signotec e-signature solutions
SOFTWARE
SOLUTIONS
signoPAD Api
signotec e-signature solutions

Guide for Developers
signoPAD API Linux

Software components for communication with signotec Sigma (Evolis Sig100), Omega (Evolis Sig200), Gamma (Evolis SigActiv) and Alpha LCD pads

This product and its documentation are manufactured by ©signotec GmbH and distributed by ©Evolis

Contents

Legal notice

Models Matching Table

Evolis Model Name	Signotec Model Name
Evolis Sig100 Lite	Sigma Lite
Evolis Sig100	Sigma
Evolis Sig200	Omega
Evolis SigActiv	Gamma

Document History

Evolis Document Name and Version	Signotec Document Name and Version
Guide_SIGAPILI_20210105_ENG	signoPAD-API_linux_documentation_EN_v.1.3

1 Function overview

The signoPAD API contains a non-visual interface, allowing programmers to implement a wide range of functions for capturing electronic signatures and displaying graphics, text and buttons on an Evolis LCD pad in their own programs.

The following table provides an overview of the components included in the signoPAD API.

File name	Short description	Version
libSTPadLib.so	Non-visual native library for activating the Sigma, Omega, Gamma and Alpha model types.	8.2.1.16
libSTCPImageEngine.so	Non-visual native library with an interface for the CImg template from http://cimg.eu.	-
property.ini	Control file to set different kind of parameters for the pad communication. In addition, debug logging can be activated for the components listed above.	-
Sample application	Application and source code in C++ to demonstrate the functions of the STPad components.	-

2 System requirements

2.1 signoPAD API components for Linux

The signoPAD API for Linux can be run on all Linux versions from Kernel 3.19.0 onwards. It was tested under the following systems and development environments:

- Ubuntu 15.04
- Eclipse Luna 4.4.2

2.1.1 Dependencies

- libX11
- libusb-1.0
- libpthread
- libtiff
- libpng12
- libcairo
- libpangocairo-1.0

2.2 signoPAD API components for Java

Please use signoPAD API Java, which you can download for free on our website.

2.3 signoPAD API components for Windows

Please use signoPAD API Windows, which you can download for free on our website.

3 General information on the signoPAD API

3.1 32- and 64-bit variants of the signoPAD API

The signoPAD API is available in both x86 (32-bit) and x64 (64-bit).

The x86 version only contains components and applications that were compiled for the x86 platform.

The x64 version only contains components and applications that were compiled for the x64 platform. All components and applications can only be used on 64-bit versions of Linux.

Since the two versions of the components differ neither in name nor in the interface, it does not matter which one is used for development purposes. But the appropriate component for the present target platform must be used in the implementation. The following table shows which version of the components must be used for specific operating system or application versions:

Operating system	Application	Component
x86 (32 Bit)	x86 (32 Bit)	x86 (32 Bit)
x64 (64 Bit)	x64 (64 Bit)	x64 (64 Bit)

3.2 libSTPadLib.so

libSTPadLib.so is a native and dynamically loadable library. A C header file (STPadLib.h) is included. Initialisation is performed automatically as soon as the library is activated; before it is unloaded again, the `STControlExit()` method must be called to release resources used internally.

The library must be in the libs search path at runtime. For Ubuntu, the path is: /usr/local/lib.

3.3 libSTCPImageEngine.so

libSTCPImageEngine.so is a native and dynamically loadable library. The library is an interface for the CImg template from http://cimg.eu.

The library must be in the libs search path at runtime. For Ubuntu, the path is: /usr/local/lib.

3.4 Using multiple instances

The components included in the signoPAD API can be instantiated more than once. If multiple instances of a component are used in different memory areas (for example, different programs), these instances are completely independent of each other and there is nothing else to be aware of.

If multiple instances of a component are used in the same memory area, please note the following:

- When `DeviceGetCount()` is called, it is valid for all instances and therefore only needs to be executed in one instance.
- If a connection to a device has already been opened by an instance, only the previously determined value is returned when `DeviceGetCount()` is called in another instance, i.e., no new search is carried out.
- A maximum of eight connections can be opened simultaneously. In general, there is therefore no point in using more than eight instances simultaneously.

3.5 SignData structures

The signoPAD API components can return a captured signature as a **SignData** data structure. This format is an encrypted, compressed, biometric structure that can be stored in a database or as a tag in a TIFF or PDF document.

3.6 Notes for redistribution

You can, of course, redistribute individual files from the signoPAD API in a separate package. Essentially, only the 'STPad' component used by your application, possibly the proterty.ini file, is required to support the Evolis Sigma, Omega, Gamma and Alpha LCD signature pads.

The signoPAD API package consists of the following files:

Component	Installation path
libSTPadLib.so	/usr/local/lib für Ubuntu /usr/lib64 für CentOS x64 /usr/lib für CentOS x86
libSTCPImageEngine.so	/usr/local/lib für Ubuntu /usr/lib64 für CentOS x64 /usr/lib für CentOS x86
STPadLibDemoApp	-
property.h	Next to STPadLibDemoApp

STPadLibDemoApp serves an example of how the signoPAD API should be used. The demo is included with the C++ source code in the signoPAD API package. The source code can be compiled with an Eclipse development environment, and the demo can be subsequently built. It is a simple console program. At the beginning, the demo searches for Evolis LCD pads, and the first pad found is used. The signature process is initiated, and the signature is captured and saved as a PNG image and SignData files.

4 Description of possible error messages

Most of the libSTPadLib.so methods return an integer value, which is always negative in the case of an error. A description of the respective error messages is provided in the following table. By calling the `STControlGetErrorString()` method, you can get an error description at runtime.

The error descriptions are available in German, English, French and Italian.

Error	Description
-1	A NULL pointer was passed.
-3	One of the parameters that were passed contains an invalid value.
-4	The signing pad is already being used by another application.
-5	No connection has been opened to this signature pad.
-6	A connection has already been opened.
-7	No further connections can be opened.
-8	No device with this ID is connected.
-9	The LED colour that was passed cannot be set.
-12	The function could not be executed because the signature capture process is running.
-13	No further hotspots can be added
-14	The coordinates are overlapping with the signature window or one of the hotspots already set.
-15	The function could not be executed because no signature capture area has been set.
-17	The function could not be executed because no signature capture process was started.
-18	An error occurred while attempting to reserve memory.
-19	An error occurred while initialising a system resource.
-20	An error occurred while communicating with the signing pad.
-21	The rectangle that was passed is invalid.
-22	No compatible devices connected or the connection to a device has been cut.
-25	The connected device does not support this function or one of the parameters.
-26	Error while reading or writing a file.
-93	The function could not be executed because an overlay rectangle is set.
-94	The function could not be executed because the display content is scrolled.
-95	The function could not be executed because it would have activated the scroll mode that is not possible if a hotspot outside the overlay rectangle is defined.
-97	An error occurred during initialisation. Please restart the software.

5 Information about the available image memory

The Evolis LCD Signature Pads have several image memory, which can be used by different methods. An image memory has at least the size of the display and can store one picture in a maximum of this size. Adding another image overrides the areas it overlaps with the existing memory content. Adding multiple images to one memory can therefore create a collage.

Depending on the model, a different number of volatile and non-volatile memories are available.

5.1 Volatile image memory

All Evolis LCD Signature Pads have at least two volatile image memories, one foreground buffer containing the current display content and one background buffer, which can be used to prepare the display content. It can be written in both of the buffers.

The content of the volatile image memory is lost when you close the connection to the device.

5.1.1 Model type Sigma

The two volatile image memories have the size of the display (320 x 160 pixels).

The transmission and representation of images is usually so fast that there is no visible lag. For more complex representations that consist of several individual images, it may be useful to first save them in the background buffer before moving them to the foreground buffer.

5.1.2 Model type Omega

The Omega model has three volatile image memories, two that have the doubled size of the display (640 x 960 pixels) to be used as foreground and background buffers and one that has the size of the display (640 x 480 pixels) to be used as overlay buffer. Its contents can be overlaid over the current display content.

The speed of displaying a picture in Omega model depends on the size and content of the images, usually it's visible. Therefore, images should always be stored first in the background buffer and then moved into the foreground buffer.

5.1.3 Gamma model

The Gamma model has three volatile image memories, two that are larger than the display (800 x 1440 pixels) to be used as foreground and background buffers and one that has the size of the display (800 x 480 pixels) to be used as overlay buffer. Its contents can be overlaid over the current screen content.

With the Gamma model, an image is only displayed after it has been transferred; the image composition is not visible. The speed of the image transmission depends on the size and content of the images. For more complex representations that consist of several individual images, it is generally useful to first save them in the background buffer before copying them into the foreground buffer.

5.1.4 Model type Alpha

The Omega model has three volatile image memories, two that are larger than the display (2048 x 2048 pixels) to be used as foreground and background buffers and one that has the size of the display (768 x 1366 pixels) to be used as overlay buffer. Its contents can be overlaid over the current screen content.

With the Alpha model, an image is only displayed after it has been transferred; the image composition is not visible. The speed of the image transmission depends on the size and content of the images. For more complex representations that consist of several individual images, it is generally useful to first save them in the background buffer before copying them into the foreground buffer.

5.2 Non-volatile image memory

Depending on the model, a different number of non-volatile memories are available. The saving of images in non-volatile image memory lasts longer than storing in volatile image memory, but the content remains unchanged even after switching off the device. An intelligent memory management detects whether an image to be stored is already stored in the device so that only the first time it's stored it comes to a delay.

5.2.1 Model type Sigma

The Sigma model has one non-volatile image memory in the size of the display (320 x 160 pixels), which can only be used for the standby image. Due to the rapid transmission and display of pictures, it is not necessary to be able to save other images permanently.

