ELVEFLOW an ELVESYS brand

SDK Elveflow Software Development Kit

DOCUMENT REF: UGSDK 211004

USER GUIDE

Symbols used in this document

Important information. Disregarding this information could increase the risk of damage to the equipment, or the risk of personal injuries.

Helpful information. This information will facilitate the use of the instrument and/or contribute to its optimal performance.

Additional information available on the internet or from your Elveflow representative.

READ THIS MANUAL CAREFULLY BEFORE USING THE SOFTWARE

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

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

Important SDK safety notices:

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

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

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

Table of contents

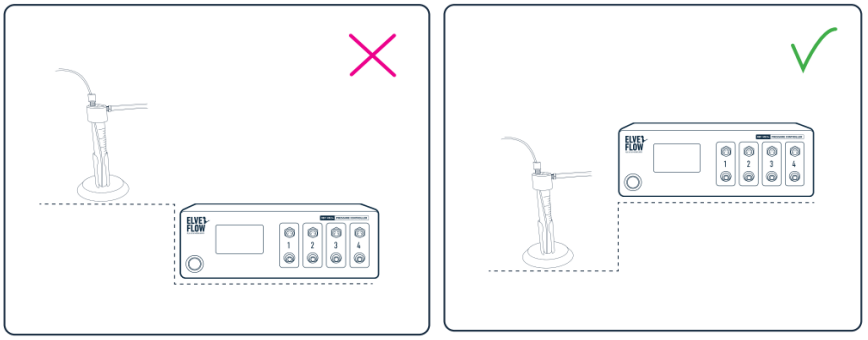
Getting started

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

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

Before starting

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

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

- **Using the SDK requires some knowledge regarding the development tools that will be used** (LabVIEW / MATLAB / Python etc ...). To ensure the best possible experience, Elveflow already provides lots of valuable information (comments in code, examples, etc...).
Feel free to read the comments and modify the example according to your specific situation.
If you still find yourself lacking answers, then we'd be happy to open a consulting request, and bring you the specific support you need.

- **The SDK user experience strongly depends on the drivers and software environment.** Depending on the computer environment where ESI or the SDK is installed, the computer might prompt some errors that are correlated to a lack of compatibility that is beyond our control. This kind of problem occurs especially when some components are already installed on the targeted computer.

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 read the comments and 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 specific support offer designed to your specific needs.

If you would like our expert advice on a particular section of your code, please be sure to give us some details about your issue.

Such as

- the SDK function you're using: e.g. OB1_init.vi
- the version of the SDK are you using: e.g. V3.06.00
- your Windows version: Windows 7 32-bit, or Windows 10 64-bit ?
- the environment you're using e.g. e.g. LabVIEW dev 2018.
- the elveflow device(s) you will use this code with (e.g. OB14000452042983 with a MUX-D-42989).
- etc... (add any relevant information you may have).

This will help minimize the turnaround time for understanding what your issue is and what the general fix would look like.

Where to find the Elveflow software development kit (SDK)

The Elveflow Software Development Kit latest stable version can be downloaded from the Elveflow website.
To alleviate bandwidth and access issues, two links for the same file are provided. The mirror link is the same file hosted on another server. In that way you always have an accessible version to work with, 24/7.
You can also find the ESI in the same zip archive file.

Do not install the SDK directly from the Zip file, and do not install directly from a USB key, this is likely to cause issues. Always copy the ESI.zip source to your computer, then unzip it before launching the installation process.

LabVIEW's SDK programming

All VI are included in ElveflowLLB.llb file.

For every instrument, we provide an example to show how to use the SDK LabVIEW.

__AF1_Example__.vi for AF1, __F_S_R_Example__.vi for Flow Reader or Sensor Reader, __MUX_D-R-I_Example__.vi for MUX Distribution/Distributor/Recirculation/Injection, __MUX_Example__.vi for other MUX series, and __OB1_Example__.vi for OB1.

OB1:

All the available VI for the programming 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, please remember to close the OB1 reference using OB1_Close.vi

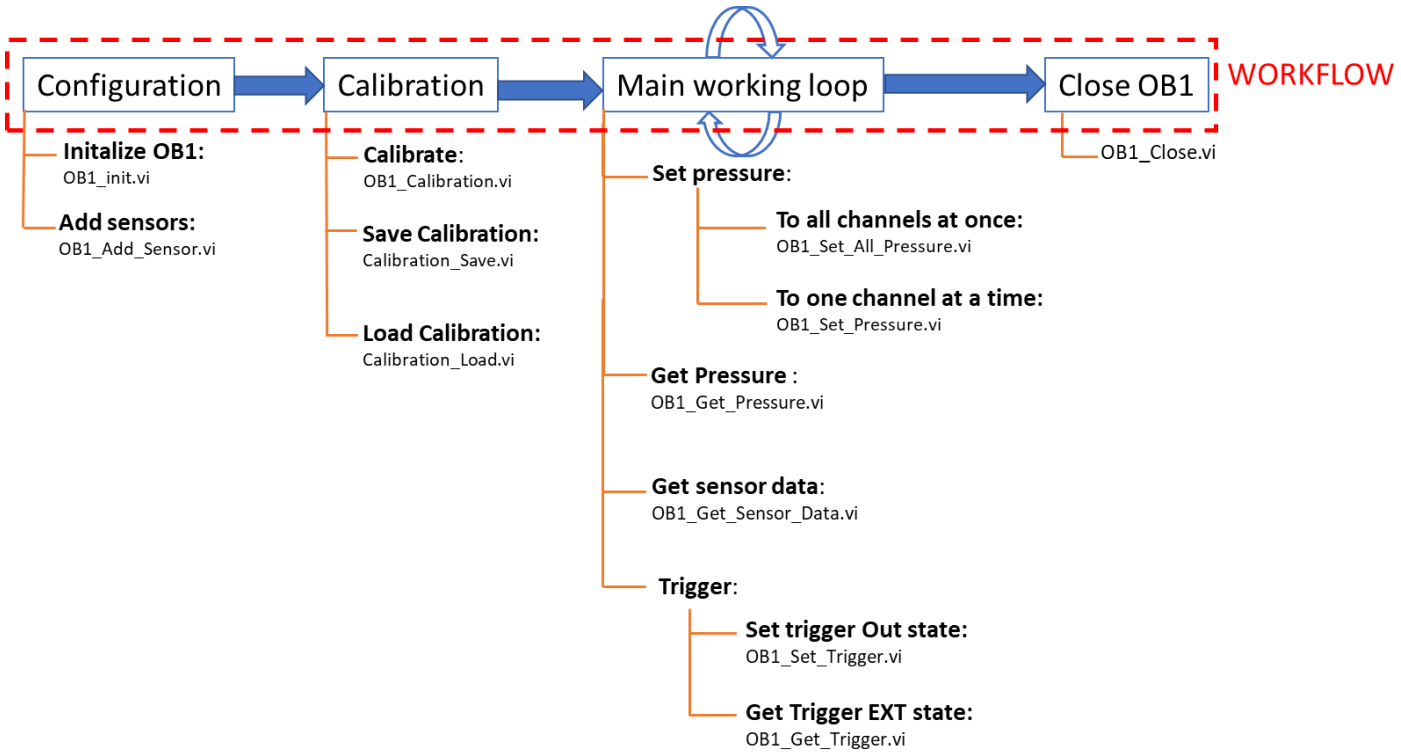


Figure 1 Typical workflow of a custom OB1 program representing the different types of the OB1 SDK VIs

New from 3.06.00: The main working loop can be simplified with a new set of functions to launch a remote control and monitoring loop. Start the loop calling OB1_Start_Remote_Measuing.vi and stop the loop calling OB1_Stop_Remote_Measuing.vi. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

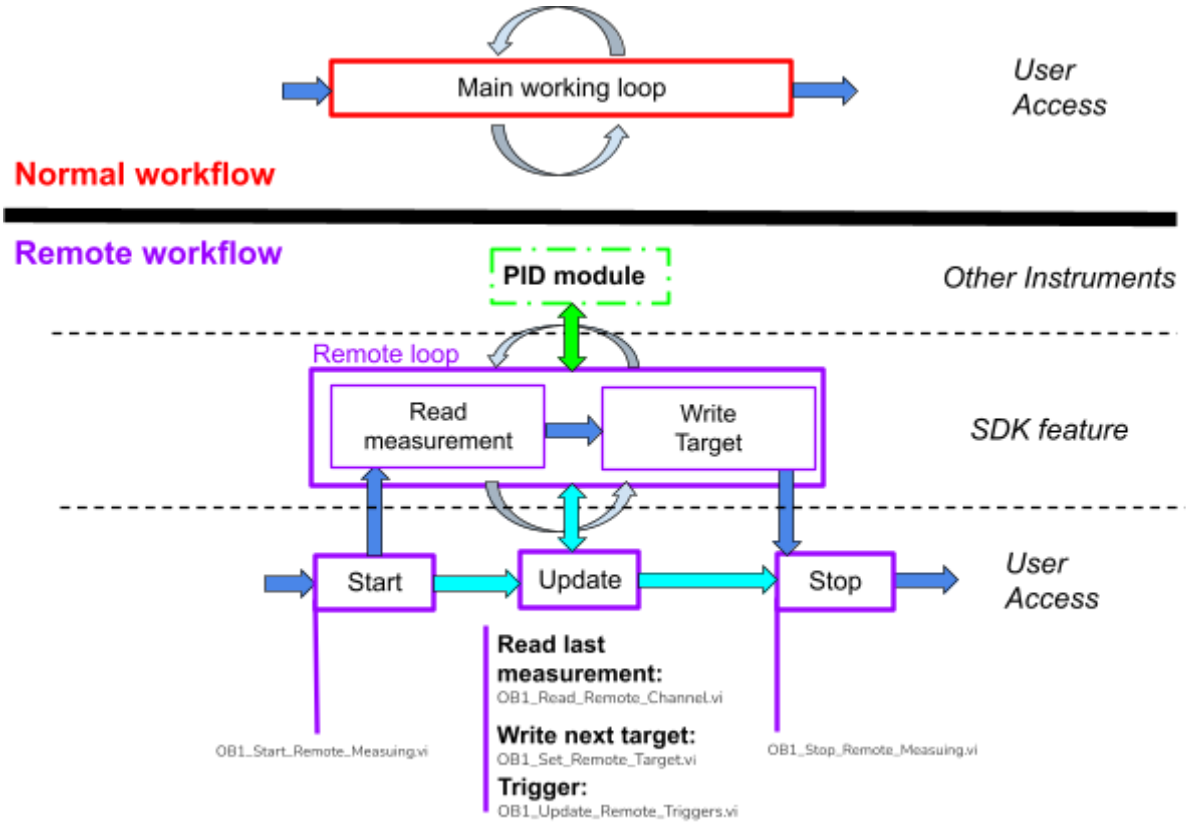
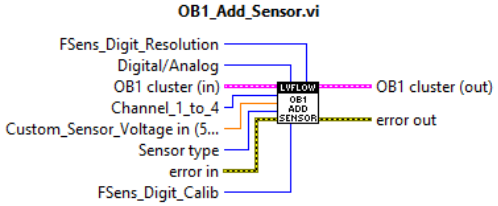
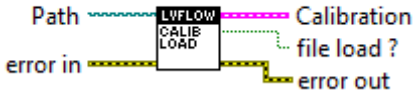
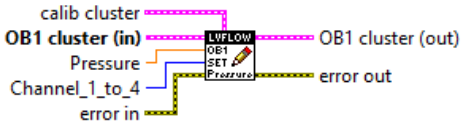
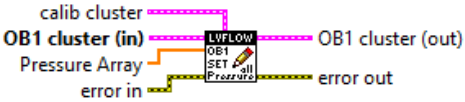
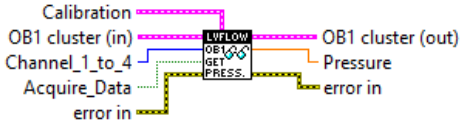
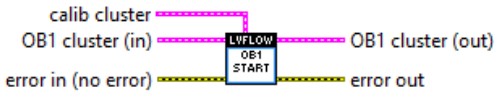
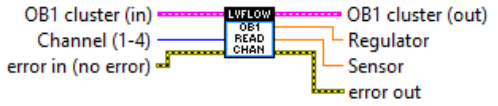
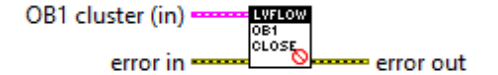
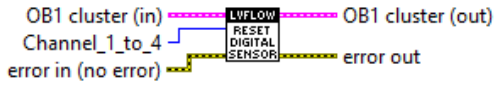


Figure 2 Difference between the normal workflow and the remote workflow

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 the 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. The Custom_Sensor_Voltage in can be set from 5 to 25V and is used to set a voltage for custom analog sensors (works only for OB1 from 2020 and after).

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 regulators and analog sensors 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 converts data acquired in these units: flow rate: µl/min, pressure: mbar "Acquire_Data" works 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, the OB1 will be reset 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)
Remote Operation		

	OB1_Start_Remote_Measuring.vi	Start a control & monitoring loop in the background, which automatically reads all sensors and regulators. No direct call to the OB1 can be made until the Stop measuring vi is called. Until then only vi accessing this loop (read channel, set target, triggers) are recommended. Only 10 OB1 can be remotely controlled at the same time
	OB1_Read_Remote_Channel.vi	Read the measured regulator and sensor values from the background control & monitoring loop
	OB1_Set_Remote_Target.vi	Change, in the running control & monitoring loop, either the regulator target or the target of the PID module if a PID loop is configured. Warning: the target units may vary. If the channel has a flow sensor **and** the PID is running in the loop, then the target will be a flow. Otherwise the target is a pressure
	OB1_Update_Remote_Triggers.vi	Set the input trigger and get the output trigger of the OB1 in the running control & monitoring loop.
	OB1_Stop_Remote_Measuring.vi	Stop the background control & monitoring loop
Close OB1 resource		
	OB1_Close.vi	Close the communication with the OB1 defined by its appropriate cluster.
Special use (advanced features)		
	OB1_Reset.vi	Warning: advanced feature. Reset OB1 communication for pressure and flow.
	OB1_Reset_Sensor.vi	Warning: advanced feature. Reset digital sensor communication from the selected channel. Select again resolution and calibration type (H2O/Isopro).

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.

