERR and BYTESREAD are both zero.

INSTR

Description

The *INSTR* function finds substrings that are parts of a source string. If the source string contains that substring, *INSTR* returns a 1-based offset. The return value of *INSTR* is 0, if the substring could not be found.

Example

```
outmode -2
let a$ = "helloWorld"
let b$ = "wo"
let i = instr (a$, b$)
print i
end
```

The output is 6 since "wo" is found at the sixth position of a\$

Remarks

Internally the C-library function strstr is used.

KEYS

Description

KEYS is a pseudo variable which can be used to read the state of various digital input lines. *KEYS* is read-only. Any attempt to write *KEYS* will be rejected by the compiler. Each bit of *KEYS* belongs to one input line. Currently, these bits are defined:

BIT 0 Port P1.17 on the LPC2366. This pin is used for the external key. This signal is inverted by the firmware: Pulling the key pin to ground causes the *KEYS* command to issue a one.

BIT 1 The CTS input of the RS232 interface which is P2.2 on the LPC2366, if RS232 flow control is *not* set to RTS/CTS. Otherwise this bit is always 0.

BIT 2 The DSR input of the RS232 interface which is P2.4 on the LPC2366.

BIT 3 The DCD input of the RS232 interface which is P2.3 on the LPC2366

New since version 3.48: The following inputs can also be read with the *KEYS* pseudo variable. But this only works if that pin is configured as input. Please see the *PORT* command.

BIT4 Pin 2 on the base module

BIT5 Pin 3 on the base module

BIT6 Pin 4 on the base module

BIT7 Pin 5 on the base module

BIT8 Pin 6 on the base module

BIT9 Pin 7 on the base module

BIT10 Pin 8 on the base module

BIT11 Pin 9 on the base module

BIT12 Pin **10** on the base module

BIT13 Pin **11** on the base module

BIT14 Pin **12** on the base module

BIT15 Pin **15** on the base module

BIT16 Pin **16** on the base module

Example

This program reads the external key in an endless loop and shows if it is pressed:

```
outmode -2
do
  if KEYS and 1 then
    print "pressed"
    while KEYS and 1
      wend
    end if
  loop
```

Remarks

To extract single bits, you can use the AND operator.

KILL

Description

KILL can be used to delete files from the SD card. The file must not be in use for this command to succeed. KILL takes a single argument which is the name of the file that should be deleted.

Example

Deletes the file "hello.txt" from disk:

```
OUTMODE -2
KILL "hello.txt"
IF LASTERR = 0 THEN
  PRINT "the file was successfully deleted"
ELSE
  PRINT "something's gone wrong ";
  PRINT LASTERR
END IF
```

Remarks

KILL sets the pseudo variable LASTERR to report success or failure.

LASTERR

Description

LASTERR is a read-only pseudo variable. Any write attempt results in an error. *LASTERR* is somewhat like a volatile status value that is used by many functions to report success or error. Most functions and commands set *LASTERR* to 0 (ERR_OK) on entry. When the Scripting Language detects inconsistencies, *LASTERR* is set to a value other than 0 to report the problem to the program.

The system defines the following error codes:

Name	Value	Description
ERR_OK	0	Everything works fine
ERR_NO_COMMAND	1	The input was not a known command
ERR_NO_FRAME	2	Packet Interface only: Wrong frame format
ERR_PARAMCOUNT	3	Too much or too less arguments for that command
ERR_ARGUMENT	4	One of the arguments was wrong
ERR_LENGTH	5	The argument has a wrong length
ERR_CRC	6	Packet Interface only: CRC error on incoming packet
ERR_UNSPEC	7	The command or argument is not yet specified
ERR_NO_DATA	8	There's currently no data
ERR_NO_DISK	9	The SD card is missing
ERR_INVALID_HANDLE	10	Handle number out of permitted range
ERR_TRUNCATED	11	The data was truncated
ERR_REJECTED	12	Command or argument currently not valid
ERR_FR_NOT_READY	13	The file system is not yet initialized
ERR_FR_NO_FILE	14	File does not exist
ERR_FR_NO_PATH	15	Path does not exist
ERR_FR_INVALID_NAME	16	The file name is invalid

ERR_FR_INVALID_DRIVE	17	A drive parameter was not recognized
ERR_FR_DENIED	18	Access is denied
ERR_FR_EXIST	19	File or directory already exists
ERR_FR_RW_ERROR	20	Low level error while trying to access the disk
ERR_FR_WRITE_PROTECTED	21	The disk is write protected
ERR_FR_NOT_ENABLED	22	File system not mounted
ERR_FR_NO_FILESYSTEM	23	There's no file system on the disk
ERR_FR_INVALID_OBJECT	24	Internal FAT error
ERR_FS_UNKNOWN	25	The file system could not be recognized
ERR_FIL_EXHAUSTED	26	All file handles are in use
ERR_ID_USED	27	This file handle or other object is already in use
ERR_NOT_OPEN	28	The file or other object is not open
ERR_NO_READ	29	Read access denied
ERR_NO_WRITE	30	Write access denied
ERR_TOO_MUCH	31	Too much data
ERR_FILE_OPEN	32	The file or other object is already open
ERR_EOF	33	File pointer is at the end
ERR_DISK_FULL	34	The disk is full
ERR_FW_IMAGE	35	The firmware was rejected
ERR_ALREADY_RUNNING	36	A script is already running
ERR_NOT_RUNNING	37	The script is not running
ERR_NOCONN	38	There's no connection, The connection is gone
ERR_NET_DOWN	39	The network connection is broken

