DAB-EMBEDDED

USB2IO manual

Table of content

What this device can do?

USB2IO – is a multifunctional interface allows to work with various electronics devices using common electronic industry interfaces.

Features of USB2IO device:

- Communication with electronics using I2C/SPI/I2S/USART/CAN FD/MDIO and other interfaces;
- 16-channel pattern generator;
- 16-pin GPIO interface;
- 8-LVDS pairs available;
- 1A max programmable LDO with voltage range 1.8V – 3.3V designed for:
    - Powering external (customer's) device;
    - Same voltage used for powering IO interface;
- Automatic electronic functional testing (USB2IO able to play various scenarios);
- Communication with Host PC using USB HID device (Virtual COM port);
    - Transmitting data possible using ASCII or RAW protocol;
- Upgradable firmware for control modules (FPGA, MCU);
- Failsafe bootloader for safe firmware upgrade;
- Python and C++ SDKs for developer's support;
- OS supported: Microsoft Windows OS, MacOS and Linux;
- Device powered by USB-C cable from Host PC.

Description

Functionality depends on FPGA image + STM32H7 firmware. System have default FPGA image and bootloader + application on STM32H7 MCU.

After power up – using USB HID VCOM port and ASCII commands – possible to upload new FPGA image and STM32H7 code (and run it).

STM32H7 has internal watchdog and able to fall back into bootloader mode.

Usecases

- I2C read/write/scan
  - Master mode only for now
- SPI read/write
  - Master mode only for now
- USART send/receive
  - RS485 support using TI SN75HVD11D transceiver
- Digital microphone data acquisition (audio I2S interface)
  - As example we are using Knowles SPH0645LM4H MEMS microphone
- I2S audio codec playback and audio record
  - Using AKM4954 codec on Microchip AC324954 board we can do:
    - Record audio stream from onboard microphone
    - Stream audio data to Headphone output
    - Record audio steam from Headphone input
- PDM (Pulse Density Modulation) interface for digital microphones
  - Up to 4 digital microphone pairs can be connected in parallel
- CAN 2.0B bus logger/Communication
  - As example we are using TI TCAN337G transceiver
- Stepper motor control
  - As example we are using TI DRV8711EVM kit
- Color camera (RGB/Bayer/JPEG) supported
  - As example we are using Omnivision OV5640 camera
    - Set resolution to be taken
    - Get single frame
- Management Data Input/Output (MDIO) interface controller (IEEE 802.3)
  - Only Master mode
  - Can read/write data
- GPIO 16-bit port
- 16x I/O pattern generator

Technical specification

External memory	32MBytes, SDRAM
Additional logic	FPGA-based
Supported logic levels	1.8V – 3.3V, LVDS 2.5V
Max I/O speed	300 MHz
Programmable PLL (connected to FPGA) frequency range	10 MHz – 250 MHz
USB Type	2.0 High speed (USB-C connector)
Power	Using USB interface (5V @ 1.5A max)
Dimensions	100 x 65 x 20 mm
Weight	80 g
Certification	CE

Internal structure

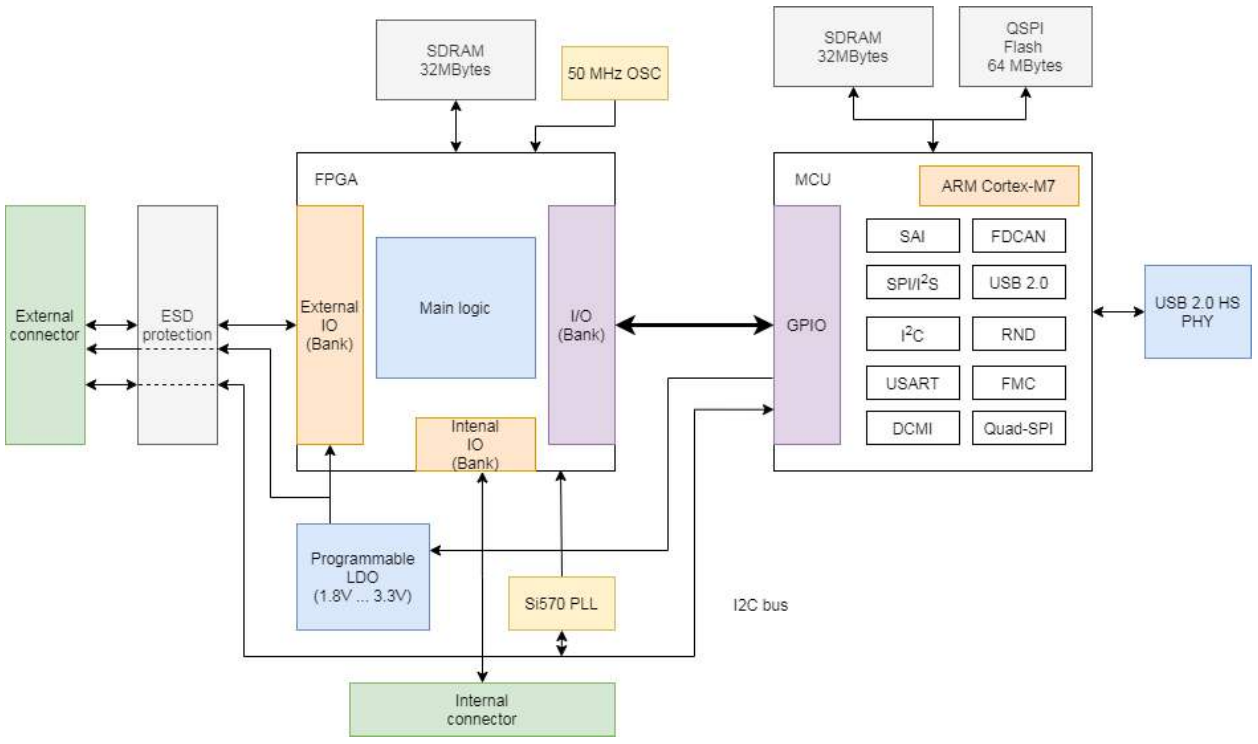


Figure 1: Internal structure of USB2IO device

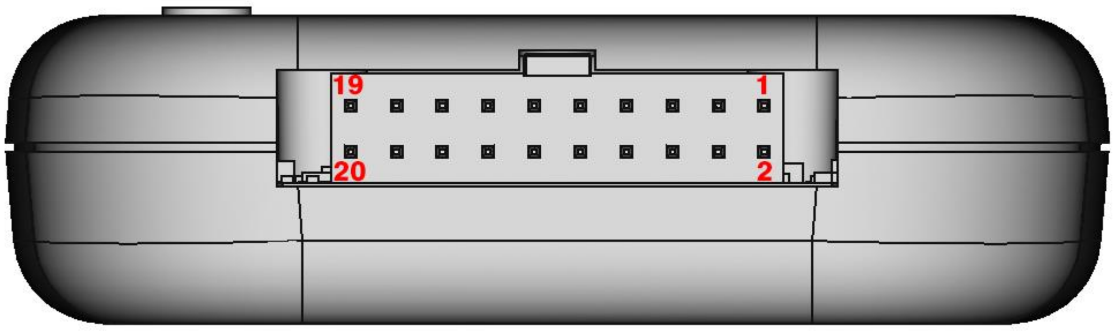
Internal structure description:

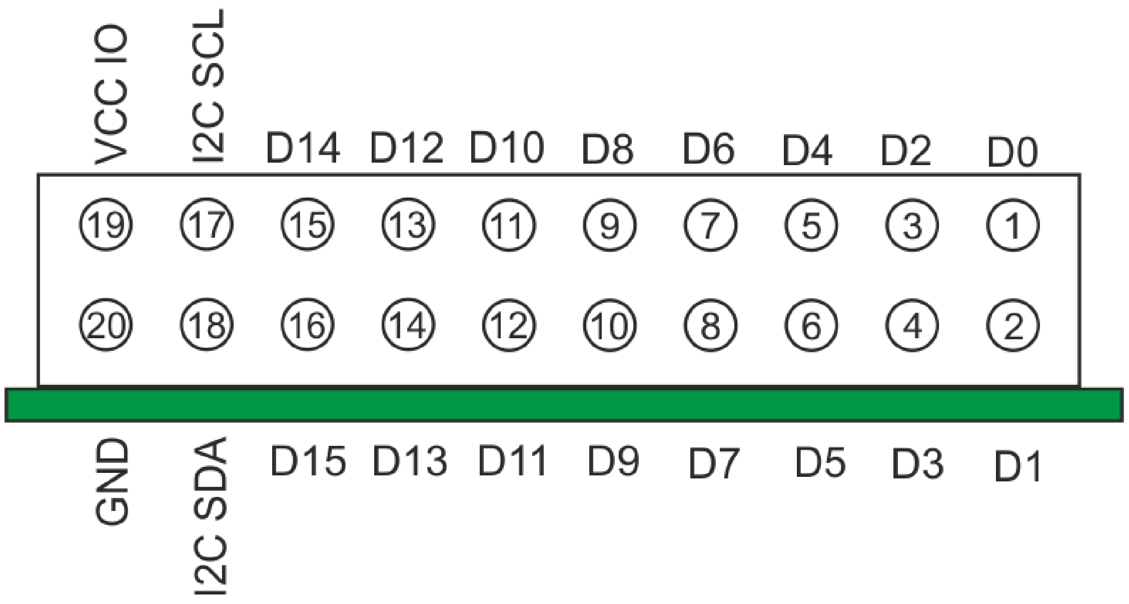
1. MCU - STM32H743 ARM Cortex-M7 based microcontroller with 480MHz CPU clock frequency:
    a. External 32MB SDRAM memory for storing captured data or data waiting to be send;
    b. External 64MB QSPI Flash for extra FPGA code storage;
    c. External USB2.0 High speed PHY;
2. FPGA - Intel Cyclone 10LP (10CL040) with external components:
    a. 32MB SDRAM memory;
    b. Si570 programmable PLL (10MHz ... 945MHz);
    c. 50MHz oscillator;
    d. Programmable LDO supplies power for External IO FPGA bank;
    e. Depends on FPGA firmware – possible to select 16x GPIO mode (single-ended), 8x LVDS pair mode or mix of two modes.

Interface with Host PC – USB 2.0 High speed (USB-C connector);
External IO connector – 20-pin connector (2-raw, 2.54mm pin pitch).

Interface pinout and possible functionality

External connector:





⚠	**WARNING**: HW I²C bus, pin 17 and 18 have always 3.3V levels.

External connector pin multiplication possibilities (based on default FPGA firmware image):

External connector	Pin 1	Pin 2	Pin 3	Pin 4	Pin 5	Pin 6
Data pin	D0	D1	D2	D3	D4	D5
FPGA pin	IO_B26n	IO_B26p	IO_B13n	IO_B13p	IO_B7n	IO_B7p
Function 0	GPIO0_IN0	GPIO0_IN1	GPIO0_IN2	GPIO0_IN3	GPIO0_IN4	GPIO0_IN5
Function 1	Logic “0”	Logic “0”	Logic “0”	Logic “0”	Logic “0”	Logic “0”
Function 2	Logic “1”	Logic “1”	Logic “1”	Logic “1”	Logic “1”	Logic “1”
Function 3	GPIO0_OUT0	GPIO0_OUT1	GPIO0_OUT2	GPIO0_OUT3	GPIO0_OUT4	GPIO0_OUT5
Function 4	GEN_OUT0	GEN_OUT1	GEN_OUT2	GEN_OUT3	GEN_OUT4	GEN_OUT5
Function 5	50 MHZ					
Function 6		Si570 PLL				
Function 7			SPI_NCS	SPI_MISO	SPI_CLK	SPI_MOSI
Function 8			UART4_CTS	UART4_RXD	UART4_TXD	UART4_RTS
Function 9						
Function 10			SPIM_NCS	SPIM_MISO	SPIM_CLK	SPIM_MOSI
Function 11			MII_MDC	MII_MDIO		
Function 12	CAM_D0	CAM_D1	CAM_D2	CAM_D3	CAM_D4	CAM_D5
Function 13	MOTOR1_STEP	MOTOR1_DIR				
Function 14						
Function 15						
Function 16						
Function 17			RS485_nRE	RS485_RXD	RS485_TXD	RS485_DE
Function 19					SAI2_SD_A	SAI2_FS_A
Function 20						
Function 21						

