Scripting Manual

"Programming Avisaro 2.0 Series Products with
its build-in scripting language"

Version / Date 28.04.2014

1 TABLE OF CONTENT

2 THIS DOCUMENT

2.1 WORKING WITH THIS DOCUMENT

Using <cntrl> and mouse click, you can navigate through the document

2.2 RELATED DOCUMENTS

Avisaro Serie 2 - Command Interface Guide (ENG).pdf

Detailed description of the Command Interfae / API

Avisaro Serie 2 - Scripting Manual (ENG).pdf

Detailed description of the Scripting capability of the devices

Avisaro Serie 2 - Technical User Manual (ENG).pdf

All hardware related information and functional details for technical person

Avisaro Serie 2 - User Manual (ENG).pdf

Higher level information mainly for users of the devices

2.3 HISTORY

2.4 COPYRIGHT

Avisaro AG
Grosser Kolonnenweg 18 / D1
30163 Hannover
Germany

info@avisaro.com

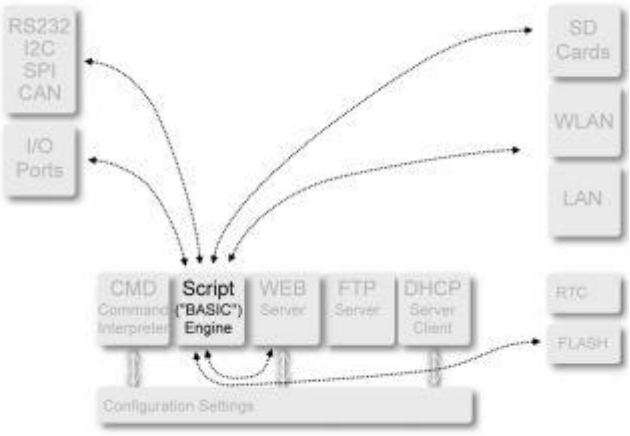
3 INTRODUCTION

The Avisaro 2.0 products have a build-in scripting language. This scripting can be used to define standalone behavior of the device. The scripts are like little BASIC programs which are stored on the Avisaro 2.0 product and start automatically when the product is powered up.

Typical applications are formatting of data to be stored on a SD card or contacting a server in the network to post collected data. The scripting engine is designed for small applications - use an external controller for large full scale applications.

The Scripting Engine allows to:

- Get and send data to a data interface (RS232, CAN,...)
- Control I/O ports, PWM and analog input
- Write and read data from the SD memory card
- Send and receive data through WLAN and LAN
- Store data into internal flash
- Post and get information from the build-in webserver



4 USAGE: COMMAND INTERFACE ("API") VRS. SCRIPTING ("APPS")

There are two main methods to operate the Avisaro 2.0 product:

The Command Interface allows to send commands from an external unit. This unit is typically a SPS control or a micro controller. Most commands can be send in an easy to read ASCII or compact binary format. Commands exists to ...

- Read and write data on SD cards
- Establish connections and share data through WLAN and LAN
- Configure the product and check status

The Scripting Engine allows to have the Avisaro 2.0 product perform functions on a self-sustained basis. The engine is designed to perform small tasks rather than large scale applications. Scripts are stored in internal flash and executed upon power up. Structures exists to ...

- Read and write data on SD cards

- Establish connections and share data through WLAN and LAN
- Parse and reformat data
- Control I/O ports, PWM and analog input
- Do if-then-else, do-loops, for-next, gosub-return,.... structures

Decision matrix:

Application	BASIC Scripting	Command Interface	Details
Send one datastream from A to B through WLAN	√		Ready to use scripts are available. Connection handling is done by Avisaro, user simply sends data.
Send several parallel datastreams from A to B		√	A SPS or micro controller sends or retrieves data to several server. Commands are send to control connections.
Store datastream into file	√		Ready to use scripts are available. File handling is done by Avisaro, user simply sends data.
Poll sensor and store data into file	√		Scripting is powerfull enough to poll a sensor through i.e. rs232, i2c, ..., than format data and store data into file. No aditional controller is needed.
High Speed Data		√	For high performance applications, the "packet commands" in binary form is used.

Note:

Avisaro 2.0 "Box" and "Cube" products are shipped with pre-installed Scripts. Those Scripts can be changed to perform a different funciton or they can be disabled to use the command interface.

Avisaro 2.0 "Modules" are shipped with no Script installed.

5 LIMITS OF SCRIPTING IN NUMBERS

Scripting is a powerful tool. It allows to adjust easily the product to your needs.

Please bear in mind:

The scripting capability is designed for smaller and targeted tasks. Is is not designed to be a generic, general purpose programmable device. Were the limits are, depends on size of script, speed and amount of data to be stored during operation.

A few limits are (unless otherwise stated, all products have the "regular firmware"):

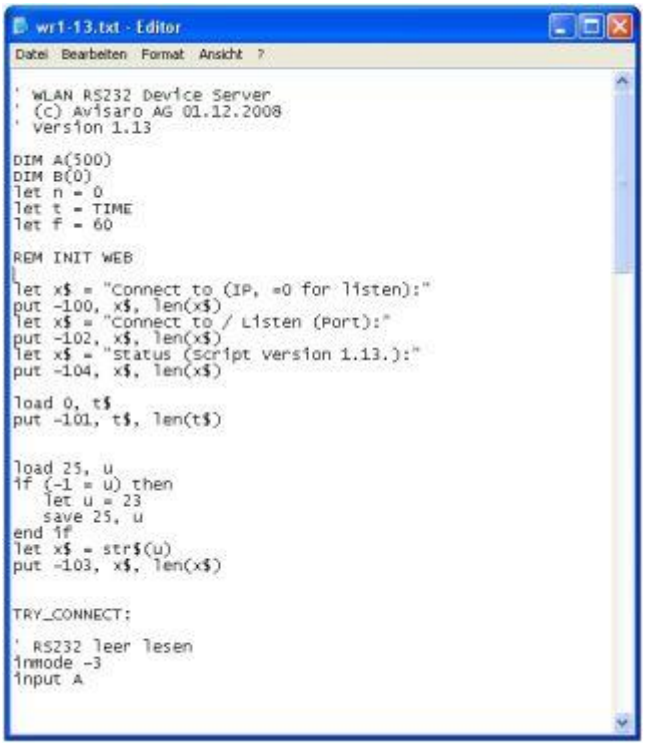
Ressource	Limit (regular Firmware)	Limit (Extended Firmware)	Comment
Length of source code	12.288 Bytes	24.576 Bytes	equals the size of Source Code file loaded into the module
Default size of tokenised code.	3.072 Bytes	6.144 Bytes	Can be modified using command memcfg
Number of 'goto' commands	30	60	Including gosub commands
Nested gosubs	26	26	
Number of variable definitions	30	30	This is the number of variables, not the amount of memory they are using.
RAM Size: Standard Split: Heap size Code segment Data segment	12.288 Bytes 8.192 Bytes 3.072 Bytes 1.024 Bytes	24.576 Bytes 16.384 Bytes 6.144 Bytes 2.048 Bytes	The memory split can be changed by the scripting command MEMCFG

6 WRITE AND UPLOAD SCRIPTS

6.1 WRITE AND EDIT SCRIPTS

Scripts for the Avisaro 2.0 products can be written with any text editor. Windows "Notepad" or "Editor" is sufficient. Typical scripts have less than 200 lines of code.