Example

This program demonstrates *LASTERR*. On startup, *LASTERR* is always 0 (ERR_OK). Variable assignments don't affect *LASTERR*, so the first program line reads *LASTERR* and keeps its old value. In the 4.th line, there's a faulty instruction, which sets *LASTERR* to 4 (ERR_ARGUMENT). The *PRINT* statement after that resets *LASTERR* to 0 again.

```
let a = LASTERR
outmode -2
print a
open "x", 999, "... "
let b = LASTERR
print b
print LASTERR
end
```

Remarks

Because *LASTERR* is only valid immediatly after command execution, it needs immediate evaluation or must be stored into a variable for

LCASE\$

Description

The *LCASE\$* function is a string function that returns another string where all upper case characters are changed into lower case characters. The original string remains unchanged.

Example

The following example demonstrates this:

```
let a$ = "HeLI0 WoRlD"
let b$ = lcase$(a$)
print a$
print b$
```

Remarks

Internally, the C library function "tolower" is used". See also [UCASE\\$](#).

LEFT\$

Description

LEFT\$ is a string functions, that extracts a number of characters from the left of a string and generates a new string. *LEFT\$* needs two arguments, the source string and how many character should be extracted.

Example

Extracts "hello" from "hello world" and prints source and new string

```
outmode -2
a$ = "hel lo world"
b$ = left$ (a$, 5)
print a$
print b$
end
```

Remarks

See also [RIGHT\\$](#) and [MID\\$](#)

LEN

Description

LEN is a function that returns the count of characters of a string. *LEN* needs only a single argument that is the string which characters should be counted.

Example

Printing out the length of "Hello"

```
outmode -2
let a$ = "hel lo"
print len (a$)
end
```

Remarks

Internally, the C-library function "strlen" is used.

LET

Description

The *LET* keyword is used to introduce new variables or to assign new values to existing ones. A variable generated with *LET* always has an initial value because every *LET* must be followed by the assignment operator. *LET* is mandatory for numeric and string variables. To generate arrays use the *DIM* statement.

Example

The following code snippet, which is meant to be the top of a program, introduces and initializes two new variables *a\$* and *b*. In the third line, the existing variable *a\$* gets a new value.

```
LET a$ = "hel lo"
LET b = 1
LET a$ = "worl d"
REM and so on ...
```

Remarks

With few exceptions, e.g. the *READ* statement, new variables must be generated with *LET*. If you omit *LET* and write only "x=y", the compilation is cancelled, program execution fails or the program behaves unpredictable.

LISTEN

Description

The *LISTEN* command passively opens TCP connections, that is, it allocates a socket and switches the socket into listen mode. A client might then connect to that socket and communication can take place. *LISTEN* needs three arguments. The first one must be an arbitrary number in the range from 101 to 200, which is used as the handle for this connection. The second argument is the port number on that the socket should listen for connection requests. The last one is a time-out value used only for streaming mode. Programs that don't use streaming can set the timeout to any value.

Example

The following example waits for a connection on port 123. If a client connects, it sends a little message and closes the connection.

```
outmode -2
listen 101, 123, 0
if LASTERR <> 0 then
  print "can't listen"
end
end if
print "waiting for connection...";
while status(101) <> 9
  sleep 50
wend
print "connected!"
let a$ = "hello"
put 101, a$
sleep 1000
close 101
print "disconnected"
```

Remarks

LISTEN changes the *LASTERR* pseudo-variable. If *LASTERR* is not zero after *LISTEN*, an error occurred. A socket must always be closed with the *CLOSE* command if it's no more in use. This must also be done if the remote station has closed the TCP connection. *CLOSE* puts the socket into the pool of free sockets, so it can be reused. See also the *CONNECT* command.

LOAD

Description

The LOAD command transfers data from internal non-volatile memory into a target variable. This differs from the standard BASIC LOAD command which loads programs into memory. This Scripting Language's LOAD command behaves similar to the GET command, except that the source is always internal NVRAM. LOAD needs two arguments. The first one is the zero-based memory address. The second one is the variable that should receive the data.

The total available NVRAM space that a program can use is 4096 bytes. Addresses above 4095 will wrap around to $x \bmod 4096$.

These types of variables can be filled by LOAD:

- - Single 32 bit signed integer variables
- - Entire byte and integer arrays
- - Single array elements of byte and integer arrays
- - String variables

Example

This example first stores a string at address 2000, then reads that string into another string variable and print both strings.

```
outmode -2
let a$ = "hello"
save 2000, a$
load 2000, b$
print a$ print b$
```

Remarks

The total available NVRAM space that a program can use is 4096 bytes. Addresses above 4095 will wrap around to $x \bmod 4096$.

See also: [SAVE](#) command.

LOC

Description

LOC is a function that serves three purposes. The key purpose is to determine the position of the file pointer of an open file. In this case, *LOC* must be called with the handle number of that file. The file must be open for reading or writing. To query the number of used Kbytes on disk, *LOC* must be called with 0 as argument. Whether *LOC* was called with 0 or a handle number, if there's no disk inserted, *LOC* returns 0 and LASTERR is set to 9 (ERR_NO_DISK).