External connector	Pin 7	Pin 8	Pin 9	Pin 10	Pin 11	Pin 12
Data pin	D6	D7	D8	D9	D10	D11
FPGA pin	IO_B1n	IO_B1p	IO_B27n	IO_B27p	IO_B21n	IO_B21p
Function 0	GPIO0_IN6	GPIO0_IN7	GPIO0_IN8	GPIO0_IN9	GPIO0_IN10	GPIO0_IN11
Function 1	Logic “0”	Logic “0”	Logic “0”	Logic “0”	Logic “0”	Logic “0”
Function 2	Logic “1”	Logic “1”	Logic “1”	Logic “1”	Logic “1”	Logic “1”
Function 3	GPIO0_OUT6	GPIO0_OUT7	GPIO0_OUT8	GPIO0_OUT9	GPIO0_OUT10	GPIO0_OUT11
Function 4	GEN_OUT6	GEN_OUT7	GEN_OUT8	GEN_OUT9	GEN_OUT10	GEN_OUT11
Function 5						
Function 6						
Function 7						
Function 8						
Function 9	CAN_TX		CAN_TX	CAN_RX	CAN_RX	CAN_TX
Function 10						
Function 11						
Function 12	CAM_D6	CAM_D7	CAM_D8	CAM_D9	CAM_D10	CAM_D11
Function 13						
Function 14						
Function 15						
Function 16						
Function 17						
Function 18					UART1_RX	UART1_TX
Function 19	SAI2_SCK_A	SAI2_SD_B	SAI2_FS_B	SAI2_SCK_B	SAI2_MCLK_B	SAI4_CK1
Function 20						
Function 21						

External connector	Pin 13	Pin 14	Pin 15	Pin 16
Data pin	D12	D13	D14	D15
FPGA pin	IO_B18n	IO_B18p	IO_B14n	IO_B14p
Function 0	GPIO0_IN12	GPIO0_IN13	GPIO0_IN14	GPIO0_IN15
Function 1	Logic “0”	Logic “0”	Logic “0”	Logic “0”
Function 2	Logic “1”	Logic “1”	Logic “1”	Logic “1”
Function 3	GPIO0_OUT12	GPIO0_OUT13	GPIO0_OUT14	GPIO0_OUT15
Function 4	GEN_OUT12	GEN_OUT13	GEN_OUT14	GEN_OUT15
Function 5				
Function 6				Si570 PLL
Function 7				
Function 8				
Function 9				
Function 10				
Function 11				
Function 12	CAM_VSYNC	CAM_PIXCLK	CAM_HSYNC	
Function 13				
Function 14	TIM5_CH1_OUT	TIM5_CH2_OUT	TIM5_CH3_OUT	TIM2_CH1_IN
Function 15				
Function 16			MOTOR2_STEP	MOTOR2_DIR
Function 17				
Function 18	UART1_RTS	UART1_CTS		
Function 19	SAI4_D1			
Function 20			IIC_SCL	IIC_SDA
Function 21	Encoder_In1	Encoder_In2		

Description:

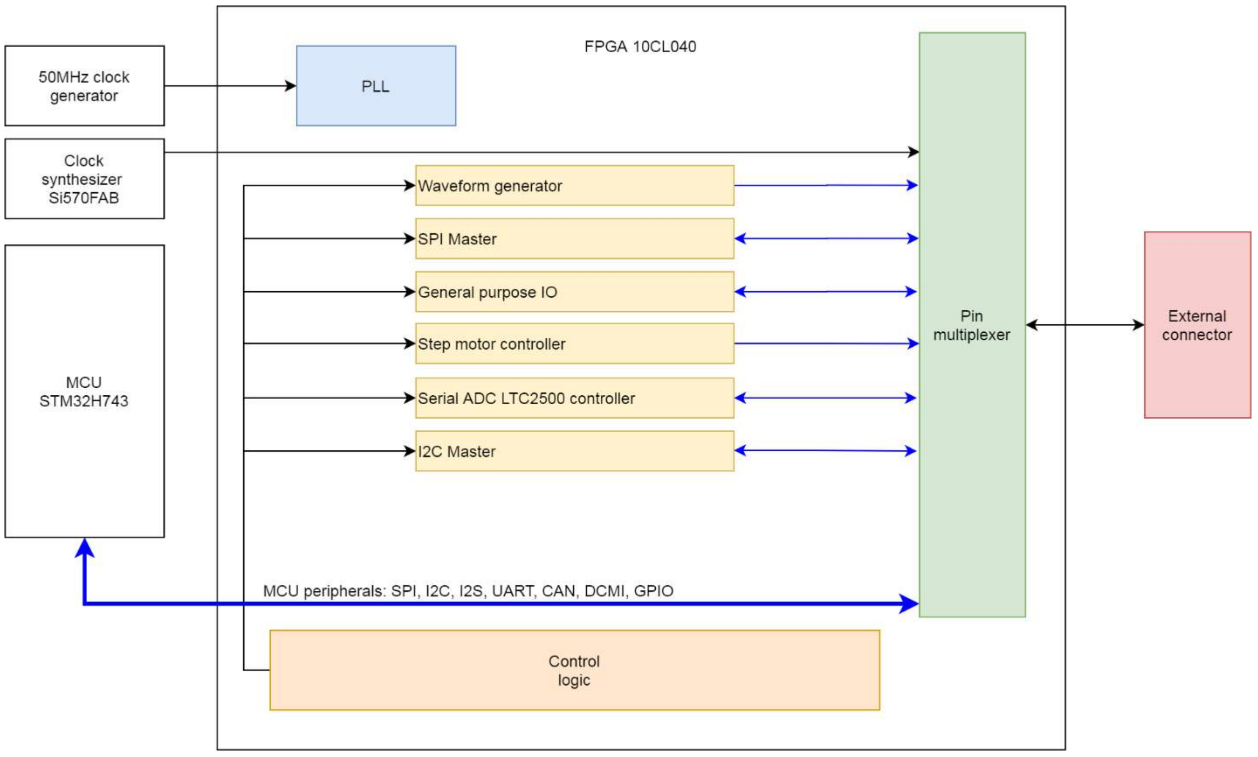
GPIO0_INx	– GPIO inputs;
GPIO0_OUTx	– GPIO outputs;
GEN_OUTx	– Waveform generator outputs;
50 MHZ	– 50MHz output;
Si570 PLL	– Frequency generator: 10MHz to 300MHz programmable output;
SPI_NCS, SPI_MISO, SPI_CLK, SPI_MOSI	– SPI interface (connected to STM32 MCU);
UART4_CTS, UART4_RXD, UART4_TXD, UART4_RTS	– UART (Universal asynchronous receiver-transmitter) interface (connected to STM32 MCU);
CAN_RX, CAN_TX	– CAN 2.0 FD bus (external physical transceiver required, like TI TCAN337 or similar) (connected to STM32 MCU);
SPIM_NCS, SPIM_MISO, SPIM_CLK, SPIM_MOSI	– SPI Master (in FPGA);
MII_MDC, MII_MDIO	– Management Data Input/Output (MDIO) interface (IEEE 802.3) (connected to STM32 MCU);
CAM_Dx	– 12-bit (10-bit and 8-bit) parallel CMOS camera interface (connected to STM32 MCU);
MOTOR1_STEP, MOTOR1_DIR	– Step motor controller 1 outputs (controller implemented in FPGA);
MOTOR2_STEP, MOTOR2_DIR	– Step motor controller 2 outputs (controller implemented in FPGA);

TIM1_CH1_OUT, TIM1_CH2_OUT, TIM1_CH3_OUT	– 32-bit timer with 3 outputs (PWM mode, output compare mode) (connected to STM32 MCU);
TIM2_CH1_IN	– 16-bit timer input (capture mode) (connected to STM32 MCU);
UART1_CTS, UART1_RXD, UART1_TXD, UART1_RTS	– UART (Universal asynchronous receiver-transmitter) interface (connected to STM32 MCU);
RS485_nRE, RS485_RXD, RS485_TXD, RS485_DE	– RS485 interface (TTL levels, external physical transceiver required, like TI SN75HVD11D or similar) (connected to STM32 MCU);
SAI2_SD_A (data line), SAI2_FS_A (frame sync), SAI2_SCK_A (clock)	– Serial Audio interface (block A) (connected to STM32 MCU);
SAI2_MCLK_B (master clock output), SAI2_SD_B (data line), SAI2_FS_B (frame sync), SAI2_SCK_B (clock)	– Serial Audio interface (block B) (connected to STM32 MCU);
SAI4_CK1 (clock), SAI4_D1 (data)	– Serial Audio interface dedicated for PDM microphones (connected to STM32 MCU);
IIC_SDA (data), IIC_SCL (clock)	– I^{2}C master interface (Implemented in FPGA);
Encoder_In1, Encoder_In2	– Encoder input lanes (connected to STM32 MCU timer 5);

Default FPGA firmware description

FPGA logic can be divided into several subsystems:

- Clocking
- Interface and control
- Periphery functions
- Pin mux



Clocking subsystem

A 50 MHz clock generator is used as primary clock source for FPGA. Its output is fed into embedded PLL, which produces clock for internal logic. In most cases 200MHz used as internal clock provided by PLL.

Interface and control

The FPGA works under control of MCU. 16-bit parallel bus is used for MCU-FPGA communication.

The FPGA has a set of registers (register map), which control all functionality of the FPGA. These registers are accessible by MCU.

Periphery functions

The main task of FPGA is to realize peripheral functions, which are not realized or not accessible in MCU. Each function is represented as a separate unit inside the FPGA. At the moment next functions are implemented:

- Waveform generator
- SPI master
- General purpose IO
- Step motor controller
- Serial ADC LTC2500 controller

This list will be expanded according to user needs.

Note, that peripheral functions can work in parallel if they do not share same pins of external connector (see Pin Mux section).

Waveform generator (WFG)

This list will be expanded according to user needs.

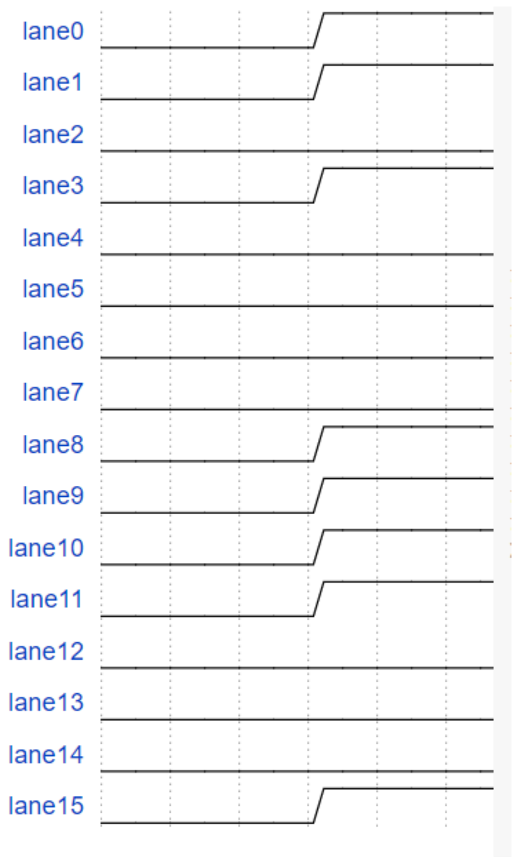
This function allows to generate arbitrary logic waveforms on up to 16 lanes.

“Logic waveform” means, that each lane can be in one of two states – ”High” or “Low”. “Low” is a 0-volt level, “High” is configurable in the range 1.8V – 3.3V.

WFG has internal memory for storing waveform data. The memory organization is 16K x 16 bits. Each 16-bit words is called “sample”.

The sample represents the state of all 16 lanes of generator: bit 0 – lane 0, bit 1 – lane 1, and more.

Let assume, you generate 0x8F0B sample after 0x0000 sample:

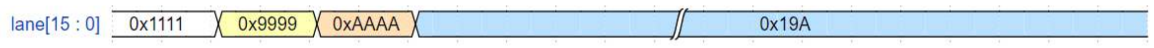


During generation the samples are sequentially read from memory and placed to external connector pins. The chunk of data words, which should be sequentially outputted (generated) is called “segment”. Maximum segment size is equal to memory size (16K).

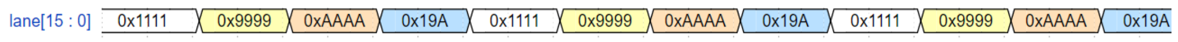
There are two modes of generation: single segment, cyclic. In single segment mode when all samples are generated, the WFG is stopped and the last sample remains on output pins. In cyclic mode a segment is repeated directly after previous segment is generated.

For example, you have a segment of 4 samples: {0x1111, 0x9999, 0xAAAA, 0x19AF}.