To start off easily, take an existing script as a basis and start to modify it.



6.2 LOAD AND RUN SCRIPTS

Scripts are loaded into the flash memory of the Avisaro module. Thus, they are available even after power cycling the module. Scripts can also be configured to start automatically after power up.

In addition, there is an option to have scripts loaded temporarily, leaving the script in flash memory unchanged. This feature is good for 'once in a while' tasks or for testing.

There are three methods to load and run a script:

- **Build-in Website:** If the product is equipped with a WLAN or LAN interface, scripts can be loaded through the build-in configuration web site. This is probably the most easy way during development.
- **SD Cards:** If the product is equipped with a SD card slot, the Scripts can be uploaded using a batch file. This is a good methods for high volume deployment.
- **Command interface:** Particular using the RS232 interface and a terminal program, scripts can be uploaded this way. A good way for interactive debugging during development.

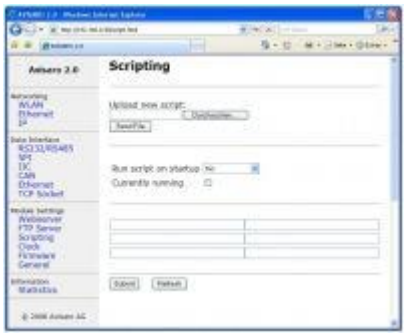
6.3 LOAD AND RUN SCRIPTS USING W/LAN INTERFACE (CONFIGURATION WEBSITE)

To load a script using the build-in configuration website, follow the following steps:

1. Navigate to the configuration web site of the module using any suitable browser. If default IP settings are used, simply type "192.168.0.74" in the address field of your browser.
2. Navigate to the "Scripting" section of "Module Settings".
3. Click on 'search' and select the script file you want to upload.
4. Click on 'send file' - that's all.

If the Script does not start, there are probably syntax errors. Navigate to the page "cmd" and type in the command "run". If there is an error message, it will be displayed here.

6.3.1 Upload new script:



Select a script file and press 'send file'

Run script on start-up:

No: Script is not executed when devices is powered up

Asynchronous: Script is executed automatically, when devices is powered up.

This option is recommended !

Exclusive: Script is started exclusively - for experts only.

Currently running

Checked: The script is up and running. To manually start a script, select this box and click 'submit'

Not checked: The script is not running. To manually stop a script, uncheck this box and click 'submit'. Notice: after pressing 'submit', the box comes up check again. Press 'refresh' to see whether the script really stopped.

3 x 2 Table

This table can be used by scripts to display data on the web site or the ask user to enter data. Content of this table therefore depends on the loaded script. See script documentation for details.

6.3.2 Hidden: CMD

This page is not linked and can be reached only by entering the page adress directly (xxx.xxx.xxx.xxx/cmd). The reason for this procedure is, that this page is very powerful since it allows to enter commands.

Enter any command from the "Command Interface" and press enter or hit submit. The command is executed - no matter even if a Script is currently executed.

This feature is particularly useful when:

Trying to get statistics (sstat?, fstat?)

Get a listing of an uploaded BASIC Script (list) or check for syntax errors (run)

6.4 LOAD AND RUN SCRIPTS USING SD MEMORY CARDS

To load a script into the internal flash memory, follow the following steps:

1. Copy the script (a regular text file) onto the SD card
2. Create a text file on the same SD card with the name 'autorun.txt'. Enter the line "load yourfile.txt" in this file. Close this line with 'enter' .
3. Place the SD card in the Avisaro 2.0 product and turn on power and wait 5 seconds
4. The script is loaded into the flash memory of the Base Module. Both files (autorun.txt and yourfile.txt) can be removed from SD card.
5. To have the script start right away, add a 'run' command in the autorun file. To have the script start after power cycling the product, add a 'run auto' command.

Example for a “autorun.txt”[1] file:

```
load yourfile.txt

run auto
```

6.5 LOAD AND RUN SCRIPTS TEMPORARILY

For special purposes or for testing it might be necessary to execute a script, but not to store it into the internal flash memory. Thus, after power cycling, the previously stored script remains active.

To run a script temporarily, follow the following steps:

1. Copy the script onto a SD card. Name the script "temp_run.bas". This particular file name is recognized by Avisaro Loggers.
2. Insert the SD card into the logger. The logger automatically finds the "temp_run.bas" file. This script is then started without being copied into internal flash memory. Currently executed other scripts are stopped.
3. To clear the temporarily loaded script, restart the logger by i.e. power cycling.

Notice:

- The automatic execution of scripts works when the logger is powered up while the SD card with "temp_run.bas" is inserted. To load and run the script temporarily upon power up, add a 'autorun.txt' file. Enter the command "run temp_run.bas" to force execution of this script without loading it into internal flash.

Details:

[1] Notice: the file system has 8.3 naming: file names have max 8 characters and 3 extension character

- The execution of "temp_run.bas" forces currently running scripts to be stopped. However, a mechanism exists to avoid that just loaded scripts are stopped and started again when the same SD card is ejected and inserted. This works based on the card ID - a unique ID each SD card has.

6.6 LOAD AND RUN SCRIPTS USING COMMAND INTERFACE

6.6.1 Stop Script

To use the command interface it is substantial that the script is not running:

A script can be easily stopped by using the command RUN MANUAL in a autorun.txt

Use a normal text editor and create a file on the SD card with the name autorun.txt with the following content:

```
run manual
```

(press the enter key at the end of the line)

Insert the SD card in the slot of the logger and restart the logger.

All LED's remain off.

6.6.2 Using the Command Interface

Now you can communicate with the logger via interface. Use the referring commands and send it to the logger.

To restart the script just send the commands

```
run auto
```

```
run
```

Then the command interface is switched off.

7 BASCOM – PC COMPILER

Scripts can be loaded as a (1) text-file or (2) in a tokenized way into the Avisaro 2.0 products. For the first case (1) there is no need for this tool - just load the text file as it is. For the second case (2), use this Bascom tool.

The Bascom PC Compiler allows to compile BASIC scripts on a PC. This is particularly helpful to:

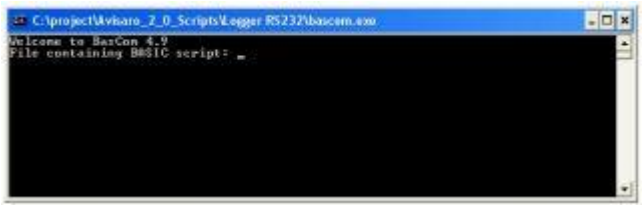
- Shorten the development cycle
- Scramble scripts to protect intellectual property when distributing scripts

The compiler does not allow to run or emulate scripts on a PC.