Example

This example prints the number of used Kbytes on disk:

```
outmode -2 let a = loc(0)
let e = LASTERR
```

```

if e = 0 then
  print "used space: ";
  print a
else
  print "error: ";
  print e
end if

```

Remarks

Since the Scripting Language works with 32 bit signed integers, which maximum positive value is 2,147,483,647, *LOC* might return negative or wrong values on files bigger than that. See also the [LOF](#) page

LOF

Description

LOF is a function that can be used to determine the size of files.

The input to *LOF* must be a handle of an open file or the reserved value 0. If 0 is given, *LOF* returns the media size in Kbytes.

Example

This program checks if disk is present. If so, it prints the size of the disk.

```

outmode -2
let a = lof(0)

if LASTERR <> 0 then
  print "no disk"
end
end if

print "the disk size is: ";
print a

end

```

Remarks

Because the return value of *LOF* is a signed 32 bit integer in the range from -2,147,483,648 to +2,147,483,647, very large disks may produce negative or incorrect results. See also the page describing the [LOC](#) function.

LTRIM\$

Description

LTRIM\$ can be used to remove all space characters from the beginning of a string. *LTRIM\$* needs a single argument that is the source string, and generates a new string where all trailing spaces are stripped off.

Example

This example demonstrates the effect of *LTRIM\$*:

```

outmode -2

```

```
let a$ = "hello "  
let b$ = " world !"  
let c$ = ltrim$(a$) + ltrim$(b$  
print c$
```

Remarks

Only spaces (ASCII value 0x20) are removed. Other white space characters are not. See also [RTRIM\\$](#).

MEMCFG

Description

This command can be used to configure the memory environment of the program that contains the command as its first line. If used, *MEMCFG must be the first statement* of a BASIC script, otherwise it is rejected. The configuration is only valid for this program.

MEMCFG requires three arguments in the following order:

- 1) Size of the BASIC heap (space for variables, arrays, string operations and subroutine pointers)
- 2) Size of the code segment, that is, memory which contains the compiled byte code.
- 3) Size of the data segment. This segment holds constant data that is accessible by the READ statement.

All arguments must be a multiple of four.

The following restrictions apply:

The heap (first argument) should not be smaller than 3500 bytes, since the virtual machine which executes the script uses the heap internally for temporary data like string copies and so on.

None of the arguments shall be zero.

The sum of all arguments can not exceed 12288 bytes.

If *MEMCFG* is not used, the default configuration is:

Heap size: 8192 Bytes

Code segment: 3072 Bytes

Data segment: 1024 Bytes

Example

This is the first line of the script

```
MEMCFG 3500, 4000, 100  
' script continues ...
```

Result: This sets the heap to 3500, the code segment to 4000 and data segment to 100 bytes

Remarks

The majority of BASIC scripts doesn't need to change the memory configuration, since the default settings are suitable for most needs. Though, most configuration mistakes are detected by the VM, programs that use a faulty configuration can behave unexpectedly.

MID\$

Description

Similar to *RIGHT\$* and *LEFT\$*, *MID\$* is a string function that can be used to extract character sequences from a source strings. But differently from those other string functions, *MID\$* needs a starting position and a length counter to generate new strings. More precise: *MID\$* needs three arguments, the first one is the source string, the second one is the offset from the right and the last one is the length of the result string.

Example

Extract "567" out of "0123456789abcdef"

```
outmode -2
let a$ = "0123456789abcdef"
let b$ = mid$ (a$, 6, 3)
print a$
print b$
```

Remarks

See also [RIGHT\\$](#) and [LEFT\\$](#)

MILLIS

Description

MILLIS is a read-only pseudo variable that can be read by a program to query the internal free-running millisecond counter. When the module boots, that counter is set to zero and then keeps incrementing every 1/1000 second. Since the counter is a 32-bit integer, it will wrap around to zero after 49.71 days. Because the scripting language is only aware of signed integers, *MILLIS* wraps around to -1 after 24.86 days and then counts up to 2147483647 before it wraps again to -1. Please consider that if you're make long-time calculations using *MILLIS*.

Example

Repeatedly output of the current counter value

```
OUTMODE -2
DO
  PRINT MILLIS
LOOP
```

Remarks

Because *MILLIS* is read-only, most attempts to change it will be rejected by the compiler or the runtime system.

MOVE

Description

The *MOVE* command can be used to move files between directories and also to rename files. It takes two arguments as strings, the first one is the current path and name of the file and the second one is the new path/name. *MOVE* is also able to move or rename directories.

Example

Moves (renames) the file "hello.txt" on the memory card:

```
OUTMODE -2
MOVE "hello.txt", "newname.txt"
IF LASTERR = 0 THEN
    PRINT "the file was renamed successfully"
ELSE
    PRINT "something's gone wrong ";
    PRINT LASTERR
END IF
```

Remarks

MOVE sets the pseudo variable *LASTERR* to report success or failure. Files that are currently open cannot be moved. Do not move or rename directories which contain files that are open for reading or writing. *MOVE* is new since version 4.57

OPEN

Description

This command opens files for read or write access. *OPEN* requires three arguments:

A string that is the access type

"R" - Open file for reading. The file must exist

"W" - Open file for writing. A new file will be created.

"WB" - Same as W but uses buffering. (new since V4.20)

"A" - Open file for appending data. The file must exist.

