Application Note for Liquid Flow Sensors

Implementation Guide to the SHDLC Protocol for the RS485 Sensor Cable

Summary

This document describes the main features of the Sensirion High-level Data Link Control (SHDLC) protocol and provides a guide on how to implement the protocol on a controller system (master) for the communication with a single SHDLC device (slave).

Introduction

This document describes the general implementation of the SHDLC protocol. Consult the RS485 Sensor Cable SHDLC Command Reference (RS485_Sensor_Cable_SHDLC_Commands_EN_x_D1) for detailed information on individual commands.

RS485 Sensor Cable Hardware Settings

Communication Hardware

Compatible hardware configurations for use with the RS485 Sensor Cable include:

- PC with RS485 PCI board
- PC with USB to RS485 converter
- PC with RS232 to RS485 converter
- Microcontroller with UART (Universal Asynchronous Receiver/Transmitter) interface and RS485 transceiver
- PC with USB slot (when using the cable with the integrated USB-to-RS485 converter)

The RS485 Sensor Cable is available in 3 versions:

- RS485 side with open wire ends, article code 1-100804-01
- RS485 side with D-sub DE-9 connector and external power supply, article code 1-100839-01
- Cable with integrated USB-to-RS485 converter, article code TBD

Serial Port Configuration

The RS485 Sensor Cable uses the following settings:

- 115’200 baud (May be configured to baudrates between 1200 and 115’200)
- Half Duplex
- 8 Data bits, Least-significant bit (LSb) first
- No Parity
- 1 Stop bit

SHDLC Protocol

The Sensirion High-level Data Link Control (SHDLC) protocol is a master/slave protocol without the need for bus arbitration. It is based on a byte oriented, bidirectional interface without hardware handshaking.

Frame Definition

MOSI (Master Out Slave In) Frame

The following diagram shows the data flow for a MOSI (Master Out Slave In) frame:

	Frame Content					
Start (0x7E)	Adr 1 Byte	CMD 1 Byte	Length 1 Byte	Tx Data 0…255 Bytes	CHK 1 Byte	Stop (0x7E)

The MOSI frame consists of the following components:

- **Start** The byte 0x7E marks the beginning of the frame.
- **Adr** Device Address of the slave to which the frame is sent. Addresses 0...254 may be assigned to individual slaves, the address 255 is reserved for sending commands in broadcast mode to all slaves on the bus.
- **CMD** Command ID of the command which is sent to the slave device. See the RS485 Sensor Cable SHDLC Command Reference for details.
- **Length** Indicates the number of bytes sent in the Data block
- **Data** The data format depends on the command, see the RS485 Sensor Cable SHDLC Command Reference for details.
- **CHK** Check sum over the frame content.
- **Stop** The second byte 0x7E marks the end of the frame.

MISO (Master In Slave Out) Frame

The following diagram shows the data flow for a MISO (Master In Slave Out) frame:

	Frame Content						
Start (0x7E)	Adr 1 Byte	CMD 1 Byte	State 1 Byte	Length 1 Byte	Rx Data 0…255 Bytes	CHK 1 Byte	Stop (0x7E)

The MISO frame follows a similar structure as the MOSI frame:

- **Start** The byte 0x7E marks the beginning of the frame.
- **Adr** Device Address of the slave which is sending the frame.
- **CMD** Command ID of the command to which the slave device is responding. See the RS485 Sensor Cable SHDLC Command Reference for details.
- **State** The slave sends a state byte to report execution errors or communication errors to the master. The value 0x00 corresponds to 'no error'.
- **Length** Indicates the number of bytes sent in the Data block.
- **Data** The data format depends on the command, see the RS485 Sensor Cable SHDLC Command Reference for details.
- **CHK** Check sum over the frame content.
- **Stop** The second byte 0x7E marks the end of the frame.

Checksum

The checksum is calculated over the frame content in the following way:

- sum all bytes in the frame content (from and including Adr to and including Data)
- take the least significant byte of this sum
- invert the least significant byte

Example:

Send command 'Start Continuous Measurement' with sampling time 250 ms to Address 0:

Frame Content			
Adr	CMD	Length	Tx Data
`0x00`	`0x33`	`0x02`	`0x00, 0xFA`

frame content: `[0x00, 0x33, 0x02, 0x00, 0xFA]`
sum all bytes: `0x00 + 0x33 + 0x02 + 0x00 + 0xFA = 0 + 51 + 2 + 0 + 250 = 303 = 0x12F`
least significant byte: `0x12F & 0xFF = 0x2F` (the operator '&' stands for the bit-wise AND)
invert: `0x2F ^ 0xFF = 0xD0` (the operator '^' stands for the bit-wise XOR, 'exclusive OR')
Checksum: 0xD0

Byte Stuffing

Because there is no hardware handshaking, the frame start and stop are signaled by a unique data content:

- Start: `0x7E` (binary `01111110`)
- Stop: `0x7E` (binary `01111110`)

If this special start/stop byte (`0x7E`) occurs anywhere else in the frame (i.e. in the frame content or the checksum), it needs to be replaced. The same is true for 3 more special bytes: Escape (`0x7D`), XON (`0x11`) and XOFF (`0x13`).