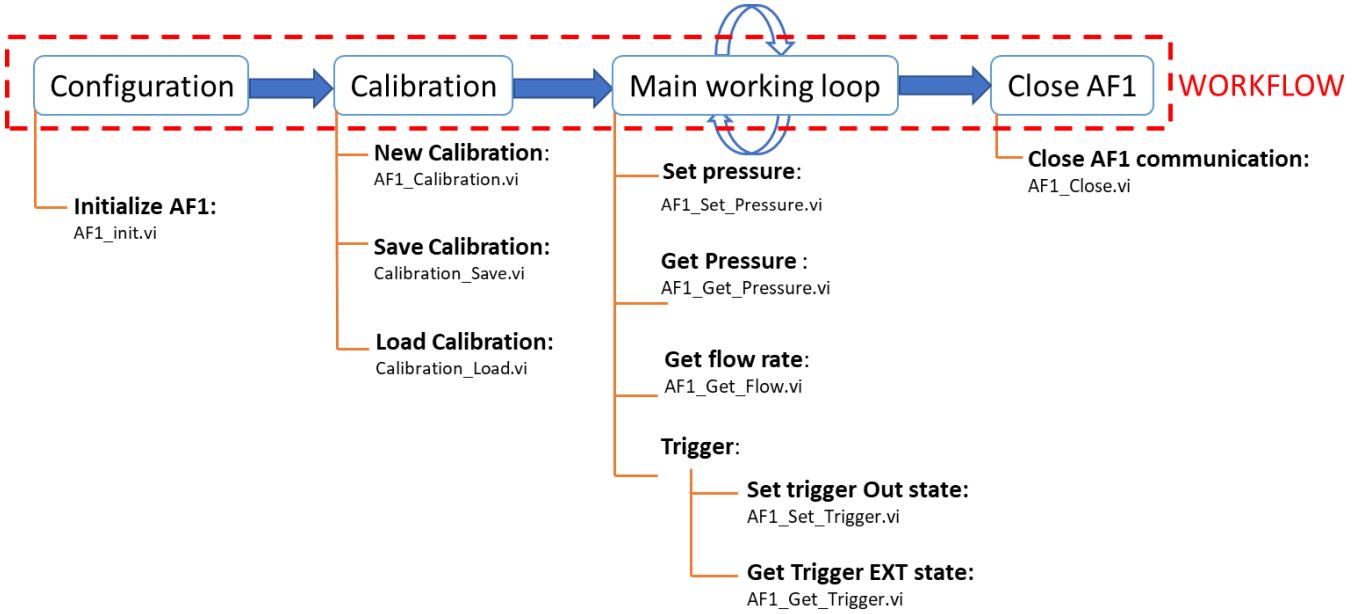
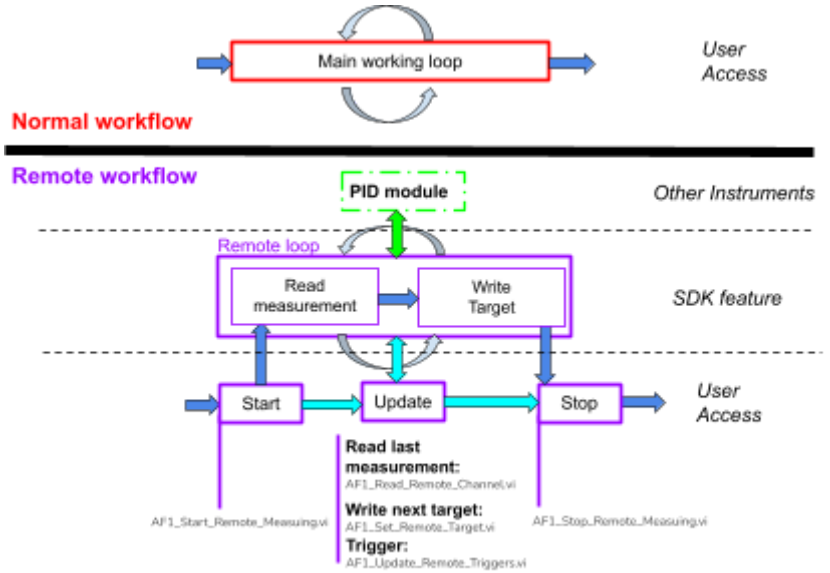
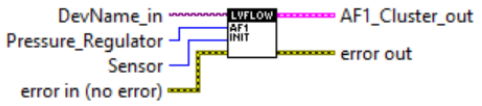
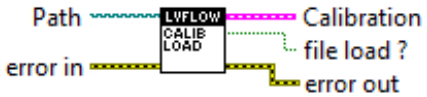
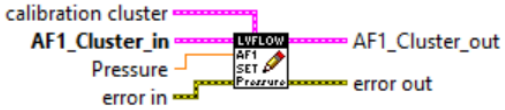


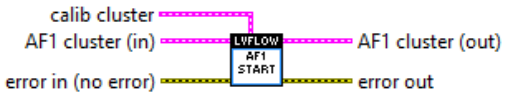
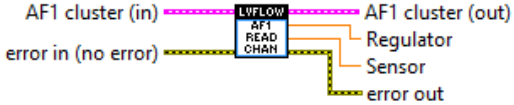
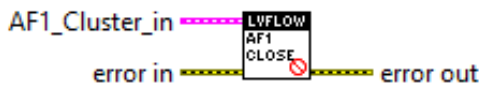
Figure 2 Typical workflow of a custom AF1 program representing the different types of the AF1 SDK functions

New from 3.06.00: The mainworking loop can be simplified with a new set of functions to launch a remote control and monitoring loop. Start the loop calling AF1_Start_Remote_Measuing.vi and stop the loop calling AF1_Stop_Remote_Measuing.vi. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.



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)
Remote Operation		
	AF1_Start_Remote_Measuring.vi	Start a control & monitoring loop in the background, which automatically reads all sensors and regulators. No direct call to the AF1 can be made until the Stop measuring vi is called. Until then only vi accessing this loop (read channel, set target, triggers) are recommended. Only 10 AF1 can be remotely controlled at the same time
	AF1_Read_Remote_Channel.vi	Read the measured regulator and sensor values from the background control & monitoring loop
	AF1_Set_Remote_Target.vi	Change, in the running control & monitoring loop, either the regulator target or the target of the PID module if a PID loop is configured. Warning: the target units may vary. If the channel has a flow sensor **and** the PID is running in the loop, then the target will be a flow. Otherwise the target is a pressure
	AF1_Update_Remote_Triggers.vi	Set the input trigger and get the output trigger of the AF1 in the running control & monitoring loop.
	AF1_Stop_Remote_Measuing.vi	Stop the background control & monitoring loop
Close OB1 resource		
	AF1_Close.vi	Close the communication with the AF1 defined by its appropriate cluster.

FSR and old version of the 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:

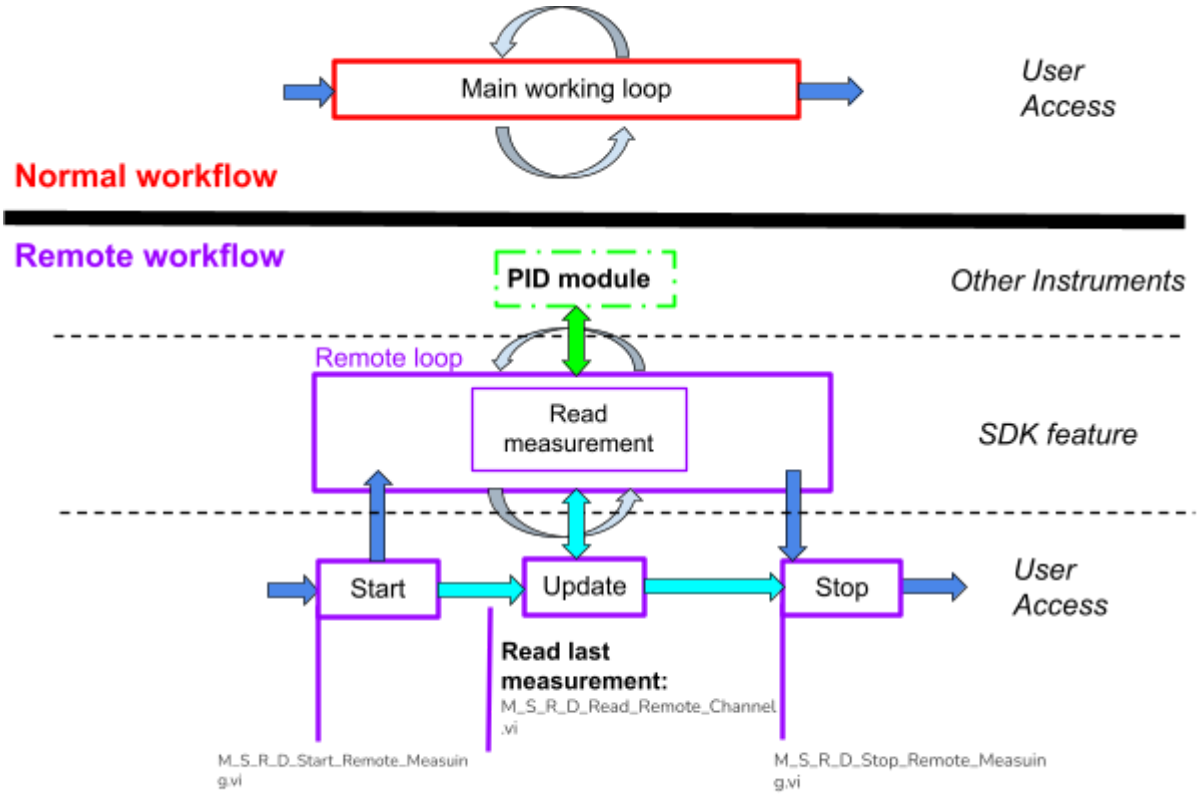
- Flow reader can only accept Flow sensors.
- Sensors connected to channel 1-2 and 3-4 should be the same type otherwise they will not be considered and the user will be informed by a prompt message.
- Sensor reader and Flow reader cannot read digital sensors.

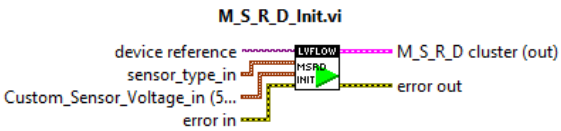
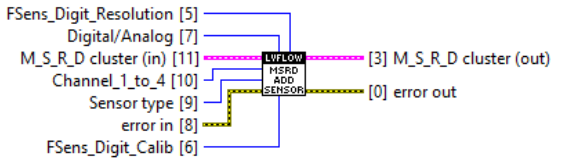
MSRD:

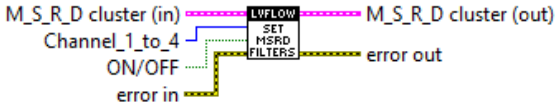
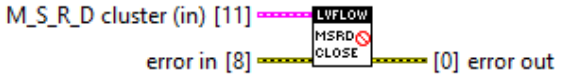
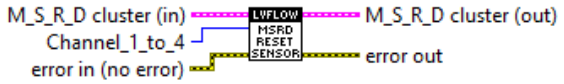
All the available vi for the programming of a customized LabVIEW program are used in the VI "_M_S_R_D_Example.vi" contained in the LLB library "ElveflowLLB.llb".

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.

New from 3.06.00: The mainworking loop can be simplified with a new set of functions to launch a remote monitoring loop. Start the loop calling M_S_R_D_Start_Remote_Measuing.vi and stop the loop calling M_S_R_D_Stop_Remote_Measuing.vi. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.



Icon	File name	Description
Configuration		
	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. The Custom_Sensor_Voltage_in can be set from 5 to 25V and is used to set a voltage for custom analog sensors.
	M_S_R_D_Add_Sensor	Add a sensor (flow or pressure) connected to the OB1. You must define the 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.
Operation		

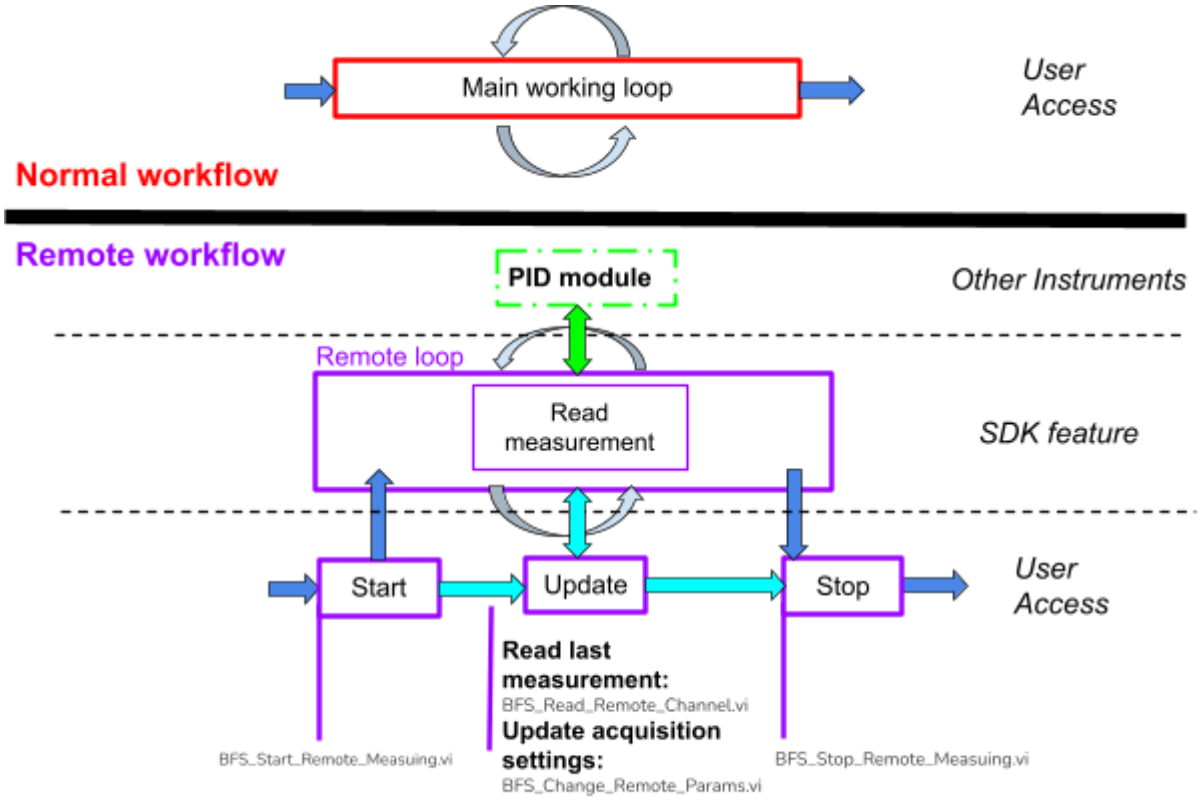
	M_S_R_D_Get_Data.vi	Read the sensor data on the requested channel with a unit of flow rate in µl/min and pressure in mbar.
	M_S_R_D_Set_Filters.vi	Set filter for the corresponding channel.
Remote Operation		
	M_S_R_D_Start_Remote_ Measuing.vi	Start a monitoring loop running in the background which automatically reads all sensors. No direct call to the MSRD can be made until the Stop measuring function is called. Until then only functions accessing this loop (read channel) are recommended. Only 10 MSRD can be remotely controlled at the same time
	M_S_R_D_Read_Remote_ Channel.vi	Read the measured sensor values from the background monitoring loop.
	M_S_R_D_Stop_Remote_ Measuing.vi	Stop the background monitoring loop
Close MSRD resource		
	M_S_R_D_Close.vi	Close the communication with the sensor reader and free the resources.
Special use (advanced features)		
	M_S_R_D_Reset.vi	Warning: advanced feature. Reset MSRD communication.
	M_S_R_D_Reset_Sensor. vi	Warning: advanced feature. Reset digital sensor communication from the selected channel. Select again resolution and calibration type (H2O/Isopro).

