ELVEFLOW an ELVESYS brand

Elveflow User Guide

# SDK SOFTWARE DEVELOPMENT KIT

DOCUMENT REF: UGSDK-Cpp-230517


Support: customer@elveflow.com

# SYMBOLS USED IN THIS DOCUMENT

**IMPORTANT INFORMATION.** Disregarding this information may increase the risk of: damage to the equipment, personal injuries, and impact your user experience.

**HELPFUL INFORMATION.** This information facilitates the use of the instrument and contributes to its optimal performance.

**ADDITIONAL INFORMATION.** Available on Elveflow website or from your Elveflow representative.

# READ THIS MANUAL CAREFULLY BEFORE USING THE SOFTWARE

**This manual must be read by every person who is or will be responsible for using the Elveflow software development kit (SDK).**

Due to the continual development of the products, the content of this manual may not correspond to the new software. Therefore, we retain the right to make adjustments without prior notification.

## Important SDK safety notices:

1. The SDK gives the user complete control over Elveflow products. Beware of pressure limits for containers, chips and other parts of your setup. They might be damaged if the pressure applied is too high.

2. Use a computer with enough power to avoid software freezing.

**If these conditions are not RESPECTED, the user is exposed to dangerous situations** and the instrument can undergo permanent damage. Elvesys and its partners cannot be held responsible for any damage related to the misuse of the instruments.

# Table of contents

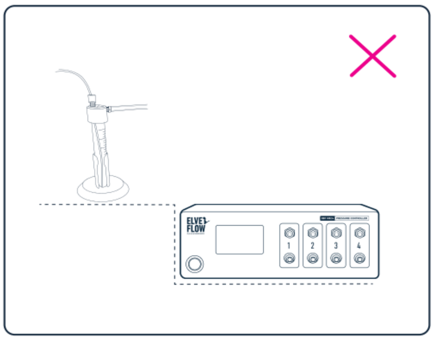
# Getting started

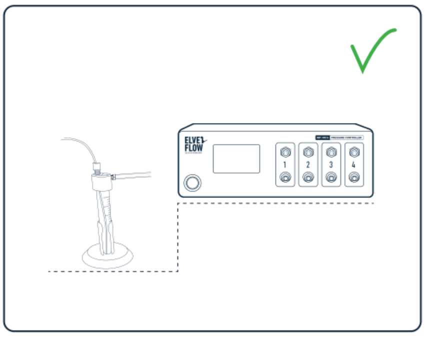
Elveflow proposes a standard development kit for LabVIEW, C++, Python and MATLAB

The following sections will guide you through the steps to add a new instrument or sensor, explore its basic and advanced features and use it with other instruments to automate your experiment.

## Before starting

To prevent backflow in the pressure regulator, always place liquid reservoirs under the instrument.





## Important remarks

**For all programming languages:**

- **If MUX Distribution/Distributor/Recirculation/Injection, BFS or OB1 MK4 are used,** FTDI drivers are required (http://www.ftdichip.com/Drivers/D2XX.htm). You can find these drivers in the same folder the ESI is installed. Default location would be C:\Program Files (x86)\Elvesys\driver (look for driver_MUX_distAndBFS.exe).
- **Do not simultaneously use the ESI software and the SDK**, some conflict would occur.
- **For X64 libraries running on an AMD computer,** the impossibility of communicating with instruments has been occasionally reported. Contact us if it is the case. The only fix found so far is to set the environment variable **MKL_DEBUG_CPU_TYPE** to **4.**

# DLL programming:

## Introduction

For C++, MATLAB, and Python programming languages, two C++ DLL libraries common to all languages are available. One for x64 and one for x32 operating systems (DLL32 and DLL64 folders). These libraries (Elveflow32.dll and Elveflow64.dll) contain all the needed functions for your custom software development and integration of Elveflow instruments.

Since the source library is the same for each programming language (C++, MATLAB, and Python), the SDK functions are the same for each language and will be described only once in this guide. Please see the appropriate section for a complete description of all the available functions.

Due to their difference in operation, a description of the essential differences between each SDK's language will be described in a dedicated document. They will allow you to quickly grasp the specifics of each language and to start developing your custom software.

- Instruments are designated using their device name. The device name can be known and changed using National Instruments Measurement and Automation Explorer (NI MAX). The NI MAX Software should be automatically installed with Elveflow Smart Interface.
- The function "Check_Error" or "CheckError" is common for all the instruments. It is used to check errors from all functions, it uses LabVIEW errors that could be checked on the internet.
- An example function that could be used for feedback control is included in all libraries as an illustration only (see the specific prototype). It is provided as an example to help you create your own regulation system. Alternatively the remote mode, available for OB1, MSRD and BFS devices from V3_05_04, enables the library to handle the regulation, for the same device or between different ones. See the OB1 example file.

# Description of SDK functions for each instrument:

## OB1:

### Normal workflow

The example "OB1_Ex__" illustrates the working principle of all the available SDK functions for the OB1.

The structure of the main program you would develop including Elveflow instruments should follow the same workflow as represented in the following figure. Using this workflow, you will start with a **configuration** and a **calibration** before starting to operate the OB1 and its connected sensors. Then, you can perform your instrumentation using the functions represented in the "**main working loop**".

After the end of the operations, you **end the communication** by closing the communication with the OB1, clear the pointers and unload the DLL.