If any of these 4 special bytes occur anywhere in the frame, they are replaced by `0x7D`, followed by the original byte with bit 5 inverted. See the following table:

Tab. 1: Byte Stuffing (transmission of special bytes)

Original byte	Transferred bytes
`0x7E`	`0x7D, 0x5E`
`0x7D`	`0x7D, 0x5D`
`0x11`	`0x7D, 0x31`
`0x13`	`0x7D, 0x33`

Example 1:

Send command 'Start Continuous Measurement' with sampling time 250 ms to address 0:

Frame Content				
Adr	CMD	Length	Tx Data	CHK
`0x00`	`0x33`	`0x02`	`0x00, 0xFA`	`0xD0`

Convert to byte array: `[0x00, 0x33, 0x02, 0x00, 0xFA, 0xD0]`

None of the special bytes (0x11, 0x13, 0x7D, 0x7E) occurs in the frame content or the checksum.

The following byte array is sent: `[0x7E, 0x00, 0x33, 0x02, 0x00, 0xFA, 0xD0, 0x7E]`

Example 2:

Send command 'Start Continuous Measurement' with sampling time 250 ms to address 17 (hex `0x11`):

Frame Content				
Adr	CMD	Length	Tx Data	CHK
`0x11`	`0x33`	`0x02`	`0x00, 0xFA`	`0xBF`

Note that the check sum has changed with respect to example 1.

Convert to byte array: `[`**`0x11`**`, 0x33, 0x02, 0x00, 0xFA, 0xBF]`

The special byte `0x11` appears in the byte array. It needs to be replaced by `0x7D, 0x31`:
`[`**`0x7D, 0x31`**`, 0x33, 0x02, 0x00, 0xFA, 0xBF]`
Note that the checksum (`0xBF` in this case) is computed before the byte stuffing, it remains therefore unchanged.

The following byte array is sent: `[0x7E, 0x7D, 0x31, 0x33, 0x02, 0x00, 0xFA, 0xBF, 0x7E]`

Example 3:

Send command 'Start Continuous Measurement' with sampling time 19 ms (hex `0x13`) to Address 0:

Frame Content				
Adr	CMD	Length	Tx Data	CHK
`0x00`	`0x33`	`0x02`	`0x00`, **`0x13`**	`0xB7`

Note that again the checksum has changed with respect to examples 1 and 2.

Convert to byte array: `[0x00, 0x33, 0x02, 0x00,` **`0x13`**`, 0xB7]`

The special byte `0x13` appears in the byte array. It needs to be replaced by `0x7D, 0x33`:
`[0x00, 0x33, 0x02, 0x00, 0x7D, 0x33, 0xB7]`
Note that the Length (here: `0x02`) of the data is computed before the byte stuffing, it remains therefore unchanged. Also the checksum remains unchanged as in example 2.

The following byte array is sent: `[0x7E, 0x00, 0x33, 0x02, 0x00, 0x7D, 0x33, 0xB7, 0x7E]`

Error Handling

There are 3 error modes for which error handling should be implemented on the master:

Error State

The master should recognize if an execution error has occurred on the slave device and the error state in the MISO frame is different from 0x00. See the RS485 Sensor Cable SHDLC Command Reference for errors codes and their descriptions.

MOSI checksum error

If the slave device receives a frame with an erroneous checksum (i.e. the check sum does not match the frame content) it will silently ignore the command, i.e. the slave will not send any reply to the master. To detect such errors it is necessary that the master always waits for a correct answer from the slave device before sending the next command. This is obvious when the master requests some data from the slave (e.g. when reading a measurement) but the reply should also be checked when the master expects no data (e.g. when starting a measurement on the device).

MISO checksum error

To detect communication errors, the master should always check that the incoming checksum matches the incoming frame content. If this is not the case, a communication error has occurred.

