Click analyzer protocol for developers

OVERVIEW

Click analyzer offers a variety of input and output protocols that are best for WEB applications (AJAX, python etc.). Direct human interactions and M2M applications. For best convenience Click analyzer offers facilities for auto-detection and firmware update.

Click analyzer offers 2 communications protocols:

1) Webmode and binary mode inputs for m2m applications
2) XTERM mode, including mouse interaction for HMI

Click analyzer offers 3 different output protocols:

1) Webmode (default) – JSON formatted data output
2) Terminal mode – minimalistic GUI for direct human interface for ANSI terminals
3) Binary mode – For easiest M2M communication with highest data throughput

Based on baud-rate selected for communication, the board offers 3 modes of operations:

1) Baud ≤ 4800 – Start XBOOT bootloader
2) 4800 < Baud ≤ 115200 – Reset analyzer and send welcome message
3) 115200 < Baud – Connect to the device in current state

GENERIC RULES FOR COMMUNICATION

The PC to Analyzer command rules:

1) Input text in UPPERCASE
2) A # character in the input stream causes immediate reset (see The device reports unrecognized or bad commands and incorrect parameters)
3) Commands are terminated with <separator commands> character (see)
4) Command arguments are separated with <separator parameters> character (see COMMANDS)
5) Some command arguments have one numerical parameter assigned with <assign number> character (see COMMANDS)

Example input commands

```
COMMANDS;
GET LED_INFO;
LS FREQ=100K NUMSMP=100;
```

Generic remarks about response:

- The device reports unrecognized or bad commands and incorrect parameters
- If negative response sent, command fully rejected
- Based on communication mode, the response content is different

MESSAGE FORMATS

Autodetecting message format

- JSON format always starts with "{" <0x7b> character
- Binary messages never start with "{" <0x7b> or "<ESC>" <0x1B>
- ANSI messages are not binary and not JSON formatted ("else")

JSON format

When the communication mode is set to JSON, the input commands has to obey the standard rules for input commands (e.g. all uppercase, proper message termination, etc.)

All responses from the device are properly formatted JSON responses. Care should be taken for JSON readers to stop parsing the response with the first invalid JSON character, and pass the already received data to the JSON parser. (Needed for mode auto-detection).

All responses may be combined into a global response structure, with every command overwriting only the structure elements it is reporting.

The JSON communication mode is activated with the SET command.

BINARY format

Example for activating JSON mode

```
SET OUTPUT JSON;
Response:<none>
```

When the communication mode is set to BIN, the input commands has to obey the standard rules for input commands (e.g. all uppercase, proper message termination, etc.)

The binary format encapsulates responses in a CRC protected response structure. The response payload can be JSON formatted (for settings, command list, help, etc.) or pure binary data (e.g. logic readouts, voltages, etc.)

Binary message generic format

Example for activating the BINARY mode

```
SET OUTPUT BIN;
Response:<none>
```

The binary message starts with a CRC16 value, followed by payload identification word and

payload length word. The actual payload bytes can be 0...65535 bytes. (In version 1.0, the maximum length of payload is 1100 bytes) Words are little-endian (Intel) format.
E.g. (bytes stream <FF><00> is 255 decimal)

Header			Payload <max 65535 bytes>
CRC16 <2bytes>	payloadID <2bytes>	payloadLen <2bytes>	

The CRC16 is calculated with x16 + x12 + x5 + 1 (0x1021) polynomial and 0xFFFF seed values. Many tools call this CRC16-CCITT-FALSE. You can test your algorithm results with the following online tool: ***https://www.lammertbies.nl/comm/info/crc-calculation.html***

If the calculated CRC16 message LSB is 0x7b or 0x1B then extra 0 is fed in CRC calculation. This is necessary to maintain autodetect compatibility.

