Technical Note
XSCore DLL

Terence Russell
Director Software Development
2022-09-28

This document describes the functioning and usage of the XSENSOR XSCore DLL.

INTRODUCTION

Welcome to the exciting world of pressure sensing interfaces! The XSENSOR XSCore DLL ("the DLL") is used to record live pressure data from XSENSOR pressure sensors. The DLL supports X3 and X4 sensor models in both USB wired and Bluetooth wireless configurations.

XSENSOR sensors capture pressure data in a 2D grid of cells. These cells are referred to as "sensels". The entire grid of cells is referred as a "pressure map". However, this document also uses the terms "frame" and "sample" to describe the grid of cells. Capturing pressure data is referred to as "sampling" or "recording".

DLL INTERFACE

This DLL operates on Microsoft Windows 7 through Windows 10 in either 32-bit or 64-bit modes. It may have a dependency on the *Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017 and 2019*. At the time of this writing the following Microsoft links go to the 32-bit and 64-bit installations respectively.

https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads-2647da03-1eea-4433-9aff-95f26a218cc0

- x86: vc_redist.x86.exe
- x64: vc_redist.x64.exe

The DLL exports a standard Windows 'C' interface. Many other programming languages support integration with 'C' DLLs. The DLL package includes integration wrappers for Python 3 and the C# programming language.

THEORY OF OPERATION

The DLL is initialized with a call to `XS_InitLibrary` which establishes its major operating mode (described below). Sensor enumeration is performed to find all sensors connected to the computer. A sensor configuration is then constructed that contains one or more sensors. Operating parameters are then specified for the configuration.

Next, a connection is opened to the sensor configuration and the DLL begins sampling. A call to `XS_Sample` copies each sensor's pressure map into an internal memory buffer. The `XS_GetPressure` function is used to retrieve a specific sensor's data from this buffer. The memory buffer remains unchanged until another `XS_Sample` call is made.

When sampling is no longer needed the connection should be closed. Further recordings can be made with the configured sensors by reopening the connection. If no more recordings are desired with the configuration, then the configuration should be released. Finally, the DLL should be uninitialized with a call to `XS_ExitLibrary`.

Sensor Enumeration

Sensor enumeration is the process of making the DLL aware of any sensors connected to the computer. Sensors may be connected via a USB wired connection or via Bluetooth wireless. For sensors operating in the Bluetooth mode, they must have already been added as a Bluetooth device using the Windows Bluetooth settings panel.

Sensor enumeration is accomplished through either the `XS_EnumSensors` function or via one of the `XS_AutoConfig*` functions. This operation temporarily opens a connection to each sensor to retrieve their identifying information. Bluetooth connections can sometimes take up to 4 seconds to open and so enumeration can be a slow process.

The DLL may fail to find any or all connected sensors. This can occur when the sensors are already open in another software or when there are hardware system issues. Call `XS_GetLastEnumState` to retrieve an error code which may indicate the cause of the problem.

The list of found sensors can be retrieved using the `XS_EnumSensorPID` function. Provided a zero-based index to retrieve each sensor's `SENSORPID`. The `SENSORPID` uniquely identifies each sensor. The `SENSORPID` is used in various other functions to retrieve details about the sensor, such as its display name and dimensions. Use the `XS_EnumSensorCount` to determine how many sensors were found by the enumeration process.

Given the expensive of finding all sensors connected to a computer, there are settings that enable or disable certain sensors from the enumeration scan. By default, only wired X3 sensors are enumerated.

The rare wireless X3 model can be added to the scan via `XS_SetAllowWireless`. X4 sensors are enabled in their USB wired and Bluetooth wireless modes with calls to `XS_SetAllowX4` and `XS_SetAllowX4Wireless` respectively.

Sensor Configuration

Before any sampling can occur, a sensor configuration must be constructed with one or more sensors. Sensor configurations can be constructed manually, or automatically using the `XS_AutoConfig*` functions. Normally a sensor configuration is setup to discard pressure frames keeping only the most recent frame in memory. However, its possible to specify a configuration that records all pressure data to an XSENSOR session file (XSN). These session files can be later processed with the XSENSOR Desktop software products or the XSNReader.DLL. Pressure readings can still be obtained directly even while recording to a session file.

Automatic Configuration

Configuration creation can be simplified using the `XS_AutoConfig*` functions. These functions automatically configure all found sensors and handle setup of the calibration state. (Calibration is described below). `XS_AutoConfigByDefault` and `XS_AutoConfigByDefaultXSN` are the simplest functions for constructing a configuration.

Manual Configuration

In some cases where two or more sensors are present, it might be desirable to configure only some of the sensors. Manual configuration allows this goal.

Begin manual configuration with a call to `XS_NewSensorConfig` or `XS_NewSensorConfigXSN`. This clears out any existing configuration and preps the DLL for a new one.

Next, add sensors to the configuration using the `XS_AddSensorToConfig*` functions. These functions require some indication of calibration state. (Calibration is described below).

Finally, call `XS_ConfigCompete` to tell the DLL that no more sensors are going to be added to the configuration. This call is only required for manual configurations.

Sensor Calibration

The XSENSOR electronics hardware converts electrical signals from the sensor pads into an array of 16-bit digital readings. These digital readings are converted to pressure by way of a file that contains mappings from the digital form to a real number pressure. This conversion process is called “calibration” and the mapping file is called a “calibration file” (file extension is .xsc).

Each sensor that is calibrated at the XSENSOR manufacturing facility contains one or more calibration files located onboard its memory. Each calibration file is created for a specific pressure range (for example 1 psi to 50 psi). While most sensors are only calibrated for a single pressure range, some are calibrated for mor. In those cases, the sensor’s memory hold two or more calibration files.

The DLL is responsible converting the digital readings into pressure readings, and so it requires access to the calibration files found on the sensors.

In order to use a calibration file, the DLL must download it from a sensor’s memory. This download can take between 30 and 120 seconds, depending on the size of the file and the speed of the connection to the sensor. By default, the DLL only holds only a temporary copy of the downloaded file in memory. This file is lost when the DLL is shutdown and must be downloaded again on subsequent use. However, its possible to save calibration files to a folder and avoid repetitive downloads.

