Elveflow® SDK

USER GUIDE

June 2019

ELVEFLOW
PLUG & PLAY MICROFLUIDICS

READ THIS MANUAL CAREFULLY BEFORE USING THE SDK

This manual must be read by every person who is or will be responsible for using the SDK.

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

Important ESI 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.

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 pressure regulator, always place liquid reservoirs under instrument (OB1, AF1…)

Specific Guides for Elveflow Instruments

User guides are available for every Elveflow instrument. Check the dedicated guide to correctly set up your experiment before using the Elveflow Smart Interface.

Important remarks

For all programming languages:

Important! If MUX distributor or BFS are used, FTDI drivers are required (http://www.ftdichip.com/Drivers/D2XX.htm)

Also important! Elveflow Smart interface has to be installed to ensure the installation of every resource required to communicate with the instrument. **For X64 libraries**, LabVIEW 2015 X64 Run-time should be installed. It is included in the installation file (Extra Installer for X64 Libraries)

Very important! Do not use simultaneously ESI software and the SDK, some conflict would occur.

Quickstart

1. Read the SDK User Guide,
2. Install ESI software anyway (this will install some required components that will be used with SDK),
3. Open the SDK.zip folder that is located in the ESI software folder,
4. Choose the development tool that's right for you (LabVIEW / MATLAB / Python etc ...),
5. Open the appropriate folder and try to run the example having modified the necessary elements described in the SDK User Guide (path, instrument name, etc ...):

a. If the example works, you have all the key information required to control the instrument in your personal program.

b. If the example does not work, we can explain to you how to make it work.

c. If you have difficulties and would like to have tailor-made assistance, we can send you a price offer designed to your specific needs.

LabVIEW's SDK programming

All VI are included in ElveflowLLB.llb file.

For every instrument an example to show how to use the SDK LabVIEW is provided.

__AF1_Example__.vi for AF1, __F_S_R_Example__.vi for Flow Reader or Sensor Reader, __MUX_Dist_Example__.vi for MUX Distributor, __MUX_Example__.vi for other MUX series, and __OB1_Example__.vi for OB1.

OB1:

All the available vi for the programing of a customized LabVIEW program are used in the VI "_OB1_Example.vi" contained in the LLB library "ElveflowLLB.llb".

The structure of the main VI 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 finishing operations, please remember to close the OB1 reference using OB1_Close.vi

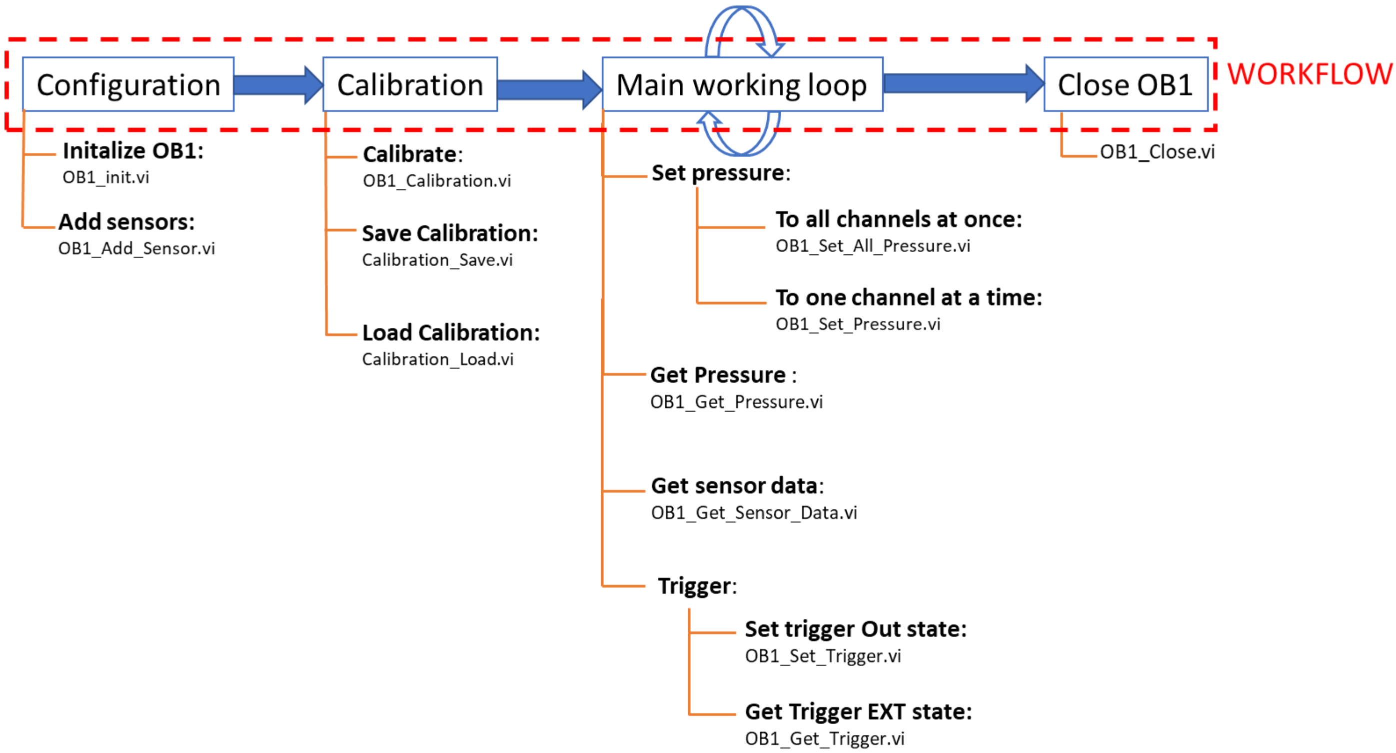
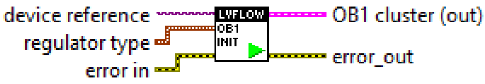
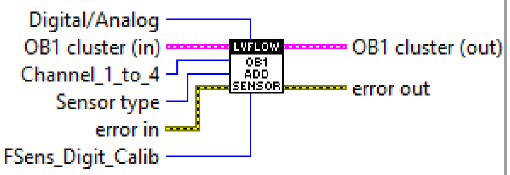
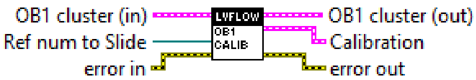
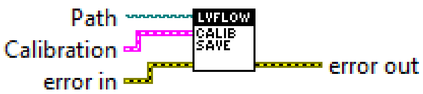
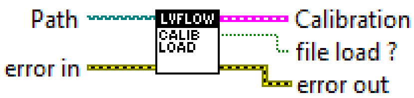
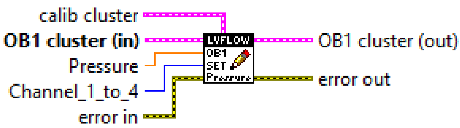
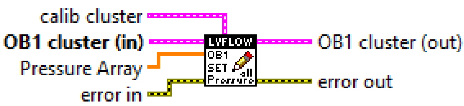
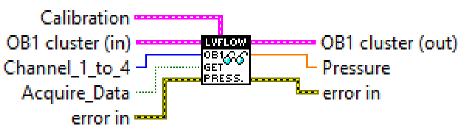
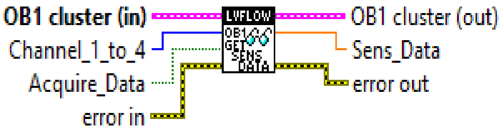
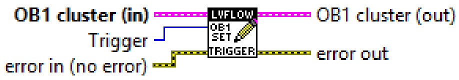
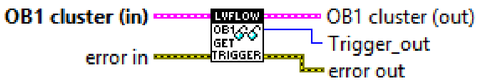
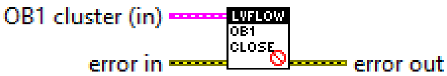


Figure 1 Typical workflow of a custom OB1 program representing the different types of the OB1 SDK VIs

A description of each VI can be found in the table below or by using the "context help" window in LabVIEW.

Icon	File name	Description
Configuration		
	OB1_init.vi	Initialize OB1 with the device reference and the type of regulators to be used. This VI generates an identification cluster of the OB1 to be used in other VIs.
	OB1_Add_Sensor.vi	Add a sensor (flow or pressure) connected to the OB1. You must define type of sensor (digital or analog), the channel it is connected to, the sensor type (80µL/min…. etc.) and the type of fluid for calibration. For digital sensors, the sensor type is automatically detected. For other sensors, these parameters are not considered. In case the sensor is not compatible with the OB1 version, or no digital sensor is detected a pop-up will inform the user.
Calibration		
	OB1_Calibration.vi	Launch a new OB1 calibration and return the calibration array. Ref num to Slide indicates the progress of the calibration. Once the calibration is done, a cluster of calibration data is generated as an output to use for pressure control. Before Calibration, ensure that ALL channels are properly closed with adequate caps.
	Calibration_Save.vi	Saves the actual calibration to the desired path. The function prompts the user to choose a path if no path is specified.
	Calibration_Load.vi	Load the calibration file located at Path and returns the calibration parameters in the Calibration cluster. 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
Operation		
	OB1_Set_Pressure.vi	Set the desired value of pressure in the desired channel. Must use the Calibration cluster and the OB1 cluster for the setting of pressure to work properly.
	OB1_Set_All_Pressure.vi	Works similarly as the vi "OB1_Set_Pressure.vi" except that it sets all the target values of pressure at once using an array as input. This vi needs the calibration and OB1 clusters.
	OB1_Get_Pressure.vi	Read the pressure of a selected channel. As with get-sensor_data, if Aquire_Data is TRUE, values of all regulator and analog sensor are read at once. Thus, to save computational time, you can set the value on FALSE for the other channels and iterations.

	OB1_Get_Sensor_Data.vi	Read the sensor data on the requested channel. This function only convert data that are acquired in these units: flow rate: *µl/min*, pressure: *mbar* "Acquire_Data" work as described in the above description of *OB1_Get_Pressure.vi*. For Digital Sensors, this parameter has no impact NB: For Digital Flow Sensor, If the connection is lost, OB1 will be reseted and the returned value will be zero.
	OB1_Set_Trigger.vi	Set the trigger Out (EXT) of the OB1 0=>Low(0V) 1=>High(3.3V)
	OB1_Get_Trigger.vi	Get the state of the trigger IN (INT). If nothing is connected it returns a High state. 0=>Low(0V) 1=>High(3.3V)
Close OB1 resource		
	OB1_Close.vi	Close the communication with the OB1 defined by its appropriate cluster.

AF1:

AF1 has the same basic functions as an OB1. Thus, it is composed of the same kind of SDK VIs. Please see the example VI "_AF1_Example.vi" contained in the LLB library "ElveflowLLB.llb" for a practical use of all the available AF1 development VIs in one example. There are some VIs that are jointly used for OB1 and AF1 handling (Calibration_Save.vi and Calibration_Load.vi). A schematic description of the working example and the four groups of VIs is illustrated in the following figure.

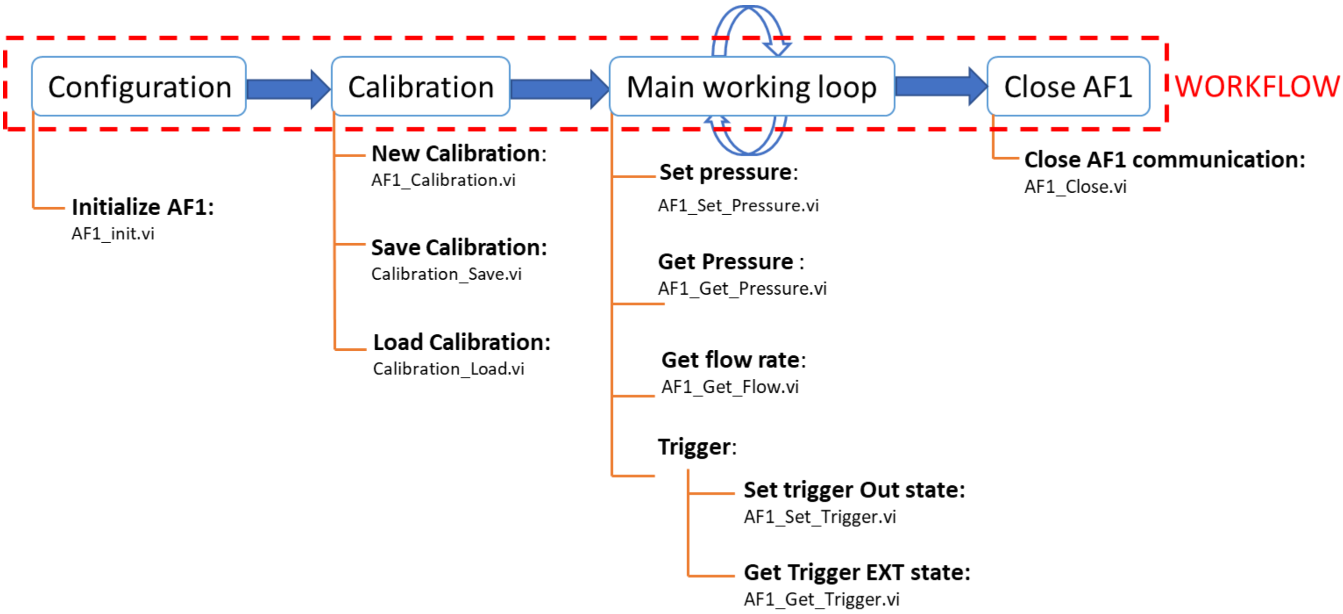
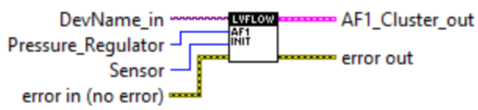
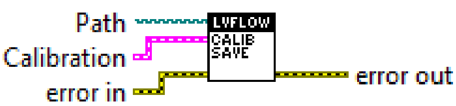
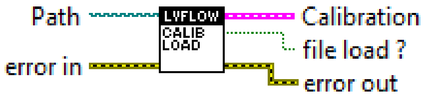
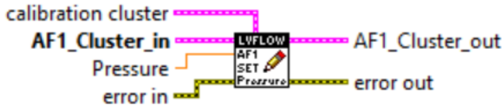
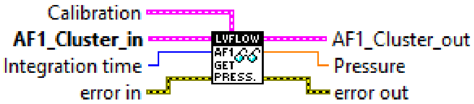
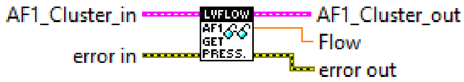
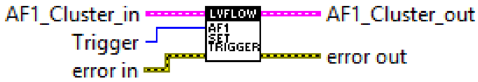
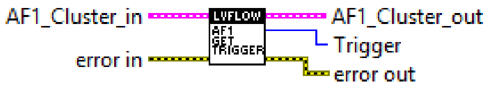
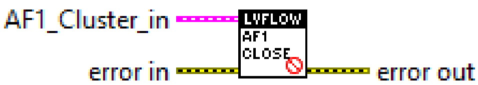


Figure 2 Typical workflow of a custom AF1 program representing the different types of the AF1 SDK functions

A description of each VI can be found in the table below or by using the "context help" window in LabVIEW.

