Documentation

signoPAD-API Java

Software components for communication with
signotec Sigma, Zeta, Omega, Gamma, Delta and Alpha LCD pads

Version: **4.1**
Date: **18.09.2020**

© signotec GmbH
www.signotec.de
Tel.: **(+49-2102) 5357-510**
E-mail: **info@signotec.de**

Contents

Legal notice

1 Document history

Version	Date	Person responsible	Status/note
3.12	16 August 2017	Markus Mensinger	Changes to section 8.1.3
3.13	10 November 2017	Markus Mensinger	Changes to sections 8.3, 8.3.7.
3.14	30 November 2017	Markus Mensinger	Changes to sections 3.1, 10.26–10.27, 10.35.2, 10.48–10.49, 10.56, 10.60–10.61, 10.66–10.67, 10.79–10.80, 10.83–10.84, 10.92 Sections 4.3, 10.93, 22 added.
3.15	9 January 2018	Markus Mensinger	Changes to sections 3.1–3.4
3.16	24 January 2018	Markus Mensinger	Changes to section 3.6.
3.17	1 June 2018	Markus Mensinger	Changes to sections 6.11, 8.1.3, 9.21, 10.60-10.61, 10.66, 10.90-10.91 Sections 6.10–6.13, 9.23–9.35, 10.94–10.97 added
3.18	7 June 2018	Markus Mensinger	Changes to section 3.1
3.19	22 February 2019	Markus Mensinger	Changes to section 21
3.20	17 September 2019	Markus Mensinger	JWinPointer library added Changes to sections 3.1, 3.7, 8.3.1 Sections 3.8 and 6.14 added
3.21	18 November 2019	Markus Mensinger	Changes to section 10.7, 10.90
4.0	18 March 2020	Markus Mensinger	Interface documentation moved to new JavaDoc documentation, sections 4.1 and 4.2 added, Java 6 removed, changes to sections 3.1 and 6
4.1	18. September 2020	Markus Mensinger	Zeta pad added to sections 5.1.1, 5.2.1 Changes to section 6

2 Introduction

signoPAD-API Java contains several components that allow programmers to implement a wide range of functions related to electronic signature capture and integrate them in their applications. This covers not only the capture of the actual signature, but also the display of graphics, text and buttons on a signotec LCD pad or any pen display.

This document provides an introduction to the API as well as full details on the technical options and requirements.

Interface description

The interface description for programmers contains technical information on each of the individual Java classes. The documentation (in JavaDoc format) is located in the `doc/javadoc` folder of the delivery package.

```
doc/javadoc/index.html
```

3 System requirements

signoPAD-API Java can run with the full range of functions on all Windows versions starting with Windows 7. Other operating systems are supported with limited functionality. See below for details.

3.1 JRE dependency

signoPAD-API Java requires version 1.7 or later of the Java Runtime Environment (JRE). Both 32 bit and 64 bit versions are supported.

Java 7

Java version 7u76 or later is required if the application is run in an environment in which the code signing certificate is checked. Older versions of Java do not support the signature algorithm and will reject the library as unsigned. Typical use cases in which only signed code is used include Java Web Start applications and applets.

The table below provides the Internet addresses where Java can be downloaded free of charge.

Version	Download address
Java 7	http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html
Java 8	http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase8-2177648.html
Java 9	http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase9-3934878.html
Java 10	http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase10-4425482.html

All components not included as standard in the JRE are included. The dependencies are listed in the following tables:

Dependency	Note
stpad-native.jar	Only when using the `SigPadFacade`.
bcprov-jdk15on-1.64.jar	-
jna-5.5.0.jar	-
jna-platform-5.5.0.jar	-
swt-gtk-linux-x86_64-3.8.2.jar swt-gtk-linux-x86-3.8.2.jar swt-win32-win32-x86_64-3.8.2.jar swt-win32-win32-x86-3.8.2.jar	Only when using SWT Control `SignatureWidget`
jpen-2.0.0.jar jpen-2-3.dll jpen-2-3-64.dll libjpen-2-4.so libjpen-2-4-x86_64.so	Only when using `PenDisplayFacade` with the `JPen` library. See `PenDisplayFacade.getInstance()` method.
jni4net.j-0.8.8.0.jar jwinpointer-se-1.0.0.jar	Only when using `PenDisplayFacade` with the `JWinPointer` library. See `PenDisplayFacade.getInstance()` method.