"AB" - Same as A but uses buffering. (new since V4.20)

A number in the range from 0...100

This is the file handle that must be used for subsequent access to the file.

The file name

This must be a FAT16 compatible string in 8.3 format.

OPEN sets *LASTERR* to 0 (ERR_OK) on success. Any other value indicates an error.

The buffered modes "AB" and "WB" are useful when small chunks of data must be written at a relatively high frequency. In buffered mode, the size of a single write (e.g. *PUT*) is restricted to 1550 bytes.

Example

The following example opens the file "test.txt" for reading and prints out the first few characters. It's a cheap UNIX "head" command.

```
outmode -2
open "R", 1, "test.txt"
if LASTERR = 0 then
  get 1, a$
  print a$
close 1
else
  print "could not open!"
endif
```

Remarks

OPEN only works on modules that have some kind of mass storage such as an USB Stick or SD Card.

OUTMODE

Description

OUTMODE is an extended function of the Avisaro Scripting Language that exists to define modes for output operations such as *PUT* and *PRINT*. Currently these output modes exist:

(-1) No output. All data is thrown away.

(-2) Synchronous output to the current I/O interface. That means, *PUT* and *PRINT* wait until all data is send out. The other end can block the BASIC program if it uses some kind of flow control.

(-3) Asynchronous output to the current I/O interface. In this case, *PUT* and *PRINT* never wait and the other end cannot stop data flow. Data will be lost if the other end can't process all in time.

(0 ... 100) Any valid and open file handle. Only valid for *PUT*. *PUT* then stores its data into an open file.

(-4 ... -9) Auxiliary ports. Only valid for *PUT*. Data is send to the auxialiary port.

(-100 ... -105) Web Controls. Only valid for *PUT*. Data is then displayed on the web page.

(-101 ... - 200) TCP connections. Only valid for *PUT*. Data is send over a TCP connection.

(-201 ... -300) UDP channels. Only valid for *PUT*. Data is send over UDP.

Example

Please see other samples. Most of them use outmode to switch on terminal output.

Remarks

Any value that does not fit into the above list can cause undefined behaviour.

PRINT

Description

PRINT is a standard BASIC command that can be used to output variables and other stuff to the terminal. In the Avisaro Scripting Language, all *PRINT* outputs go to the currently selected I/O interface, that can be RS232, CAN, SPI or IIC. Printable are string constants, numeric variables, string variables and array elements. Normally, every *PRINT* adds a CR/LF to the printed object in order to jump to the next line on a terminal. If you don't want that, put a semicolon at the end of a *PRINT* statement. In this case, CR/LFs are suppressed and the next *PRINT* statement continues in the same line. *PRINT* only works if the *OUTMODE* is correctly set.

Example

This little example shows some *PRINT*ed lines:

```
outmode -2
let a = 1
let b$ = "hello"
print "This is line: ";
print a
print "This is another line"
print "Printing a string variable -->";
print b$
```

Remarks

Use the *OUTMODE* command with -2 or -3 in order to print on the terminal. Because of memory constraints, it is advisable that every *PRINT* statement should only be used to print a single object.

PUT

Description

The *PUT* command is the counterpart of the *GET* command. With *PUT*, one can send contents of variables and arrays to various destinations. *PUT* needs two or three arguments. The first one is the handle to the I/O interface, where data is sent to. The second one is the variable that contains the data. If this is an array, *PUT* needs a third argument which indicates how many array elements should be sent.

PUT knows these destinations (handles):

Handle number	Interface type	Description	Firmware required
300 to 201	UDP	Handle for UDP channel	
200 to 101	TCP	Handle for TCP connection	
100 to 1	File	Handle for file on SD card	
-2	Data	Currently selected data interface (synchronous)	

	Interface		
-3	Data Interface	Currently selected data interface (asynchronous). Async. mode means, that <i>PUT</i> always send data and does not wait for the receiver to acknowledge it.	
-4	RS232	Auxiliary (= 2nd) RS232 interface (asynchronous)	
-5	I2C	I2C interface in Master mode using stop condition	
-6	I2C	I2C interface in Master mode not using stop condition	
-7	CAN	Primary CAN interface. Only arrays are allowed. Either the array size or <i>PUT</i> 's third argument must be a multiple of 28. The array is seen as one or more consecutive CAN frames. For a description of the structure of those CAN frames, please see the <i>GETCAN</i> and <i>PUTCAN</i> commands.	> 3.36
-8	CAN	Auxiliary (2nd) CAN interface. Same restrictions apply as for the normal CAN(-1) interface.	
-100 to -105	WEB	One of the six edit controls of the web page	
-201 to -224	I/O ports	Digital I/O and PWM ports	> 3.48

Valid sources for *PUT* are:

The *TIME* pseudo variable

The *TIME\$* pseudo variable

The *KEYS* pseudo variable

The *DATE\$* pseudo variable

Arrays, either complete or the first few elements

Single elements of byte- and integer arrays

Constant values

Strings

32 bit signed integer variables

Example

The following example sends the first three bytes of an array asynchronously to the current I/O interface and the *TIME\$* pseudo variable afterwards.

```

DIM a(20)
LET a(0) = 1
LET a(1) = 2
LET a(2) = 3
PUT -3, a, 3
PUT -3, TIME$

```

Remarks

PUT tries to send as much bytes as the length of the source variable in bytes is. Except for arrays, where the number-of-elements argument must be given.