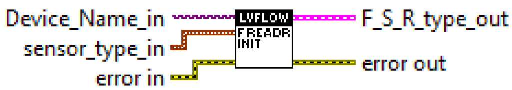
Icon	File name	Function
Configuration		
	AF1_init.vi	Initialize AF1 with the device reference and the type of regulator and sensor to be used. This VI generates an identification cluster of the AF1 to be used with other VIs. AF1 can only work with analog flow sensors.
Calibration		
	AF1_Calibration.vi	Launch a new AF1 calibration and return the calibration cluster. Ref num to Slide indicates the progress of the calibration. Once the calibration is done, a cluster of calibration data is generated as an output to use for pressure control. Before Calibration, ensure the channel is properly closed.
	Calibration_Save.vi	Saves the actual calibration to the desired path. The function prompts the user to choose a path if no path is specified.

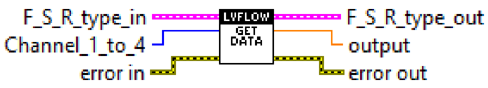
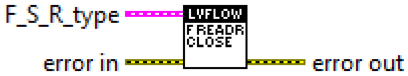
	Calibration_Load.vi	Loads the calibration cluster from a selected path. This VI asks the user to choose the path if Path is not valid or empty.
Operation		
	AF1_Set_Pressure.vi	Set the desired value of pressure in the desired channel. This vi needs the Calibration cluster and the AF1 cluster for the setting of pressure to work properly.
	AF1_Get_Pressure.vi	Read the pressure of the AF1 with a certain integration time. Calibration cluster is required. Pressure unit: mbar.
	AF1_Get_Flow.vi	Get the Flow rate from the flow sensor connected on the AF1. Units: µl/min
	AF1_Set_Trigger.vi	Set the trigger Out (EXT) of the AF1 0=>Low(0V) 1=>High(5V)
	AF1_Get_Trigger.vi	Get the state of the trigger In (INT). If nothing is connected it returns a High state. 0=>Low(0V) 1=>High(5V)
Close OB1 resource		
	AF1_Close.vi	Close the communication with the AF1 defined by its appropriate cluster.

FSR and old version MSR:

All the available vi for the programming of a customized LabVIEW program are used in the VI "_F_S_R_Example.vi" contained in the LLB library "ElveflowLLB.llb".

There are three available VI's that can be used when using the sensor reader (FSR or old MSR).

Icon	File name	Description
	F_S_R_Init.vi	Initiate the communication with the FSR (MSR). This VI generates an identification cluster of the instrument to be used with other VIs.

	F_S_R_Get_Data.vi	Read the sensor data on the requested channel with a unit of flow rate in µl/min.
	F_S_R_Close.vi	Close the communication with the sensor reader and free the resources.

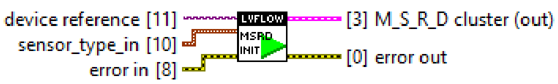
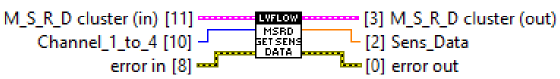
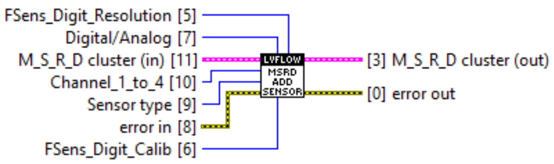
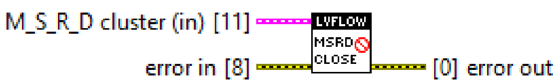
Important notes:

- Flow reader can only accept Flow sensors.
- Sensors connected to channel 1-2 and 3-4 should be the same type otherwise they will not be considered and the user will be informed by a prompt message.
- Sensor reader and Flow reader cannot read digital sensors.

MSRD:

All the available vi for the programming of a customized LabVIEW program are used in the VI "_M_S_R_D_Example.vi" contained in the LLB library "ElveflowLLB.llb".

There are four available VI's that can be used when using the sensor reader which is able to read digital sensors. If your sensor reader is new, it is a MSRD. These VI do not work with FSR or old MSR.

Icon	File name	Description
	M_S_R_D_Init.vi	Initiate the communication with the MSR. Sensor type has to be defined here and in the M_S_R_D_Get_Data.vi. This VI generates an identification cluster of the instrument to be used with other VIs.
	M_S_R_D_Get_Data.vi	Read the sensor data on the requested channel with a unit of flow rate in µl/min and pressure in mbar.
	M_S_R_D_Add_Sensor	Add a sensor (flow or pressure) connected to the OB1. You must define type of sensor (digital or analog), the channel it is connected to, the sensor type which has to be the same as for the Init step (80µL/min…. etc.) and the type of fluid for calibration. For digital sensors, the sensor type is automatically detected. For other sensors, these parameters are not considered. In case the sensor is not compatible with the MSRD version, or no digital sensor is detected a pop-up will inform the user.
	M_S_R_D_Close.vi	Close the communication with the sensor reader and free the resources.

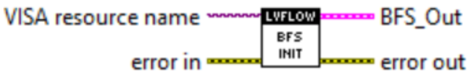
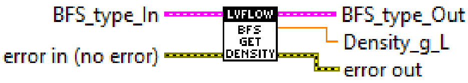
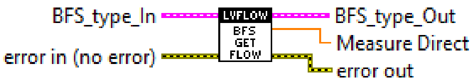
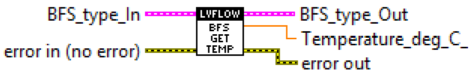
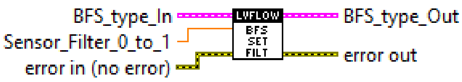
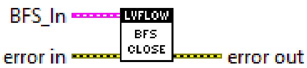
Important notes:

- Sensors connected to channel 1-2 and 3-4 should be the same type otherwise they will not be considered and the user will be informed by a prompt message.
- Sensor type has to be declared in the Init and in the Add Sensor step and has to be the same for both.

BFS:

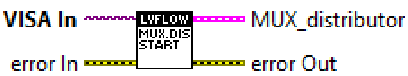
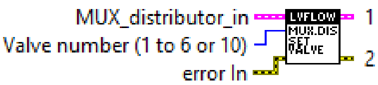
Please see the example file "_BFS_Example.vi" for a standard usage of the available BFS VI's. As with other instruments, there are three steps for programming: Initialization, instrumentation and resource liberation. Please note that for this particular sensor, in order to measure the flow rate (µL/min),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

The table below gives a description of each vi.

Icon	File name	Description
Configuration		
	BFS_Init.vi	Initiate the communication with the BFS sensor and gets the actual sensor configuration (scale)
Operation		
	BFS_Get_Density_val.vi	Get the actual volumetric mass density (g/L). This operation is required in order to obtain the flow rate.
	BFS_Get_Flow_val.vi	Measure the fluid flow in µL/min. **You have to measure the density beforehand so that flow measurement works properly**. Please ensure that the target fluid is inside the BFS when measuring the density. If you get -inf or +inf, the density wasn't correctly measured.
	BFS_Get_Temperature_val.vi	Measure the fluid temperature in °C.
	BFS_Set_Filter_val.vi	Set the instrument's filter. Default value is "0.1". Maximum filtering value (slow response): 0.000001 Minimum filtering value, no filter (fast response time):1.
Close BFS resource		
	BFS_Close.vi	Close the BFS communication and free the resource.

MUX DISTRIBUTOR:

Please see the example file "_Mux_Dist_Example.vi" for a standard usage of the available MUX DISTRIBUTOR VI's. The following table gives a description of the three MUX DISTRIB VIs.

Icon	File name	Function
	MUX_Dist_Init.vi	Establish connection with MUX DISTRIBUTOR. You must input a VISA reference and choose the COM port.
	MUX_Dist_SetValve.vi	Switch the MUX DISTRIBUTOR to the desired valve.

MUX_distributor_in, error In → MUX_distributor_out, selected_Valve, error Out	MUX_Dist_GetValve.vi	Get the actual valve number. If the valve is changing, function returns 0.
MUX_distributor_in, error In → error Out	MUX_Dist_Close.vi	End the communication with the MUX DISTRIBUTOR and free the VISA resource.

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 VIs to start and end the communication. The table below gives the description of each VI. The example VI "_Mux_Example_.vi" illustrate the usage of all these VIs.

Instrument	Icon and VI name	Description
Common to all MUX Series	**MUX_G_Init.vi** DevName, error in → MUX_type, error out	Initializes the instrument using the device name and return the identification cluster.
	MUX_G_Close.vi MUX_type, error in → error out	Closes the task and releases the allocated resources.
	MUX_G_Set_Trigger.vi MUX_type (in), Trigger, error in → MUX_type (out), error out	Set the trigger Out (EXT) 0=>Low(0V) 1=>High(5V)
	MUX_G_Get_Trigger.vi MUX_type (in), error in → MUX_type (out), Trigger, error out	Get the state of the trigger In (INT). If nothing is connected it returns a High state. 0=>Low(0V) 1=>High(5V)
MUX WIRE	**MUX_G_Wire_Set_Valve_Array.vi** MUX_type (in), MUX_Wire_Valves_type, error in → MUX_type (out), error out	Set the valve array of the MUX Wire. The Valve array is a 1x16 matrix of booleans representing the valves connected to the instrument (TRUE for open and FALSE for close).
MUX FLOW SWITCH	**MUX_G_Set_Valve_Array.vi** MUX_type (in), Valves, error in → MUX_type (out), error out	Set the valve array of the instrument. Valve array here is a matrix of 4x4 booleans that control the internal valves. An ON value opens the corresponding internal valve and lets the fluid flow.
MUX CROSS CHIP	**MUX_G_Set_Valve.vi** Open/Close, MUX_type (in), Input, Ouput, error in → MUX_type (out), error out	Set the state of one valve of the instrument using the Input and Output parameters. Theses parameters correspond to the fluidic inputs and outputs. This function has the particularity to open and close the communication channel on each call. You can then use it without initialization or closing steps.

C++, MATLAB and Python SDK programming:

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 the next section. They will allow you to quickly grasp the specifics of each language and to start developing your custom software.

Finally, at the end of this document, you can find an exhaustive list of constants and prototypes. You can also find a list of errors with their corresponding signification.

Important notes:

- 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.

Specifics of C++, MATLAB and Python SDK programming:

C++:

Not all compilers work with the DLL. Visual studio works.
An Example has been written for every instrument, to shows how to use every function of the SDK. These examples are included in the SDK folder (…\DLL64\Example_DLL64_Visual_Cpp\ElveflowDLL\OB1.cpp for example).

Please remember to add the directory that contains the DLL library in Visual studio or another compiler.

Note: Please remember to include the "Elveflow64.h" located in the DLL library to the source code you are developing. It contains all the constants definition, aliases and functions.

Example using visual C++:

Some complete examples are compiled and embedded within the SDK. Each example has a source code that allows to use all the available SDK functions.

For x32 or x64 operating systems:

- ...\DLL32\Example_DLL32_Visual_Cpp\ElveflowDLL\Debug

- ...\DLL32\Example_DLL32_Visual_Cpp\ElveflowDLL\Release

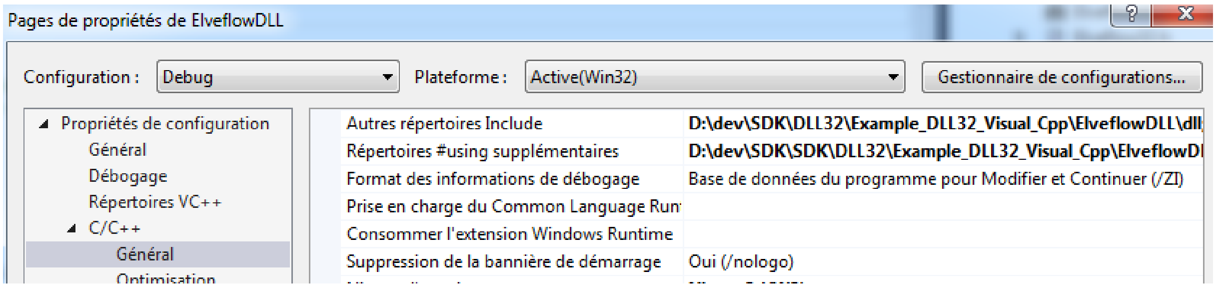
For x64 operating system only:

- ...\DLL64\Example_DLL64_Visual_Cpp\ElveflowDLL\x64\Debug

- ...\DLL64\Example_DLL64_Visual_Cpp\ElveflowDLL\x64\Release

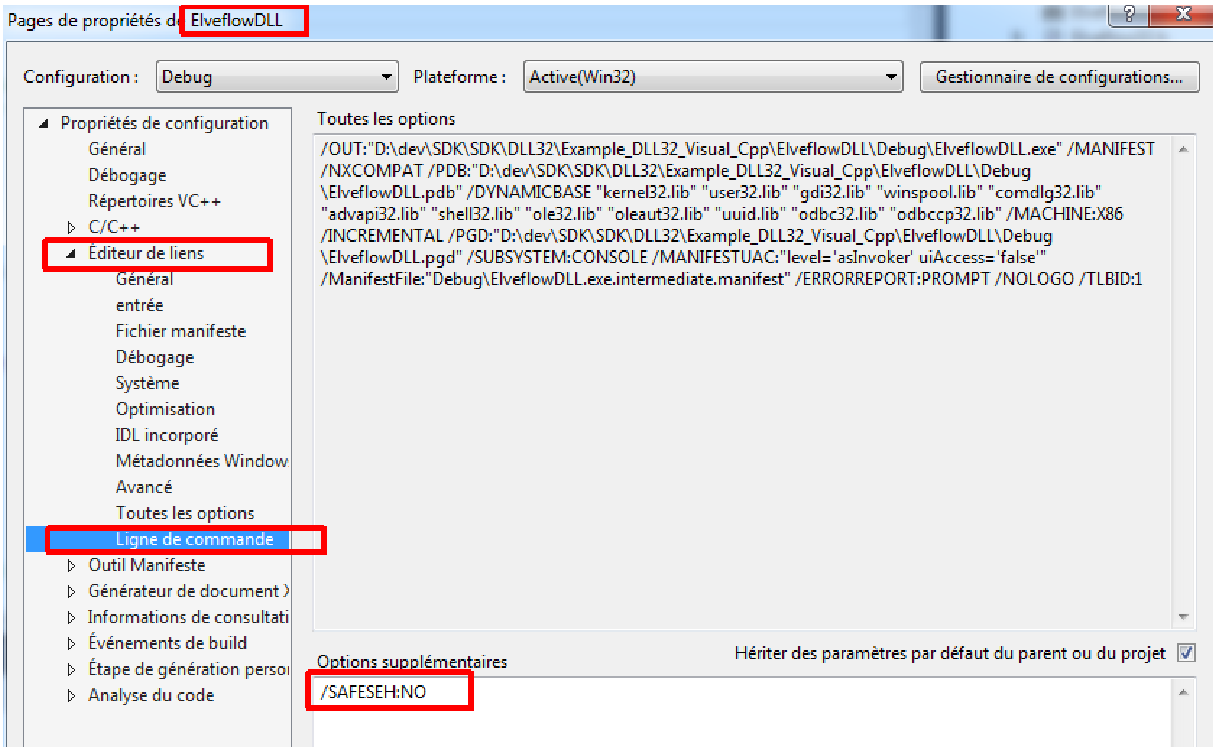
Those examples will not work properly for your specific device (because the device name and configuration are hard coded within the code). However, because each example has a source code that allows to use all the available SDK functions, testing these executables will allow you to see if the DLL is properly working.