3.2 'Pure' Java interface

signotec LCD signature pads can (depending on the configuration) be used via TCP/IP. An interface with limited functionality is available for devices connected in this way. It does not require a platform-dependent library (`stpad-native.jar`) and, as a result, can be used on all systems that can run Java applications. Please refer to chapter 'Starting point: SigPadPureFacade' for details on this interface.

3.3 Windows

In Windows, `stpad-native.jar` is required in the classpath when using `SigPadFacade`. The native library it contains is automatically loaded with JNA.

Alternatively, the .dll file can be extracted from the `stpad-native.jar` and its storage location can be defined with the system property `jna.library.path` at application start. All other .dll and .manifest files, which have been included in the version 8.0.22 and earlier, are no longer required.

3.4 Linux

In Linux, `stpad-native.jar` is required in the classpath when using `SigPadFacade`. The native library it contains is automatically loaded with JNA.

Alternatively, the .so file can be extracted from the `stpad-native.jar` and its storage location can be defined with the system property `jna.library.path` at application start.

When using HID or WinUSB devices, libusb 1.0.16 or higher is also required, which can be downloaded free of charge from http://www.libusb.org.

The authorisations for libusb must be adjusted if necessary. For this, the MODE for USB must be changed to 0666:

```
# libusb device nodes
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", MODE="0666"
```

`udev` must be restarted afterwards:

```
udevadm control --reload-rules
```

In Debian, this setting is located in the file `/lib/udev/rules.d/50-udev-default.rules` or `/lib64/udev/rules.d/50-udev-default.rules`. The setting may be in a different location when using other distributions.

3.5 Java Advanced Imaging

Supported image formats are JPEG, BMP, WBMP and PNG. 'Java Advanced Imaging Image I/O Tools' is required to be able to export signatures in other image formats (JPEG 2000, PNM, RAW or TIFF). These can be downloaded from the following address: https://jai-imageio.dev.java.net/

3.6 Security policies

`SigPadPureFacade` and the `SigPadApi.decryptSignatureData()` method use cryptographic processes that are deactivated by default in Oracle Java up to Version 9.

The files `local_policy.jar` and `US_export_policy.jar` in the folder `%JRE_HOME%/lib/security` must be replaced in Versions up to Java 8u150. The files for the different Java versions can be found in the `policy` directory supplied.

The required policy files of the Java installation are included from Java 8u151 and must be selected with the security property `crypto.policy=unlimited` *before* the JCE framework is initialised. The signoPAD-API Java configures this property automatically as early as possible. However, if the enclosing application uses and initialises the JCE framework, it must be ensured that the security property has been set:

```
java.security.Security.setProperty("crypto.policy", "unlimited");
```

Please refer to the Java 8u151 Release Notes for further information.

3.7 JPen library

When using `PenDisplayFacade`, the JPen library can be used for communication with the device. JPen project page: https://sourceforge.net/projects/jpen

Depending on the system, JPen loads a native library with JNI (see `JRE dependency`). The folder from which the JNI loads the native libraries can be defined with the System.property `java.library.path` at application start.

Here is an example for the relative subfolder 'lib':

```
> java -Djava.library.path=lib
```

A native interface to the pen display is required for capturing with the pen. This is generally contained in the manufacturer's drivers. In Windows, for example, JPen uses the Wintab interface. For signotec devices, all required components are contained in the 'signotec Pen Display Manager' driver.

3.8 JWinPointer library

When using `PenDisplayFacade`, the JWinPointer library can be used for communication with the device. Unlike `JPen`, JWinPointer only runs on Windows, requires .NET 4.5 or higher and uses the system's own API to capture data.

JWinPointer must have write permissions in the user's temp folder to store native libraries there. By default, the folder is obtained from the `java.io.tmpdir` environment variable. A user-defined folder can be set using the `jwinpointer.library.path` variable.

```
> java -Djwinpointer.library.path=C:\Temp
```

A native interface to the pen display is required for capturing with the pen. This is generally contained in the manufacturer's drivers. For signotec devices, all required components are contained in the 'signotec Pen Display Manager' driver.