When *PUT* returns, the pseudo variable *LASTERR* can be read to detect failures. If *LASTERR* is **not** zero (ERR_OK), an error occurred.

The handles -4, -5 and -6 must be opened with *AUXOPEN*, -2 and -3 are always open, if any kind of default I/O interface is active. Also -100...-105 are always open. Other handles must be explicitly opened using the appropriate functions.

If the CAN interfaces (-7 and -8) are used as destination, only arrays are allowed to *PUT*. In addition, if a complete array shall be send over the CAN interface, the size of that array must be a multiple of 28. If only the first few bytes of an array shall be send, then *PUT*'s third argument must be a multiple of 28.

If *PUT* is used on digital I/O ports (handles -201...-224), the second argument must be a constant or a single integer variable. In this case, *PUT* automatically switches the pin to output mode.

See also:

[PORT](#), [EXEC](#), [AUXOPEN](#), [LASTERR](#), [GET](#)

PUTCAN

Description

The *PUTCAN* command can be used to immediately send messages onto the CAN bus. *PUTCAN* needs a single argument which is the name of a byte-array. The array must be big enough to hold an entire CAN frame including additional information such like header and time stamps. Or that is to say: 28 bytes are needed.

The 8 bytes CAN byte-array is arranged in this way:

4 Bytes Header This is an arbitrary header that can freely be used. It is not send over the CAN BUS

4 Bytes Frame Descriptor

This field holds information regarding to the CAN frame. It is subdivided in:

Bits 16...19 are the data length counter. This is the number of data bytes the frame has.

Bit 30 is the RTR bit. If this bit is one, the frame was send out with RTR=1 right after the identifier.

Bit 31 defines the frame type. If this bit is set, an extended frame will be send. Otherwise the frame is a standard frame.

Other bits are unused and should be zero

4 Bytes Message ID This field contains the message ID.

8 Bytes Data This field contains the 8 data bytes that are transmitted to the CAN bus.

4 Bytes RTC timestamp This field is only be used with received frames. It contains the time stamp from the on-board real time clock.

4 Bytes millisecond counter This field is also only used with received frames. It contains the millisecond time stamp from a free running counter.

With exception of the data field, members of a CAN byte-array should not be modified directly. There's a convenient command named "*SETCAN*" that should be used to perform changes.

Example

The following example sends a CAN frame with 3 data bytes 200, 201, 202 and message ID 100:

```
dim a(28)
a(12) = 200
a(13) = 201
a(14) = 202
setcan a, 4, 3
setcan a, 1, 100
putcan a
```

Remarks

Even though, *PUTCAN* needs only 16 bytes of the array, it has to be as long as a *GETCAN* array. This enables applications to receive a frame using *GETCAN*, do some modifications and send it out without having to copy something. When everything's gone right, *PUTCAN* sets *LASTERR* to *ERR_OK* (0). If the argument of *PUTCAN* was wrong (e.g. it wasn't an array), *PUTCAN* sets *LASTERR* to *ERR_ARGUMENT* (4). *PUTCAN* does not buffer CAN frames. They are immediately sent to the hardware transmitter. If the transmitter is busy, *PUTCAN* does some retries before giving up. In this case, *LASTERR* becomes *ERR_REJECTED* (12). The application is free to re-send the frame or ignore that error. See also the [GETCAN](#) and [SETCAN](#) pages.

PWM

Description

To output square waves of variable length and pulse widths, one can use the *PWM* command.

PWM requires three arguments:

1. The first one is the pin number on the Avisaro Module. .
2. The second argument is the length of the pause in 1/2 μ s units.
3. The third argument is the length of the pulse, also in 1/2 μ s units. Both together form a square wave which period is determined by the sum of the second and third argument.

To make this clear: If you want to generate square waves with a frequency of 1 μ s, call *PWM* with 1 and 1 as second and third argument. Unfortunately, the pulse width for an 1 μ s wave is not adjustable because this is the highest possible frequency. To generate waves that have a period of exactly 1s, but a very small pulse of 1 μ s, call *PWM* with 199998 as second and 2 as third argument, and so on...

Pin 5, 6, 7, 9, 10, 11 can be used as PWM outputs simultaneously. Since all outputs share a common timer, the frequency of each signal must be equal to the the others. To follow this rule, simply make sure that the sum of the last two arguments of the *PWM* command is the same for all outputs.

Example

The following example slowly moves a hobbyist RC servo (Compatible to a S03NXF Std. Servo) from one end to the other and vice versa:

```
outmode -2
lab:
print "+"
for p = 1900 to 4500 step 10
pwm 7, 40000, p
sleep 50
next
print "-"
for p = 4500 to 1900 step -10
pwm 7, 40000, p
sleep 50
next
goto lab
```

Remarks

Please note: The PWM command immediately changes pin configuration for the selected ports. No explicit initialization is required.

REM

Description

REM is the standard BASIC keywords for single-line comments. A line that begins with REM is completely ignored by the compiler.

Example

This example shows the effect of REM. Only 1 and 3 are printed

```
outmode -2
print 1
REM print 2
print 3
```

Remarks

You can also use the inverted comma ' to start a comment line.

RESOLV

Description

RESOLV transforms IP addresses, given in standard dotted format, into 32 bit signed integer values. RESOLV is also able to resolve domain names, if a name server is registered in the IP configuration of the module.

