Click’o’lyzer protocol for developers

# Overview

Click’o’lyzer offers a variety of input and output protocols that are best fir for WEB applications (AJAX, python etc.), Direct human interactions and M2M applications. For best convenience click’o’lyzer offers facilities for autodetection and firmware update

Click’o’lzer offers 2 communications protocols:

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

Click’o’lyzer 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 Welcome message)
3. Commands are terminated with *<separator\_commands>* character (see Welcome message)
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 autodetection)

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.

Example for activating JSON mode

SET OUTPUT JSON;

Response:*<none>*

## BINARY format

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.)

Example for activating the BINARY mode

SET OUTPUT BIN;

Response:*<none>*

### Binary message generic format

The binary message starts with a CRC16 value, followed by payload indentification 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](https://www.lammertbies.nl/comm/info/crc-calculation.php?crc=21210000&method=hex) look at: CRC-CCITT (0xFFFF) )

- payload ID = 0x2121 (NAK response)

- Payload length = 0x0000 (no payload)

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](https://www.lammertbies.nl/comm/info/crc-calculation.php?crc=47543A007B2270696E73223A7B224C4544223A7B2259454C4C4F57223A302C224F52414E4745223A302C22475245454E223A302C22524544223A307D7D7D&method=hex) 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>

### BIN encapsulated JSON

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.

Example for entering the HMI modes

SET OUTPUT ANSI;

Response:*<terminal autodetect sequence>*

# Welcome message

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 BIN encapsulated JSON)

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"}}}

Understanding the response (reformatted for easier reading )

{  
 "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 commandline 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":"="  
 }, … … …

<to analyzer>LS FREQ=100k NUMSMP=500;

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>

CCCCCCCCCCCCCCCC

## 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 =

at 0 sec PIN3 (LS.data[0] & pins[2] > 0) (144 & 128 > 0) ( =logic High)

at 10.002 µsec PIN3 (LS.data[1] & pins[2] > 0) (144 & 128 > 0) ( =High)

at 90.022 µsec PIN3 (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: Binary message generic format)

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](https://www.lammertbies.nl/comm/info/crc-calculation.php?crc=4C5323000E0906070B0D080A0E02010F000C039000900090009000900090009000900090009000&method=hex) 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 = 29 = 512 )

… … …

pinmap[2] = 0x07 (Pinmask for pin3 = 27 = 128 )

… … …

- pinmap[12] = 0x0C (Pinmask for pin13 = 212 = 4096 )

- pinmap[13] = 0x03 (Pinmask for pin14 = 23 = 8 )

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

- data[1] = 0x0090 (0x10+0x80=2pinmap[2]+<unkonwnpin> → pin3 is high)

… … …

- data[9] = 0x0090 (0x10+0x80=2pinmap[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.

**JSON mode**: JSON object

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

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.

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

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

## GET

## SET