4 General information

4.1 Major upgrades

This section contains key information you will need when updating from one major version of signoAPI Java to another. It also describes the changes that make it no longer possible to compile a project after the API is updated.

Version 8.x to 9.x

- Outdated (`@Deprecated`) methods and classes deleted. Remove from use *before* upgrading the API.
- Minimum Java version increased from JavaSE 6 to JavaSE 7.
- The `de.signotec.STPad` package has been renamed `de.signotec.stpad`
- The `de.signotec.stpad.api.exceptions.Error` class has been removed; it is no longer required.
- The `de.signotec.stpad.api.PenDisplayLibrary` class has been moved to `de.signotec.stpad.enums.PenDisplayLibrary`.
- The `de.signotec.stpad.api.exceptions.SigPadApiException` top-level exception has been replaced by `SigPadException` (new).
- Methods now also throw standard exceptions (including unchecked exceptions).
    - `java.lang.IllegalArgumentException` in the event of invalid parameters.
    - `java.lang.IllegalStateException` if a function cannot be carried out in the current status, for example, if signature capture is started without first defining the signature area.
    - `java.lang.UnsupportedOperationException` if the function is not supported by the signature device or by the facade.
    - `java.io.IOException` in the event of I/O errors.
    - `java.security.SignatureException` when processing biometric data and errors to cryptography functions.
- The `de.signotec.stpad.api.ForegroundBuffer`, `BackgroundBuffer` and `PermanentStore` classes have been removed. The API now only uses its shared `de.signotec.stpad.api.ImageMemory` superclass. It is possible to use the following methods to identify the memory type:
    - `ImageMemory.isPermanentStore()`
    - `ImageMemory.isForegroundBuffer()`
    - `ImageMemory.isBackgroundBuffer()`
    - `ImageMemory.isOverlayBuffer()`
- The `de.signotec.stpad.control.SignatureJPanel`, `SignatureCanvas` and `SignatureWidget` classes no longer respond to the `SigPadListener.errorOccurred()` event. The error is no longer logged (`Throwable.printStackTrack()`), and the user is no longer shown an error dialog.
- The `de.signotec.stpad.api.SigPadApi.getSignatureData_Byte()` method has been renamed `getSignatureDataBytes()`.
- The `SignatureGraphics.setSampleRate(int)` method has been removed; it is no longer required.
- The `de.signotec.stpad.enums.RSAScheme.NoOID` constant has been renamed `NO_OID`.
- Dependency to the `commons-codec-1.4` library has been removed.
- BouncyCastle updated to version 1.64. If JCE validation errors occur with Java 7 when loading the library, an alternative JAR file should be used.
  "Further note (users of Oracle JVM 1.7 or earlier, users of 'pre-Java 9' toolkits): As of 1.63 we have started including signed jars for 'jdk15to18'; if you run into issues with either signature validation in the JCE or the presence of the multi-release versions directory in the regular 'jdk15on' jar files, try the 'jdk15to18' jars instead."

For more information, see https://www.bouncycastle.org/latest_releases.html.

4.2 Licence keys

signoPAD-API Java is unregistered when it is delivered. In this state, the range of functions is restricted by a watermark when capturing and displaying the signature on a pen display. If you are using a signotec signature LCD pad, *unrestricted* use of the API is possible even without licence keys.

There are two different types of licence keys:

1. Hardware-dependent single-user licences
2. Hardware-independent company licences

For **type 1** licences, an authorisation key (software code) is generated during installation. This key is bound to specific hardware components and is only ever valid for one computer (single user).

After a single-user licence has been purchased, a licence key can be requested and entered with the help of the supplied `license-tool/license-tool.exe` program.

As soon as a valid licence key is available, this is stored in the computer's registry and the demo stamp is removed from all SignoPAD API for Java components.

Type 2 licences are hardware-independent company licences that are not bound to one particular computer/workstation and whose keys are not stored in the registry. Instead, the licence key must be set before using the API by calling the `SigPadApi.setSerialKey()` method.

4.3 Data formats

Signatures can be output in BMP, WBMP, JPEG, PNG, GIF, JPEG 2000, PNM, RAW and TIFF format. Generally speaking, you should use PNG as it offers the best results with the smallest file size. JPEG is an image format with lossy compression and is not recommended.