- 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:

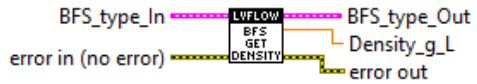
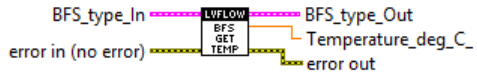
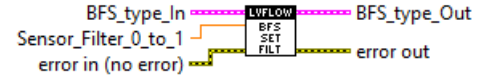
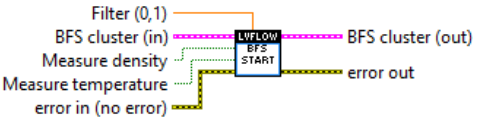
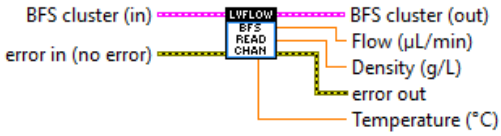
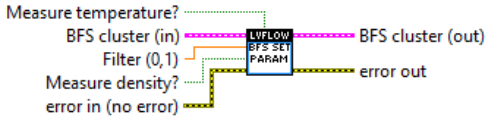
Please see the example file "_BFS_Example.vi" for a standard usage of the available BFS VI's. As with other instruments, there are three steps for programming: Initialization, instrumentation and resource liberation. Please note that for this particular sensor, in order to measure the flow rate (µL/min),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

New from 3.06.00: The mainworking loop can be simplified with a new set of functions to launch a remote monitoring loop. Start the loop calling BFS_Start_Remote_Measuing.vi and stop the loop calling BFS_Stop_Remote_Measuing.vi. You can access the device while this loop is running with the set of remote VIs. Do not access the device with non remote VIs while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.



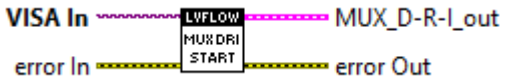
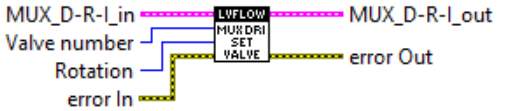
The table below gives a description of each vi.

Icon	File name	Description
Configuration		
VISA resource name, error in, LVFLOW BFS INIT, BFS_Out, error out	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.
	BFS_Do_Zeroing.vi	Perform zero calibration of the BFS. Ensure that there is no flow when performed; it is advised to use valves. The calibration procedure is finished when the green LED stops blinking.
Remote Operation		
	BFS_Start_Remote_Measuing.vi	Start a monitoring loop in the background which automatically reads all sensors. No direct call to the BFS should be made until the Stop measuring function is called. Until then only vi accessing this loop (read channel, set params) are recommended. Only 10 BFS can be remotely controlled at the same time
	BFS_Read_Remote_Channel.vi	Read the measured sensor values from the background monitoring loop, If the Measure Density ? or Measure Temperature ? parameters are set to FALSE, then the corresponding indicator will return 0.
	BFS_Change_Remote_Params.vi	Change the BFS acquisition parameters.
	BFS_Stop_Remote_Measuing.vi	Stop the background monitoring loop
Close BFS resource		
	BFS_Close.vi	Close the BFS communication and free the resource.

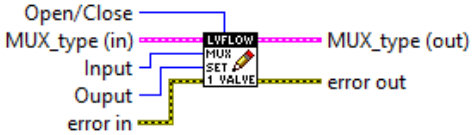
MUX D-R-I (DISTRIBUTION, DISTRIBUTOR, RECIRCULATION or INJECTION):

Please see the example file "_MUX_D-R-I_Example.vi" for a standard usage of the available MUX D-R-I VIs. The following table gives a description of the MUX D-R-I VIs.

Icon	File name	Function
	MUX_D-R-I_Init.vi	Establish connection with MUX Distribution/Distributor/Recirculation/Injection. You must input a VISA reference and select the COM port.
	MUX_D-R-I_SetValve.vi	Switch the MUX Distribution/Distributor/Recirculation/Injection to the desired valve. For MUX Distribution 12, between 1-12. For MUX Distributor (6 or 10 valves), between 1-6 or 1-10. For MUX Recirculation 6 or MUX Injection (6 valves), the two states are 1 or 2. Rotation indicates the path the valve will follow to select a valve, either shortest, clockwise or counter clockwise.
	MUX_D-R-I_GetValve.vi	Get the current valve number. If the valve is changing, function returns 0.
	MUX_D-R-I_SendCommand.vi	!This function only works for MUX Distribution 12 or Recirculation 6! Get the Serial Number or Home the valve. Remember that Home the valve takes several seconds. Home the valve is necessary as an initialization step before using the valve for a session.
	MUX_D-R-I_Close.vi	Close the communication with the MUX Distribution, Distributor, Recirculation or Injection and free the VISA resource.

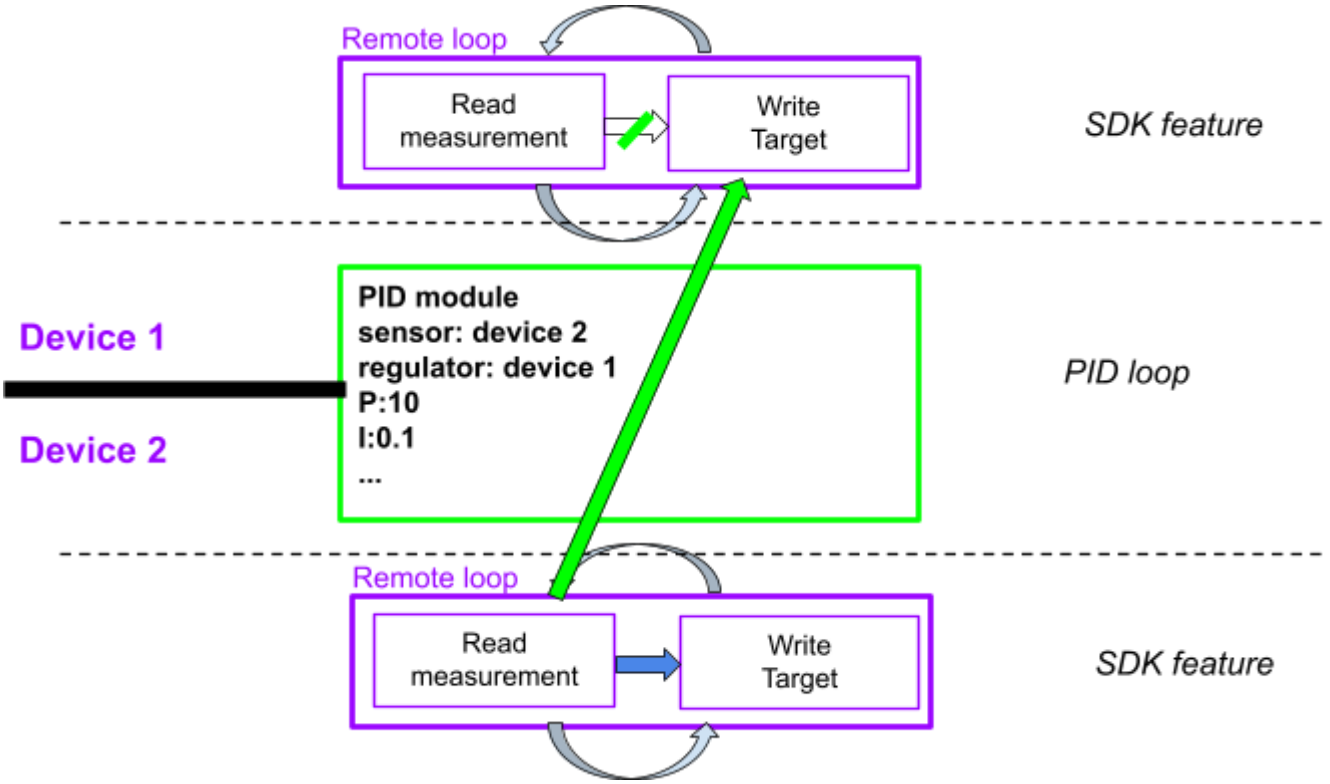
Other MUX Series:

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" illustrates the usage of all these VIs.

Instrument	Icon and VI name	Description
Common to all MUX Series	**MUX_G_Init.vi** 	Initializes the instrument using the device name and returns the identification cluster.
	MUX_G_Close.vi 	Closes the task and releases the allocated resources.
	MUX_G_Set_Trigger.vi 	Set the trigger Out (EXT) 0=>Low(0V) 1=>High(5V)
	MUX_G_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)
MUX WIRE	**MUX_G_Wire_Set_Valve_Array.vi** 	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** 	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** 	Set the state of one valve of the instrument using the Input and Output parameters. These 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.

Remote PID:

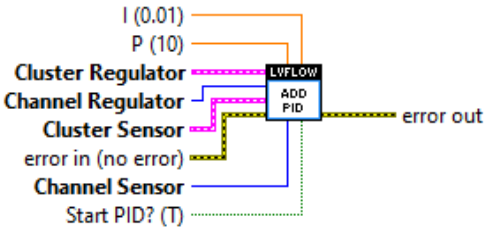
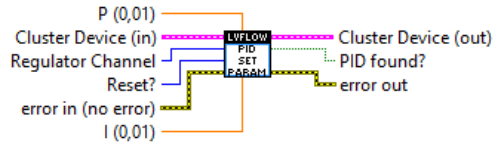
If you configure any of the compatible instruments (OB1, AF1, MSRD, BFS) in the remote mode, where the acquisition is run autonomously, you will also be able to configure, start and stop PID loops between these instruments without having to write the code itself of the loop. A fully working flow regulation can be started with an OB1 and an MFS with a single function call. Subsequent modification of the PID target is achieved in the remote loop of the device controlling the pressure/flow. A PID loop can be started on a single remote loop if the device can regulate the pressure/flow and a sensor is also connected to it.



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

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

Icon	File name	Description
Configuration		

	PID_Add.vi	Add a PID loop between a regulator and a sensor. The PID loop can later be called with the device hosting the regulator coupled with its channel (if the device has more than 1). Only works when using the remote measurement vi.
Operation		
	PID_Set_Running.vi	Adjust the running status of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote measurement vi.
	PID_Set_Params.vi	Adjust the PID parameters of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote measurement vi.

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.

- Instruments are designated using their device name. The device name can be known and changed using National Instruments Measurement and Automation Explorer (NI MAX). The NI MAX Software should be automatically installed with Elveflow Smart Interface.
- The function "Check_Error" or "CheckError" is common for all the instruments. It is used to check errors from all functions, it uses LabVIEW errors that could be checked on the internet.
- An example function that could be used for feedback control is included in all libraries as an illustration only (see the specific prototype). It is provided as an example to help you create your own regulation system. Alternatively the remote mode, available for OB1, AF1, MSRD and BFS devices from V3_05_04, enables the library to handle the regulation, for the same device or between different ones. See the OB1 example file.

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 show 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.
- 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 systems only:

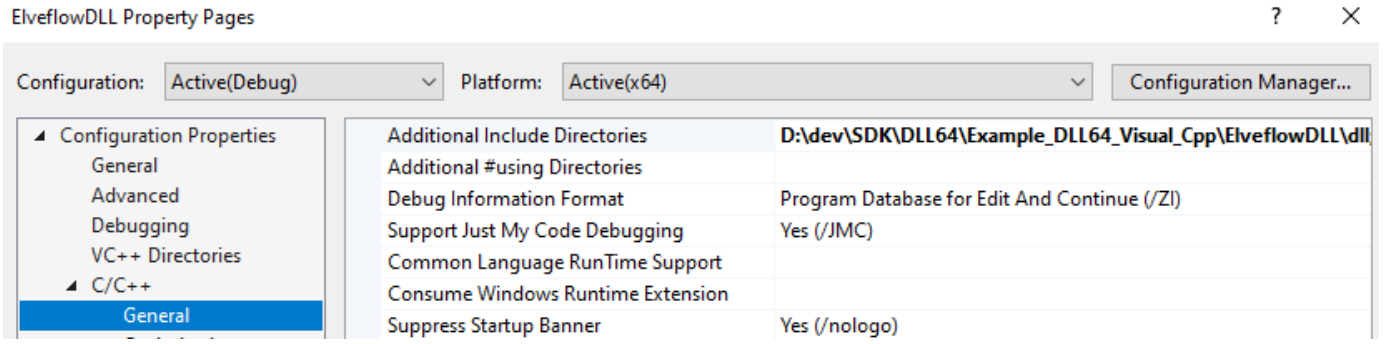
- ...\DLL64\Example_DLL64_Visual_Cpp\ElveflowDLL\x64\Debug

- ...\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.