5.2.2 Model type Omega

The Omega model has eleven non-volatile image memories, which can be used for the standby image, the slide show and optimizations of the program. The memories, used for the standby image or the slide show, are read-only and can be freed only by disabling the standby image or the slide show.

One non-volatile image memory has the doubled size of the display (640 x 960 pixels), ten memories have the size of the display (640 x 480 pixels).

To use a non-volatile memory, this must be reserved first. This is done by calling the `STDisplaySetTarget()` method. The size of the currently selected memory can be queried using the `STDisplayGetTargetWidth()` and `STDisplayGetTargetHeight()` methods.

5.2.3 Gamma model

The Gamma model has ten non-volatile image memories, which can be used for the standby image, the slide show and optimisations of the program. The memories, used for the standby image or the slide show, are read-only and can be freed only by disabling the standby image or the slide show.

The ten non-volatile memories are the same size as the display (800 x 480 pixels).

To use a non-volatile memory, this must be reserved first. This is done by calling the `DisplaySetTarget()` method. The size of the currently selected memory can be queried using the `STDisplayGetTargetWidth()` and `STDisplayGetTargetHeight()` methods.

5.2.4 Model type Alpha

The Alpha model has ten non-volatile image memories, which can be used for the standby image, the slide show and optimisations of the program. The memories, used for the standby image or the slide show, are read-only and can be freed only by disabling the standby image or the slide show.

The ten non-volatile memories are the same size as the volatile memories (2048 x 2048 pixels).

To use a non-volatile memory, this must be reserved first. This is done by calling the `DisplaySetTarget()` method. The size of the currently selected memory can be queried using the `STDisplayGetTargetWidth()` and `STDisplayGetTargetHeight()` methods.

5.3 Copying between image memories

The contents can be copied between the most of the available image stores. The content of the background buffer cannot be copied to the foreground buffer; it can only be moved. The contents of the overlay buffer cannot be copied but only overlaid over the display content.

Typical copy operations are copying from a non-volatile image memory in a volatile image memory and moving from the volatile background buffer into the foreground buffer. Copying an image within the device is always faster than sending this image from the PC to the device. Please refer to the descriptions of the `STDisplaySetImageFromStore()` and `STDisplaySetOverlayRect()` methods for details.

5.4 The typical process

Most applications use the same images with possibly variable units (such as document-related texts) for the signature process. It therefore makes sense to store images that are the same each time in one of the non-volatile memory if possible. The following is the typical work flow for this scenario

First, the images are loaded, which will be permanently stored in the device, since they change rarely. A memory is reserved by calling the `STDisplaySetTarget()` method with the `STPAD_TARGET_STANDARDSTORE` value. The return value of the method is the ID of the memory used. If no non-volatile image memory is available, the ID is returned as `STPAD_TARGET_BACKGROUND`, which means that the background memory is set as an image memory. This is always the case when using the Sigma model. When using the Omega, Gamma and Alpha models, the number of available memories can be less than expected when a slide show is configured.

Text and images that are added to a non-volatile memory are only saved locally to begin with and are sent to the device only when `STDisplaySetImageFromStore()` or `STDisplayConfigSlideShow()` is called in order to be able to compare the image (which may be composed of several texts and images) with the image already stored in the device. Thus only when one of these methods is called, there will be a noticeable delay.

```
LONG nTarget = STPAD_TARGET_STANDARDSTORE;
LONG nRc = STDisplaySetTarget(nTarget);
if (nRc < 0)
   return nRc;
nTarget = nRc;
nRc = STDisplaySetImageFromFile(10, 10, L"./1.png");
if (nRc < 0)
   return nRc;
nRc = STDisplaySetText(200, 160, kLeft, L"Signature:");
if (nRc < 0)
   return nRc;
nRc = STDisplaySetImageFromFile(220, 400, L"./2.png");
if (nRc < 0)
   return nRc
```

The content can now be copied to a volatile image memory, typically the background buffer (STDisplaySetTarget(STPAD_TARGET_BACKGROUND)). If the images have already been written to the background buffer because no non-volatile memory was available (see above), the STDisplaySetImageFromStore() method will not function, however, it will also not produce any errors and can therefore be safely called.

```
nRc = STDisplaySetTarget(STPAD_TARGET_BACKGROUND);
if (nRc < 0)
   return nRc;
nRc = STDisplaySetImageFromStore(nTarget);
if (nRc < 0)
   return nRc;
```

Now content, that change with every signature process, can be added to the background buffer.

```
nRc = STDisplaySetImageFromFile(120, 400, L"./3.png");
if (nRc < 0)
   return nRc;
nRc = STDisplaySetText(200, 160, kLeft, L"01.01.2010");
if (nRc < 0)
   return nRc;
```

In the background buffer there's now a collage of two images and a text copied from a non-volatile memory and an image and a text that have been sent from the PC. This collage can now be moved into the foreground buffer and thus displayed on the screen. The total composition has happened before in the background buffer and thus "invisible".

```
nRc = STDisplaySetTarget(STPAD_TARGET_FOREGROUND);
if (nRc < 0)
   return nRc;
nRc = STDisplaySetImageFromStore(STPAD_TARGET_BACKGROUND);
if (nRc < 0)
   return nRc;
```

The process described must be performed every time a connection is opened. When a connection is closed all information about reserved memories is lost. Only information regarding which display content is stored in which non-volatile memory remains saved on the device (even when it is switched off).

5.5 The standby feature

The Evolis LCD signature pads (Omega, Gamma and Alpha models only) can display one or more images automatically when not in use (no established connection). These images are stored permanently in the device and they are displayed without launching any application on the PC.

Image memories that are used by the standby feature are write protected and cannot be used for by an application.

5.5.1 Displaying a logo

In all devices, an image that is displayed automatically in standby can be stored permanently. Please refer to the descriptions of the STDisplaySetStandbyImageFromFile() method for details.

5.5.2 Displaying a slide show

Alternatively, the Omega, Gamma and Alpha models can display a slide show containing up to eleven (Omega) or ten (Gamma and Alpha) images. To configure a slide show, please follow these steps:

First, a standby mode that may be configured must be disabled by calling `STDisplayConfigSlideShow()` in order to remove write protection from all images. The current configuration can be queried with the `STDisplayGetStandbyId()` method.

Then any contents can be written to one or more of the non-volatile image memories (as described in 6.4). When all the contents have been written, the desired image memories must be configured using the `STDisplayConfigSlideShow()` method.

6 Methods

Methods are named according to the following naming convention:

- Methods that set or query general hardware properties begin with '**STDevice**'
- Methods that set or query sensor properties begin with '**STSensor**'
- Methods that apply to the signature begin with '**STSignature**'
- Methods that set or query LCD properties begin with '**STDisplay**'
- Methods that set or query component properties begin with '**STControl**'

6.1 STDeviceGetConnectionType method

This method returns the type of connection via which a device is connected.

Available from Version 8.2.0.

```
LONG STDeviceGetConnectionType(LONG nIndex)
```

<table>
<tr><th>Parameter</th><th>Values</th><th>I/O</th><th>Description</th></tr>
<tr><td>LONG nIndex</td><td>>= 0</td><td>I</td><td>Index of the device whose port number is to be queried</td></tr>
<tr><th>Return value</th><th>Values</th><th colspan="2">Description</th></tr>
<tr><td>LONG</td><td>0</td><td colspan="2">HID</td></tr>
<tr><td></td><td>1</td><td colspan="2">WinUSB</td></tr>
<tr><td></td><td>2</td><td colspan="2">Serial</td></tr>
<tr><td></td><td>3</td><td colspan="2">Ethernet</td></tr>
<tr><td></td><td>< 0</td><td colspan="2">Error</td></tr>
</table>

6.1.1 Usage:

```
LONG nType = STDeviceGetConnectionType(0);
switch (nType)
{
    case 0:
        wprintf(L"The device is connected via HID.");
        break;
    case 1:
        wprintf(L"The device is connected via WinUSB.");
        break;
    case 2:
        wprintf(L"The device is connected to a serial port.");
        break;
    case 3:
        wprintf(L" The device is connected via IP.");
        break;
    default:
        wprintf(L"Error %d", nType);
        break;
}
```

6.2 STDeviceGetCount method

This method searches for connected devices, generates an internal index beginning with 0 and returns the number of devices detected. This value should be cached so that the method only needs to be called, if the number of connected devices has changed. A device's index is retained until the method is called again. The index can be assigned to a device via the information returned by `STDeviceGetInfo()`.

By default, a search will only be made for HID devices that are locally connected. A search will only be made for other devices if this has been configured previously by calling up `STDeviceSetComPort()`.

Please observe the relevant information in the 'Using multiple instances' section.

Available from Version 8.2.0.

```
LONG STDeviceGetCount()
```

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	>= 0	Number of devices detected	
	< 0	Error	

6.2.1 Usage:

```
LONG nDeviceCount = STDeviceGetCount();
if (nDeviceCount < 0)
    wprintf(L"Error %d", nDeviceCount);
else
    wprintf(L"%d devices detected.", nDeviceCount);
```

6.3 STDeviceGetInfo method

You can use this method to retrieve the serial number and model type of a connected device in order to uniquely identify it.