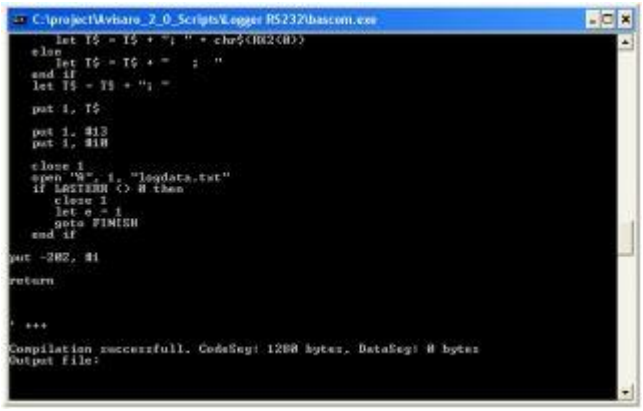
7.1 USAGE

To use the Bascom compiler, unpack the zip file and copy the "bascom.exe" into the folder which contains the script file. Double-click bascom.exe to start.

1) Enter the filename of the script file.

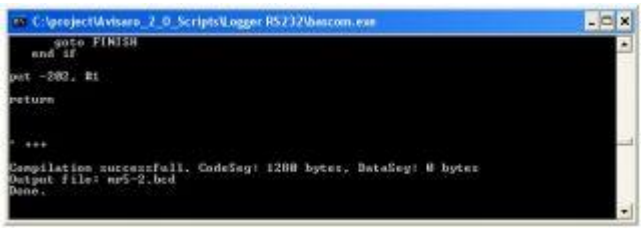


2) If all is ok, the compiler reports success. Enter the name of an output file. This output file contains the script in a machine coded (tokenized) way. This output file can be loaded into the Avisaro 2.0 products the same way as a script in Ascii format can be.



3) In case of an error, the line number is displayed. Only one error is displayed at a time. Correct the error and start over.

4) If all is ok, close the cmd window.



8 HOW TO HANDLE THE DATA FLOW

A script needs the following basic elements:

- Define Variables: Using the command DIM you have to define your variables and with LET you can assign values to the variable.
- Open interface: With the command AUXOPEN open the ports you want to address
- Receiving data: This can be done by the commands GET or IMPUT
- Processing data: For example reformatting data or checking the data flow. FOR..NEXT or IF loops are very helpful
- Output data: Mainly this can be done by the PUT command.

There is a long list of additional commands for further purpose, like controlling the files, wlan network and data flow or simply printing lines.

8.1 RS232 INTERFACE

There are two RS232 interfaces which can be used on an individual basis. RS232 interface 'Port 1' is the primary one, 'Port 2' is the auxiliary one. The primary port can be used to transmit commands to configure the Avisaro product. Both ports can be used to receive and send RS232 data to be stored or to be forwarded wirelessly over a network.

8.1.1 Setup and configuration

8.1.1.1 Activating and configuration of RS232 interface 1 (Primary)

The RS232 port 1 is designed to receive user data as well as commands addressed to configure the Avisaro product.

For all Avisaro RS232 Data Logger and Avisaro RS232 WLAN products, the RS232 interface is already activated and operates with default values (see below). Change baud rate and filter settings using the web interface or SD memory card:

If a Avisaro Module product was purchased or after a 'reset to factory settings' command, the RS232 interface is activated by default.

8.1.1.2 Activating and configuration of RS232 interface 'Port 2' (auxiliary)

The RS232 port 2 is designed to be used in scripting language only. It is well suited to receive and send RS232 data for data logging or data transfer over wireless networks.

The RS232 port 2 is activated only within BASIC scripts:

Use the command auxopen to activate the second RS232 interface. Example:

```
auxopen -4, 9600, ASC("N"), 1, 8, ASC("N")
```

Most users decide to set baudrate and other settings for this port within the script - since scripts can be loaded easily, changes are easy to do.

8.1.2 Working with RS232 in Scripts

8.1.2.1 Read RS232 data

RS232 data is presented a stream of bytes. They can be read one-by-one or copied into an array of bytes. The script needs to define that the RS232 interface should be handled by the scripting engine, rather than by the command engine. Use the inmode command to do so:

```
inmode -3
```

As an example, define an array with 100 bytes. This array will be used to hold RS232 data

```
dim A(100)
```

Using the get command, a message from RS232 port 1 is retrieved:

```
get -3, A
```

To read a message from RS232 port 2, use '-4' instead of '-3'. Alternatively to the 'get -3' command, the input command can be used.

The get command is not blocking - if no data were received, the command returns with no data. To verify how much bytes were received, use the 'bytesread' system variable:

```
if BYTESREAD > 28 then
     ' data were received
     ' do something with it
end if
```

Of course, there is always the good old 'input' command:

```
input a$
```

8.1.2.2 Process RS232 data

The content of the RS232 data can be checked for specific data, reformatted, ... One way to do so is to check the array byte by byted:

```
for n = 0 to BYTESREAD-1
       ' check with i.e. if a(n) = ASC("g")
       ' modify with let a(n) = a(n) + 5
next n
```

8.1.2.3 Output RS232 data

The byte array can be send out using the put command: (tl_files/dynamic_dropdown/link.gif more). In this case the same amount of data as previously read in is printed out again:

```
put -4, a, BYTESREAD
```

Of course there is always the good old 'print' command:

```
print "Hello world"
```

8.2 CAN BUS INTERFACE

The are two CAN interfaces which can be used on an individual basis. CAN interface 'Port 1' is the primary one, 'Port 2' is the auxiliary one. The primary port can be used to transmit commands to configure the Avisaro product. Both ports can be used to receive CAN messages to be stored or to be forwarded wirelessly over a network.

8.2.1 Setup and configuration

Activating and configuration of CAN interface 1 (Primary)

The CAN port 1 is designed to receive user data as well as commands addressed to configure the Avisaro product.

For all Avisaro CAN Data Logger and Avisaro CAN WLAN products, the CAN interface is already activated and operates with default values. Change baudrate and filter settings using the web interface or SD memory card.

If a Avisaro Module product was purchased or after a 'reset to factory settings' command, the CAN interface must be activated and configured.

Activating and configuration of CAN interface 'Port 2' (auxiliary)

The CAN port 2 is designed to be used in Scripting language only. It is well suited to receive and send CAN messages for data logging or data transfer over wireless networks.

The CAN port 2 is activated only within BASIC scripts:

Use the command auxopen to activate the second CAN interface. Example:

```
auxopen -8, 125000, 0, 10000, 0, 0
```

Most users decide to set baudrate and other settings for this port within the script - since scripts can be loaded easily, changes are easy to do.

8.2.2 Working with CAN in Scripts

Principle of operation

CAN messages are stored internally in a fixed 28 Byte long format - no matter how long the original CAN messages was. This message format is documented at the end of this site. All operations with CAN messages are based on the 28 Byte format: To receive a message, a 28 Byte array must be declared; sending CAN messages over TCP is done in 28 Bytes (or multiple of it) chunks.