Important! Remember to add the directory that contains the dll in the additional directory (project -> property: C++ -> general -> additional Include Directories) and to include all files in the dll folder.



Be careful: Ensure that you are in Project properties, and not in one of the CPP file properties.

For release executable add /SAFESEH:NO to the linker (Project properties -> linker -> Command lines)



MATLAB:

Important:

- In order to load and use DLL, run MATLAB as administrator.
- In order to load and use Elveflow DLL, the compiler should be either Visual C++ Professional or Windows SDK 7.1. To check what is the actual default compiler, type mex -setup c++ in MATLAB command line.

Microsoft visual studio can be downloaded from the following link:

https://visualstudio.microsoft.com/fr/vs/older-downloads/

To check which compilers are compatible with your version of MATLAB, check the following link:

https://mathworks.com/support/compilers.html

https://mathworks.com/support/sysreq/previous_releases.html

Once installed, set the new compiler as default using the command mex -setup c++.

MATLAB does not support pointers natively, therefore the function "libpointer" can be called to create them.

A description of the function is provided in the .m file. It uses a similar prototype as the C++ dll. To learn how to use them, one example for every instrument is available in Elveflow SDK VY\Elveflow SDK VY\MATLAB_XX\Example where XX is either 32 or 64 depending on your MATLAB version and Y is your working version.

For each custom program that you will develop, please remember to add the path to the functions "***.m", to the DLL library and to the path of your main program. The SDK "**.m" functions are linked with their corresponding DLL functions.

Once these various paths added, you will need to load the Elveflow DLL library using the function ***Elveflow_Load***. This function doesn't need any parameter and is **required** to program all the instruments.

At the end of your program, you should **end the communication** by closing the communication with the instrument, clear the pointers and unload the DLL with **Elveflow_Unload**.

Python:

For the Python code to work, you should add the paths to the **DLL library** and the path to the **ElveflowXX.py** (XX=32 or 64). These will allow to load the corresponding C based functions and to define all functions prototypes for use with the Python library respectively.

Note 1: Please remember to edit the path of the DLL library in the 'Elveflow64.py' file.

In order to load pointer array (as calibration) the library **ctypes** is used:

C_double*1000 for calibration (AF1 &OB1)

c_double*4 for pressure_array_out (OB1)

c_int32*16 for array_valve_in (MUX).

Call these variables with the function byref(). The byref() function allows to pass the parameters by reference (i.e whenever you need to handle pointers). "byref()" is used in the instruments examples to declare the mentioned arrays above. This function is needed because some of the DLL library's functions expect a parameter as a pointer to a data type to write into the corresponding location. This is also known as passing parameters by reference.

An example has been written for every instrument, to show how to use every SDK function. Those examples are included in the SDK folder (in python_XX/Example where XX is either 32 or 64 depending on your Python version).

Note 2: Please remember to encode the string of characters (for example with the device name or library path) using ASCII (.encode('ascii')).

In a following section of this document, a description of each instrument's functions will be given.

Description of SDK functions for each instrument:

OB1:

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.

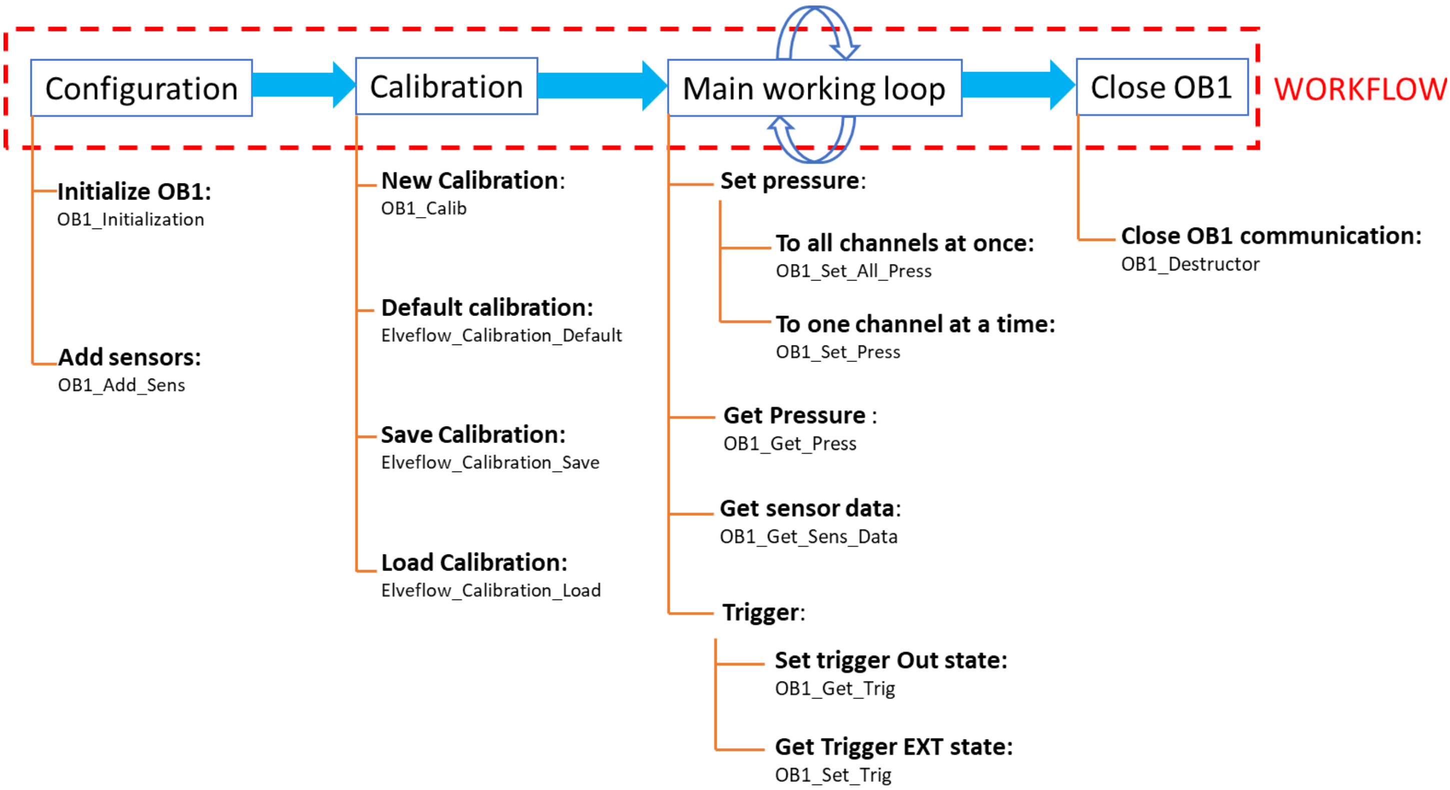


Figure 3 Typical workflow of a custom OB1 program representing the different types of the OB1 SDK functions

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.

Function / File name	Inputs/outputs	Description
Configuration		
`OB1_Initialization (Device_Name, Reg_Ch_1, Reg_Ch_2, Reg_Ch_3, Reg_Ch_4, OB1_ID_out)`	**Inputs:** - Device_Name: Instrument ID (found using NI Max tool) - Reg_Ch_X: 4 regulators type numbers (see Z_regulator_type table) **Output:** - OB1_ID_out: Generated OB1 identification number	Initialize the OB1 device using device name and regulators type. It returns the OB1 ID number (number >=0) to be used with other functions. If an error occurs the return value will be -1. This ID is needed and will be used with other functions to identify the targeted OB1.

`OB1_Add_Sens` `(Channel_1_to_4, SensorType, DigitalAnalog, FSens_Digit_Calib, FSens_Digit_Resolution )`	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization - Channel_1_to_4: Channel number (1 to 4) - SensorType: see Z_sensor_type table. - DigitalAnalog : see Z_Sensor_digit_analog table. - FSens_Digit_Calib : see values Z_Sensor_FSD_Calib in the table. - FSens_Digit_Resolution: see values Z_D_F_S_Resolution in the table.	Add sensor to OB1 device. Select the channel n° (1-4) and the sensor type. For Flow sensor, the type of communication (Analog/Digital) and the Calibration (H20 or IPA) should be specified (only for the digital sensors). For digital sensors, the sensor type is automatically detected. For other sensors, these parameters are not considered. In case the sensor is not compatible with the OB1 version, or no digital sensor is detected a pop-up will inform the user.
Calibration		
`Elveflow_Calibration_Default` `(Calib_Array_out, len)`	**Input**: - len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: Calibration array (pointer)	Get the default calibration and set it as the chosen calibration.
`OB1_Calib` `(OB1_ID_in, Calib_array_out, len)`	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization - len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: output calibration array (pointer)	Launch a new OB1 calibration and return the calibration array. Before Calibration, ensure that ALL channels are properly closed with adequate caps.
`Elveflow_Calibration_Save` `(Path, Calib_Array_in, len)`	**Inputs**: - Path: Calibration path - Calib_Array_in: Calibration array to be saved (pointer) - len: length of the calibration array	Save the Calibration array in the file located at *Path*. The function prompts the user to choose the path if Path is not valid, empty or not a path.
`Elveflow_Calibration_Load` `(Path, Calib_Array_out, len)`	**Inputs**: - Path: Calibration path - len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: Calibration array to be loaded (use a pointer)	Load the calibration file located at Path and returns 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.
Operation		
`OB1_Set_Press` `(OB1_ID, Channel_1_to_4, Pressure, Calib_array_in, Calib_Array_len)`	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Channel_1_to_4: Channel number (1 to 4) - Pressure: Target pressure in mbar - Calib_Array_in: Calibration array to be saved (pointer)	Set the pressure of the OB1 selected channel, Calibration array and length are required.

	- Calib_Array_len: length of the calibration array (use default value = 1000)	
OB1_Set_All_Press (OB1_ID, Pressure_array_in, Calib_array_in, Pressure_Array_Len, Calib_Array_Len)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Pressure_array_in: array of target pressure values (mbar) for all channels. The first number of the array correspond to the first channel, the seconds number to the seconds channels and so on. All the number above 4 are not taken into - Calib_Array_in: Calibration array to be saved (pointer) - Pressure_Array_Len: size of the pressure array. - Calib_Array_len: length of the calibration array (use default value = 1000).	Works similarly as the function "OB1_Set_Press" except that it sets all the target values of pressure at once using an array as input. A calibration array is required (use Set_Default_Calib if required).
OB1_Get_Press (OB1_ID, Channel_1_to_4, Acquire_Data1True0False, Calib_array_in, Pressure, Calib_Array_len)	**Inputs:** - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Channel_1_to_4: Channel number (1 to 4) - Acquire_Data1True0False: new value acquisition (=1) or buffered value acquisition (=0). - Calib_Array_in: Calibration array to be saved (pointer) - Calib_Array_len: length of the calibration array (use default value = 1000) **Output**: - Pressure: pointer to read pressure. Changed value "Pressure.Value"	Read the pressure of a selected channel. Calibration array and length are required. As with get-sensor_data, if "Acquire_Data1True0False" = 1, values of all regulators and analog sensors are read at once and stored in computer memory. Thus, to save computational time, you can set the value on to 0 for the other channels and iterations in order to read from the buffer. For Digital Sensors, this parameter has no impact NB: For Digital Flow Sensor, If the connection is lost, OB1 will be reseted and the returned value will be zero.
OB1_Get_Sens_Data (OB1_ID, Channel_1_to_4, Acquire_Data1True0False, Sens_Data)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Channel_1_to_4: Channel number (1 to 4) - Acquire_Data1True0False: new value acquisition (=1) or buffered value acquisition (=0). **Output**: - Sens_Data: Read value (pointer) stored in "Sens_Data.Value".	Read the sensor data on the requested channel. This Function only convert data that are acquired in these units: flow rate *µl/min*, pressure: *mbar* "Acquire_Data1True0False" works as described in OB1_Get_Press. For digital sensors, this parameter has no impact NB: For digital flow sensors, If the connection is lost, OB1 will be reseted and the return value will be zero.
OB1_Set_Trig (OB1_ID, trigger)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Trigger: trigger state (High or Low)	Set the trigger Out (EXT) of the OB1 0=>Low(0V) 1=>High(3.3V)

`OB1_Get_Trig (OB1_ID, Trigger)`	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) **Output**: - Trigger: Read trigger state (High or Low)	Get the state of the trigger In (INT). If nothing is connected it returns a High state. 0=>Low(0V) 1=>High(3.3V)
Close OB1 resource		
`OB1_Destructor (OB1_ID)`	**Input**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value)	Close the communication with the OB1 using its identification number.