Available from Version 8.2.0.

```
LONG STDeviceGetInfo(WCHAR szSerial[16], LONG* pnType, LONG nIndex)
```

Parameter	Values	I/O	Description
WCHAR szSerial[16]	max. 16 chars	O	Serial number
LONG* pnType	1	O	'Sigma HID' model type
	2	O	'Sigma serial' model type
	11	O	'Omega HID' model type
	12	O	'Omega serial' model type
	15	O	'Gamma USB' model type
	16	O	'Gamma serial' model type
	31	O	'Alpha USB' model type
	32	O	'Alpha serial' model type
	33	O	'Alpha IP' model type

	other	O	Reserved for further model types
`LONG nIndex`	>= 0	I	Index of the device whose information is to be queried
Return value	**Values**	**Description**	
`LONG`	0	Method was executed successfully	
	< 0	Error	

6.3.1 Usage:

```
WCHAR szSerial[16];
LONG nType = 0;
LONG nRc = STDeviceGetInfo(szSerial, &nType, 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"Type: %d, Serial: %s", nType, szSerial);
```

6.4 STDeviceGetVersion method

You can use this method to retrieve the version number of a connected device's firmware. It is intended primarily for support purposes.

Available from Version 8.2.0.

`LONG STDeviceGetVersion(WCHAR szVersion[16], LONG nIndex)`

Parameter	Values	I/O	Description
`WCHAR szVersion[16]`	max. 16 chars	O	Firmware version number (major.minor)
`LONG nIndex`	>= 0	I	Index of the device whose information is to be queried
Return value	**Values**	**Description**	
`LONG`	0	Method was executed successfully	
	< 0	Error	

6.4.1 Usage:

```
WCHAR szVersion[16];
LONG nRc = STDeviceGetVersion(szVersion, 0);
WCHAR szText[64];
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"Firmware: %s", szVersion);
```

6.5 STDeviceOpen method

This method opens a connection to a device.

Please observe the relevant information in the 'Using multiple instances' section.

Available from Version 8.2.0.

`LONG STDeviceOpen(LONG nIndex, BOOL bEraseDisplay=TRUE)`

Parameter	Values	I/O	Description
`LONG nIndex`	`>= 0`	I	Index of the device to which a connection is to be opened
`BOOL bEraseDisplay`	`true`	I	The device display screen will be erased (Default)
	`false`	I	The content of the display screen will not change; the screen content displayed cannot be copied into another image memory at a later stage (optional)
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.5.1 Usage:

```
LONG nRc = STDeviceOpen(0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.6 STDeviceClose method

This method closes the connection to a device. It can also be opened in another instance, provided it is running in the same memory area as the instance that is currently being used. Before closing, a currently running signature capture process is terminated and the backlight is switched off (if on).

Captured signature data is discarded.

Available from Version 8.2.0.

`LONG STDeviceClose(LONG nIndex)`

Parameter	Values	I/O	Description
`LONG nIndex`	`>= 0`	I	Index of the device whose connection is to be closed
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.6.1 Usage:

```
LONG nRc = STDeviceClose(0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.7 STDeviceSetLed method

This method sets the colour of the LED on the front of the pad. The `STDeviceSetLedDefaultFlag()` method should be called with FALSE when this method is used to ensure that the colour is not changed when `STSignatureStart()`,`STSignatureCancel()`

and `STSignatureConfirm()` are called. The LED always lights up yellow as soon as the device has been detected by the PC operating system and is ready for use.

Available from Version 8.2.0.

```
LONG STDeviceSetLed(LONG nLedColor)
```

Parameter	**Values**	**I/O**	**Description**
`LONG nLedColor`	Bitmask containing one or more hexadecimal values from the following list:		
	`0x01`	I	Yellow
	`0x02`	I	Green
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

The following values defined in the header file can be used for the `nLedColor` parameter:

```
#define STPAD_LED_YELLOW 0x01
#define STPAD_LED_GREEN  0x02
```

6.7.1 Usage:

```
LONG nRc = STDeviceSetLed(STPAD_LED_YELLOW);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.8 STSensorGetSampleRateMode method

This method returns the configured sample rate with which the signature is captured.

Available from Version 8.2.0.

```
LONG STSensorGetSampleRateMode()
```

Parameter	**Values**	**I/O**	**Description**
–	–	-	-
Return value	**Values**	**Description**	
`LONG`	`3`	280 Hz	
	`2`	500 Hz	
	`1`	250 Hz	
	`0`	125 Hz	
	`< 0`	Error	

6.8.1 Usage:

```
LONG nMode = STSensorGetSampleRateMode();
switch (nMode)
{
    case 0:
        wprintf(L"Sample rate is 125 Hz.");
        break;
    case 1:
        wprintf(L"Sample rate is 250 Hz.");
        break;
    case 2:
        wprintf(L"Sample rate is 500 Hz.");
        break;
    case 3:
        wprintf(L"Sample rate is 280 Hz.");
        break;
    default:
        wprintf(L"Error %d", nMode);
        break;
}
```

6.9 STSensorSetSampleRateMode method

This method sets the sample rate with which the signature is captured. The default setting is mode 1 (250 Hz) or mode 3 (280 Hz) when using the Alpha model. This mode provides high-quality signature data while at the same time ensures that the data record is of moderate size. When using the Sigma, Gamma and Omega models, this value can easily be set to 2 (500 Hz) for high-speed data lines.

Available from Version 8.2.0.

```
LONG STSensorSetSampleRateMode(LONG nMode)
```

Parameter	Values	I/O	Description
LONG nMode	0	I	125 Hz (currently only Sigma, Gamma and Omega)
	1	I	250 Hz (currently only Sigma, Gamma and Omega)
	2	I	500 Hz (currently only Sigma, Gamma and Omega)
	3	I	280 Hz (currently only Alpha)
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.9.1 Usage:

```
LONG nRc = STSensorSetSampleRateMode(1);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.10 STSensorSetSignRect method

This method defines the rectangle in which the signature is captured. If the rectangle overlaps one of the hotspots that has been defined (see `STSensorAddHotSpot()`), an error is returned.

Available from Version 8.2.0.

```
LONG STSensorSetSignRect(LONG nLeft, LONG nTop, LONG nWidth, LONG
nHeight)
```

Parameter	Values	I/O	Description
`LONG nLeft`	`>= 0`	I	Left boundary; 0 is on the far left of the display
`LONG nTop`	`>= 0`	I	Upper boundary; 0 is at the top of the display
`LONG nWidth`	`> 3`	I	Width; `STDisplayGetWidth()` returns the width of the LCD used
	`0`	I	Right boundary is automatically set to the maximum value (right margin of the LCD)
`LONG nHeight`	`> 3`	I	Height; `STDisplayGetHeight()` returns the height of the LCD used
	`0`	I	Lower boundary is automatically set to the maximum value (lower margin of the LCD)
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.10.1 Usage:

```
LONG nRc = STSensorSetSignRect(0, 40, 0, 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.11 STSensorClearSignRect method

This method erases the signature window.

Available from Version 8.2.0.

```
LONG STSensorClearSignRect()
```

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.11.1 Usage:

```
LONG nRc = STSensorClearSignRect();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.12 STSensorAddHotSpot method

This method defines a rectangular subarea of the sensor surface that responds to user clicks. See also `STSensorHotSpotPressed()`. The rectangle must lie in the area defined by `STDisplaySetOverlayRect()` if a scroll hotspot has already been defined. It should not overlap the defined signature window (see `STSensorSetSignRect()`) or a hotspot that was previously set.

Available from Version 8.2.0.

```
LONG STSensorAddHotSpot(LONG nLeft, LONG nTop, LONG nWidth, LONG
nHeight)
```

Parameter	Values	I/O	Description
`LONG nLeft`	`>= 0`	I	Left boundary; 0 is on the far left of the display
`LONG nTop`	`>= 0`	I	Upper boundary; 0 is at the top of the display
`LONG nWidth`	`> 3`	I	Width; `STDisplayGetWidth()` returns the width of the LCD used
	`0`	I	Right boundary is automatically set to the maximum value (right margin of the LCD)
`LONG nHeight`	`> 3`	I	Height; `STDisplayGetHeight()` returns the height of the LCD used
	`0`	I	Lower boundary is automatically set to the maximum value (lower margin of the LCD)
Return value	**Values**	**Description**	
`LONG`	`>= 0`	ID of the hotspot that was generated	
	`< 0`	Error	

6.12.1 Usage:

```
LONG nRc = STSensorAddHotSpot(0, 0, 0, 40);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.13 STSensorSetHotspotMode method

This method defines the behavior of a monitored area (hotspot).

Available from Version 8.2.0.

```
LONG STSensorSetHotSpotMode(HOTSPOTMODE nMode, LONG nHotSpotId)
```

Parameter	Values	I/O	Description
`HOTSPOTMODE nMode`	`0`	I	Deactivates the monitored area
	`1`	I	Activates the monitored area (default after calling `STSensorAddHotSpot()` or `STSensorAddScrollHotSpot()`)
	`2`	I	Activates the monitored area but disables the automatic inverting when the area is clicked
`LONG nHotSpotId`	`>= 0`	I	ID of the hotspot that is to be changed
Return value	**Values**	**Description**	
`LONG`	`>= 0`	ID of the hotspot that was generated	
	`< 0`	Error	

The `HOTSPOTMODE` enumeration is defined as follows:

```
kInactive = 0,
kActive = 1,
kInvertOff = 2
```

6.13.1 Usage:

```
LONG nRc = STSensorSetHotspotMode(kActive, 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.14 STSensorClearHotSpots method

This method removes all monitored areas (hotspots).

Available from Version 8.2.0.