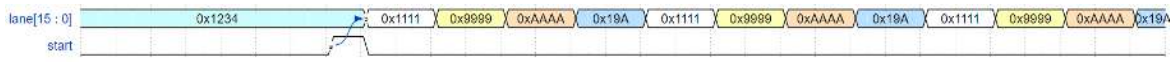
In single segment mode the waveform will look like:



In cyclic mode you'll get:



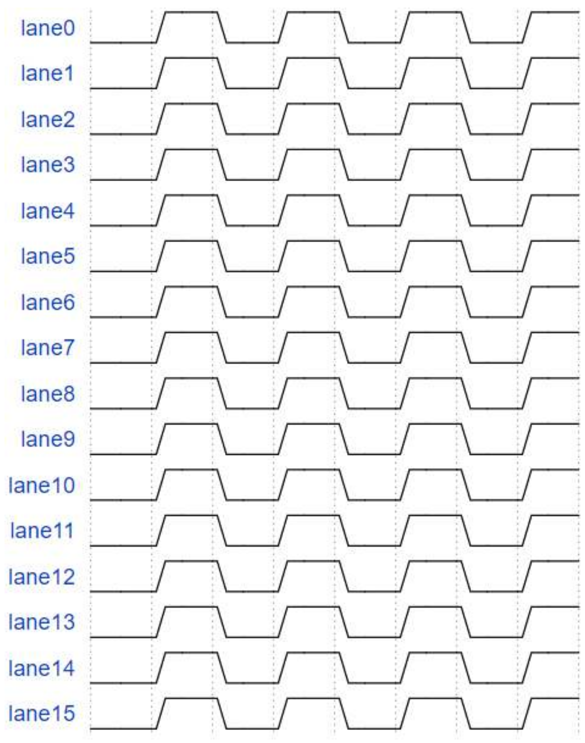
It is possible to "preload" output lanes with initial sample, which will be held on lines until "start" command is applied. For example, initial = 0x1234, the segment is same, mode is cyclic:



During generation the samples are read from memory in user defined time intervals – sample rate. The sample rate is multiply of FPGA main clock period (5 nanoseconds). Minimal value is 5 ns, maximal is $2^{(32-1)}$ * 5 ns.

Example 1.

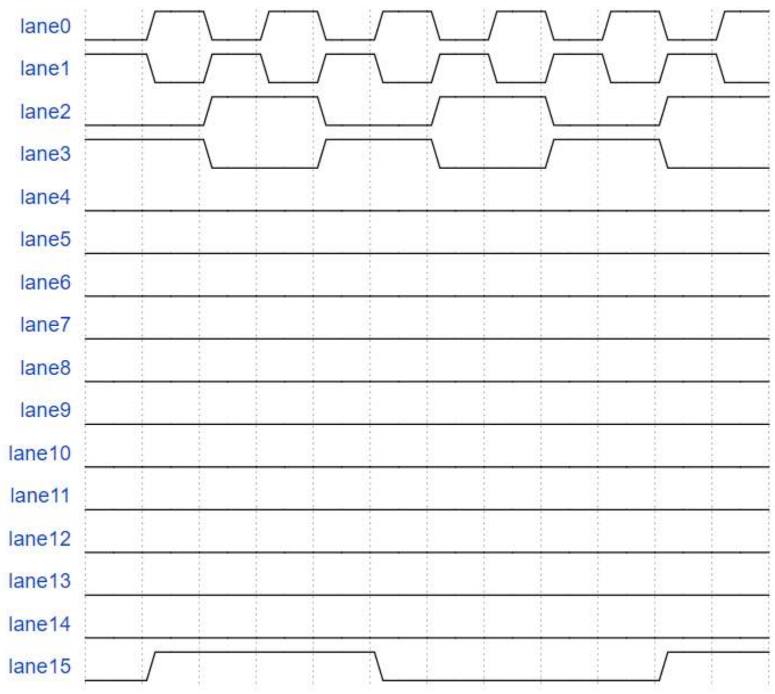
Need to generate 100MHz meander signal on all 16 channels:



Choose fastest sample rate, segment size = 2, data samples = {0x0000, 0xFFFF} and cyclic mode.

Example 2.

Lane 0 – 1 – differential 100MHz, lane 2 – 3 differential 50MHz, lane 15 – 20 ns pulse every 50 ns:



Choose fastest sample rate, segment size = 10, data samples = {0x000A, 0x8009, 0x8006, 0x8005, 0x800A, 0x0009, 0x0006, 0x0005, 0x000A, 0x0009} and cyclic mode.

SPI Master (SPIM)

In comparison to MCU SPI master unit, FPGA offers more flexible solution for SPI master.

SPIM transmits data word to / from slave device. CS signal is asserted during transmission of single word.

The unit has internal memory with 16K x 16-bit organization for transmission and the same memory for reception. The data for transmission are sequentially fetched from the memory. Received data are stored to the memory in the same approach.

SPI clock period is configurable in 20 ns steps and can be in range 20ns – $2^{(16-1)}$ * 20 ns. Clock polarity can be inverted.

Bit width of data word is configurable and can be set to a value in range from 1 to 16. Data is sent/received MSB first. Note, that if used bit width is smaller than 16, the data word for transmission should be left-aligned. For example, if 8-bit width is used, and you want to transmit 0xFE, you have to store 0xFExx word. As well, received data with bit width smaller than 16, should be recovered by shifting right in appropriate number of bits. For example, with bit width = 6, the received word 0x8C00 become 0x23 (right shift by 10).

There are three modes of operation:

- Single word
- Single segment
- Cyclic

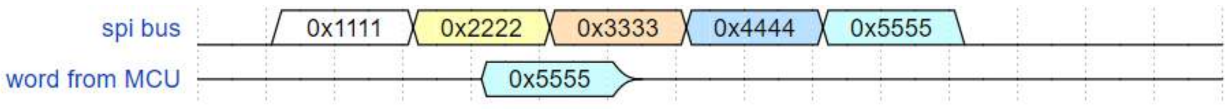
In single word mode, the SPIM transmits a single data word over SPI and stops. This is most simple mode.

Two other modes work with sequence of data words called segment. Data words are transmitted with user specified period – sample rate. The sample rate is multiply of 5 ns and can be as slow as 2(24 –1) * 5 ns.

In single segment mode the SPIM transmits a segment until segment end is reached. During transmission it is possible to add new data to segment end, expanding segment length. There is an option called “stop when empty”. If this option is set, the transmission is stopped after last data word is sent. If this option is not set, the very last data word is continuously transmitted with the same sample rate, as previous data words.

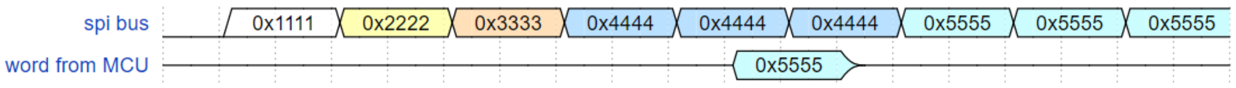
Example 1.

Transmission is started with segment of 4 words {0x1111, 0x2222, 0x3333, 0x4444}. During transmission a 5-th word 0x5555 is added by MCU, “stop when empty” is set:



Example 2.

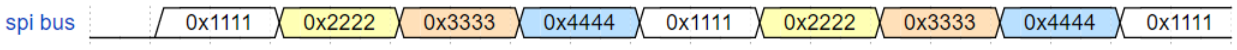
Transmission is started with segment of 4 words {0x1111, 0x2222, 0x3333, 0x4444}, “stop when empty” is cleared. When all four words are transmitted, the last one (0x4444) is retransmitted again. Then a 5-th word 0x5555 is added by MCU. The SPIM starts to continuously transmit 0x5555 word until next word is received from MCU:



In cyclic mode the SPIM transmits a segment until segment end is reached. Then the segments are transmitted again. It is not possible to add new data words during transmission.

Example 1.

Transmission is started with segment of 4 words {0x1111, 0x2222, 0x3333, 0x4444}:



During transmission in all modes, MISO lane is captured and received data words are stored in internal memory. The memory works in FIFO mode. The data is accessible as soon as new word is received. There is no need to wait for transmission complete. Due to this fact, amount of received data could be bigger, than memory depth, because read data are removed from FIFO.

General purpose IO (GPIO)

The user has direct access to pins of external connector. There are two registers implemented in FPGA – output register and input register.

Data written to output register are fed to external pins, while input register reflects actual state of these pins.

This function can be used for constant signals driving/capture, bit-bang mode end etc.

Note, that it is not necessary to drive all 16 output pins by output register. Driven pins can be individually selected. For details see pin mux description.

Step motor controller (SMC)

This function generates control signals for step motor driver (like TI DRV8711) – Step and Direction. There are two independent SMC channels. Both channels are equal.

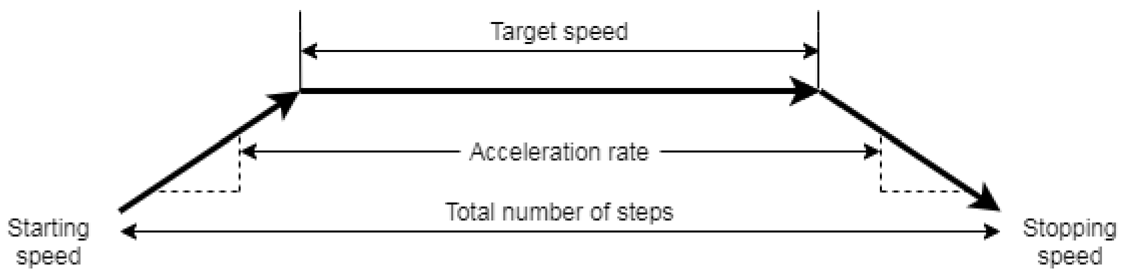
Usually, is not possible to start motor at maximum speed. Instead you have to start from minimal acceptable speed (this parameter is motor specific) and then speed up with constant acceleration. Similar to start, it is not allowed to stop motor immediately. You have to slow down with constants acceleration.

The main task of SMC module to speedup/slowdown motor by specified number of steps in a shortest time in respect to motor specifications.

As input parameters the user have to specify direction of rotation, minimal frequency of Step pulses, maximal frequency of Step pulses, acceleration rate and number of steps.

Minimal frequency of Step pulses corresponds to motor starting speed and motor stopping speed. The whole process starts and ends with this speed.

Maximal frequency of Step pulses corresponds to motor target speed. When SMC reaches this speed, it will keep it until slowing down:



In some cases, the target speed is unreachable. For example, if number of steps is low, or when acceleration rate is to slow. In this case, form of the trapeze on the picture above became a triangle with highest corner < target speed.
Used formula:

$$T_{next}=T_0 * (1 + T_0At)$$

Where:

T_{next} – step time;
T_0 – initial step time;
t – current step time;
A – acceleration.

Minimal and maximal frequencies should be in range 1Hz – 250 KHz. Acceleration speed range is 1 – 50KHz / s. Number of pulses should be in range: 1 – $2^{(24-1)}$.

Serial ADC LTC2500 controller

This function allows to configure serial ADC LTC2500 and capture data from it.
To configure the ADC, you have to specify filter type and down sampling factor:

Filter type	Value
SINC1	1
SINC2	2
SINC3	3
SINC4	4
SSINC	5
FLAT PASSBAND	6
AVERAGING	7

Downsampling factor	Value
2	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14

For detailed description of LTC2500 configuration refer to LTC2500 datasheet.

LTC2500 function captures data from ADC with user defined sample rate. Sample rate is in steps of 5 ns and should be in range 1000 ns – $2^{(24-1)}$ * 5 ns.

Captured data are stored in external SDRAM. Each sample is 32 bit wide. Count of samples can be set to a value in range 1 – 8 * 1024 * 1024.

Acquisition process can be started in three ways:

By applying Start command from PC;

By rising edge on external connector pin 8;

By falling edge on external connector pin 8.

In last two cases a Start command should be applied. After it the function starts for edge detection. When selected edge in detected, acquisition process starts.

IIC master (I^2C)

IIC Master implemented in FPGA for communication with various IIC slave devices. II2C FPGA Master has next features:

- 3 baud rate options: 100kHz, 400kHz and 1MHz;
- 7-bit slave address;

Pin mux block

Main goal of this unit is to connect pins of external connector to peripheral functions, to MCU periphery or to other FPGA signals.

Peripheral functions are described above. MCU periphery can be SPI2, UART4, FDCan1, MDIO, DCMI.

“Other FPGA signals” mean clock sources, static levels.

To avoid redundant complexity, each pin has a set of sources to which it can be connected.

All sources are divided on groups by types:

Group number	Type	Description
0	GPIO function	Pin is an input, connected to input register of GPIO.
1	Static ‘0‘	Pin is output, driving static ‘0’.
2	Static ‘1‘	Pin is output, driving static ‘1’.
3	GPIO function	Pin is an output, connected to output register of GPIO.
4	WFG function	Pin is output, connected to WFG function
5	Clock source	Pin is output, generating 50Mhz signal
6	Clock source	Pin is output, connected to onboard clock synthesizer
7	MCU SPI2	Pin is connected to SPI2 pins of MCU
8	MCU UART4	Pin is connected to UART4 pins of MCU
9	MCU FDCAN1	Pin is connected to FDCAN1 pins of MCU
10	SPIM function	Pin is output, connected to SPIM function
11	MCU MDIO	Pin is connected to MDIO pins of MCU
12	MCU DCMI	Pin is connected to DCMI pins (Camera) of MCU
13	SMC function	Pin is output, connected to SMC function (block 1)
14	MCU timer function	Pin is input, connected to MCU timer function
15	LTC2500 function	Pin is output, connected to LTC2500 function
16	Stepper motor controller	Pin is output, connected to stepper motor controller function
17	SMC function	Pin is output, connected to SMC function (block 2)
18	USART1	Pin is connected to USART1 pins (Camera) of MCU
19	SAI	Pin is connected to SAI (Serial Audio interface) pins of MCU
20	FPGA I^{2}C	Pin is input, connected to FPGA I^{2}C block
21	Encoder inputs	Pin is connected to Timer5 pins of MCU

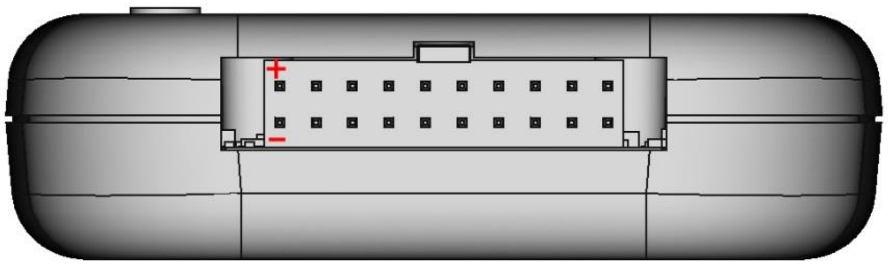
USB2IO command list

NOTE: All commands should end with <CR><LF> (like AT-commands).

