Legal notices

Following are legal notices that apply to Sprout and to use of the Sprout Platform and Software Development Kit (SDK).

Copyright Notice

© Copyright 2015 Hewlett-Packard Development Company, L.P.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

License Agreements

Using the Sprout SDK to develop apps and subsequent submission of the apps for possible distribution are governed by two agreements:

- **HP Software Development Kit License Agreement**: This is the agreement that defines terms and conditions for receiving access to the Sprout Platform and the Sprout SDK as software developers. The Sprout Platform and Sprout SDK are needed to develop Sprout apps.
  
  The [HP Software Development Kit License Agreement](#) is displayed before download of the Sprout Platform and SDK, and during installation of the Sprout SDK, and it can be accessed through the Desktop shortcut [Sprout Developer Docs](#).

- **Sprout by HP Application Distribution Program Agreement**: This is the agreement that defines terms and conditions for participation in the Sprout by HP Application Distribution Program, which will allow you to submit your Sprout apps for possible distribution through a variety of different distribution channels offered by HP.

  The [Sprout by HP Application Distribution Program Agreement](#) is available in the Sprout Developer Center.
Trademark Notices

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Intel and Intel RealSense are trademarks of Intel Corporation in the U.S. and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Hewlett-Packard Company under license.

SD, SDHC, and SDXC are trademarks or registered trademarks of SD-3C in the United States, other countries or both.
Contents

Legal notices ................................................................................................................ 2
Copyright Notice ......................................................................................................... 2
License Agreements .................................................................................................... 2
Trademark Notices ..................................................................................................... 3

Contents....................................................................................................................... 4

Part 1: Preliminaries .................................................................................................. 9

Documentation ........................................................................................................... 10
Sprout Platform and SDK Developer Documentation website ................................ 10
API references .......................................................................................................... 11
Sprout Developer Guide ............................................................................................ 12
License agreement for the Sprout SDK ................................................................. 12
Licenses for open source software ........................................................................... 13
Documentation at the Sprout Developer Center ................................................... 13

Support .................................................................................................................... 14
FAQ .............................................................................................................................. 14
Using the Sprout Developer Forum ........................................................................... 14
Reporting issues ......................................................................................................... 16

Part 2: Sprout and the Sprout Platform .................................................................... 18

Sprout ......................................................................................................................... 19
Sprout hardware highlights ....................................................................................... 19
Immersive computer ................................................................................................. 20
Sprout hardware controlled by the Sprout Platform .............................................. 21
Sprout Platform ......................................................................................................... 22
Sprout SDK ................................................................................................................ 22
Sprout Workspace ..................................................................................................... 23
Third-party Sprout software ..................................................................................... 23
Building blocks of Sprout experiences ..................................................................... 23
Multiple windows from one app on the mat screen ................................................................. 110
Windows from different Sprout apps on the mat screen ............................................................ 110

**Touch** ........................................................................................................................................ 111
About the touch mat and mat screen .......................................................................................... 111
User control of mat screen projection and the touch mat ....................................................... 111
Approaches for managing touch events for user touches on the touch mat ............................ 112
Overview of touch management using the operating system .................................................. 113
Overview of touch management using the platform ............................................................... 114
The touch layers ........................................................................................................................... 116
Examples ..................................................................................................................................... 117
Touch events ............................................................................................................................... 119
Step-by-step ............................................................................................................................... 120
The reliability of user touches .................................................................................................... 121

**Moments** ................................................................................................................................ 123
Capturing a moment .................................................................................................................... 123
What a captured moment contains ............................................................................................ 124
Monitoring the progress of a moment capture .......................................................................... 125
Creating a moment from an image ............................................................................................. 127

**Pictures and outlines** ............................................................................................................. 129
Extract options for pictures and outlines .................................................................................. 129
Outlines ........................................................................................................................................ 134
Pictures ........................................................................................................................................ 137
Using the skew angle to make a picture or outline upright ....................................................... 140

**Assisted segmentation** .......................................................................................................... 144
Assisted segmentation of an object in a captured moment ......................................................... 144
Assisted segmentation of the image in a moment created from an image .............................. 146
How an alpha channel is handled during assisted segmentation ............................................ 148

**Object classification** ............................................................................................................ 150
Objects and classification ........................................................................................................... 150

**Text extraction** ...................................................................................................................... 151
Extracting text from a moment ................................................................................................... 151
Impact of prior picture or outline extraction ............................................................................. 151
Text extraction functionality has known limitations ................................................................. 152
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprout Developer Guide</td>
<td></td>
</tr>
<tr>
<td>App logic</td>
<td>153</td>
</tr>
<tr>
<td>Using the skew angle to make text upright</td>
<td>153</td>
</tr>
<tr>
<td><strong>Object recognition and tracking</strong></td>
<td><strong>155</strong></td>
</tr>
<tr>
<td>Overview</td>
<td>155</td>
</tr>
<tr>
<td>Training</td>
<td>157</td>
</tr>
<tr>
<td>Suitable objects for training and recognition</td>
<td>157</td>
</tr>
<tr>
<td>Controlling object tracking</td>
<td>161</td>
</tr>
<tr>
<td>Object-tracking events</td>
<td>161</td>
</tr>
<tr>
<td>App logic (C++)</td>
<td>162</td>
</tr>
<tr>
<td>App logic (WPF)</td>
<td>163</td>
</tr>
<tr>
<td>A sample use case (WPF)</td>
<td>164</td>
</tr>
<tr>
<td><strong>Inter-app and intra-app communication</strong></td>
<td><strong>165</strong></td>
</tr>
<tr>
<td>Overview</td>
<td>165</td>
</tr>
<tr>
<td>Communication handler and observer (C++)</td>
<td>166</td>
</tr>
<tr>
<td>Communication handler (WPF)</td>
<td>167</td>
</tr>
<tr>
<td><strong>Part 6: Troubleshooting</strong></td>
<td><strong>168</strong></td>
</tr>
<tr>
<td>Troubleshooting the Sprout Platform and SDK</td>
<td>169</td>
</tr>
<tr>
<td>Sprout Platform diagnostics log files</td>
<td>169</td>
</tr>
<tr>
<td><strong>Part 7: Appendices</strong></td>
<td><strong>170</strong></td>
</tr>
<tr>
<td>Interfaces and classes</td>
<td><strong>171</strong></td>
</tr>
<tr>
<td>Overview of interfaces and classes in the C++ binding</td>
<td><strong>171</strong></td>
</tr>
<tr>
<td>Overview of interfaces and classes in the WPF binding</td>
<td>177</td>
</tr>
<tr>
<td><strong>Open Source Software</strong></td>
<td><strong>184</strong></td>
</tr>
<tr>
<td>Open source software used by the platform</td>
<td>184</td>
</tr>
<tr>
<td>License files for open source software</td>
<td>186</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td><strong>187</strong></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td><strong>193</strong></td>
</tr>
</tbody>
</table>
Part 1:
Preliminaries