AF1:

AF1 has the same basic functions as an OB1. Thus, it is composed of the same kind of SDK VIs. Please see the example "AF1_Ex__" for a practical use of all the available AF1 development functions in one example. There are some functions that are jointly used for OB1 and AF1 handling. A schematic description for each category's functions is illustrated in the following figure.

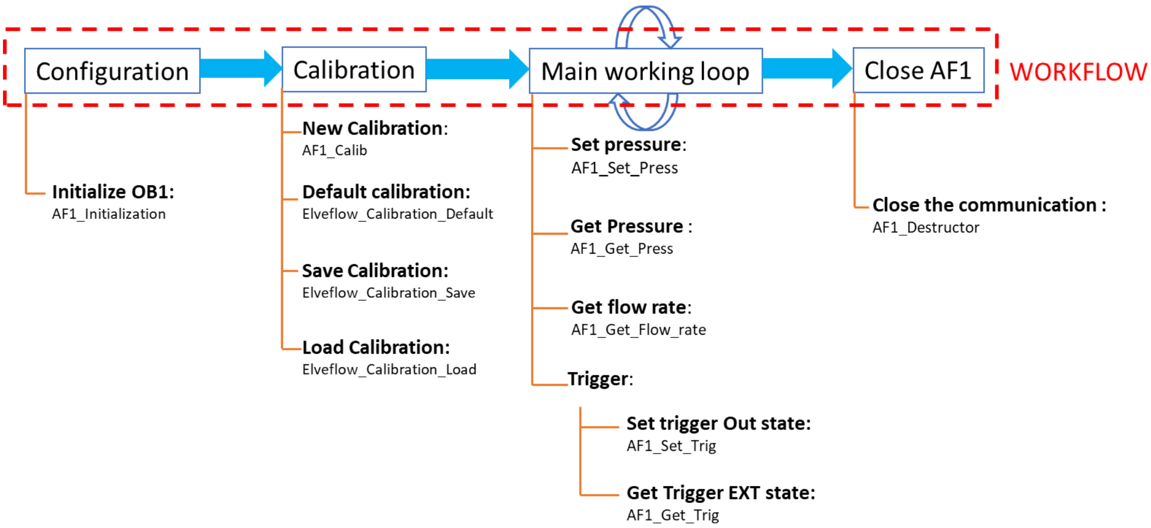


Figure 4 Typical workflow of a custom AF1 program representing the different types of the AF1 SDK functions

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.

Function / File name	Inputs/outputs	Description
Configuration		
`AF1_Initialization (Device_Name, Pressure_Regulator, Sensor, AF1_ID_out)`	**Inputs**: - Device_Name: Instrument ID (found using NI Max tool) - Pressure_Regulator: 4 regulators type numbers (see Z_regulator_type table) - Sensor: see Z_sensor_type table for sensor types corresponding numbers. **Outputs**: - AF1_ID_out: AF1 ID number	Initialize AF1 with the device reference and the type of regulators and sensor to be used. This function returns an identification number of the AF1 to be used in with the other functions. AF1 can only work with analog flow sensors.

Calibration		
`Elveflow_Calibration_Default (Calib_Array_out, len)`	**Inputs**: - len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: Calibration array (pointer)	Get the default calibration and set it as the chosen calibration.
`AF1_Calib (AF1_ID_in, Calib_array_out, len)`	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization - len: length of the calibration array (use default value = 1000) **Outputs**: - Calib_Array_out: output calibration array (pointer).	Launch a new AF1 calibration and return the calibration array. Before Calibration, ensure the channel is properly closed.
`Elveflow_Calibration_Save (Path, Calib_Array_in, len)`	**Inputs**: - Path: Calibration path - Calib_Array_in: Calibration array to be saved (pointer) - len: length of the calibration array (use default value = 1000)	Save the Calibration array in the file located at “Path”. The function prompts the user to choose the path if “Path” is not valid or empty.
`Elveflow_Calibration_Load (Path, Calib_Array_out, len)`	**Inputs**: - Path: Calibration path - len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: Calibration array to be loaded (use a pointer)	Load the calibration file located at Path and returns the calibration parameters in the Calib_Array_out. The function asks the user to choose the path if Path is not valid or empty. The function indicates if the file was found.
Operation		
`AF1_Set_Press (AF1_ID_in, Pressure, Calib_array_in, len)`	**Inputs**: - AF1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Pressure: Target pressure in mbar - Calib_Array_in: Calibration array to be saved (pointer) - Calib_Array_len: length of the calibration array (use default value = 1000)	Set the pressure of the AF1, Calibration array and length are required. Pressure is in mbar.

`AF1_Get_Press` `(AF1_ID_in, Integration_time, Calib_array_in, Pressure, Calib_Array_len)`	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) - Integration_time: sets the integration time of the pressure reading in ms (default value is 100). - Calib_Array_in: Calibration array to be saved (pointer) - Calib_Array_len: length of the calibration array (use default value = 1000) **Output**: - Pressure: pointer to read pressure. Changed value "Pressure.Value"	Read the pressure of the AF1 with a certain integration time. Calibration array and length are required for this function.
`AF1_Get_Flow_rate` `(AF1_ID_in, Flow)`	**Inputs**: - AF1_ID_in: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) **Output**: - Flow: Flow rate expressed in µL/min.	Get the Flow rate from the flow sensor connected on the AF1. Units: µl/min.
`AF1_Set_Trig` `(AF1_ID_in, trigger)`	**Inputs**: - AF1_ID_in: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) - Trigger: set the trigger to 1 (High) or 0 (low)	Set the trigger Out (EXT) of the AF1 0=>Low(0V) 1=>High(5V)
`AF1_Get_Trig` `(AF1_ID_in, trigger)`	**Inputs**: - AF1_ID_in: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) **Output**: - Trigger: Read the trigger value: 1 (High) or 0 (low)	Get the state of the trigger In (INT). 0=>Low(0V) 1=>High(5V)
Close AF1 resource		
`AF1_Destructor` `(AF1_ID_in)`	**Inputs**: - AF1_ID_in: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value)	Close the communication with the AF1 defined by its created ID number.

FSR and old version MSR:

All the available functions for the programming of a customized program are detailed in the example "F_S_Reader_Ex__" contained in the appropriate example folder. These functions work for both the sensor reader (MSR) and the flow reader (FSR).

There are three available functions that can be used:

Function / File name	Inputs/outputs	Description
`F_S_R_Initialization (Device_Name, Sens_Ch_1, Sens_Ch_2, Sens_Ch_3, Sens_Ch_4, F_S_Reader_ID_out)`	**Inputs**: - Device_Name: Instrument ID (found using NI Max tool) - Sens_Ch_X: sensor type (see Z_sensor_type values in table) **Output**: - F_S_Reader_ID_out: MSR ID number	Initiate the F_S_R device using device name (could be obtained in NI MAX) and sensors. It returns the F_S_Reader_ID_out number (number >=0) to be used with other functions.
`F_S_R_Get_Sensor_data (F_S_Reader_ID_in, Channel_1_to_4, output)`	**Inputs:** - F_S_Reader_ID_in: FSR ID number created by F_S_R_Initialization (stored in F_S_Reader_ID_out.Value) - Channel_1_to_4: channel to read (1 to 4). **Output:** - output: read value.	Read the sensor data on the requested channel with a unit of flow rate in µl/min. This function needs the ID number created with F_S_R_Initialization.
`F_S_R_Destructor (F_S_Reader_ID_in)`	**Inputs:** - F_S_Reader_ID_in: FSR ID number created by F_S_R_Initialization (stored in F_S_Reader_ID_out.Value)	Close the communication with the sensor reader defined by its created ID number.

Important notes:

- Flow reader can only accept Flow sensor
- Sensor connected to channel 1-2 and 3-4 should be the same type otherwise they will not be considered and the user will be informed by a prompt message.
- Sensor reader and Flow reader cannot read digital sensors.

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. These functions work for both the sensor reader (MSR) and the flow reader (FSR).

There are four available functions that can be used with the sensor reader which is able to read digital sensors. If your sensor reader is new, it is a MSRD. These VI do not work with FSR or old MSR.

Function / File name	Inputs/outputs	Description
M_S_R_D_Initialization (Device_Name, Sens_Ch_1, Sens_Ch_2, Sens_Ch_3, Sens_Ch_4, M_S_R_D_ID_out)	**Inputs**: - Device_Name: Instrument ID (found using NI Max tool) - Sens_Ch_X: sensor type (see Z_sensor_type values in table) **Output**: - M_S_R_D_ID_out: MSR ID number	Initiate the M_S_R_D device using device name (could be obtained in NI MAX) and sensors type only (to check compatibility). It returns the M_S_R_D_ID_out number (number >=0) to be used with other functions.
M_S_R_D_Get_Sens_Data (M_S_R_D_ID, Channel_1_to_4, Sens_Data)	**Inputs:** - M_S_R_D_ID_in: MSR ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value) - Channel_1_to_4: channel to read (1 to 4). **Output:** - Sens_Data: read value.	Read the sensor data on the requested channel with a unit of flow rate in µl/min and a unit of pressure in mbar. This function needs the ID number created with M_S_R__D_Initialization.
M_S_R_D_Add_Sens (M_S_R_D_ID, Channel_1_to_4, SensorType, DigitalAnalog, FSens_Digit_Calib, FSens_Digit_Resolution)	**Inputs**: - M_S_R_D_ID: MSR ID number created by M_S_R_D_Initialization - Channel_1_to_4: Channel number (1 to 4) - SensorType: see Z_sensor_type table. - DigitalAnalog : see Z_Sensor_digit_analog table. - FSens_Digit_Calib : see values Z_Sensor_FSD_Calib in the table. - FSens_Digit_Resolution: see values Z_D_F_S_Resolution in the table.	Add sensor to MSRD device. Select the channel n° (1-4) and the sensor type (same as for Initialization). For Flow sensor, the type of communication (Analog/Digital) and the Calibration (H2O or IPA) should be specified (only for the digital sensors). For digital sensors, the sensor type is automatically detected. For other sensors, these parameters are not considered. In case the sensor is not compatible with the MSRD version, or no digital sensor is detected a pop-up will inform the user.
M_S_R_D_Destructor (M_S_R_D_ID)	**Inputs:** - M_S_R_D_ID: MSR ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value)	Close the communication with the sensor reader defined by its created ID number.

BFS:

Please see the example file "_BFS_Example.vi" 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.

Function / File name	Inputs/outputs	Description
Configuration		
`BFS_Initialization (Visa_COM, BFS_ID_out)`	**Inputs**: - Visa_COM: Device VISA name in the form of "ASRLXXX::INSTR" that could be found using NI MAX under "VISA resource name". **Output**: - BFS_ID_out: BFS ID number	Initiate the BFS device using device com port. It returns the BFS ID (number >=0) to be used with other functions.
Operation		
`BFS_Get_Density (BFS_ID_in, Density)`	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) **Output** : - BFS_ID_out: BFS ID number	Get fluid density (in g/L) for the BFS defined by the BFS_ID.
`BFS_Get_Flow (BFS_ID_in, Flow)`	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) **Output** : - Flow: flow rate in µL/min.	Measure the fluid flow in µL/min. **You have to measure the density (BFS_Get_Density) beforehand so that the flow measurement works properly.** Please ensure that the target fluid is inside the BFS when measuring the density. If you get -inf or +inf, the density wasn't correctly measured.
`BFS_Get_Temperature (BFS_ID_in, Temperature)`	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) **Output** : - Temperature: temperature in °C.	Measure the fluid temperature in °C.
`BFS_Set_Filter (BFS_ID_in, Filter_value)`	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) - Filter_value: your filter value.	Set the instrument's filter. Default value is "0.1". Maximum filtering value (slow response): 0.000001 Minimum filtering value, no filter (fast response time):1.

Close BFS resource		
`BFS_Destructor (BFS_ID_in)`	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value)	Close the communication with the BFS.

MUX DISTRIBUTOR:

Please see the example file "MUX_Dist_Ex__" for a standard usage of the available MUX Distributor functions. The following table gives a description of the 4 MUX Distributor functions.

Function / File name	Inputs/outputs	Description
`MUX_Dist_Initialization (Visa_COM, MUX_Dist_ID_out)`	**Inputs**: - Visa_COM: Device VISA name in the form of "ASRLXXX::INSTR" that could be found using NI MAX under "VISA resource name". **Output**: - MUX_Dist_ID_out: MUX_Dist ID number	Initiate the MUX_Dist device using device COM port. It returns the MUX_Dist ID (number >=0) to be used with other functions.
`MUX_Dist_Set_Valve (MUX_Dist_ID_in, selected_Valve)`	**Inputs**: - MUX_Dist_ID_in: MUX_Dist ID number created by BFS_Initialization (stored in MUX_Dist_ID_in.Value). - Selected_Valve: desired valve.	Switch the MUX distributor to the desired valve.
`MUX_Dist_Get_Valve (MUX_Dist_ID_in, selected_Valve)`	**Inputs**: - MUX_Dist_ID_in: MUX_Dist ID number created by BFS_Initialization (stored in MUX_Dist_ID_in.Value). **Output**: - Selected_Valve: Active valve.	Get the actual valve number. If the valve is changing, the function returns 0.
`MUX_Dist_Destructor (MUX_Dist_ID_in)`	**Inputs**: - MUX_Dist_ID_in: MUX_Dist ID number created by BFS_Initialization (stored in MUX_Dist_ID_in.Value).	End the communication with the MUX Distributor and frees the VISA resource.

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__" illustrate the usage of all these functions.

Function / instrument	Inputs/outputs	Description
Common to all Mux Series		
`MUX_Initialization (Device_Name, MUX_ID_out)`	**Inputs**: - Device_Name: Instrument ID (found using NI Max tool) **Output**: - MUX_ID_out: MUX ID number	Initialize the instrument using the device name and return the communication identifier (MUX_ID_out) to be used with other functions.