Read CAN messages from CAN bus

The script needs to define that CAN messages should be handled by the scripting engine, rather than by the command engine. Use the inmode command to do so:

```
inmode -3
```

Define an array with 28 bytes. This array will be used to hold a CAN message:

```
dim A(28)
```

Using the get command, a message from CAN port 1 is retrieved:

```
get -3, A
```

To read a message from CAN port 2, use '-8' instead of '-3'. Alternatively to the 'get -3' command, the getcan command can be used.

The get command is not blocking - if no message was received, the command returns with no data. To verify whether or not data were received, use the 'bytesread' system variable:

```
if BYTESREAD = 28 then
    ' message was received
    ' do something with it
end if
```

Process CAN messages

The content of the CAN message is packaged into those 28 bytes. The storage format is described at the end of this page. To avoid bit operations, the command 'caninfo' was introduced to extract specific CAN fields:

```
if caninfo(1) = 5 then
   ' a CAN message with CAN ID 5
   ' was received
```

```
end if
```

The data bytes of the can message are accessed by addressing the array: A(13) contains the first byte of the payload ... all the way to A(20) for the 8th byte.

8.2.3 Details of 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	Real time clock time stamp in seconds since 01.01.2007
22	15...8	
23	23..16	
24	31..23	
25	7...0	Millisecond time stamp. Value in ms since module power on.
26	15...8	
27	23..16	
28	31..23	

Example:

The following shows some example frames that can be send from the Avisaro module to the CAN bus. In this examples, header and timestamp fields are not used.

Extended frame, ID is 0x0301, three bytes of data: 01, 02, 03

00 00 00 00 00 00 03 80 01 03 00 00 01 02 03 00 00 00 00 00 00 00 00 00 00 00 00 00

Standard frame, ID is 0x0301, eight bytes of data: 01, 02, 03, 04, 05, 06, 07, 08

00 00 00 00 00 00 08 00 01 03 00 00 01 02 03 04 05 06 07 08 00 00 00 00 00 00 00 00

Extended frame, ID is 0x11223344, three bytes of data: 01, 02, 03

00 00 00 00 00 00 03 80 44 33 22 11 01 02 03 00 00 00 00 00 00 00 00 00 00 00 00 00

Extended frame, RTR bit is on, ID is 0x11223344, three bytes of data: 01, 02, 03

00 00 00 00 00 00 03 c0 44 33 22 11 01 02 03 00 00 00 00 00 00 00 00 00 00 00 00 00

Legend

- Header bytes
- Frame Descriptor
- ID
- Data Field
- Timestamps

Please note that unused bits of the Frame Descriptor and ID Field *always must be zero*.

8.3 Using I2C interface (Slave and Master)

The I2C interface is available within Scripting in master and in slave mode. To use the slave mode, the module can be either configured as regular data interface or as auxiliary interface. The master mode can be activated only as auxiliary interface in scripts.

8.3.1 Setup and configuration

Activating I2C Slave as primary Data Interface

To operate I2C in slave mode, the regular data interface settings are used.

... via Web interface

If there is a network interface, the build-in configuration website can be used. Navigate to:

- General: Enter I2C as active Data Interface
- I2C: Enter the I2C address the module should respond to.

... via SD card

To activate the I2C interface, copy the following commands into a file called 'autorun.txt'. Place this file on a SD card, insert card and power on device. See for details on how to configure Avisaro products using the 'autorun.txt' file.

```
prot i2c

i2c 73
```

See Mehr here for details on the 'prot' command

See Mehr here for details on the 'i2c' command

Activating I2C Slave as second (auxiliary) interface

The I2C interface can be activated as a second (auxiliary) interface in scripting. If for example, the primary data interface is set to RS232, the Avisaro device still can be connected to a I2C bus as a slave. Data can be send and received from those devices to be stored on SD card or forwarded wirelessly.

Use the command auxopen to activate the I2C interface in slave mode. Example:

```
auxopen -5, 0, 43, 1, 0, 0
```

This example opens the I2C interface, sets the I2C bus address to communicate on to 43.

Activating I2C Master as second (auxiliary) interface

The I2C interface in master mode is configured in scripting language only. It is well suited to connect sensors or other I2C devices to the Avisaro products. Data can be send and received from those devices to be stored on SD card or forwarded wirelessly.

Use the command auxopen to activate the I2C interface in Master mode. Example:

```
auxopen -5, 100000, 43, 0, 0, 0
```

This example opens the I2C interface, sets the I2C bus address of the client to communicate with to 43.

8.3.2 Working with I2C (Master) in Scripts

Send data to I2C device

To following example sends the string "hello" to a I2C client with the bus address 51 (decimal). :

```
dim a(10)
auxopen -5, 100000 , 51 , 0 , 0, 0
let a(0) = 104
let a(1) = 101
let a(2) = 108
let a(3) = 108
let a(4) = 111
put -5, a, 5
```

The dim command is used to define an array of 5 bytes to hold the message to be send.

The auxopen command opens the I2C Master interface. The client address is defined with this auxopen. Thus before sending data to different clients, the auxopen command as to be repeated with the appropriate bus address.

The following let commands fill the array with the ASCII codes for "hello".

The put command finally sends the data to I2C client. In this example, 5 bytes are send.

This example uses a 8 bit I2C address. See later chapter on how to use 11 bit addresses.

Read data from I2C device

The following example reads data from the I2C device. We assume the I2C is still configured from the above write command. This I2C device requires to write the desired register (here: 0) and than start reading.

```
let a(0) = 0
put -6, A, 1
get -5 , A
```

In order to perform a write and than read, we use the handle "-6" in the put command which means "send no stop condition". Than, the get follows to read bytes. In this case, 10 bytes are read since the array a() is 10 bytes long.

8.3.3 Working with I2C (Slave) in Scripts

Read I2C data (Slave)

I2C data is presented a stream of bytes. They can be read one-by-one or copied into an array of bytes.

The script needs to define that the I2C interface should be handled by the scripting engine, rather than by the command engine. Use the inmode command to do so:

```
inmode -3
```

As an example, define an array with 100 bytes. This array will be used to hold I2C data

```
dim A(100)
```

Using the get command, a message from I2C interface is retrieved:

```
get -3, A
```

The get command is not blocking - if no data were received, the command returns with no data. To verify how much bytes were received, use the 'bytesread' system variable:

```
if BYTESREAD > 28 then
    ' data were received
    ' do something with it
end if
```

Of course, there is always the good old 'input' command:

```
input a$
```

8.4 I/O, PWM, ANALOG PORTS

If a pin is not used for data transmission, it can be used as an input, output, pulse width modulation (PWM) or analog pin. This feature is available for the WLAN and Logger Module.

8.4.1 Command Interface vrs. Basic Scripting

Those pins can be controlled using Basic Scripting (described in this article) or the Command Interface. Choose basic scripting option if the Avisaro Module should perform stand alone functions.

8.4.2 Setting and querying I/O ports

To set an I/O port, use the 'put' command. To read an I/O port, use the 'get' command. There is also the 'setleds' and the 'key' command. Do not use those commands- they are there for compability reason only.

There is no need to declare a port as input or output port. By performing a put command, this port is declared as an output port. By performing a get, this port is declared as an input port. By default, a port is an input port.