This part of the guide contains information that you should know about the documentation and about how to get support and report defects.
Documentation

After installing the Sprout SDK, you will be able to access documentation through the *Sprout Platform and SDK Developer Documentation* website.

The Sprout SDK installer installs the *Sprout Platform and SDK Developer Documentation* website on your computer’s disc drive. The installer places a shortcut to the website on the Desktop (the shortcut is named “Sprout SDK Docs”):

To view the website, touch-and-hold the shortcut icon, and then tap **Open**, or double-click the icon.

The documentation files are located here:

C:\Program Files\Hewlett-Packard\Sprout\SDK\Documentation\index.html

You can also view the *Sprout Platform and SDK Developer Documentation* website by opening the file *index.html* in that folder in a browser.

**Note:** Additional developer documentation and updates to this documentation are available in the Sprout Developer Center:

http://sprout.hp.com/developer

**Sprout Platform and SDK Developer Documentation website**

The *Sprout Platform and SDK Developer Documentation* website provides links to:

- **API Reference for the C++ Binding**: A website that documents the native C++ binding APIs for apps that are written in C++.
- **API Reference for the WPF Binding**: A website that documents the WPF (Windows Presentation Framework) binding APIs for apps that are written in CLR (Common Language Runtime) languages such as C#.
This guide, the Sprout Developer Guide

A copy of the HP Software Development Kit License Agreement

A folder that contains licenses for open source software that is included in the Sprout Platform

This is the Sprout Platform and SDK Developer Documentation website:

The next three sections describe documentation and other documents and files that are available through the Sprout Platform and SDK Developer Documentation website.

API references

The Sprout Platform and SDK Developer Documentation website directs the developer to subordinate API Reference websites that document language bindings for the Sprout SDK. This release supports two bindings, so there are two subordinate websites:

- **Native C++ Binding for C++ Apps**: This website documents the Sprout SDK APIs in the native C++ binding for apps written in C++.

- **WPF Binding for CLR Language Apps**: This website documents the Sprout SDK APIs in the Windows Presentation Foundation (WPF) binding for apps written in Common Language Runtime (CLR) languages such as C#.

The subordinate websites are generated using doxygen. They provide information about APIs.

To view the API reference for C++ or WPF, open the Sprout Platform and SDK Developer Documentation website, and then tap or click the appropriate link.
This is the API Reference for WPF:

Browse with the upper and left menus. Search with the Search box.

The API Reference websites are also available in the Sprout Developer Center.

**Sprout Developer Guide**

This is the Sprout Developer Guide. It describes Sprout and provides the information needed by software engineers that is not in the API reference.

The guide is available on the Sprout Platform and SDK Developer Documentation website. It is also available in the Sprout Developer Center.

Contents entries are links. You can search for content in the Sprout Developer Guide in a PDF viewer such as Adobe Reader. Cross references, for example to sections and steps, are links. The guide contains an index.

**License agreement for the Sprout SDK**

The Sprout Platform and SDK Developer Documentation website provides a link to a copy of the HP Software Development Kit License Agreement, which is in the file la_sdk_en.rtf. This is the agreement that you accept when installing the Sprout SDK.
Licenses for open source software

The Sprout Platform and SDK Developer Documentation website provides a link to a folder that contains licenses for open source software that is included in the Sprout Platform. For more information about open source software, see “Open source software used by the platform.”

Documentation at the Sprout Developer Center

Additional developer documentation and updates to this documentation are available in the Sprout Developer Center:

http://sprout.hp.com/developer
Support

This chapter directs you to a developer FAQ. It also describes how to use the Sprout developer forum, how to get support, and how to report defects.

FAQ

The FAQ in the Sprout Developer Center provides answers to frequently asked questions. To view the FAQ for Sprout developers:

1. Open a web browser and open the Sprout Developer Center at:
   http://sprout.hp.com/developer
2. In the blue footer of the Sprout Developer Center, below the heading Support, tap FAQ.

Using the Sprout Developer Forum

Use the Sprout Developer Forum to:

- **Ask and answer questions.** Members of the Sprout Development Team will do their best to answer questions. Sprout developers are also welcome to answer questions from other developers, and to contribute to discussions.
- **Report issues.** Depending on what issues you encounter, the Sprout Development Team might request additional information such as diagnostics log files. For more information, see “Reporting issues.”
- **Make enhancement requests.**

**Note:** HP will triage the issues that you report and consider enhancement requests. **We do not** commit to fix reported issues or to make requested enhancements.

Everything in the forum is a discussion that a developer or member of the Sprout Development team starts, and to which a developer or member of the Sprout Development team can contribute.
**Signing in to the Sprout Developer Forum**

To sign in to the Sprout Developer Forum:

1. Open a web browser and open the Sprout Developer Center at:
   
   http://sprout.hp.com/developer

2. Log in with your HP Connected account.
   
   a. On the upper right of the Sprout Developer Center home page, click **Account/Profile** or hover over **Account/Profile** and click **Log In** from the menu that appears.
   
   b. Provide your email address and password, and then click **Sign In**.