`MUX_Set_Trig (MUX_ID_in, Trigger)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value). - Trigger: set the trigger to 1 (High) or 0 (low).	Set the trigger Out (EXT) 0=>Low(0V) 1=>High(5V)
`MUX_Get_Trig (MUX_ID_in, Trigger)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value). **Output**: - Trigger: read the trigger value: 1 (High) or 0 (low).	Get the state of the trigger In (INT). 0=>Low(0V) 1=>High(5V)
`MUX_Destructor (MUX_ID_in)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value).	Close the communication of the MUX device.
Specific to the type of instrument		
MUX WIRE instrument `MUX_Wire_Set_all_valves (MUX_ID_in, array_valve_in, len)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value). - array_valve_in: Array of 16 elements representing the valve state (0 or 1). - len: array length.	Set the valve array of the MUX WIRE. Valves are set by an array of 16 elements. If the valve value is equal or below 0, valve is close, if it's equal or above 1 the valve is open. If the array does not contain exactly 16 element nothing happened.
MUX FLOW SWITCH instrument `MUX_Set_all_valves (MUX_ID_in, array_valve_in, len)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value). - array_valve_in: Array of 16 elements representing the valve state (0 or 1). - len: array length.	Set the valve array of the instrument. Valve array here is a matrix of 4x4 Booleans that control the internal valves. An ON value opens the corresponding internal valve and lets the fluid flow. If the valve value is equal or below 0, valve is close, if it's equal or above 1 the valve is open. The index in the array indicate the selected valves 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 element nothing happens.
MUX CROSS CHIP `MUX_Set_indiv_valve (MUX_ID_in, Input, Output, OpenClose)`	**Inputs**: - MUX_ID_in: number created by MUX_Initialization (stored in MUX_ID_in.Value). - Input: choice of the input valve - Output: choice of the output valve - OpenClose: set valvle state	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.

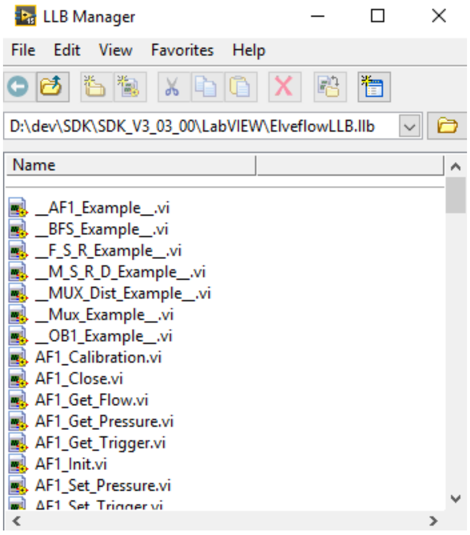
Quick start examples:

This section is here to guide you on how to use and modify the examples in each language. First of all, unzip the SDK file to have your uncompressed SDK folder.

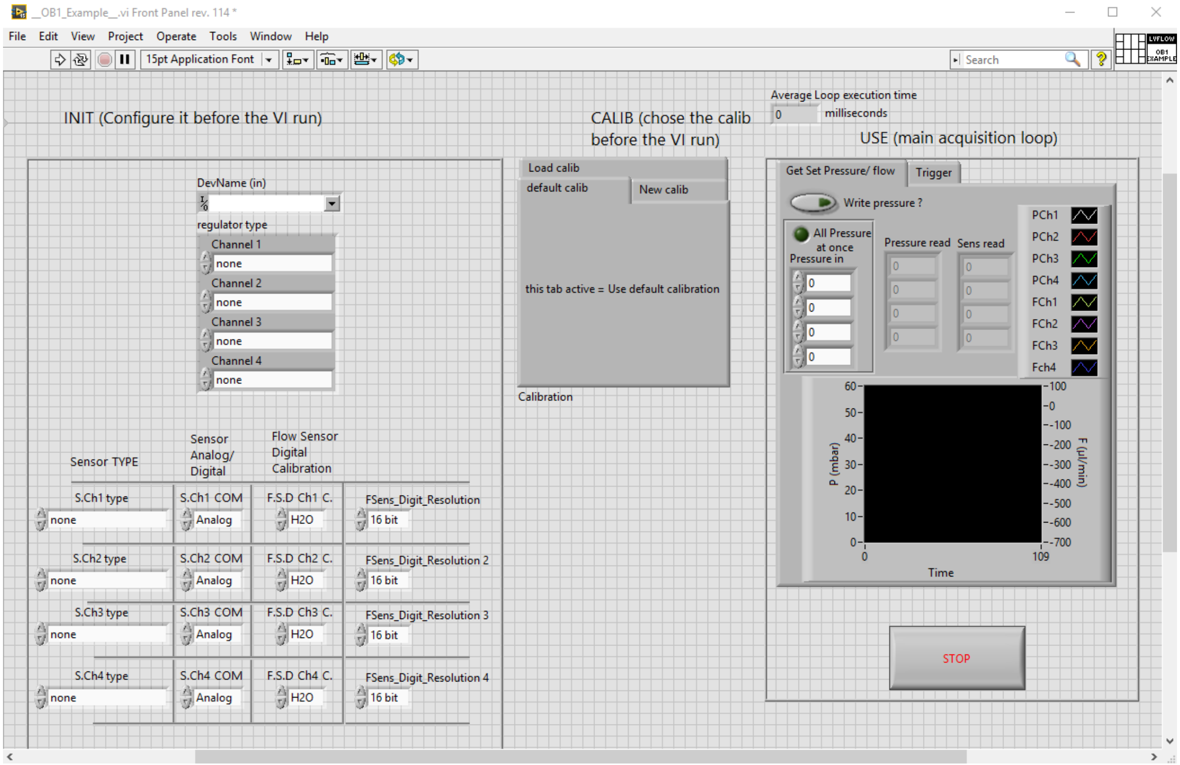
All explanations described here are only for OB1 example, but the principle is the same for other examples/instruments. We will consider for this quick start that we are using an OB1 MK3+ with two regulators 0-200 mbar on channel 1 and 2, one regulator -1-1 bar on channel 3 and one regulator 0-8 bar on channel 4. On this OB1 we have a 1000 µL/min digital flow sensor that we want to use with $H2O$ calibration and 16 bits resolution connected on channel 1 and a 1 bar pressure sensor connected on channel 3.

LabVIEW:

1) In your SDK folder, go to “LabVIEW” folder. There should be a file named “ElveflowLLB.llb”
2) Double click on the file. It will open the following window:

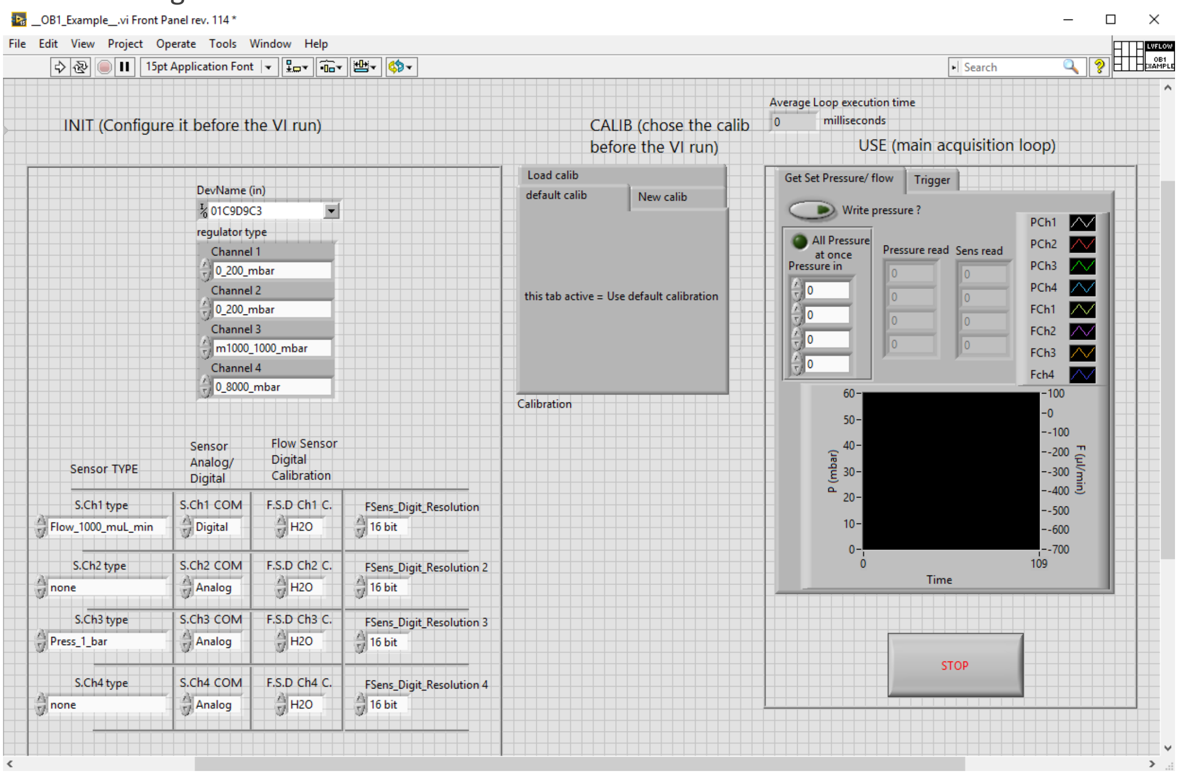


3) Double click on the example you want to run. Here we will open __OB1_Example__.vi
4) If some warnings appear, ignore them. You should now have the following window opened:



5) To test the example, be sure that your OB1 is connected to the computer and turned on. Connect also all the flow sensors you want to use.
6) Considering the OB1 used you need to modify the following elements prior running the VI:
    - DevName (in) (a list will appear with connected
    - regulator type (Channel 1 to 4)
    - S.ChX type (with X=1 to 4)
    - S.ChX COM (with X=1 to 4)
    - FSens_Digit_ResolutionX (with X=1 to 4)

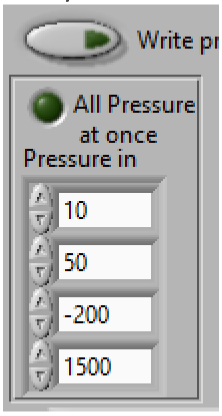
   Considering the OB1 used for this example (described at the beginning of the section) the modified VI should be configured as followed:



7) On the middle tab you can choose to perform calibration, load an existing calibration or use default calibration. For this example we will only use default calibration. Remember that calibration files

generated through ESI cannot be used with SDK. Only SDK generated calibration files can be loaded using SDK.

8) Then the example is ready to be launched and will output pressure readings and sensor readings in the graph and tables.
9) When VI is running, to change pressure, modify values from this table:



And click on “Write pressure?” button to write and unclick it.

Now the example should run, for more details please refer to the block diagram and User Guide.

MATLAB:

1) In your SDK folder, go to “MATLAB_64/Example” or “MATLAB_32/Example”. There should be a file named “OB1_Ex__.m”. Open this file in your MATLAB software. Remember that MATLAB has to run in administrator mode. In this example we will consider that compiler has been configured as recommended previously in this User Guide (MATLAB section).
2) You should obtain a window similar to this one (depending on SDK version you are running):

OB1_Ex__.m

```
1  %%%%%%%%%%%%%%%%%%
2  %INITIALIZATION
3  %%%%%%%%%%%%%%%%%%
4  %add path where the lib Elveflow are stored, load library and set all
5  %required variable (some are pointer to communicate with DLL)
6  %and start the instrument
7  %%%%%%%%%%%%%%%%%%
8
9  %define here the directory where .m, Dll and this script are
10 addpath('D:\dev\SDK\SDK_V3_02_01\MATLAB_64\MATLAB_64');% path for Mathlab"***.m" file
11 addpath('D:\dev\SDK\SDK_V3_02_01\MATLAB_64\MATLAB_64\DLL64');% path for DLL library
12 addpath('D:\dev\SDK\SDK_V3_02_01\MATLAB_64\Example')% path for your script
13
14 %%Always use Elveflow_Load at the begining, it load the DLL
15 Elveflow_Load;
16
17 error =0;% int error to zero, if an error occurs in the dll, an error is returned
18 answer='empty_sring';% store the user answer in this variable
19
20
21 %create equivalent of char[] to communicate with DLL
22 %the instrument name can be found in NI Max
23 Instrument_Name = libpointer('cstring','01C9D9C3'); %01C9D9C3 is the name of my instrument
24
25 %create a pointer for calibrationset
26 CalibSize = 1000;
27 Calibration = libpointer('doublePtr',ones(CalibSize,1));
28
29 %pointer to store the instrument ID (no array)
```

3) To make this example works you need to modify the code to adapt it to your setup. Read the comments to have more details about elements to change (for other examples and this one too). First of all, you need to modify the paths where the dll, scripts and .m files are. Please modify these 3 lines:

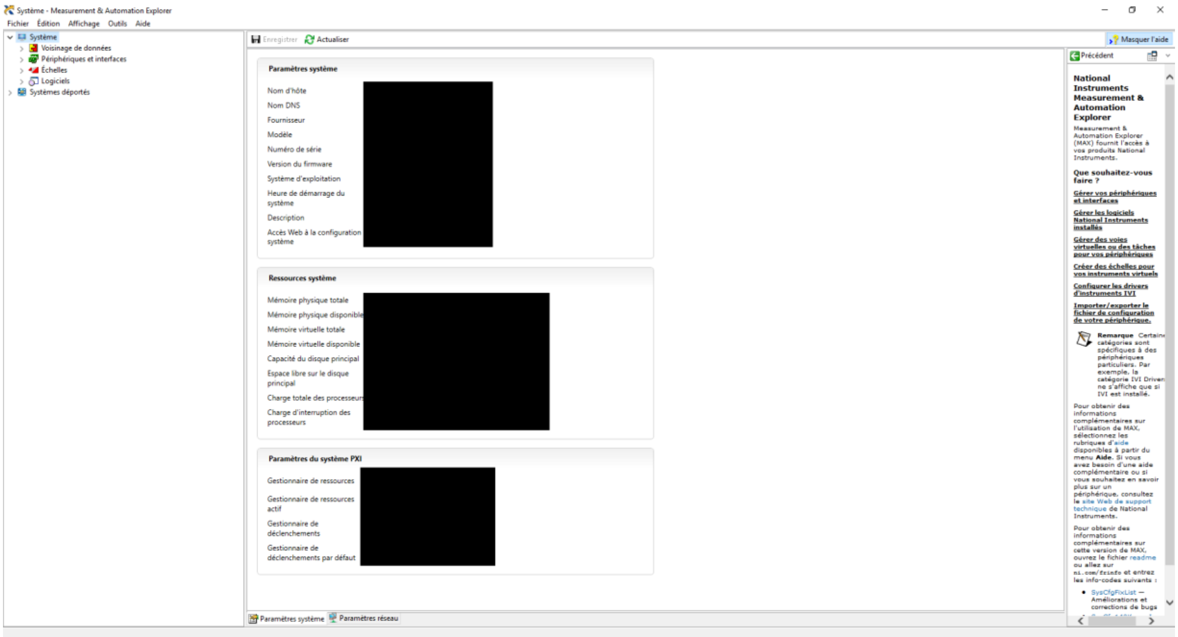
```
9  %define here the directory where .m, Dll and this script are
10 addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\MATLAB_64');% path for Mathlab"***.m" file
11 addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\MATLAB_64\DLL64');% path for DLL library
12 addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\Example')% path for your script
```

