# PEPPERL+FUCHS

# VsxProtocolDriver C-Wrapper

3.4.3+gc72f69f

*Driver package (C) to communicate with P+F SmartRunner devices via VSX protocol*

# 1 Introduction

The driver VsxProtocolDriver (VsxSdk) provides full access to the input and output data of the sensor. The driver connects to the sensor and handles communication in accordance with the communication protocol. The user can access functions for setting parameters on the sensor, retrieving parameter values from the sensor, and saving and loading entire parameter sets both locally and on the sensor. The user can also receive sensor data like images, 3D-data or lines. Each function also contains an error object from which information can be obtained in the event of an error in the function.

## 1.1 Supported devices

The official supported devices are the following:

- SmartRunner 3D (Stereo + ToF)

- SmartRunner 2D

## 1.2 Requirements

The driver is available for multiple architecture

- Windows 64 bit / 32 bit

- Linux AMD64, ARM64, ARM32

The main driver is based on the C# (.NET). There are wrapper for C and Python programming language available.

For usage the Microsoft .NET Runtime 6.0.x framework or higher must be installed (See `https://dotnet.↩ microsoft.com/en-us/download/dotnet`).

# 2 Usage with C interface

The driver VsxProtocolDriver (VsxSdk) facilitates integration in a C- based programming environment.

The main driver is implemented in C# and requires .NET 6.0 or higher.

The functions of the C-wrapper can only be used synchronously.

## 2.1 Requirements

The driver is available as C library and header for multiple architecture

- Windows 64 bit / 32 bit

- Linux AMD64, ARM64, ARM32

The driver is based on the VsxProtocolDriver, which is based on C#. So for usage the Microsoft .NET Runtime 6.0.x framework or higher must be installed (See `https://dotnet.microsoft.↩ com/en-us/download/dotnet`). There is also still support for .Net 5.0, but this will propably be dropped in the next version.

## 2.2 Installation

In order to use the SDK, the files are located inside the zipped driver. Unzip the files and select the correct architecture, e.g. "win-x64".

The driver package contains the following files:

- Header
    - PF.VsxProtocolDriver.WrapperNE.h (main header)
    - dnne.h (internal used header)
- Library
    - PF.VsxProtocolDriver.WrapperNE.lib (import library)
    - PF.VsxProtocolDriver.WrapperNE.dll/.so (dynamic library file)
- Referenced .net dlls
    - Multiple dlls located in `PF.VsxProtocolDriver.WrapperNE` folder

The remaining files are needed for the .net driver. These files and the dynamic library file must be located in the same folder as the executable to run.

## 2.3  Documentation

The `PF.VsxProtocolDriver.Wrapper` is a C# interface to communicate with VSX based sensors. It is based on the C# based `VsxProtocolDriverSync` driver. From the `PF.VsxProtocolDriver.Wrapper` an C-interface is build automatically.

### 2.3.1  Memory management

When the user requests string, images or other objects, the memory will be allocated on the C# side and needed to be release later with the "ReleaseXXX" function

```c
#include "PF.VsxProtocolDriver.WrapperNE.h"
const char* version = NULL;
VsxStatusCode ret;
ret = vsx_GetLibraryVersion(&version);
if (ret == VSX_STATUS_SUCCESS)
{
    printf("Version %s\n", version);
    ret = vsx_ReleaseString(&version);
}
```

The reference of the object pointer (e.g. `const char* version`) must be set to `NULL`, before allocating the object. Otherwise the function will return an error. After allocation, the pointer is valid and can be used.