Example

This little example prints the 32 bit signed integer representations of `www.avisaro.com` and `192.168.0.1`

```
outmode -2
let a = resolve ("www.avisaro.com")
if a = 0 then
    print "could not resolve domain name"
else
    print a
end if
let a = resolve ("192.168.0.1")
if a = 0 then
    print "could not resolve ip address"
else
    print a
end if
```

Remarks

Since dotted IP conversion depends only on code, it works always, even the module does not have a network interface. On the other hand, resolving a domain name implies a functional network and, in most cases, internet access.

RIGHT\$

Description

RIGHT\$ is a string function that can be used to extract the rightmost characters of a given input string. *RIGHT\$* needs a source string and the number of characters which shall be extracted. It then constructs a new string which contains the requested characters.

Example

Prints the second word of "hello world":

```
outmode -2
let a$ = "hello-world"
let b$ = right$ (a$, 5)
print b$
```

Remarks

If the number of requested characters is equal or greater than the number of characters in the source string, the new string is then equal to the source string. See also [MID\\$](#) and [LEFT\\$](#)

RTRIM\$

Description

RTRIM\$ can be used to remove all space characters from the end of a string. *RTRIM\$* needs a single argument that is the source string, and generates a new string where all trailing spaces are stripped off.

Example

This example demonstrates the effect of *RTRIM\$*:

```
outmode -2
let a$ = "hello "
let b$ = "world "
let c$ = rtrim$(a$) + rtrim$(b$) + "!"
print c$
```

Remarks

Only spaces (ASCII value 0x20) are removed. Other white space characters are not. See also [LTRIM\\$](#).

SAVE

Description

The SAVE command can be used to store data into internal flash memory. There's a dedicated section for BASIC programs where they can store non-volatile information. SAVE needs two arguments:

- 1) The first one is a zero-based address of the flash memory, where the object should be stored.
- 2) The second one is the object itself, which can be a complete array, a string, an array-element or a single 32 bit signed integer variable.

A string is stored "0" terminated - thus there is one more byte for each string. A 32 bit integer is stored in 4 bytes, a byte stored as one byte.

The length of the storage area is 4k Byte. There is no 'wear' leveling - so write cycles are limited to 100.000.

Example

This program first stores a string into flash memory at address 2000, then reads it back and prints it out.

```
outmode -2
let a$ = "hello"
save 2000, a$
load 2000, b$
print a$
print b$
```

Remarks

See also: [LOAD](#) command.

SCAN

Description

The *SCAN* command enables a BASIC program to search for WLANs in the neighbourhood. *SCAN* requires two arguments. The first one is the scan mode, which can be 0 or -1. For a normal scan, the scan mode must be 0. If -1 is given, *SCAN* throws away its previous result buffer. The second argument is the time (in milliseconds) that *SCAN* stays on each channel while listening for beacons. If the first argument is -1 (free previous buffer), the second argument can be any value.

On success *SCAN* sets the *LASTERR* status variable to ERR_OK(0). *LASTERR* becomes ERR_FIL_EXHAUSTED(26) if a result buffer couldn't be allocated. If a previous result buffer is still pending, *LASTERR* is set to ERR_REJECTED(12).

Example

This example starts a scan, wait until finished and then prints out the result.

```
outmode -2
' Start a scan with 300ms per channel
scan 0, 300
' Busy wait until scan finished
while scanned < 0
  sleep 100
  print ". ";
wend
' Store number of found WLAN nets and print it
let a = scanned
print "found wlan: ";
' Loop thru all records and print them
let s$ = ""
for s = 1 to a
  getscan s$
  print s$
next
```

Remarks

A New *SCAN* can only be started, if the last result buffer was completely read or thrown away by calling *SCAN* with the first argument set to -1. See also [GETSCAN](#).

See also

[SCAN](#) : Search for WLAN networks

[SCANNED](#) : Returns number of found WLAN networks

[GETSCAN](#) : Read results from scan command

[STATUS\(-4\)\]](#) : Returns status of WLAN connection

SCANNED

Description

SCANNED is a read-only pseudo variable. Writes to *SCANNED* prohibited and treated as error. *SCANNED* can be used in conjunction with the *SCAN* command to read the current scan status or, if finished, to get the number of found WLAN nets.

Example

The following program starts a scan and polls *SCANNED* until the scan is complete. It then reads the number of found nets and prints the result.

```
outmode -2
' Start a scan with 300ms per channel
scan 0, 300
' Busy wait until scan finished
while SCANNED < 0
  sleep 100
  print ".";
wend
' Store number of found WLAN nets and print it
let a = SCANNED
print "found wlans: "; a
' Loop thru all records and print them
let s$ = ""
for s = 1 to a
  getscan s$
  print s$
next
```

Remarks

SCANNED reads -1 if there's no scan in progress. While a scan is running, *SCANNED* reads -2 and if the scan is over, *SCANNED* becomes a positive value that is the number of found WLANs in the neighbourhood. See also the [SCAN](#) command.

See also

[SCAN](#) (Search for WLAN networks), [SCANNED](#) (Returns number of found WLAN networks), [GETSCAN](#) (Read results from scan command), [STATUS\(-4\)](#) (Returns status of WLAN connection)

SEEK

Description

SEEK moves the file pointer of an open file to the specified position. *SEEK* needs two arguments. The first one is the file handle and the second one is the new file pointer position. When an offset above the file size is specified in write mode, the file size is extended to the offset and the data in the extended area is undefined. This is suitable to create a large file quickly, for fast write operations without cluster allocation delay.