Possible causes for checksum errors include

- incorrect implementation of the checksum computation on the master
- overlapping commands. For instance, if the master sends the next command before the reply to the previous command has arrived, then the reply from the slave may overlap with that next command sent by the master.
- several devices on the bus have the same address and their replies to a command overlap.
- electrical interference from very harsh electromagnetic environments.

Data Types and Representation

The data in the frames is transmitted in big-endian order, i.e. Most-Significant Byte (MSB) first.

Integer

Integers can be transmitted as signed or unsigned integers. The following types of integers are used:

Tab. 2: Integer data types

Integer Type	Size	Range
unsigned, 8-bit (u8t)	1 Byte	0 … 2^{8}-1
unsigned, 16-bit (u16t)	2 Byte	0 … 2^{16}-1
unsigned, 32-bit (u32t)	4 Byte	0 … 2^{32}-1
unsigned, 64-bit (u64t)	8 Byte	0 … 2^{64}-1
signed, 8-bit (i8t)	1 Byte	-2^{7} … 2^{7}-1
signed, 16-bit (i16t)	2 Byte	-2^{15} … 2^{15}-1
signed, 32-bit (i32t)	4 Byte	-2^{31} … 2^{31}-1
signed, 64-bit (i64t)	8 Byte	-2^{63} … 2^{63}-1

Signed integers are represented according to the two's complement convention. This means that the *N*-bit binary representation of a negative number –*x* is the two's complement of that number's absolute value |-*x*|. The following recipes may be used to obtain the binary representations of negative numbers and to reconstruct the numerical value from the binary representations, respectively.

Find the N-bit signed integer representation m corresponding to a number x

```
if x<0:                          # if the number is negative
    m = (|x| ^ (2**N – 1))+1     # compute the two's complement of its absolute value
else:                            # else the number is positive
    m = x                        # no computation needed
```

Here the operator '| |' denotes the absolute value, '^' denotes the bit-wise XOR (exclusive OR), '**' denotes the power as in `2**3 = 8`.

Examples:

-7 as 8-bit signed integer:
`(|-7| ^ (2**8-1)) + 1 = (7 ^ (256-1))+1 = (7 ^ 255) +1 = 248 + 1 = 249 = 0xF9`

-7 as 16-bit signed integer:
`(|-7| ^ (2**16-1)) + 1 = (7 ^ 65535) + 1 = 65528 + 1 = 65529 = 0xFFF9`

Find the number x represented by the N-bit signed integer m

```
if m & 2**(N-1) == 2**(N-1):     # if the most-significant bit of m is '1', the number is negative.
    x = -((m ^ (2**N – 1)) + 1)  # compute the two's complement
else:                            # else the number is positive
    x = m                        # no computation needed
```

Here the operator '&' denotes the bit-wise AND, '^' denotes the bit-wise XOR (exclusive OR), '**' denotes the power as in `2**3 = 8`.

Examples:

Find the number represented by the 8-bit signed integer 0xF7:
`m=0xF7 = 247`
`247 & 2**7 == 2**7 : True`
therefore: `x= -((247 ^ (2**8-1)) + 1) = -((247 ^ 255) + 1) = -(8 + 1) = -9`

Find the number represented by the 16-bit signed integer represented by the bytes [0xF7, 0x34]:

```
m = 0xF7 * 2**8 + 0x34 = 247 * 256 + 52 = 63232 + 52 = 63284
63284 & 2**15 == 2**15 : True
```

therefore: `x= -((63284 ^ 65535) + 1) = -(2251 + 1) = -2252`

Boolean

A boolean is represented by 1 byte:

- False = 0
- True = 1…255

String

Strings are transferred as C-strings. This means in ASCII encoding, one byte per character and terminated with a final null-character (0x00). The first letter will be sent first.

Examples of Communication Sequences (Use Cases)

Device Reset (receive no data)

We want to send the command 'DeviceReset' to device 0.

- Consult the RS485 Sensor Cable SHDLC Command Reference:

5.1.3 DEVICE RESET

Device Reset			
Description	Execute a reset on the device. The device will reply and then do the reset. If the command is sent with broadcast, the reset is done immediately after reception of the command. Wait 100ms before sending the next command to give time to reboot.		
Command ID	0xD3	**for Sensor Type**	0, 1, 2
Access Level	0	**Availability**	Always
Response Time max	250ms	**Storage**	-
MOSI Data (0 Bytes)	no data		
MISO Data (0 Bytes)	no data		

- Build frame content:

Adr	CMD	Length	Data
`0x00`	`0xD3`	`0x00`	

- Compute the checksum over the frame content:
  sum all bytes in the frame content: `0x00 + 0xD3 + 0x00 = 0xD3`
  take the Least-Significant Byte (LSB): `0xD3`
  invert: `0x2C`

- Add checksum to frame content

Adr	CMD	Length	Data	CHK
`0x00`	`0xD3`	`0x00`		`0x2C`

- Convert to byte array: `[0x00, 0xD3, 0x00, 0x2C]`
- Byte stuffing
  check: none of the special characters (`0x11`, `0x13`, `0x7D`, `0x7E`) appears in the byte array.
  Byte array after byte stuffing: `[0x00, 0xD3, 0x00, 0x2C]`
- Add Start / Stop Bytes.