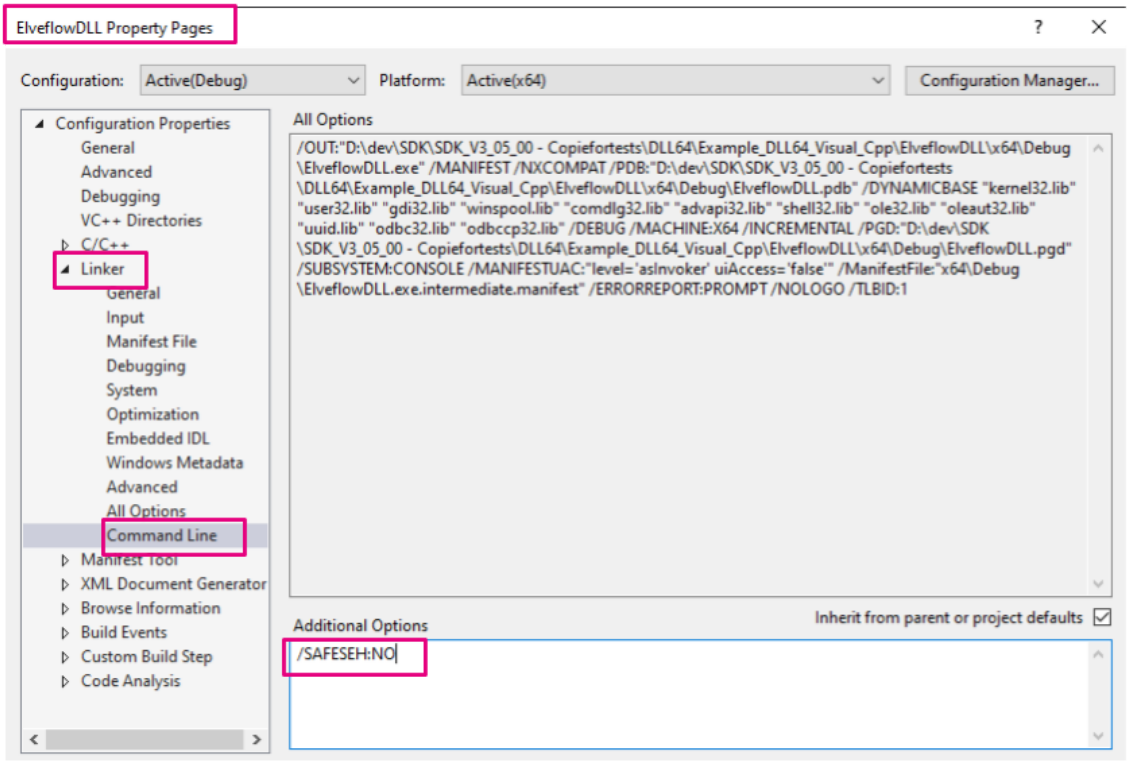
Important! Remember to add the directory that contains the dll in the additional directory (project ->

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:

- 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 are 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). This enables Python 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 you to pass the parameters by reference (i.e whenever you need to handle pointers). "byref()" is used in the instrument's 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 the next 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.

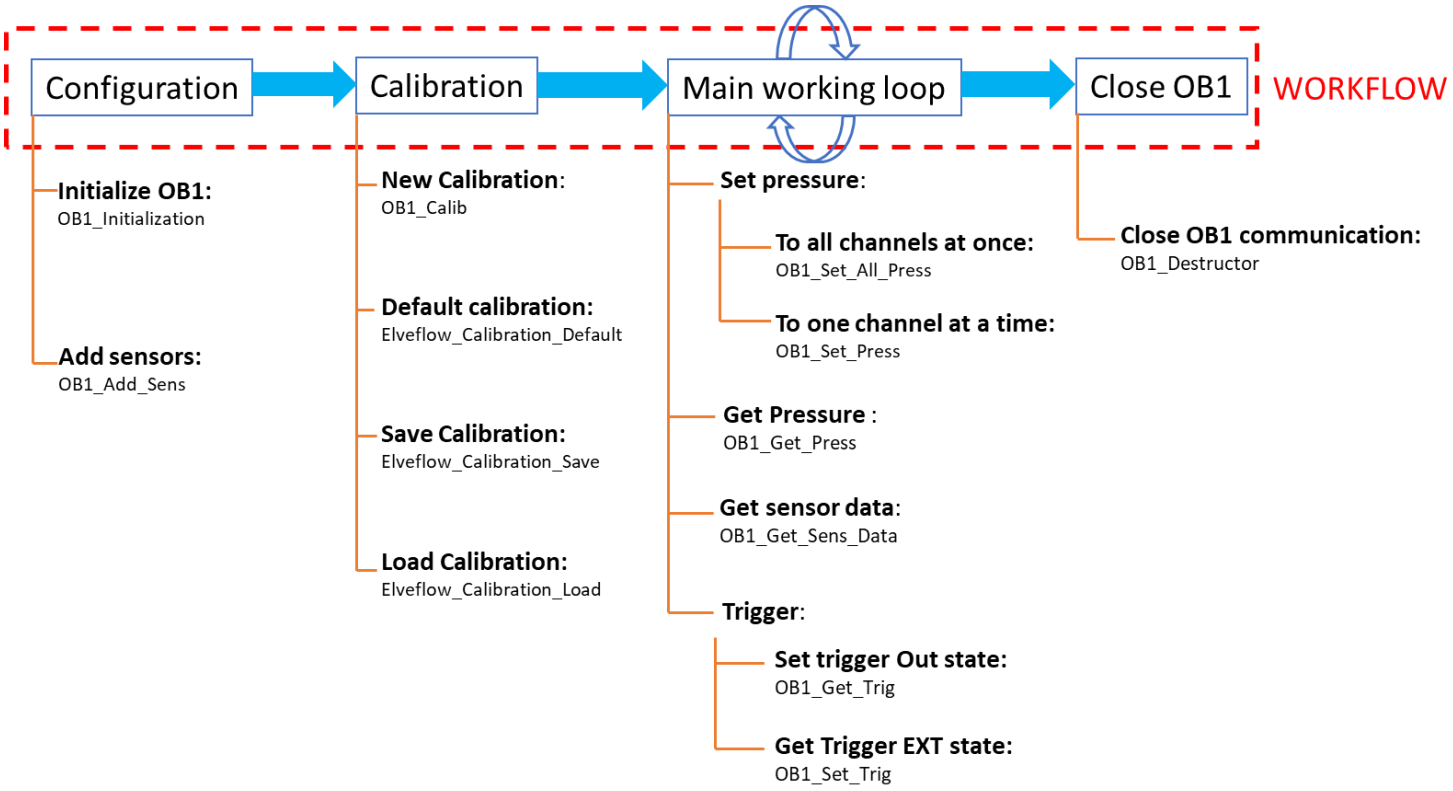


Figure 3 Typical workflow of a custom OB1 program representing the different types of the OB1 SDK functions

New from 3.05.04: The main working loop can be simplified with a new set of functions to launch a remote control and monitoring loop. Start the loop calling OB1_Start_Remote_Measurement and stop the loop calling OB1_Stop_Remote_Measurement. You can access the device while this loop is running with the set of remote functions. Do not access the device with non remote functions while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

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

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 (OB1_ID, Channel_1_to_4, SensorType, DigitalAnalog, FSens_Digit_Calib, FSens_Digit_Resolution, CustomSens_Voltage_5_to_25)	**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. - CustomSens_Voltage_5_to_25: voltage for custom sensors, values can be set from 5 to 25 V.	Add sensor to OB1 device. Select the channel n° (1-4) and the sensor type. For Flow sensors, 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. The Custom_Sensor_Voltage_5_to_25 can be set from 5 to 25V and is used to set a voltage for custom analog sensors (works only for OB1 from 2020 and after).
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_Aray_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	Load the calibration file located at Path and return the calibration parameters in the Calib_Array_out.

	- len: length of the calibration array (use default value = 1000) **Output**: - Calib_Array_out: Calibration array to be loaded (use a pointer)	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) - Calib_Array_len: length of the calibration array (use default value = 1000)	Set the pressure of the OB1 selected channel, Calibration array and length are required.
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 corresponds to the first channel, the seconds number to the seconds channels and so on. All the numbers above 4 are not taken into account - 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 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 reset 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 converts data 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 reset 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)
Remote Operation		
OB1_Start_Remote_Measurement (OB1_ID, Calib_array_in, Calib_Array_len)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - Calib_Array_in: Calibration array to be saved (pointer) - Calib_Array_len: length of the calibration array (use default value = 1000)	Start a control & monitoring loop in the background, which automatically reads all sensors and regulators. No direct call to the OB1 can be made until the Stop measuring function is called. Until then only function accessing this loop (read channel, set target, triggers) are recommended.
OB1_Stop_Remote_Measurement (OB1_ID)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value)	Stop the background control & monitoring loop
OB1_Remote_Triggers (OB1_ID,TriggerIn,TriggerOut)	**Inputs**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value) - TriggerIn: trigger state (High or Low) **Output**: - TriggerOut : Read trigger state (High or Low)	Set the input trigger and get the output trigger of the OB1 in the running control & monitoring loop.
OB1_Get_Remote_Data (OB1_ID,Channel_1_to_4,Pressure, 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)	Read the measured regulator and sensor values from the background control & monitoring loop

	Output: - Pressure: pointer to read pressure. Changed value "Pressure.Value" - Sens_Data: Read value (pointer) stored in "Sens_Data.Value".	
OB1_Set_Remote_Target (OB1_ID, Channel_1_to_4,Target)	**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) - Target : Target pressure in mbar	Change, in the running control & monitoring loop, either the regulator target or the target of the PID module if a PID loop is configured. Warning: the target units may vary. If the channel has a flow sensor **and** the PID is running in the loop, then the target will be a flow. Otherwise the target is a pressure
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.
Special use (advanced feature)		
OB1_Reset_Instr(OB1_ID)	**Input**: - OB1_ID: OB1 ID number created by OB1_Initialization (stored in OB1_ID_out.Value)	Warning: advanced feature. Reset OB1 communication for pressure and flow.
OB1_Reset_Digit_Sens(OB1_ID, Channel_1_to_4)	**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)	Warning: advanced feature. Reset digital sensor communication from the selected channel. Select again resolution and calibration type (H2O/Isopro).

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.

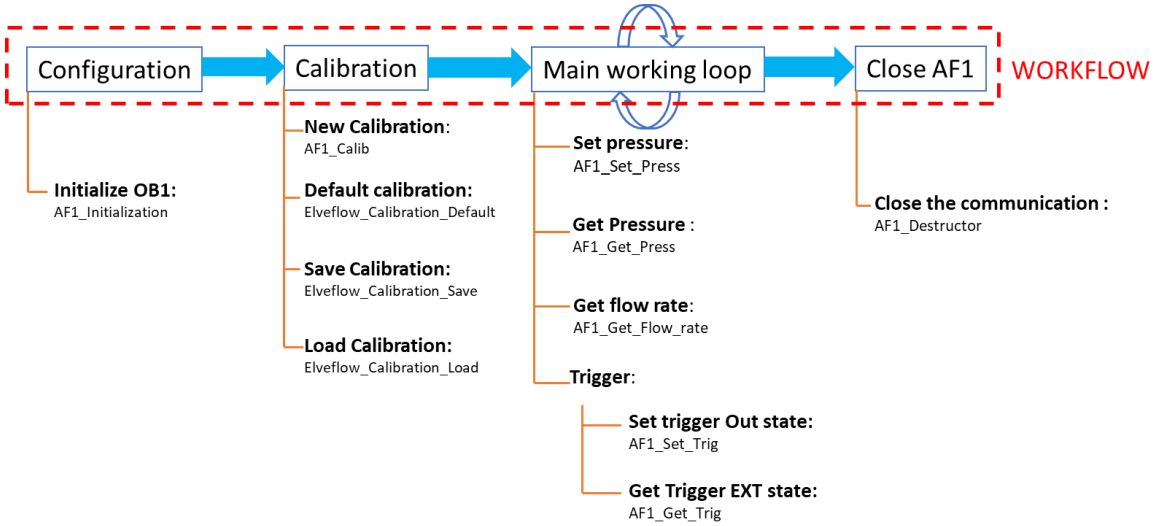


Figure 4 Typical workflow of a custom AF1 program representing the different types of the AF1 SDK functions

New from 3.06.00: The mainworking loop can be simplified with a new set of functions to launch a remote control and monitoring loop. Start the loop calling AF1_Start_Remote_Measurement and stop the loop calling AF1_Stop_Remote_Measurement. You can access the device while this loop is running with the set of remote functions. Do not access the device with non remote functions while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

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

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 sensors to be used. This function returns an identification number of the AF1 to be used 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 it 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)
Remote Operation		
AF1_Start_Remote_Measurement (AF1_ID, Calib_array_in, Calib_Array_len)	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) - Calib_Array_in: Calibration array to be saved (pointer) - Calib_Array_len: length of the calibration array (use default value = 1000)	Start a control & monitoring loop in the background, which automatically reads the sensor and regulator. No direct call to the AF1 can be made until the Stop measuring function is called. Until then only functions accessing this loop (read channel, set target, triggers) are recommended.
AF1_Stop_Remote_Measurement (AF1_ID)	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value)	Stop the background control & monitoring loop

AF1_Remote_Triggers (AF1_ID, TriggerIn, TriggerOut)	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) - TriggerIn: trigger state (High or Low) **Output**: - TriggerOut : Read trigger state (High or Low). Read value (pointer) with TriggerOut .Value	Set the input trigger and get the output trigger of the AF1 in the running control & monitoring loop.
AF1_Get_Remote_Data (AF1_ID, Pressure, Sens_Data)	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) **Output**: - Pressure: pointer to read pressure. Changed value "Pressure.Value" - Sens_Data: Read value (pointer) stored in "Sens_Data.Value".	Read the measured regulator and sensor values from the background control & monitoring loop
AF1_Set_Remote_Target (AF1_ID, Target)	**Inputs**: - AF1_ID: AF1 ID number created by AF1_Initialization (stored in AF1_ID_out.Value) - Target : Target in mbar or µL/min	Change, in the running control & monitoring loop, either the regulator target or the target of the PID module if a PID loop is configured. Warning: the target units may vary. If the channel has a flow sensor **and** the PID is running in the loop, then the target will be a flow. Otherwise the target is a pressure
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 of the 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.

- A Flow reader can only be used with a Flow sensor
- 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 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.

New from 3.06.00: The mainworking loop can be simplified with a new set of functions to launch a remote monitoring loop. Start the loop calling M_S_R_D_Start_Remote_Measurement and stop the loop calling M_S_R_D_Stop_Remote_Measurement. You can access the device while this loop is running with the set of remote functions. Do not access the device with non remote functions while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

Function / File name	Inputs/outputs	Description
Configuration		
M_S_R_D_Initialization(Device_Name, Sens_Ch_1, Sens_Ch_2, Sens_Ch_3, Sens_Ch_4, CustomSens_Voltage_Ch12, CustomSens_Voltage_Ch34, 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) - CustomSens_Voltage_Ch12 (or Ch34): voltage for custom sensors, values can be set from 5 to 25 V (the voltage is set for Channels 1-2 at the same value, and it is the same for Channels 3-4). **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. The Custom_Sensor_Voltage can be set from 5 to 25V and is used to set a voltage for custom analog sensors (unused in other cases).
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.	Add sensor to MSRD device. Select the channel n° (1-4) and the sensor type (same as for Initialization). For Flow sensors, 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.