Setting external connector voltage

```
expv write a
```

a - Voltage in millivolts (possible values are: 3300, 2500, 1800, 0) for IO port.



WARNING:
Before connecting any external hardware, we recommend to turn off external voltage on IO connector.

Example:

```
expv write 0
```

Turns off external voltage.

```
expv write 3300
```

Device will provide 3.3V @ 1.0A.

External connector pinmux

Configure pinmux:

```
gpio configure a,b
```

a - pin No. Can be 0-15, 255. 255 means all pins.
b - function code.
Each of 16 IO pins on external connector can be individually programmed for specific function. Default value for every pin is "GP input". See “External connector pin multiplication possibilities” section for function codes.

Example:
gpio configure 0,5

Configures pin 1 to 50MHz clock output.

Binary data buffer in SDRAM

SDRAM capacity is 32Mbytes. Upper 2Kbytes are used by firmware. The rest space is available as data buffer. Its size is 32Mbytes – 2Kbytes = 32766Kbytes. The buffer is accessible for read and write operations. There set of commands, which use buffer as data source or storage. Normally these commands have "buffer" in the name, like "spi write **buffer**".

Get raw data from SDRAM buffer:

```
get buffer
```

This command will be acknowledged with "READY", if USB2IO device is ready to execute this command.
After successful acknowledgement, 8 bytes (2x 32-bit words) have to be sent:

32-bit word	Description
0	Requested buffer size (in bytes)
1	0

When bytes will be sent, USB2IO device will return byte array (with requested size). Requested size should not exceed size of available SDRAM buffer.
A the end of data OK or FAIL will be sent.

Upload raw data to SDRAM buffer:

```
put buffer
```

This command will be acknowledged with "READY", if USB2IO device is ready to execute this command.
After successful acknowledgement, 8 bytes (2x 32-bit words) have to be sent:

32-bit word	Description
0	Buffer size to be sent (in bytes)
1	CRC32 % 0x100000000

CRC32 is calculated on buffer data only. ZLIB.crc32 is used.
When 8 bytes will be sent, data bytes need to be sent to USB2IO device with exact size provided earlier.
Buffer size should not exceed size of available SDRAM buffer.

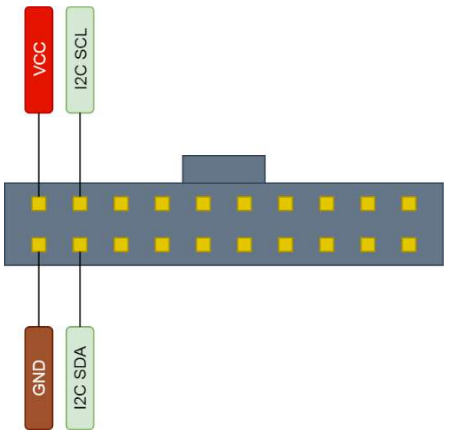
Command returns: OK in case of success, FAIL in case of error.

I2C bus

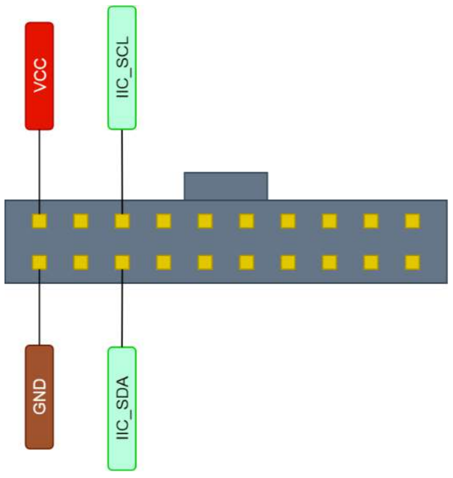
I2C bus master has next features:

- 3 baud rates (100kHz, 400kHz and 1MHz);
- 7-bit slave address support;
- SMBus (system management bus) and PMBus (power management bus) compatible;
- I2C master in MCU;
- I2C master in FPGA.

MCU I2C pinout:



FPGA I2C pinout:



Configure I2C bus:

```
i2c configure a,b
```

a – I2C channel [1 – STM32, 2 - FPGA]
b - I2C baud rate [0 - 100KHz, 1 - 400KHz, 2 - 1MHz].

When STM32 channel is used, two dedicated pins if external connector are used: 17 – SCL, 18 – SDA., voltage level is fixed to 3.3V.
When FPGA channel is used, pin 15 is SCL and pin 16 is SDA, voltage level can be set with “expv write a” command.
Make sure, that GPIOs are configured to I^{2}C function.

Command returns: OK in case of success, FAIL in case of error.

Example:

```
i2c configure 1, 0
```

Will configure STM32 I^{2}C channel with 100KHz baud rate.

I2C scan command:

```
i2c scan a
```

a – I2C channel [1 – STM32, 2 - FPGA]

Return: OK, <List of slave address of I2C devices>

<u>Example:</u>
i2c scan 2

Scans slave devices using FPGA channel.

Write to I2C device

```
i2c write a,b,c...
```

a – I2C channel [1 – STM32, 2 - FPGA]
b – Slave address;
c – comma separated array of data to be sent. Size of array should not exceed 255 bytes.

Send bytes over I2C bus to specific slave address:

Command returns: OK in case of success, FAIL in case of error

<u>Example</u>:

i2c write 1, 0x40,0x10,0x11,0x12
Send 0x10,0x11,0x12 bytes to device with slave address 0x40, use STM32 channel.

Read from I2C device

```
i2c read a,b, c...
```

a – I2C channel [1 – STM32, 2 - FPGA]
b – Slave address;
c – count of bytes to receive

Receives bytes over I2C bus from specific slave address. Count of bytes should not exceed 256.

Command returns: OK, <data> in case of success, FAIL in case of error.

<u>Example</u>:
i2c read 2, 0x40, 3

Receive 3 bytes from I2C address 0x40, use FPGA channel.

Write I2C data from SDRAM buffer:

```
i2c writebuffer a, b, c
```

a – I2C channel [1 – STM32, 2 - FPGA]
b – Packet length
c – Packet size

The data from SDRAM buffer are sent by packets. First byte of the packet is I²C slave address, the rest bytes in the packet are data, will be sent to that slave address. The data should be placed to SDRAM with "put buffer" command.
Total number of bytes to write should not exceed size of available SDRAM buffer.

Command returns: OK in case of success, FAIL in case of error.

Example:

i2c writebuffer 1, 5, 10
Send 10 packets with 4 bytes of data in each. Use STM32 channel.

Write I2C data from SDRAM buffer synchronously:

```
i2c writebuffersync a, b, c
```

a – I2C channel [1 – STM32, 2 - FPGA]
b – Packet length
c – Packet size
d – Trigger timeout

This function is similar to "i2c writebuffer". The only difference is: each packet is sent when trigger event occurs. USB2IO device waits for trigger event not more than "timeout" miliseconds.
Trigger should be configured with "trigger configure" command.

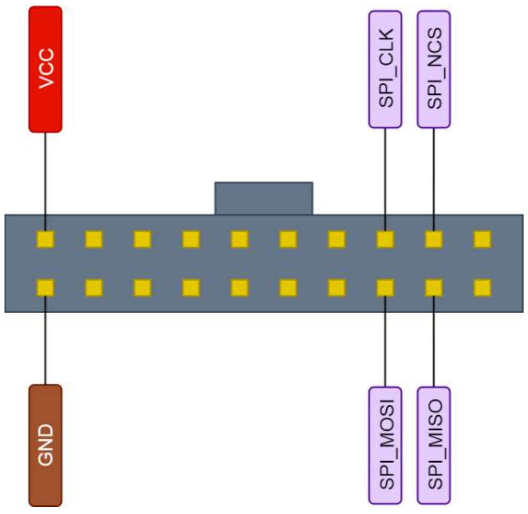
Command returns: OK in case of success, FAIL in case of error.

Example:

i2c writebuffersync 1, 5, 10,1000
Send 10 packets with 4 bytes of data in each. Use STM32 channel. Wait for trigger not longer than 1000 ms.

SPI bus

SPI bus pinout:



Configure spi bus:

```
spi configure a,b,c,d,e,f
```

a - SPI Channel [1],
b - Chip select polarity [0 – Active low, 1 – Active high],
c - Clock polarity [0 - Low, 1 - High],
d - Clock phase [0 - 1st edge, 1 - 2nd edge],
e - Clock frequency in Hz
f - First bit [0 - MSB, 1 - LSB]

Command returns: OK in case of success, FAIL in case of error.

Example:
spi configure 1,0,0,0,12000000,0

SPI clock polarity low, chip select active low, 1st clock edge, 12MHz clock, first bit – MSB.

Direct write SPI data:

```
spi write a,b...
```

a - SPI Channel [1],
b – comma separated array of data to be sent. Size of array should not exceed 256 bytes.

Command returns: OK in case of success, FAIL in case of error.

Example:
spi write 1,0xAA,0x55,0x01,0x22

Send 4 bytes: 0xAA 0x55 0x01 0x22

Direct read SPI data:

```
spi read a,b...
```

a - SPI Channel [1],
b – Number of bytes to read. Requested number should not exceed 256 bytes.

Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
spi read 1, 4

Read 4 bytes.

Direct read and write SPI data simultaneously *(using both MISO and MOSI, read same amount of data as sent):*

```
spi wread a,b…
```

a - SPI Channel [1],
b – comma separated array of data to be sent. Size of array should not exceed 256 bytes.

Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
spi wread 1, 4, 5, 6, 7

Send 4 bytes (4, 5, 6, 7) and receive 4 bytes.

Write SPI data from SDRAM buffer:

```
spi writebuffer a, b, c
```

a - SPI Channel [1],
b – Packet length,
c – Packets count

The data from SDRAM buffer are sent by packets. All bytes inside the packet are sent with CS signal constantly active. CS goes active before first byte of the packet, and goes inactive after all bytes of the packet are sent.
The data should be placed to SDRAM with “put buffer” command.
Toltal number of bytes to write should not exceed size of available SDRAM buffer.

Command returns: OK in case of success, FAIL in case of error.

Example:
spi writebuffer 1, 4, 10

Send 10 packets, each of 4 bytes.

Write SPI data from SDRAM buffer synchronously:

```
spi writebuffersync a, b, c, d
```

a - SPI Channel [1],
b – Packet length,

c – Packets count
d – Timeout in milliseconds

This function does the same as “spi writebuffer”, but a packet transmission is started when trigger signal is received from FPGA. Trigger event is waited for "timeout" milliseconds.
Trigger should be configured with “trigger configure" command.
The data should be placed to SDRAM with “put buffer” command.
Toltal number of bytes to write should not exceed size of available SDRAM buffer.

Command returns: OK in case of success, FAIL in case of error.

<u>Example</u>:
spi writebuffersync 1, 4, 10, 1000

Send 10 packets, each of 4 bytes, wait for trigger event not more than 1000 ms.

I2S bus (Audio interface)

I2S is using same pins as SPI:

SPI_CLK = I2S_CK;
SPI_NCS = I2S_WS;
SPI_MISO = I2S_SDI;
SPI_MOSI = I2S_SDO.

Configure I2C bus:

```
i2s configure a,b,c,d,e,f,g,h
```

a - I2S Channel [1],
b - Audio standard [0 - Philips audio, 1 - I2S MSB, 2 - I2S LSB, 3 - PCM Short, 4 - PCM Long],
c - Audio data format [0 - 16-bit data, 1 - 16-bit extended data, 2 - 24-bit data, 3 - 32-bit data],
d - Audio frequency [0 - 8KHz, 1 - 11KHz, 2 - 16KHz, 3 - 22KHz, 4 - 32KHz, 5 - 44KHz, 6 - 48KHz, 7 - 96KHz, 8 - 192KHz]
e - Clock polarity [0 - Clock low, 1 - Clock high]
f - First bit [0 - MSB, 1 - LSB]
g - Frame sync inversion [0 - No inversion, 1 - Enable inversion]
h - 24bit frame alignment[0 - 24-bit alignment right, 1 - 24-bit alignment left]

Pins of external connector should be configured to SPI function (7).
Command returns: OK in case of success, FAIL in case of error.