Byte array sent to Tx Buffer: `[0x7E, 0x00, 0x32, 0x00, 0xCD, 0x7E]`

Byte array received at Rx Buffer: `[0x7E, 0x00, 0xD3, 0x00, 0x00, 0x2C, 0x7E]`

- remove start and stop bytes: `[0x00, 0xD3, 0x00, 0x00, 0x2C]`
- Byte (un-)stuffing
  check for special characters marker (`0x7D`): No special characters marker.
  byte array after byte un-stuffing: `[0x00, 0xD3, 0x00, 0x00, 0x2C]`
- Now the byte array may be interpreted as frame content and checksum:

Adr	CMD	State	Length	Data	CHK
`0x00`	`0xD3`	`0x00`	`0x00`		`0x2C`

- remove checksum to obtain frame content

Adr	CMD	State	Length	Data
`0x00`	`0xD3`	`0x00`	`0x00`	

- compute checksum of received frame content
  sum all bytes `0x00 + 0xD3 + 0x00 + 0x00 = 0xD3`
  take LSB: `0xD3`
  invert: `0x2C`
  checksum of received frame content matches received checksum. OK.
- check: address in the received frame is the same as in the sent frame. OK
- check command ID in the received frame is the same as in the sent frame. OK
- check: State is `0x00` (no error). OK
- data length is `0x00`, so no Data is received.

Get Device Info (receive a string)

We want to send the command 'Get Device Information' to device 0 to retrieve the product name from the device.

- Consult the RS485 Sensor Cable SHDLC Command Reference:

5.1.1 GET DEVICE INFORMATION

Get Device Information			
Description	On this command, the device will return an identification string which contains device type, article code and serial number.		
Command ID	0xD0	**for Sensor Type**	0, 1, 2
Access Level	0	**Availability**	Always
Response Time max	1ms	**Storage**	-
MOSI Data	**Byte #**	**Description**	
	0	*Information Type : u8t* This parameter defines which information is requested: 1: Product Name → Name of the connected device 2: Article code 3: Serial number	
MISO Data	**Byte #**	**Description**	
	0 … n	*Identification : string* String which contains the requested information	

- Build frame content:

Adr	CMD	Length	Data
`0x00`	`0xD0`	`0x01`	`0x01`

- Compute the checksum over the frame content:
  sum all bytes in the frame content: `0x00 + 0xD0 + 0x01 + 0x01 = 0xD2`
  take the Least-Significant Byte (LSB): `0xD2`
  invert: `0x2D`
- Add checksum to frame content

Adr	CMD	Length	Data	CHK
`0x00`	`0xD0`	`0x01`	`0x01`	`0x2D`

- Convert to byte array: `[0x00, 0xD0, 0x01, 0x01, 0x2D]`
- Byte stuffing
  check: none of the special characters (`0x11`, `0x13`, `0x7D`, `0x7E`) appears in the byte array.
  Byte array after byte stuffing: `[0x00, 0xD0, 0x01, 0x01, 0x2D]`
- Add Start / Stop Bytes.

Byte array sent to Tx Buffer: `[0x7E, 0x00, 0xD0, 0x01, 0x01, 0x2D, 0x7E]`

Byte array received at Rx Buffer: `[0x7E, 0x00, 0xD0, 0x00, 0x7D, 0x33, 0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00, 0x45, 0x7E]`

- remove start and stop bytes: `[0x00, 0xD0, 0x00, 0x7D, 0x33, 0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00, 0x45]`
- Byte (un-)stuffing
  check for special characters marker (`0x7D`): Special character `0x7D` occurs:
  `[0x00, 0xD0, 0x00,` **`0x7D, 0x33,`** `0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00, 0x45]`
  replace `0x7D, 0x33` → `0x13` according to Tab. 2, above.

  byte array after byte un-stuffing: `[0x00, 0xD0, 0x00, 0x13, 0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00, 0x45]`

- Now the byte array may be interpreted as frame content and checksum:

Adr	CMD	State	Length	Data	CHK
0x00	0xD0	0x00	0x13	0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00	0x45