This script listing is a simple example of how to use I/O ports within Basic. The script turns and off Pin 2 . It reads in Pin4 and sets Pin 3 the same :

```
' Script to set and read ports
' for demonstration purpose

let in = 0          ' declare variable in
do
   put -202, #1     'pin 2: on
                    'pin 2 is often the red led
                    'sleep 1 second
   sleep 1000
   put -202, #0     'pin 2: off
   sleep 1000
   get -204, in
   put -203, in     ' set pin 3 same as
                    ' input pin 4
loop
```

8.4.3 Details

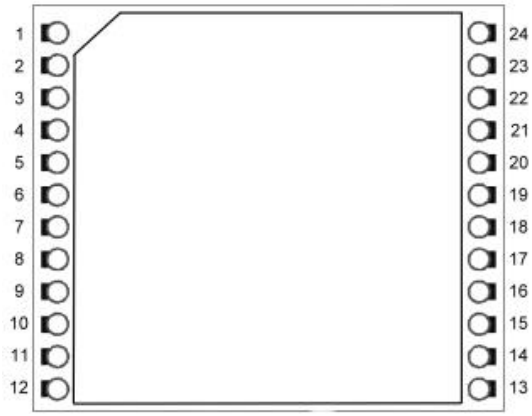
- There is no need to declare a port as I/O port. Simply use the port.
- A new port command overwrites a previous configuration. Thus, if a pin was used for data communication, a port command can interrupt this functionality
- Make sure to follow the electrical rules pins

8.4.4 Pin layout

I/O, PWM, Analog Ports

If a pin is not used for data transmission, this pin can be used as a input, output, pulse width modulation (PWM) or analog pin as described here:

No	I/O	Name		Name	I/O	No
1	P	VBAT		VCC (3.3V)	P	24
2	I/O	GPIO		n.c.	-	23
3	I/O	GPIO		n.c.	-	22
4	I/O	GPIO		n.c.	-	21
5	I/O	GPIO, PWM		n.c.	-	20
6	I/O	GPIO, PWM		n.c.	-	19
7	I/O	GPIO, PWM		n.c.	-	18
8	I/O	GPIO, ADC		n.c.	-	17
9	I/O	GPIO, PWM, ADC		I/O (open drain)	I/O	16
10	I/O	GPIO, PWM		I/O (open drain)	I	15
11	I/O	GPIO, PWM		Reset	I	14
12	I/O	GPIO		GND	P	13



'I/O': pin is input or output or pwm or analog input (ADC), 'p' power, 'n.c.': do not connect, leave open

9 DOING THINGS ...

9.1 WORKING WITH TCP CONNECTIONS

This chapter describes how to open and manage a TCP connection using the BASIC scripting. For example, an external sensor only sends data but can't be modified to send commands to handle a connection.

See (Mehr here) to get details on how open and manage a TCP connection using commands send from an external device such as a micro controller.

9.1.1 TCP/IP fundamentals

This document assumes you are familiar with TCP/IP fundamentals. For further information on TCP/IP please search the internet or follow the external links:

- Wikipedia: TCP/IP
- PDF state diagram

9.1.2 Open TCP connections

A TCP connection can be established two ways:

1.) Waiting for a incoming connection (Avisaro is TCP server)

Use the command "LISTEN" to create a 'TCP socket' in listen mode. Specify an internal 'handle' number - this handle number addresses this TCP socket. Also specify a TCP port number.

2.) Connection to a server (Avisaro is TCP client)

Use the text command "CONNECT" to actively connect to a TCP server. Specify an internal 'handle' number, the other TCP address and port number.

Once the TCP socket with its handle number is declared, you can query the module whether the TCP connection was established successfully:

9.1.3 Control TCP connections

Using the "STATUS" command it can be verified whether or not a connection was established. A return value of '9' for example means a successfully established connection.

9.1.4 Sending and receiving data

Once the connection is established, data can be exchanged using the "PUT" and "GET" command. One TCP packet can hold 1452 bytes of data. Ideally, an array with this length should be used for sending and receiving data.

Sending data with PUT

The PUT command is used to send data. The handle number specifies which connection to use since more than one TCP connection can be handled. As data to send, a string variable or an array can be

given as a parameter. Remember to check the dummy variable 'LASTERR' after sending since transmitting data can fail. See command description for details.

Receiving data with GET

The GET command is used to receive data. It is advised to define an array with 1500 Bytes as the receiving array - otherwise a received tcp/ip packet does not fit. See command details.

9.1.5 Closing TCP/IP connection

When done with sending or receiving data, the connection can be closed. Use the "CLOSE" command to perform this function.

9.1.6 Monitoring TCP/IP connection

While receiving and transmitting data, the TCP/IP connection can be interrupted. Reasons could be the loss of wireless connection or the other party was switched of or terminated. The command "STATUS" allows to monitor the connection and reaction properly to a change in connection status.

9.1.7 Example: HTTP Get to Google

Even embedded systems can connect to a server in the Internet. This example shows how to connect to google using Scripting:

```
' HTTP Get Beispiel
' 2008/08/12
' Avisaro AG

' Enter Port and target here
let p = 80
let t$ = "www.google.de"

outmode -2

let g = 0
dim b(1500)

print "reading "; t$ ;
print " on port " ;
print p

START:

let g = resolv(t$)
sleep 1000

connect 101, g, p, 5

sleep 1000

let y = status(101)

if y = 9 then
   ' connected
   print "connected"

   let a$ = "GET /search?hl=de"
   let a$ = a$ + "&q=avisaro"
   let a$ = a$ + "&meta= HTTP/1.1"
```

```
   let a$ = a$ + chr$(13) + chr$(10) + chr$(13) + chr$(10)

   put 101, a$

   GET_RESPONCE:

   sleep 100

   get 101, b

   close 101

   if (BYTESREAD > 0) then
      ' Ausgabe der Anwort über RS232
      for n = 0 to (BYTESREAD-1)
         print chr$(b(n));
      next n
   end if

else
   print "connection failed"
   ' no connect, do something else
end if

close 101
print "end"
```

9.2 WORKING WITH UDP DATA STREAMS

UDP is a "connectionless" way to exchange data. It is very simple to set up and operate, however data are not protected by acknowledges.

Using UDP sockets

A UDP socket is opened using the "UDPOPEN" command.

UDP is a packet oriented format. One UDP packet can hold 1460 bytes. Thus an array with this length should be ideally defined for receiving and sending data.

Some details:

- If an array shorter than the max lenght is used during a 'get' command, the left over data from that packet are thrown away.