The easiest method for saving is using the `XS_SetCalibrationFolder` function. Just pass in a full file path to the function (ex: “c:\\mycalibrationfiles”) and the DLL automatically saves any downloaded calibration files into that calibration folder. On enumeration the DLL scans this folder for existing calibration files that match the found sensors. If match is found, the calibration download is skipped, and the DLL uses that sensor’s file from the folder. This is a good timer saver for subsequent recording sessions.

An alternative method is to manually download a sensor’s calibration files using the `XS_DownloadCalibrations` function. In this case the calibration files must be manually managed during the configuration process. Some of the configuration functions take a calibration file path such as `XS_AddSensorToConfig` and `XS_AutoConfig_SingleSensor`.

In most cases, using `XS_SetCalibrationFolder` and allowing the DLL to manage the calibration files is ideal. The main reason for manually managing these files is when a sensor contains more than one calibration file. This allows selection of a specific calibration file. Otherwise, when a sensor has more than one calibration file, the DLL always selects the first one it encounters.

Sensor Pre-connection

After sensor configuration is complete and prior to opening a connection, some of the recording parameters can be modified. Changing these settings after a connection is open has no effect.

The Sample Average Count is abstractly the number of electronic readings the hardware performs for each scan of each sensel. By default, the DLL uses the sensors recommended counts. Increasing the count may slightly improve the stability of the readings, but at the expensive of achievable framerate. This setting is accessed via the `XS_GetSampleAverageCount` and the `XS_SetSampleAverageCount` functions. (The XSCore90.h header gives more detail on values to use.)

The Overlap Mode flag is used to sequence the recording of pressure frames from two or more sensors in cases where the sensors might be in physical contact. Reading from one sensor while another one is energized leads to noisy readings. This flag is ignored when only one sensor is configured or X4 sensors are configured. Use `XS_SetOverlapMode` and `XS_GetOverlapMode`.

The Read Timeout controls how long the DLL will wait for a non-responsive sensor to resume communications before self closing the connection to all configured sensors. Use `XS_SetReadTimeout` and `XS_GetReadTimeout`.

The Streaming Mode setting allows buffering of more than one pressure frame when using a sensor configuration that does not record to an XSN session file. Normally when not recording to an XSN session file, the DLL only buffers a single frame and discards it as new frames appear. This mode allows the DLL to build a cache of frames while waiting for `XS_Sample` to be called. Once the cache is filled, the DLL begins removing the oldest frame from the cache. A call to `XS_Sample` removes one frame from the cache.

Use `XS_SetStreamingMode` to enable streaming mode. The `StreamingMaxFrameCache` count is the number of frames to cache. If a value of 0 is specified, then the DLL caches all frames. If `XS_Sample` is not called to consume the frames, then the DLL will cache frames until is runs out of memory.

Sensor Connection

A connection to the configured sensors is made with the `XS_OpenConnection` function. This function takes the target frame rate, which is set in terms of frames per minute. Once the connection is made, the sensors begin taking pressure readings immediately.

However, the DLL may or may not be able to support the target frame rate. An unsupported frame rate may or may not cause `XS_OpenConnection` to fail depending on the type of connected sensors.

Sampling frame rates are limited by a few factors: including the size of the sensor; the speed of the sensor's SPK hardware; the speed of the connection (USB or Bluetooth); and the CPU load of the computer.

XSENSOR X3 sensors are sampled by polling the hardware for each frame. Basically, the frames are pulled from the X3 sensor hardware by the computer software. This means `XS_OpenConnection` succeeds regardless of the target framerate, but the X3 sensors can only sample at a maximum framerate depending on the previously mentioned limiting factors.

XSENSOR X4 sensors sample themselves at fixed rates and the frames are pushed by the X4 to the computer. The X4 sensor can refuse a too high framerate, and this can cause `XS_OpenConnection` to fail. Note that connection factors can still limit the framerate below what the X4 can handle. Noisy Bluetooth (Wi-Fi included) areas can cause slowdowns in frame transmission, leading to buffer overruns on the X4.

Sampling and Pressure Frames

The `XS_Sample` function is called for each new frame requested. When a new frame is available for all configured sensors, then those frames are copied into a DLL memory buffer and the associated sample timestamp is updated with the computer's current time. The `XS_Sample` function either blocks until a new frame is available or returns immediately depending on the major operating mode of the DLL (described below.)

When new pressure frames are available, they are retrieved with the `XS_GetPressure` or `XS_GetPressure`Safe functions.

`XS_GetPressure` retrieves the pressure data for a single sensor in the form of an array of 32-bit floating point values (IEEE 754).

Pressure in the array is stored in row-major order. This means for a sensor with R rows and C columns, the first C values in the array correspond to the first row of the sensor, the next C values correspond to the second row of the sensor, and so on.

Remember, call `XS_Sample` once per frame followed by a call to `XS_GetPressure` for each sensor.

The values returned by `XS_GetPressure` are returned in units of pressure specified by the `XS_GetPressureUnit` function. Pass in a value of type `EPressureUnit` to `XS_SetPressureUnit` to change the returned readings.

The pressure data is expected to be within the calibrated pressure range. The calibration range for a sensor configuration can be determined with a call to `XS_GetConfigInfo`. This returns the minimum and maximum pressure supported by the calibration range across all sensors in the configuration. It is possible to have pressure values that go outside the calibrated range. However, those values are extrapolations and are of questionable accuracy.

The data returned is also affected by the noise threshold value. Any pressure readings below the noise threshold are set to zero. By default, the noise threshold value matches the minimum pressure of the calibration pressure range. The current value can be retrieved with `XS_GetNoiseThreshold`. The noise threshold can be changed with `XS_SetNoiseThreshold`.

Usage Epilogue

When sampling is completed, call `XS_CloseConnection` to close all open sensors. This allows the sensors to be used by other software.

Finally, call `XS_ExitLibrary` to ensure all library resources are properly released.

OPERATING MODES

The operational behavior of the `XS_Sample` function changes depending on how the DLL library is initialized and whether the sensor configuration is utilizing the XSN session file for data recording.