4.4 SignData structures

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

A separate API (signoAPI) is available for the (ISO-compliant) signature of PDF and TIFF documents. This API includes a wide range of functions for PDF management along with much more. Please contact your signotec contact if you are interested.

SignData structures can be visualised in real time by implementing the `SigPadListener` interface from the `de.signotec.stpad.api.events` package. The `SignatureCanvas` (AWT), `SignatureJPanel` (Swing) and `SignatureWidget` (SWT) classes already implement this interface and are therefore recommended for visualising signature capture.

4.5 Security-critical data

Passwords and private keys are considered security-critical data and must be handled especially carefully. When using this API, you should observe the following precautions in addition to common security standards in order to keep your software's security level as high as possible.

- For passwords, only use data structures that can be overwritten.
  signoPAD-API Java uses the `char[]` data type. Data of an unchangeable type such as `string` cannot be deliberately overwritten or deleted and may remain in RAM for a very long time under certain circumstances.
- Passwords should be deleted immediately after use.
  To minimise the time frame in which passwords can be read from RAM, they should be overwritten immediately after use. The signoPAD-API Java offers the method `KeyLoader.clearPassword(char[])` for this purpose.
- Delete private keys immediately after use.
  For Java 8 and later versions, keys that implement the `Destroyable` interface should be overwritten/rendered unrecognisable in memory via the `destroy()` method once they are no longer needed.

5 Image memory

The signotec LCD signature pads have several image memories, which can be used by different methods of the class `SigPadApi`. An image memory has always 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 signotec 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 and Zeta

The two volatile image memories are the same size as the display (Sigma 320 x 160 pixels, Zeta 320 x 200 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 copying them into the foreground buffer.

5.1.2 Model type Omega

The Omega model has three volatile image memories; two that are double the size of the display (640 x 960 pixels) to be used as foreground and background buffers and one that is the same size as 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 Model type Gamma

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 is the same size as 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 and the operating mode. If possible, the device should always be operated in WinUSB mode. To this end, it is necessary to install the signotec WinUSB driver separately. For more complex representations that consist of several individual images, it can generally be useful to first save them in the background buffer before copying them into the foreground buffer.

5.1.4 Model type Alpha

The Alpha 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 is the same size as 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 and the operating mode. If possible, the device should always be operated in WinUSB mode. To this end, it is necessary to install the signotec WinUSB driver separately. For more complex representations that consist of several individual images, it can generally be useful to first save them in the background buffer before copying them into the foreground buffer.

5.1.5 Model type Delta

The Delta model has three volatile image memories; two that are larger than the display (1280 x 37600 pixels) to be used as foreground and background buffers and one that is the same size as the display (1280 x 800 pixels) to be used as overlay buffer. Its contents can be overlaid over the current screen content.

With the Delta 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 and the operating mode. If possible, the device should always be operated in WinUSB mode. To this end, it is necessary to install the signotec WinUSB driver separately. For more complex representations that consist of several individual images, it can generally be useful to first save them in the background buffer before copying them into the foreground buffer.

5.1.6 Pen display

A pen display does not have a physical image memory. To capture the graphical components, two volatile image memories are simulated in the main memory in the size of the capture area (`PenDisplayFacade.setDisplaySize()`).

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 and Zeta

The Sigma and Zeta models have one non-volatile image memory in the same size as the display (Sigma 320 x 160 pixels, Zeta 320 x 200 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 is double the size of the display (640 x 960 pixels); ten memories are the same size as the display (640 x 480 pixels).

To use a non-volatile memory, this must be reserved first. This is done by calling the `ImageMemory.requestPermanentStore()` method. The size of a memory can be queried using the `ImageMemory.getWidth()` and `ImageMemory.getHeight()` properties.

5.2.3 Model type Gamma

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 `ImageMemory.requestPermanentStore()` method. The size of a memory can be queried using the `ImageMemory.getWidth()` and `ImageMemory.getHeight()` properties.

Unlike Omega, Gamma does not require the use of non-volatile memory to optimise the program in the WinUSB operating mode, as image transmission is very fast. However, it depends on the individual case at hand and the developer should make the final decision.