Example pseudo code for CRC16

```
while ((CRC16 & 0x00FF == 0x7B) || (CRC16 & 0x00FF == 0x1B)) {
 CRC16 = CRC16_add(CRC16, 0);
}
```

The augmented CRC16 calculation does not affect the payload length or the payload itself. The zero padding happens only during CRC16 calculation.

For python the recommended CRC function is:

```
crcmod.predefined.mkPredefinedCrcFun('crc-ccitt-false')
```

Example of a zero-length payload message – easy CRC test

```
Invalid msg;
Response: CC 74 21 21 00 00
- CRC16 = 0x74CC (verify online look at: CRC-CCITT (0xFFFF) )
- payload ID = 0x2121 (NAK response)
- Payload length = 0x0000 (no payload)
```

BIN encapsulated JSON

Example of a message

```
LED;
Response: E4 05 47 54 3A 00 7B 22 70 69 6E 73 22 3A 7B 22 4C 45 44 22 3A 7B 22 59 45 4C 4C 4F 57 22 3A 30 2C 22 4F 52 41 4E
47 45 22 3A 30 2C 22 47 52 45 45 4E 22 3A 30 2C 22 52 45 44 22 3A 30 7D 7D 7D
- CRC16 = 0x05E4 (verify online look at: CRC-CCITT (0xFFFF) )
- payload ID = 0x5447 (Encapsulated JSON formatted response)
- Payload length = 0x003A (72 bytes payload)
- Payload = <0x7b><0x22><0x70><0x69>… … … <0x30><0x7D><0x7D><0x7D>
```

Many responses are formed as JSON, encapsulated in BIN header. This help maintaining universal information exchange among firmware versions.

When receiving and encapsulated JSON message:

- First verify the CRC of the full datastream
- Verify Payload ID, it is 0x5447
- Strip the header and pass the full payload to a JSON parser

ANSI AND XTERM FORMAT

The HMI format input and output are user friendly and comes in a separated document. The requirements for this format is an ANSI / XTERM compliant terminal software and the user actions are guided onscreen. This mode is not recommended for M2M applications or GUI implementations.

Welcome message

Example for entering the HMI modes

```
SET OUTPUT ANSI;
Response:<terminal autodetect sequence>
```

After each reset the device automatically sends out a welcome message.

The resets happen:

1) After power-up
2) After exiting from bootloader mode
3) After connecting to serial port with 4800 < Baud ≤ 115200
4) When # character is received on the UART

The beginning of the welcome message is JSON formatted followed by a terminal autodetect sequence. Most JSON parsers will report an error if the whole data stream is provided for parsing due to the binary content on the end. For JSON parsing best practice is to split the message at the first character with ASCII value below 0x20 [<space>]

Example welcome message

```
Response
as ASCII: {"commandline":{"separator_commands":";"}}<ESC>[5n

as HEX: 7B 22 63 6F 6D 6D 61 6E 64 6C 69 6E 65 22 3A 7B 22
73 65 70 61 72 61 74 6F 72 5F 63 6F 6D 6D 61 6E 64 73 22 3A
22 3B 22 7D 7D 1B 5B 35 6E
```

Command reference

In the commands reference only the JSON and BIN format responses are discussed. All JSON responses are designed to be subject to DEEP_MERGE.
[Merging information from all responses into one structure – on the host PC]

COMMANDS command

This function is used to discover communication parameters and all available commands. The command list also specifies help command for each. The help command gives details about a specific command, including description, and parameters.

JSON mode: JSON object

BIN mode: 0x5447 (Encapsulated JSON formatted response see)

Understanding the response (reformatted for easier reading)