9.3 WORKING WITH DATA FILES

The storing of the data is regularly managed by the script. Only in cases when the script is deactivated and the device is totally ruled by the command interfaces, commands via command interface can be placed. For the command interface please see the referring chapter. Using the script data can be handled variable in terms of format, selection, conjunction, number and time of files or other issues like timestamp etc. Here you can find some examples of changes in the scripting:

RS232: adding line feed after each dataset

With RS232 the first questions how to define the end of a dataset:

- either there is a character send only at the end of each dataset or
- you define a pause as a end of dataset

In this example we use the second criteria defining a pause of 100 ms as the trigger for a line feed. The duration of the break can be defined variably. The red command are the changes necessary to implement the line feed.

```
let ms = 0
do
' Read data and write to file
INPUT A
if BYTESREAD > 0 then
put -202, #0
put 1, A, BYTESREAD
let ms = millis + 100
end if
if (millis > ms) and (ms > 0) then
put 1, #13
put 1, #10
let ms = 0
end if
```

9.4 STORE AND READ USER DATA

Some applications require to store data i.e. for configuring an attached device, print messages to a screen or to recall ip addresses to connect to. The simplest way to store data within a BASIC script is to do a hardcoded line such as ' let ip$ = "192.168.0.3" '. If there is more data to store, or data needs to be changed without modifying the script, there other ways:

9.4.1 Command: DATA - READ - RESTORE

These set of commands allow to store a number of data within a BASIC script. This is particular usefull if i.e. config data needs to be send to device or messages need to printed to a display. Data stored using this method are more or less 'hardcoded' and always come with the script itself (it depends on the application whether or not this is of advantage or not).

9.4.2 See DATA - READ - RESTORE for details on those commands.

The number of data which can be stored using the data command can be changed. By default it is 1kByte. See MEMCFG on how the modify memory allocation.

9.4.3 Command: SAVE - LOAD

These set of commands allow to store a number of data permanently in the Avisaro device. This is particular usefull for configuration data (i.e. IP adress to connect to) or temporary user data such as a history of events.

This storage are can be used also to store data to be send to a connected device for configuration purpose. A little 'helper' script or a special function (... if file exist ... copy content to this store area ...) can be used to preload this storage area. Once loaded, the script can access the data within this area.

This storage area is non-volatile. Its size is 4kByte - so quite some data fit into this area. Unlike the other method, this area is fixed in size.

See SAVE - LOAD for details on those commands.

9.4.4 Files: Store data on SD card

Last but not least, data can be stored on SD card - if a memory card slot is available. Storage space is almost unlimited, however data are only available if a SD card is inserted. See (Mehr here) for details.

10 Optimization Hints

Some applications need to optimize the setup. The marginal values can be increased increasing the performance or concentrating all resources to this application.

10.1 General thoughts

"Performance" depends on the type of application. In general, performance is separated into "throughput" and "delay". Applications like data logging tend to required high throughput, were as voice type application a short delay time require.

10.2 Delay considerations

The Avisaro 2.0 system contains a multitasking architecture. The TCP/IP Stack, WLAN driver and BASIC Scripting run in separate tasks. The multitasking has two different modes:

Time sliced mode: Each task gets a defined timeslot assigned. That is independent of whether the task has something to do or not. As a result, the system is very predictable how it reacts to input.

Work load mode: Each task gives up computing resources as soon as there is nothing to do. To avoid a blocking system, there is a maximum time of 20ms for each task. As a result, the system reacts more dynamically to the inputs but is less predictable.

Time Slice Mode

Use the command "sched" to configure the length of the timeslot each task should be given. A value of 20 ms ("sched 50" : 1sec/50 = 20ms) is a good starting point.

Work Load Mode

Use the command "sched" to switch on the Work Load Mode ("sched 0").

It is advised to tell the BASIC Script when to give up computing resources. This is done with the command "sleep 0". Typically this command is inserted after sending a data packet:

```
repeat_put:
put 201, C, BYTESREAD
if LASTERR <> 0 then
   sleep 0
   goto repeat_put
end if
```

There is a 'safety net' - if no sleep 0 is used, every about 20ms the BASIC Scripting task is switched anyway. This is to avoid a 'hanging' system: If BASIC would not free computing resources to the TCP/IP task, no network traffic could be done.

The overall result is a system with short delay times.

10.2.1 Windows PC: Delayed Ack

When running a Windows PC, there is problem called "Delayed Ack".

10.2.2 Throughput considerations

Stay tuned ... will be discussed later.

11 ARITHMETIC OPERATORS

11.1 BITWISE OPERATORS

Bitwise operators operate on 32 bit signed integer values.

The Avisaro Scripting Language supports these four bitwise operators:

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.

11.1.5 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

11.2 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.

11.2.1 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
```

11.3 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 nor 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
```

11.4 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.

12 LIST OF ALL COMMANDS

12.1 ABS

	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.

12.2 ASC

	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.

12.3 AUXOPEN

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 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 = Clock is active high 1 = active low	
4.	# of Stop bits (1,2)	0 = master 1 = slave	Filter (upper ID)	Clock phase 0 = Data is sampled on first closk 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 active low 1 = 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	

Example

```
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 ouput 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.

12.3.1 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
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
Defines the number of stop bits for the connection. Allowed values are 1 and 2.

5.) Bits per Character or special CAN features
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 hadshake (XON/XOFF). You can use the ASC function to convert a character into that value

12.3.2 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.

12.3.3 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.

12.4 BIND

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.

12.5 BYTESREAD

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.

12.6 CANCSV

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, accessingCANCSV$ without a previous successfull call to GETCAN should be avoided. Internally, CANCSV$ generates CSV strings of variable length, depening 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.

12.7 CANINFO

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 beween 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 GETCAN page.

12.8 CHR$

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.

12.9 CLOSE

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 woks 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.

12.10 CONNECT

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 tranfer 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 woks

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_FIL_EXHAUSTED)
if the system has too few ressources

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, resolv ("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.

12.11 CSV$

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.

12.12 DATA-READ-RESTORE

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.

12.13 DATE$

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.

12.14 DIM

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.

12.15 DO…LOOP…UNTIL

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 cane, the enclosing block is executed over and over. To get out of an endless loop use the GOTO statement. If there's an UNTIL imediately 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.

12.16 END

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.

12.17 EXEC

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.

12.18 FLOAT$

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).

12.19 FOR…NEXT…TO…STEP

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.

12.20 FREEMEM

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.

12.21 GET

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.

12.22 GETCAN

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 headerThis 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 an exits.

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

See next paragraph for the CAN data format

12.22.1 CAN internal 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	Real time clock time stamp in seconds since 01.01.2007
22	15...8	
23	23..16	
24	31..23	
25	7...0	Millisecond time stamp. Value in ms since module power on.
26	15...8	
27	23..16	
28	31..23	

12.23 GETSCAN

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 broadcasts 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 Toshiba_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

12.24 GOSUB…RETURN

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.

12.25 GOTO

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 concidered 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.

12.26 GSM

GSM

Description

The GSM command makes is possible to submit AT- and USSD requests to the integrated GSM modem. GSM requires two arguments, a string (the request) and a string variable that receives the answer.

Example