`LONG STSensorClearHotSpots()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.14.1 Usage:

```
LONG nRc = STSensorClearHotSpots();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.15 STSensorStartTimer method

This method starts a Timer, which starts a defined function, if there was no interaction on the sensor of the pad for the given time periods. This Functionality is intended primarily to capture a signature without user interaction, but it can also be used to get a confirmation for a displayed text, if the belonging hotspot is not pressed for a given time period.

Available from Version 8.2.1.15 onwards

`LONG STSensorStartTimer(LONG nWaitBeforeAction, LONG nWaitAfterAction, LONG nOptions)`

Parameter	Values	I/O	Description
LONG nWaitBeforeAction	0	I	No timer waiting for the first interaction is started
	> 0	I	Maximum time to wait for the first interaction (in milliseconds) before the defined function is triggered (for example, before the start of a signature); the timer is restarted with this value after the calling of `STSignatureRetry()`.
LONG nWaitAfterAction	0	I	After the first interaction no timer waiting for the next interaction is started
	> 0	I	Maximum time to wait after the last interaction was noticed in millisecond. If the given time period elapsed without a new interaction, the wanted function will be called (usually this takes places after the signing is finished).

LONG nOptions	0	I	If the timer has expired, the `SensorTimeoutOccured()` event is called.
	1	I	If the time period of `nWaitBeforeAction` has elapsed, `STSignatureCancel()` is called; if the time period of `nWaitAfterAction` has elapsed, `STSignatureConfirm()` is called
	2	I	If the timer has expired, `STSignatureCancel()` is called
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.15.1 Usage:

```
LONG nRc = STSensorStartTimer(10000, 1000, 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.16 STSensorStopTimer method

This method stops a timer started with `STSensorStartTimer()` without triggering the function defined there. The method is called automatically if `STSignatureConfirm()` or `STSignatureCancel()` is called.

Available from Version 8.2.1.15 onwards.

`LONG STSensorStopTimer()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully (will be returned also, if no timer was set before)	
	< 0	Error	

6.16.1 Usage:

```
LONG nRc = STSensorStopTimer();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.17 STSignatureStart method

This method starts the signature capture process provided a connection has been opened via `DeviceOpen()`. The entire sensor is used as a writing surface provided no signature window has been defined. Signature data is only received, if a signature is actually entered on the pad. The method sets the colour of the LED to green unless the `STDeviceGetLedDefaultFlag()` method returns FALSE. This method automatically restores the previous content of the LCD unless this has been explicitly erased by calling `STDisplayErase()`.

Available from Version 8.2.0.

`LONG STSignatureStart()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.17.1 Usage:

```
LONG nRc = STSignatureStart();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.18 STSignatureStop method

This method terminates the signature capture process that is currently running, and caches the captured signature data. Unlike the `STSignatureConfirm()` method, it does not change the display content. `STSignatureStop()` sets the colour of the LED to yellow unless the `STDeviceGetLedDefaultFlag()` method returns FALSE.

Available from Version 8.2.0.

`LONG STSignatureStop()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	>= 0	Number of points captured	
	< 0	Error	

6.18.1 Usage:

```
LONG nRc = STSignatureStop();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"%d points captured.", nRc);
```

6.19 STSignatureConfirm method

This methods terminates the signature capture process that is currently running (if any), caches the captured signature data and, unlike `STSignatureStop()`, erases the entire LCD. `STSignatureConfirm()` sets the colour of the LED to yellow unless the `STDeviceGetLedDefaultFlag()` method returns FALSE.

Available from Version 8.2.0.

`LONG STSignatureConfirm()`

Parameter	Values	I/O	Description
–	–	-	-

Return value	Values	Description
LONG	>= 0	Number of points captured
	< 0	Error

6.19.1 Usage:

```
LONG nRc = STSignatureConfirm();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"%d points captured.", nRc);
```

6.20 STSignatureRetry method

This method discards the signature data without ending the signature capture process, and deletes the rendered signature in the LCD. This method starts a new capture process if the signature capture process was terminated beforehand with `STSignatureStop()`.

Available from Version 8.2.0.

`LONG STSignatureRetry()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.20.1 Usage:

```
LONG nRc = STSignatureRetry();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.21 STSignatureCancel method

This method ends the capture process, discards the signature data and deletes the entire LCD or just the signature. The colour of the LED is set to yellow unless the `STDeviceGetLedDefaultFlag()` method returns FALSE. This method is called automatically when `STDeviceClose()` is called.

Available from Version 8.2.0.

`LONG STSignatureCancel(ERASEOPTION nErase=kComplete)`

Parameter	Values	I/O	Description
ERASEOPTION nErase	0	I	The entire LCD will be deleted (Default)
	1	I	Only the signature will be deleted (Optional)
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

The `ERASEOPTION` enumeration is defined as follows:

```
kComplete = 0,
kSignature = 1
```

6.21.1 Usage:

```
LONG nRc = STSignatureCancel();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.22 STSignatureGetState method

This method returns the current state of the signature capture process.

Available from Version 8.2.0.

`BOOL STSignatureGetState()`

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
BOOL	TRUE	Signature capture process is running	
	FALSE	Signature capture process is not running	

6.22.1 Usage:

```
if (!STSignatureGetState())
    STSignatureStart();
else
    STSignatureConfirm();
```

6.23 STSignatureGetSignData method

This method returns the digitalised signature in SignData format.

Available from Version 8.2.0.

`LONG STSignatureGetSignData(BYTE* pbtSignData, LONG* pnSize)`

Parameter	Values	I/O	Description
BYTE* pbtSignData	NULL	I	The method returns the required size of the array in the `pnSize` parameter.
	other	I/O	Array (in the required size) in which the SignData is written; `pnSize` must correspond to the value returned for the previous call.
LONG* pnSize	> 0	I/O	Size of the array (in bytes) in which the SignData is to be written
Return value	**Values**	**Description**	
LONG	0	Method was executed successfully	
	< 0	Error	

6.23.1 Usage:

```
LONG nSize = 0;
LONG nRc = STSignatureGetSignData(NULL, &nSize);
BYTE* pbtSignData = NULL;
if (nRc == 0)
{
    pbtSignData = new BYTE[nSize];
    nRc = STSignatureGetSignData(pbtSignData, &nSize);
}
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.24 STSignatureSaveAsFileEx method

This method can be used to save a captured signature as an image file on a hard disk. The colour depth depends on the file type, the device used and the settings. If no further settings are made (see `nOptions` parameter), the image is created with the aspect ratio of the rectangle that surrounds the signature.

Available from Version 8.2.0. The status described is available from Version 8.2.1.16.

```
LONG STSignatureSaveAsFileEx(LPCWSTR szPath, LONG nResolution, LONG
nWidth, LONG nHeight, FILETYPE nFileType, LONG nPenWidth, COLORREF
clrPen, LONG nOptions)
```

Parameter	Values	I/O	Description
LPCWSTR szPath	!= NULL	I	Storage location for the image file as a full path that includes the file name
LONG nResolution	>=75 <=600	I	Resolution of the image file in pixels per inch (ppi); for the signature to be displayed in its original size, this value must be identical to the resolution of the document, in which the signature is to be integrated
LONG nWidth	0	I	The image will be created in original size; the `nHeight` parameter is ignored.
	> 0	I	Maximum width of the image in pixels
LONG nHeight	0	I	The image will be created in original size; the `nWidth` parameter is ignored
	> 0	I	Maximum height of the image in pixels
FILETYPE nFileType	0	I	Use TIFF with CCITT4 compression (b/w image) or LZW compression (colour image) as the file format (recommended)
	1	I	Use PNG file format
	2	I	Use BMP file format. The type is not supported on Linux
	3	I	Use JPEG with a quality setting of 75 as the file format
	4	I	Use GIF file format (the resolution will always be 96 ppi). The type is not supported on Linux

`LONG nPenWidth`	`< 0`	I	Fixed stroke width in pixels (absolute value); the pressure values are visualised by drawing in variable brightness
	`0`	I	A variable pen width is used that is dependent on the resolution and the pressure values
	`> 0`	I	Fixed pen width in pixels
`COLORREF clrPen`	`>= 0`	I	Signature colour
`LONG nOptions`	Bitmask containing one or more hexadecimal values from the following list:		
	`0x0001`	I	A visual timestamp is added to the image beneath the signature
	`0x0002`	I	The signature is rendered in the image displayed during capture; the image always has the aspect ratio of the display that is used, `nWidth` or `nHeight` may be ignored
	`0x0004`	I	The defined hotspot areas are whitened in the image (only if `0x0002` is set).
	`0x0008`	I	White areas at the sides of the signature are not removed; if `nWidth` and `nHeight` are greater than 0, the signature is scaled to the defined height or width depending on aspect ratio, and the image to be created has the exact size defined by `nWidth` and `nHeight` (only if `0x0002` is not set).
	`0x0010`	I	The signature will be aligned to the left (only if `0x0008` is set).
	`0x0020`	I	The signature will be aligned to the right (only if `0x0008` is set)
	`0x0040`	I	The signature will be aligned to the top (only if `0x0008` is set)
	`0x0080`	I	The signature will be aligned to the bottom (only if `0x0008` is set)
	`0x0100`	I	The timestamp size is relative to the height of the created image, not to the height of the display; this setting is useful if the signature is scaled to a given image size to make sure that the timestamp size is independent from the size of the actual signature (only if `0x0001` is set)
	`0x0200`	I	The signature is never smoothed independent from all other settings; this will create small files
	`0x0400`	I	The signature is always smoothed independent from all other settings
	`0x0800`	I	The image includes the overlay rectangle if displayed (only if `0x0002` is set)
	`0x1000`	I	White areas are stored as transparent (only if `0x0002` is not set and PNG is selected as the file format)
	`0x2000`	I	The current display content and not the content displayed during capture is used as the background image (only if `0x0002` is set)
	`0x4000`	I	The pen width will vary by the value indicated depending on the pressure values

Return value	Values	Description
LONG	0	Method was executed successfully
	< 0	Error

The FILETYPE enumeration is defined as follows:

```
kTiff = 0,
kPng = 1,
kBmp = 2,
kJpeg = 3,
kGif = 4
```

The following values defined in the header file can be used for the nOptions parameter:

```
#define STPAD_SIMG_TIMESTAMP          0x0001
#define STPAD_SIMG_BACKIMAGE          0x0002
#define STPAD_SIMG_HOTSPOTS           0x0004
#define STPAD_SIMG_NOCROPPING         0x0008
#define STPAD_SIMG_ALIGNLEFT          0x0010
#define STPAD_SIMG_ALIGNRIGHT         0x0020
#define STPAD_SIMG_ALIGNTOP           0x0040
#define STPAD_SIMG_ALIGNBOTTOM  0x0080
#define STPAD_SIMG_TIMESTAMPIMGREL    0x0100
#define STPAD_SIMG_DONTSMOOTH         0x0200
#define STPAD_SIMG_SMOOTH       0x0400
#define STPAD_SIMG_OVERLAYIMAGE 0x0800
#define STPAD_SIMG_TRANSPARENT  0x1000
#define STPAD_SIMG_CURRENTIMAGES0x2000
#define STPAD_SIMG_VARIABLEPENWIDTH   0x4000
```

6.24.1 Usage:

```
LONG nRc = STSignatureSaveAsFileEx(L"./Signature.tif", 300, 0, 0,
                                kTiff, 0, RGB(0, 0, 255), 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.25 STSignatureGetBounds method

This method delivers the coordinates of the rectangle in which the captured signature is given.

Available from Version 8.2.0.

```
LONG STSignatureGetBounds(LONG* pnLeft, LONG* pnTop, LONG* pnRight,
LONG* pnBottom, LONG nOptions)
```

Parameter	Values	I/O	Description
LONG* pnLeft	>= 0	O	Left border of the signature rectangle
LONG* pnTop	>= 0	O	Upper border of the signature rectangle
LONG* pnRight	>= 0	O	Right border of the signature rectangle
LONG* pnBottom	>= 0	O	Bottom border of the signature rectangle

`LONG nOptions`	0	I	The coordinates will be delivered relative to the size of the used LCD
	1	I	The coordinates will be returned relative to the defined size of the signature rectangle (see `STSensorSetSignRect()`)
Return value	**Values**	**Description**	
`LONG`	0	Method was executed successfully	
	< 0	Error	

The following values defined in the header file can be used for the `nOptions` parameter:

```
#define STPAD_BOUNDS_DISPLAY   0
#define STPAD_BOUNDS_SIGNRECT  1
```

6.25.1 Usage:

```
LONG nLeft, nTop, nRight, nBottom;
LONG nRc = STSignatureGetBounds(&nLeft, &nTop, &nRight, &nBottom,
                             STPAD_BOUNDS_DISPLAY);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
{
    wprintf(L"The Bounds of the Signature are: %d (left), "%d
           (top), %d (right) & %d (bottom).", nLeft, nTop, nRight,
           nBottom);
}
```

6.26 STSignatureScaleToDisplay method

This method converts the sensor coordinates delivered by the`SignatureDataReceived()` event into display coordinates.

Available from Version 8.2.0.

```
LONG STSignatureScaleToDisplay(LONG nSensorValue)
```

Parameter	**Values**	**I/O**	**Description**
`LONG nSensorValue`	>= 0	I	x or y value of a sensor coordinate
Return value	**Values**	**Description**	
`LONG`	0	x or y value of a display coordinate	
	< 0	Error	

6.26.1 Usage:

```
LONG nRc = STSignatureScaleToDisplay(1000);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"Display Value: %d", nRc);
```

6.27 STDisplayGetWidth method

This method returns the width of the LCD. It can only be called after a device has been opened.

Available from Version 8.2.0.

```
LONG STDisplayGetWidth()
```

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	>= 0	Width of the display in pixels	
	< 0	Error	

6.27.1 Usage:

```
wprintf(L"Display width is %d", STDisplayGetWidth());
```

6.28 STDisplayGetHeight method

This method returns the height of the LCD. It can only be called after a device has been opened.

Available from Version 8.2.0.

```
LONG STDisplayGetHeight()
```

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	>= 0	Height of the display in pixels	
	< 0	Error	

6.28.1 Usage:

```
wprintf(L"Display height is %d", STDisplayGetHeight());
```

6.29 STDisplayGetTargetWidth method

This method returns the width of the memory defined with the `STDisplaySetTarget()` method. It can only be called after a device has been opened.

Available from Version 8.2.0.

```
LONG STDisplayGetTargetWidth()
```

Parameter	Values	I/O	Description
–	–	-	-
Return value	**Values**	**Description**	
LONG	>= 0	Width of the memory in pixels	
	< 0	Error	

6.29.1 Usage:

```
wprintf(L"Target width is %d", STDisplayGetTargetWidth());
```

6.30 STDisplayGetTargetHeight method

This method returns the height of the memory defined with the `STDisplaySetTarget()` method. It can only be called after a device has been opened.

Available from Version 8.2.0.

`LONG STDisplayGetTargetHeight()`

Parameter	Values	I/O	Description
–	–	-	-

Return value	Values	Description
LONG	>= 0	Height of the memory in pixels
	< 0	Error

6.30.1 Usage:

```
wprintf(L"Target height is %d", STDisplayGetTargetHeight());
```

6.31 STDisplayErase method

This method erases both the foreground and the background buffer and removes the overlay rectangle if set. Thus the entire contents of the LCD is erased. To erase only parts of the memory defined with `STDisplaySetTarget()`, please use `STDisplayEraseRect()`.

Available from Version 8.2.1.15 onwards.

`LONG STDisplayErase()`

Parameter	Values	I/O	Description
–	–	-	-

Return value	Values	Description
LONG	0	Method was executed successfully
	< 0	Error

6.31.1 Usage:

```
LONG nRc = STDisplayErase();
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.32 STDisplayEraseRect method

This method erases a rectangle in the memory defined with `STSisplaySetTarget()`.

Available from Version 8.2.0.

```
LONG STDisplayEraseRect(LONG nLeft, LONG nTop, LONG nWidth, LONG
nHeight)
```

Parameter	Values	I/O	Description
`LONG nLeft`	`>= 0`	I	Left boundary; 0 is on the far left of the display
`LONG nTop`	`>= 0`	I	Upper boundary; 0 is at the top of the display
`LONG nWidth`	`> 0`	I	Width; `STDisplayGetWidth()` returns the width of the LCD used
	`0`	I	Right boundary is automatically set to the maximum value (right edge of the LCD)
`LONG nHeight`	`> 0`	I	Height; `STDisplayGetHeight()` returns the height of the LCD used
	`0`	I	Lower boundary is automatically set to the maximum value (lower edge of the LCD)
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.32.1 Usage:

```
LONG nRc = STDisplayEraseRect(10, 50, 30, 20);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.33 STDisplayConfigPen method

This method sets the pen width and colour used to display a signature on the LCD. The pen width is always stored permanently in the device; the pen colour is stored permanently only on Omega devices with firmware 1.4 or later.

Available from Version 8.2.0.

```
LONG STDisplayConfigPen(LONG nWidth, COLORREF clrPen)
```

Parameter	Values	I/O	Description
`LONG nWidth`	`1 – 3`	I	Width in pixels
`COLORREF clrPen`	`>= 0`	I	Colour; this parameter is ignored for the Sigma model
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.33.1 Usage:

```
LONG nRc = STDisplayConfigPen(2, RGB(0, 0, 255));
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.34 STDisplaySetFont method

This method permanently sets the font that is used to output text to the LCD. Text that has already been output is not modified. Ubuntu 20 pt (Sigma model) or 40 pt (Omega, Gamma and Alpha models) is set when `STDeviceOpen()` is called.

Available from Version 8.2.1.15 onwards.

```
LONG STDisplaySetFont(LPCWSTR szName, LONG nSize, LONG nOptions)
```

Parameter	Values	I/O	Description
`LPCWSTR szName`	`!= NULL`	I	Full name of the font, which must be installed on the PC.
`LONG nSize`	`12 - 200`	I	Font size
`LONG nOptions`	Bitmask containing one or more hexadecimal values from the following list:		
	`0x01`	I	Bold
	`0x02`	I	Underlined; the option is not supported on Linux
	`0x04`	I	Italicised
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

The following values defined in the header file can be used for the `nOptions` parameter:

```
#define STPAD_FONT_NORMAL  0x00
#define STPAD_FONT_BOLD         0x01
#define STPAD_FONT_UNDERLINE    0x02
#define STPAD_FONT_ITALIC  0x04
```

6.34.1 Usage:

```
LONG nRc = STDisplaySetFont(L"Ubuntu Mono", 20,
                                 STPAD_FONT_NORMAL);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.35 STDisplaySetFontColor method

This method permanently sets the colour in which the text is displayed on the LCD. Text that has already been output is not modified. The given values will be ignored, if a pad without a color LCD is used. The colour black is set when the component is initialised.