   **Note:** If you do not have an HP Connected account, click **Create Account** and create an account.

3. In the main menu of the Sprout Developer Center, click **Forum**.

**Starting a discussion**

To start a discussion:

1. Log in and enter the forum, as explained in “Signing in to the Sprout Developer Forum.”

2. On the upper right of the Forum page, click **Start a Discussion**.

3. Enter a **Discussion Title** for the discussion.

4. Choose a **Category** from the drop-down menu.

5. In the text box, provide any information about the topic of the discussion, for example, a question, issue, or enhancement request.

6. Click **Post Discussion**.

**Contributing to a discussion**

To contribute to a discussion:

1. Log in and enter the forum, as explained in “Signing in to the Sprout Developer Forum.”

2. On the Forum page, navigate to the discussion to which you want to contribute.

3. Click the **Title** of the discussion.

4. In the **Your Comment** text box, enter your comment.

5. Click **Post your comment**.

**Managing your email and popup preferences**

When you participate in discussions, you can choose to be notified by email and/or popup messages when other developers post comments.
To manage your email and popup message preferences:

1. Log in and enter the forum, as explained in “Signing in to the Sprout Developer Forum.”
2. On the Forum page, click the Email Preferences tab.
3. Click check boxes to select or unselect them.
4. Click **Save Preferences**.

**Reporting issues**

To report an issue or enhancement request back to the Sprout development teams, use the Sprout Developer Forum. For information about how to use the forum, see “Using the Sprout Developer Forum.”

Depending on the issue, we might ask you to gather the following information.

**Note:** HP will triage the issues that you report. We do not commit to fix reported issues or to make requested enhancements.

**Information to gather for Sprout Platform and SDK issues**

To report defects, usability issues, and enhancement requests for the Sprout Platform and SDK, gather the following information and include it in a new issue:

- **Hardware configuration:**
  - Dual-display development system: Models of the displays for the mat screen and monitor screen
  - or
  - Sprout immersive computer

- **What you were doing:** Information about the Sprout SDK method that you called, prior SDK method calls if any, and the data provided to the method. The more details you provide, the better.

- **What happened:** What went wrong, did not meet your expectations, or could have been enhanced.

- **When it happened:** Information about when an issue occurred can help us find relevant information in the diagnostics log files.

- **Error information:** Any errors reported by the Sprout Platform or by other software

- **Diagnostics log files:** Attach the Sprout Platform diagnostics log files. For information about the diagnostics log files, see “Sprout Platform diagnostics log files.”
Information to gather for Sprout hardware issues

To report defects, usability issues, and enhancement requests for the Sprout immersive computer, gather the following information and include it in a new issue:

- **Hardware configuration**: Sprout immersive computer

- **What hardware issue was encountered**: Information about an apparent hardware defect, usability issue, or an enhancement request for the Sprout immersive computer. The more details that you provide, the better.

- **Software information**: Include all of the relevant information described in the section “Information to gather for Sprout Platform and SDK issues.”

Reporting an issue

To report an issue or enhancement request back to the Sprout development teams, use the Forum in the Sprout Developer Center. For that procedure, see “Using the Sprout Developer Forum.”
Part 2:

Sprout and the Sprout Platform

This part of the guide gives an overview of Sprout and the Sprout Platform.
Sprout

Sprout is an *immersive computer*. It has components that are typical of an All-in-One PC, and much more. Sprout has a monitor screen, the HP Touch Mat onto which the mat screen is projected, a front-facing webcam, and an assortment of cameras that face the mat. This synergistic combination provides users with an immersive experience for capturing objects, creativity, sharing, and collaboration.

**Sprout hardware highlights**

This diagram shows the hardware of a Sprout immersive computer:

Sprout also has other hardware that is typical for an All-in-One PC, including built-in speakers, a microphone, and a variety of ports. For a complete overview of Sprout Hardware, see “Immersive computer” and the *Sprout User Guide*. For information about the hardware that is controlled through the Sprout Platform, see “Sprout hardware controlled by the Sprout Platform.”
Sprout Developer Guide

**Immersive computer**

Sprout is an immersive computer. Here are some of the aspects of Sprout that make the Sprout experience immersive.

*Note:* Hardware specifications are subject to change. The list below is meant to be indicative and reflects the hardware that was present when this version of the *Sprout Developer Guide* was published.

Sprout has:

- **Processing and graphics power:** A 4th-generation Intel® Core™ i7-4790S desktop processor with Intel® HD Graphics 4600 and onboard NVIDIA GeForce GT 745A discrete graphics with 2GB of dedicated graphics memory.

- **Memory:** 8GB DDR3-1600 memory, expandable to 16GB maximum memory

- **Hard drive:** 1TB 2.5" SATA SSHD (solid state hybrid drive)

- **Touch monitor:** A 23" diagonal, 10-point touch-enabled, Full HD (1920x1080) Wide Viewing Angle, White-LED backlit LCD Display

- **HP Touch Mat:** 20" diagonal, 20-point touch-enabled HP Touch Mat with an ultra-resistant top coating. The touch mat provides a large surface for touch input. This surface is also the backdrop for capturing moments.

  *Note:* If you disconnect the touch mat, you can use Sprout as an All-in-One PC (though you miss the full Sprout experience). Disconnecting the touch mat also turns off the projector, so the mat screen is not present.

- **Second touch screen:** Sprout’s HP DLP Projector projects the mat screen onto the touch mat. The mat screen is XGA (1024x768). The combination of the touch mat and the projected screen provides a second touch screen. Sprout apps can use both screens.

- **A physical desktop experience:** in the sense of a desk on which I might sit a photograph or a magazine. Sprout has a built-in LED desk lamp.

- **Cameras for capturing moments:** The HP High-Resolution Camera with up to 14.6 megapixel resolution and the Intel® RealSense™ 3D Camera allow Sprout to capture moments. Moments contain 2D and 3D information that Sprout can use to distinguish separate objects.