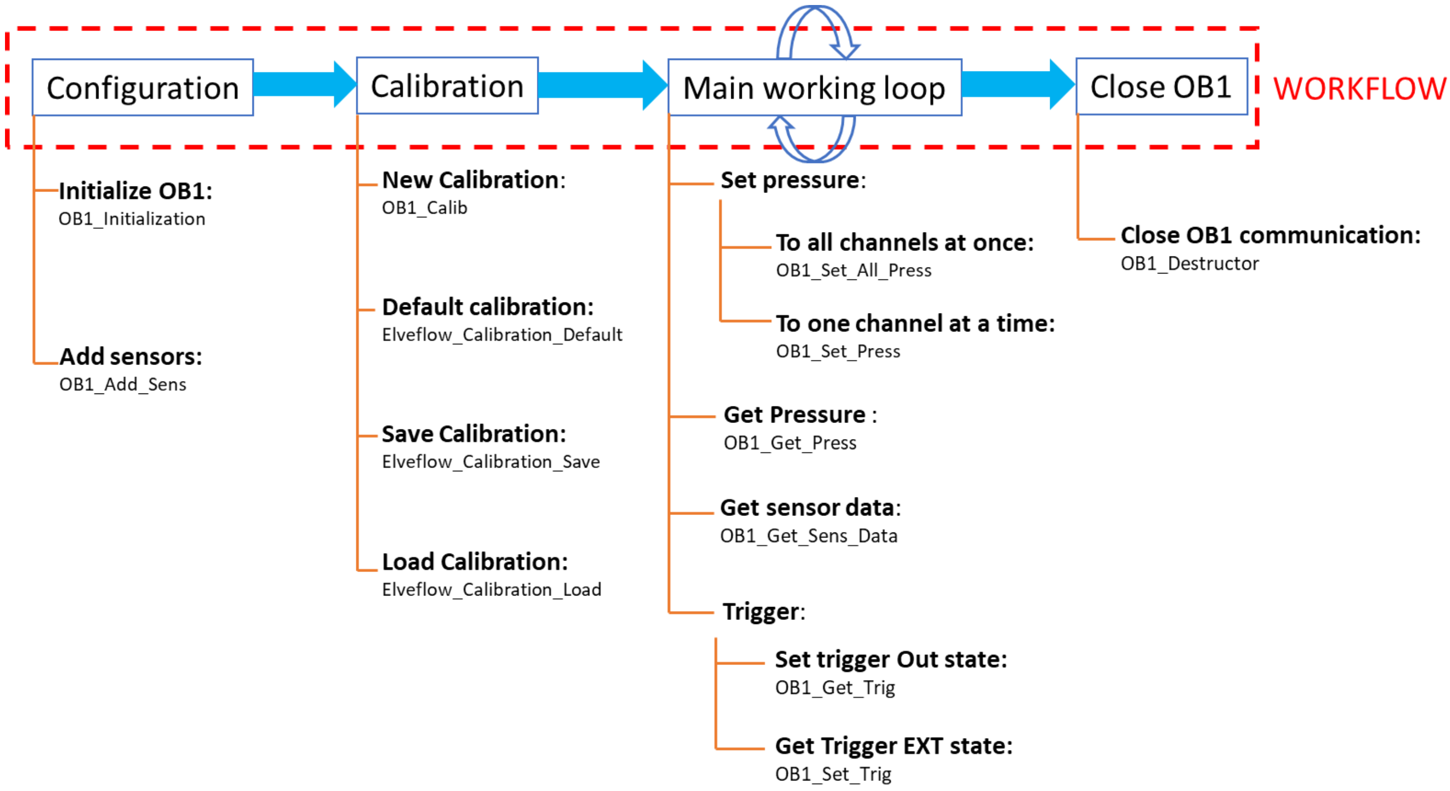



**Figure 3** Typical workflow of a custom OB1 program representing the different types of the OB1 SDK functions

## Remote workflow

The main working loop can be simplified with a new set of functions to launch a remote control and monitoring loop. Start the loop calling OB1_Start_Remote_Measuing.vi and stop the loop calling OB1_Stop_Remote_Measuing.vi. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