- remove checksum to obtain frame content

Adr	CMD	State	Length	Data
0x00	0xD3	0x00	0x00	0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00

- compute checksum of received frame content
  sum all bytes `0x00 + 0xD0 + 0x00 + 0x13 + 0x52 + ... + 0x00 = 0x6BA`
  take LSB: `0xBA`
  invert: `0x45`
  checksum of received frame content matches received checksum. OK.
- check: address in the received frame is the same as in the sent frame. OK
- check command ID in the received frame is the same as in the sent frame. OK
- check: State is `0x00` (no error). OK
- data length is `0x13`, so 19 bytes of Data have been received.
- Data = `[0x52, 0x53, 0x34, 0x38, 0x35, 0x20, 0x53, 0x65, 0x6E, 0x73, 0x6F, 0x72, 0x20, 0x43, 0x61, 0x62, 0x6C, 0x65, 0x00]`
- Translate the remaining bytes according to the ASCII encoding:

0x52	0x53	0x34	0x38	0x35	0x20	0x53	0x65	0x6E	0x73	0x6F	0x72	0x20	0x43	0x61	0x62	0x6C	0x65	0x00
R	S	4	8	5		S	e	n	s	o	r		C	a	b	l	e	

  The final Null character (0x00) is due to the definition as C-string. The product name is therefore
  `RS485 Sensor Cable`

Get Single Measurement (receive one i16t or u16t)

We want to send the command 'GetSingleMeasurement' to device 0, to read the measurement result of a previously started single measurement.

- Consult the RS485 Sensor Cable SHDLC Command Reference:

5.2.3 GET SINGLE MEASUREMENT

Get Single Measurement				
Description	Read out the measurement result from the sensor when the measurement is finished. A single measurement must be started before using the Start Single Measurement command.			
Command ID	**0x32**		**for Sensor Type**	0, 2
Access Level	0		**Availability**	After start single Measurement
Response Time max	1ms		**Storage**	-
MOSI Data (0 Bytes)	no data			
MISO Data (0 Bytes)	no data (measurement not yet finished or Error)			
MISO Data (2 Bytes)	**Byte #**	**Description**		
	0,1	*Measurement result : u16t/i16t (if measurement finished)*		

- Build frame content:

Adr	CMD	Length	Data
`0x00`	`0x32`	`0x00`	

- Compute the checksum over the frame content:
  sum all bytes in the frame content: `0x00 + 0x32 + 0x00 = 0x32`
  take the Least-Significant Byte (LSB): `0x32`
  invert: `0xCD`

- Add checksum to frame content

Adr	CMD	Length	Data	CHK
`0x00`	`0x32`	`0x00`		`0xCD`

- Convert to byte array: `[0x00, 0x32, 0x00, 0xCD]`
- Byte stuffing
  check: none of the special characters (`0x11`, `0x13`, `0x7D`, `0x7E`) appears in the byte array.
  Byte array after byte stuffing: `[0x00, 0x32, 0x00, 0xCD]`
- Add Start / Stop Bytes.

Byte array sent to Tx Buffer: `[0x7E, 0x00, 0x32, 0x00, 0xCD, 0x7E]`

Byte array received at Rx Buffer: `[0x7E, 0x00, 0x32, 0x00, 0x02, 0xFF, 0xC6, 0x06, 0x7E]`

- remove start and stop bytes: `[0x00, 0x32, 0x00, 0x02, 0xFF, 0xC6, 0x06]`
- Byte (un-)stuffing
  check for special characters marker (`0x7D`): No special characters marker.
  byte array after byte un-stuffing: `[0x00, 0x32, 0x00, 0x02, 0xFF, 0xC6, 0x06]`
- Now the byte array may be interpreted as frame content and checksum:

Adr	CMD	State	Length	Data	CHK
`0x00`	`0x32`	`0x00`	`0x02`	`0xFF, 0xC6`	`0x06`

- remove checksum to obtain frame content

Adr	CMD	State	Length	Data
`0x00`	`0x32`	`0x00`	`0x02`	`0xFF, 0xC6`

- compute checksum of received frame content
  sum all bytes `0x00 + 0x32 + 0x00 + 0x02 + 0xFF + 0xC6 = 0x1F9`
  take LSB: `0xF9`
  invert: `0x06`
  checksum of received frame content matches received checksum. OK.
- check: address in the received frame is the same as in the sent frame. OK
- check command ID in the received frame is the same as in the sent frame. OK
- check: State is `0x00` (no error). OK
- data length is `0x02`, so Data has 2 bytes.
- Data = `[0xFF, 0xC6]`

The two bytes returned by the command `GetSingleMeasurement` need to be combined into one unsigned 16bit integer.