Available from Version 8.2.1.15 onwards.

```
LONG STDisplaySetFontColor(COLORREF clrFont)
```

Parameter	Values	I/O	Description
`COLORREF clrFont`	`>= 0`	I	Text colour
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.35.1 Usage:

```
LONG nRc = STDisplaySetFontColor(RGB(238, 121, 0));
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.36 STDisplaySetTarget method

This method defines the device memory that is used by the following methods and properties: `STDisplayEraseRect()`, `STDisplaySetText()`,`STDisplaySetTextInRect()`, `STDisplaySetImage()`,`STDisplaySetImageFromFile()`, `STDisplaySetImageFromStore()`,`STDisplaySetScrollPos()`, `STDisplayGetScrollPos()`,`STDisplayGetTargetWidth()` and `STDisplayGetTargetHeight()`. The set memory remains valid until the next call of this method or of `STDeviceClose()`. Contents stored in a non-visible memory can be displayed with the `STDisplaySetImageFromStore()` method. For more details, see Chapter 6.

After the calling of `STDeviceOpen()`, the methods specified above are all executed directly on the LCD (foreground buffer) as long as `STDisplaySetTarget()` is not called .

Available from Version 8.2.0.

```
LONG STDisplaySetTarget(LONG nTarget)
```

Parameter	Values	I/O	Description
`LONG nTarget`	-2	I	A permanent memory that can hold an image of the same width and double the height of the display is reserved inside the device; the memory can be used for writing from now on; if there is no permanent memory available, the return value will be 1 (see below); the value -2 is handled as -1 for the Gamma and Alpha models (see there for details)
	-1	I	A permanent memory that can hold an image in the size of the display (Gamma and Omega models) or up to a size of 2048 x 2048 pixels (Alpha model) is reserved inside the device; the memory can be used for writing from now on; if there is no permanent memory available, the return value will be 1 (see below)
	0	I	All content is displayed directly on the LCD and stored in the foreground buffer; the content is lost if the device is switched off or if `STDisplayErase()` or `STDeviceClose()` is called
	1	I	All content is written to the non-visible background buffer; the background buffer is used internally during the signature process, so content is no longer available after `STSignatureStart()` has been called; the content is also lost if the device is switched off or if `STDisplayErase()` or `STDeviceClose()` is called

	2	I	All content is written to the overlay buffer; it is visible immediately if an overlay rectangle has already been defined; the content is lost if the device is switched off or if `STDisplayErase()` or `STDeviceClose()` is called; this value can only be used for the Omega, Gamma and Alpha models
	other	I	Enables direct access to a permanent storage; The storage must be reserved before it can be used (see above for value -1 and -2)

Return value	**Values**	**Description**
`LONG`	`>= 0`	ID of the store which is now selected; all the above referred methods will now be applied to this store; this ID can be used when calling this method again to specifically address this store
	`< 0`	Error

The following values defined in the header file can be used for the `nTarget` parameter:

```
#define STPAD_TARGET_LARGESTORE    -2
#define STPAD_TARGET_STANDARDSTORE -1
#define STPAD_TARGET_FOREGROUND     0
#define STPAD_TARGET_BACKGROUND     1
#define STPAD_TARGET_OVERLAY        2
```

6.36.1 Usage:

```
LONG nStoreId = STDisplaySetTarget(STPAD_TARGET_STANDARDSTORE);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.37 STDisplaySetText method

This method can be used to write any text to the memory defined with the `STDisplaySetTarget()` method. The rectangle enclosing the text overlays existing information in the memory. The text can also appear outside of the display and it is not wrapped. Ubuntu 20 pt (Sigma model) or 40 pt (Omega, Gamma and Alpha models) is used, unless another font has been set using the `STDisplaySetFont()` method. The colour of the text will be black unless another colour has been set using the `STDisplaySetFontColor()` method.

Available from Version 8.2.0.

```
LONG STDisplaySetText(LONG nXPos, LONG nYPos, ALIGN nAlignment, LPCWSTR
szText)
```

Parameter	**Values**	**I/O**	**Description**
`LONG nXPos`	all	I	X coordinate of the starting point; 0 is on the far left of the display; `STDisplayGetWidth()` returns the point on the far right of the display
`LONG nYPos`	all	I	Y coordinate of the starting point; 0 is at the top of the display; `STDisplayGetHeight()` returns the point at the very bottom of the display
`ALIGN nAlignment`	0	I	Text is aligned to the right of the starting point
	1	I	Text is centred horizontally at the starting point

	2	I	Text is aligned to the left of the starting point
`LPCWSTR szText`	`!= NULL`	I	Text to be output
Return value	**Values**	**Description**	

`LONG`
`int`
`Integer`
	`>= 0`	Width of the rectangle enclosing the text	

	`< 0`	Error	

The `ALIGN` enumeration is defined as follows:

```
kLeft = 0,
kCenter = 1,
kRight = 2,
kLeftCenteredVertically = 3,
kCenterCenteredVertically = 4,
kRightCenteredVertically = 5,
kLeftNoWrap = 6,
kCenterNoWrap = 7,
kRightNoWrap = 8
```

6.37.1 Usage:

```
LONG nRc = STDisplaySetText(50, 20, kLeft, L"Text");
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.38 STDisplaySetTextInRect method

This method can be used to write any text to the memory defined with the `STDisplaySetTarget()` method. The specified rectangle overlays existing information in the memory. The text is placed in the rectangle. No check is made regarding whether the rectangle is within the display. Ubuntu 20 pt (Sigma model) or 40 pt (Omega, Gamma and Alpha models) is used, unless another font has been set using the `STDisplaySetFont()` method. If the text is too long, the font size is automatically reduced to a minimum of 12pt. The colour of the text will be black unless another colour has been set using the `STDisplaySetFontColor()` method.

Available from Version 8.2.0.

```
LONG STDisplaySetTextInRect(LONG nLeft, LONG nTop, LONG nWidth, LONG
nHeight, ALIGN nAlignment, LPCWSTR szText)
```

Parameter	Values	I/O	Description
`LONG nXPos`	all	I	X coordinate of the starting point; 0 is on the far left of the display
`LONG nYPos`	all	I	Y coordinate of the starting point; 0 is on the top of the display
`LONG nWidth`	`> 0`	I	Width; `STDisplayGetWidth()` returns the width of the LCD used
	`0`	I	Right boundary is automatically set to the maximum value (right margin of the LCD)

LONG nHeight	> 0	I	Height; STDisplayGetHeight() returns the height of the LCD used
	0	I	Lower boundary is automatically set to the maximum value (lower margin of the LCD)
ALIGN nAlignment	0	I	Text is left-aligned and wrapped automatically
	1	I	Text is centred and wrapped automatically
	2	I	Text is right-aligned and wrapped automatically
	3	I	Text is left-aligned and centred vertically in the rectangle with no wrapping (breaks are ignored)
	4	I	Text is centred vertically and horizontally in the rectangle with no wrapping (breaks are ignored); this setting is ideal for button text
	5	I	Text is right-aligned and centred vertically in the rectangle with no wrapping (breaks are ignored)
	6	I	Text is left-aligned and not wrapped automatically (breaks are retained)
	7	I	Text is centred and not wrapped automatically (breaks are retained)
	8	I	Text is right-aligned and not wrapped automatically (breaks are retained)
LPCWSTR szText	!= NULL	I	Text to be output
Return value	**Values**	**Description**	
LONG	>=12	The font size that is actually used	
	< 0	Error	

The ALIGN enumeration is defined as follows:

```
kLeft = 0,
kCenter = 1,
kRight = 2,
kLeftCenteredVertically = 3,
kCenterCenteredVertically = 4,
kRightCenteredVertically = 5,
kLeftNoWrap = 6,
kCenterNoWrap = 7,
kRightNoWrap = 8
```

6.38.1 Usage:

```
LONG nRc = STDisplaySetTextInRect(0, 0, 20, 40, kLeft, L"Text");
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.39 STDisplaySetImageFromFile method

This method allows an image whose path is transferred to be written to the memory defined using the STDisplaySetTarget() method. Although the colour depth is automatically adjusted to the connected LCD, it is still advisable to correctly generate the image beforehand (for example, a 1-bit monochrome image is required for the Sigma model). The transfer time for the Omega, Gamma and Alpha models depends on the image material; the best pictures have few colours, so they can be compressed well. The image overlays the existing information in the memory and any signature that is present is completely erased. The image may also be positioned outside of the display.

Available from Version 8.2.0. The status described is available from Version 8.2.1.16.

```
LONG STDisplaySetImageFromFile(LONG nXPos, LONG nYPos, LPCWSTR szPath)
```

Parameter	Values	I/O	Description
`LONG nXPos`	all	I	X coordinate of the point from which the bitmap is output to the right; 0 is on the far left of the display; `STDisplayGetWidth()` returns the point on the far right of the display
`LONG nYPos`	all	I	Y coordinate of the point from which the bitmap is output downwards; 0 is at the top of the display; `STDisplayGetHeight()` returns the point at the very bottom of the display
`LPCWSTR szPath`	`!= NULL`	I	Full file path of the image; TIFF, PNG, BMP and JPEG can be used as image formats; JPEG is not supported on Linux with the model Sigma
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.39.1 Usage:

```
LONG nRc = STDisplaySetImageFromFile(0, 0, L"./Image.png");
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.40 STDisplaySetImageFromStore method

This method allows an image that has been stored in a device memory to be written to the memory defined by the `STDisplaySetTarget()` method. The content to copy will overlay the content which is currently stored in the target storage. For more details, see Chapter 6.

If the memory defined by `nStoreId` was not reserved beforehand by calling `STDisplaySetTarget()`, the content is copied as desired; however, it is not available within the component for storing with the `DisplaySaveImage…()` or `SignatureSave…()` method. To differentiate this case from a call with a reserved `nStoreId`, `nStoreId` is returned instead of 0.

The scroll position of the source memory will be assigned to the destination memory, if both have the same size, else it will be set to 0 / 0.

Available from Version 8.2.0.