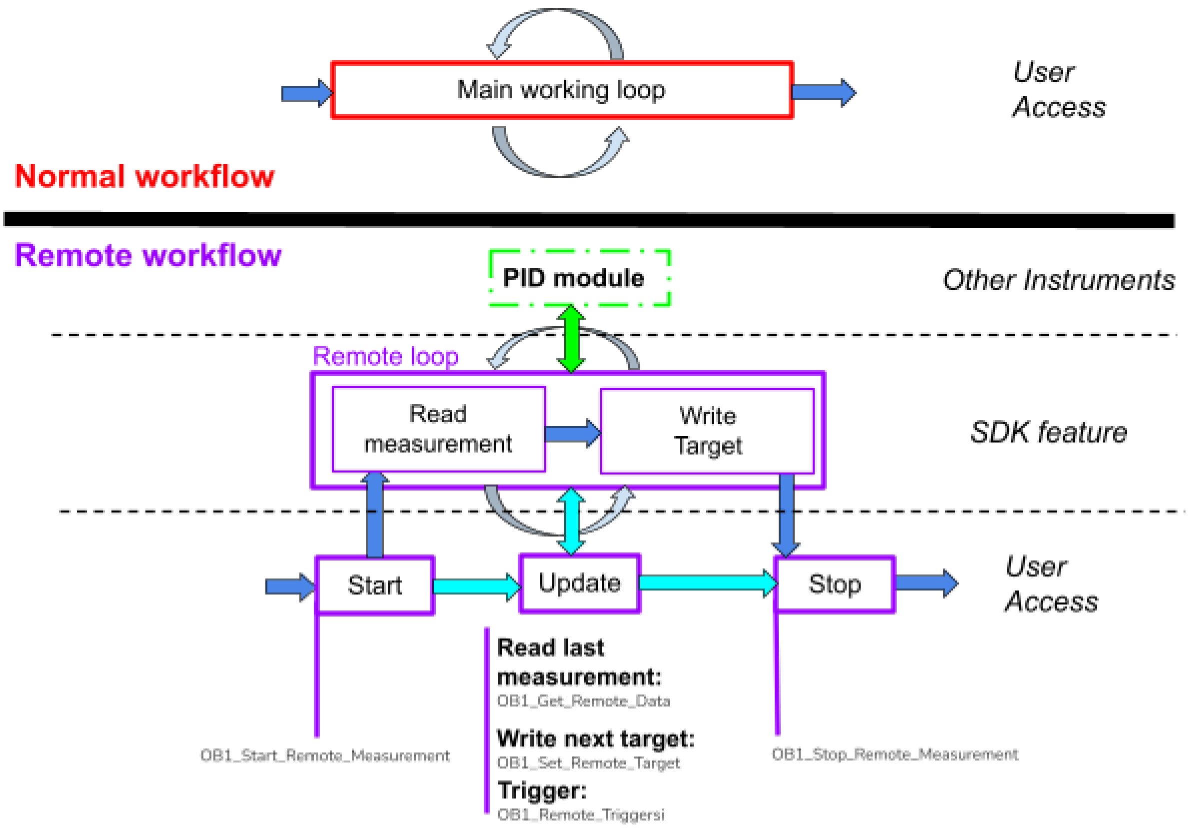



**Figure 4** Differences of workflow with the remote functions for the OB1 devices

A description of each function can be found in the table below or in the form of script comments in the functions. To help debug the code, all functions will return an error code.

## OB1_Initialization

**Arguments :**

- char : Device_Name[]
- Z_regulator_type: reg_ch_1
- Z_regulator_type: reg_ch_2
- Z_regulator_type: reg_ch_3
- Z_regulator_type: reg_ch_4
- int32_t: OB1_ID_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Initialize the OB1 device using device name and regulator type (see SDK Z_regulator_type for corresponding numbers). It modifies the OB1 ID (number >=0). This ID can be used with other functions to identify the targeted OB1. If an error occurs during the initialization process, the OB1 ID value will be -1.

**Note for OB1 MK4 devices:**

- the Device_Name is the VISA resource name, in the form of “COMX”
- The regulator type should be left at 0

## OB1_Destructor

**Arguments :**

- int32_t : OB1_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Close communication with OB1

## OB1_Add_Sens

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- Z_sensor_type : sensor_type
- Z_Sensor_digit_analog : digital_or_analog
- Z_Sensor_FSD_Calib : fsens_digit_calib
- Z_D_F_S_Resolution : fsens_digit_resolution
- double : customsens_voltage_5_to_25

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Add sensor to OB1 device. Select the channel n° (1-4) the sensor type. For the Flow sensor, the type of communication (Analog/Digital), the Calibration for digital version (H20 or IPA) should be specified as well as digital resolution (9 to 16 bits). (see SDK user guide, Z_sensor_type_type , Z_sensor_digit_analog, Z_Sensor_FSD_Calib and Z_D_F_S_Resolution for number correspondence).
For digital sensors, the sensor type is automatically detected during this function call.
For analog sensors, the calibration parameters are not taken into account.

If the sensor is not compatible with the OB1 version, or no digital sensor is detected an error will be thrown as output of the function.

---

## OB1_Get_Sens_Data

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- int32_t : acquire_data1true0false
- double : *sens_data

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Read the sensor of the requested channel. ! This Function only converts data acquired in OB1_Acquire_data Units : Flow sensor µl/min Pressure : mbar

If Acquire_data is true, the OB1 acquires ALL regulator AND ALL analog sensor value. They are stored in the computer memory. Therefore, if several regulator values (OB1_Get_Press) and/or sensor values (OB1_Get_Sens_Data) have to be acquired simultaneously, set the Acquire_Data to true only for the First function.

All the others can use the values stored in memory and are almost instantaneous. Digital sensors unfortunatly require another communication protocol, thus this parameter has no impact NB: For Digital Flow Sensor, If the connection is lost, OB1 will be reset and the return value will be zero.

---

## OB1_Get_Press

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- int32_t : acquire_data1true0false
- double : calib_array_in[]
- double : *pressure
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the pressure of an OB1 channel. Calibration array is required (use Set_Default_Calib if required) and return a double . calib_array_length corresponds to the Calib_array_in length.

If Acquire_data is true, the OB1 acquires ALL regulators AND ALL analog sensor value. They are stored in the computer memory. Therefore, if several regulator values (OB1_Get_Press) and/or sensor values (OB1_Get_Sens_Data) have to be acquired simultaneously, set the Acquire_Data to true only for the first function call. All the others can use the values stored in memory and are almost instantaneous.