	- FSens_Digit_Resolution: see values Z_D_F_S_Resolution in the table.	
Operation		
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_Set_Filt(M_S_R_D_ID, Channel_1_to_4, ONOFF);	**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). - ONOFF: true or false (1 or 0) to activate (true) or deactivate (false) filter on the corresponding channel	This is an additional feature to activate or deactivate analog filters on the corresponding channel.
Remote Operation		
M_S_R_D_Start_Remote_Measurement (M_S_R_D_ID)	**Inputs**: - M_S_R_D_ID: M_S_R_D ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value)	Start a monitoring loop in the background, which automatically reads all sensors. No direct call to the M_S_R_D can be made until the Stop measuring function is called. Until then only functions accessing this loop (get_remote_data) are recommended.
M_S_R_D_Stop_Remote_Measurement (M_S_R_D_ID)	**Inputs**: - M_S_R_D_ID: M_S_R_D ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value)	Stop the background monitoring loop
M_S_R_D_Get_Remote_Data (M_S_R_D_ID,Channel_1_to_4, Sens_Data)	**Inputs**: - M_S_R_D_ID: M_S_R_D ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value) - Channel_1_to_4: Channel number (1 to 4) **Output**:	Read the sensor value from the background monitoring loop

	- Sens_Data: Read value (pointer) stored in "Sens_Data.Value".	
Close MSRD resource		
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.
Special use (advanced feature)		
M_S_R_D_Reset_Instr(M_S_R_D_ID)	**Input**: - M_S_R_D_ID: M_S_R_D ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value)	Warning: advanced feature. Reset M_S_R_D communication for pressure and flow.
M_S_R_D_Reset_Digit_Sens(M_S_R_D_ID, Channel_1_to_4)	**Inputs**: - M_S_R_D_ID: M_S_R_D ID number created by M_S_R_D_Initialization (stored in M_S_R_D_ID_out.Value) - Channel_1_to_4: Channel number (1 to 4)	Warning: advanced feature. Reset digital sensor communication from the selected channel. Select again resolution and calibration type (H2O/Isopro).

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.

New from 3.05.04: The mainworking loop can be simplified with a new set of functions to launch a remote monitoring loop. Start the loop calling BFS_Start_Remote_Measurement and stop the loop calling BFS_Stop_Remote_Measurement. You can access the device while this loop is running with the set of remote functions. Do not access the device with non remote functions while you are still in remote mode. This remote feature also allows users to easily configure PID loops between devices.

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 :** - Density: density in g/L. Read value (pointer) stored in Density.value	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. Read value (pointer) stored in Flow.value	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. Read value (pointer) stored in Temperature.value	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.
BFS_Zeroing(BFS_ID_in)	**Inputs**: - BFS_ID_in: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value)	Perform zero calibration of the BFS. Ensure that there is no flow when performed; it is advised to use valves. The calibration procedure is finished when the green LED stops blinking.
Remote Operation		
BFS_Start_Remote_Measurement (BFS_ID)	**Inputs**: - BFS_ID: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value)	Start a monitoring loop in the background, which automatically reads the sensor value. No direct call to the BFS can be made until the Stop measuring function is called. Until then only functions accessing this loop (get_remote_data, set_remote_params) are recommended. Note: This function starts by calling BFS_Get_Density.
BFS_Stop_Remote_Measurement (BFS_ID)	**Inputs**: - BFS_ID: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value)	Stop the background monitoring loop
BFS_Set_Remote_Params(BFS_ID, Filter, Mtemp, Mdens)	**Inputs**: - BFS_ID: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) - Filter: your filter value. - Mtemp: boolean, measures the temperature in the remote loop - Mdens: boolean, measures the density in the remote loop	Adjusts the BFS filter, and updates if the background monitoring loop measures, in addition to reading the flow, the temperatures and/or the density. For these two booleans, zero means False and non-zero values indicate True.
BFS_Get_Remote_Data (BFS_ID,Temperature, Density, Sens_Data)	**Inputs**: - BFS_ID: BFS ID number created by BFS_Initialization (stored in BFS_ID_out.Value) **Outputs:** - Temperature: temperature in °C. Read value (pointer) stored in Temperature.value	Read the sensor values from the BFS device. If the sensor is not configured to be read, the corresponding entry will return 0

	- Density: density in g/L. Read value (pointer) stored in Density.value - Sens_data: flow rate in µL/min.Read value (pointer) stored in Sens_data.value	
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 D-R-I (DISTRIBUTION, DISTRIBUTOR, RECIRCULATION or INJECTION):

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

Function / File name	Inputs/outputs	Description
MUX_DRI_Initialization (Visa_COM[], MUX_DRI_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_DRI_ID_out: MUX_DRI ID number	Initiate the MUX Distribution, Distributor, Recirculation or Injection device using device COM port (ASRLXXX::INSTR where XXX is usually the COM port that could be found in Windows device manager). It returns the MUX D-R-I ID (number >=0) to be used with other functions.
MUX_DRI_Set_Valve (MUX_DRI_ID_in, selected_Valve, Rotation)	**Inputs**: - MUX_DRI_ID_in: MUX_DRI ID number created by DRI_Initialization (stored in MUX_DRI_ID_in.Value). - Selected_Valve: desired valve. - Rotation: see Z_MUX_DRI_Rotation table.	Switch the MUX Distribution, Distributor, Recirculation or Injection to the desired valve. For MUX Distribution 12, between 1-12. For MUX Distributor (6 or 10 valves), between 1-6 or 1-10. For MUX Recirculation 6 or MUX Injection (6 valves), the two states are 1 or 2. Rotation indicates the path the valve will perform to select a valve, either shortest 0, clockwise 1 or counterclockwise 2.
MUX_DRI_Get_Valve (MUX_DRI_ID_in, selected_Valve)	**Inputs**: - MUX_DRI_ID_in: MUX_DRI ID number created by DRI_Initialization (stored in MUX_DRI_ID_in.Value). **Output**: - Selected_Valve: Current valve.	Get the current valve number. If the valve is changing, function returns 0.
MUX_DRI_Send_Command (MUX_DRI_ID_in, Action, Answer[], len);	**Inputs**: - MUX_DRI_ID_in: MUX_DRI ID number created by DRI_Initialization (stored in MUX_DRI_ID_in.Value). - Action: see Z_MUX_DRI_Action table. - len: length of the Answer (40 is enough). **Output**: - Answer: Answer of the command (either Home or Serial Number).	!This function only works for MUX Distribution 12 or Recirculation 6! Get the Serial Number or Home the valve. len is the length of the Answer. Remember that Home the valve takes several seconds. Home the valve is necessary as an initialization step before using the valve for a session.
MUX_DRI_Destructor (MUX_DRI_ID_in)	**Inputs**: - MUX_DRI_ID_in: MUX_DRI ID number created by DRI_Initialization (stored in MUX_DRI_ID_in.Value).	Close Communication with MUX Distribution, Distributor, Recirculation or Injection device.

Other MUX Series:

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

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 to 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 to 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, the valve is closed, if it's equal to or above 1 the valve is open. If the array does not contain exactly 16 elements, nothing 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 to or below 0, the valve is closed, if it's equal to or above 1 the valve is open. The index in the array 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 elements 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 **Outputs**: - Output: choice of the output valve - OpenClose: set valve 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.

Remote PID:

If you configure any of the compatible instruments (OB1, AF1, MSRD, BFS) in remote mode, then you will also be able to configure, start and stop PID loops between these instruments without having to build your while loop and worry about timing.

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

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

Icon	File name	Description
	Configuration	
	PID_Add_Remote(Regulator_ID, Regulator_Channel_1_to_4, ID_Sensor, Sensor_Channel_1_to_4, P, I, Running);	**Inputs**: - Regulator_ID: Regulator ID - Regulator_Channel_1_to_4: Regulator Channel - ID_Sensor : Sensor ID - Sensor_Channel_1_to_4 : Sensor Channel - P : Proportional parameter - I : Integral parameter - Running : Run (1) or stop (0) the PID	Add a PID loop between a regulator and a sensor. The PID loop can later be called with the device hosting the regulator coupled with its channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved
	Operation	
	PID_Set_Running_Remote(Regulator_ID, Channel_1_to_4, Running);	**Inputs**: - Regulator_ID: Regulator ID - Channel_1_to_4: Regulator Channel - Running : Run (1) or stop (0) the PID	Adjust the running status of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved.

PID_Set_Params_Remote(Regulator_ID, Channel_1_to_4,Reset, P, I);	**Inputs**: - Regulator_ID: Regulator ID - Channel_1_to_4: Regulator Channel - Reset : "1" to reset "error value" - P : Proportional parameter - I : Integral parameter	Adjust the PID parameters of a PID loop. The PID loop is chosen based on the input device hosting the regulator coupled with the regulator channel (if the device has more than 1). Only works when using the remote mode for the device(s) involved.

Quick start examples:

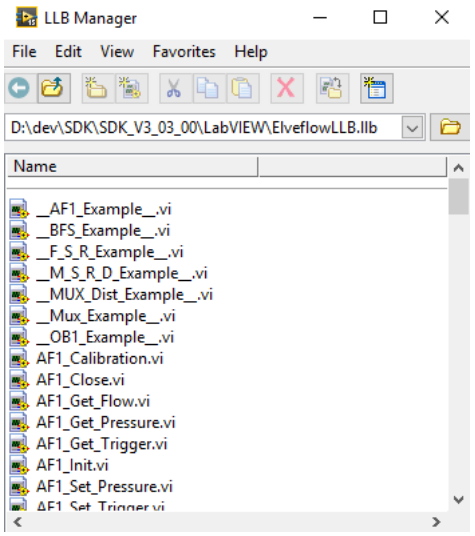
This section is here to guide you on how to use and modify the examples in each language. First of all, unzip the SDK file to have your uncompressed SDK folder.

All explanations described here are only for OB1 examples, 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.

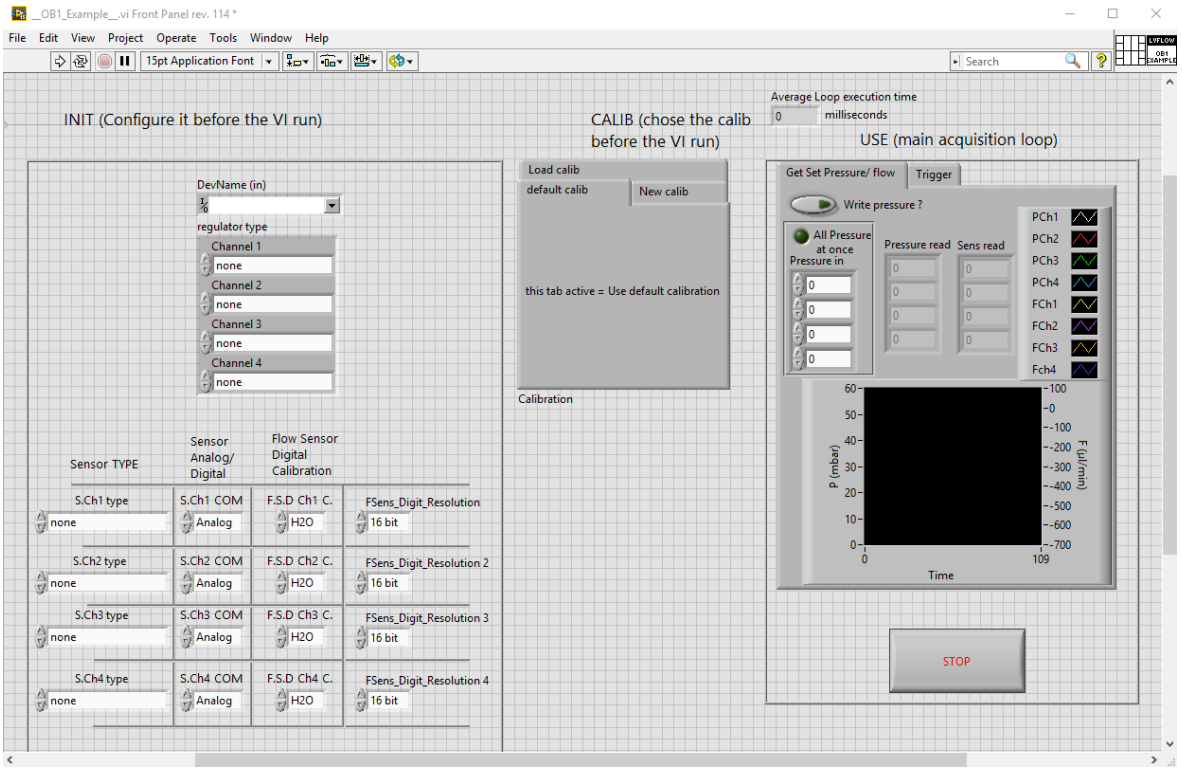
LabVIEW:

Quick start

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 warnings appear, ignore them. You should now have the following window opened:

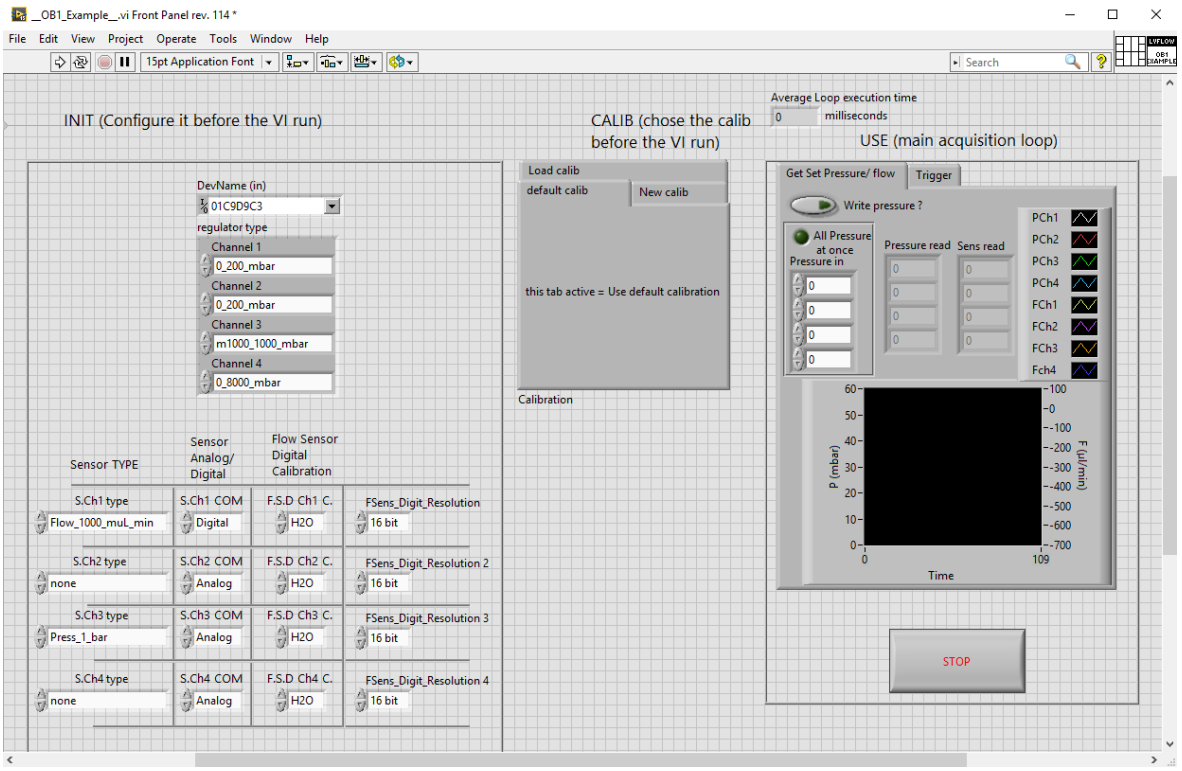


5) To test the example, be sure that your OB1 is connected to the computer and turned on. Connect also all the flow sensors you want to use.