Example

This example opens a file named "test.txt" for reading. It then moves the file pointer 3 bytes ahead and reads some bytes beginning at the fourth byte of the file.

```
outmode -2
open "R", 1, "test.txt"
if LASTERR = 0 then
  seek 1, 3
  get 1, a$
  print a$
  close 1
else
  print "could not open!"
end if
```

Remarks

Because the second argument is a 32 bit signed integer, the furthestmost position that can be reached with SEEK is at 2,147,483,647

SETCAN

Description

The *SETCAN* command enables a skript to easily change attributes of CAN arrays without having to fiddle around with array-indexing and bit operations. *SETCAN* needs three arguments. The first one is the array name, the second one tells *SETCAN* what to do, and the last one is the new value that *SETCAN* should write into the CAN array.

Here's a list of actions (second argument of *SETCAN*) that *SETCAN* can perform:

- 1 Set the message ID
- 2 Set the frame type (0 is standard, 1 is extended frame)
- 3 Set the RTR bit. (0 is OFF, 1 in ON)
- 4 Set the number of data bytes (0...8)
- 5 Set the RTC (seconds) time stamp
- 6 Set the millisecond time stamp
- 7 Set the random 32 bit header value

Please see the *PUTCAN* page for description of the layout of CAN arrays.

Example

Please see the *PUTCAN* page. The *PUTCAN* example also demonstrates *SETCAN* calls.

Remarks

If everything's gone right, *SETCAN* sets *LASTERR* to ERR_OK (0). Otherwise, if an argument is out of range, *LASTERR* will become ERR_REJECTED (4)

SETLEDS

Description

There is a new structure to control I/O ports more effectively. See [here](#) for a description on how to control I/O ports.

The SETLEDS command enables a program to switch some digital I/O lines. SETLEDS needs exactly one argument which contains the bits. A zero bit switches a port OFF as a one bit switches it ON. The pin-assignment of the bits is:

Bits 0,1,2,3 These bits control the internal LEDs on the MCU module.

Bits 4 and 5 These bits control the external LEDs on the trailer module

Bit 6 Controls the RTS line on the RS232 connector, but only if the RS232 driver doesn't use hardware flow control

Bit 7 Controls the DTR line on the RS232 connector.

Example

The following example let all the LEDs flash

```
let a = 1
do
  setleds a
  let a = a*2
  if a = 64 then
    let a = 1
  end if
  sleep 100
loop
```

Remarks

All bits above Bit7 are ignored. See also the [KEYS](#), [PUT](#) and [GET](#) commands.

SLEEP

Description

SLEEP puts the program asleep for the specified interval. To set this interval, *SLEEP* needs a single argument that is, the milliseconds to sleep. After time-out elapses, the program awakes and continues execution. As a special case, if you supply zero as argument, *SLEEP* gives away its time quantum so that another task is immediately scheduled.

Example

Prints some numbers with a delay of one second:

```
outmode -2
for s = 1 to 10
  print s
  sleep 1000
next
print "ready"
```

Remarks

SLEEP 0 initiates an immediate context switch to another task. Which task runs next belongs to the scheduling algorithm of the RTOS. The Scripting Language can't specify that.

STATUS

Description

The STATUS function allows a program to get the state of various handles. It needs one single argument that is the handle number to be queried. STATUS can be used on files, TCP connections, UDP channels and on some pseudo handles. The list below describes that in detail:

Files. Handles # 0...100

- 1 File is open for reading
- 2 File is open for writing
- 1 File is closed
- 0 Not a file handle

TCP sockets. Handles # 101...200

- 1 Socket is in listening state
- 2 Socket was in listening state and has just received a connection request (SYN)
- 3 Socket wants to connect to another TCP (Has sent a SYN)
- 4 Socket is about to be closed (in FIN_WAIT_1 state)
- 5 Socket is about to be closed (in FIN_WAIT_2 state)
- 6 Socket wants to close the connection
- 7 Socket has received last packet before the communication ends
- 8 Socket waits for last packet before the communication ends
- 9 Socket is connected
- 0 Socket is closed

UDP channels. Handles # 201...300

- 1 UDP channel is open, Handle in use
- 0 Handle not in use

WLAN, pseudo handle -4

Returns whether or not the WLAN module has a connection to an Access Point.

- 1 WLAN connected
- 0 WLAN not connected

See also

[SCAN](#) (Search for WLAN networks), [SCANNED](#) (Returns number of found WLAN networks), [GETSCAN](#) (Read results from scan command), [STATUS\(-4\)](#) (Returns status of WLAN connection)

Free packet buffers, pseudo handle -5

xNumber of free packet buffers

Auxiliary IIC Port, -15

xNumber of Bytes in TX Fifo

If IIC is used in Slave mode, Incoming and outgoing data is stored into FIFO buffers. STATUS(-15) returns the number of bytes that the TX buffer currently holds. Data is transmitted in the background, so this value may change from call to call.