<u>Example</u>:
i2s configure 1,1,3,4,0,0,0,0

I2S audio MSB, 32-bit data, 32KHz audio, clock polarity low, no frame sync inversion, 24-bit alignment right (not used)

Direct write I2S data:

```
i2s write a, b...
```

a - I2S Channel [1],
b – comma saparated array of audio data samples. Size of array should not exceed 256 bytes.

Even samples are sent to left channel, odd – to rigth. Each sample should be 2 or 4 bytes wide depending on I2S configuration.
Command returns: OK in case of success, FAIL in case of error.

<u>Example</u>:
i2s write 1,0x0100,0x2000,0x0101,0x2001

Send to seft channel: 0x100, 0x101
Send to right channel: 0x2000, 0x2001

Direct read I2S data:

```
i2s read a, b...
```

a - I2S Channel [1],
b - count of audio samples to capture. Requested count should not exceed 256 bytes.

Capture b samples (b/2 left + b/2 right) via I2S channel a.
Each sample should be 2 or 4 bytes wide depending on I2S configuration.
Even samples belongs to left channel, odd – to rigth.

Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
i2s read 1,64

Read 32 samples from left channel and 32 samples from rigth.

Write I2S data from SDRAM buffer:

```
i2s writebuffer a,b
```

a – I2S Channel [1],
b – Data size to be sent.

The data should be placed to SDRAM with "put buffer" command. Data size should not exceed size of available SDRAM buffer.

Command returns: OK in case of success, FAIL in case of error.

Example:
i2s writebuffer 1, 64

Write 64 samples from SDRAM buffer (32 to left channel. 32 to rigth).

Read I2S data to SDRAM buffe*r*:

```
i2s readbuf a,b
```

a – I2S Channel [1],
b – Data size to be read.

Received data are placed to SDRAM and are available with "get buffer" command. Requested data size should not exceed size of available SDRAM buffer.

Command returns: OK in case of success, FAIL in case of error.

Example:
i2s readbuf 1, 64

Receives 32 samples from right channel and 32 from left.

Write and read I2S data from/to SDRAM buffer *(write data will be replaced with received data):*

```
i2s wreadbuf a,b
```

a – I2S Channel [1],
b – Data size to be sent/read.

Send data should be placed to SDRAM with “put buffer” command, received data are available with “get buffer” command. Data size should not exceed size of available SDRAM buffer.

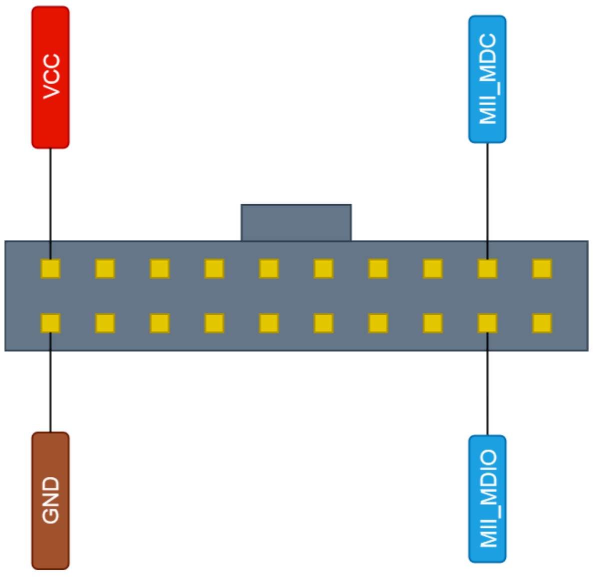
Command returns: OK in case of success, FAIL in case of error.

Example:
i2s wreadbuf 1, 64

Sends 32 samples to right channel, 32 samples to left, while receiving 32 samples from right channel and 32 from left.

MDIO bus

MDIO bus pinout:



MDIO read register:

```
mdio read a,b
```

a - PHY address;
b - Register address;

Reads 16-bit register in specified PHY.
Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
mdio read 1, 10

Read register 10 in PHY 1.

MDIO write register:

```
mdio write a,b,c
```

a - PHY address
b - Register address
c - Data value to be written

Writes 16-bit value to PHY's register.

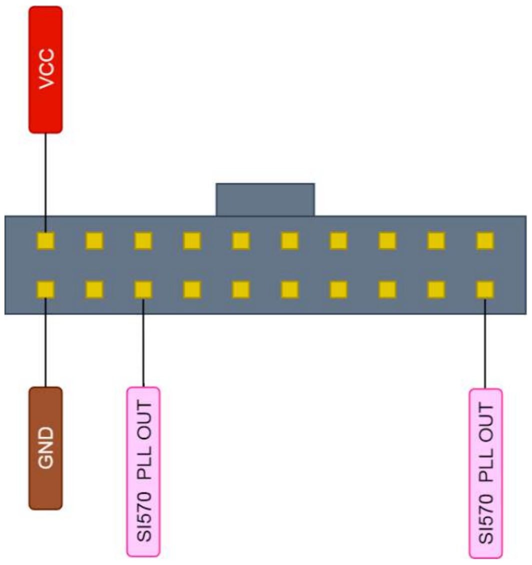
Command returns: OK in case of success, FAIL in case of error.

Example:
mdio write 1, 10, 0xCAFE

Write 0xCAFE to register 10 in PHY 1.

Frequency generator (Si570 PLL)

Frequency generator pinout:



Set PLL output frequency:

```
pll setfreq a
```

a – Frequency in Hz.

PLL able to work in range from 10MHz to 945MHz (limited to 350MHz).

Command returns: OK in case of success, FAIL in case of error.

Example:
pll setfreq 50000000

Set pll output to 50MHz

Get current PLL frequency:

```
pll getfreq
```

Requests current value of output frequency in Hz.

Command returns: OK, <data> in case of success, FAIL in case of error.

Reset PLL:

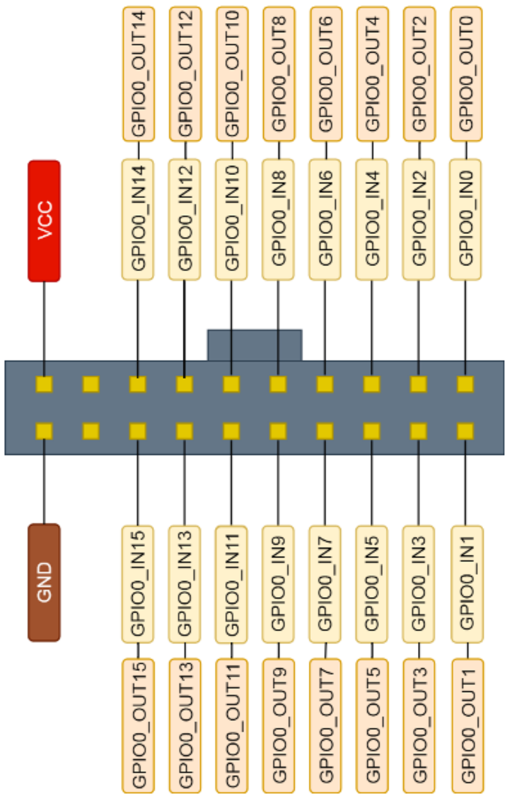
```
pll reset
```

Resets PLL to default state.

Command returns: OK in case of success, FAIL in case of error.

General purpose I/O (GPIO)

GPIO pinout:



Set GPIO value:

```
gpio write a
```

a – 16-bit value (16-pins).

Writes constant value directly to output pins of external connector. Note, that pin function should be set to GPIO_OUT.

Command returns: OK in case of success, FAIL in case of error.

Example:
gpio write 0xFF00

Write "1" to upper 8 pins, "0" to lower 8.

Get GPIO value:

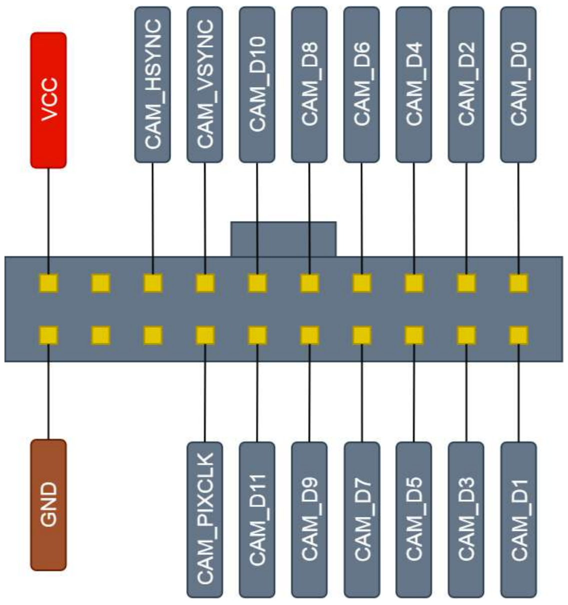
```
gpio read
```

Reads state on external connector pins. Note, that the value represents pins state regardless of pins functions.

Command returns: OK,<data> in case of success, FAIL in case of error.

Camera interface (parallel camera interface)

Camera interface pinout:



Setup CMOS camera interface and start capture process:

```
camera start a,b,c,d,e
```

a – Pixel clock polarity (1 - Rising, 0 – Falling),
b – HSYNC line polarity (1 - HSYNC polarity high, 0 - HSYNC polarity low),
c – VSYNC line polarity (1 - VSYNC polarity high, 0 - VSYNC polarity low),
d – Frame size in quadwords (4 bytes),
e – 0 - single snapshot, 1 - continuous mode.

Captured frame will be stored in SDRAM.

Command returns: OK in case of success, FAIL in case of error.

Example:
camera start 1, 1, 1, 10000, 0

Capture single frame with rising pixel clock, positive HSYNC and VSYNC, frame size = 10000,

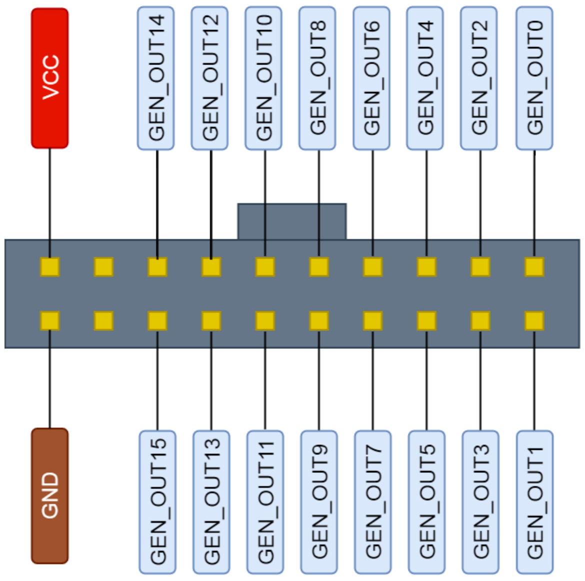
Stop frame capture process:

```
camera stop
```

Command returns: OK in case of success, FAIL in case of error.

Waveform generator (WFG)

Waveform generator pinout:



Configure WFG:

```
waveform gen configure
```

Settings and waveform data should be placed to SDRAM with "put buffer" command.
The structure is (offset in bytes, LITTLE_ENDIAN):

0 - 3 : sample rate in nanoseconds. Should be multiply of 5ns
4 - 7 : segment length in 16-bit words
8 - 11 : flags

flags[1 : 0] - sync mode.

00 - SW
01 - external rising edge
10 - external falling edge

flags[2] - cycle mode

0 - waveform is repeated continuously
1 - waveform is generated only once. In external sync mode, the waveform is started again in next trig edge

12 - 15: initial value

[15 : 0] - initial value, which is placed to IO lines before generation begins

16 - 19 : Reserved
20 - 23 : Reserved
24 - 27 : Reserved
28 - 31 : Reserved
32 - .. waveform data - array of 16-bit words. Each bit feeds corresponding bit in external connector, configured to function "Waveform generator".
See description of WFG for parameters range.

Command returns: OK in case of success, FAIL in case of error.

Start WFG:

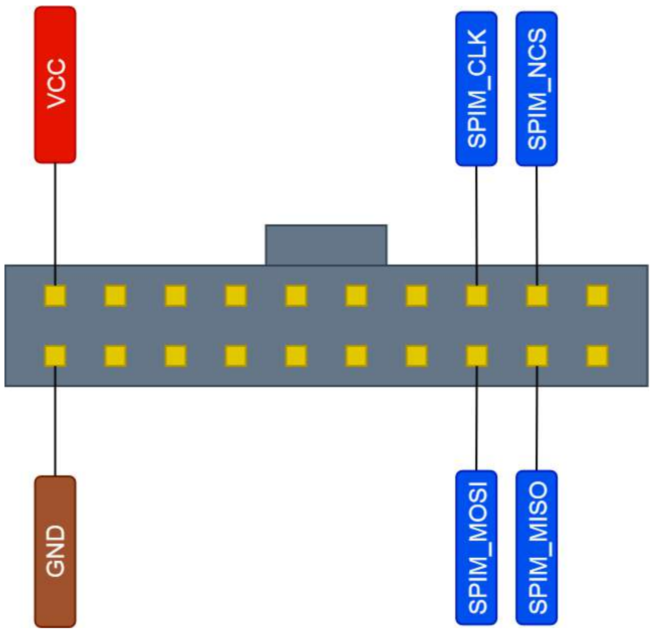
```
Waveform gen start
```

When configured, WFG can be start with this command. It can be repeated without re-configuration.