There are the following release functions available:

| Function | Object |
|---|---|
| vsx_ReleaseString | const char∗ |
| vsx_ReleaseSensor | VsxSystemHandle∗ |
| vsx_ReleaseDataContainer | VsxDataContainerHandle∗ |
| vsx_ReleaseImage | VsxImage∗ |
| vsx_ReleaseLine | VsxLine∗ |
| vsx_ReleaseDisparityDescriptor2 | VsxDisparityDescriptor2∗ |
| vsx_ReleaseTransformation | VsxTransformation∗ |
| vsx_ReleaseCaptureInformation | VsxCaptureInformation∗ |
| vsx_ReleaseOlr2CaptureInformation | VsxOlr2CaptureInformation∗ |
| vsx_ReleaseOlr2ModbusData | VsxOlr2ModbusData∗ |
| vsx_ReleaseTagList | VsxTagList∗ |
| vsx_ReleaseDevice | VsxDevice∗ |
| vsx_ReleaseDeviceList | VsxDeviceList∗ |
| vsx_ReleaseParameter | VsxParameter∗ |
| vsx_ReleaseParameterList | VsxParameterList∗ |
| vsx_ReleaseStatusItemList | VsxStatusItemList∗ |

When the release function is called, the reference of the object will be set back to `NULL`.

### 2.3.2  Error handling

Normally all functions return a `VsxStatusCode`. The enum is available in the `PF.VsxProtocolDriver.↩ WrapperNE.h` header. A successful function return `VSX_STATUS_SUCCESS` (0). Errors starts with `VSX_↩ STATUS_ERROR_XXX` and have a negative number.

To access the enum status code as text, the `vsx_GetErrorText` can be used:

```
#include "PF.VsxProtocolDriver.WrapperNE.h"
const char* error_text = NULL;
VsxStatusCode status_code = <call_another_function>(...);
VsxStatusCode ret = vsx_GetErrorText(status_code, &error_text);
if (ret == VSX_STATUS_SUCCESS)
{
    printf("Error %s\n", error_text);
    ret = vsx_ReleaseString(&error_text);
}
```

This will also return dynamic error as additional text, when available.

### 2.3.3  Support of dynamic container (including 3D data)

With the new available `Dynamic container` inside the VSX protocol, multiple images and data message can be send together. So now a container for e.g. SR3D stereo can contain a left image, right image and a disparity map. Other products can contain an image and a result message. For compatibility reason non dynamic container packages will be packed into a new dynamic container, when reaching the driver. So they can be used in the same manner.

The single message data (e.g. image, result) can be accessed by an `tag` name. This could be e.g. `LeftRaw`, `Image`. The naming is product specific.

The dynamic container are stored internally by a queue, which can be configured by the follwing function:
```
VsxStatusCode vsx_ResetDynamicContainerGrabber(VsxSystemHandle* vsx, int32_t bufferSize, int32_t
      startCondition, VsxStrategy strategy);
```

The buffer size allocates queue with a certain size. The strategy option change the behaviuour of new arriving dynamic containers. With `VSX_STRATEGY_DROP_OLDEST` the oldest are dropped, so that the queue is filled up with new dynamic container. With `VSX_STRATEGY_DROP_WRITE` the newest are dropped and the queue will only be filled until it is full.

The optional parameter `startCondition` is not used now.

To generate a queue, which always show the last transmitted `dynamic container` use `bufferSize=1` and `strategy=VSX_STRATEGY_DROP_OLDEST`.

To capture the next 5 trigger after the reset set `bufferSize=5` and `strategy=VSX_STRATEGY_DROP_↵ WRITE`.
```
// allocatee sensor and connect it -> vsx
VsxStatusCode ret;
ret = vsx_ResetDynamicContainerGrabber(vsx, 1, -1, VSX_STRATEGY_DROP_OLDEST);
// Trigger sensor (SW or HW)
VsxDataContainerHandle* dch = NULL;
ret = vsx_GetDataContainer(vsx, &dch, 1000);
if (ret == VSX_STATUS_SUCCESS)
{
    ret = vsx_SaveData(dch, "Image", "Image.bmp");
    ret = vsx_ReleaseDataContainer(&dch);
}
```

### 2.3.4  Firmware Update

The firmware update will now be able to support both hardware platforms:

- Linux based system with rescue system
    - The firmware will be uploaded to the system.
    - Afterwards the sensor will start automatically to the rescue system and update the system
    - When the system starts again, the function will return.
- Texas Instruments (flash based system)
    - The update will be written into the flash memory.
    - The function will return afterswards.
    - The sensor must be started *manually* new by a power down-up cycle to get started with new firmware.

```
VsxStatusCode vsx_SendData(VsxSystemHandle* vsx, const char* fileName);
```