---

## OB1_Set_Press

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- double : pressure
- double : calib_array_in[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf [Error handling](#) )

**Description :**

Set the pressure of the OB1 selected channel, Calibration array is required (use Set_Default_Calib if required). Len corresponds to the Calib_array_in length.

---

## OB1_Set_All_Press

**Arguments :**

- int32_t : OB1_ID
- double : pressure_array_in[]
- int32_t : pressure_array_length
- double : calib_array_in[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf [Error handling](#) )

**Description :**

Set the pressure of all the channels of the selected OB1. Calibration array is required (use Set_Default_Calib if required). Calib_Array_Len corresponds to the Calib_array_in length. It uses an array as a pressure input. Pressure_Array_Len corresponds to the pressure input array. The first number of the array corresponds to the first channel, the seconds number to the seconds channels and so on. All the numbers above 4 are not taken into account. If only One channel needs to be set, use OB1_Set_Pressure.

---

## OB1_Get_Trig

**Arguments :**

- int32_t : OB1_ID
- int32_t : *trigger

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the trigger of the OB1 (0 = 0V, 1 =3,3V for MK3, 5V for MK4)

## OB1_Set_Trig

**Arguments :**

- int32_t : OB1_ID
- int32_t : trigger

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the trigger of the OB1 (0 = 0V, 1 =3,3V, 5V for MK4)

## OB1_Calib

**Arguments :**

- int32_t : OB1_ID
- double : calib_array_out[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Launch OB1 calibration and return the calibration array. Before Calibration, ensure that ALL channels are properly closed with adequate caps.

## OB1_Reset_Digit_Sens (OB1 MK3+)

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Reset the digital sensor

## OB1_Reset_Instr (OB1 MK3+)

**Arguments :**

- int32_t : OB1_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Reset the instrument

## Elveflow_Calibration_Default

**Arguments :**

- double : calib_array_out[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set default Calib in Calib array

## Elveflow_Calibration_Load

**Arguments :**

- char : path[]
- double : calib_array_out[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Load the calibration file located at Path and return the calibration parameters in the Calib_Array_out.. The function asks the user to choose the path if Path is not valid, empty or not a path. The function indicates if the file was found.

## Elveflow_Calibration_Save

**Arguments :**

- char : path[]
- double : calib_array_in[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Save the Calibration cluster in the file located at Path. len is the Calib_Array_in array length. The function prompts the user to choose the path if Path is not valid, empty or not a path.

---

## Elveflow_EXAMPLE_PID

**Arguments :**

- int32_t : PID_ID_in
- double : actualValue
- int32_t : Reset
- double : p
- double : i
- int32_t : *PID_ID_out
- double : *value

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

This function is only provided for illustration purposes, to explain how to do your own feedback loop. Elveflow does not guarantee neither efficient nor optimum regulation with this illustration of PI regulator.

With this function the PI parameters have to be tuned for every regulator and every microfluidic circuit. This function needs to be initiated with a first call where PID_ID =-1. The PID_out will provide the newly created PID_ID. This ID should be used in further calls. General remarks of this PI regulator : The error "e" is calculated for every step as e=target value-actual value There are 2 contributions to a PI regulator: proportional contribution which only depend on this step and Prop=eP and integral part which is the "memory" of the regulator. This value is calculated as Integ=integral(Iedt) and can be reset.

---

## OB1_Start_Remote_Measurement

**Arguments :**

- int32_t : OB1_ID
- double : calib_array_in[]
- int32_t : calib_array_length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Start a loop running in the background, and automatically reads all sensors and regulators. No direct call to the OB1 can be made until the Stop measuring function is called. Until then only functions accessing this loop (get_remote_data, set_remote_target, remote_triggers) are recommended.

---

## OB1_Set_Remote_Target

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- double : target

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the Target of the OB1 selected channel. Modify the pressure if the PID is off, or the sensor is a pressure sensor. Modify a flow if the sensor is  a flow sensor and the PID is on.

---

## OB1_Remote_Triggers

**Arguments :**

- int32_t : OB1_ID
- int32_t : trigger_in
- int32_t : *trigger_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the Trigger input and get the Trigger output of the OB1 device.

---

## OB1_Get_Remote_Data

**Arguments :**

- int32_t : OB1_ID
- int32_t : channel_1_to_4
- double : *reg_data
- double : *sens_data

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Read the sensor and regulator values of the requested channel. Warning: This Function only extracts data obtained in the remote  measurement loop Sensor unit : mbar if pressure sensor, µl/min if flow sensor Regulator unit : mbar

**Note** : For Digital Flow Sensor, If the connection is lost, OB1 will be reset and the return value will be zero

---

## OB1_Stop_Remote_Measurement

**Arguments :**

- int32_t : OB1_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Stop the background measure & control loop

# MSRD:

All the available functions for the programming of a customized program are detailed in the example “M_S_R_D_Ex__” contained in the appropriate example folder.