When `XS_InitLibrary` is called with the `bThreadMode` parameter set to `xTRUE`, the function `XS_Sample` behaves in an asynchronous manner. Calls to `XS_Sample` return immediately, however, there may or may not be a new sample ready depending on how fast each subsequent `XS_Sample` call is made and how fast the sensors are sampling. A `XS_GetSampleTime*` function should be called, and the timestamp checked against the previous timestamp to know if the frame has changed.

When the `XS_InitLibrary` is called with `bThreadMode` set to `xFALSE`, the `XS_Sample` function waits until a new frame arrives from all configured sensors.

When a configuration is made without the use of an XSN session file, only one frame is held by the DLL at a time. The framerate of samples gathered is affected by how frequent `XS_Sample` is called. The caller of `XS_Sample` must call frequently enough to ensure the target framerate is achieved.

When an XSN session is used in the configuration, the `XS_InitLIbrary bThreadMode` parameter affects the behavior of frames recording to the file. If `bThreadMode` is set to `xFALSE`, then each call to `XS_Sample` causes a new frame to be record to the file. The framerate is dependent on the recording speed of the system and how frequently `XS_Sample` is called. If `bThreadMode` is set to `xTRUE`, pressure frames are record to the session file at the achievable framerate, regardless as to how often `XS_Sample` is called. In that case, the `XS_Sample` function retrieves the most current recorded frame.

The DLL also has an X4 operating mode for remote connections. See the **X4 Remote Connection Management** section for more information.

X4 SPECIFIC CONFIGURATION

The X4 sensor can convert its digital readings into 8-bit samples. There is a slight loss of precision in 8-bit mode. However, the benefit is to allow higher transmission rates over Bluetooth connections. Use `XS_SetX4Mode8Bit` to enable or disable this mode. Changes to this mode only take effect before a connection is open.

The X4 foot sensors can collect IMU data with each frame. Use `XS_SetEnableIMU` to enable or disable this mode. Frame IMU data can then be retrieved via `XS_GetIMU`.

X3 PRO HUB – External Sync Port

The X3 PRO hub has a synchronization port that is used to synchronize data collection with an external signal. The hub allows two modes of synchronization: “trigger start of streaming” and “trigger single frame”. Only one trigger mode can be enabled at a time. This hub is for X3 sensors only and is not compatible with configurations containing X4 sensors.

While a trigger mode is enabled, the `XS_Sample` function blocks until the X3 PRO hub is signaled or the read-timeout timer expires. This blocking behavior overrides any DLL threading modes while the trigger is active.

When in the “trigger start of streaming” mode, the DLL waits for an external signal to start recording pressure frames. After a signal arrives the trigger is disabled, and the DLL begins recording normally. The DLL ignores all subsequent signals from the hub. This mode is enabled with the `XS_EnableRecordTrigger_FirstFrameOnly` function.

When in the “trigger single frame” mode, the DLL waits for an external signal to capture a single frame. After a signal arrives, a single pressure frame is captured and the DLL resumes waiting for another external signal. This mode is enabled with the `XS_EnableRecordTrigger_EachFrame` function.

Either trigger mode can be enabled, disabled or reset at any time.

The operational specs of the external sync port:

- The sync connector is a 3.5mm stereo (TSR) jack. Position 1 is GND, position 2 is the sync voltage. The voltage is LV TTL (0 to 3.6V max). Higher voltages are possible by using an external series resistor inline with the sync voltage. The equation to calculate the series resistor value:
    - Rseries = Vsync x 75 - 264

X4 Remote Connection Management

This section coming soon!

API DESCRIPTION

The most up-to-date API description is found in the XSCore90.h file included in the DLL distribution package.

This API listing is grouped by functional category.

Library Initialization / Deinitialization

```
XBOOL XS_InitLibrary(XBOOL bThreadMode)

       Initialize the XS software engine. This must be the first call to the XS Library.
       When bThreadMode is xTRUE, the DLL samples independently of any calls to XS_Sample.
       When bThreadMode is xFALSE, the DLL waits for each XS_Sample call to fetch a new sample.
       Returns xTRUE if the call succeeded.
```

```
XBOOL XS_ExitLibrary()

       Uninitializes the XS software engine. Frees all allocated resources.  This should be the
       last call to the XS Library.
```

```
XBOOL XS_GetVersion(uint16_t& major, uint16_t& minor, uint16_t& build, uint16_t& revision)

       Fetch the DLL version - major.minor.build.revision.
       Returns xTRUE if the call succeeded.
```

```
EXSErrorCodes XS_GetLastErrorCode()

       Retrieves the last error code. Check this value when a function reports a failure. (See
       enum EXSErrorCodes in XSCore90.h)
```

```
const char* XS_GetLastErrorCodeAsString()

       Returns the last error code in the form of an ASCII English string.
```

Sensor Enumeration

```
uint32_t XS_EnumSensors()

        Scans the computer for all possible sensor connections and builds a list of connected
        sensors.

        This function attempts to open a temporary connection to each candidate sensor. Some
        connections such as Bluetooth can take several seconds per candidate. The enumeration
        process can take between 5 and 180 seconds, depending on what type of sensors have been
        selected for enumeration and the general state of the computer.

        Any sensors connected to a computer after an initial XS_EnumSensors call will not be seen
        until a subsequent call to XS_EnumSensors is made.

        Returns the number for found sensors.


void XS_SetAllowEnumInterruptX4Remote(XBOOL allow)
XBOOL XS_GetAllowEnumInterruptX4Remote()

        XS_EnumSensors normally interrupts the remote recording session of any X4 sensors found.
        This behavior is required to query the X4 sensor for its information. When the DLL is
        used to establish a remote record, its preferable for XS_EnumSensors to not interrupt the
        remote sessions.

        Set this to xFALSE to prevent XS_EnumSensors from interrupting any active remote
        recordings.


void XS_SetAllowFastEnum(XBOOL bAllow)
XBOOL XS_GetAllowFastEnum()

        When this flag is enabled, the DLL skips looking for new sensors and only attempts to
        talk to sensors it already knows about. Temporarily enable this flag this when scanning
        for a known sensor whose connection was lost. This avoids the expense of looking for new
        sensors.


EEnumerationError XS_GetLastEnumState()

        Retrieves an error bit mask indicating why XS_EnumSensors or XS_AutoConfig* returned 0.
        One or more errors may have occurred and are OR'd into a bitmask. See enum
        EEnumerationError for details.


const char* XS_GetLastEnumStateAsString()

        Retrieves an internal English string describing the errors.
```

```
uint32_t XS_EnumSensorCount()

       Returns a count of the number of sensors found during the XS_EnumSensors() call.
```

```
SENSORPID XS_EnumSensorPID(uint32_t nEnumIndex)

       Returns a sensor's 64-bit product ID at the indexed (0-index) location of the enumerated
       sensors list. If the return value is 0, the function call failed. Failure is typically
       caused by an index that is out of bounds.
```