- **Forward-facing webcam:** An HP High Definition 1-Megapixel Webcam

- **Downward-facing webcam:** Downward-facing video is obtained from the high-resolution camera through a virtual camera driver.

- **Audio:** Integrated premium stereo speakers, dual digital MEMS microphones, an Audio-out (headphone)/Audio-in (microphone) jack, and an Audio-out (headphone) jack.

- **HDMI output:** The HDMI output port can be used to duplicate either the monitor screen or the mat screen on an external monitor.
Sprout Developer Guide

- **Wired LAN**: Sprout has 10/100/1000 (gigabit) Ethernet and an Ethernet network jack
- **Wireless LAN**: Sprout has internal 802.11 b/g/n WLAN with Bluetooth 4.0.
- **USB ports**: Dual USB 2.0 ports; and dual USB 3.0 ports, including a powered port to charge phones or other USB devices
- **Media card reader**: HP 3-in-1 single-slot Secure Digital (SD/SDHC/SDXC) media card reader
- **Physical keyboard and mouse**: Wireless Windows-compliant keyboard and mouse

  **Note**: Sprout also displays a Windows touch keyboard on the touch mat when a keyboard is needed.

## Sprout hardware controlled by the Sprout Platform

How you programmatically access the hardware components of a Sprout immersive computer varies depending on the component:

- Some Sprout hardware can only be accessed through the Sprout Platform.
- Some Sprout hardware can only be accessed through the operating system (OS), hardware drivers, and/or other software.
- Some Sprout hardware can be accessed both through the Sprout Platform and through other means, though possibly not at the same time.

Following is information about the hardware that is only controlled by the Sprout Platform, or that is under joint control by the Sprout Platform and the OS or other software (other applications or drivers). Hardware that is not listed in this table (for example, the forward-facing webcam) is controlled through the OS or other software.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Access through the Sprout Platform</th>
<th>Access through the OS</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP Touch Mat</td>
<td>Yes</td>
<td>Yes</td>
<td>Two cases are possible:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- No Sprout touch controller: Access to touch events is through OS APIs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Using a Sprout touch controller: Primary access is through the Sprout Platform. Depending on the use of touch layers, it is possible to handle some touch events using operating system APIs.</td>
</tr>
<tr>
<td>Hardware</td>
<td>Access through the Sprout Platform</td>
<td>Access through the OS</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------</td>
<td>------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Projector</td>
<td>Yes</td>
<td>No</td>
<td>The Sprout Platform controls the projector when displaying app windows on the mat screen.</td>
</tr>
<tr>
<td>High-resolution camera for still images</td>
<td>Yes</td>
<td>No</td>
<td>The Sprout Platform uses the high-resolution camera when capturing a moment.</td>
</tr>
<tr>
<td>Depth camera</td>
<td>Yes</td>
<td>No</td>
<td>The Sprout Platform uses the depth camera when capturing a moment.</td>
</tr>
<tr>
<td>Downward-facing webcam (high-resolution camera for video)</td>
<td>Indirect case</td>
<td>Yes</td>
<td>Sprout apps access the downward facing webcam using a ManyCam driver. The Sprout Platform uses the downward-facing webcam to obtain video frames when doing object tracking.</td>
</tr>
</tbody>
</table>

**Sprout Platform**

The Sprout Platform makes Sprout hardware features (for example, cameras) and software features (for example, the ability to capture moments) available to Sprout apps through language bindings in the Sprout SDK.

The Sprout Platform includes:

- **Sprout Service**: A system service that manages access to Sprout hardware and software resources
- **Sprout Process instances**: Per-user background processes that manage access to per-user data
- **Sprout Settings Panel**: The Sprout Settings Panel allows a user or developer to control some aspects of a Sprout immersive computer or dual-display development system.

For more information about the Sprout Platform, see “Sprout Platform.”

**Sprout SDK**

The Sprout Software Development Kit (SDK) contains language bindings that allow Sprout apps to use Sprout Platform features.
Note: In this release, two bindings are available. A native C++ binding is available for apps written in C++; we refer to this as the C++ binding. A Windows Presentation Foundation (WPF) binding is available for apps written in Common Language Runtime (CLR) languages such as C#; we refer to this as the WPF binding. The language bindings are installed as a part of Sprout Platform installation.

For more information about the Sprout SDK, see “Sprout SDK.”

**Sprout Workspace**

Sprout Workspace is Sprout software that is developed by HP and available on Sprout immersive computers. Sprout Workspace allows users to:

- Capture images of objects that are placed on the touch mat.
- Capture still images from the forward-facing webcam.
- Capture screenshots of the monitor screen.
- Use other images (for example, from a digital camera or the Web).
- Create projects by combining images with drawing, handwriting, and typed text.
- Save projects as PDF files or images, and print them.
- Share creative work with others

**Third-party Sprout software**

Third-party software is available for Sprout. Third-party software can take advantage of Sprout’s special hardware, and extend Sprout’s immersive experiences.

Apps that you develop to take advantage of Sprout’s immersive experiences (Sprout apps) are published in the Sprout Marketplace.

**Building blocks of Sprout experiences**

Sprout combines all of the hardware and software into an immersive computing experience. Immersive computing includes use of dual touch screens, capturing objects, creativity, sharing, and collaboration.

Here we describe some of the core experiences from the viewpoint of the Sprout Platform and the Sprout SDK. These are more “atomic” experiences than gaming and collaboration, for example. These are the building blocks of immersive experiences.
**Dual screens and dual touch surfaces**

Sprout has a touch monitor and a screen that is projected onto a touch mat. A Sprout app can take advantage of both screens. A Sprout app can also manage the touch mat and receive touch events from the touch mat, either directly or via the operating system or both.

Although Sprout apps do not use the Sprout Platform to display windows or manage touch on the monitor, the Sprout Platform does concern itself with the monitor in three regards:

- The Sprout Platform tries to ensure that the optimal screen resolution is used on the monitor (that is, 1920x1080 pixels) and on the touch mat (that is, 1024x768 pixels). Applications should use these resolutions.
- The Sprout Platform tries to ensure that the keyboard is always displayed on the touch mat.
- When an external monitor is connected through the HDMI output port, the Sprout Platform can duplicate the monitor screen or the mat screen onto the external monitor.