## M_S_R_D_Initialization

**Arguments :**

- char: Device_Name[]
- Z_sensor_type: Sens_Ch_1
- Z_sensor_type: Sens_Ch_2
- Z_sensor_type: Sens_Ch_3
- Z_sensor_type: Sens_Ch_4
- double : CustomSens_Voltage_Ch12
- double : CustomSens_Voltage_Ch34
- int32_t : *MSRD_ID_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Initialize the Sensor Reader device able to read digital sensors (MSRD) using device name and sensor type (see SDK Z_sensor_type for corresponding numbers). It modifies the MSRD ID (number >=0). This ID can be used with other functions to identify the targeted MSRD. If an error occurs during the initialization process, the MSRD ID value will be -1. Initiate the communication with the Sensor Reader able to read digital sensors (MSRD). This VI generates an identification cluster of the instrument to be used with other VIs. NB: Sensor type has to be written here in addition to the "Add_Sens". NB 2: Sensors connected to channel 1-2 and 3-4 have to be the same type otherwise they will not be taken into account.

## M_S_R_D_Destructor

**Arguments :**

- int32_t : M_S_R_D_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Close communication with MSRD.

## M_S_R_D_Add_Sens

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : channel_1_to_4
- Z_sensor_type: sensor_type
- Z_Sensor_digit_analog: digital_or_analog
- Z_Sensor_FSD_Calib: fsens_digit_calib
- Z_D_F_S_Resolution: fsens_digit_resolution

**Returns :**

- int32_t : error constant (cf [Error handling](#) )

**Description :**

Add sensor to MSRD device. Select the channel n° (1-4) the sensor type.  For the Flow sensor, the type of communication (Analog/Digital), the  Calibration for digital version (H20 or IPA) should be specified as well as  digital resolution (9 to 16 bits). (see SDK user guide, Z_sensor_type_type , Z_sensor_digit_analog, Z_Sensor_FSD_Calib and Z_D_F_S_Resolution for the table of correspondence)  For digital versions, the sensor type is automatically detected during this  function call. For the Analog sensor, the calibration parameters are not taken into account.  If the sensor is not compatible with the MSRD version, or no digital sensor is detected, an error will be thrown as output of the function.  NB: Sensor type has to be the same as in the "Initialization" step.

---

## M_S_R_D_Get_Sens_Data

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : channel_1_to_4
- double: sens_data

**Returns :**

- int32_t : error constant (cf [Error handling](#) )

**Description :**

Read the sensor of the requested channel.s Units: Flow sensor: µl/min Pressure: mbar  NB: For Digital Flow Sensor, If the connection is lost, MSRD will be reseted  and the return value will be zero.

---

## M_S_R_D_Get_Trig (MSR V3)

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : *trigger

**Returns :**

- int32_t : error constant (cf [Error handling](#) )

**Description :**

Get the trigger of the M_S_R_D_ID (0 = 0V, 1 =5V).

---

## M_S_R_D_Set_Trig (MSR V3)

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : trigger

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the trigger of the M_S_R_D_ID (0 = 0V, 1 =5V).

## M_S_R_D_Set_Filt

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : channel_1_to_4
- LVBoolean : ONOFF

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set filter for the corresponding channel.

## M_S_R_D_Reset_Instr (MSR V2)

**Arguments :**

- int32_t : M_S_R_D_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Reset the MSRD device.

## M_S_R_D_Reset_Sens (MSR V2)

**Arguments :**

- int32_t : M_S_R_D_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Reset the digital sensors connected to the MSRD.

## M_S_R_D_Start_Remote_Measurement

**Arguments :**

- int32_t : M_S_R_D_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Start the monitoring loop for the MSRD device.

---

## M_S_R_D_Stop_Remote_Measurement

**Arguments :**

- int32_t : M_S_R_D_ID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Stop the monitoring loop for the MSRD device. Should only be called at the end of the program

---

## M_S_R_D_Get_Remote_Data

**Arguments :**

- int32_t : M_S_R_D_ID
- int32_t : channel_1_to_4
- double : sens_data

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Read the sensor of the requested channel.s Units: Flow sensor: µl/min Pressure: mbar NB: For Digital Flow Sensor, If the connection is lost, MSRD will be reseted and the return value will be zero

---

## BFS:

Please see the example file "_BFS_Example.cpp" for a standard usage of the available BFS functions. As with other instruments, there are three steps for programming: Initialization, instrumentation and resource liberation. Please note that for this particular sensor, in order to measure the flow rate (µL/min), you must first measure the volumetric mass density (g/L). Please see the table below for a description of each function.

The mainworking loop can be simplified with a new set of functions to launch a remote monitoring loop. Start the loop calling BFS_Start_Remote_Measuing and stop the loop calling BFS_Stop_Remote_Measuing. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

---

## BFS_Initialization

**Arguments :**

- char: Visa_COM[]
- int32_t : *BFS_ID_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Initiate the BFS device using device com port (ASRLXXX::INSTR where XXX is the com port that could be found in windows device manager). It returns the BFS ID (number >=0) to be used with other functions.

---

## BFS_Destructor

**Arguments :**

- int32_t : BFS_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Close Communication with BFS device

---

## BFS_Get_Flow

**Arguments :**

- int32_t : BFS_ID_in
- double: *flow

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Measure the fluid flow in (microL/min). !!! This function required an earlier density measurement!!! The density can either be measured only once at the beginning of the experiment (ensure that the fluid flows through the sensor prior to density measurement), or before every flow measurement if the density might change. If you get +inf or -inf, the density wasn't correctly measured.

---

## BFS_Zeroing

**Arguments :**

- int32_t : BFS_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Perform zero calibration of the BFS. Ensure that there is no flow when performed; it is advised to use valves. The calibration procedure is finished when the green LED stops blinking.