Sensor Details

```
uint16_t XS_GetSerialFromPID(SENSORPID spid)

       Returns a sensor's 4 digit serial number (ie: 0014)
```

```
XBOOL XS_GetSensorDimensions(SENSORPID spid, uint16_t& nRows, uint16_t& nColumns)

       Fetch the number of rows and columns defining the sensor grid.
       Returns xTRUE if the call succeeds.
```

```
XBOOL XS_GetSensorName(SENSORPID spid, uint32_t& charCount, wchar_t* pBuffer = NULL)

       Fetch the display name of the sensor. Example: HX210.11.31.M9-LF S0002

       The string is made of 16 bit wchar_t characters (UTF-16/UCS-2) which is null terminated.

       If pBuffer is 0, the length of the required buffer (in wchar_t character counts) is
       returned in bufferSize. This count includes space for a null terminator character.  (Byte
       size of the buffer should be at least 2 x charCount).

       If pBuffer is valid, then charCount is expected to be the number of wchar_t that pBuffer
       can hold (including the zero terminator).

       Returns xTRUE if the call succeeds.
```

```
XBOOL XS_GetSenselDims(SENSORPID spid, float& nWidthCM, float& nHeightCM)

       Fetch the physical dimensions of a single cell in centimeters. This function is only
       usable after a sensor configuration is created.
       Returns xTRUE if the dimensions could be fetched.
```

```
XBOOL XS_IsX4Sensor(SENSORPID spid)

       Returns xTRUE if this an X4 sensor.
```

```
XBOOL XS_IsX4FootSensor(SENSORPID spid, XBOOL& bFootSensor, XBOOL& bLeftFoot)

       Returns xTRUE if this an X4 sensor.
       Sets bFootSensor to xTRUE if the sensor is an X4 foot sensor.
       Sets bLeftFoot to xTRUE if it’s a left foot sensor, and xFALSE for right foot sensors.
```

Calibration Management

```
XBOOL XS_SetCalibrationFolder(const wchar_t* szCalibrationFolder)

       When the DLL is provided with a folder for caching calibration files, it will
       automatically manage downloading and selection of an appropriate calibration file. While
       enabled, the other calibration file functions are not enabled.

       Typically, this is the best option for managing calibration files.
       Returns xTRUE if the call succeeds.

       Alternatively, the use the following functions for direct management of a sensor's
       calibration files.

uint32_t XS_DownloadCalibrations(SENSORPID spid, const wchar_t* szCalibrationFolder)

       Downloads the calibration files stored on the sensor and saves them to a folder.
       Returns number of files downloaded. There may be 0, 1 or more calibration files on the
       sensor. If 0 is returned, call XS_GetLastErrorCode to determine if an error occurred.

       Usually When more than one file is present, each file is a different calibrated pressure
       range.


XBOOL XS_GetCalibrationName(uint32_t nCalIndex, uint32_t& dwBufferSize, wchar_t* pBuffer=NULL)

       This function retrieves the full file path and file name of the indexed calibration file.
       The index is 0-based.

       This function only works after a calling XS_DownloadCalibrations, and that function
       returns a count of 1 or more.

       If pBuffer is NULL, the size of the required buffer (in bytes) is returned in
       dwBufferSize. This includes space for a null terminator.
       Returns xTRUE if the call succeeds.


XBOOL XS_GetCalibrationInfo(const wchar_t* szCalibrationFile, uint8_t nPressureUnits, float&
nMinPressure, float& nMaxPressure)

       Scans the calibration file and returns its pressure range (using the pressure units - see
       enum EPressureUnit).
       szCalibrationFile is the full path to the file on disk. eg: c:\\CalFiles\\MyCalFile.xsc
       Returns xTRUE if the call succeeds.



XBOOL XS_GetCalibrationInfoEx(const wchar_t* szCalibrationFile, SENSORPID& spid)

       Retrieves the SENSORPID associated with the calibration file.
       szCalibrationFile is the full path to the cal file on disk.  eg:
       c:\\CalFiles\\MyCalFile.xsc
       Returns xTRUE if the call succeeds.
```

Sensor Configuration

```
void XS_SetPressureUnit(uint8_t PressureUnit)

       Sets the calibrated data's pressure units (see enum EPressureUnits). Pressure readings
       are presented in these units. These can be changed at any time.


uint8_t XS_GetPressureUnit()

       Returns the current pressure units (see enum EPressureUnits).


XBOOL XS_HasConfig();

       Returns xTRUE if the DLL has a sensor configuration.


XBOOL XS_ReleaseConfig();

       Closes any open connection and releases the configuration (including any XSN file).
       Subsequent calls to XS_OpenConnection will fail until a new config is created.
```

Sensor Configuration - AUTOMATIC

The automatic sensor configuration functions simplify the process of configuring any connected sensors.

```
XBOOL XS_AutoConfigByDefault()
XBOOL XS_AutoConfigByDefaultXSN(const wchar_t* szXSNFile)

       Constructions a sensor configuration using all found sensors. Default calibration
       pressure ranges are utilized. This function sets the pressure units to ePRESUNIT_MMHG! Be
       sure to call XS_SetPressureUnit() after calling this function.
       Returns xTRUE if the call succeeds.

       XSN variant function takes a full file path and file name to construct an XSENSOR session
       file from all recorded frames.

       Example:
         wchar_t* szPath[] = L"c:\\mysessions\\session5.xsn";
              XS_AutoConfigByDefaultXSN(szPath);



XBOOL XS_AutoConfig(uint8_t pressureUnits = ePRESUNIT_MMHG, float targetPressure = -1.0f)

       Constructions a sensor configuration using all found sensors.
       If targetPressure is -1.0, then the default calibration pressure range is used.
       If targetPressure is >= 0, then the nearest matching pressure range is used.
       Sets the calibrated data's pressure units (see enum EPressureUnits). Pressure readings
       are presented in these units.
```

```
XBOOL XS_AutoConfigXSN(const wchar_t* szXSNFile, uint8_t pressureUnits = ePRESUNIT_MMHG, float
targetPressure = -1.0f)

        The XSN variant of XS_AutoConfig function takes a full file path and filename and
        constructs an XSENSOR session file from all recorded frames.
```

```
XBOOL XS_AutoConfig_SingleSensor(const wchar_t* szCalFile=NULL)
XBOOL XS_AutoConfig_SingleSensorXSN(const wchar_t* szXSNFile, const wchar_t* szCalFile=NULL)

        These XS_AutoConfig variants allow manual selection of the calibration file. This only
        works if there is a single sensor present.
```