**Capturing moments**

Sprout captures moments. A moment is a moment in time. It is like a snapshot with a camera, but Sprout has multiple cameras. The moment contains multiple pictures and metadata related to the pictures.

Sprout is a digital on-ramp for physical objects. Do you want to share digital copies of photographic prints? Are you selling a model car on eBay? Would you like to send your niece a birthday card with an image of flowers picked from your garden? Sprout can help by letting you bring objects from the physical world into the digital world.

To obtain pictures and other data from a moment, an application must call specific methods that take the moment as an input parameter and that return the desired pictures or data. For example, to obtain pictures from a moment, call the method `ExtractPicture` (C++) or `ExtractPictureAsync` (WPF). To obtain outlines from a moment, call the method `ExtractOutline` (C++) or `ExtractOutlineAsync` (WPF).

**Note:** Except for the `Async`, these method names are identical. This is true of many method names. For the sake of brevity, these will be referred to like this, with the C++ method name first and the WPF method name second: `ExtractPicture | ExtractPictureAsync`.

**Creating moments from images from the Web and from digital photos**

When using Sprout, users will want to work with image files from the Web, digital photos taken with cameras and mobile phones, previously scanned images, and so forth. You could provide users with this capability using Windows libraries and image-processing libraries. But Sprout has built-in libraries for image processing, so you might as well leverage those. To do so, you first create a moment from the image file.

**Note:** For this release, the only image-processing capability that is provided is background removal.
A moment created from an image file contains a tree structure, and returns a tree structure when using the extract methods. The tree structure is built into the interface `IPcPicture`. But for a moment created from an image, the tree has only the parent image and no children, because the Sprout Platform does not perform automatic segmentation on the image. Without information from Sprout’s cameras, the Sprout Platform cannot distinguish objects in the image or remove the background, so it does not populate the children in the tree.

**Working with pictures and outlines**

Sprout captures moments. Captured moments contain image data taken with Sprout’s cameras. A Sprout immersive computer has a high-resolution camera and a depth camera. The depth camera provides both infrared image data and depth data. Sprout’s cameras might change. And the camera-related features provided through the Sprout Platform might change. For this reason, the interface `IPcCameraSpecification` defines cameras as *software interfaces* to the hardware cameras. A Sprout app can use the interface `IPcCameraSpecification` to enumerate the cameras that are available on a Sprout immersive computer, and to obtain an instance of the interface `IPcCamera` for each. The interface `IPcCamera` provides information about a specific camera.

**Note:** In this release of the Sprout Platform, you can only extract pictures taken with the high-resolution camera.

After automatic segmentation, which is described in the next section, moments contain images and related data for the individual objects that were on the touch mat when the moment was captured. Pictures are returned in a tree structure. An image of the entire work area of the touch mat is at the top level. Children contain images of the separate objects that the Sprout Platform detected during automatic segmentation.

Moments also contain outlines. After automatic segmentation, the outlines are outlines of the individual objects that were on the touch mat when the moment was captured. For a specific extract option used for pictures and outlines, the tree structures for pictures and outlines match.

**Objects and automatic segmentation**

When Sprout captures a moment, it places data from the cameras in a data structure (the moment). This data includes a high-resolution image of the work area of the touch mat. This is the top-level image in a tree structure that will be constructed that will also contain information about the individual objects.

Sprout can distinguish the individual objects on the touch mat (this process is called *segmentation*) and store images of the objects and related data (for example, bounding rectangles and outlines) in the moment (in the tree structure just mentioned).
Note: Automatic segmentation does not happen when a moment is captured. It happens, if it happens, when a request is made to extract pictures, outlines, or text from a moment. Whether it happens depends on the extract option that is specified. In some use cases, there might be no need to distinguish objects, for example, if only an image of the work area of the touch mat is needed.

During automatic segmentation, the Sprout Platform creates separate images for separate objects, and computes bounding rectangles and outlines.

The Sprout Platform does not perform automatic segmentation when creating a moment from an image file. The segmentation algorithms use data from several of Sprout’s cameras, and rely on sharp distinctions between objects and the white background of the touch mat. Most photographs have foregrounds and backgrounds that are difficult to distinguish automatically. For these images, assisted segmentation is possible. For moments created from images, the programmatic and user-centric processes of assisted segmentation are identical to those for assisted segmentation of images extracted from captured moments. Nonetheless, we distinguish the two cases as assisted segmentation (for images extracted from moments) and background removal (for images extracted from moments created from images).

Classification of objects

Sprout is able to distinguish between flat (two-dimensional) and three-dimensional objects. It can also distinguish between rectangular objects and nonrectangular ones.

The ability to distinguish objects that are both flat and rectangular might be useful to provide document-specific user interaction possibilities, for example.

Assisted segmentation and background removal

For both captured moments and created moments, users can perform assisted segmentation:

- For captured moments, assisted segmentation can be used to refine the edges of the captured images of individual objects on the touch mat.
- For moments created from images, assisted segmentation can be used to remove unwanted parts of a picture, for example, the background. For this reason, we refer to this variety of assisted segmentation as background removal.

Assisted segmentation is powerful because, in many cases, the user only needs to provide a small number of suggestive strokes regarding inclusion and exclusion. The Sprout Platform considers those strokes and applies the apparent intention of the strokes to the entire image. The result can be impressive. For an example, see “Assisted segmentation of the image in a moment created from an image.”

Programmatically, one cycle of assisted segmentation is a two-step process:

1. You refine an outline using a bitmap that represents a set of strokes that describe regions to include and exclude.
2. You refine the corresponding picture based on the new outline.

By next providing the refined outline and picture as the new starting point for refinement, an iterative process of improvement is possible. No algorithm is perfect, some pictures will present challenges, and the user might have given a poor indication of the desired outcome. So, an iterative process makes sense. Remember that you also need to provide the means of stepping backwards (undo) as well as of starting over.