```
LONG STDisplaySetImageFromStore(LONG nStoreId)
```

Parameter	Values	I/O	Description
`LONG nStoreId`	`>= 0`	I	ID of the memory from which the image is to be read; the ID is the value returned by the `STDisplaySetTarget()` method

Return value	Values	Description
LONG	other	The memory defined by nStoreId has not been reserved beforehand; the content was successfully copied, but is not available within the component; the returned value is identical to the value of nStoreId
	0	Method was executed successfully
	< 0	Error

The following values defined in the header file or the ID of a reserved, non-volatile memory can be used for the nStoreId parameter:

```
#define STPAD_TARGET_FOREGROUND      0
#define STPAD_TARGET_BACKGROUND      1
```

6.40.1 Usage:

```
LONG nRc = STDisplaySetImageFromStore(STPAD_TARGET_BACKGROUND);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.41 STDisplaySetOverlayRect method

This method defines a rectangular subarea of the display whose contents will be covered by the contents of the overlay buffer. The foreground buffer is covered within this rectangle until it is removed again or until STDisplayErase(), STSignatureConfirm() or STSignatureCancel() is called. This functionality is ideal for a toolbar that displays hotspots, for example, to scroll.

If the storage defined with STDisplaySetTarget() is not the foreground buffer, the rectangle is not overlaid until STDisplaySetImageFromStore() (with the foreground buffer as the destination) is called to synchronise the display.

The parameters must be multiples of 8 and may be rounded.

This method cannot be called when a both a standard hotpot lying outside of the given rectangle and a scroll hotspot has been defined previously.

This method only works with Omega, Gamma and Alpha models.

Available from Version 8.2.0.

```
LONG STDisplaySetOverlayRect(LONG nLeft, LONG nTop, LONG nWidth, LONG
nHeight)
```

Parameter	Values	I/O	Description
LONG nLeft	>= 0	I	Left boundary; 0 is on the far left of the display
LONG nTop	>= 0	I	Upper boundary; 0 is at the top of the display
LONG nWidth	>= 8	I	Width; STDisplayGetWidth() returns the width of the LCD used
	0	I	The overlay rectangle is removed; the complete contents of the foreground buffer will be visible again

`LONG nHeight`	`>= 8`	I	Height; `STDisplayGetHeight()` returns the height of the LCD used
	`0`	I	The overlay rectangle is removed; the complete contents of the foreground buffer will be visible again
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.41.1 Usage:

```
LONG nRc = STDisplaySetOverlayRect(0, 400, 640, 80);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.42 STDisplaySetScrollPos method

This method defines the X/Y position where the contents of the storage defined with the `STDisplaySetTarget()` method will be displayed. This method only works for image memories with a size larger than the display size. Please refer to the descriptions of the `STDisplayGetTargetWidth()` and `STDisplayGetTargetHeight()` methods.

Available from Version 8.2.0.

`LONG STDisplaySetScrollPos(LONG nXPos, LONG nYPos)`

Parameter	Values	I/O	Description
`LONG nXPos`	`>= 0`	I	Horizontal offset of the memory contents to the left, in pixels; the maximum possible value is calculated from `STDisplayGetTargetWidth()` - `STDisplayGetWidth()`
`LONG nYPos`	`>= 0`	I	Vertical offset of the memory contents to the top, in pixels; the maximum possible value is calculated from `STDisplayGetTargetHeight()` - `STDisplayGetHeight()`
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.42.1 Usage:

```
LONG nRc = STDisplaySetScrollPos(0, 100);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.43 STDisplayGetScrollPos method

This method returns the X/Y position where the contents of the storage defined with the `STDisplaySetTarget()` method is displayed.

Available from Version 8.2.0.

```
LONG STDisplayGetScrollPos(LONG* pnXPos, LONG* pnYPos)
```

Parameter	Values	I/O	Description
`LONG* pnXPos`	`!= NULL`	O	Horizontal offset of the memory contents to the left, in pixels
`LONG* pnYPos`	`!= NULL`	O	Vertical offset of the memory contents to the top, in pixels
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.43.1 Usage:

```
LONG nXPos, nYPos;
LONG nRc = STDisplayGetScrollPos(&nXPos, &nYPos);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"Scroll pos: %d / %d", nXPos, nYPos);
```

6.44 STDisplaySaveImageAsFile method

This method allows the actual display content to be stored into an image file on the hard disk. Any existing signature will be ignored for saving. The image has the size and resolution of the screen on the device used. The colour depth depends on the file type and the device used.

Available from Version 8.2.1.15 onwards. The status described is available from Version 8.2.1.16.

```
LONG STDisplaySaveImageAsFile(LPCWSTR szPath, FILETYPE nFileType, LONG
nOptions)
```

Parameter	Values	I/O	Description
`LPCWSTR szPath`	`!= NULL`	I	Storage location for the image file as a full path that includes the file name
`FILETYPE nFileType`	`0`	I	Use TIFF with CCITT4 compression (b/w image) or LZW compression (colour image) as the file format
	`1`	I	Use PNG file format
	`2`	I	Use BMP file format. The type is not supported on Linux
	`3`	I	Use JPEG with a quality setting of 75 as the file format
	`4`	I	Use GIF as the file format. The type is not supported on Linux
`LONG nOptions`	Bitmask containing one or more hexadecimal values from the following list:		
	`0x01`	I	The whole content of the display will be stored; The hotspot areas (buttons) stay white in this mode.
	`0x02`	I	Instead of the current display content the content of the whole foreground buffer without the overlay rectangle is saved

Return value	Values	Description
LONG	0	Method was executed successfully
	< 0	Error

The FILETYPE enumeration is defined as follows:

```
kTiff = 0,
kPng = 1,
kBmp = 2,
kJpeg = 3,
kGif = 4
```

The following values defined in the header file can be used for the nOptions parameter:

```
#define STPAD_DIMG_HOTSPOTS     0x01
#define STPAD_DIMG_BUFFER  0x02
```

6.44.1 Usage:

```
LONG nRc = STDisplaySaveImageAsFile("./Image.tif", kTiff, 0);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.45 STDisplaySetStandbyImageFromFile method

This method allows an image, whose path is passed, to be stored permanently in the selected device. The image is automatically displayed when a connection to the device has not yet been opened (while a connection is being established, for example). Although the colour depth is automatically adjusted to the connected LCD, it is still advisable to correctly generate the image beforehand (for example, a 1-bit monochrome image is required for the Sigma model). If the image is too small, it is centred. If the image is too large, it is cropped on the right and at the bottom.

The image is only transmitted when the memory management determines that the image is not already stored in the device. A slide show configuration is removed by calling this method.

Available from Version 8.2.1.15 onwards.

```
LONG STDisplaySetStandbyImageFromFile(LPCWSTR szPath)
```

Parameter	Values	I/O	Description
LPCWSTR szPath	!= NULL	I	Full file path of the image; PNG and TIFF can be used as image formats

Return value	Values	Description
LONG	0	Method was executed successfully
	< 0	Error

6.45.1 Usage:

```
LONG nRc = STDisplaySetStandbyImageFromFile(L"./Image.png");
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.46 STDisplayConfigSlideShow method

With this method, a slide show of permanently stored images can be configured to be played automatically on the target device, if the device is not in use. A possibly saved standby image is removed.

This method only works with Omega, Gamma and Alpha models.

Available from Version 8.2.1.15 onwards.

```
LONG STDisplayConfigSlideShow(LPCWSTR szSlideList, LONG nDuration)
```

Parameter	Values	I/O	Description
`LPCWSTR szSlideList`	`NULL, ""`	I	The slide show will be disabled
	other	I	A list of up to 16 (Gamma) or 32 (Omega and Alpha) IDs of image stores separated by semicolon; these IDs must be reserved previously by the `STDisplaySetTarget()` method and must be filled with text or images; an ID of an image store can be included multiple times; the slide show will be displayed in the given order
`LONG nDuration`	`1000 – 300000`	I	Time in milliseconds for each image to be displayed.

Return value	Values	Description
`LONG`	`>= 0`	Number of images in the slide show
	`< 0`	Error

6.46.1 Usage:

```
LONG nRc = STDisplayConfigSlideShow("5;6;8;5;7", 2000);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.47 STDisplayGetStandbyId method

This method returns the number of images configured for standby operation, as well as a hexadecimal character string that identifies the standby image currently set or the slide show currently configured. Thus it can be checked, for example, whether the current configuration matches the desired one.

Available from Version 8.2.1.15 onwards.

```
LONG STDisplayGetStandbyId(LPCWSTR szId, LONG* pnStringLength)
```

Parameter	Values	I/O	Description
`LPCWSTR szId`	`NULL`	I	The method returns the length of the character string in the `pnStringLength` parameter
	`!= NULL`	I/O	Array in which the character string that identifies the current configuration is written; if the array is too small, the end characters are cut off
`LONG* pnStringLength`	`>= 0`	I/O	Length of the character string or size of the `szId` array in bytes

Return value	Values	Description
LONG	>= 0	Number of reserved permanent stores used for the standby image or the slide show
	< 0	Error

6.47.1 Usage:

```
LONG nLen = 0;
LONG nRc = STDisplayGetStandbyId(NULL, &nLen);
if (nRc == 0)
    wprintf(L"No standby mode configured!");
else if (nRc > 0)
{
    WCHAR* szId = new WCHAR[nLen / sizeof(WCHAR)];
    nRc = STDisplayGetStandbyId(szId, &nLen);
    if (nRc > 0)
        wprintf(L"%d stores configured, ID is: %s", nRc, szId);
    delete [] szId;
}
if (nRc < 0)
    wprintf(L"Error %d", nRc);
```

6.48 STControlGetVersion method

This method returns the version number of the component.

Available from Version 8.2.0.

`LONG STControlGetVersion(WCHAR szVersion[16])`

Parameter	Values	I/O	Description
WCHAR szVersion[16]	!= NULL	O	Buffer where the version number is written
Return value	**Values**		**Description**
LONG	0		Method was executed successfully
	< 0		Error

6.48.1 Usage:

```
WCHAR szVersion[16];
LONG nRc = STControlGetVersion(szVersion);
if (nRc < 0)
    wprintf(L"Error %d", nRc);
else
    wprintf(L"Version: %s", szVersion);
```

6.49 STControlSetAppName method

This method can be used to assign the name of the application that uses the component. Users can use this name to exclusively assign one or more image memories. Please refer to chapter 'Exclusive use of non-volatile memory' for details.

Available from Version 8.2.1.15 onwards.