Sensor Configuration – MANUAL

These functions give better control over the configuration of any connected sensors.

```
XBOOL XS_NewSensorConfig()

        Prepares the library for creating a configuration of sensors. Remove any existing sensor
        configuration.
        Returns xTRUE if the call succeeds.
```

```
XBOOL XS_NewSensorConfigXSN(const wchar_t* szXSNFile)

        Similar to XS_NewSensorConfig(), but also creates a standalone XSN session file.
        The szXSNFile file path must be fully specified. eg: "c:\MySessions\Test.xsn".
        Returns xTRUE if the call succeeds.
```

```
XBOOL XS_AddSensorToConfig(SENSORPID spid, const wchar_t* szCalibrationFile)

        Adds a sensor to the sensor configuration. The full path to the sensor's calibration file
        should be specified in szCalibrationFile.
        If szCalibrationFile is 0, then only raw data is collected.
        Returns xTRUE if the call succeeds.

        NOTE: All sensors in the configuration must have calibration files, otherwise only raw
        values are collected.
```

```
XBOOL XS_AddSensorToConfig_AutoCal(SENSORPID spid, uint8_t pressureUnits, float
targetMaxPressure)

        Adds a sensor to the sensor configuration. Will download any calibration files from the
        sensor if they are not present. Attempts to use the nearest matching calibration pressure
        range.
        Returns xTRUE if the call succeeds.

        NOTE: The calibration download process can be slow. Be sure to set a folder for caching
        the calibration files using XS_SetCalibrationFolder().
```

```
XBOOL XS_AddSensorToConfig_AutoCalByDefault(SENSORPID spid)

        Similar to XS_AddSensorToConfig_AutoCal(), but attempts to use the default pressure range
        for the sensor.
        Returns xTRUE if the call succeeds.
```

```
XBOOL XS_ConfigComplete()

        When finished adding sensors to the configuration, call this function to initialize the
        configuration for use.
        This function must be called before calling XS_OpenConnection.

        Returns xTRUE if the call succeeds.
```

Sensor Configuration – Details

```
SENSORPID XS_ConfigSensorPID(uint32_t nConfigIndex)

        Returns the sensor PRODUCT ID at the indexed location of the configured sensors list. The
        index is zero based.
        If 0 is returned, check XS_GetLastErrorCode.
```

```
uint32_t XS_ConfigSensorCount()

        Returns the number of configured sensors.
```

```
XBOOL XS_IsCalibrationConfigured()

        Returns xTRUE if the configuration is setup for calibrated data.
```

```
XBOOL XS_GetConfigInfo(uint8_t pressureUnits, float& nMinPressure, float& nMaxPressure)

        After configuration is complete, call this to check the pressure range of the
        configuration.

        pressureUnits - the returned minimum and maximum pressures are returned in these units.
        (See enum EPressureUnits).

        When operating in RAW mode (i.e.: no calibration file specified), pressureUnits is
        ignored.

        Returns xTRUE if the call succeeds.
```

Sampling State

Sampling state values should be called before XS_OpenConnection as changes to them are ignored otherwise.

```
void XS_SetReadTimeout(uint32_t TimeoutSeconds)

       Sets the XS_Sample function's timeout period in seconds.
       If the software loses connection to a sensor, it tries to re-open the connection for this
       length of time.


uint32_t XS_GetReadTimeout()

       Returns the XS_Sample timeout period in seconds.


void XS_SetOverlapMode(XBOOL bOverlap)

       This mode causes the DLL to sample each configured sensor in sequence (versus in
       parallel).
       This mode prevents electrical crosstalk between sensors when they are physically in
       contact.
       NOTE: This mode is not available when X4 sensors are configured.


XBOOL XS_GetOverlapMode()

       Returns xTRUE when Overlap mode is on.  Overlap mode is off by default.


void XS_SetAllowWireless(XBOOL bAllow)
XBOOL XS_GetAllowWireless()

       A flag that allows use of the X3 wireless series.


void XS_SetAllowX4(XBOOL bAllow)
XBOOL XS_GetAllowX4()

       A flag that allows use of the X4 sensor series (wired).


void XS_SetAllowX4Wireless(XBOOL bAllow)
XBOOL XS_GetAllowX4Wireless()

       A flag that allows use of the X4 sensor series (wireless).


void XS_SetX4Mode8Bit(XBOOL bEnable8BitMode)
XBOOL XS_GetX4Mode8Bit()

       A flag that puts the X4 in 8-bit mode.
       - This can improve the reliability of Bluetooth transmission speeds. It has no impact on
       actual recording speed.
```

`XBOOL XS_HasHardwareIMU(SENSORPID spid)`

Determines if the sensor has IMU capability.

`void XS_SetEnableIMU(XBOOL bEnable)`

Enables/disables IMU collection for supported sensors.

`XBOOL XS_GetEnableIMU()`

Indicates whether the IMU enable flag is set. (Does not indicate whether IMU is supported!)

`void XS_SetStreamingMode(XBOOL bStreaming, uint16_t streamingMaxFrameCache)`

Streaming mode buffers all incoming frames and releases them with each XS_Sample() call.
When streaming mode is off, sensors are sampled at the time XS_Sample is called.

NOTE: When streaming mode is on, the DLL will allocate large amounts of buffer memory if the samples are not consumed fast enough.