```
outmode -2


let b$ = ""
gsm "*101#", b$
if LASTERR = 0 then
   print "'USSD *101# answered with: ";
   print b$
else
   print "USSD error !!!"
end if


let b$ = ""
gsm "AT+CPIN?", b$
if LASTERR = 0 then
   print "'AT+CPIN? answered with: ";
   print b$
else
   print "AT Command error !!!"
end if
end
```

Remarks

The GSM command distinguishes by the first character of the first argument if it should spawn an AT or USSD request. If this character is 'A' or 'a', then the string is interpreted as AT-command, otherwise as USSD request. LASTERR ist set to 12 (Rejected) if an error occured.

GSM is available on firmware version 5.06 and above. The module must have as GSM modem.

12.27 HEX$

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.

12.28 HSET

HSET

Description

The HSET command exists to change various settings of e.g. files, sockets, communication ressources at runtime.
HSET requires three arguments. The first one is a handle number that must either refer to an open handle, a file or socket, or it can be a magic number that identifys a communication ressource, such as -4 for the second RS232 interface.

Using HSET on open sockets

If the first one is a handle, the second 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.

Using HSET to set buffer size of communication channels

If the first one is a magic, negative number, the second argument denotes the size of the receive buffer, and the second one the size of the transmit buffer of a communication channel. Currently, these magic numbers are defined:

-4 The second RS232 interface (RX and TX buffer)

-5 The IIC interface (RX and TX buffer)

-8 The second CAN interface (RX buffer only)

-10 The SPI master channel (RX buffer only)

The maximum size of any buffer is 4096, the minimum size is 2. Values outside this range are forced to the next valid value. For interfaces that have only one buffer, the other value does not have any effect.
To set the buffer size properly, HSET must be called before AUXOPEN is called.

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
```

Sets the RX FIFO of the auxilliary RS232 to 1024 and the TX FIFO to 32 bytes

```
HSET -4, 1024, 32
```

Remarks

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

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.

12.29 IF…THEN…ELSE

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
                pint " 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.

12.30 INMODE

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 OUTMODE command.

12.31 INPUT

INPUT

Description

The INPUT command reads data from either the currenrt 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.

12.32 INSTR

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.

12.33 KEYS

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.

12.34 KILL

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.

12.35 LASTERR

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 inconsitencies, *LASTERR* is set to a value other than 0 to report the problem to the program.
The system defines the error codes as shown in the next paragraph.

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 later evaluation.

12.35.1 Error codes and numbers

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

12.36 LCASE$

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

Remarks

Internally, the C library function "tolower is used". See also UCASE$.

12.37 LEFT$

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$ = "hello world"
b$ = left$ (a$, 5)
print a$
print b$
end
```

Remarks

See also RIGHT$ and MID$

12.38 LEN

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$ = "hello"
print len (a$)
end
```

Remarks

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

12.39 LET

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$ = "hello"
LET b = 1
LET a$ = "world"
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.

12.40 LISTEN

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 occured. 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 comma

12.41 LOAD

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. Adresses above 4095 will wrap around to x mod 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 sboth 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. Adresses above 4095 will wrap around to x mod 4096.

See also: SAVE command.

12.42 LOC

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 so 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

12.43 LOF

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.

12.44 LTRIM$

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$.

12.45 MEMCFG

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 exceeed 12288 bytes.

If MEMCFG is not used, the default configuration is (extended firmware):

Heap size: 16.384 Bytes
Code segment: 6.144 Bytes
Data segment: 2.048 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.

12.46 MID$

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$

12.47 MILLIS

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.

12.48 MOVE

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

12.49 OPEN

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!"
end if
```

Remarks

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

12.50 OUTMODE

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.

12.51 PRINT

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 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 an the next PRINT statement continues in the same line. PRINT only works if the OUTMODE is correctly set.

Example

This little example shows some PRINTed 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.

12.52 PUT

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 Interface	Currently selected data interface (synchronous)	
-3	Data Interface	Currently selected data interface (asynchronous). Async. mode means, that PUT always send data and does not wait for the receiver to acknowledge it.	
-4	RS232	Auxiliary (= 2nd) RS232 interface (asynchrounous)	
-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 PUTs third argument must be a multiple of 28. The array is seen as one ore more consecutive CAN	> 3.36

		frames. For a description of the structure of those CAN frames, please see the GETCAN and PUTCAN commands.	
-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 lenght 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 occured.

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 explicitely 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 PUTs 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

12.53 PUTCAN

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 lenght 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 befor 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.

12.54 PWM

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. For pin number assignments, please see here: click.
2.) The second argument is the length of the pause in 1/2 µs units.
3.) The third argument is the length of the pulse, also in 1/2 µs 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 1µs, call PWM with 1 and 1 as second and third argument. Unfortunately, the pulse width for an 1µs 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 1µs, 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.

12.55 READSMS

READSMS

Description

With the READSMS command, a BASIC skript can read SMS (Short Message Service) messages that are stored onto the SIM card of the GSM-Modem. READSMS requires two string variables as arguments. After a successfull call to READSMS, the first variabe contains the sender's address (mobile phone number), while the second variable contains the message text.

Example

```
            outmode -2
            let a$ = ""
            let b$ = ""
            readsms a$, b$
            if LASTERR = 0 then
               print "Got SMS from: ";
               print a$
               print "Text: ";
               print b$
            end if
            end
```

Remarks

On reading, a SMS-message is deleted from the SIM-card so it cannot be read twice. Messages are read in time-reverse order, that means the youngest message is read first. If there are no more messages stored, both strings will contain "NONE" and LASTERR is set to 8 (NO_DATA).

The maximum length of an incoming message is the default string length (of 255 bytes). Longer messages are truncated.

READSMS is available on firmware version 5.06 and above. The module must have as GSM modem.

12.56 REM

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.

12.57 RESOLV

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 = resolv ("www.avisaro.com")
if a = 0 then
   print "could not resolve domain name"
else
   print a
end if
let a = resolv ("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.

12.58 RIGHT$

RIGHT$

Description

RIGHT$ is a string function that can be used to extract the rightmost charcacters of a given input string. RIGHT$ needs a source string and the number of characters which shall be extracted. It then constucts 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 charcters 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$ ans LEFT$

12.59 RTRIM$

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$.

12.60 SAVE

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.

12.61 SCAN

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 it's 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 wlans: ";
' 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.

12.62 SCANNED

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

12.63 SEEK

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 furthermost position that can be reached with SEEK is at 2,147,483,647

12.64 SENDSMS

SENDSMS

Description

With the SENDSMS command, a BASIC skript can send SMS (Short Message Service) messages over the GSM network to other SMS-capable stations. SENDSMS requires two strings as arguments. The first one is the address (phone number, e.g.) of the recipient, while the second one is the message text.

Example

```
outmode -2
sendsms "017636175395", "Hello_from_avisaro_module"
if LASTERR = 0 then
   print "SMS transmitted successfully"
end if
end
```

Remarks

LASTERR is set to 12 (Rejected) if the SMS could not be sent. SENDSMS is available on firmware version 5.06 and above. The module must have as GSM modem.

12.65 SETCAN

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

12.66 SETLEDS

SETLEDS

Description

There is a new structure to control I/O ports more effectively. See Mehr 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.

12.67 SLEEP

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 algoritm of the RTOS. The Scripting Language can't specify that.

12.68 STATUS

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

x	Number of free packet buffers

Auxiliary IIC Port, -15

x	Number of Bytes in TX Fifo

If IIC is used in Slave mode, Incomiing 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.

12.69 STR$

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$
```

Remarks

+ is the strings concatenation operator.