6) Considering the OB1 used you need to modify the following elements prior to 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)

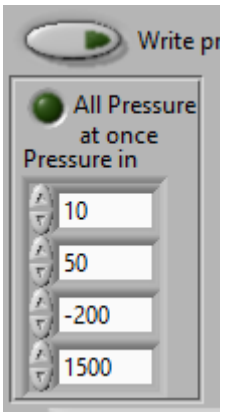
Considering the OB1 used for this example (described at the beginning of the section) the modified VI should be configured as follows:



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:

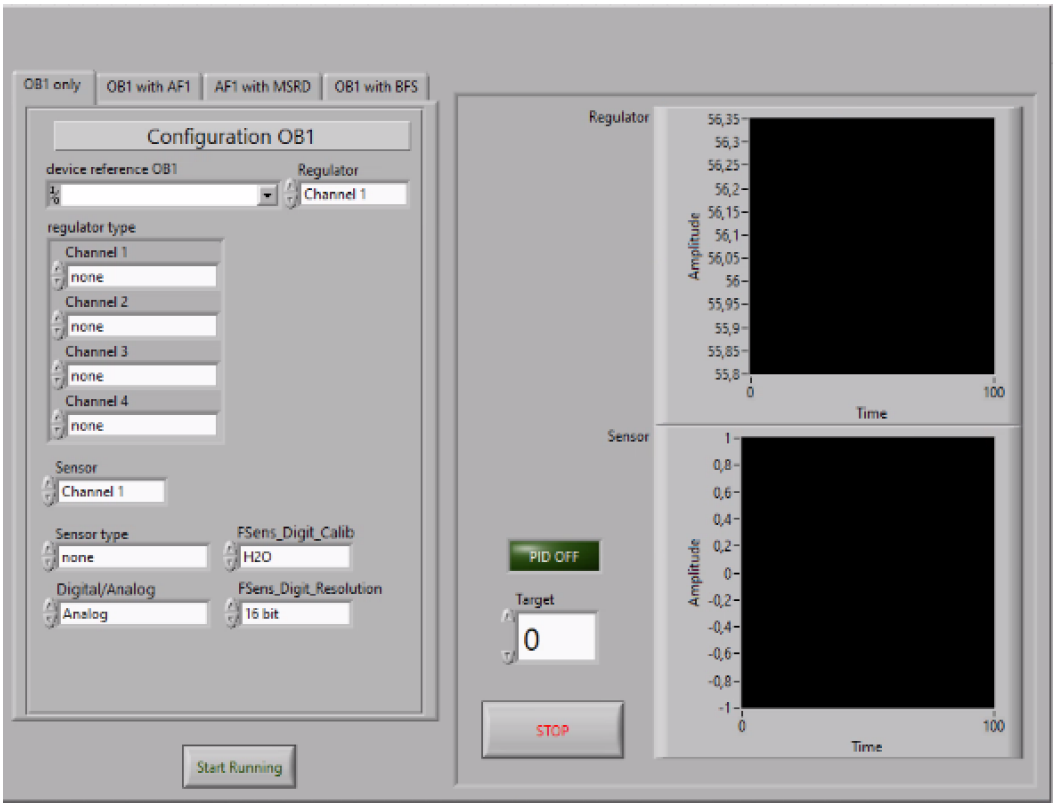


And click on “Write pressure?” button to write and unclick it.

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

Remote mode

10) Go back to the llb menu. Now we will open __Remote_PID_Examples__.vi



11) Fill in the left panel the information about your OB1 regulator and sensor you want to configure in a PID loop, then press 'Start Running' at the bottom.

12) You can now activate the PID with the green indicator at the center. Change the Target to observe the behavior of the loop.

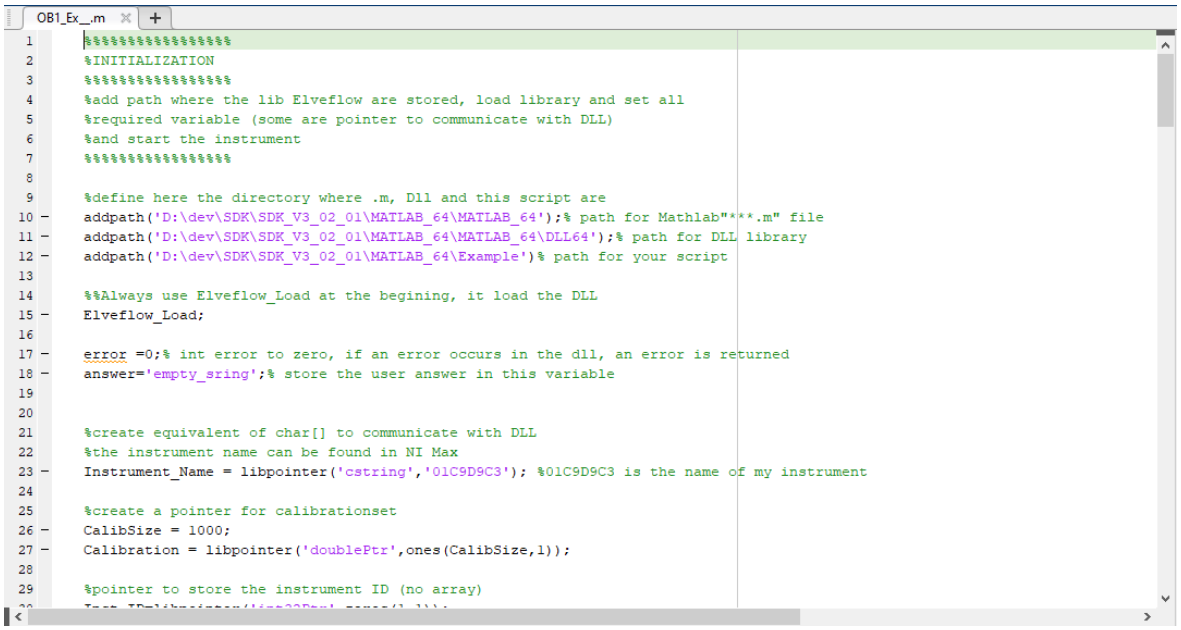
13) Once satisfied with the process, you can press 'stop' and look at the code itself.

MATLAB:

Quick start

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 the 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 the SDK version you are running):



3) To make this example work 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:

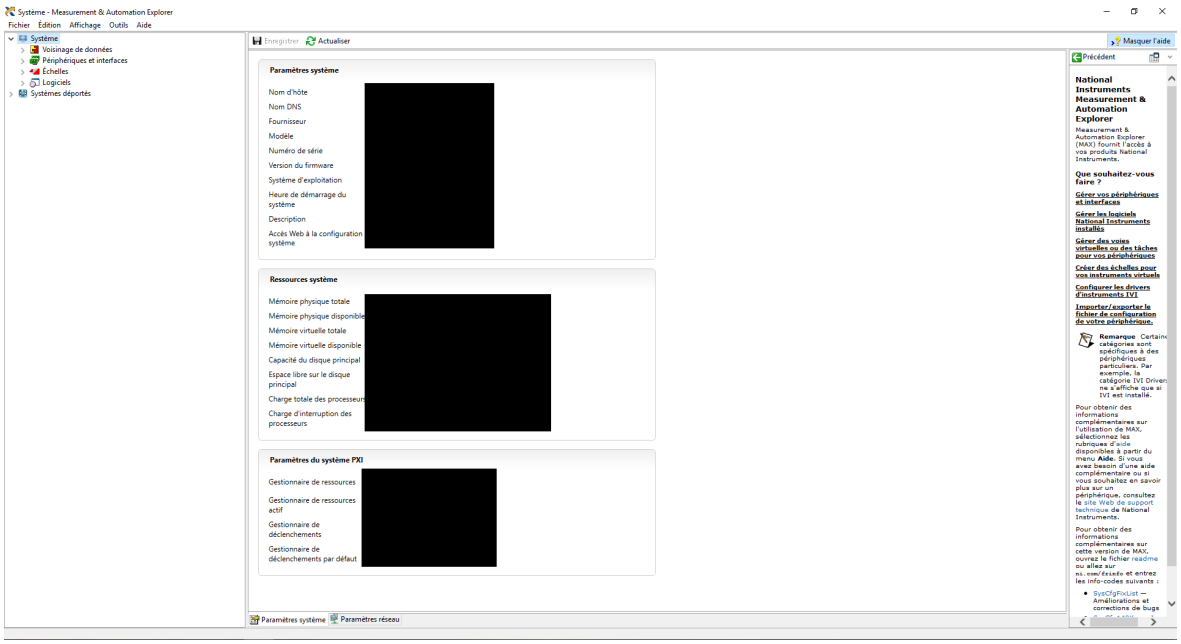
```
9       %define here the directory where .m, Dll and this script are
10 -    addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\MATLAB_64');% path for Mathlab"***.m" file
11 -    addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\MATLAB_64\DLL64');% path for DLL library
12 -    addpath('D:\dev\SDK\SDK_V3_03_00\MATLAB_64\Example')%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

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

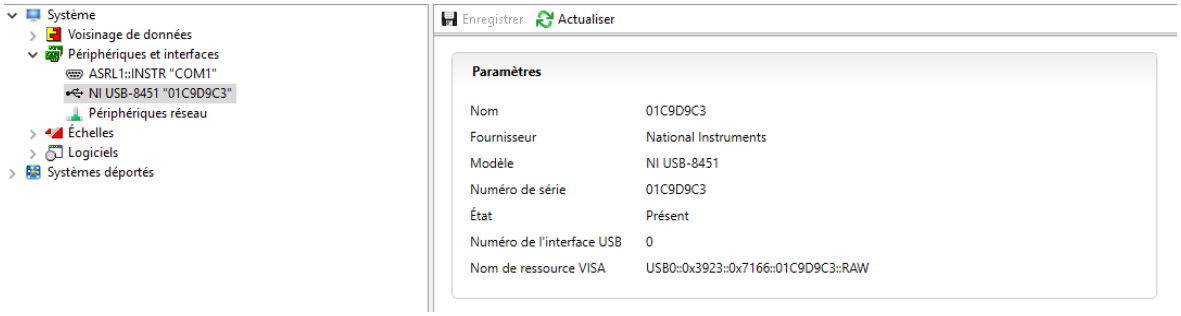
```
21      %create equivalent of char[] to communicate with DLL
22      %the instrument name can be found in NI Max
23 -    Instrument_Name = libpointer('cstring','01C9D9C3'); %01C9D9C3 is the name of my instrument
```

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 find 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 this instrument the modified window is as follows:

```
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
```

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

```
32      %Initiate the device and all regulators and sensors types (see user
33      %guide for help
34 -    error=OB1_Initialization(Instrument_Name,1,2,4,3,Inst_ID);
35 -    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:

```
32      %Initiate the device and all regulators and sensors types (see user
33      %guide for help
34 -    error=OB1_Initialization(Instrument_Name,1,1,4,3,Inst_ID);
35 -    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:

```
37    %add digital flow sensor. Valid for OB1 MK3+ only, if sensor not detected it will throw an error ;
38    %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
39    %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 follows:

```
38 -    error=OB1_Add_Sens(Inst_ID.Value,1,5,1,0,7);
39 -    CheckError(error);
40 -    error=OB1_Add_Sens(Inst_ID.Value,3,8,0,0,7);
41 -    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.

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...

Remote mode

16) While the example is running and asking for a new command, select ‘start’ which will run the function “**OB1_Start_Remote_Measurement**”. The OB1 is now in remote mode, reading data in an asynchronous loop.

17) To check that the remote loop is operating correctly, now select ‘read_channel’ which will call the function “**OB1_Get_Remote_Data**” and will return the latest measured value from the OB1.

18) You can now immediately start a PID loop by calling ‘add_PID’ which will call the “add_PID” function. By calling read_channel again, you should see the regulator (control) is now following the sensor.

19) Update your new target for the PID loop by simply calling the ‘set_target’ at the channel regulating the PID loop, in this case channel 1, which will call the function “**OB1_Set_Remote_Target**”.

20) You can confirm the PID loop is updated by reading the channel again with ‘read_channel’

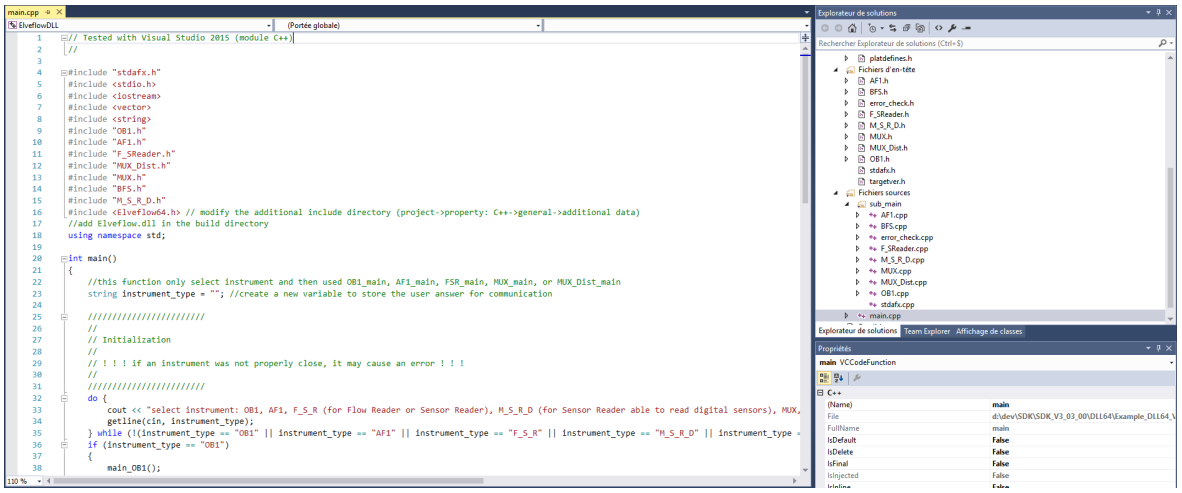
21) To exit the program, first stop the remote mode by selecting ‘stop’ which will call the function “**OB1_Stop_Remote_Measurement**”.

C++:

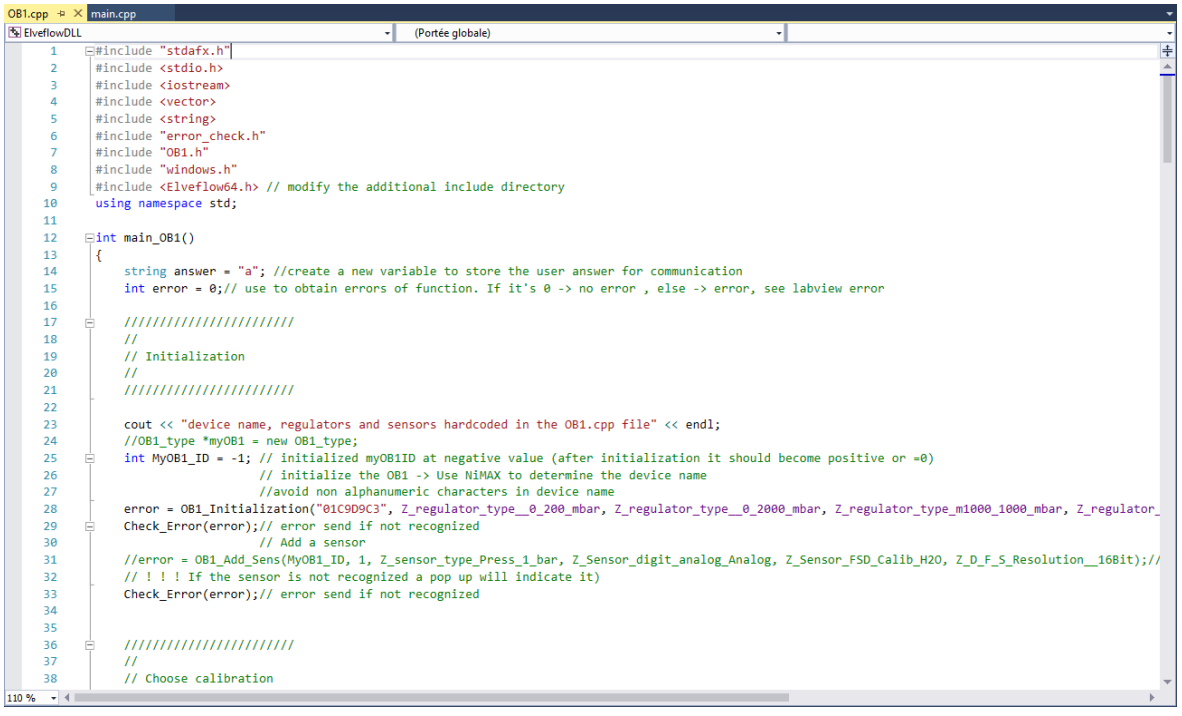
Quick start

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 the 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 the SDK version you are running):

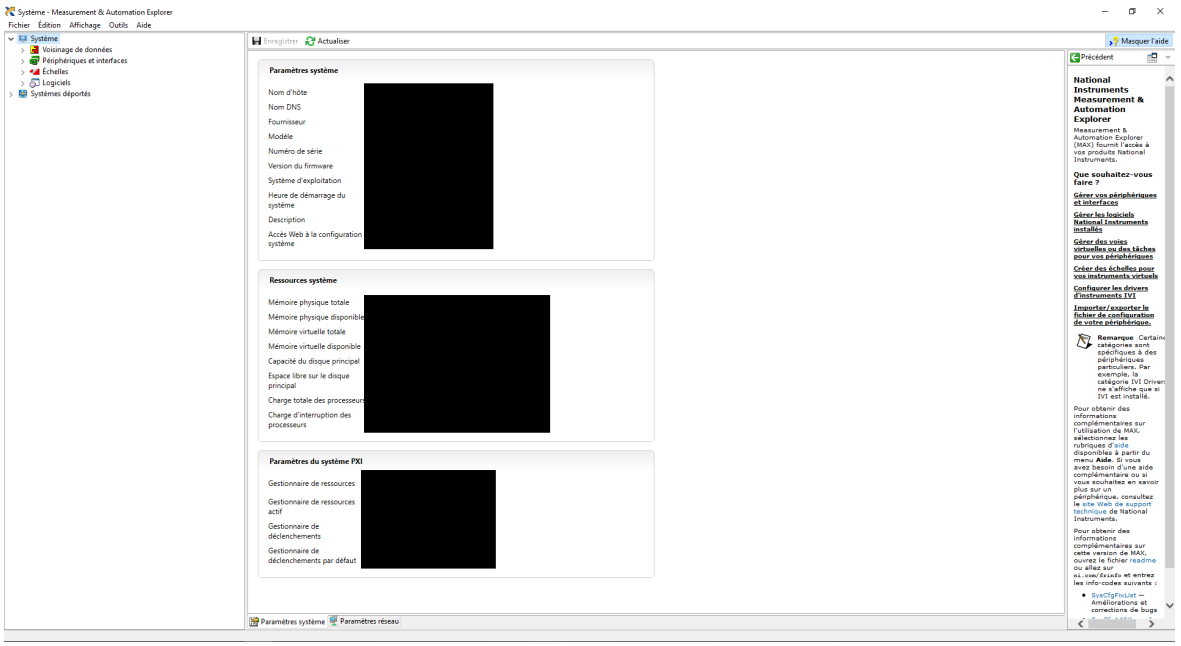


4) To make this example work you need to modify the code to adapt it to your setup. Read the comments to have more details about elements to change (for other examples and this one too). First of all, you need to modify the Initialization function of your instrument here:

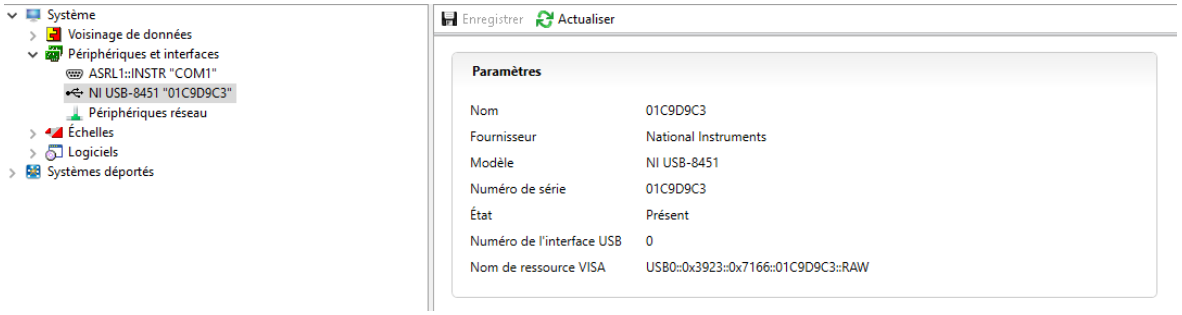
```
23      cout << "device name, regulators and sensors hardcoded in the OB1.cpp file" << endl;
24      //OB1_type *myOB1 = new OB1_type;
25      int MyOB1_ID = -1; // initialized myOB1ID at negative value (after initialization it should become positive or =0)
26                         // initialize the OB1 -> Use NiMAX to determine the device name
27                         //avoid non alphanumeric characters in device name
28      error = OB1_Initialization("01C9D9C3", Z_regulator_type__0_200_mbar, Z_regulator_type__0_2000_mbar, Z_regulator_type_m1000_1000_mbar, Z_regulator_
29      Check_Error(error);// error send if not recognized
```

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 find 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 this instrument the modified window is as follows:

```
28      error = OB1_Initialization("01C9D9C3", Z_regulator_type__0_200_mbar, Z
29      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:

```
28      error = OB1_Initialization("01C9D9C3", 1, 1, 4, 3, &MyOB1_ID);
29      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 the 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:

```
31  //error = OB1_Add_Sens(MyOB1_ID, 1, Z_sensor_type_Press_1_bar, Z_Sensor_digit_analog_Analog, Z_Sensor_FSD_Calib_H2O, Z_D_F_S_Resolution__16Bit);//
32  // ! ! ! If the sensor is not recognized a pop up will indicate it)
33  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 follows:

```
31      error = OB1_Add_Sens(MyOB1_ID, 1, 5, 1, 0, 7);// Add digital flow sensor with H2O Calibration
32      // ! ! ! If the sensor is not recognized a pop up will indicate it)
33      Check_Error(error);// error send if not recognized
34      error = OB1_Add_Sens(MyOB1_ID, 3, 8, 0, 0, 7);
35      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...

Remote mode

15) While the example is running and asking for a new command, select 'start' which will run the function "**OB1_Start_Remote_Measurement**". The OB1 is now in remote mode, reading data in an asynchronous loop.

16) To check that the remote loop is operating correctly, now select 'read_channel' which will call the function "**OB1_Get_Remote_Data**" and will return the latest measured value from the OB1.

17) You can now immediately start a PID loop by calling 'add_PID' which will call the "add_PID" function. By calling read_channel again, you should see the regulator (control) is now following the sensor.

18) Update your new target for the PID loop by simply calling the 'set_target' at the channel regulating the PID loop, in this case channel 1, which will call the function "**OB1_Set_Remote_Target**".

19) You can confirm the PID loop is updated by reading the channel again with 'read_channel'

20) To exit the program, first stop the remote mode by selecting 'stop' which will call the function "**OB1_Stop_Remote_Measurement**".

Python:

Quick start

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):

```
#Tested with Python 3.5.1 (IDE Eclipse V4.5.2 + Pydev V5.0.0)
#add python_xx and python_xx/DLL to the project path
# coding: utf8

import sys
from email.header import UTF8
sys.path.append('D:/dev/SDK/python_64/DLL64'.encode('utf-8')) #add the path of the library here
sys.path.append('D:/dev/SDK/python_64'.encode('utf-8'))#add the path of the LoadElveflow.py

from ctypes import *

from array import array

from Elveflow64 import *


#
# Initialization of OB1 ( ! ! ! REMEMBER TO USE .encode('ascii')
#
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))
# all functions will return error code to help you to debug your code, for further information see user guide
print('error:%d' % error)
print('OB1 ID: %d' % Instr_ID.value)

#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)



#
#Set the calibration type
#

Calib=(c_double*1000)() # always define array that way, calibration should have 1000 elements
repeat=True
while repeat==True:
```

3) To make this example work 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
```

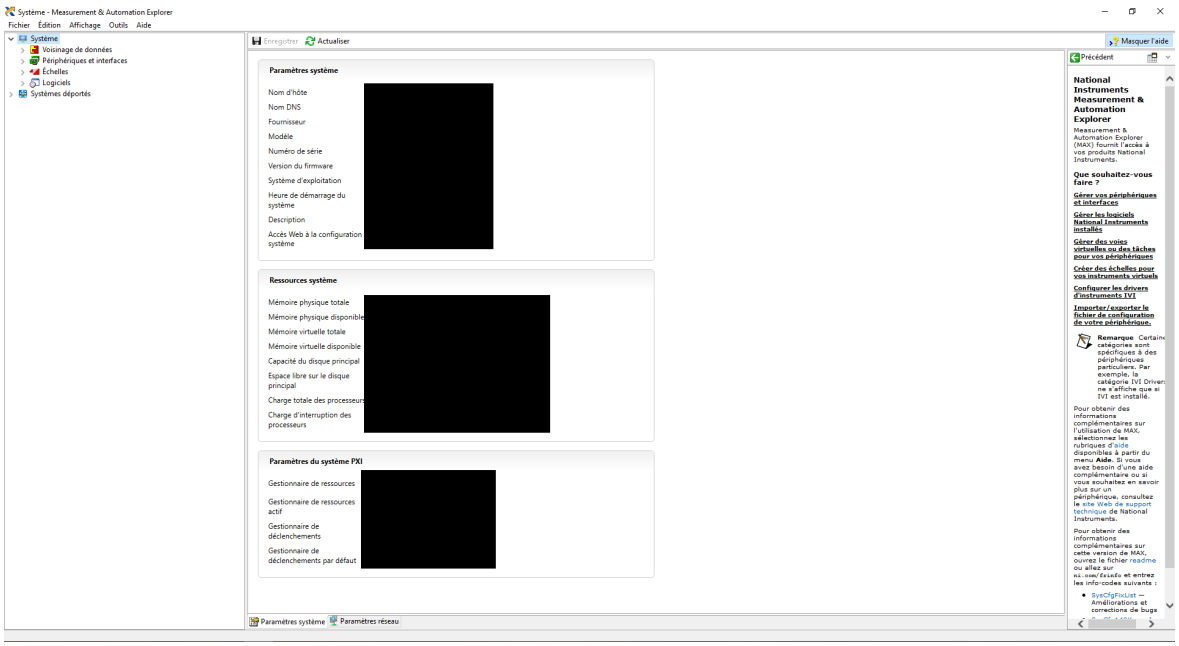
5) Go back to “_OB1_EX_.py”. Then you need to write your instrument name here:

```
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))
```

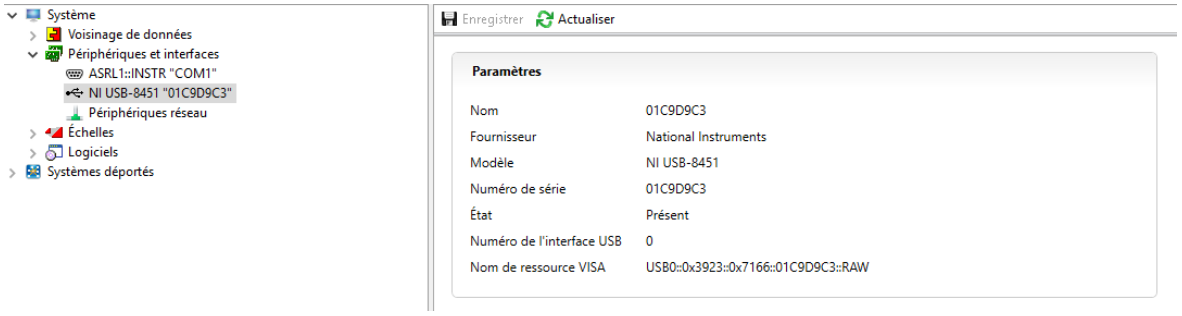
6) To modify this line (23 here) you need to open a software that has been automatically installed on your computer while installing ESI software. This is “NI MAX”. Find NI MAX on your computer by typing “NI MAX” on the Windows search for example and open it.