The streamingMaxFrameCache parameter determines how many frames the streaming mode buffers. This is the cache limit.

If streamingMaxFrameCache is set to zero, then the DLL caches all frames until it runs out of memory.
If streamingMaxFrameCache is non-zero, the DLL discards older frames once the cache limit is reached.

`XBOOL XS_GetStreamingMode()`

Returns xTRUE if streaming mode is on, xFALSE otherwise. This is off by default.

`void XS_SetSampleAverageCount(uint16_t nCount)`

Set the number of per-sensel readings that are averaged together per sensel scan.

This averaging occurs within the sensor's DSP hardware. Higher cycles lead to slower frame rates but may produce more accurate readings.

Valid X3 numbers are 1, 2, 4, 8 and 16.
X4 sensors can only run at 4 and 8 cycles. X4 foot sensors run fastest with 4 cycles.

This value can only be set before calling any of the XS_AutoConfig* functions or XS_ConfigComplete().

Typically, this value does not need to be set as the sensor is normally programmed with an expected default.

`uint16_t XS_GetSampleAverageCount()`

Get the number of per-sensel readings that are averaged together per sensel scan.

Connection Management

```
XBOOL XS_OpenConnection(uint32_t nThreadedFramesPerMinute=0)

        Opens all configured sensors and begins sampling.
        Call this only after calling XS_ConfigComplete() or XS_AutoConfig*().

        nThreadedFramesPerMinute
        - When using the DLL in threaded mode, this specifies the target framerate in frames per
        minute. eg: 600 FPM = 10 Hz

        Returns xTRUE if the call succeeds.
```

```
XBOOL XS_IsConnectionOpen()

        returns xTRUE if the connection is open.
        Typically, this function returns xFALSE when a sensor is disconnected, or a sensor is in
        use by another program.
```

```
XBOOL XS_CloseConnection()

        Stops all sampling and closes the sensor connections.
        If the sensor configuration is using an XSN file, then that file stays open.
        Subsequent calls to XS_OpenConnection continue to write to the same XSN file.
        To close the XSN file, call XS_ReleaseConfig.
```

```
XBOOL XS_IsConnectionThreaded()

        Returns xTRUE if the connection is open and the connection is running in an asynchronous
        threaded mode.
```

Sampling

```
XBOOL XS_SetNoiseThreshold(float threshold)

       Any pressure readings below this noise threshold are zeroed. The threshold is applied to
       all configured sensors.

       The default threshold is normally the lowest calibrated pressure of all configured
       sensors.

       threshold - The threshold should be in the active pressure units. (see
       XS_GetPressureUnit)


float XS_GetNoiseThreshold()

       Returns the threshold below which pressure is zeroed.
       If -1.0 is returned, this indicates the function failed. Check XS_GetLastErrorCode.


XBOOL XS_Sample()

       Ask the configured sensors to record a new sample to an internal buffer.
       Use XS_GetPressure() to retrieve the pressure map from the internal buffer.

       When using the library in a threaded mode, this call returns immediately. However, the
       internal buffer may or may not have a new sample as it only updates when a new one is
       available. Check the timestamp to see if it has changed.

       When using the library in an un-threaded mode, this call blocks until a sample is
       recorded or a read timeout occurs.

       Returns xTRUE if the call succeeds. Note this does not necessarily mean there is a new
       sample ready. Be sure to check timestamps!


XBOOL XS_GetSampleTimestampUTC(
       uint16_t& year, uint8_t& month, uint8_t& day,
       uint8_t& hour, uint8_t& minute, uint8_t& second, uint16_t& millisecond)
       Fetch the timestamp of the last sample collected.
       Components are normal UTC ranges. month (1-12), day (1-31). Hour is (0-23), minute and
       second (0-59), millisecond (0-999)

XBOOL XS_GetSampleTimeUTC(SYSTEMTIME & tTime)
       Fetch the timestamp of the last sample collected.
       Similar to XS_GetSampleTimestampUTC, but using the Win32 structure SYSTEMTIME.

XS_GetSampleTimestampExUTC(
       uint16_t& year, uint8_t& month, uint8_t& day,
       uint8_t& hour, uint8_t& minute, uint8_t& second, uint16_t& millisecond, uint16_t&
microsecond, bool bAsLocalTime)
       Fetch the timestamp of the last sample collected.
       Includes microseconds when supported by the sensor hardware. This version allows
       conversion of the timestamp to the local time zone.
```

```
XBOOL XS_GetPressure(SENSORPID spid, float* pData)

       Retrieves the calibrated sample data for a single sensor.

       (Call XS_Sample once for all sensors before calling XS_GetPressure for each sensor.)

       The pressure readings are in the units set by XS_SetPressureUnit() or XS_AutoConfig().

       - pass in a pre-allocated buffer (pData) which is larger enough to hold (rows X columns)
       elements, each 32-bits.
       - rows and columns are the dimensions of the sensor.
       - The array is organized in a typical row-major order. IE: The first N values in the
       array correspond to the first row, where N is the number of columns.


       Returns xTRUE if the call succeeds.


XBOOL XS_GetPressureSafe(SENSORPID spid, uint32_t dataCount, float data[])

       This version takes a pre-allocated buffer of floats (32-bit).
       dataCount is the number of elements in the data[] array.

       Returns xTRUE if the call succeeds.



XBOOL XS_GetRaw(SENSORPID spid, uint16_t* pData)

       Similar to XS_GetPressure(), except it takes an array of uint16_t and returns
       uncalibrated raw 16 bit values from the sensor.

       Returns xTRUE if the call succeeds.

XBOOL XS_GetRawSafe(SENSORPID spid, uint32_t dataCount, uint16_t data[])

       This version takes a preallocated buffer of uint16_t (16-bit).
       dataCount is the number of elements in the data[] array.
       Returns xTRUE if the call succeeds.



XBOOL XS_GetIMU(
       SENSORPID spid,
       float& qx, float& qy, float& qz, float& qw,
       float& ax, float& ay, float& az,
       float& gx, float& gy, float& gz)

       Fetches the IMU data (if any) for the current Sample
```

```
XBOOL XS_GetHardwareFrameState(
	SENSORPID spid,
	uint32_t& sequence, uint32_t& ticks)

	Retrieves the hardware frame header state.
	-   'sequence' increments for each frame.
	-   'ticks' is milliseconds for X4
	Returns 0 if there is no hardware state, or an error has occurred


float* XS_AllocPressureBuffer(SENSORPID spid)

	Allocate a buffer for accessing the pressure values. Use this for multiple readings.
	The DLL will allocate enough memory to hold all rows and columns of the sensor.
	Returns 0 if the SENSORPID is invalid or the system is low in memory.

void XS_ReleasePressureBuffer(float* pBuffer)
	Releases the DLL allocated buffer.

uint16_t * XS_AllocRawBuffer(SENSORPID spid)

	Allocate a buffer for accessing the raw values. Use this for multiple readings.
	The DLL will allocate enough memory to hold all rows and columns of the sensor.
	Returns 0 if the SENSORPID is invalid or the system is low in memory.

void XS_ReleaseRawBuffer(uint16_t * pBuffer)
	Releases the DLL allocated buffer.

XBOOL XS_GetDeadMap(SENSORPID spid, uint8_t* pData)

	Maps out the sensels that are dead and alive.  1 = dead, 0 = alive
	Only profiles that mask out dead cells in the grid will have this. (ie: X4 foot sensors)
	"uint8_t* pData" should be allocated with space for ROWS x COLUMNS for the target sensor.

	The buffer is in row-major order. (IE: The first N values are from row 1, where N equals
	the number of columns. The next N values are from row 2. Etc.)
```