4) Then you need to write your instrument name here:

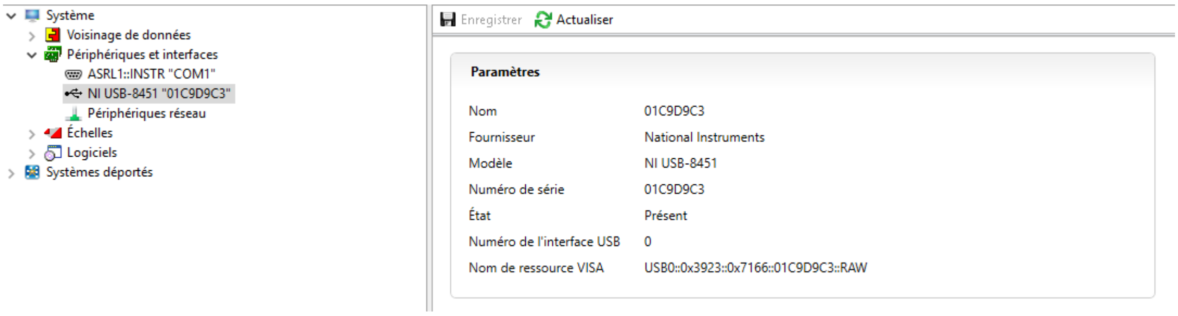
```
21 %create equivalent of char[] to communicate with DLL
22 %the instrument name can be found in NI Max
23 Instrument_Name = libpointer('cstring','01C9D9C3'); %01C9D9C3 is the name of my instrument
```

5) To modify this line (23 here) you need to open a software that has been automatically installed on your computer while installing ESI software. This is “NI MAX”. Find NI MAX on your computer by typing “NI MAX” on the Windows search for example and open it.

6) You should obtain a window similar to this one (except the black squares and language):



7) Expand the “Devices and Interfaces” tab to reveal connected instruments. Depending on the devices connected to your computer you will find multiple lines. In our case (OB1) we have to find a line with “NI USB-8451”. When you found it, click on it. You should obtain a window similar to this one:



8) The name of the instrument is written in the “Name” part. Here it is “01C9D9C3” but depending on the device connected (MSR, AF1 etc...) the name could be “Dev1” for example.
Please copy the name of the instrument and go back to MATLAB.

9) Write the name of the instrument instead of the already written name between ‘xxxxx’. In the case of my instrument the modified window is as follow:

```
%create equivalent of char[] to communicate with DLL
%the instrument name can be found in NI Max
Instrument_Name = libpointer('cstring','01C9D9C3'); %01C9D9C3 is the name of my instrument
```

10) Now we need to initialize our instrument. The initialization function for this example is the following:

```
%Initiate the device and all regulators and sensors types (see user
%guide for help
error=OB1_Initialization(Instrument_Name,1,2,4,3,Inst_ID);
CheckError(error);
```

11) Modify the function depending on your OB1. Considering the OB1 used for this example (described at the beginning of the section) the modified function should look like this:

```
%Initiate the device and all regulators and sensors types (see user
%guide for help
error=OB1_Initialization(Instrument_Name,1,1,4,3,Inst_ID);
CheckError(error);
```

Refer to the corresponding function in the User Guide (here it is “OB1_Initialization”) and the table to know which parameters to input in your case.

12) In the case of OB1, we need to declare sensors that are connected to the OB1. If you do not have any sensor, please skip the following step to step 14. In the case of this example we need to add new lines to the code. The following lines are used to add sensors:

```
%add digital flow sensor. Valid for OB1 MK3+ only, if sensor not detected it will throw an error ;
%error=OB1_Add_Sens(Inst_ID.Value,1,8,1,0,7); %add digital flow sensor. Valid for OB1 MK3+ only, if sensor not detected it
%CheckError(error);
```

13) Uncomment the "OB1_Add_Sens" and "CheckError" functions. Depending on the sensor connected the "OB1_Add_Sens" has to be modified. In our example we need to add two sensors. The new code should be as follow:

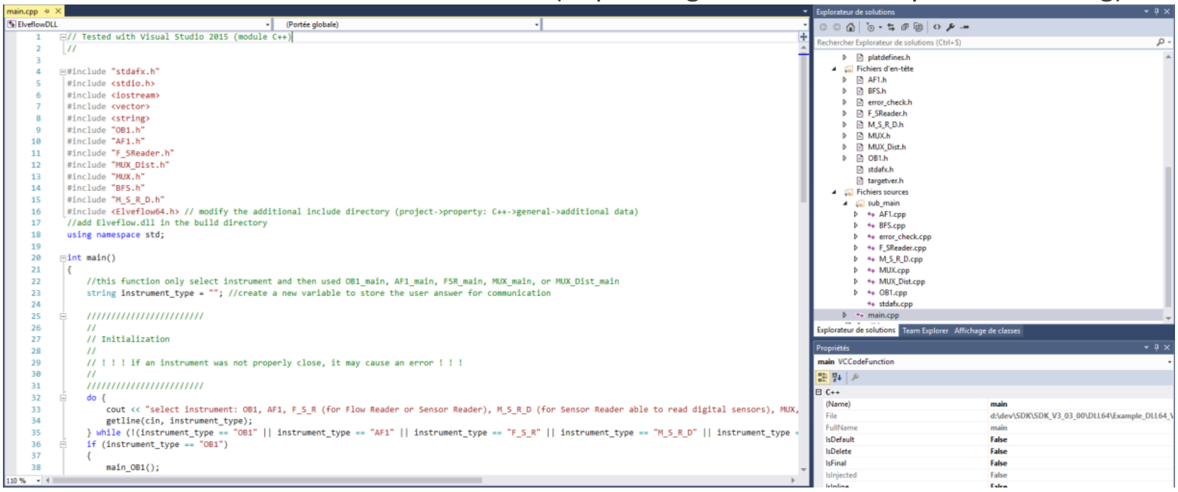
```
error=OB1_Add_Sens(Inst_ID.Value,1,5,1,0,7);
CheckError(error);
error=OB1_Add_Sens(Inst_ID.Value,3,8,0,0,7);
CheckError(error);
```

Note that the last two parameters are unused in the case of the pressure sensor used in this example. Refer to the corresponding function in the User Guide (here it is "OB1_Add_Sens") and the table to know which parameters to input in your case.
If you have more than two sensors, add that many lines (followed by CheckError) up to 4 (which are the four channels one the OB1 where the sensors are physically connected). Channel number is the first parameter of the function after ID.

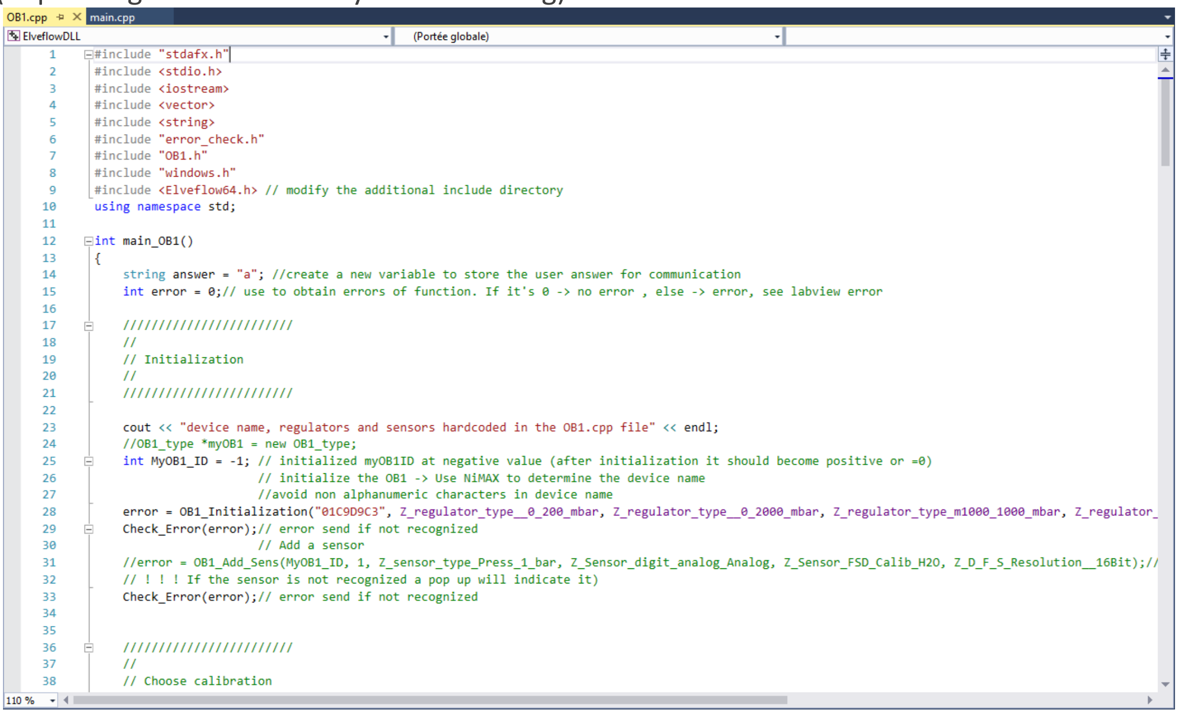
10) Then the example is ready to be launched. You can use default calibration or perform a new one. You can also load a calibration file but remember that calibration files generated through ESI cannot be used with SDK. Only SDK generated calibration files can be loaded using SDK.
11) Please follow instructions displayed on the screen to ask for pressure, sensor data etc...

C++:

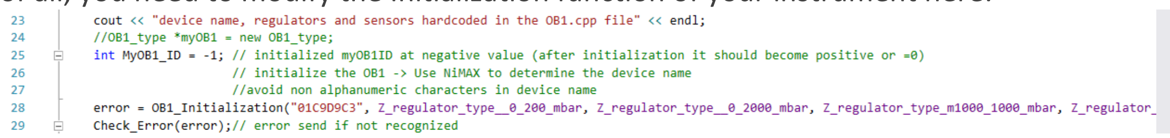
1) In your SDK folder, go to “DLL64/Example_DLL64_Visual_Cpp/ElveflowDLL” or “DLL32/Example_DLL32_Visual_Cpp/ElveflowDLL”. Open the project “ElveflowDLL.vcxproj” in visual C++. Remember that MATLAB has to run in administrator mode. In this example we will consider that visual C++ has been configured as recommended previously in this User Guide (C++ section).
2) You should obtain a window similar to this one (depending on SDK version you are running):



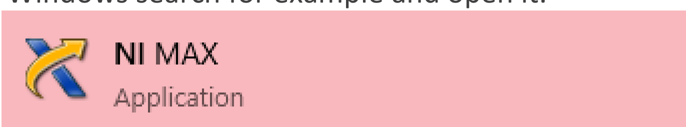
3) In the explorer on the right, search “OB1.cpp” and open it. You should obtain a code similar to this one (depending on SDK version you are running):



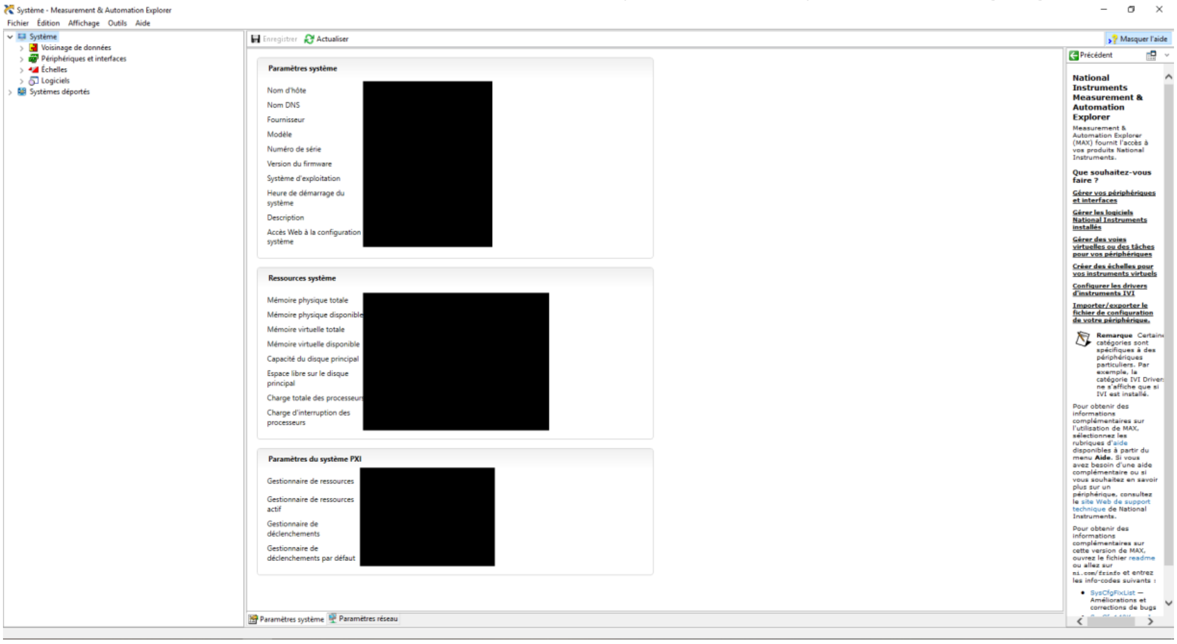
4) To make this example works you need to modify the code to adapt it to your setup. Read the comments to have more details about elements to change (for other examples and this one too). First of all, you need to modify the Initialization function of your instrument here:



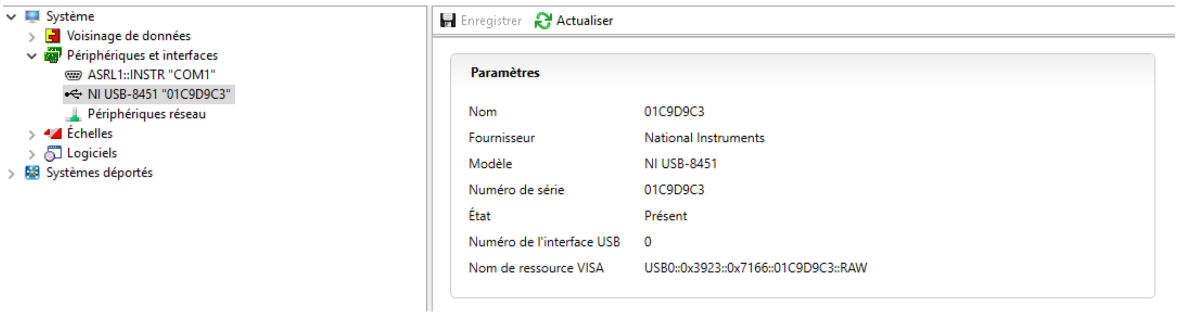
5) To modify this line (28 here) you need to open a software that has been automatically installed on your computer while installing ESI software. This is “NI MAX”. Find NI MAX on your computer by typing “NI MAX” on the Windows search for example and open it.