Example COMMANDS JSON content

```
{"commandline":
  {"separator_commands":";","separator_parameters":" ",
"assign_number":"="},"commands":{
"COMMANDS":{"details":"GET COMMANDS_INFO"},
"GOTOBOOTLOADER":{"details":"GET BLDR_INFO"},
"LS":{"details":"GET LS_INFO"},
"LED":{"details":"GET LED_INFO"},
"DVM":{"details":"GET DVM_INFO"},
"GET":{"details":"GET GET_INFO"},
"SET":{"details":"GET SET_INFO"}}}
```


```
{
  "commandline":{
    "separator_commands":";",
    "separator_parameters":" ",
    "assign_number":"="
  },
  "commands":{
    "COMMANDS":{
      "details":"GET COMMANDS_INFO"
    },
    "GOTOBOOTLOADER":{
      "details":"GET BLDR_INFO"
    },
    "LS":{
      "details":"GET LS_INFO"
    },
    "LED":{
      "details":"GET LED_INFO"
    },
    "DVM":{
      "details":"GET DVM_INFO"
    },
    "GET":{
      "details":"GET GET_INFO"
    },
    "SET":{
      "details":"GET SET_INFO"
    }
  }
}
```

The command line field: Commands are understood based on these parameters. The command separator, assignment and parameter separator are fixed only in a particular firmware version. To be universally compatible to future firmware revisions, you need to for the communication according to these parameters.

Example input command forming

```
<welcome message>{"commandline":{"separator_commands":";"}}… …
<to analyzer>COMMANDS;
<Response>{"commandline":{
      "separator_commands":";",
      "separator_parameters":" ",
      "assign_number":"="
```

The commands field:
Lists all command that are accepted by the analyzer. Invoking the filed details as command will revel detailed information about parameters, etc.

Example get command details

```
<to analyzer>COMMANDS;
<Response> … … …
      "GOTOBOOTLOADER":{
         "details":"GET BLDR_INFO"
      }, … … …
<to analyzer> GET BLDR_INFO;
<Response> {"commands":{"GOTOBOOTLOADER":{"description":"Start bootloader
session","parameters":{}}}}

Meaning: GOTOBOOTLOADER command will Start a bootloader session, no parameters
accepted.
```

From the example above, deep merging the information gives the following structure:

```
{
  "commandline":{
    "separator_commands":";",
    "separator_parameters":" ",
    "assign_number":"="
  },
  "commands":{
    "COMMANDS":{
      "details":"GET COMMANDS_INFO"
    },
    "GOTOBOOTLOADER":{
      "details":"GET BLDR_INFO",
      "description":"Start bootloader session",
      "parameters":{}
    },
    "LS":{
      "details":"GET LS_INFO"
    },
    "LED":{
      "details":"GET LED_INFO"
    },
    "DVM":{
      "details":"GET DVM_INFO"
    },
```

```
    "GET":{
      "details":"GET GET_INFO"
    },
    "SET":{
      "details":"GET SET_INFO"
    }
  }
}
```

GOTOBOOTLOADER command

This command is used to invoke to bootloader. The device is reset, and the bootloader takes action. If no bootloading done, the device will restart (details in bootloader description)

Example for entering bootloader

```
<to analyzer>GOTOBOOTLOADER;
<response>
Xboot:Gotoboot
Xboot:Version 1.0
Xboot:Wait XMODEM>
CCCCCCCCCCCCCCC
```

LS command

Logic scope. This function will sample all digital inputs at FREQ sample rate and return result after NUMSMP samples gathered. The NUMSMP parameter range is dependent on output mode. The actual number of samples are returned by invoking commnds.LS.details command. After deep-merging the commands.LS.parameters.NUMSMP.values.range filed will specify the maximum number of samples in [min,max,scaledmin,scaled_max] format.

The sampling of the Inputs are initiated by sampling time and two atomic samples are obtained from IO ports PORTB and PORTC (hw. Version 1.xx) This gives minimal delay at lower sampling rates and getting more important at high sample rates. Above 1MSPS the sampling delay of atomic reads is leading to delay between reads. The delay between reading PORTB and PORTC is approximately 125nsec. This could be omitted at low sample-rates (1MSPS and below)

The response format is also depending on the commands.LS.bytesPerSample field. This filed defines the number of bytes for a single sample. This parameter is dependent on the hardware.