Command returns: OK in case of success, FAIL in case of error.

FPGA SPI Master (SPIM)

SPI master (FPGA-based) bus pinout:



Configure SPIM:

```
spimaster configure
```

Settings and SPI data should be placed to SDRAM with "put buffer" command.
The structure is (offset in bytes, LITTLE_ENDIAN):

0 - 3 : SCLK period in nanoseconds[31 : 0]. Should be multiply of 20ns
4 - 7 : SPI transactions rate in nanoseconds [31 : 0]. Should be multiply of 5ns
8 - 11 : Config[31 : 0]

Config[0] - Mode.

1 - Cyclic mode. All data are placed to memory before "spimaster start" command. After "spimaster start" they are transmitted
over SPI. The buffer starts from beginning after all data are transmitted.
It is not possible to add new data after "start" command.
0 - FIFO mode. Data can be placed to memory before "spimaster start". During transmission, new data can be added to
buffer.

Config[1] - Stop when empty.

0 - If buffer is empty, the very last word is repeated
1 - If buffer is empty, the transmission stops

Config[12 : 8] - bit count in SPI transaction
Config[15] - SCLK polarity

12 - 15 : data count (in 16 bit words)
16 - 19 : Reserved
20 - 23 : Reserved
24 - 27 : Reserved

28 - 31 : Reserved
32 - .. SPI data, if exist.

See description of SPIM for parameters range.
This command should be executed before using FPGA SPI Master.

Command returns: OK in case of success, FAIL in case of error.

Start SPIM:

```
spimaster start
```

FPGA SPI Master starts transmission with parameters set with "spimaster configure" command. It can be repeated without re-configuration.
External connector pins should be configured to "SPI Master".

Command returns: OK in case of success, FAIL in case of error.

Add to SPIM buffer:

```
spimaster write buffer a
```

a - data word is inserted to buffer for transmission.
Used when SPI Master works in FIFO mode.

Command returns: OK in case of success, FAIL in case of error.

Send single word over SPIM:

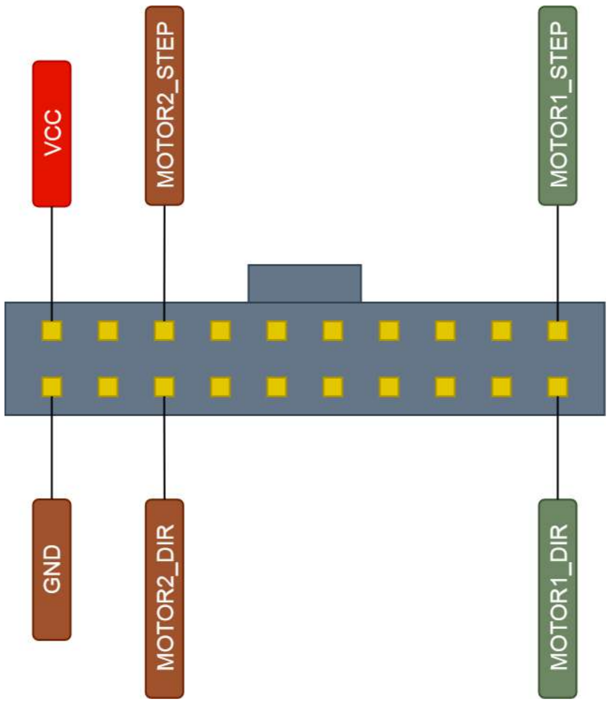
```
spimaster send single a
```

a - data word
SPI settings should be done by "spimaster configure" command.
"Mode" and "Stop when empty" parameters are temporary set to 0 and 1 respectively
External connector pins should be configured to "SPI Master".

Command returns: OK in case of success, FAIL in case of error.

Step motor controller (SMC)

SMC interface pinout:



Configure SMC

```
stepmotor configure a, b, c, d, e, f
```

a - channel. 0 - all, 1 - 1, 2 - 2
b - min pulses speed (frequency) in Hz (1Hz min);
c - max pulses speed (frequency) in Hz (250 Khz max);
d - acceleration (speed change);
e - total pulses count. Motor will be moved by this count of steps;
f - direction. 1 - CW, 0 – CCW.

Configures step motor controllers. Both controllers can be configured at the same time with channel parameter = 0.

Command returns: OK in case of success, FAIL in case of error.

Example:
stepmotor configure 1, 100, 500, 100, 1000, 1

Configure SMC to start from 100 Hz, raise speed to 500 Hz with 100Hz/s rate, then work with constant speed 500Hz and finally slow down to 100Hz with 100Hz/s rate. Total number of steps is 1000, CW direction.

Start SMC

```
stepmotor start a
```

a - channel. 0 - all, 1 - 1, 2 - 2

Command returns: OK in case of success, FAIL in case of error.

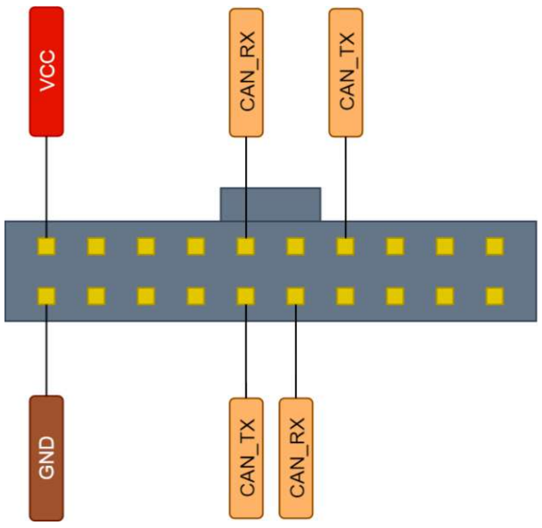
Example:
```
stepmotor start 0
```

Starts both channels simultaneously with parameters, specified by “step motor configure”.

CAN-bus interface (with FDCAN capability)

CAN module in USB2IO are compliant with ISO 11898-1: 2015 (CAN protocol specification version 2.0 part A, B) and CAN FD protocol specification version 1.0.

CAN-bus interface pinout:



Configure CAN interface

```
fdcan configure a, b
```

a – baud rate
b – reserved, 0

Configures fdcan with baud rate;.
Command returns: OK in case of success, FAIL in case of error.

Example:
fdcan configure 50000, 0

Configures with 50Kbit rate.

Send message over CAN-bus

```
fdcan send type id, data
```

type – 'std' or 'ext' identifier type (standard or extended)
id – message identifier,
data – comma separated array of message data.

Message id and data count should match type of the message.

Command returns: OK in case of success, FAIL in case of error.

Example:
fdcan sent ext 0x1BAC1234, 10, 12, 13

Sends extended frame with 0x1BAC1234 id and {10, 12, 13} message

Read single message from CAN-bus (without timeout)

```
fdcan recv single
```

Command returns: “type:id message” in case of success, FAIL in case of error.

Example:
fdcan read

>STD: 123 00 02

Receiving message with standard identifier 0x123 and message data [0x00, 0x02].

After timeout time, device will send DONE message.

Read message from CAN-bus within timeout

```
fdcan recv single timeout a
```

a - timeout

Requests one message from receive queue. USB2io device should wait for message no longer than ‘a’ milliseconds.

Command returns: “type:id message” in case of success, FAIL in case of error.

Example:
fdcan read timeout 1000

>STD: 123 00 02

Receiving message with standard identifier 0x123 and message data [0x00, 0x02].

Write/Read messages from SDRAM buffer over FDCAN

```
fdcan buffer a, b
```

a – count of messages in SDRAM buffer
b - timeout

Sends specified number of messages from SDRAM buffer. USB2IO device must finish transmission in “b” milliseconds, otherwise the function will fail. In parallel, incoming messages are stored in SDRAM buffer and are available after command execution with “get buffer” command.

Messages in SDRAM buffer should have following structure:
0xC0 - start marker
0x00 or 0x01 - type (0 - std, 1 - extended)
ID - identifier. 2 bytes for std, 4 bytes for ext
NN - data count
DT – payload

The should be places to SDRAM with ‘put buffer’ command.

Received messages have slightly different structure in SDRAM buffer:

0x0C -start marker
time stamp, 4 bytes
ID type (0 - std, 1 - extended)
ID - identifier. 2 bytes for std, 4 bytes for ext
data count (0 - 64)
message data

Command returns: OK.<bytes_count_in_buffer> in case of success, FAIL in case of error.

Example:
fdcan buffer 4, 1000

>OK.24

Send 4 messages from buffer for time not longer than 1000 ms. 24 bytes are in buffer with received messages.

Write/Read messages from SDRAM buffer over FDCAN synchronously

```
fdcan writebuffersync a, b
```

a – count of messages in a packet
b – count of packets is SDRAM buffer
c – trigger timeout

Sends CAN messages from SDRAM as packets of messages. Each packet is sent as soon as trigger event is received.
Messages are placed to SDRAM buffer with “put buffer” command. The structure is the same as with “fdcan buffer” command.
Trigger should be configured with “trigger configure" command.

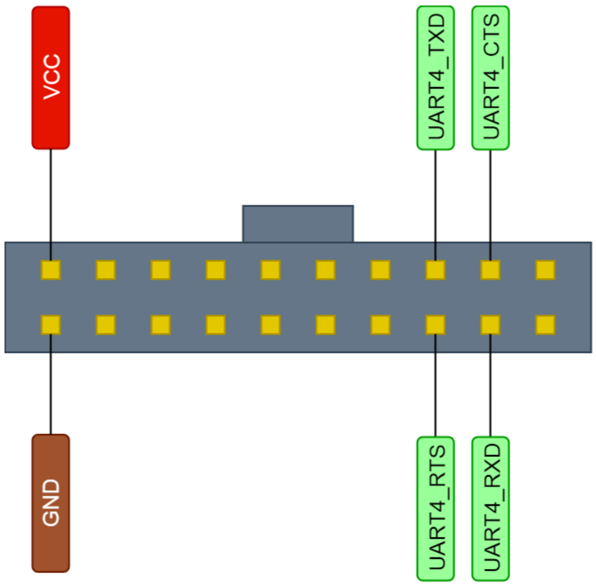
Command returns: OK in case of success, FAIL in case of error.

Example:
fdcan writebuffersync 4, 10, 1000

Send 10 packets with 4 messages each. Wait for trigger event not longer than 1000 ms.

Serial interface UART (Universal Asynchronous Receiver/Transmitter)

Serial interface UART pinout:



Configure UART

```
uart configure a, b, c, d, e
```

a – channel
b – baud rate
c – word length
d – stop bits count
e - parity

Configures uart channel with specified parameters.
There are two available channels for UART communication: 1 and 4.
Word length defines bit count in data word. Possible values are 7, 8, 9.
Stop bits count parameter could be 0 - 0.5 stop bit, 1 – 1 stop bit, 2 – 1.5 sop bits, 3 – 2 stop bits.
Parity 0 – no parity, 1 – even, 2 – odd.

Command returns: OK in case of success, FAIL in case of error.

Example:
uart configure 1, 50000, 8, 1, 0

Configure channel 1 with 50Kbaud, 8 bit word, 1 stop bit without parity.

Direct write UART data

```
uart write a, b
```

a – UART channel
b – data word

Send one word over selected UART channel.

Command returns: OK in case of success, FAIL in case of error.

Example:
uart write 1, 0xAA

Direct read UART data

```
uart read a, b
```

a – UART channel
b – timeout

Reads one data word from receive queue of selected channel. USB2IO device waits for receive data not longer that “b” milliseconds.

Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
uart read 4, 1000
>OK, 1F

Write UART data from buffer

```
uart writebuffer a, b
```

a – UART channel [1];
b – words count.

Sends to selected channel ‘b’ words from SDRAM buffer. Data should be places to SDRAM with ‘put buffer’ command.

Command returns: OK in case of success, FAIL in case of error.

Example:
uart writebuffer 1, 100

Send 100 words from SDRAM buffer over channel 1.

Write UART data from buffer and read to buffer simultaneously

```
uart wreadbuffer a, b
```

a – UART channel;
b – words count

Sends to selected channel ‘b’ words from SDRAM buffer. Data should be places to SDRAM with ‘put buffer’ command. While sending, receive queue is checked and received messages are stored in SDRAM buffer. After transmission finish, received data are available with “get_buffer” command.

Command returns: OK,<rx_count> in case of success, FAIL in case of error.
rx_count – count of received words.

Example:
uart wreadbuffer 1, 100

>OK,20

Send 100 words from SDRAM buffer over channel 1. 20 words are received.