```
VOID STControlSetAppName(LPCWSTR szName)
```

Parameter	Values	I/O	Description
`LPCWSTR szName`	`NULL`	I	Application does not use any memories exclusively
	`!= NULL`	I	Name of the application (may contain spaces)
Return value	**Values**	**Description**	
–	–	-	

6.49.1 Usage:

```
STControlSetAppName(L"My Great App");
```

6.50 STControlGetErrorString method

This method returns an error description in German, English, French or Italian, depending on the system language.

Available from Version 8.2.0.

```
LONG STControlGetErrorString(LPCWSTR szError, LONG* pnStringLength, LONG
nErrorId)
```

Parameter	Values	I/O	Description
`LPCWSTR szError`	`NULL`	I	The method returns the length of the error description in the `pnStringLength` parameter
	`!= NULL`	I/O	Array in which the error description is written; if the array is too small, the end characters are cut off
`LONG* pnStringLength`	`>= 0`	I/O	Length of the error description or size of the `szError` array in bytes
`LONG nErrorId`	`0`	I	The description of the last error that occurred will be returned.
	`< 0`	I	Error number, for which the description should be returned.
Return value	**Values**	**Description**	
`LONG`	`0`	Method was executed successfully	
	`< 0`	Error	

6.50.1 Usage:

```
LONG nLen = 0;
LONG nRc = STControlGetErrorString(NULL, &nLen, 0);
if (nRc == 0)
{
    WCHAR* szError = new WCHAR[nLen / sizeof(WCHAR)];
    nRc = STControlGetErrorString(szError, &nLen, 0);
    if (nRc == 0)
        wprintf(szError);
    delete [] szError;
}
```

6.51 STControlSetCallback method

This method defines a callback routine that is called if one of the events is triggered. For more information, see the Events chapter.

Available from Version 8.2.0.

```
VOID STControlSetCallback(CBPTR pCallback, LPVOID pCustomPar)
```

Parameter	Values	I/O	Description
CBPTR pCallback	NULL	I	No callback is used
	!= NULL	I	Pointer to the callback routine
LPVOID pCustomPar	all	I	Any parameter that is passed when the callback routine is called; normally a pointer to the class whose methods are called from the callback routine
Return value	**Values**	**Description**	
–	–	-	

The CBPTR type is defined as follows:

```
typedef VOID (*CBPTR)(LONG nEvent, LPVOID pData, LONG nDataSize, LPVOID
pCustomPar);
```

Parameter	Values	I/O	Description
LONG nEvent	Index of the triggered event from the following list:		
	0	I	DeviceDisconnected()
	1	I	SensorHotSpotPressed()
	2	I	SensorTimeoutOccured()
	3	I	DisplayScrollPosChanged()
	4	I	SignatureDataReceived()
LPVOID pData	!= NULL	I	Array of data that is given as a parameter to the event; please refer to the respective event for a description of the parameters
LONG nDataSize	> 0	I	Size of the pData array in bytes
LPVOID pCustomPar	all	I	Parameter that was passed when calling STControlSetCallback(); normally a pointer to the class whose methods are called from the callback routine
Return value	**Values**	**Description**	
–	–	-	

The following values defined in the header file can be used for the nEvent parameter:

```
#define STPAD_CALLBACK_DISCONNECT    0
#define STPAD_CALLBACK_HOTSPOT  1
#define STPAD_CALLBACK_TIMEOUT  2
#define STPAD_CALLBACK_SCROLL        3
#define STPAD_CALLBACK_SIGNATURE 4
```

6.51.1 Usage:

```
VOID Callback(LONG nEvent, LPVOID pData, LONG nDataSize, LPVOID
             pCustomPar)
{
    if (!pCustomPar)
        return;

    CMyClass* pCls = (CMyClass*)pCustomPar;
    switch (nEvent)
    {
        case STPAD_CALLBACK_DISCONNECT:
            if (nDataSize >= sizeof(LONG))
               pCls->DeviceDisconnected(*(LONG*)pData);
           break;
       case STPAD_CALLBACK_HOTSPOT:
           if (nDataSize >= sizeof(LONG))
               pCls->SensorHotSpotPressed(*(LONG*)pData);
            break;
       case STPAD_CALLBACK_TIMEOUT:
           if (nDataSize >= sizeof(LONG))
               pCls->SensorTimeoutOccured(*(LONG*)pData);
           break;
       case STPAD_CALLBACK_SCROLL:
           if (nDataSize >= (2 * sizeof(LONG)))
                pCls->DisplayScrollPosChanged(*(LONG*)pData,
                                          *((LONG*)pData + 1));
           break;
       case STPAD_CALLBACK_SIGNATURE:
           if (nDataSize >= (4 * sizeof(LONG)))
                pCls->SignatureDataReceived(*(LONG*)pData,
                      *((LONG*)pData + 1), *((LONG*)pData + 2),
                      *((LONG*)pData + 3));
           break;
     }
}

CMyClass::CMyClass()
{
    STControlSetCallback(&Callback, (VOID*)this);
}
```

6.52 STControlExit method

This method releases used resources; it must be called before the component is de-initialised.

Available from Version 8.2.0.

```
VOID STControlExit()
```

Parameter	Values	I/O	Description
–	–	-	-

Return value	Values	Description
–	–	-

6.52.1 Usage:

```
STControlExit();
```

7 Events

Events are named according to the following naming convention:

- General hardware events begin with ‘**Device**’
- Events that apply to the signature begin with ‘**Signature**’
- Sensor events begin with ‘**Sensor**’
- Display events begin with ‘**Display**’

The component uses a callback mechanism to pass events through to the application. For more information, see the `STControlSetCallback()` method.

7.1 DeviceDisconnected event

This event is called as soon as a device is disconnected through an external event (e. g. unplugging the device).

Available from Version 8.2.0.

```
VOID DeviceDisconnected(LONG nIndex)
```

Parameter	Values	Description
`LONG nIndex`	`>= 0`	Index of the disconnected device
Return value	**Values**	**Description**
–	–	-

7.1.1 Usage:

```
VOID CMyClass::DeviceDisconnected(LONG nIndex)
{
    wprintf(L"Device %d disconnected!", nIndex);
}
```

7.2 SignatureDataReceived event

This event is called when signature data is received from the pad.

Available from Version 8.2.0.

```
VOID SignatureDataReceived(LONG nXPos, LONG nYPos, LONG nPressure, LONG
nTimestamp)
```

Parameter	Values	Description
`LONG nXPos`	`>= 0`	x value of the received data record
`LONG nYPos`	`>= 0`	y value of the received data record
`LONG nPressure`	`0 – 1024`	Pressure value of the received data record
`LONG nTimestamp`	`>= 0`	Timestamp of the received data record
Return value	**Values**	**Description**
–	–	-

7.2.1 Usage:

```
VOID CMyClass::SignatureDataReceived(LONG nXPos, LONG nYPos, LONG
                                 nPressure, LONG nTimestamp)
{
    wprintf(L"X: %d; Y: %d; P: %d; T: %d", nXPos, nYPos,
                                      nPressure, nTimestamp);
}
```

7.3 SensorHotSpotPressed event

This event is called as soon as the user lifts the pen off a rectangle defined with `STSensorAddHotSpot()`.

Available from Version 8.2.0.

`VOID STSensorHotSpotPressed(LONG nHotSpotId)`

Parameter	**Values**	**Description**
`LONG nHotSpotId`	`>= 0`	ID of the activated hotspot
Return value	**Values**	**Description**
–	–	-

7.3.1 Usage:

```
VOID CMyClass::SensorHotSpotPressed(LONG nHotSpotId)
{
    wprintf(L"Hotspot %d!", nHotSpotId);
}
```

7.4 SensorTimeoutOccured event

This event is called as soon as the timer started with `STSensorStartTimer()` has expired.

Available from Version 8.2.0.

`VOID STSensorTimeoutOccured(LONG nPointsCount)`

Parameter	**Values**	**Description**
`LONG nPointsCount`	`>= 0`	Number of points captured if any
Return value	**Values**	**Description**
–	–	-

7.4.1 Usage:

```
VOID CMyClass::SensorTimeoutOccured(LONG nPointsCount)
{
    wprintf(L"Timeout, captured points: %d!", nPointsCount);
}
```

7.5 DisplayScrollPosChanged event

This event is called as soon as the scroll position of the display contents has changed.

Available from Version 8.2.0.

```
VOID STDisplayScrollPosChanged(LONG nXPos, LONG nYPos)
```

Parameter	Values	Description
`LONG nXPos`	`>= 0`	Horizontal offset of the display contents to the left, in pixels
`LONG nYPos`	`>= 0`	Vertical offset of the display contents to the top, in pixels
Return value	**Values**	**Description**
–	–	-

7.5.1.1 Application

```
VOID CMyClass::DisplayScrollPosChanged(LONG nXPos, LONG nYPos)
{
    wprintf(L"Scroll pos: %d / %d", nXPos, nYPos);
}
```

www.evolis.com