6) You should obtain a window similar to this one (except the black squares and language):



7) Expand the "Devices and Interfaces" tab to reveal connected instruments. Depending on the devices connected to your computer you will find multiple lines. In our case (OB1) we have to find a line with "NI USB-8451". When you found it, click on it. You should obtain a window similar to this one:



8) The name of the instrument is written in the "Name" part. Here it is "01C9D9C3" but depending on the device connected (MSR, AF1 etc...) the name could be "Dev1" for example.
Please copy the name of the instrument and go back to visual C++.

9) Write the name of the instrument instead of the already written name between "xxxxx". In the case of my instrument the modified window is as follow:

```
error = OB1_Initialization("01C9D9C3", Z_regulator_type__0_200_mbar, Z
Check_Error(error);// error send if not recognized
```

10) The second part of the Initialization function is used to define regulator type. Modify the function depending on your OB1. Considering the OB1 used for this example (described at the beginning of the section) the modified function should look like this:

```
error = OB1_Initialization("01C9D9C3", 1, 1, 4, 3, &MyOB1_ID);
Check_Error(error);// error send if not recognized
```

You can also use the defined variables, they are meant to be used but it is not used in this example to be able to display all information in one screenshot. Refer to the corresponding function in the User Guide (here it is "OB1_Initialization") and the table to know which parameters to input in your case.

11) In the case of OB1, we need to declare sensors that are connected to the OB1. If you do not have any sensor, please skip the following step to step 13. In the case of this example we need to add new lines to the code. The following lines are used to add sensors:

```
//error = OB1_Add_Sens(MyOB1_ID, 1, Z_sensor_type_Press_1_bar, Z_Sensor_digit_analog_Analog, Z_Sensor_FSD_Calib_H20, Z_D_F_S_Resolution__16Bit);//
// ! ! ! If the sensor is not recognized a pop up will indicate it)
Check_Error(error);// error send if not recognized
```

12) Uncomment the "OB1_Add_Sens" and "CheckError" functions. Depending on the sensor connected the "OB1_Add_Sens" has to be modified. In our example we need to add two sensors. The new code should be as follow:

```
error = OB1_Add_Sens(MyOB1_ID, 1, 5, 1, 0, 7);// Add digital flow sensor with H2O Calibration
// ! ! ! If the sensor is not recognized a pop up will indicate it)
Check_Error(error);// error send if not recognized
error = OB1_Add_Sens(MyOB1_ID, 3, 8, 0, 0, 7);
Check_Error(error);
```

You can also use the defined variables, they are meant to be used but it is not used in this example to be able to display all information in one screenshot. Note that the last two parameters are unused in the case of the pressure sensor used in this example. Refer to the corresponding function in the User Guide (here it is “OB1_Add_Sens”) and the table to know which parameters to input in your case.

If you have more than two sensors, add that many lines (followed by Check_Error) up to 4 (which are the four channels one the OB1 where the sensors are physically connected). Channel number is the first parameter of the function after ID.

13) Then the example is ready to be launched. You can use default calibration or perform a new one. You can also load a calibration file but remember that calibration files generated through ESI cannot be used with SDK. Only SDK generated calibration files can be loaded using SDK.
14) Please follow instructions displayed on the screen to ask for pressure, sensor data etc...

Python:

1) In your SDK folder, go to “Python_64/Example” or “Python_32/Example”. There should be a file named “_OB1_Ex_.py”. This is the example code for OB1. There is also the “Elveflow64.py” or “Elveflow32.py” located in the “Python_64” or “Python_32” folder that will be necessary. Please use the IDE you want but in this example we will use IDE Eclipse V4.5.2 + Pydev V5.0.0. This code is tested with Python 3.5.1. In this example we will consider that configuration has been done properly and we will focus on the code itself.
2) Please find below a screenshot of a part of the code (depending on SDK version you are running):

```
1  #Tested with Python 3.5.1 (IDE Eclipse V4.5.2 + Pydev V5.0.0)
2  #add python_xx and python_xx/DLL to the project path
3  # coding: utf8
4
5  import sys
6  from email.header import UTF8
7  sys.path.append('D:/dev/SDK/python_64/DLL64'.encode('utf-8')) #add the path of the library here
8  sys.path.append('D:/dev/SDK/python_64'.encode('utf-8'))#add the path of the LoadElveflow.py
9
10 from ctypes import *
11
12 from array import array
13
14 from Elveflow64 import *
15
16
17 #
18 # Initialization of OB1 ( ! ! ! REMEMBER TO USE .encode('ascii')
19 #
20 Instr_ID=c_int32()
21 print('Instrument name and regulator types hardcoded in the python script'.encode('utf-8'))
22 #see User guide to determine regulator type NI MAX to determine the instrument name
23 error=OB1_Initialization('01C9D9C3'.encode('ascii'),1,2,4,3,byref(Instr_ID))
24 # all functions will return error code to help you to debug your code, for further information see user guide
25 print('error:%d' % error)
26 print('OB1 ID: %d' % Instr_ID.value)
27
28 #add one digital flow sensor with water calibration (OB1 MK3+ only for MK3, it return error 8000)
29 #error=OB1_Add_Sens(Instr_ID, 1, 1, 1, 0, 7)
30 #print('error add digit flow sensor:%d' % error)
31
32
33 #add one analog flow sensor
34 #error=OB1_Add_Sens(Instr_ID, 2, 1, 0, 0, 7)
35 #print('error add analog flow sensor:%d' % error)
36
37
38
39 #
40 #Set the calibration type
41 #
42
43 Calib=(c_double*1000)() # always define array that way, calibration should have 1000 elements
44 repeat=True
45 while repeat==True:
```

3) To make this example works you need to modify the code to adapt it to your setup. Read the comments to have more details about elements to change (for other examples and this one too). First of all, you need to modify the paths where the dll and library are. Please modify these 2 lines:

```
7  sys.path.append('D:/dev/SDK/SDK_V3_03_00/Python_64/DLL64'.encode('utf-8')) #add the path of the library here
8  sys.path.append('D:/dev/SDK/SDK_V3_03_00/Python_64'.encode('utf-8'))#add the path of the LoadElveflow.py
```

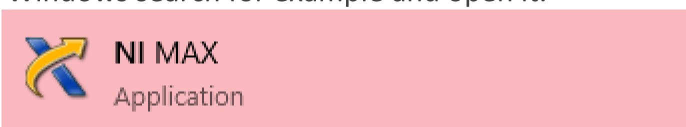
4) Then you need to open “Elveflow64.py” or “Elveflow32.py”, modify the following path and save it:

```
5  ElveflowDLL=CDLL('D:/dev/SDK/SDK_V3_03_00/Python_64/DLL64/Elveflow64.dll')# change this path
```

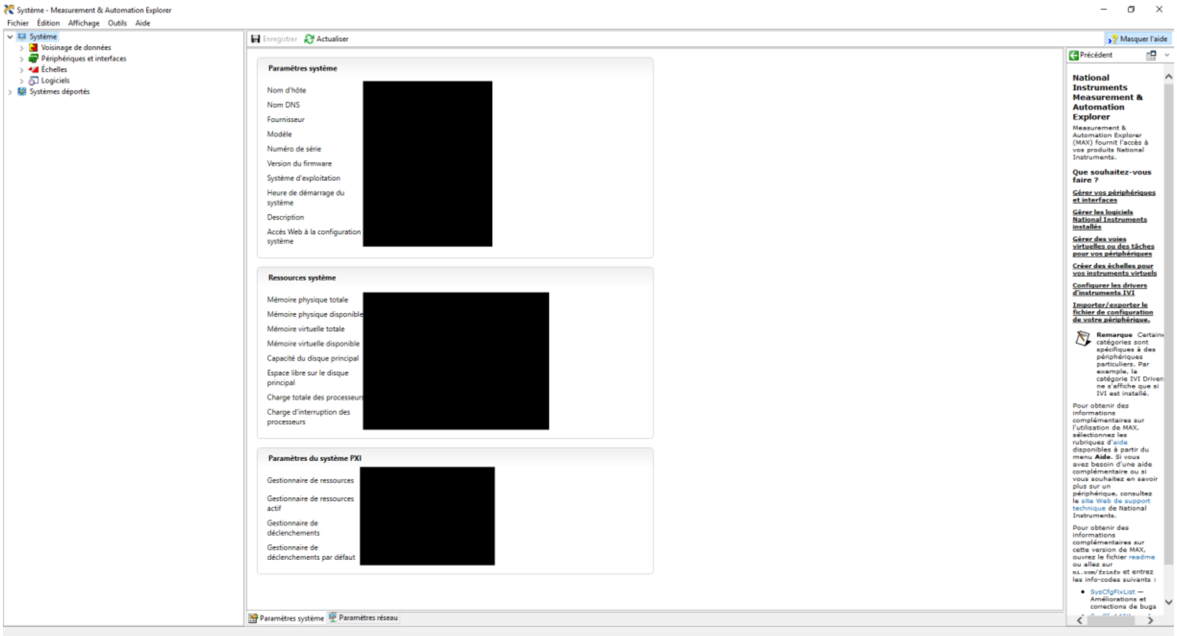
5) Go back to “_OB1_EX_.py”. Then you need to write your instrument name here:

```
20 Instr_ID=c_int32()
21 print('Instrument name and regulator types hardcoded in the python script'.encode('utf-8'))
22 #see User guide to determine regulator type NI MAX to determine the instrument name
23 error=OB1_Initialization('01C9D9C3'.encode('ascii'),1,2,4,3,byref(Instr_ID))
```

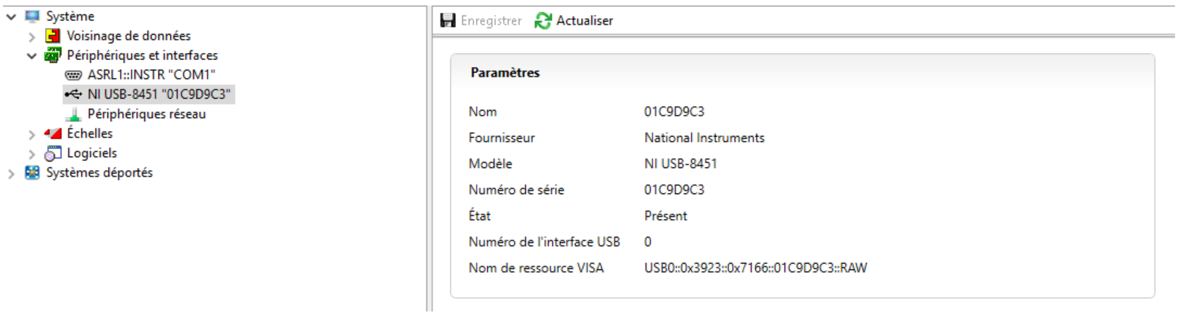
6) To modify this line (23 here) you need to open a software that has been automatically installed on your computer while installing ESI software. This is “NI MAX”. Find NI MAX on your computer by typing “NI MAX” on the Windows search for example and open it.



7) You should obtain a window similar to this one (except the black squares and language):



8) Expand the "Devices and Interfaces" tab to reveal connected instruments. Depending on the devices connected to your computer you will find multiple lines. In our case (OB1) we have to find a line with "NI USB-8451". When you found it, click on it. You should obtain a window similar to this one:



9) The name of the instrument is written in the "Name" part. Here it is "01C9D9C3" but depending on the device connected (MSR, AF1 etc...) the name could be "Dev1" for example.
Please copy the name of the instrument and go back to the example.

10) Write the name of the instrument instead of the already written name between 'xxxxx'. In the case of my instrument the modified window is as follow:

```
Instr_ID=c_int32()
print('Instrument name and regulator types hardcoded in the python script'.encode('utf-8'))
#see User guide to determine regulator type NI MAX to determine the instrument name
error=OB1_Initialization('01C9D9C3'.encode('ascii'),1,2,4,3,byref(Instr_ID))
```

11) The second part of the Initialization function is used to define regulator type. Modify the function depending on your OB1. Considering the OB1 used for this example (described at the beginning of the section) the modified function should look like this:

```
error=OB1_Initialization('01C9D9C3'.encode('ascii'),1,1,4,3,byref(Instr_ID))
```

Refer to the corresponding function in the User Guide (here it is "OB1_Initialization") and the table to know which parameters to input in your case.

12) In the case of OB1, we need to declare sensors that are connected to the OB1. If you do not have any sensor, please skip the following step to step 14. In the case of this example we need to add new lines to the code. The following lines are used to add sensors:

```
#add one digital flow sensor with water calibration (OB1 MK3+ only for MK3, it return error 8000)
#error=OB1_Add_Sens(Instr_ID, 1, 1, 1, 0, 7)
#print('error add digit flow sensor:%d' % error)


#add one analog flow sensor
#error=OB1_Add_Sens(Instr_ID, 2, 1, 0, 0, 7)
#print('error add analog flow sensor:%d' % error)
```

13) Uncomment the “OB1_Add_Sens” and “print” functions. Depending on the sensor connected the “OB1_Add_Sens” has to be modified. In our example we need to add two sensors. The new code should be as follow:

```
#add one digital flow sensor with water calibration (OB1 MK3+ only for MK3, it return error 8000)
error=OB1_Add_Sens(Instr_ID, 1, 5, 1, 0, 7)
print('error add digit flow sensor:%d' % error)
#add one analog flow sensor
error=OB1_Add_Sens(Instr_ID, 3, 8, 0, 0, 7)
print('error add analog flow sensor:%d' % error)
```

Note that the last two parameters are unused in the case of the pressure sensor used in this example. Refer to the corresponding function in the User Guide (here it is “OB1_Add_Sens”) and the table to know which parameters to input in your case.