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 `ImageMemory.requestPermanentStore()` method. The size of a memory can be queried using the `ImageMemory.getWidth()` and `ImageMemory.getHeight()` properties.

Unlike Omega, Alpha does not require the use of non-volatile memory to optimise the program in the WinUSB operating mode, as image transmission is very fast. However, it depends on the individual case at hand and the developer should make the final decision.

5.2.5 Model type Delta

The Delta model has 32 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 32 non-volatile memories are the same size as the display (1280 x 800 pixels).

To use a non-volatile memory, this must be reserved first. This is done by calling the `ImageMemory.requestPermanentStore()` method. The size of a memory can be queried using the `ImageMemory.getWidth()` and `ImageMemory.getHeight()` properties.

Unlike Omega, Delta does not require the use of non-volatile memory to optimise the program in the WinUSB operating mode, as image transmission is very fast. However, it depends on the individual case at hand and the developer should make the final decision.

5.2.6 Pen display

A pen display does not have any non-volatile image memories.

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 memory in a volatile memory and copy from the volatile back 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 description of the `SigPadApi.setImageFromStore()` and `SigPadApi.setOverlayArea()` 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. The static `ImageMemory.requestPermanentStore()` method is used for this. The method returns a reference to a non-volatile image memory, which can then be used to add images or text to this memory. If no non-volatile image memory is available, the method returns a reference to a volatile memory, which can then be used to add images or text to the background buffer.

To compare the image (which may be composed of several texts and images) with the image already stored in the device, the texts 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 `SigPadApi.setImageFromStore()` or `SigPadApi.configSlideShow()` is called. Thus only when one of these methods is called, there will be a noticeable delay.

```
ImageMemory store = ImageMemory.requestPermanetStore();
padApi.setImage(10, 10, img1, store);
padApi.setText(220, 160, SigPadAlign.LEFT, "Signature:", store);
padApi.setImage(220, 400, img2, store);
```

The content can now be copied to a volatile memory, typically the background buffer. The following method has no function, if the images already have been written into the background buffer, but also produces no errors, so it can be called safe.

```
ImageMemory backBuffer = ImageMemory.requestBackgroundBuffer();
padApi.setImageFromStore(dest1, backBuffer);
```

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

```
padApi.setImage(120, 400, img3, backBuffer);
padApi.setText(220, 180, SigPadAlign.LEFT, "01.01.2010", backBuffer);
```

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 copied into the foreground buffer and thus displayed on the screen. The overall image composition has happened before in the background buffer and is thus 'invisible'.

```
padApi.setImageFromStore(backBuffer);
```

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).

6 Starting points

The starting points described below are possible for signature capture. The first step always entails selecting the facade based on the device, technology or connection type you wish to use. This facade is then used to create a `SigPadDevice` object which represents a connected signature device. In the final step, an instance of the `SigPadApi` class is generated for `SigPadDevice`. `SigPadApi` is the most important class when you are using the signature device.

See Interface description for details.

6.1 SigPadFacade

One of the three starting points is the `de.signotec.stpad.api.SigPadFacade` class. It provides general methods for initialising and using signoPAD-API Java and should always be used, provided the platform-dependent libraries for Windows and Linux can be used.

As this is a Singleton class, there is no public constructor; the call is via the `getInstance()` method. In a second step, `initializeApi()` must be called to initialise the interface. Step 3 is the calling of the `getSignatureDevices()` method, which returns all connected pads as `SigPadDevice` objects. With an object like this, the actual main functionality can then be called in the `SigPadApi` class. When all operations have finished, `finalizeApi()` is called to correctly end signoPAD-API Java. Applications, which only need occasional access to the signature device, can execute this process in full for each signature; however, the API should generally be initialised at the start of the application and finalised upon termination.

The call sequence is shown schematically below:

```
SigPadFacade facade = SigPadFacade.getInstance();
// initialize
facade.initializeApi();
// search for devices
SigPadDevice[] pads = facade.getSignatureDevices();
// create API for device
SigPadApi padApi = new SigPadApi(pads[0]);
// some custom operations
padApi.XXX();
// clean up
facade.finalizeApi();
```

6.2 SigPadPureFacade

The second starting point is the `de.signotec.stpad.api.SigPadPureFacade` class. It provides general methods for initialising and using signoPAD-API Java and can only be used if a signotec LCD signature pad is connected via Ethernet in the LAN. The advantage of this starting point is that no platform-dependent libraries are used, which means there is no restriction to Windows and Linux.