The sampling frequency range is defined by commands.LS.parameters.FREQ.values.range field.

After completing the conversion, the command returns a response. Both in JSON and BIN format will return a pin-mask and raw data. The binary value of a single channel is calculated by simple formula:
<channel logic state> = (<pin-mask channel> & <raw datapoint>) > 0

Data format in JSON mode

The return value is an object with three fields:

LS.samplerate: returns the exact sample rate of the pins. (see details above about delay)
LS.pins: array, length is number of inputs sampled.
LS.pins[n]: pinmask for the channel. Ch1 is LS.pins[0], Ch2 is LS.pins[1] ... Chn+1 is LS.pins[n]
LS.data: array, length is number of samples collected. For every sample in time, only one integer is sent.

Example JSON data for logic scope

```
<to analyzer>LS FREQ=100K NUMSMP=10;
<response>
{
   "LS":{
      "samplerate":99976.000000,
      "pins":[512,64,128,2048,8192,256,1024,16384,4,2,32768,1,4096,8],
      "data":[144,144,144,144,144,144,144,144,144,144]
   }
}
```

Interpreting the response above

Sample time = $\frac{1}{99976.0\ Hz} = 10.002\ \mu sec$

at 0 sec PIN_3 (LS.data[0] & pins[2] > 0) (144 & 128 > 0) (=logic High)
at 10.002 µsec PIN_3 (LS.data[1] & pins[2] > 0) (144 & 128 > 0) (=High)
at 90.022 µsec PIN_3 (LS.data[9] & pins[2] > 0) (144 & 128 > 0) (=High)

Data format in BIN mode

The return value is encapsulated in a generic frame with CRC (see:)

The generic header payload ID is 0x534C (Logic Scope sample v1).

Reply structure:

Offset	Length	Name	Description
0	6 bytes	BINheader	Binary message generic format header
6	1 byte	pinmapEntries	# entries in pinmap
7	[pinmapEntries] bytes	pinmap	Power of two mask
7+pinmapEntries	[NUMSMP × commands.LS.bytesPerSample] bytes	data	Sampled data

Short code to get a certain pin state at a given sample:

```
bool getPinVal(const lsmsg_t *msg, pin, sampleid) {
        if (msg &&
        (pin>0) &&
        commands.LS.bytesPerSample > 0
        (msg->pinmapEntries <= pin) &&
        (msg->payloadLen>=(sampleid+1)*commands.LS.bytesPerSample+msg->pinmapEntries+1)) {
            //call parameters validated
        return (msg->data[sampleid] & (1<<(msg->pinmap[pin-1])));
        }
        return 0;  //Invalid queries are "low" or throw an exception...
        }
```

Example BIN data for logic scope

```
<to analyzer>LS FREQ=100K NUMSMP=10;
<response>
71 48 4C 53 23 00 0E 09 06 07 0B 0D 08 0A 0E 02 01 0F 00 0C 03 90 00 90 00 90 00 90
00 90 00 90 00 90 00 90 00 90 00 90 00
```

Interpreting the response above

- CRC16 = 0x4871 (verify online look at: CRC-CCITT (0xFFFF))
- payload ID = 0x534C (Logic Scope Data)
- Payload length = 0x0023 (35 bytes payload)
- pinmapEntries = 0x0E (14 pins mapped)
- pinmap[0] = 0x09 (Pinmask for pin1 = 2^9 = 512)

… … …

pinmap[2] = 0x07 (Pinmask for pin3 = 2^7 = 128)

… … …

- pinmap[12] = 0x0C (Pinmask for pin13 = 2^{12} = 4096)
- pinmap[13] = 0x03 (Pinmask for pin14 = 2^3 = 8)
- data[0] = 0x0090 (0x10+0x80=$2^{pinmap[2]}$+<unkonwnpin> → pin3 is high)
- data[1] = 0x0090 (0x10+0x80=$2^{pinmap[2]}$+<unkonwnpin> → pin3 is high)

… … …

- data[9] = 0x0090 (0x10+0x80=$2^{pinmap[2]}$+<unkonwnpin> → pin3 is high)

LED command

This function is for visualizing activity on a pin. The configured LED will be on for 100ms after the last pin state change. The pin state changes are monitored by hardware and will monitor extremely short pulses or long pulses. This function is only for visual readout from the board.