12.70 TAB

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.

12.71 TIME$

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.

12.72 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.

12.73 UCASE$

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$.

12.74 UDPOPEN

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.

12.75 VAL

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.

12.76 WHILE…WEND

WHILE…WEND

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.

13 APPENDIX

13.1 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

13.2 DOKUMENTIERTES SCRIPT

Sorry – this is in German:

Der dokumentierte Source Code soll einen grundsätzlichen Einblick in die Programmierung von Scripten geben. Das aktuelle Script kann ggf. von dieser Doku etwas abweichen.

Script "MR3" mit Kommentaren

Das Script-Beispiel führt Befehle für eine RS232-Schnittstelle aus. Dei anderen Box-Typen verfahren Sie den Schnittstellen entsprechend.

Kommentare werden mit ' oder mit REM markiert:

```
'
' Datenlogger Rev 1.8 (c) Avisaro AG, 14.1.2009
'
```

Mit DIM wird eine Array definiert, dass später verwendet wird, um Daten zu halten. Für Dateioperationen ist eine Größe von 512 geeignet.

```
DIM A(512)
```

Falls die interne Uhr beim Avisaro Modul nicht batteriegepuffert ist, wird sie im Folgenden auf einen Default-Wert gesetzt. Bei der Avisaro Box und Cube ist dieser Schritt nicht notwendig. Zusätzlich werden ein paar weitere Konfigurations-Einstellungen gemacht - diese könnten genau so gut außerhalb des Scripts mit einer 'autorun.txt' Datei gemacht werden.

```
'if no battery
if time < 10000 then
   exec "time 2009 01 01 00 00 01"
end if


exec "fsync 1000"
exec "sched 0 fix"
```

Das Scriptbeispiel benutzt eine RS232 Schnittstellen. Die erste wird über die "Data Interface" Einstellung festgelegt. Mit dem 'inmode' und 'outmode' Befehl wird festgelegt, dass die Daten vom Data Interface zum Script verarbeitet werden (und somit nicht als Kommandos interpretiert werden).

Für die einfache Ausgabe von z.B. Startmeldungen kann 'print' verwendet werden.

```
print "Avisaro Logger Rev 1.8 (c) 2009 Avisaro AG"
```

Sprungmarke (Anker) um an den Anfang des Programms zu springen (für die spätere Verwendung des Befehls 'goto')

```
BEGIN:
```

Typischerweise ist am IO-Pin 3 eine grüne LED angeschlossen. Diese wird nun eingeschaltet.

Wenn die Taste (typischerweise an I/O-Pin 4) gedrückt ist, wird wieder zum Anfang gesprungen. Später im Programm wird eine gedrückte Taste zum Stop der Aufzeichnung verwendet - hiermit wird nun sichergestellt, dass die Taste auch wieder losgelassen wird.

```
if (KEYS & 1) = 1 then
   goto BEGIN
end if
```

Nun wird überprüft, ob ein Datenträger eingelegt worden ist. So lange die Speicherkapazität ("lof") den Wert 0 hat, ist dies nicht der Fall und es wird nach einer kurzen Pause ("sleep") zum Anfang zurückgesprungen.

```
if lof(0) = 0 then
   sleep 100
   goto BEGIN
end if
```

Ist ein Datenträger eingelegt, wird versucht die Datei "log-1.txt" in dem Modus "Daten Anhängen" zu öffnen. Falls also die Datei schon vorhanden ist, werden Daten angehängt. Schlägt dies fehl ("LASTERR") ist die Datei nicht auffindbar und wird dann neu erzeugt. Schlägt auch das fehl, ist etwas nicht in Ordnung (z.B. Speicherkarten wurde wieder herausgenommen) und es wird zum Anfang gesprungen.

```
open "AB", 1, "log-1.txt"
if LASTERR <> 0 then
   open "WB", 1, "log-1.txt"
   if LASTERR <> 0 then
      close 1
      goto BEGIN
   end if
end if
```

Bei erfolgreichem Abschluß soll nun die rote LED (meistens I/O-Pin 2) anschalten um zu zeigen, dass nun aufgezeichnet wird. Die aktuelle Zeit in Sekunden wird gespeichert - so kann später die LED im Takt blinken, wenn Daten eintreffen. Mit der 'DO - LOOP' Schleife beginnt nun die Hauptschleife.

```
   put -202, #1
   let t = time
do
```

Mit dem Input Befehl werden nun die Daten von der ersten RS232 abgeholt. Es werde so viele Daten wie verfügbar in das Array A gepackt. Mit der Systemvariablen BYTESREAD kann abgefragt werden, wie viele Daten gelesen wurden. Wurden Daten gelesen (">0"), dann wird die rote LED ausgeschaltet und die Daten werden in die Datei mit dem Handle 1 geschrieben.

```
INPUT A
if BYTESREAD > 0 then
   put -202, #0
   put 1, A, BYTESREAD
end if
```

Im Folgenden wird laufend überprüft, ob die Speicherkarte weiterhin eingelegt ist (mit "lof"). Kommt es zu einem Schreibfehler - z.B. der Datenträger ist voll, dann wird die Datei automatisch

geschlossen - somit ändert sich auch der Status des Dateihandles. Ist dies der Fall, werden vorsorglich beide Dateien geschlossen, die LED ausgeschaltet und zum Anfang gesprungen.

```
      if lof(0) = 0 or status(1) <> 2 then
         close 1
         close 2
         put -202, #0
         goto BEGIN
      end if
```

Wurde die Taste gedrückt, werden die Dateien geschlossen und die Aufzeichnung beendet. Es wird dann zur späteren Marke "FIN_KEY" gesprungen.

```
      if (KEYS & 1) = 1 then
         close 1
         close 2
         goto FIN_KEY
      end if
```

Aus rein optischen Gründen lassen wir die rote LED mit etwas Verzögerung wieder leuchten, wenn sie nach dem Eintreffen von Daten ausgeschaltet wurde.

```
      if t < time then
         let t = time
         put -202, #1
      end if
   loop
```

Bei "FIN_KEY" wird nur noch gewartet, bis die Taste wieder losgelassen wurde. Bei "FINISH" wird nun auf eine weitere Benutzeraktion gewartet. Damit die Aufzeichnung wieder startet, muss entweder erneut die Taste gedrückt werden oder die Speicherkarte entnommen werden.

```
   FIN_KEY:
      if (KEYS & 1) = 1 then
         goto FIN_KEY
      end if


   FINISH:
      put -202, #0
      let x = lof(0)
      if (x = 0) or ((KEYS & 1) = 1) then
         goto BEGIN
      else
         goto FINISH
      end if
```

Zum Schluss nur noch ein Goto zum Anfang - eigentlich nicht nötig, aber zur Sicherheit. Das "' +++" dient lediglich dazu, dass wenn die Datei über die Datenschnittstelle hochgeladen wird, wird damit das Ende angezeigt. "+++" ist dabei die - veränderbare - Stopsequenz.

```
   goto BEGIN

   ' +++
```