---

## BFS_Get_Mass_Flow

**Arguments :**

- int32_t : BFS_ID_in
- double: *massflow

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Measure the fluid flow in (microL/min). !!! This function required an earlier density measurement!!! The density can either be

---

## BFS_Get_Density

**Arguments :**

- int32_t : BFS_ID_in
- double: *density

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get fluid density (in g/L) for the BFS defined by the BFS_ID

---

## BFS_Get_Temperature

**Arguments :**

- int32_t : BFS_ID_in
- double: *temperature

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the fluid temperature (in °C) of the BFS defined by the BFS_ID

---

## BFS_Set_Filter

**Arguments :**

- int32_t : BFS_ID_in
- double: *filter_value

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Elveflow Library BFS Device Set the instrument Filter. 0.000001= maximum filter -> slow change but very low noise. 1= no filter-> fast change but noisy. Default value is 0.1

---

## BFS_Start_Remote_Measurement

**Arguments :**

- int32_t : BFS_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Start the monitoring loop for the BFS device.

---

## BFS_Set_Remote_Params

**Arguments :**

- int32_t : BFS_ID_in
- double : filter
- int32_t : M_temp
- int32_t : M_density

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Modify the parameters of the remote monitoring loop: M_density: a new measure of the density will be taken before each flow measurement M_temp: a new temperature measurement will be taken after each flow measurement Filter: change the filter used to measure the flow

## BFS_Get_Remote_Data

**Arguments :**

- int32_t : BFS_ID_in
- double : temperature
- double : density
- double : flow

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Read the sensors from the remote monitoring loop: Units: Flow sensor: µl/min Density: g/m3 Temperature: Celcius

## BFS_Stop_Remote_Measurement

**Arguments :**

- int32_t : BFS_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Stop the monitoring loop for the BFS device.

# MUX D-R-I (DISTRIBUTION, DISTRIBUTOR, RECIRCULATION or INJECTION):

Please see the example file “MUX_DRI_Ex__” for a standard usage of the available MUX DRI functions. The following table gives a description of the MUX DRI functions.

## MUX_DRI_Initialization

**Arguments :**

- char : Visa_COM[]
- int32_t : *MUX_DRI_ID_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Initiate the MUX Distribution, Distributor, Recirculation or Injection device using device COM port (ASRLXXX::INSTR where XXX is usually the COM port that could be found in Windows device manager). It returns the MUX D-R-I ID (number >=0) to be used with other functions.

## MUX_DRI_Destructor

**Arguments :**

- int32_t : MUX_DRI_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Close Communication with MUX Distribution, Distributor, Recirculation or Injection device.

## MUX_DRI_Set_Valve

**Arguments :**

- int32_t : MUX_DRI_ID_in
- int32_t : selected_Valve
- Z_MUX_DRI_Rotation: Rotation

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Switch the MUX Distribution, Distributor, Recirculation or Injection to the desired valve. For MUX Distribution 12, between 1-12. For MUX Distributor (6 or 10 valves), between 1-6 or 1-10. For MUX Recirculation 6 or MUX Injection (6 valves), the two states are 1 or 2. Rotation indicates the path the valve will perform to select a valve, either shortest 0, clockwise 1 or counterclockwise 2.

## MUX_DRI_Get_Valve

**Arguments :**

- int32_t : MUX_DRI_ID_in
- int32_t : *selected_Valve

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the current valve number. If the valve is changing, the function returns 0.

## MUX_DRI_Send_Command (MUX Distribution 12 & MUX Recirculation 6)

**Arguments :**

- int32_t : MUX_DRI_ID_in
- Z_MUX_DRI_Action : action
- char: answer[]
- int32_t: length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the Serial Number or Home the valve. len is the length of the Answer. Remember that Home the valve takes several seconds. Home the valve is necessary as an initialization step before using the valve for a session.

## Other MUX Series:

The MUX Series encompasses three instruments: MUX CROSS CHIP, MUX FLOW SWITCH and MUX WIRE. They are grouped together here because they use the same functions to start and end the communication. The table below gives the description of each function. The example “MUX_Ex__” illustrates the usage of all these functions.

---

### MUX_Initialization

**Arguments :**

- char : Device_Name[]
- int32_t : *MUX_ID_out

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Initiate the MUX device using device name (could be obtained in NI MAX). It return the F_S_R ID (number >=0) to be used with other functions.

---

### MUX_Destructor

**Arguments :**

- int32_t : MUX_ID_in

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Close the communication of the MUX device.

---

### MUX_Get_Trig

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : *Trigger

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Get the trigger of the MUX device (0=0V, 1=5V).

---

## MUX_Set_Trig

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : Trigger

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the Trigger of the MUX device (0=0V, 1=5V).

---

## MUX_Set_all_valves (MUX FLOW SWITCH)

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : array_valve_in[]
- int32_t : length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Valves are set by an array of 16 elements. If the valve value is equal to or below 0, the valve is closed, if it's equal to or above 1 the valve is open. The index in the array indicates the selected valve as shown below:

0 1 2 3

4 5 6 7

8 9 10 11

12 13 14 15

If the array does not contain exactly 16 elements nothing happens.

---

## MUX_Set_indiv_valve(MUX CROSS CHIP)

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : Input
- int32_t : Output
- int32_t : OpenClose

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the state of one valve of the instrument. The desired valve is addressed using Input and Output parameters which correspond to the fluidics inputs and outputs of the instrument.