As this is a Singleton class, there is no public constructor; the call is via the `getInstance()` method. In a second step, the `getSignatureDevices()` method must be called to search for a connected pad that can be returned as `SigPadDevice`. With this object, the actual main functionality can then be called in the `SigPadApi` class.

The call sequence is shown schematically below:

```
SigPadPureFacade facade = SigPadPureFacade.getInstance();
// search for devices
String address = "192.168.100.100";
int port = 1002;
SigPadDevice[] pads = facade.getSignatureDevices(address, port, false);
// create API for device
SigPadApi padApi = new SigPadApi(pads[0]);
// some custom operations
padApi.XXX();
```

6.3 PenDisplayFacade

The third starting point is the `de.signotec.stpad.api.PenDisplayFacade` class. It provides the option to capture a signature using the pen on a pen display, such as the signotec Delta PD, or using the mouse on the desktop. The graphical Swing component `SignatureJPanel` or `SignatureCanvas` is used as an input area.

The pen signature differs greatly in quality from the mouse signature. The accuracy of the points captured with the mouse corresponds to the screen resolution. The accuracy is generally several times higher when using the pen, depending on the digitiser. The mouse only offers two pressure values (button pressed/button not pressed). Over 1,000 pressure values are possible with the pen.

Drivers are required for capturing with the pen. See the following sections: `JPen library` and `JWinPointer library`.

Using this class involves taking three steps.

1. Creating a new instance with the `getInstance()` method.
2. Configuring the pen display properties such as the serial number, size and resolution of the capture area and whether a pen and/or mouse is active.
3. Calling the `getSignatureDevice()` method, which returns a `SigPadDevice` instance that can be used to call the actual main functionality in the `SigPadApi` class.

The call sequence is shown schematically below:

```
SigPadApi.setSerialKey("myCompanyKey");
PenDisplayFacade facade = PenDisplayFacade.getInstance();
// set the serial number "1111" of the device
facade.setSerialNumber(1111L);
// use pen only, not the mouse
facade.setPenEnabled(true);
facade.setMouseEnabled(false);
// record the pen in area 320x240
facade.setDisplaySize(320, 240);
// create device
SigPadDevice device1 = facade.getSignatureDevice();
if (device1 != null) {
    SigPadApi padApi = new SigPadApi(device1);
    // open/connect device
    SignatureJPanel panel = new SignatureJPanel(padApi, 320, 240);
    padApi.openDevice(panel);
    // custom operations
    padApi.XXX();
}
```

The facade's range of functions is restricted by a watermark when capturing and displaying the signature. The watermark can be removed by purchasing a licence from signotec GmbH. See section Licence keys.

7 Keys and certificates

The `de.signotec.stpad.api.util.KeyLoader` class is a utility class for loading private keys and certificates. It offers simplified access to Java Keystores, PKCS#12 formatted files and certificate files. For more information, see Interface description.

8 Control elements for visualisation

The `de.signotec.stpad.control` package contains visual control elements based on Swing, AWT or SWT for displaying the captured signatures in real time.

All control elements can be operated in two modes. One mode is to display a SignData structure (a signature), and another is to visualise the signature capture.

8.1 SignData mode

The SignData mode is active if a constructor is used without `SigPadApi` parameters.

- The control element is not connected to the pad and cannot be used to display the signature capture.
- It is not possible to interactively control hotspots or scroll with the mouse.
- The control element can be of any size.
- The signature to be displayed is defined using either `setSignatureData()` or `animateSignature()`.

8.2 Signature capture mode

Signature capture mode is active if a constructor is used with `SigPadApi` parameters.

- The control element is connected to the pad and visualises the signature capture on the display. To this end, the control element needs to be connected to the `SigPadApi` class via `SigPadListener`. (`SigPadApi.addSigPadListener(ctrl)`)
- The control element has a fixed size that cannot be changed.
- The `setSignatureData()` and `animateSignature()` methods cannot be used.
- It is possible to interactively control hotspots or scroll with the mouse following activation with `setMouseEnabled(true)`.