## 2.4 Functions

### 2.4.1 Common library functions

As described in the memory managment section, every string returned by a function must be freed by release string.
```
VsxStatusCode vsx_ReleaseString(const char** pString);
```

The version of driver can be accessed by the following function:
```
VsxStatusCode vsx_GetLibraryVersion(const char** version);
```

To get a text for a `VsxStatusCode` (including dynamic error text) use this function:
```
VsxStatusCode vsx_GetErrorText(int32_t error_code, const char** error_text);
```

### 2.4.2 Sensor handling

With the driver multiple sensor can be accessed.

To generate a tcp or serial sensor call the init function:
```
VsxStatusCode vsx_InitTcpSensor(VsxSystemHandle** pVsx, const char* ipAddress, const char* pluginName);
VsxStatusCode vsx_InitSerialSensor(VsxSystemHandle** pVsx, const char* serialPort, int32_t baudrate, const
    char* sensorType, VsxSerialConnectionType connectionType, const char* pluginName);
```

The connection can be tried to build up by the connect function. In the case of tcp sensor, where multiple devices accessed on the same ip address the call of `vsx_ConnectEx` with a `timeout_ms > 300000` might be useful.
```
VsxStatusCode vsx_Connect(VsxSystemHandle* vsx);
VsxStatusCode vsx_ConnectEx(VsxSystemHandle* vsx, int32_t timeout_ms);
```

> The arp cache of Windows delays the recognition of a new mac address on the same ip address. The process could take up to 30s.

To disconnect a sensor call:
```
VsxStatusCode vsx_Disconnect(VsxSystemHandle* vsx);
```

To release a sensor after usage call:
```
VsxStatusCode vsx_ReleaseSensor(VsxSystemHandle** vsx);
```

### 2.4.3 Sensor handling (Reconnect functions)

For testing it could be interesting to change a current connection e.g. to another ip address.
```
VsxStatusCode vsx_ReConnectTcpSensor(VsxSystemHandle* vsx, const char* ipAddress);
VsxStatusCode vsx_ReConnectSerialSensor(VsxSystemHandle* vsx, const char* serialPort, int32_t baudrate,
    VsxSerialConnectionType connectionType);
```

### 2.4.4 Sensor functions (excluding dynamic container)

The default timeout for standard function calls can be read and written
```
VsxStatusCode vsx_GetWaitTimeout(VsxSystemHandle* vsx, int32_t* result);
VsxStatusCode vsx_SetWaitTimeout(VsxSystemHandle* vsx, int32_t timeout_ms);
```

It is used by the functions declared in this paragraph.

The `vsx_TestSystem` is used by production. It gets a `command` and `inputValue` string as input and returns an `ouputValue`string and `status` code as result. It uses the default timeout.
```
VsxStatusCode vsx_TestSystem(VsxSystemHandle* vsx, const char* command, const char* inputValue, const char**
    outputValue, int32_t* status);
```

The status code is `1`, if command call is valid, otherwise `0`.

The `vsx_TestSystemEx` is the same as above, but an indivudal timeout in ms could be set.
```
VsxStatusCode vsx_TestSystemEx(VsxSystemHandle* vsx, const char* command, const char* inputValue, const
    char** outputValue, int32_t* status, int32_t timeout_ms);
```

The status code is `1`, if command call is valid, otherwise `0`.

To change the ip address of the connected sensor, use the following function:
```
VsxStatusCode vsx_SetNetworkSettings(VsxSystemHandle* vsx, const char* ipAddress, const char* networkMask,
    const char* gateway);
```

**2.4.4.1  Parameter handling**  To save the actual parameter set onto the sensor (and override the last saved parameter set) call

```
VsxStatusCode vsx_SaveParameterSetOnDevice(VsxSystemHandle* vsx);
```

To load the parameter set from the sensor (and override the actual parameter set) call

```
VsxStatusCode vsx_LoadParameterSetOnDevice(VsxSystemHandle* vsx);
```