## MUX_Wire_Set_all_valves (MUX WIRE instrument)

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : array_valve_in[]
- int32_t : length

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Valves are set by an array of 16 elements. If the valve value is equal to or below 0, the valve is closed, if it's equal to or above 1 the valve is open. If the array does not contain exactly 16 elements nothing happens.

## MUX_Set_valves_Type (MUX WIRE V3 instrument)

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : valveNb
- int32_t : type

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Set the valve type. This function is available for MUX Wire V3 using custom Valves or Valve V2. Valve V3 type are automatically recognized by the MUX.

Custom types availables:

0 : Undefined (or delete custom valve)
4 : 2/2 Normally Closed Custom
5 : 2/2 Normally Opened Custom
6 : 3/2 Universal Custom

## MUX_Get_valves_Type (MUX WIRE V3 instrument)

**Arguments :**

- int32_t : MUX_ID_in
- int32_t : types_array[]
- int32_t : length

**Returns :**

- int32_t : error constant (cf Error handling )

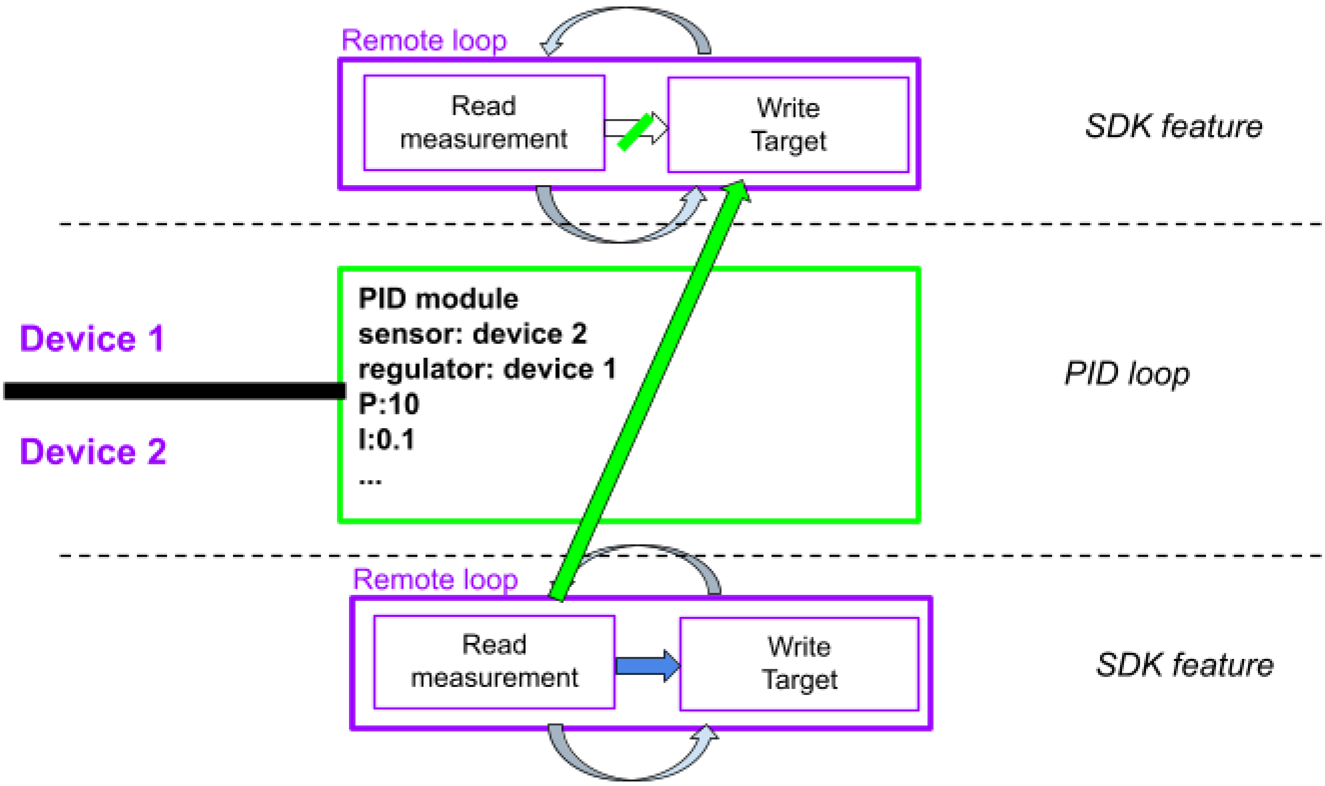
**Description :**

Read the Types values for all your MUX Wire V3 ports :

UNDEFINED = 0
2/2 Normally Closed = 1
2/2 Normally Opened = 2
3/2 Universal = 3
2/2 Normally Closed Custom = 4
2/2 Normally Opened Custom = 5
3/2 Universal Custom = 6

## Remote PID:

If you configure any of the compatible instruments (OB1, MSRD, BFS) in the remote mode, where the acquisition is run autonomously, you will also be able to configure, start and stop PID loops between these instruments without having to write the code itself of the loop. A fully working flow regulation can be started with an OB1 and an MFS with a single function call. Subsequent modification of the PID target is achieved in the remote loop of the device controlling the pressure/flow. A PID loop can be started on a single remote loop if the device can regulate the pressure/flow and a sensor is also connected to it.