The refined outlines and pictures from assisted segmentation are not stored in the moment. The only things added to the moment after the instant of capture are the results of extraction requests for pictures, outlines, and texts. Whether automatic segmentation occurs depends on the kind of moment (captured or created) and on the supplied extract option for pictures or outlines.

The current implementation of assisted segmentation has two limitations:

- **No more objects (and matching tree structures):** Assisted segmentation does not result in a different number of objects than the number that was detected during the initial automatic segmentation (if there was one). So the tree structures (parent and children) present in the new IPcPicture and IPcOutline obtained using RefineOutline | RefineOutlineAsync and RefinePicture | RefinePictureAsync match those that are present in the IPcPicture and IPcOutline that were extracted from the moment using ExtractOutline | ExtractOutlineAsync and ExtractPicture | ExtractPictureAsync.

  This is not much of a limitation in many cases. If my captured objects are a flower, a stuffed bear, and a trilobite, and I tidy up the edges of each using assisted segmentation, then there is no problem. However, if a user uses an app that provides assisted segmentation, and the user specifies inclusion and exclusion strokes that divide a picture in two pieces (or more), then the Sprout Platform deals with this by keeping the largest piece of the picture (and outline) and discarding the smaller piece or pieces of the picture (and outline), instead of keeping all of the pieces.

- **No holes:** Assisted segmentation cannot be used to cut holes in pictures, or to create outlines that represent holes.

**Text extraction**

Sprout can examine the pictures in captured and created moments to determine whether the pictures contain text. Sprout can recognize printed text, not hand writing. And Sprout can extract the text. This process is sometimes referred to as optical character recognition (OCR), but here the term text extraction makes more sense.

You can use the method ExtractText | ExtractTextAsync to extract the text from the pictures in a moment.

**Communication**

A Sprout app can communicate with other Sprout apps.
To communicate with a different Sprout app, a Sprout app creates an instance of a communication handler, `IPcCommunicationHandler`. A Sprout app can use the communication handler to:

- Identify the communication channel that it will use to send messages, and that other apps will refer to as the destination when sending messages.

- (C++) Receive messages from other Sprout apps by using both a communication handler (an instance of the interface `IPcCommunicationHandler`) and one or more instances of communication observers (instances of the class `IPcCommunicationObserver`). Each `message` event contains an instance of the interface `IPcMessageEventArgs`, which contains the message.

- (WPF) Receive messages from other Sprout apps by subscribing to the `MessageReceived` event that is declared on the interface `IPcCommunicationHandler`. Each `MessageReceived` event contains an instance of the class `PcMessageEventArgs`, which contains the message.

- Send messages to other Sprout apps

- Start and stop the communication channel

Communication features can also be used within a single Sprout app.

**Object recognition and tracking**

The Sprout Platform provides the ability to recognize and track 2D objects and 2D representations of 3D objects (for example, printed photographs).

Training uses a set of one or more bitmap images of specific objects. The bitmaps can be created from pictures extracted from one or more moments, or from other images, for example ones from the Web or taken with a digital camera.

When an instance of the object-tracking handler has been created and started, the platform looks for the objects to be tracked in video frames and raises events when the objects are recognized in the video frames. In C++, one or more object tracking observers are also needed to observe the events.

The object-tracking handler manages the video. You do not need to do so.
The Sprout Platform makes Sprout hardware features (for example, cameras) and software features (for example, the ability to capture moments) available to Sprout apps through language bindings in the Sprout SDK.

Note: In this release, two bindings are available. A native C++ binding is available for apps written in C++; we refer to this as the **C++ binding**. A Windows Presentation Foundation (WPF) binding is available for apps written in Common Language Runtime (CLR) languages such as C#; we refer to this as the **WPF binding**. The language bindings are installed as a part of Sprout Platform installation.

The Sprout Platform:

- Exposes the language bindings of the Sprout SDK, which save developers from having to consider the messy details for each Sprout camera and the touch mat, and from having to use multiple image-processing and optical character recognition (OCR) libraries, instead presenting a unified and user-friendly set of APIs.
- Ensures that Sprout functionality is available reliably by providing the functionality through a system service (the Sprout Service) and per-user background processes (Sprout Process instances).
- Provides features that allow Sprout apps to enhance the user experience, for example, by restricting the keyboard to the mat screen and by bypassing operating system touch handling.

A Sprout app is an app that you write, or a sample app, that uses the Sprout SDK to provide Sprout Platform features.
**Sprout Platform architecture**

The following diagram illustrates the Sprout Platform architecture with and without Sprout hardware. Numbers in the diagram correspond to explanations that follow the diagram.

In the diagram above:

1. A Sprout app uses a language binding to communicate with the Sprout Process.
2. As a first step to make Sprout features available to the Sprout app, the app creates an instance of a platform link, a link between the Sprout app and the Sprout Process. The platform link is an instance of the interface `IPcLink`. Until the platform link is disposed of, the Sprout app can use Sprout features and user data that depends on the platform link persists.

3. Optionally, an app can create additional platform links. Data that can be used by the app is local to a specific platform link. For example, moments captured in one link cannot be accessed through a different platform link.

4. To access Sprout Platform functionality, a second app for the same user (in this case, User A) also creates one or more platform links to the Sprout Process for that user. Again, data is local to the platform link, and it is local to the app.

5. Sprout Platform functionality is provided by both the Sprout Process (which is one per user) and the Sprout Service (which we will refer to as the Sprout Service). The Sprout Service is a Windows service that serves all users. Instances of the Sprout Process communicate with the Sprout Service.

6. All Sprout apps for a different user (for example, User B) use a separate, user-specific instance of the Sprout Process.

7. On a Sprout immersive computer, the Sprout Service communicates with the Sprout hardware on behalf of a Sprout Process and a Sprout app, for example, to capture a moment. Saved capture data is also available for use for testing purposes. The saved capture data is systemwide, not user specific.