To load the default parameter set from the sensor (and override the actual parameter set) call

```
VsxStatusCode vsx_LoadDefaultParameterSetOnDevice(VsxSystemHandle* vsx);
```

To upload or download the actual parameter set on the sensor the following functions are avaiable.

```
VsxStatusCode vsx_UploadParameterSet(VsxSystemHandle* vsx, const char* fileName);
VsxStatusCode vsx_DownloadParameterSet(VsxSystemHandle* vsx, const char* fileName);
```

To set and get a single parameter inside the parameter set, the following functions can be used.

```
VsxStatusCode vsx_SetSingleParameterValue(VsxSystemHandle* vsx, uint32_t settingsVersion, const char*
     configurationId, uint32_t configurationVersion, const char* parameterId, const char* value);
VsxStatusCode vsx_GetSingleParameterValue(VsxSystemHandle* vsx, uint32_t settingsVersion, const char*
     configurationId, uint32_t configurationVersion, const char* parameterId, const char** value);
```

The parameter structue has two layer. A configuration group (e.g. "Base") contains multiple parameters. The sensor itself contains one or multiple configuration groups. To make a serialization / deserialization of data possible, each layer has its own version number. With this in place, there is the possiblity to add new parameter / configuration, remove unused parameter and even change the unit of a parameter.

To define a parameter, the following parameters are needed:

settingsVersion -> Version number, which tells the senosr, which configurations are available configurationId -> Name of current configuration group configurationVersion -> Version number, which tells the senosr, which parameters are available in given configuration group parameterId -> Name of parameter id

An example would be the following parameter for the Smartrunner 3D Stereo sensor:

settingsVersion -> 2 configurationId -> "Base" configurationVersion -> 2 parameterId -> "ExposureTime"

When a parameter is set, with e.g. vsx_SetSingleParameterValue(vsx, 2, "Base", 2, "ExposureTime", "1000"); the serialzation will guarantee also in case of an sensor firmware upgrade, will still work.

**2.4.4.2  Send other data**  To send data (like firmware, images or xml commands) to the sensor the following functions can be used:

```
VsxStatusCode vsx_UploadData(VsxSystemHandle* vsx, const char* fileName);
VsxStatusCode vsx_SendXmlMessageData(VsxSystemHandle* vsx, const char* xmlCommand);
VsxStatusCode vsx_SendFirmware(VsxSystemHandle* vsx, const char* fileName);
```

> Which kind of data is supported is sensor type specific

**2.4.5  Log message handling**

The log messages could also be buffered by the driver. With `vsx_ResetLogMessageGrabber` the handling of incoming data can be configured (see chapter "Support of dynamic container"). The type masked should be defined by the following bitmask:

```
LOGT_DBG = 0x01,
LOGT_INFO = 0x02,
LOGT_RESOK = 0x04,
LOGT_RESNOK = 0x08,
LOGT_WARN = 0x10,
LOGT_ERR = 0x20,
LOGT_CRIT = 0x40,
LOGT_ASSERT = 0x80,
LOGT_ALL = 0xFFFFFFFF
```

With `VSX_STRATEGY_DROP_OLDEST` the oldest are dropped, so that the queue is filled up with new dynamic container. With `VSX_STRATEGY_DROP_WRITE` the newest are dropped and the queue will only be filled until it is full.

```
VsxStatusCode vsx_ResetLogMessageGrabber(VsxSystemHandle* vsx, int32_t bufferSize, int32_t typeMask,
     VsxStrategy strategy);
VsxStatusCode vsx_GetLogMessage(VsxSystemHandle* vsx, const char** log, int32_t timeout_ms);
```

The `vsx_GetLogMessage` function tries to receive data in a certain time period (parameter `timeout_ms`) from the receiver queue.

### 2.4.6 Dynamic container handling

The dynamic container grabber is used to receive images, result and other data from the sensor.