Write UART data from buffer synchronously

```
uart writebuffersync a, b, c, d
```

a – UART channel
b – packet length
c – packet count
d – trigger timeout

Data are sent to selected channel in packets with ‘b’ length. ‘c’ is a count of packets. Each packet is sent when trigger event occurs.
Data should be places to SDRAM with ‘put buffer’ command.
Trigger should be configured with “trigger configure" command.
USB2IO device waits for trigger event for time not longer than ‘d’ milliseconds.

Command returns: OK in case of success, FAIL in case of error.

<u>Example</u>:
uart writebuffersync 1, 4, 10, 1000

Send 10 packets with 4 words each from SDRAM buffer over channel 1. Wait for trigger not longer than 1000 ms.

Read UART data to buffer:

```
uart readbuffer a, b, c
```

a – UART channel
b – words count
c - timeout

Received words are placed to SDRAM buffer. They are available with ‘get buffer’ command.
USB2IO device collects data not longer than ‘c’ milliseconds.

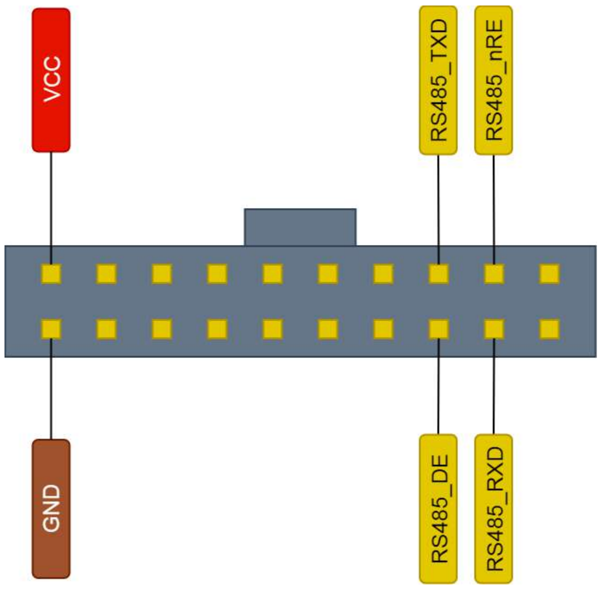
Command returns: OK in case of success, FAIL in case of error.

<u>Example</u>:
uart readbuffer 1, 40, 1000

Reads 40 bytes over channel 1

RS-485 serial bus

RS0485 interface pinout:



Configure RS-485

```
rs485 configure a, b, c, d, e
```

a – channel [4];
b – baud rate;
c – word length;
d – stop bits count;
e – parity.

Configures uart channel with specified parameters.
RS-485 interface is available on channel 4 only. All commands to RS-485 unit should use this channel.
Word length defines bit count in data word. Possible values are 7, 8, 9.
Stop bits count parameter could be 0 - 0.5 stop bit, 1 – 1 stop bit, 2 – 1.5 sop bits, 3 – 2 stop bits.
Parity 0 – no parity, 1 – even, 2 – odd.

Command returns: OK in case of success, FAIL in case of error.

Example:

rs485 configure 4, 50000, 8, 1, 0

Configure with 50Kbaud, 8 bit word, 1 stop bit without parity.

Direct write RS-485 data

```
Rs485 write a, b
```

a – channel [4];
b – data word.

Send one word over RS-485 channel.

Command returns: OK in case of success, FAIL in case of error.

Example:
rs485 write 4, 0xAA

Direct read RS-485 data

```
rs485 read a, b
```

a – channel [4]
b – timeout

Reads one data word from receive queue. USB2IO device waits for data not longer that “b” milliseconds.

Command returns: OK, <data> in case of success, FAIL in case of error.

Example:
rs485 read 4, 1000
>OK, 1F

Write RS-485 data from buffer

```
rs485 writebuffer a, b
```

a – channel [4];
b – words count

Sends to channel ‘b’ words from SDRAM buffer. Data should be places to SDRAM with ‘put buffer’ command.

Command returns: OK in case of success, FAIL in case of error.

Example:
rs485 writebuffer 4, 100

Send 100 words from SDRAM buffer.

Write RS-485 data from buffer and read to buffer simultaneously

```
rs485 wreadbuffer a, b
```

a – channel [4];
b – words count

Sends to selected channel ‘b’ words from SDRAM buffer. Data should be places to SDRAM with ‘put buffer’ command. While sending, receive queue is checked and reveived messages are stored in SDRAM buffer. After transmission finish, received data are available with “get_buffer” command.

Command returns: OK,<rx_count> in case of success, FAIL in case of error.
rx_count – count of received words.

Example:
rs485 wreadbuffer 4, 100

>OK,20

Send 100 words from SDRAM buffer. 20 words are received.

Write RS-485 data from buffer synchronously

```
rs485 writebuffersync a, b, c, d
```

a – channel[4];
b – packet length;
c – packet count;
d – trigger timeout.

Data are sent to channel in packets with ‘b’ length. ‘c’ is a count of packets. Each packet is sent when trigger event occurs.
Data should be places to SDRAM with ‘put buffer’ command.
Trigger should be configured with “trigger configure" command.

USB2IO device waits for trigger event for time not longer than ‘d’ milliseconds.

Command returns: OK in case of success, FAIL in case of error.

Example:
rs485 writebuffersync 4, 4, 10, 1000

Send 10 packets with 4 words each from SDRAM buffer. Wait for trigger not longer than 1000 ms.

Read RS-485 data to buffer:

```
rs485 readbuffer a, b, c
```

a – channel[4];
b – words count;
c – timeout.

Received words are placed to SDRAM buffer. They are available with ‘get buffer’ command.
USB2IO device collects data not longer than ‘c’ milliseconds.

Command returns: OK in case of success, FAIL in case of error.

Example:
rs485 readbuffer 4, 40, 1000

Reads 40 bytes over rs-485 channel.

External Trigger

Trigger configure:

```
trigger configure a
```

a – GPIO lane

Some commands rely on trigger events during execution.
It is possible to assign source of the trigger signal to any of 16 external connector GPIO lanes.
If certain lane is selected as trigger source, it is linked to trigger input of STM32 processor and rising edge generates trigger event.

Signal from GPIO lane is fed to STM32 regardless of function of corresponding GPIO pin. For example, pin function can be set to WFG output, and waveform on that WFG output will define trigger events.

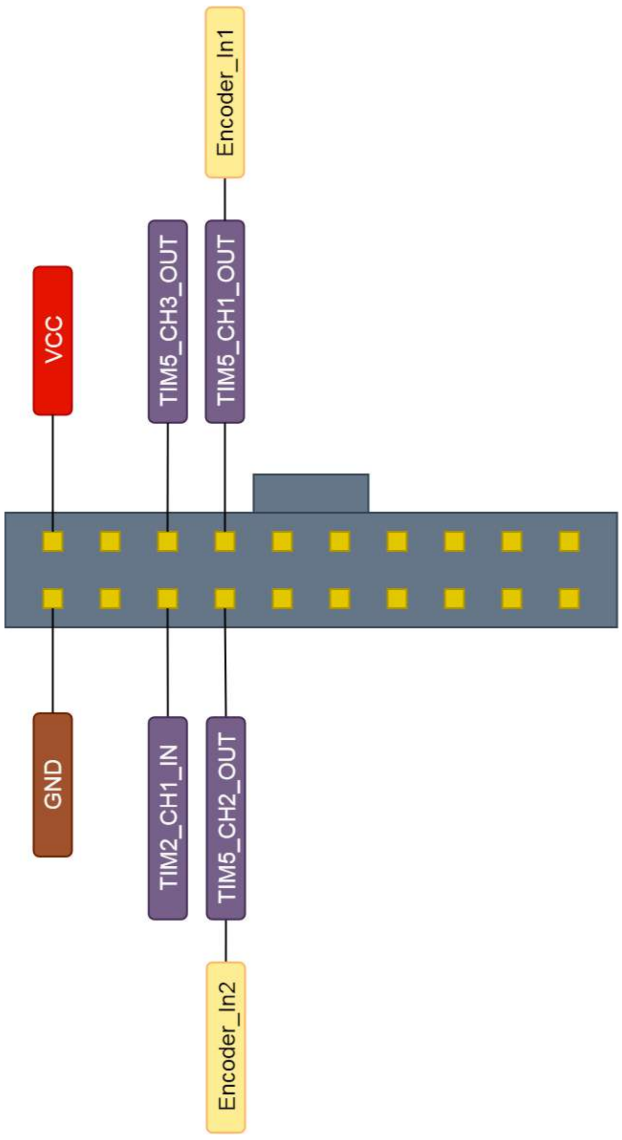
Command returns: OK in case of success, FAIL in case of error.

Example:
trigger configure 9

Sets GPIO9 (pin10) as trigger source.

Timers (with PWM and encoder functionality)

Timer pinout:



PWM generation on timer 5:

```
tim5 pwm start a, b, c
```

a – channel (1 - 3);
b – frequency;
c – duty cycle (0 – 100%).

Timer5 has 3 outputs (1 - 3) which are fed to external connector. It is possible to generate pwm signals on these channels. Important: all signals have the same frequency, because they are derived from the same counter.
Duty cycle specifies relationship between ''0"and '1' parts of the signal. 0 – means constant '0', 100 – constant '1', 50 – same width of '0' and '1'.
The USB2IO device tries to find frequency closer to requested frequency first, and then sets up duty cycle. That's why in some situations the duty cycle will be inaccurate.

Command returns: OK in case of success, FAIL in case of error.

Example:
tim5 pwm start 1, 10000, 50

Starts PWM generation on channel 1 with 10KHz frequency and 50% duty cycle.

Enable encoder on timer 5:

```
tim5 encoder start a
```

a – mode

Timer5 can be used to count pulses from external encoder. 2 lanes are used.
There are 3 modes of operation:
0 – pulses on lane A are counted, direction of count is set by lane B
1 – pulses on lane B are counted, direction of count is set by lane A
2 – pulses on lane A and on lane B are counted, direction depends on state of opposite lane.

Command returns: OK in case of success, FAIL in case of error.

Example:
tim5 encoder start 2

Starts encoder in mode 2.

Get state of encoder on timer 5:

```
tim5 encoder get
```

Requests current state of encoder counter and direction of count.

Command returns: OK,<counter>, <direction> in case of success, FAIL in case of error.

Example:
```
tim5 encoder get
>OK, 1000, Up
```

Counter value is 1000, encoder counts up.

Service functions

Firmware version and link test:

```
hello
```

Command returns string with uC version, FPGA version and communication test result, like :

```
USB2IO, uC Ver 1.36.0 Aug 9 2020, FPGA Ver 0x16, IF test: Passed
```

Reboot the device

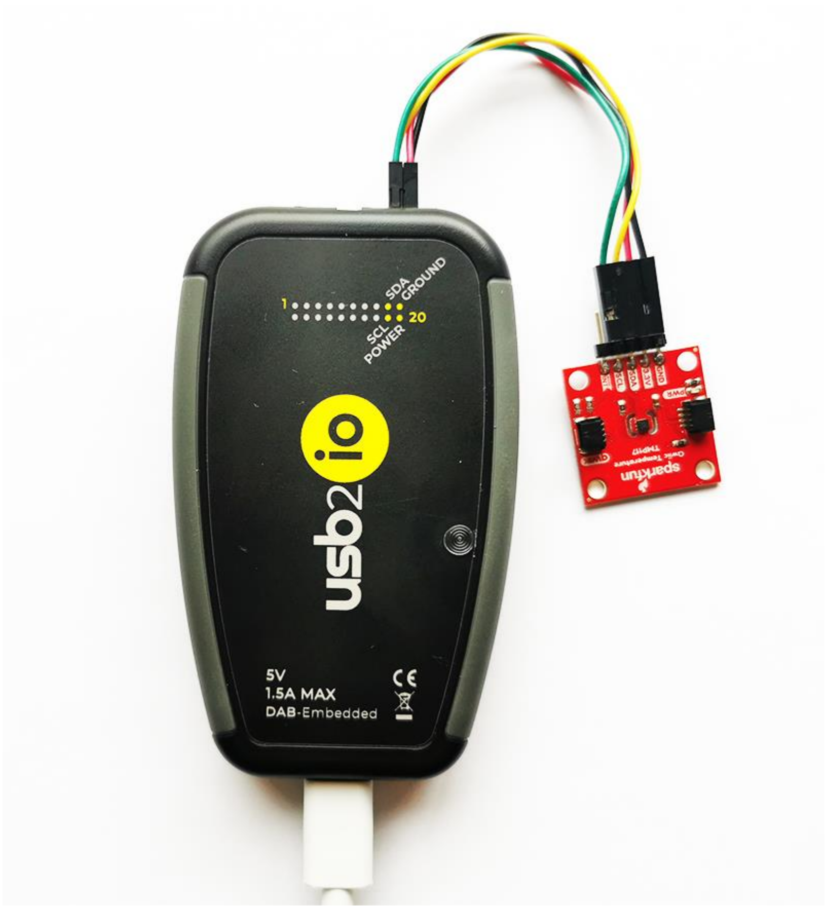
```
reboot
```

Command does not return anything.