8. Without Sprout hardware, the Sprout Platform must use saved capture data.

**Running the Sprout Service and Sprout Process**

To provide Sprout Platform functionality to an app through the Sprout SDK, both the Sprout Service and a Sprout Process must be running:

- The Sprout Service is a Windows service that is used by all users who are logged on to the computer. When Windows starts, it starts the Sprout Service automatically.

- The Sprout Process is a per-user application. The Sprout Service starts a Sprout Process for a user when the user logs on, as well as when the Sprout Service is restarted if no Sprout Process for the user is already running. If a Sprout Process for the user is already running, then the Sprout Service uses that one and does not start a new Sprout Process.

Together, the Sprout Service and Sprout Process provide Sprout features, including control of the touch mat and mat screen. If the Sprout Service is stopped, then the touch mat and projector are turned off.

**Note:** Users should never have to worry about starting or stopping the Sprout Service or Sprout Process. You should never have to worry about starting or stopping the Sprout Service or Sprout Process. With that said, it is possible that one or both might crash or fail to start. We provide information below that will help in these cases.
State diagram for the Sprout Service

This is a state diagram for the Sprout Service. Numbers in the diagram correspond to explanations that follow the diagram.

In the diagram above:

1. Windows starts the Sprout Service automatically when Windows starts.
2. A user can start a stopped Sprout Service by selecting the service in the Windows Task Manager, and then tapping or clicking Start.
3. A user can stop a running Sprout Service by selecting the service in the Task Manager, and then tapping or clicking Stop.
4. A user can restart a running Sprout Service by selecting the service in the Task Manager, and then tapping or clicking Restart.
5. Windows stops the Sprout Service automatically when Windows shuts down.
State diagram for the Sprout Process

This is a state diagram for the Sprout Process. Numbers in the diagram correspond to explanations that follow the diagram.

In the diagram above:

1. The Sprout Service starts the Sprout Process when a user logs on.
2. When switching users, the Sprout Process for the current user is suspended and the Sprout Service starts a Sprout Process for the new user.
3. When switching back to the prior user, the Sprout Process for the current user is suspended and the Sprout Process for the prior user resumes running.
4. When a user signs out, the Sprout Service stops the Sprout Process for the user.
5. When the Sprout Service is restarted, it starts a Sprout Process for the user if one is not already running.
6. If a Sprout Process for the user is already running, the Sprout Service uses that process.
Note the following about starting and stopping the Sprout Service and the Sprout Process:

- The Sprout Service starts the Sprout Process when a user logs in and when the Sprout Service is restarted. If the Sprout Process is already running when the Sprout Service is restarted, the running Sprout Process is used.
- If you stop either the Sprout Service or the Sprout Process, or if either crashes, then the Sprout Platform and its functionality will not be available.
- If the Sprout Service is not running, you can use the Task Manager to start it.
- If the Sprout Process is not running, use the Task Manager to restart the Sprout Service. When the Sprout Service starts, it will start the Sprout Process.
- When either the Sprout Service or the Sprout Process stops and is restarted, in-progress work that was not saved in a Sprout app will be lost.
- When a user logs out, the Sprout Service deactivates the Sprout Process for that user. In a Sprout app, in-progress work that was not saved will not be lost.
- Following a restart of the Sprout Service and the Sprout Process, it might be necessary to restart a Sprout app that was already running.

### Starting, stopping, and restarting the Sprout Service

To start, stop, or restart the Sprout Service using the Task Manager:

1. Start the Task Manager. On the Windows Start screen, type “Task Manager,” and then tap or click Task Manager.
2. Tap or click the Services tab.
3. Tap-and-hold or right-click the entry for Sprout Service, and then tap or click Start, Stop, or Restart.

### Starting the Sprout Process

**Note:** It should not be necessary to start the Sprout Process. The Sprout Process is started when necessary by the Sprout Service.

To start the Sprout Process, restart the Sprout Service. For that procedure, see “Starting, stopping, and restarting the Sprout Service.”

### Stopping the Sprout Process

**Note:** It should not be necessary to stop the Sprout Process.

To stop the Sprout Process:

1. Start the Task Manager. On the Windows Start screen, type “Task Manager,” and then tap or click Task Manager.
2. Tap or click the Processes tab.
3. Tap-and-hold or right-click the entry for Sprout Process, and then tap or click End task.

**Restarting the Sprout Process**

To restart the Sprout Process:

1. Stop the Sprout Process. For that procedure, see “Stopping the Sprout Process.”
2. Restart the Sprout Service. For that procedure, see “Starting, stopping, and restarting the Sprout Service.”
Part 3:  
Getting Started

This part of the guide describes hardware and software requirements for developing Sprout apps, and explains how to obtain and install the Sprout Platform and SDK. It also explains calibration, which might be required periodically on Sprout immersive computers.
Hardware and software for development

Developing Sprout apps on Sprout immersive computers is the best approach, because you have the real hardware and the full user experience. However, it is also possible to develop Sprout apps on specific dual-display development systems. This chapter explains the hardware and software that is needed to develop Sprout apps. It also explains the Sprout Settings Panel, which gives you control over both Sprout immersive computers and dual-display development systems.

Supported hardware configurations

This section describes the hardware configurations that are supported for the development of Sprout apps.

A Sprout immersive-computer

For development of Sprout apps, you can use a Sprout immersive computer. This is the best approach, because you have real hardware and the full Sprout experience.

Dual-display development systems

It is also possible to develop Sprout apps on specific dual-display computer configurations.

Note: Many Sprout features can be simulated using a dual-display setup. But not all features are simulated.

The diagrams below show the two supported dual-display hardware configurations:

- A configuration with two monitors and a PC: This configuration uses two monitors as displays. One or both of the monitors can be touch displays. If there is only one touch display, use it for the mat screen. See “Configuration with two monitors and a PC” for a diagram.

- A configuration with one monitor and a laptop: This configuration uses a touch or non-touch monitor for one display and a laptop for the other display. See “Configuration with one monitor and a laptop” for a diagram.
When setting up the hardware and software, you extend the Desktop over two screens. In the diagrams below, the monitor and mat are identified, and the green numbers indicate the display identifications. Sprout uses the main display (identified with the number 1) as the monitor.