Frame Statistics

```
XBOOL XS_CenterOfPressure(SENSORPID spid, float zeroThreshold, float& column, float& row)
```

```
Computes the center of pressure (COP) of the last sample recorded.

The coordinates have negative values if there is a problem. Call XS_GetLastErrorCode() if
that occurs.

The center of pressure calculation ignores pressure which is below the zero threshold.
zeroThreshold must be in the same pressure units as XS_GetPressureUnits().

Row coordinates are from 1 to row count.
Column coordinates are from 1 to column count.
Row and column counts are retrieved with XS_GetSensorDimensions().

Whole numbers (no decimal fractions) indicate the COP is over the center of the sensel.
The fractional part of the coordinate indicates a normalized distance between the centers
of two sensels.

Example:
Suppose the function returns these values: column = 1.53f; row = 2.25f. This means the
COP is 53% of the distance between the center of column 1 and the center of column 2. The
COP is 25% of the distance between the center of row 2 and the center of row 3. (It is
closer to row 2).
```

Utility Functions

```
XBOOL XS_WideToUTF8(const wchar_t* wide, uint32_t & bufferSize, char* utf8)

       Converts a wide string (wchar_t) to UTF8. You must supply a UTF8 buffer that is large
       enough to hold the converted string. The string is expected to be null (zero) terminated.

       const wchar_t* wide
              - the wide string to convert

       uint32_t& bufferSize
              - returns the size of the buffer (in byte count) needed to hold the conversion.
              - If utf8 is not null, pass in the utf8's buffer size in this field to allow for
              an extra buffer safety check.

       char* utf8
              - the buffer to hold the utf8 string. Pass in nullptr (or 0) to retrieve just the
              size of the buffer (in bytes) required in bufferSize.
```

```
XBOOL XS_UTF8ToWide(const char* utf8, uint32_t & bufferSize, wchar_t* wide)

       Converts a UTF8 (or plain ASCII) string to wide format (wchar_t). You must supply a
       wchar_t* buffer that is large enough to hold the converted string. The string is expected
       to be null (zero) terminated.

       const char* utf8
              - the utf8/ASCII string to convert

       uint32_t& bufferSize
              - returns the size of the buffer (in byte count) needed to hold the conversion.
              wchar_t uses 2 bytes per character.
              -  (eg: If bufferSize == 100, then  wchar_t wide[50]  will hold the string.
              - If wide is not null, pass in the wide's buffer size (in bytes) in this field to
              allow for an extra buffer safety check.

       wchar_t* wide
              - the buffer to hold the wide string. Pass in nullptr (or 0) to retrieve just the
              size of the buffer (in bytes) required in bufferSize.
```

X3 PRO HUB – External Sync

```
XBOOL XS_IsRecordTrigger_FirstFrameOnly()

       Returns xTRUE if this trigger is enabled.
```

```
void XS_EnableRecordTrigger_FirstFrameOnly(XBOOL bEnable)

       The “FirstFrameOnly” trigger blocks the XS_Sample() command until the XS PRO hub is
       signaled on the external sync port.  Subsequent calls to XS_Sample() return immediately
       with a collected sample.
```

```
XBOOL XS_IsRecordTrigger_EachFrame()

       Returns xTRUE if this trigger is enabled.
```

```
void XS_EnableRecordTrigger_EachFrame(XBOOL bEnable)

       The "EachFrame" trigger blocks the XS_Sample() command until the XS PRO hub is signaled
       on the external sync port.  Subsequent calls to XS_Sample() also block until the port is
       signaled.
```

X4 Remote Connection Management

These functions allow starting & stopping remote X4 recordings (made to the onboard SD card), as well as managing the files.

These functions should not be used in the context of live sessions created with XS_OpenConnection().

`XBOOL X4_OpenRemote(SENSORPID spid)`

Opens a remote connection to an X4 sensor. Can fail if the sensor doesn't not have an SD card, check error codes.

The initial open call may take a while as the DLL downloads (and caches) various sensor details on the first call.

`XBOOL X4_CloseRemote(SENSORPID spid)`

Closes a remote connection to an X4 sensor.

`XBOOL X4_IsRemoteRecording(SENSORPID spid, XBOOL* remoteRecording)`

Is the X4 currently recording remotely?
Sets remoteRecording to xTRUE if it is.

`XBOOL X4_StartRemoteRecording(SENSORPID spid, float framesPerSecond)`

Starts the X4's remote recording.
framesPerSecond – allows for a maximum of 160 Hz recording to the SD card.

`XBOOL X4_StopRemoteRecording(SENSORPID spid)`

Stops the X4's remote recording. This ends the active session.