Adding LED to a 2MHz 1% PWM signal will show no light on the LED. The LED command will detect these pulses and turn on the analyzer LEDs for 100ms which is very well visible for humans. The continuous change on the pin will make the LED constantly on.

Changing the LED assignment will return the current assignment status. Issuing the command without parameters will return the current pin to LED assignment configuration.

If a LED is assigned for pin "0" it means LED is not assigned to a pin.

Data format in JSON mode: JSON object
Data format in BIN mode: 0x5447 (Encapsulated JSON formatted response, see)

If changing assignment, always all assignment status is returned. This function is also good candidate for DEEP_MERGE.

Example of getting LED assignments

```
<to analyzer>LED;
<response>
{"pins":{
    "LED":{"YELLOW":0,"ORANGE":0,"GREEN":0,"RED":0}
}}

Meaning LEDs are not assigned to any input pin.
```

Example of assigning pin7 change to RED LED

```
<to analyzer>LED RED=7;
<response>
{"pins":{"LED":{"YELLOW":0,"ORANGE":0,"GREEN":0,"RED":7}}}

Now all changes on PIN7 will be visible on RED LED.
```

Example of unassigning RED LED

```
<to analyzer>LED RED;
<alternative command>LED RED=0;
<response>
{"pins":{"LED":{"YELLOW":0,"ORANGE":0,"GREEN":0,"RED":0}}}
```

Parameter details are obtained by executing command from COMMAND.LED.details.

DVM

This function is for measuring a voltage on all pins quasi simultaneously. The DVM is mimicking the functionality of connecting 14 digital voltmeters (DVM) to all of 14 analyzer pins. DVM temporarily disables the LED feature while the command is active.

This command does not need parameters.

Data format in JSON mode

The return value is an object with one field:
DVM.voltages : The field value is list of voltages for each pin as an array.

Data format in BIN mode

Example of measuring floating pins

```
<to analyzer>DVM;
<response>
{"DVM":{"voltages":
[1.233796,1.450168,1.431576,1.451384,1.528016,1.547792,1.
540368,1.503296,1.588624,1.562656,1.644224,1.682560,1.759
216,1.853184]}}

This shows that the voltage on analyzer pin9 is 1.588624V
```

In binary mode you can get higher resolution (if needed) as the reference voltage and the RAW ADC data is sent. Number of channels and ADC resolution is also included in the data frame.

The generic header payload ID is 0x5644 (Digital Voltmeter sample v1).
vrefVoltage is a float24 formatted data. Float24 is a shortened IEEE 754 single precision format.

sign	Exponent (8bits)		Fraction (15bits)	
23	22	15	14	0

Bit index

To calculate the voltage on a specific pin from the RAW value, you can use the following formula:

$$V_{pin} = \frac{vrefVoltage}{2^{ADCbits} - 1} \times ADCdataRAW[V]$$

Reply structure:

Offset	Length	Name	Description
0	6 bytes	BINheader	Binary message generic format header
6	3 byte (24bit float)	vrefVoltage	ADC reference voltage
9	1 byte	ADCbits	ADC resolution
10	1 byte	numChannels	# of Sampled pins
11	[numChannels × <ADCbits to byte>] bytes	ADCdataRAW	Sampled data

Example BIN data for logic scope

```
<to analyzer>DVM;
<response>
1D F4 44 56 21 00 92 A1 40 0C 0E EE 05 97 05 5A 05 2F 05 E6 04 F5 04 AE 04 C9 04 B1
04 A6 04 72 04 5E 04 6B 04 C3 03
```