**2.4.6.1 configure, receive data and release** With `vsx_ResetDynamicContainerGrabber` the handling of incoming data can be configured (see chapter "Support of dynamic container").

```
VsxStatusCode vsx_ResetDynamicContainerGrabber(VsxSystemHandle* vsx, int32_t bufferSize, int32_t
    startCondition, VsxStrategy strategy);
```

The `vsx_GetDataContainer` function tries to receive data in a certain time period (parameter `timeout_ms`) from the receiver queue.

```
VsxStatusCode vsx_GetDataContainer(VsxSystemHandle* vsx, VsxDataContainerHandle** pDch, int32_t timeout_ms);
```

Now the data of the container be accessed. Afterwards the container must be freed by the following function:

```
VsxStatusCode vsx_ReleaseDataContainer(VsxDataContainerHandle** dch);
```

**2.4.6.2 Tag list** When a dynamic container is received, the data of the container could be listed. Therefor `vsx_GetTagList` could receive an array of tag names. This must be deleted afterwards with `vsx_Release↩TagList`.

```
VsxStatusCode vsx_GetTagList(VsxDataContainerHandle* dch, VsxTagList** tagList)
VsxStatusCode vsx_ReleaseTagList(VsxTagList** pTagList)
```

Somtimes tags are generated on the fly, e.g. for 3D data from disparity values. These are not listed.

**2.4.6.3 Access data** When a dynamic container is received, the data can be accessed. The data inside the dynamic container can be accessed by the `tag` name, which are defined by the sensor type. Examples are `Image`, `LeftRaw`, `Result`.

With `vsx_SaveData` you can save data into a file on the PC. The `tag` and the `filename` must be specified. The file ending will be used to define e.g. the image encoding (`.bmp`, `.png`). When using `tag="*"` an `.zip` as file ending the complete dynamic container will be saved compressed file.

```
VsxStatusCode vsx_SaveData(VsxDataContainerHandle* dch, const char* tag, const char* fileName);
```

For images there is also the possibility to generate a memory access. This can be done with `vsx_GetImage` and delivers a `VsxImage`, where the raw data pointer and attributes like height and width are defined. After usage the memory must be released by `vsx_ReleaseImage`.

```
VsxStatusCode vsx_GetImage(VsxDataContainerHandle* dch, const char* imageTag, VsxImage** imageData);
VsxStatusCode vsx_ReleaseImage(VsxImage** pImage);
```

To save point cloud data, all three data layer must be specified. The supported data format is only the `.pcd` (Point Cloud Data) file format.

```
VsxStatusCode vsx_Save3DPointCloudData(VsxDataContainerHandle* dch, const char* point_x_Id, const char*
    point_y_Id, const char* point_z_Id, const char* fileName);
```

For result data the whole xml string (`vsx_GetResultXml`) can be received. To receive a single value from the xml an xml path expression must be given. The return value will be converted to the selected data type (string, int, long or double)

```
VsxStatusCode vsx_GetResultXml(VsxDataContainerHandle* dch, const char* resultId, const char** result);
VsxStatusCode vsx_GetResultElementString(VsxDataContainerHandle* dch, const char* resultId, const char*
    xPath, const char** result);
VsxStatusCode vsx_GetResultElementInt32(VsxDataContainerHandle* dch, const char* resultId, const char*
    xPath, int32_t* result);
VsxStatusCode vsx_GetResultElementInt64(VsxDataContainerHandle* dch, const char* resultId, const char*
    xPath, int64_t* result);
VsxStatusCode vsx_GetResultElementDouble(VsxDataContainerHandle* dch, const char* resultId, const char*
    xPath, double* result);
```

### 2.4.7 Display device information

To read out the current device information call the following function. There is also a release function, to free up the memory by the driver.

```
VsxStatusCode vsx_GetCurrentDeviceInformation(VsxSystemHandle* vsx, VsxDevice** deviceData);
VsxStatusCode vsx_ReleaseDevice(VsxDevice** pDevice);
```

Send an UDP request to find sensor in the network. It returns a list, with the information. There is also a release function, to free up the memory by the driver.

```
VsxStatusCode vsx_GetUdpDeviceList(VsxDeviceList** deviceListData);
VsxStatusCode vsx_ReleaseDeviceList(VsxDeviceList** pDeviceList);
```

# 3 Examples

In the following the usage of the `VsxProtocolDriver` is shown with a short code example.