`XBOOL X4_StartRemotePairRecording(SENSORPID spid_left_insole, SENSORPID spid_right_insole, float framesPerSecond)`

Starts a synchronized recording for two X4 foot sensors (left and right insoles only)

`XBOOL X4_StopRemotePairRecording(SENSORPID spid_left_insole, SENSORPID spid_right_insole)`

Stops a synchronized recording

```
XBOOL X4_SampleRemote(SENSORPID spid)
```

Fetches a preview frame from the sensor. Call XS_GetPressure() to get the frame. Only available when a calibration file is cached.

Be careful to limit the calls to this as it may interfer with the remote recording.

```
XBOOL X4_GetRemoteDuration(SENSORPID spid, float framesPerSecond, uint32_t* minutes)
```

For a given recording speed, returns the available duration of recording in total minutes. This is based on SD card space and not battery life.

The FOLLOWING have not yet been implemented.

```
XBOOL X4_FetchRemoteSessionList(SENSORPID spid, uint32_t & sessionCount)
```

Retrieves the list of sessions currently on the remote sensor. The list is held internally by the DLL. Check sessionCount for a non-zero value.

```
XBOOL X4_AccessRemoteSessionInfo(
        SENSORPID spid, uint32_t sessionIndex,
        uint32_t & fileSize,
        bool& isPairedSession,
        uint32_t & nameBufferSize,
        char* sessionName)
```

Accesses the retrieved internal session list via a 0-based session index.

uint32_t& fileSize
- indicates the size of the session file in bytes

bool& isPairedSession
- indicates if the session is expected to be paired with another sensor's remote session. -XTK#### indicates the common pairing token between the two sessions.

uint32_t& nameBufferSize
- returns the expected buffer size (in bytes) to hold the session name. Pass in the known buffer size of the name buffer for extra safety.

char* sessionName
- the buffer to hold the session name (utf8/ascii).
- pass in nullptr (or 0) to just retrieve the size of the required buffer in nameBufferSize
- when passing in the buffer, set nameBufferSize to the size of the buffer for an extra buffer safety check

```
XBOOL X4_DownloadRemoteSession(
        SENSORPID spid,
        const char* sessionName,
        const char* sessionFolder,
        bool bDeleteRemoteOnSuccess,
        uint8_t * progressPct);

        Downloads the named session to the target folder. The DLL will attempt to create the
        folder if it doesn't exist.

        NOTE: The operation must be done within the context of a fetched session list!

        const char* sessionName
               - the session name as found in the session list

        const char* sessionFolder
               - a folder on the PC. The string is in UTF8 format. If you have a wchar_t string,
               use XS_WideToUTF8 to convert it.

        bool bDeleteRemoteOnSuccess
               - set true to delete the remote session file if the download succeeds.

        uint8_t* progressPct
               - (optional) an integer percentage of download completion. On error or issue, the
               value is set to 255.

        The downloaded session is in the original .x4r format. Use X4_ConvertSession or
        X4_ConvertSessionPair to generate an XSN file.

        This function may take awhile and may be run on its own thread, however don't call any
        other DLL functions while running the threaded function.

        NOTE: See X4_ConvertSession() comments for an example of threaded use.


XBOOL X4_DeleteRemoteSession(SENSORPID spid, const char* session);

        Deletes the named remote session from the SD card.

        NOTE: The operation must be done within the context of a fetched session list!
```

```
XBOOL X4_ConvertSession(const char* sessionX4R, const char* folderXSN, uint8_t * progressPct);

       Converts an X4R session file into a XSN session.

       const char* sessionX4R
              - full path and file name of the X4R session file.

       const char* folderXSN
              - the folder where the XSN file will be constructed. The DLL will attempt to
              create the folder if it doesn't exist.
              - the constructed XSN file has the same name as the X4R file with a different file
              extension.

       uint8_t* progressPct
              - (optional) an integer percentage of download completion. On error or issue, the
              value is set to 255.

       This function may take a while and may be run on its own thread. However, don't call any
       other DLL functions while running the threaded function.

// Example Win32 Threaded Usage:
struct ConvertSession
{
       ConvertSession()
       {
              sessionX4R = nullptr;
              folderXSN = nullptr;
              progressPct = 0;
              eLastError = EXSErrorCodes::eXS_ERRORCODE_OK;
              result = xFALSE;
              bRunning = false;
       }
       const char* sessionX4R;
       const char* folderXSN;
       uint8_t progressPct;
       EXSErrorCodes eLastError;
       XBOOL result;
       bool bRunning;
};

UINT __stdcall THREAD_ConvertSession(LPVOID lpParam)
{
       ConvertSession* session = (ConvertSession*)lpParam;

       session->bRunning = true;
       session->result = X4_ConvertSession(session->sessionX4R, session->folderXSN, &(session-
>progressPct));

       session->eLastError = XS_GetLastErrorCode();
       session->bRunning = false;

       return 1;
}
```

```
// example continues…

#include <thread>
#include <assert.h>

void MyFunc()
{
	ConvertSession session;
	session.sessionX4R = "c:\\mysessions\\x20210621-175348927-HX2101131M9RF-S0013.x4r";
	session.folderXSN = "c:\\mysessions\\processedXSN";
	session.bRunning = true;

	std::thread myThread = std::thread(THREAD_ConvertSession, &session);

	if (myThread.joinable())
	{
		// do something else or we can do this...
		while (session.bRunning)
		{
			Sleep(10);  // sleep 10 milliseconds - don't starve the CPU!

			if(session.progressPct != 255) // 255 is an error condition!
				printf("Conversion progress = %ld %%\n", session.progressPct);
		}

		myThread.join();  // terminate the thread

		if(session.result && (session.eLastError == EXSErrorCodes::eXS_ERRORCODE_OK))
		{
			// conversion was successful!
			assert(session.progressPct == 100);
		}
		else
		{
			 // something went wrong! Check the error code session.eLastError.
		}
	}
	// else the thread failed to run!?!
}
// End example threaded usage
```

```
XBOOL X4_ConvertSessionPair(const char* sessionLeftX4R, const char* sessionRightX4R, const char*
folderXSN, uint8_t * progressPct);
```

	Similar to X4_ConvertSession. Expects two paired sessions for the left and right insoles.
	The two X4R session names will share a common -XTK#### token.

	The resulting single XSN file is named after the left insole session file.

	See X4_ConvertSession for example usage.