NI MAX Application

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 find 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 this instrument the modified window is as follows:

```
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 follows:

```
#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...

Remote mode

16) While the example is running and asking for a new command, select ‘start’ which will run the function “**OB1_Start_Remote_Measurement**”. The OB1 is now in remote mode, reading data in an asynchronous loop.

17) To check that the remote loop is operating correctly, now select ‘read_channel’ which will call the function “**OB1_Get_Remote_Data**” and will return the latest measured value from the OB1.

18) You can now immediately start a PID loop by calling ‘add_PID’ which will call the “add_PID” function. By calling read_channel again, you should see the regulator (control) is now following the sensor.

19) Update your new target for the PID loop by simply calling the ‘set_target’ at the channel regulating the PID loop, in this case channel 1, which will call the function “**OB1_Set_Remote_Target**”.

20) You can confirm the PID loop is updated by reading the channel again with ‘read_channel’

21) To exit the program, first stop the remote mode by selecting ‘stop’ which will call the function “**OB1_Stop_Remote_Measurement**”.

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 the LabVIEW error user guide. (http://www.ni.com/pdf/manuals/321551a.pdf)

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

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

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

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

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_70_mbar	7
Z_sensor _type_Press_340_mbar	8
Z_sensor _type_Press_1_bar	9
Z_sensor _type_Press_2_bar	10
Z_sensor _type_Press_7_bar	11
Z_sensor _type_Press_16_bar	12
Z_sensor _type_Level	13
Z_sensor_type_Custom	14

Z_Sensor_digit_analog:

Z_Sensor_digit_analog_Analog	0
Z_Sensor_digit_analog_Digital	1

Z_Sensor_FSD_Calib:

Z_Sensor_FSD_Calib_H2O	0
Z_Sensor_FSD_Calib_IPA	1

Z_D_F_S_Resolution:

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

Z_MUX_DRI_Rotation:

Z_MUX_DRI_Rotation_Shortest	0
Z_MUX_DRI_Rotation_Clockwise	1
Z_MUX_DRI_Rotation_CounterClockwise	2

Z_MUX_DRI_Action:

Z_MUX_DRI_Action_Home	0
Z_MUX_DRI_Action_SerialNumber	1

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 prompts 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 corresponds 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 corresponds 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 corresponds 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).

int32_t __cdecl AF1_Start_Remote_Measurement(int32_t AF1_ID, double Calib_array_in[], int32_t longueur);
Start a loop running in the background, and automatically reads all sensors and regulators. No direct call to the AF1 can be made until the Stop measuring function is called. Until then only functions accessing this loop (get_remote_data, set_remote_target, remote_triggers) are recommended.

int32_t __cdecl AF1_Stop_Remote_Measurement(int32_t AF1_ID);
Stop the background measure & control loop

int32_t __cdecl AF1_Set_Remote_Target(int32_t AF1_ID, double Target);
Set the Target of the AF1 device. Modify the pressure if the PID is off, or the sensor is a pressure sensor. Modify a flow if the sensor is a flow sensor and the PID is on.

int32_t __cdecl AF1_Remote_Triggers(int32_t AF1_ID, int32_t TriggerIn, int32_t *TriggerOut);
Set the Trigger input and get the Trigger output of the AF1 device.

int32_t __cdecl AF1_Get_Remote_Data(int32_t AF1_ID, double *Reg_Data, double *Sens_Data);
Read the sensor and regulator values of the device. Warning: This Function only extracts data obtained in the remote measurement loop Sensor unit : mbar if pressure sensor, µl/min if flow sensor Regulator unit : mbars

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 flows through the sensor prior to density measurement), or before every flow measurement if the density might change. If you get +inf or -inf, the density wasn't correctly measured.

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

int32_t __cdecl BFS_Set_Remote_Params(int32_t BFS_ID, double Filter, int32_t M_temp, int32_t M_density);
Modify the parameters of the remote monitoring loop: M_density: a new measure of the density will be taken before each flow measurement M_temp: a new temperature measurement will be taken after each flow measurement Filter: change the filter used to measure the flow

int32_t __cdecl BFS_Start_Remote_Measurement(int32_t BFS_ID);
Start the monitoring loop for the BFS device.

int32_t __cdecl BFS_Stop_Remote_Measurement(int32_t BFS_ID);
Stop the monitoring loop for the BFS device.

int32_t __cdecl BFS_Get_Remote_Data(int32_t BFS_ID, double *Temperature, double *Density, double *Flow);
Read the sensors from the remote monitoring loop: Units: Flow sensor: µl/min Density: g/m3 Temperature: Celcius

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 returns the F_S_R ID (number >=0) to be used with other functions. NB: Flow reader can only accept Flow sensor NB 2: Sensors 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 Distribution/Distributor/Recirculation/Injection (D-R-I):

int32_t __cdecl MUX_DRI_Destructor(int32_t MUX_DRI_ID_in);
Close Communication with MUX Distribution, Distributor, Recirculation or Injection device.

int32_t __cdecl MUX_DRI_Get_Valve(int32_t MUX_DRI_ID_in, int32_t *selected_Valve);
Get the current valve number. If the valve is changing, the function returns 0.

int32_t __cdecl MUX_DRI_Initialization(char Visa_COM[], int32_t *MUX_DRI_ID_out);
Initiate the MUX Distribution, Distributor, Recirculation or Injection device using device COM port (ASRLXXX::INSTR where XXX is usually the COM port that could be found in Windows device manager). It returns the MUX D-R-I ID (number >=0) to be used with other functions.

int32_t __cdecl MUX_DRI_Set_Valve(int32_t MUX_DRI_ID_in, int32_t selected_Valve, Z_MUX_DRI_Rotation Rotation);
Switch the MUX Distribution, Distributor, Recirculation or Injection to the desired valve. For MUX Distribution 12, between 1-12. For MUX Distributor (6 or 10 valves), between 1-6 or 1-10. For MUX Recirculation 6 or MUX Injection (6 valves), the two states are 1 or 2. Rotation indicates the path the valve will perform to select a valve, either shortest 0, clockwise 1 or counterclockwise 2.

int32_t __cdecl MUX_DRI_Send_Command(int32_t MUX_DRI_ID_in, Z_MUX_DRI_Action Action, char Answer[], int32_t len);
!This function only works for MUX Distribution 12 or Recirculation 6! Get the Serial Number or Home the valve. len is the length of the Answer. Remember that Home the valve takes several seconds. Home the valve is necessary as an initialization step before using the valve for a session.

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 functions

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 an array of 16 elements. If the valve value is equal to or
below 0, the valve is closed, if it's equal to or above 1 the valve is open. The
index in the array indicates the selected valve as shown below:
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
If the array does not contain exactly 16 elements nothing 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 parameters which correspond 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 an array of 16 elements. If the valve value is equal to or below 0, the valve is closed, if it's equal to or above 1 the valve is open. If the array does not contain exactly 16 elements nothing 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 the Flow sensor, the type of communication (Analog/Digital), the Calibration for digital version (H20 or IPA) should be specified as well as digital resolution (9 to 16 bits). (see SDK user guide, Z_sensor_type_type , Z_sensor_digit_analog, Z_Sensor_FSD_Calib and Z_D_F_S_Resolution for number correspondance) For digital versions, the sensor type is automatically detected during this function call. For the Analog sensor, the calibration parameters are not taken into account. If the sensor is not compatible with the MSRD version, or no digital sensor is detected, an error will be thrown as output of the function. NB: Sensor type has to be the same as in the "Initialization" step.

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, int32_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 Sensor, If the connection is lost, MSRD will be reseted and the return value will be zero

int32_t __cdecl M_S_R_D_Set_Filt(int32_t M_S_R_D_ID, int32_t Channel_1_to_4, LVBoolean ONOFF);
Set filter for the corresponding channel.

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, double CustomSens_Voltage_Ch12, double CustomSens_Voltage_Ch34, int32_t *MSRD_ID_out);
Initialize the Sensor Reader device able to read digital sensors (MSRD) using device name and sensor type (see SDK Z_sensor_type for corresponding numbers). It modifies the MSRD ID (number >=0). This ID can be used with other functions to identify the targeted MSRD. If an error occurs during the initialization process, the MSRD ID value will be -1. Initiate the communication with the Sensor Reader able to read digital sensors (MSRD). This VI generates an identification cluster of the instrument to be used with other VIs. NB: Sensor type has to be written here in addition to the "Add_Sens". NB 2: Sensors connected to channel 1-2 and 3-4 have to be the same type otherwise they will not be taken into account.

int32_t __cdecl M_S_R_D_Reset_Instr(int32_t M_S_R_D_ID);

int32_t __cdecl M_S_R_D_Reset_Sens(int32_t M_S_R_D_ID);

int32_t __cdecl M_S_R_D_Start_Remote_Measurement(int32_t M_S_R_D_ID);
Start the monitoring loop for the MSRD device.

int32_t __cdecl M_S_R_D_Stop_Remote_Measurement(int32_t M_S_R_D_ID);
Stop the monitoring loop for the MSRD device.

int32_t __cdecl M_S_R_D_Get_Remote_Data(int32_t M_S_R_D_ID, int32_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 Sensor, If the connection is lost, MSRD will be reseted and the return value will be zero

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, double CustomSens_Voltage_5_to_25);
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 specified as well as digital resolution (9 to 16 bits). (see SDK user guide, Z_sensor_type_type , Z_sensor_digit_analog, Z_Sensor_FSD_Calib and Z_D_F_S_Resolution for number correspondance) For digital version, the sensor type is automatically detected during this function call. For Analog sensors, the calibration parameters are not taken into account. If the sensor is not compatible with the OB1 version, or no digital sensor is detected an error will be thrown as output of the function.

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 properly closed with adequate caps. Len corresponds 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 corresponds to the Calib_array_in length. If Acquire_data is true, the OB1 acquires ALL regulators AND ALL analog sensor value. They are stored in the computer memory. Therefore, if several regulator values (OB1_Get_Press) and/or sensor values (OB1_Get_Sens_Data) have to be acquired simultaneously, set the Acquire_Data to true only for the First function. All the others can use 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 converts data acquired in OB1_Acquire_data Units : Flow sensor µl/min Pressure : mbar If Acquire_data is true, the OB1 acquires ALL regulator AND ALL analog sensor value. They are stored in the computer memory. Therefore, if several regulator values (OB1_Get_Press) and/or sensor values (OB1_Get_Sens_Data) have to be acquired simultaneously, set the Acquire_Data to true only for the First function. All the others can use the values stored in memory and are almost instantaneous. Digital Sensors required another communication protocol, this parameter has no impact NB: For Digital Flow Sensor, If the connection is lost, OB1 will be reset and the return value will be zero.

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 modifies the OB1 ID (number >=0). This ID can be used with other functions to identify the targeted OB1. If an error occurs during the initialization process, the OB1 ID value will be -1.

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

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 corresponds 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)

int32_t __cdecl OB1_Start_Remote_Measurement(int32_t OB1_ID, double Calib_array_in[], int32_t longueur);
Start a loop running in the background, and automatically reads all sensors and regulators. No direct call to the OB1 can be made until the Stop measuring function is called. Until then only functions accessing this loop (get_remote_data, set_remote_target, remote_triggers) are recommended.

int32_t __cdecl OB1_Stop_Remote_Measurement(int32_t OB1_ID);
Stop the background measure & control loop

int32_t __cdecl OB1_Set_Remote_Target(int32_t OB1_ID, int32_t Channel_1_to_4, double Target);
Set the Target of the OB1 selected channel. Modify the pressure if the PID is off, or the sensor is a pressure sensor. Modify a flow if the sensor is a flow sensor and the PID is on.

int32_t __cdecl OB1_Remote_Triggers(int32_t OB1_ID, int32_t TriggerIn, int32_t *TriggerOut);
Set the Trigger input and get the Trigger output of the OB1 device.

int32_t __cdecl OB1_Get_Remote_Data(int32_t OB1_ID, int32_t Channel_1_to_4, double *Reg_Data, double *Sens_Data);
Read the sensor and regulator values of the requested channel. Warning: This Function only extracts data obtained in the remote measurement loop Sensor unit : mbar if pressure sensor, µl/min if flow sensor Regulator unit : mbar NB: For Digital Flow Sensor, If the connection is lost, OB1 will be reseted and the return value will be zero

Remote PID:

int32_t __cdecl PID_Add_Remote(int32_t Regulator_ID, int32_t Regulator_Channel_1_to_4, int32_t ID_Sensor, int32_t Sensor_Channel_1_to_4, double P, double I, int32_t Running);
Configure a PID loop between a regulator and a sensor. Only works when using the remote measurement functions.

int32_t __cdecl PID_Set_Params_Remote(int32_t Regulator_ID, int32_t Channel_1_to_4, int32_t Reset, double P, double I);
Change the parameters of the selected PID loop. Only works when using the remote measurement functions.

int32_t __cdecl PID_Set_Running_Remote(int32_t Regulator_ID, int32_t Channel_1_to_4, int32_t Running);
Set to run/pause the selected PID loop. Only works when using the remote measurement functions.