The complete examples can be found as a CMake project in the `C\example\` subfolder. It support the detection of different sensors and show the parametrization and the grabbing of data from the sensor.

```c
#include "PF.VsxProtocolDriver.WrapperNE.h"
#include <stdio.h>
void print_error(VsxStatusCode code)
{
    const char* error_text = NULL;
    VsxStatusCode ret;
    ret = vsx_GetErrorText(code, &error_text);
    if (ret == VSX_STATUS_SUCCESS)
    {
        printf("Error code %s\n", error_text);
        ret = vsx_ReleaseString(&error_text);
    }
}
int main(int argc, char** argv) {
    VsxDeviceList* devList = NULL;
    VsxStatusCode ret;
    VsxSystemHandle* vsx = NULL;
    ret = vsx_GetUdpDeviceList(&devList);
    if (ret != VSX_STATUS_SUCCESS)
    {
        return -1;
    }
    if (devList->length > 0)
    {
        // create a new VsxProtocolDriver instance
        VsxDevice dev = devList->devices[0];
        printf("Device found:  %s %s\n", dev.sensorType, dev.ipAddress);
        ret = vsx_InitTcpSensor(&vsx, dev.ipAddress, "");
    }
    else //use fix ip address
    {
        // create a new VsxProtocolDriver instance with fix ip address
        ret = vsx_InitTcpSensor(&vsx, "192.168.2.4", "");
    }
    if (ret != VSX_STATUS_SUCCESS)
    {
        print_error(ret);
        return -2;
    }
    // Connect with device
    ret = vsx_Connect(vsx);
    if (ret != VSX_STATUS_SUCCESS)
    {
        print_error(ret);
        ret = vsx_ReleaseSensor(&vsx);
        return -3;
    }
    // Get the current device information
    VsxDevice* device = NULL;
    ret = vsx_GetDeviceInformation(vsx, &device);
    if (ret == VSX_STATUS_SUCCESS)
    {
        printf("Actual IP from device %s\n", device->ipAddress);
        ret = vsx_ReleaseDevice(&device);
        if (ret != VSX_STATUS_SUCCESS)
        {
            print_error(ret);
            ret = vsx_ReleaseSensor(&vsx);
```

```
            return -5;
        }
    }
    else
    {
        print_error(ret);
        ret = vsx_ReleaseSensor(&vsx);
        return -4;
    }
    // Release sensor instance
    ret = vsx_ReleaseSensor(&vsx);
    if (ret != VSX_STATUS_SUCCESS)
    {
        print_error(ret);
        return -5;
    }
    return 0;
}
```

# 4 Device parameter

In this chapter some information about the structure of the device parameters shall be given.

The device parameters are organized in two levels. The first level includes one or more configuration groups. Each of these configuration groups in turn contains one or more parameters. To uniquely identify parameters, each configuration has a unique Id. Each parameter also contains an Id that is unique within its configuration.

In order to keep different firmware versions compatible with each other, an additional versioning exists. This comprises on the one hand a settings version, which determines, which configurations are present up-to-date, and a configuration version, which determines which parameters are present at the moment in this configuration. If changes are made to configurations or parameters, the respective version number is increased.

Four arguments are hence required to uniquely define a device parameter:

- *settingsVersion*: Version number, which tells the device which configurations are available

- *configurationId*: Id of the current configuration group

- *configurationVersion*: Version number, which tells the device which parameters are available whithin the current configuration group and how they are handled

- *parameterId*: Id of the current parameter

In order to know the individual parameters with their ids and versions, files for all supported sensor types and their various firmware versions are stored in a source file in the example subfolder named with $<$sensor_$\hookleftarrow$ name$>$ParameterIdentifier. The required informations can be taken from these files.

**Example**: The value of the following parameter for the Smartrunner 3-D device:

- *settingsVersion*: 2

- *configurationId*: "Base"

- *configurationVersion*: 2

- *parameterId*: "ExposureTime"

can be received using the driver via the function GetSingleParameterValue(settingsVersion:2, configId:"Base", configVersion:2, parameterId:"ExposureTime").