Usecases

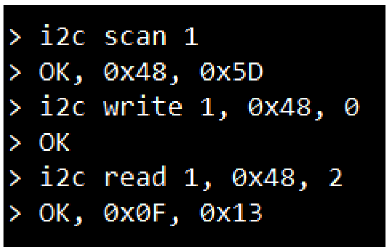
I2C bus usecases

Communication with temperature sensor TMP117 over STM32 channel:



In terminal:

```
# Scan i2c bus, type:
i2c scan 1<Enter>
# Response from USB2IO
OK,0x48,0x5D
# Ok, 0x48 is TMP117. Let’s read RAW temperature value from register 0.
# For that need set register to 0
i2c write 1, 0x48,0<Enter>
# Response:
OK
# Means device ACKed our write request
# Now ready to read back RAW value (16-bit = 2 bytes):
i2c read 1, 0x48,2<Enter>
# Response from TMP117:
OK,0x0F,0x13
```



```
> i2c scan 1
> OK, 0x48, 0x5D
> i2c write 1, 0x48, 0
> OK
> i2c read 1, 0x48, 2
> OK, 0x0F, 0x13
```

Python (2.7) script (i2c_tmp117.py):

(Python script required Python 2.7 and PySerial package to be preinstalled):

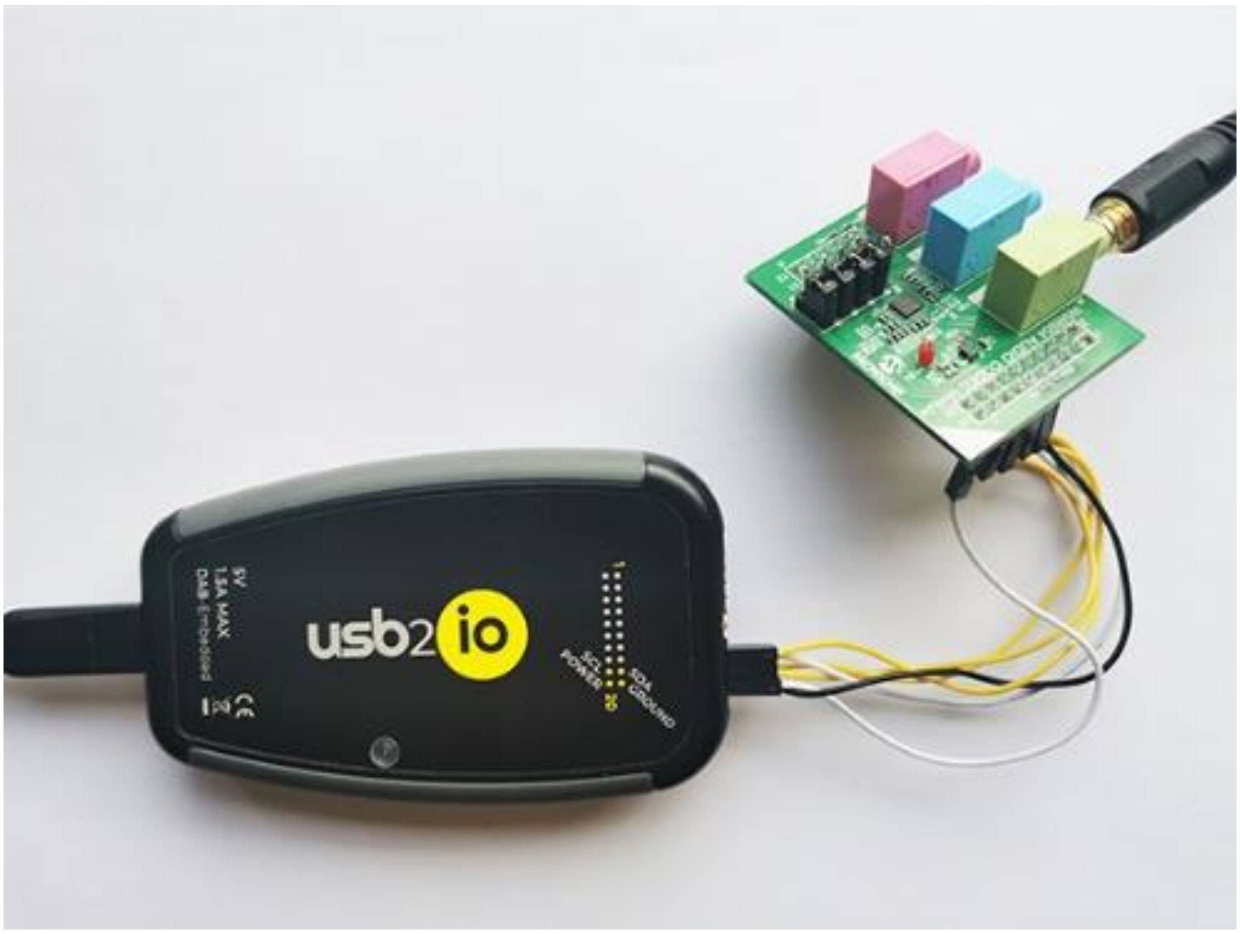
```
#======================================================
#  TMP116 functional test
#
#
#======================================================
import os
import sys
import time
import serial
import select
import ctypes as ct
from usb2io import usb2io_sendcmd

COM_PORT_ENG                = 'COM7'

TMP117_SLAVE_ID             = 0x48

TMP117_TEMP_RESULT          = 0x00
TMP117_CONFIGURATION        = 0x01,
TMP117_T_HIGH_LIMIT         = 0X02,
TMP117_T_LOW_LIMIT          = 0X03,
TMP117_EEPROM_UL            = 0X04,
TMP117_EEPROM1              = 0X05,
TMP117_EEPROM2              = 0X06,
TMP117_TEMP_OFFSET          = 0X07,
TMP117_EEPROM3              = 0X08,
TMP117_DEVICE_ID            = 0X0F

#---- Application ---------------------------------------------------------
print "==================================="
print "* Temp Reading"
print "==================================="

local_com_port = sys.argv[1] if (len(sys.argv) > 1) else COM_PORT_ENG

def tmp117_reg_read(ser, reg):
           strx = "i2c write 1, " + hex(TMP117_SLAVE_ID) + "," + hex(reg & 0xFF) + "\r\n"
           wro = ser.write(strx)
           time.sleep(0.02)
           rrd = ser.readline()
           if (rrd.find('FAIL') != -1):
                      return (rrd)
           strx = "i2c read 1, " + hex(TMP117_SLAVE_ID) + ",2\r\n"
           wro = ser.write(strx)
           time.sleep(0.02)
           rrd = ser.readline()
           return (rrd)

def tmp117_reg_write(ser, reg, value):
           strx = "i2c write 1, " + hex(TMP117_SLAVE_ID) + "," + hex(reg & 0xFF) + "," + hex(reg >> 8) + "," + hex(reg & 0xFF) + "\r\n"
           wro = ser.write(strx)
           time.sleep(0.02)
           rrd = ser.readline()

def tmp117_reg_temperature(ser):
           str_ret = tmp117_reg_read(ser, TMP117_TEMP_RESULT)
           if (str_ret.find('FAIL') != -1):
                      return (0)
           ba1 = [int(i,16) for i in str_ret.replace('OK,', '').split(',')]
           dtemp = (ba1[0] << 8) | ba1[1]
           temp_res_d = ct.c_int16(dtemp)
           temp_res_f = float(temp_res_d.value) * 0.0078125
           return (temp_res_f)

# Open serial port
print "Open serial port " + local_com_port
try:
    ser = serial.Serial(local_com_port, 115200, timeout=3)
    print "COM port OK"
except:
    print "Error COM port"
    sys.exit(0)

print "Temp: "
print(tmp117_reg_temperature(ser))


# Close serial port
ser.close()
```

SPI bus usecase

Example of reading data from SPI Flash memory chip:

I2S (Audio) bus usecases

Playing music with AK4954 codec:



Camera usecases

Capturing RAW12 images using OV2640 camera:

OV 9655 2640
SDA
GROUND
SCL
POWER
20
1
usb2io
5V
1.5A MAX
DAB-Embedded

RS485, CAN 2.0 (FD) and RS232 usecases

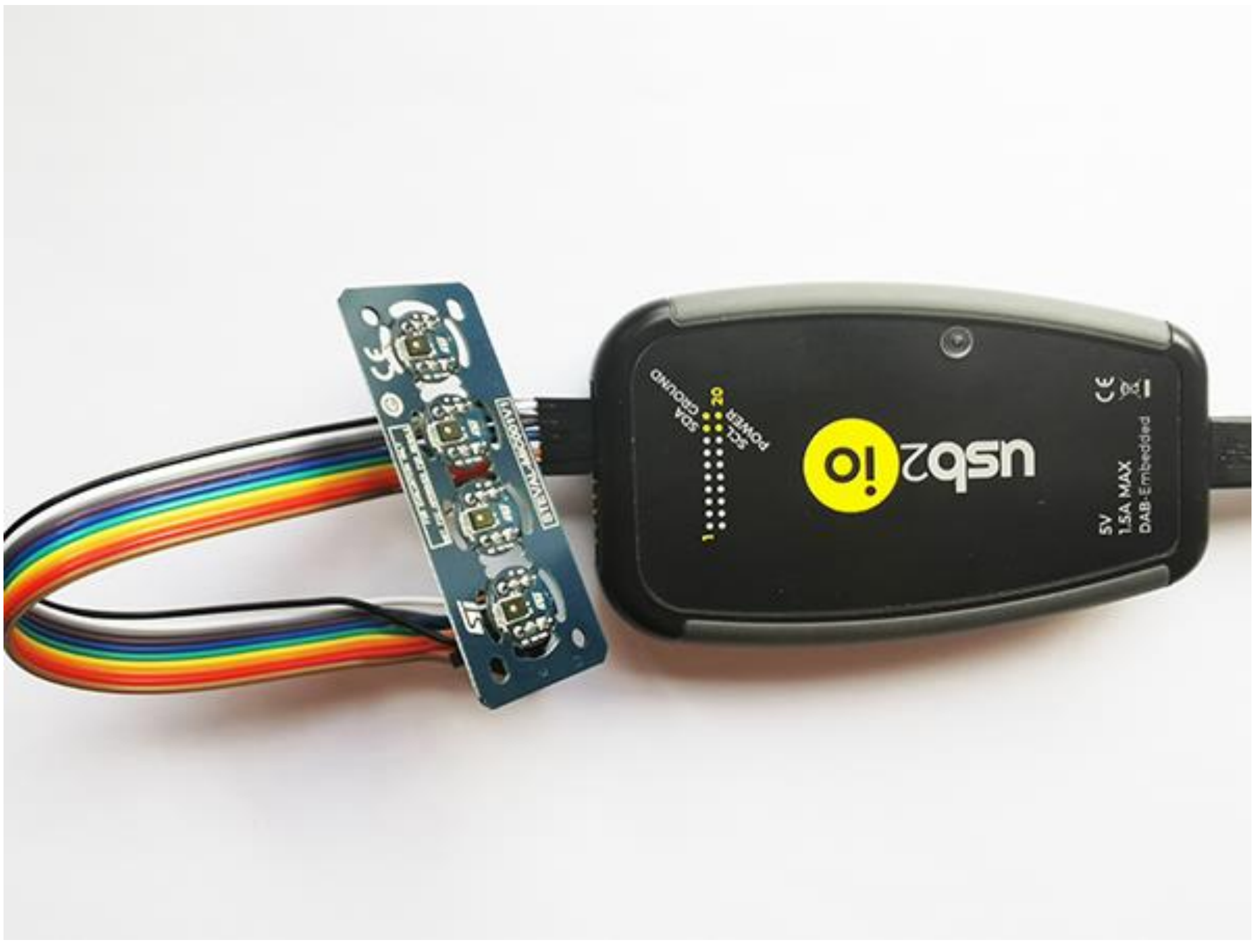
Example of communication over RS485 bus, CAN 2.0 (FD) bus and RS232 interface:

Digital microphone usecase

Example of recording data from digital I2S microphone:

Digital PDM microphone usecase

Example of recording data from digital PDM microphone:



Fiber optic transceiver usage usecase

Example of data exchange over PoF fiber using Broadcom transciever:

Regulatory

USB2IO device has been passed tests according EN 61326-1:2013 standard.

Enclosure Port Radiated Emission EN 55011:2009 + A1:2010 (30MHz-1GHz, Class A) – passed.
Enclosure Port Electrostatic Discharge EN 61000-4-2:2009 – passed.
Radiated Immunity EN 61000-4-3:2006 + A1:2008 + A2:2010 – passed.

Operating temperature

USB2IO has commercial operation temperature grade: 0 °C .. +70 °C .

Dimensions

USB2IO device has dimensions: 100x64x18 mm.

Box has dimensions: 150x100x50 mm.

Weight

USB2IO device weight (without cables): 60 grams.

Box weight: 200 grams.

Order code

Order code	
DAB-001001	USB2IO device

GTIN: 05419980085405

5 419980 085405 >

Legal