In this figure, we consider the example of two devices both capable of pressure control and sensor reading. These two devices are configured in remote mode and therefore measure the sensor, then update the pressure automatically. By calling add_PID between device 1 and device 2, the measurement from the sensor of device 2 is broadcasted to the remote loop of device 1. A PID loop is then continuously running across the two remote loops and can be stopped, modified or reset on demand.

PID loops are destroyed when the device containing the regulator of the loop is closed.

**PI or PID?** The current SDK only allows PI parameters at the moment. Users who need to use the derivative term can use their own PID loop instead.

## PID_Add_Remote

**Arguments :**

- int32_t : Regulator ID
- int32_t : Regulator Channel
- int32_t : Sensor ID
- int32_t : Sensor Channel
- double : Proportional parameter
- double  : Integral parameter
- int32_t : Run (1) or stop (0) the PID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Add a PID loop between a regulator and a sensor. The PID loop can later be called with the device hosting the regulator coupled with its channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved

---

## PID_Set_Running_Remote

**Arguments :**

- int32_t : Regulator ID
- int32_t : Regulator Channel
- int32_t : Run (1) or stop (0) the PID

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Adjust the running status of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved.

---

## PID_Set_Params_Remote

**Arguments :**

- int32_t : Regulator ID
- int32_t : Regulator Channel
- int32_t : (1) to reset the accumulated error value
- double : Proportional parameter
- double  : Integral parameter

**Returns :**

- int32_t : error constant (cf Error handling )

**Description :**

Adjust the PID parameters of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved.

---

# Appendix

## Error handling:

All functions return an error code. If this code is 0 no error occurs. Other values indicate that an error occurs. Some personalized errors were added.

| Error code: | Signification: |
|---|---|
| -8000 | No Digital Sensor found |
| -8001 | No pressure sensor compatible with OB1 MK3 |
| -8002 | No Digital pressure sensor compatible with OB1 MK3+ |
| -8003 | No Digital Flow sensor compatible with OB1 MK3 |
| -8004 | No IPA config for this sensor |
| -8005 | Sensor not compatible |
| -8006 | No Instrument with selected ID |

Other errors can be found in the LabVIEW error user guide. (http://www.ni.com/pdf/manuals/321551a.pdf)

## List of constants, prototypes and description (for C++, MATLAB and Python):

All instruments have initialization and destructor function and several other functions described below.

All functions will return an error code that could help to debug your software.

Z_sensor_type_Level stands for all types of level sensor such as bubble detectors.

**Constants** (define Elveflow.h as uint16_t):

| Z_regulator_type: | |
|---|---|
| Z_regulator_type_none | 0 |
| Z_regulator _type__0_200_mbar | 1 |
| Z_regulator _type__0_2000_mbar | 2 |
| Z_regulator _type__0_8000_mbar | 3 |
| Z_regulator_type_m1000_1000_mbar | 4 |
| Z_regulator_type_m1000_6000_mbar | 5 |

*(Note : set to 0 for OB1 MK4 devices)*

| Z_sensor_type: | |
|---|---|
| Z_sensor_type_none | 0 |
| Z_sensor _type_Flow_1_5_muL_min | 1 |
| Z_sensor _type_Flow_7_muL_min | 2 |
| Z_sensor _type_Flow_50_muL_min | 3 |
| Z_sensor _type_Flow_80_muL_min | 4 |
| Z_sensor _type_Flow_1000_muL_min | 5 |
| Z_sensor _type_Flow_5000_muL_min | 6 |
| Z_sensor_type_Press_70_mbar | 7 |
| Z_sensor _type_Press_340_mbar | 8 |
| Z_sensor _type_Press_1_bar | 9 |
| Z_sensor _type_Press_2_bar | 10 |
| Z_sensor _type_Press_7_bar | 11 |
| Z_sensor _type_Press_16_bar | 12 |
| Z_sensor _type_Level | 13 |
| Z_sensor_type_Custom | 14 |

**Z_Sensor_digit_analog:**

| | |
|---|---|
| Z_Sensor_digit_analog_Analog | 0 |
| Z_Sensor_digit_analog_Digital | 1 |

**Z_Sensor_FSD_Calib:**

| | |
|---|---|
| Z_Sensor_FSD_Calib_H2O | 0 |
| Z_Sensor_FSD_Calib_IPA | 1 |

**Z_D_F_S_Resolution:**

| | |
|---|---|
| Z_D_F_S_Resolution__9Bit | 0 |
| Z_D_F_S_Resolution__10Bit | 1 |
| Z_D_F_S_Resolution__11Bit | 2 |
| Z_D_F_S_Resolution__12Bit | 3 |
| Z_D_F_S_Resolution__13Bit | 4 |
| Z_D_F_S_Resolution__14Bit | 5 |
| Z_D_F_S_Resolution__15Bit | 6 |
| Z_D_F_S_Resolution__16Bit | 7 |

**Z_MUX_DRI_Rotation:**

| | |
|---|---|
| Z_MUX_DRI_Rotation_Shortest | 0 |
| Z_MUX_DRI_Rotation_Clockwise | 1 |
| Z_MUX_DRI_Rotation_CounterClockwise | 2 |

**Z_MUX_DRI_Action:**

| | |
|---|---|
| Z_MUX_DRI_Action_Home | 0 |
| Z_MUX_DRI_Action_SerialNumber | 1 |