**Additional notes:**

- if a configuration or parameter does not contain a version attribute, use the default value of "1".

- in addition to the information on version and Id, the xml files also contain further information on the parameters such as name, value range, etc.

- to trigger event parameters these must be set to a value of "1".

- for the Smartrunner 2-D, only a part of the parameters is listed in the corresponding xml file. Only these parameters should be used for parameterization of the device.

# 5 Changelog

This is the changelog for the C implementation of the VsxProtocolDriver. It is based on the .NET implementation (C#) of the VsxProtocolDriver. Please use also the .NET documenatation for additional informations about the release.

V3.4.3

- based on VsxProtocolDriver 3.4.3

V3.4.2

- based on VsxProtocolDriver 3.4.2

V3.4.1

- based on VsxProtocolDriver 3.4.1
- Add support for new "InitializeDevice" function

V3.4.0

- Minumum requirement .net 6.0
- Add support for bool in "GetSingleParameterValueInt32"
- based on VsxProtocolDriver 3.4.0

V3.3.3

- Dropped .NET 5.0 support
- based on VsxProtocolDriver 3.3.3

V3.3.2

- based on VsxProtocolDriver 3.3.2

V3.3.1

- based on VsxProtocolDriver 3.3.1

V3.3.0

- based on VsxProtocolDriver 3.3.0

V3.2.1

- based on VsxProtocolDriver 3.2.1

- Add access to write and read single parameter with "double" and "int32_t" type (instead of only string as before)

V3.1.5

- based on VsxProtocolDriver 3.1.5
  - Modified "Olr2CaptureInformation" data structure (incompatible with V3.1.0 and following, only for Olr2!)

V3.1.4

- Fix memory leak inside line data allocation

V3.1.3

- based on VsxProtocolDriver 3.1.3
  - use given ip address instead udp response to connect

V3.1.2

- based on VsxProtocolDriver 3.1.2
- Receive also "ApplicationResultData" as result
- Correct handling of VsxLine (when Quality is missing)
- Usage of UTF-8 for string encoding

V3.1.1

- based on VsxProtocolDriver 3.1.1
- Add ".VsxLine" (xml-based) as new and default line format

V3.1.0

- based on VsxProtocolDriver 3.1.0
- Add function "vsx_SendSessionKeepAlive" (reply to timeout announcement message)
- Added support for "Olr2ModbusData" & "Olr2CaptureInformation"

V3.0.6

- based on VsxProtocolDriver 3.0.6
  - Correct saving of images in "bmp" & "png" format again (introduced in 3.0.5)

V3.0.5

- based on VsxProtocolDriver 3.0.5

    – Allow saving of images in "bmp" format again

- Change "Parameter" struct handling, values are set now by dedicated function, e.g. "vsx_SetSingle↩ ParameterString"

- Add missing "VsxImageData2Format" enum values in header:

    – VSX_IMAGE_DATA2_FORMAT_COORD3D_A32F, VSX_IMAGE_DATA2_FORMAT_COORD3D_B32F, VSX_IMAGE_DATA2_FORMAT_COORD3D_C32F

- Add comments to explain, which value type to use for "VsxParameterValueType" & "VsxStatusItemValueType"

V3.0.4

- Remove "InitTcpSensorEx" & "ReConnectTcpDeviceEx" function from 3.0.2/3.0.3 again (not needed, if direct UDP from sensor supported)

- based on VsxProtocolDriver 3.0.4

V3.0.3

- Add "ReConnectTcpDeviceEx" function to support setting of port number

- Correct V3.0.2, where "ReConnectTcpDevice" used already new port attribute.

V3.0.2

- Add "InitTcpSensorEx" function to support setting of port number

V3.0.1

- Correct handling for 64 bit values for (really) old compiler

V3.0.0

- based on VsxProtocolDriver 3.0.0

- First support for encrypted login

- Changed function call of "vsx_ResetDynamicContainerGrabber" (removed one unused parameter)