The first received byte is the Most Significant Byte (MSB), the second byte is the Least Significant Byte (LSB):

```
sensor_output = (0xFF << 8) + 0xC6 = 0xFFC6 = 65478
```

where '<<' indicates a bit shift operation to the left. Shifting by 8 bits to the left is equivalent to multiplying by 2**8 = 256.

If the measurement data type is signed, the unsigned integer value `sensor_output` needs to be converted (type cast) to a signed integer by the 2's complement convention:

```
if measurementdatatype == 0:                             # signed
     if sensor_output & 32768 == 32768:                  # 32768 = 2**15:
          flow_ticks = -((sensor_output ^ 65535) +1)     # 65535 = 2**16 -1
     else:
          flow_ticks = sensor_output
else:                                                    # unsigned
     flow_ticks = sensor_output
```

where the operator '**' denotes the power operator such as `2**3=8` and the operator '^' denotes the boolean 'exclusive or' (XOR).

So in the present example

```
flow_ticks = -((65478 ^ 65535) + 1) = -(57 + 1)=-58
```

The flow ticks can be converted to a physical flow rate (floating point operations are needed)

```
physical_flow = flow_ticks / scale_factor
```

here (assuming the scale factor is 13 and the flow unit of the sensor is ul/s, i.e. microliters per second):

```
-58 / 13 = -4.46
```

the flow rate measured by the sensor is -4.46 ul/s

GetMeasurementBuffer(receive several i16t or u16t)

We want to send the command 'GetMeasurementBuffer' to device 0, to read the measurement results during continuous measurement mode.

- Consult the RS485 Sensor Cable SHDLC Command Reference:

5.2.7 GET MEASUREMENT BUFFER

Get Measurement Buffer				
Description	Read out the newest 127 measurements and clear the buffer. Use the "Extended Buffer Command" to work with more than 127 buffered measurements. If the returned length is 0, no new measurements are available.			
Command ID	**0x36**		**for Sensor Type**	0, 2
Access Level	0		**Availability**	Always
Response Time max	1ms		**Storage**	Device Ram
MOSI Data (0 Bytes)	no data			
MISO Data (0…254 Bytes)	**Byte #**	**Description**		
	0, 1	*Measurementresult 0 : u16t/i16t*		
	2, 3	*Measurementresult 1 : u16t/i16t*		
	2*x, 2*x+1	*Measurementresult x : u16t/i16t*		

- Build frame content:

Adr	CMD	Length	Data
`0x00`	`0x36`	`0x00`	

- Compute the checksum over the frame content:
  sum all bytes in the frame content: `0x00 + 0x36 + 0x00 = 0x36`
  take the Least-Significant Byte (LSB): `0x36`
  invert: `0xC9`

- Add checksum to frame content

Adr	CMD	Length	Data	CHK
`0x00`	`0x36`	`0x00`		`0xC9`

- Convert to byte array: `[0x00, 0x36, 0x00, 0xC9]`

- Byte stuffing
  check: none of the special characters (`0x11`, `0x13`, `0x7D`, `0x7E`) appears in the byte array.
  Byte array after byte stuffing: `[0x00, 0x36, 0x00, 0xC9]`

- Add Start / Stop Bytes.

Byte array sent to Tx Buffer: `[0x7E, 0x00, 0x36, 0x00, 0xC9, 0x7E]`

Byte array received at Rx Buffer: `[0x7E, 0x00, 0x36, 0x00, 0x06, 0xFF, 0xC6, 0xFE, 0x7D, 0x5D, 0xFF, 0xA5, 0xDF, 0x7E]`

- remove start and stop bytes: `[0x00, 0x36, 0x00, 0x06, 0xFF, 0xC6, 0xFE, 0x7D, 0x5D, 0xFF, 0xA5, 0xDF]`

- Byte (un-)stuffing
  check for special characters marker (`0x7D`): Special character `0x7D` occurrs:
  `[0x00, 0x36, 0x00, 0x06, 0xFF, 0xC6, 0xFE,` **`0x7D, 0x5D,`** `0xFF, 0xA5, 0xDF]`
  replace according to Tab. 2: `0x7D, 0x5D` → `0x7D`
  byte array after byte un-stuffing: `[0x00, 0x36, 0x00, 0x06, 0xFF, 0xC6, 0xFE, 0x7D, 0xFF, 0xA5, 0xDF]`

- Now the byte array may be interpreted as frame content and checksum:

Adr	CMD	State	Length	Data	CHK
`0x00`	`0x36`	`0x00`	`0x06`	`0xFF, 0xC6, 0xFE, 0x7D, 0xFF, 0xA5`	`0xDF`

- remove checksum to obtain frame content

Adr	CMD	State	Length	Data
0x00	0x36	0x00	0x06	0xFF, 0xC6, 0xFE, 0x7D, 0xFF, 0xA5

- compute checksum of received frame content
  sum all bytes: `0x00 + 0x36 + 0x00 + 0x06 + 0xFF + 0xC6 + 0xFE + 0x7D + 0xFF + 0xA5 = 0x520`
  take LSB: `0x20`
  invert: `0xDF`
  checksum of received frame content matches received checksum. OK.
- check: address in the received frame is the same as in the sent frame. OK
- check command ID in the received frame is the same as in the sent frame. OK
- check: State is `0x00` (no error). OK
- data length is `0x06`, so Data has 6 bytes.
- Data = `[0xFF, 0xC6, 0xFE, 0x7D, 0xFF, 0xA5]`

Each pairs of bytes returned by the command `GetMeasurementBuffer` needs to be combined into one unsigned 16bit integer.

The first received byte in each pair is the Most Significant Byte (MSB), the second byte is the Least Significant Byte (LSB):

```
sensor_output_1 = (0xFF << 8) + 0xC6 = 0xFFC6 = 65478
sensor_output_2 = (0xFE << 8) + 0x7D = 0xFE7D = 65149
sensor_output_3 = (0xFF << 8) + 0xA5 = 0xFFA5 = 65445
```

where '<<' indicates a bit shift operation to the left. Shifting by 8 bits to the left is equivalent to multiplying by 2**8 = 256.

The 3 values of the sensor output correspond to the measbuffer:

measbuffer = [sensor_output_1, sensor_output2, sensor_output_3] = [65478, 65149, 65445]

If the measurement data type is signed, each unsigned integer value `sensor_output_x` needs to be converted (type cast) to a signed integer by the 2's complement convention:

```
if measurementdatatype == 0:                            # signed
    if sensor_output_x & 32768 == 32768:                # 32768 = 2**15:
        flow_ticks_x = -((sensor_output_x ^ 65535) +1)  # 65535 = 2**16 -1
    else:
        flow_ticks_x = sensor_output_x
else:                                                   # unsigned

    flow_ticks_x = sensor_output_x
```

where the operator '**' denotes the power operator such as `2**3=8` and the operator '^' denotes the boolean 'exclusive or' (XOR).

So in the present example

```
flow_ticks_1 = -((65478 ^ 65535) + 1) = -(57 + 1)=-58
flow_ticks_2 = -((65149 ^ 65535) + 1) = -(386 + 1)=-387
flow_ticks_3 = -((65445 ^ 65535) + 1) = -(90 + 1)=-91
```

The flow ticks can be converted to physical flow rate (floating point operations are needed)

```
physical_flow = flow_ticks / scale_factor
```

here (assuming the scale factor is 13 and the flow unit of the sensor is ul/s, i.e. microliters per second):

```
-58 / 13 = -4.46, -387 / 13 = -29.77, 91 / 13 = -7.00
```

the array of flow rates returned by the sensor is [-4.46 ul/s, - 29.77 ul/s, -7.00 ul/s]

GetTotalizatorValue (receive one i64t)

We want to send the command 'GetTotalizatorValue' to device 0, to read the value of the Totalizator.

- Consult the RS485 Sensor Cable SHDLC Command Reference:

5.2.9 TOTALIZATOR VALUE

Get Totalizator Value				
Description	Get the value of the Totalizator. This value is the sum of all unscaled measurements while in continuous measurement			
Command ID	0x38		**for Sensor Type**	0, 2
Access Level	0		**Availability**	Always
Response Time max	1ms		**Storage**	Device Ram
MOSI Data (0 Bytes)	no data			
MISO Data (8 Bytes)	**Byte #**	**Description**		
	0…7	*Totalisator: i64t*		

- Build frame content:

Adr	CMD	Length	Data
`0x00`	`0x38`	`0x00`	

- Compute the checksum over the frame content:
  sum all bytes in the frame content: `0x00 + 0x36 + 0x00 = 0x38`
  take the Least-Significant Byte (LSB): `0x38`
  invert: `0xC7`

- Add checksum to frame content

Adr	CMD	Length	Data	CHK
`0x00`	`0x38`	`0x00`		`0xC7`

- Convert to byte array: `[0x00, 0x38, 0x00, 0xC7]`

- Byte stuffing
  check: none of the special characters (`0x11`, `0x13`, `0x7D`, `0x7E`) appears in the byte array.
  Byte array after byte stuffing: `[0x00, 0x38, 0x00, 0xC7]`

- Add Start / Stop Bytes.