Example

The following example demonstrates the STATUS function for file handles. You need an SD slot to run this example. If a file named "status.txt" already exists, it will be deleted.

```
outmode -2
close 1
kill "status.txt"
print "status of 1: ";
let s = status(1)
print s
open "W", 1, "status.txt"
if LASTERR <> 0 then
    print "failed to create new file"
end
end if
print "status of 1: ";
let s = status(1)
print s
close 1
print "status of 1: ";
let s = status(1)
print s
```

Remarks

STATUS will be extended in future releases in order to retrieve more system information.
V3.34 and below: STATUS can't be used in expressions and must always assigned to variables. This problem is fixed in V 3.35 and above.

STR\$

Description

STR\$ is a string function that generates strings from numeric variables and constants. STR\$ needs a single argument which is the value that should be converted.

Example

The following example proves that 1+2=12 and not 3 ;)

```
outmode -2
let a = 1
let b = 2
let c$ = str$(a) + str$(b)
print c$
```

TAB

Description

With TAB, one can generate strings that consist of many spaces. Such strings are helpful when formatting output. TAB needs a single argument that is the number of spaces the string should contain.

Example

This example prints a vertical triangular curve by using TAB statements

```
outmode -2
do
  for s = 0 to 20
    print tab(s);
    print "*"
    sleep 100
  next
  for s = 20 to 0 step -1
    print tab(s);
    print "*"
    sleep 100
  next
loop
```

Remarks

Because of memory constraints, a single string in the Avisaro Scripting Language has a maximum size of 255 characters. Therefore, TAB should not be called with a greater value than 255.

TIME\$

Description

TIME\$ is a pseudo-variable that can be queried to get the current time as string. The returned string contains hour, minute and second separated by colons:

```
HH: MM: SS
```

TIME\$ can only be read. Any write attempt is prohibited and ignored or rejected by the compiler.

Example

This simply prints the current time of day:

```
outmode -2
print TIME$
end
```

Remarks

There's no function to set the time from a BASIC program. Use the command line interface if you want to set a new time.

TIME

Description

TIME is a pseudo variable that holds seconds since 01.01.2007 in 32 bit signed integer format. *TIME* can only be read.

Example

This example prints the *TIME* value during ten seconds:

```
outmode -2
for s = 1 to 10
  print time
  sleep 1000
next
```

Remarks

TIME uses the battery-backed RTC.

UCASE\$

Description

UCASE\$ generates a new string that is the capitalized counterpart of a source string. All letters are converted to upper case. Digits and other characters are excluded from that conversion.

Example

Prints a completely capitalized "Hello World"

```
outmode -2
let a$ = ucase$ ("Hello World")
print a$
```

Remarks

Internally, the C-library function "toupper" is used. See also [LCASE\\$](#).

UDPOPEN

Description

The *UDPOPEN* command can be used to open an UDP communication channel. *UDPOPEN* requires six arguments:

1. A handle number in the range from 201 to 300. This handle will be used for subsequent communication.

2. An IP address of the receiving UDP. Use *RESOLV* to convert a dotted IP address into a properly integer number.
3. The TX port number. This is the port number for outgoing packets. Port numbers are in the range 0...65535
4. The RX port number. This is the port number for incoming packets.
5. A TX delay value in milliseconds. This value is only used when doing UDP streaming.
6. A checksum flag, either 1 or 0. If 0 is given, outgoing UDP packets have no checksum.

Example

This little application opens an UDP broadcast channel (IP address with all 255's) for reception and transmission, both on port 25. When it receives a packet, its length will be printed. Furthermore, the string "hello word" is sent out in 100 ms intervals.

```

outmode -2
let a = resolv ("255.255.255.255")
udpopen 201, a, 25, 25, 5, 0
let a$ = "hello world"
dim in1(1500)
do
  sleep 100
  put 201, a$
  get 201, in1
  if bytesread <> 0 then
    print "RX: ";
    print bytesread;
    print " bytes"
  end if
loop

```

Remarks

Calling this command only makes sense on Avisaro Modules that have some kind of network interface. Because UDP handles are a limited resource, they must be closed when the channel is no more in use.

VAL

Description

VAL is a function that converts a string containing digits into a numeric value. *VAL* needs a single argument which is the string.

Example

The following example demonstrates *VAL* by calculating the equation $30 + (-50) = -20$ where both operands are strings.

```

outmode -2
let a$ = "30"
let b$ = "-50"
let c = val (a$) + val (b$)
print c

```

Remarks

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial "+" or "-" sign followed by as many numerical digits as possible, and interprets them as a numerical value. The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function. If the first sequence of non-whitespace characters is not a valid integral number, or if no such sequence exists because either the string is empty or it contains only whitespace characters, no conversion is performed.

WHILE

Description

A *WHILE* loop is one of two BASIC's standard head-controlled loop constructs. Such a loop starts with the keyword *WHILE* and ends with *WEND*. *WHILE* loops evaluate an expression before each iteration. If that expression evaluates to true, then the loop body is executed. Otherwise, the loop is left.

Example

This example demonstrates a little counter with a *WHILE* loop:

```
outmode -2
let a = 1
while a < 11
  print a
  let a = a + 1
wend
```

Remarks

GOSUB, another *WHILE* loop or other loop types (such as *FOR...NEXT*) can be used from within a *WHILE* loop.

Contact

Avisaro AG
Grosser Kolonnenweg 18 /D1
30163 Hannover, Germany
Tel.: +49 (0)511 780 93 90
Fax,: +49 (0)511 353 196 24
E-Mail: info@avisaro.com
Web: www.avisaro.com