Interpreting the response above

```
- CRC16 = 0xF41D (verify online look at: CRC-CCITT (0xFFFF) )
- payload ID = 0x5644 (Digital Voltmeter Data)
- Payload length = 0x0021 (33 bytes payload)
- vrefVoltage = 0x40A192 → (float24) ≅ 5.05 (ADCVref is 5.05V)
- ADCbits = 0x0C (12bit ADC)
- numChannels = 0x0E (14 pins sampled)
- ADCdataRAW[0] = 0x05EE (Vref/admax*RAW = 5.05V/4095*0x05EE ≅ 1.87V)
- ADCdataRAW[1] = 0x0597 (Vref/admax*RAW = 5.05V/4095*0x0597 ≅ 1.76V)
    ⋮
  ⋮
- ADCdataRAW[numChannels-1] = 0x03C3 (Vref/admax*RAW = 5.05V/4095*0x0597≅1.19V)
```

SET

The SET command is used for changing various operation modes of the analyzer. From the COMMANDS, you can obtain all settable parameters. SET command does not generate a reply. Available SET commands and their meaning:

Command	Meaning
SET OUTPUT BIN;	Host communication in BINARY format
SET OUTPUT JSON;	Host communication in JSON format
SET OUTPUT XTERM;	Host communication with human, full ANSI compatible terminal (ExtraPuTTY)
SET OUTPUT ANSI;	Same as SET OUTPUT XTERM;
SET REPEAT;	Enable contiguous acquisition mode
SET NOREPEAT;	Disable contiguous acquisition mode (stop acquisition)

GET

The GET command is used for obtaining details about the analyzer. This command also helps obtaining communication compatibility, by describing parameters of commands and reporting version information. The GET command should be invoked in a sequential mode to read all command's parameters and complete the device information structure. [see]

Data format in JSON mode: JSON object
Data format in BIN mode: 0x5447 (Encapsulated JSON formatted response, see)

Available GET commands and their meaning:

GET PRODUCT

Get product returns HW, FW and communication version.

It returns the product structure:
product.name filed contains human readable product name. This is for display purposes only.

product.version.HW field contains the Hardware revision in string format.

product.version.FW field holds the Firmware revision number in string format.

product.version.COMM field is changed when new commands are added or command parameters changed since last revision. This parameter is for convenience only, as all communication is backward compatible and most of new commands are future-proof if the GUI is implemented in a way to read communication parameters from the device. This is the reason why hardcoding communication format (e.g. command separator is space) is not a good practice.

product.serialID is a globally unique identifier, that allows identifying the device.

GET xxx INFO

The command info details parameter, short description of the command. The GET xxx INFO is a basic help system. The COMMANDS contains the available help for each command, and the exact command that should be called. (e.g. today DVM help is GET DVM_INFO; but might change in the future to HELP DVM;)

Example of getting help for command

```
<to analyzer>GET DVM_INFO;
<response>
{"commands":{"DVM":{"description":"Digital Voltmeter","parameters":{}}}}
```

SCOPE

The scope command invokes the analog oscilloscope feature. The analog oscilloscope is a 66ksps 12bit oscilloscope, capable of sampling one input pin. When an input pin is used to measure analog signals, that pin is no longer available for digital usage. This means for example the pin change LEDs.

The NUMSMP parameter range is dependent on output mode. The actual number of samples are returned by invoking commnds.SCOPE.details command. After deep-merging the commands.SCOPE.parameters.NUMSMP.values.range filed will specify the maximum number of samples in [min,max,scaledmin,scaled_max] format.

The response format is also depending on the commands.SCOPE.bytesPerSample field. This filed defines the number of bytes for a single sample. This parameter is dependent on the hardware.

The sampling frequency range is defined by commands.SCOPE.parameters.FREQ.values.range field.

The PIN parameter defines the sampling pin. All available pins are listed in commands.SCOPE.parameters.FREQ.values.range field.