Byte array ready to send to Tx Buffer: `[0x7E, 0x00, 0x38, 0x00, 0xC7, 0x7E]`

Byte array received at Rx Buffer: `[0x7E, 0x00, 0x38, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4, 0x86, 0x7E]`

- remove start and stop bytes: `[0x00, 0x38, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4, 0x86]`

- Byte (un-)stuffing
  check for special characters marker (`0x7D`): No special characters marker.
  byte array after byte un-stuffing: `[0x00, 0x38, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4, 0x86]`
- Now the byte array may be interpreted as frame content and checksum:

Adr	CMD	State	Length	Data	CHK
`0x00`	`0x38`	`0x00`	`0x08`	`0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4`	`0x86`

- remove checksum to obtain frame content

Adr	CMD	State	Length	Data
`0x00`	`0x38`	`0x00`	`0x08`	`0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4`

- compute checksum of received frame content
  sum all bytes `0x00 + 0x38 + 0x00 + 0x08 + 0x00 + ... + 0xB4 = 0x179`
  take LSB: `0x79`
  invert: `0x86`
  checksum of received frame content matches received checksum. OK.
- check: address in the received frame is the same as in the sent frame. OK
- check command ID in the received frame is the same as in the sent frame. OK
- check: State is `0x00` (no error). OK
- data length is `0x08`, so Data has 8 bytes.
- Data = `[0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x83, 0xB4]`

The 8 bytes returned by the command `GetTotalizatorValue` first need to be combined into one unsigned 64bit integer.

The first received byte is the Most Significant Byte (MSB), the last byte is the Least Significant Byte (LSB):

```
tot_output = (0x00 << 56) + (0x00 << 48) + ... + 0xB4 = 0x0283B4 = 164788
```

where '<<' indicates a bit shift operation to the left. Shifting by 8 bits to the left is equivalent to multiplying by 2**8 = 256.

This unsigned 64 bit value needs to be converted to a signed integer by the 2's complement convention:

```
if tot_output & 2**63 == 2**63:
    tot_ticks = -((tot_output ^ (2**64-1)) +1)
else:
    tot_ticks = tot_output
```

where the operator '**' denotes the power operator such as `2**3=8` and the operator '^' denotes the boolean 'exclusive or' (XOR).

So in the present example

```
tot_output & 2**63 == 2**63: False
tot_ticks = tot_output
```

The flow ticks can be converted to physical flow rate (floating point operations are needed)

```
physical_volume = tot_ticks / scalefactor * sampling_time
```

here (assuming the scale factor is 13 and the flow unit of the sensor is ul/s, i.e. microliters per second and a sampling time of 20 ms):

```
164778 / 13 * 0.020 = 253.52
```

the integrated volume is 253.52 ul

Headquarter and Sales Offices

SENSIRION AG
Laubisruetistr. 50
CH-8712 Staefa ZH
Switzerland
Phone: + 41 (0)44 306 40 00
Fax: + 41 (0)44 306 40 30
info@sensirion.com
www.sensirion.com

SENSIRION Inc
Westlake Pl. Ctr. I, suite 204
2801 Townsgate Road
Westlake Village, CA 91361
USA
Phone: +1 805-409 4900
Fax: +1 805-435 0467
michael.karst@sensirion.com
www.sensirion.com

SENSIRION Japan
Sensirion Japan Co. Ltd.
Shinagawa Station Bldg. 7F
4-23-5 Takanawa
Minato-ku, Tokyo, Japan
Phone: +81 3-3444-4940
Fax: +81 3-3444-4939
info@sensirion.co.jp
www.sensirion.co.jp

SENSIRION Korea Co. Ltd.
#1414, Anyang Construction Tower B/D,
1112-1, Bisan-dong, Anyang-city,
Gyeonggi-Province, South Korea
Phone: +82-31-440-9925~27
Fax: +82-31-440-9927
info@sensirion.co.kr
www.sensirion.co.kr

SENSIRION China Co. Ltd.
Room 2411, Main Tower
Jin Zhong Huan Business Building,
Postal Code 518048
Futian District, Shenzhen, PR China
Phone: +86 755 8252 1501
Fax: +86 755 8252 1580
info@sensirion.com.cn/
www.sensirion.com.cn

Find your local representative at: www.sensirion.com