If you have more than two sensors, add that many lines (followed by the check of the error) up to 4 (which are the four channels one the OB1 where the sensors are physically connected). Channel number is the first parameter of the function after ID.

14) Then the example is ready to be launched. You can use default calibration or perform a new one. You can also load a calibration file but remember that calibration files generated through ESI cannot be used with SDK. Only SDK generated calibration files can be loaded using SDK.

15) Please follow instructions displayed on the screen to ask for pressure, sensor data etc...

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 with AF1
-8006	No Instrument with selected ID

Other errors can be found in 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 function will return an error code that could help to debug your software.

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

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_340_mbar	7
Z_sensor _type_Press_1_bar	8
Z_sensor _type_Press_2_bar	9
Z_sensor _type_Press_7_bar	10
Z_sensor _type_Press_16_bar	11
Z_sensor _type_Level	12

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

Calibration and PID Example (required for AF1 and OB1):

```
int32_t __cdecl Elveflow_Calibration_Default(double Calib_Array_out[], int32_t len);
Set default Calib in Calib cluster, len is the Calib_Array_out array length

int32_t __cdecl Elveflow_Calibration_Load(char Path[], double Calib_Array_out[], int32_t len);
Load the calibration file located at Path and returns the calibration  parameters in the
Calib_Array_out. len is the Calib_Array_out array length.  The function asks the user to choose the
path if Path is not valid, empty  or not a path. The function indicate if the file was found.

int32_t __cdecl Elveflow_Calibration_Save(char Path[], double Calib_Array_in[], int32_t len);
Save the Calibration cluster in the file located at Path. len is the  Calib_Array_in array length.
The function prompt the user to choose the  path if Path is not valid, empty or not a path.

int32_t __cdecl Elveflow_EXAMPLE_PID(int32_t PID_ID_in, double actualValue, int32_t Reset, double P,
double I, int32_t *PID_ID_out, double *value);
This function is only provided for illustration purpose, to explain how to  do your own feedback
loop. Elveflow does not guarante 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.    In this function need to be initiate with a first call where PID_ID =-1.
The PID_out will provide the new created PID_ID. This ID should be use in  further call.   General
remarks of this PI regulator :  The error "e" is calculate 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.
```

AF1:

```
int32_t __cdecl AF1_Calib(int32_t AF1_ID_in, double Calib_array_out[], int32_t len);
Launch AF1 calibration and return the calibration array. Len correspond to  the Calib_array_out
length.

int32_t __cdecl AF1_Destructor(int32_t AF1_ID_in);
Close Communication with AF1

int32_t __cdecl AF1_Get_Flow_rate(int32_t AF1_ID_in, double *Flow);
Get the Flow rate from the flow sensor connected on the AF1

int32_t __cdecl AF1_Get_Press(int32_t AF1_ID_in, int32_t Integration_time, double Calib_array_in[],
double *Pressure, int32_t len);
Get the pressure of the AF1 device, Calibration array is required (use  Set_Default_Calib if
required). Len correspond to the Calib_array_in  length.

int32_t __cdecl AF1_Get_Trig(int32_t AF1_ID_in, int32_t *trigger);
Get the trigger of the AF1 device (0=0V, 1=5V).

int32_t __cdecl AF1_Initialization(char Device_Name[], Z_regulator_type Pressure_Regulator,
Z_sensor_type Sensor, int32_t *AF1_ID_out);
Initiate the AF1 device using device name (could be obtained in NI MAX),  and regulator, and sensor.
It return the AF1 ID (number >=0) to be used  with other function

int32_t __cdecl AF1_Set_Press(int32_t AF1_ID_in, double Pressure, double Calib_array_in[], int32_t
len);
Set the pressure of the AF1 device, Calibration array is required (use  Set_Default_Calib if
required).Len correspond to the Calib_array_in length.

int32_t __cdecl AF1_Set_Trig(int32_t AF1_ID_in, int32_t trigger);
Set the Trigger of the AF1 device (0=0V, 1=5V).
```

BFS:

```
int32_t __cdecl BFS_Destructor(int32_t BFS_ID_in);
Close Communication with BFS device

int32_t __cdecl BFS_Get_Density(int32_t BFS_ID_in, double *Density);
Get fluid density (in g/L) for the BFS defined by the BFS_ID
```

```
int32_t __cdecl BFS_Get_Flow(int32_t BFS_ID_in, double *Flow);
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 flow 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.

int32_t __cdecl BFS_Get_Mass_Flow(int32_t BFS_ID_in, double *MassFlow);

int32_t __cdecl BFS_Get_Temperature(int32_t BFS_ID_in, double *Temperature);
Get the fluid temperature (in °C) of the BFS defined by the BFS_ID

int32_t __cdecl BFS_Initialization(char Visa_COM[], int32_t *BFS_ID_out);
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 return the  BFS ID (number >=0) to be used with other
function

int32_t __cdecl BFS_Set_Filter(int32_t BFS_ID_in, double Filter_value);
Elveflow Library BFS Device  Set the instruement Filter. 0.000001= maximum filter -> slow change but
very low noise. 1= no filter-> fast change but noisy.   Default value is 0.1

int32_t __cdecl BFS_Zeroing(int32_t BFS_ID_in);
```

Flow Reader or old version Sensor Reader:

```
int32_t __cdecl F_S_R_Destructor(int32_t F_S_Reader_ID_in);
Close Communication with F_S_R.

int32_t __cdecl F_S_R_Get_Sensor_data(int32_t F_S_Reader_ID_in, int32_t Channel_1_to_4, double
*output);
Get the data from the selected channel.

int32_t __cdecl F_S_R_Initialization(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, int32_t *F_S_Reader_ID_out);
Initiate the F_S_R device using device name (could be obtained in NI MAX)  and sensors. It return
the F_S_R ID (number >=0) to be used with other  function.  NB: Flow reader can only accept Flow
sensor NB 2: Sensor connected to channel 1-2 and 3-4 should be the same type  otherwise they will
not be taken into account and the user will be informed  by a prompt message.
```

MUX Distributor:

```
int32_t __cdecl MUX_Dist_Destructor(int32_t MUX_Dist_ID_in);
Close Communication with MUX distributor device

int32_t __cdecl MUX_Dist_Get_Valve(int32_t MUX_Dist_ID_in, int32_t *selected_Valve);
Get the active valve

int32_t __cdecl MUX_Dist_Initialization(char Visa_COM[], int32_t *MUX_Dist_ID_out);
Initiate the MUX Distributor device using device com port (ASRLXXX::INSTR  where XXX is the com port
that could be found in windows device manager).  It return the MUX Distributor ID (number >=0) to be
used with other  function

int32_t __cdecl MUX_Dist_Set_Valve(int32_t MUX_Dist_ID_in, int32_t selected_Valve);
Set the active valve
```

MUX & MUX Wire:

```
int32_t __cdecl MUX_Destructor(int32_t MUX_ID_in);
Close the communication of the MUX device

int32_t __cdecl MUX_Get_Trig(int32_t MUX_ID_in, int32_t *Trigger);
Get the trigger of the MUX device (0=0V, 1=5V).

int32_t __cdecl MUX_Initialization(char Device_Name[], int32_t *MUX_ID_out);
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 function
```

```
int32_t __cdecl MUX_Set_Trig(int32_t MUX_ID_in, int32_t Trigger);
Set the Trigger of the MUX device (0=0V, 1=5V).

int32_t __cdecl MUX_Set_all_valves(int32_t MUX_ID_in, int32_t array_valve_in[], int32_t len);
Valves are set by a array of 16 element. If the valve value is equal or
 below 0, valve is close, if it's equal or above 1 the valve is open. The
 index in the array indicate 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 element nothing happened

int32_t __cdecl MUX_Set_indiv_valve(int32_t MUX_ID_in, int32_t Input, int32_t Ouput, int32_t
OpenClose);
Set the state of one valve of the instrument. The desired valve is  addressed using Input and Output
parameter which corresponds to the  fluidics inputs and outputs of the instrument.

int32_t __cdecl MUX_Wire_Set_all_valves(int32_t MUX_ID_in, int32_t array_valve_in[], int32_t len);
Valves are set by a array of 16 element. If the valve value is equal or  below 0, valve is close, if
it's equal or above 1 the valve is open. If the  array does not contain exactly 16 element nothing
happened
```

Sensor Reader able to read digital sensors (MSRD):

```
int32_t __cdecl M_S_R_D_Add_Sens(int32_t M_S_R_D_ID, int32_t Channel_1_to_4, Z_sensor_type
SensorType, Z_Sensor_digit_analog DigitalAnalog, Z_Sensor_FSD_Calib FSens_Digit_Calib,
Z_D_F_S_Resolution FSens_Digit_Resolution);
Add sensor to MSRD device. Select the channel n° (1-4) the sensor type.   For Flow sensor, the type
of communication (Analog/Digital), the  Calibration for digital version (H20 or IPA) should be
specify 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 correspondance)  For
digital version, the sensor type is automatically detected during this  function call.   For Analog
sensor, the calibration parameters is not taken into account.   If the sensor is not compatible with
the MSRD version, or no digital sensor  are detected an error will be thrown as output of the
function.  NB: Sensor type has to be the same as in the "Initialization" step.

int32_t __cdecl M_S_R_D_Destructor(int32_t M_S_R_D_ID);
Close communication with MSRD

int32_t __cdecl M_S_R_D_Get_Sens_Data(int32_t M_S_R_D_ID, uint32_t Channel_1_to_4, double
*Sens_Data);
Read the sensor of the requested channel.s Units: Flow sensor: µl/min Pressure: mbar  NB: For
Digital Flow Senor, If the connection is lost, MSRD will be reseted  and the return value will be
zero

int32_t __cdecl M_S_R_D_Initialization(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, int32_t *MSRD_ID_out);
Initialize the Sensor Reader device able to read digital sensors (MSRD)  using device name and
sensors type (see SDK Z_sensor_type for corresponding  numbers). It modify the MSRD ID (number >=0).
This ID can be used with  other function 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: Sensor connected to channel 1-2 and 3-4 have to be the same type  otherwise
they will not be taken into account.
```

OB1:

```
int32_t __cdecl OB1_Add_Sens(int32_t OB1_ID, int32_t Channel_1_to_4, Z_sensor_type SensorType,
Z_Sensor_digit_analog DigitalAnalog, Z_Sensor_FSD_Calib FSens_Digit_Calib, Z_D_F_S_Resolution
FSens_Digit_Resolution);
Add sensor to OB1 device. Select the channel n° (1-4) the sensor type.   For Flow sensor, the type
of communication (Analog/Digital), the  Calibration for digital version (H20 or IPA) should be
specify 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 correspondance)  For
```

digital version, the sensor type is automatically detected during this function call. For Analog sensor, the calibration parameters is not taken into account. If the sensor is not compatible with the OB1 version, or no digital sensor are detected an error will be thrown as output of the function.

```
int32_t __cdecl OB1_Calib(int32_t OB1_ID_in, double Calib_array_out[], int32_t len);
```
Launch OB1 calibration and return the calibration array. Before Calibration, ensure that ALL channels are proprely closed with adequate caps. Len correspond to the Calib_array_out length.

```
int32_t __cdecl OB1_Destructor(int32_t OB1_ID);
```
Close communication with OB1

```
int32_t __cdecl OB1_Get_Press(int32_t OB1_ID, int32_t Channel_1_to_4, int32_t
Acquire_Data1True0False, double Calib_array_in[], double *Pressure, int32_t Calib_Array_len);
```
Get the pressure of an OB1 channel. Calibration array is required (use Set_Default_Calib if required) and return a double . Len correspond to the Calib_array_in length. 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 other can used the values stored in memory and are almost instantaneous.

```
int32_t __cdecl OB1_Get_Sens_Data(int32_t OB1_ID, int32_t Channel_1_to_4, int32_t
Acquire_Data1True0False, double *Sens_Data);
```
Read the sensor of the requested channel. ! This Function only convert data that are 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 other can used the values stored in memory and are almost instantaneous. For Digital Sensor, that required another communication protocol, this parameter have no impact NB: For Digital Flow Senor, If the connection is lots, OB1 will be reseted and the return value will be zero

```
int32_t __cdecl OB1_Get_Trig(int32_t OB1_ID, int32_t *Trigger);
```
Get the trigger of the OB1 (0 = 0V, 1 =3,3V)

```
int32_t __cdecl OB1_Initialization(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);
```
Initialize the OB1 device using device name and regulators type (see SDK Z_regulator_type for corresponding numbers). It modify the OB1 ID (number >=0). This ID can be used be used with other function to identify the targed OB1. If an error occurs during the initialization process, the OB1 ID value will be -1.

```
int32_t __cdecl OB1_Reset_Digit_Sens(int32_t OB1_ID, int32_t Channel_1_to_4);
```

```
int32_t __cdecl OB1_Reset_Instr(int32_t OB1_ID);
```

```
int32_t __cdecl OB1_Set_All_Press(int32_t OB1_ID, double Pressure_array_in[], double
Calib_array_in[], int32_t Pressure_Array_Len, int32_t Calib_Array_Len);
```
Set the pressure of all the channel of the selected OB1. Calibration array is required (use Set_Default_Calib if required). Calib_Array_Len correspond to the Calib_array_in length. It uses an array as pressures input. Pressure_Array_Len corresponds to the the pressure input array. The first number of the array correspond to the first channel, the seconds number to the seconds channels and so on. All the number above 4 are not taken into account. If only One channel need to be set, use OB1_Set_Pressure.

```
int32_t __cdecl OB1_Set_Press(int32_t OB1_ID, int32_t Channel_1_to_4, double Pressure, double
Calib_array_in[], int32_t Calib_Array_len);
```
Set the pressure of the OB1 selected channel, Calibration array is required (use Set_Default_Calib if required). Len correspond to the Calib_array_in length.

```
int32_t __cdecl OB1_Set_Trig(int32_t OB1_ID, int32_t trigger);
```
Set the trigger of the OB1 (0 = 0V, 1 =3,3V)