After completing the conversion, the command returns a response.

Data format in JSON mode

The return value is a SCOPE object with three fields:

SCOPE.samplerate is defining the real sample rate. This could deviate from the requested sample rate based on binary rounding of the timer divider. The SCOPE.samplerate is expressed in samples per second units.

SCOPE.pin field value is the pin number of the sampled pin. This value is '1' based. {"pin":1 means pin1}

SCOPE.voltage is a floating point number array of the sampled voltages from SCOPE.pin at SCOPE.samplerate sample per second.

Example of sampling floating pin 1 in JSON format

```
<to analyzer>SCOPE PIN=1 NUMSMP=10 FREQ=10K;
<response>
{"SCOPE":{"samplerate":9948.750000,"pin":1,"voltage":
[0.309368,0.275536,0.252572,0.233240,0.221152,0.206648,0.201816,0.187312,0.178852,0.174024]}}
```

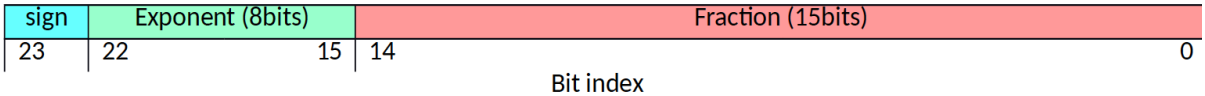
This shows that pin1 was 0.309368V and $\frac{1}{9948.75} seconds = 1.005 msec$ later it was 0.275536V

Data format in BIN mode

In binary format the number of maximum samples are more than all other formats. The sampling rate resolution, reference voltage and other information is sent along the sampled RAW ADC data.

The generic header payload ID is 0x5341 (Analog Scope samples v1).

vrefVoltage and sampleRate are float24 formatted data. Float24 is a shortened IEEE 754 single precision format.



$$V_{pin} = \frac{vrefVoltage}{2^{ADCbits} - 1} \times ADCdataRAW[V]$$

The Click Analyzer uses one based pin numbering. This means if value stored in sampledPin field equals 4, then we were sampling analyzer pin 4. Number 0 is reserved for denoting "disconnected".

Offset	Length	Name	Description
0	6 bytes	BINheader	Binary message generic format header
6	3 byte (24bit float)	vrefVoltage	ADC reference voltage
9	1 byte	ADCbits	ADC resolution
10	1 byte	sampledPin	Sampled pin
11	3 byte (24bit float)	sampleRate	Samplerate [smp/sec]
12	[NUMSMP × <ADCbits to byte>] bytes	ADCdataRAW	Sampled data

Example of sampling floating pin 1 in BIN format

```
<to analyzer>SCOPE PIN=2 NUMSMP=10 FREQ=50K;
<response>
D8 5C 41 53 1C 00 57 9E 40 0C 02 44 43 47 19 01 FA 00 EB 00 D9 00 CC 00 BE 00 B5 00
A7 00 9E 00 96 00
```

Interpreting the response above

- CRC16 = 0x5CD8 (verify online look at: CRC-CCITT (0xFFFF))
- payload ID = 0x5341 (Analog Scope Data)
- Payload length = 0x001C (28 bytes payload)
- vrefVoltage = 0x409E57 → (float24) ≅ 4.95 (ADCVref is 4.95V)
- ADCbits = 0x0C (12bit ADC)
- sampledPin = 0x02 (pin 2 is sampled)
- sampleRate = 0x474344 → *=(float24) = 49988Hz
- ADCdataRAW[0] = 0x0119 (Vref/admax*RAW = 4.95V/4095*0x0119 ≅ 339.7mV)
- ADCdataRAW[1] = 0x00FA (Vref/admax*RAW = 4.95V/4095*0x00FA ≅ 302.2mV)

⋮

⋮

- ADCdataRAW[NUMSMP-1] = 0x0096 (Vref/admax*RAW = 4.95V/4095*0x0096 ≅ 181.3mV)