**Configuration with two monitors and a PC**

This is a supported configuration for development of Sprout apps using two monitors and a PC:

1. **Monitor**
   - Touch or non-touch display
   - If the display for the monitor screen cannot be placed above the display for the mat screen, then the displays can be placed side-by-side.

2. **Mat**
   - Touch or non-touch display
   - With only one touch display, use the touch display for the mat screen.
   - Either display can be the main display.

   - **PC**
   - A laptop cannot be used because the Merlin software does not support extension of the desktop over three displays.
Configuration with one monitor and a laptop

This is a supported configuration for development of Sprout apps using a monitor and a laptop:

1. If the display for the monitor screen cannot be placed above the display for the mat screen, then the displays can be placed side-by-side.

2. Either display can be the main display. Using the monitor as the main display makes most sense.

Required software

Following is the software that is required for developing Sprout apps that use the Sprout Platform and SDK:

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>8.1 Professional Update</td>
<td>Windows 7 is not supported. Only a 64-bit operating system is supported for development and for production code. You can use the Sprout SDK to develop 64-bit or 32-bit apps. However, because of the memory consumption by the Sprout Platform when capturing moments and performing other tasks, we recommend that you develop 64-bit apps. You must use an account that has Administrator privileges.</td>
</tr>
<tr>
<td>Microsoft .NET Framework</td>
<td>4.0 or later</td>
<td>Earlier versions of the Microsoft .NET Framework are not supported.</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>2013 Update 3 or Update 4</td>
<td>Supported editions are Ultimate with MSDN, Premium with MSDN, Professional with MSDN, and Professional.</td>
</tr>
</tbody>
</table>
Note: Integrated development environments other than Visual Studio are not supported for development of WPF apps.

**Required software for building the C++ sample app CppQtSample**

The C++ sample app `CppQtSample` was developed using the application and UI framework Qt. To run the sample app executable file, you do not need to install Qt. To build the sample app, you need to install Qt. For development using Visual Studio, you also need the Visual Studio Add-in.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qt</td>
<td>5.3.2</td>
<td>Use Qt Creator to build the sample app <code>CppQtSample</code>.</td>
</tr>
<tr>
<td><strong>Visual Studio</strong></td>
<td><strong>1.2.3</strong></td>
<td>Use Visual Studio to build the sample app <code>CppQtSample</code>. You must also install Qt.</td>
</tr>
<tr>
<td>Add-in 1.2.3 for Qt5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After installing Qt and the Visual Studio Add-in 1.2.3 for Qt5, perform the following configuration steps:

2. In the main menu, tap QT5 > Qt Options.
3. Tap Add, and then enter the following information:
   - Version name: Qt 5.3
   - Path: The path to the location where the Qt SDK was installed, for example: `C:\Qt\Qt5.3.2\msvc2013_64`
4. Tap OK.

**Downloading and installing the Sprout Platform and SDK**

Two separate Windows installers install the Sprout Platform and the Sprout SDK. To install the Sprout Platform and SDK, you must register as a Sprout developer and agree to the *HP Software Development Kit License Agreement*.

For more information and to download the installer, visit this website:

**Calibrating a Sprout immersive computer**

Calibration is performed the first time a user sets up a Sprout immersive computer. It might be required at other times. In general, it is not required after installing a new version of the Sprout Platform (though there might be exceptions to this rule in the future). If captures fail after installing a new version of the Sprout Platform, try calibrating Sprout.

Calibration is alignment of the projected mat screen with the work area of the touch mat. This calibration happens during initial setup and can also be done at any time thereafter. During calibration, Sprout detects the alignment of the projected mat screen and the printed gray border around the work area of the touch mat, and adjusts the projector accordingly.

Calibration can adjust the vertical size, horizontal size, and vertical and horizontal locations of the projected screen. If the projector is rotated relative to the ideal orientation or if the mirror has been bent, then the projection can be warped. Sprout calibration cannot completely correct these problems. But it can “split the difference.” For example, if the lower edge of the mat screen is exactly aligned with the printed border on one corner, but 3mm below the printed border on the other side, calibration will split the difference so that each corner is off by 1.5 mm (one low and one high).

To calibrate a Sprout immersive computer:

1. Start the Windows Control Panel. On the Windows Start screen, type “Control Panel,” and then tap or click **Control Panel**.
2. Tap or click the **Sprout Settings** item. The Sprout Settings Panel opens.
3. Tap or click the **Calibration** tab.
4. Tap or click **Calibrate**.
5. On the monitor, tap or click **Start Calibration**. Do not put your hands or objects on or above the mat until calibration completes.
6. Tap or click **X** to close the Sprout Settings Panel.
7. Tap the **X** to close the Windows Control Panel.

**Sprout Settings Panel**

The Sprout Platform has a settings panel, the Sprout Settings Panel, which is an item in the Windows Control Panel.
Opening the Settings Panel

Note: The Sprout Service and Sprout Process must be running to use the Sprout Settings Panel. It is possible to start the Sprout Settings Panel when the Sprout Service is not running, and to use the Sprout Settings Panel to start the Sprout Service.

To open the Sprout Settings Panel:

1. Start the Windows Control Panel. On the Windows Start screen, type “Control Panel,” and then tap or click Control Panel.
2. Tap or click the Sprout Settings item. The Sprout Settings Panel opens.
3. To enter developer mode, type Alt-Z.
4. Respond Yes on the User Account Control dialog box.

User options in the Sprout Settings Panel

Following are the options in the Sprout Settings Panel that are available to all users:

<table>
<thead>
<tr>
<th>User option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restore default</td>
<td>This option applies solely to Sprout hardware. This option restores Display, Pen and Touch, and Tablet PC settings to values that are needed for a Sprout immersive computer to function correctly.</td>
</tr>
<tr>
<td>settings</td>
<td>Automatic adjustment of the mat screen (projector) brightness is off (unchecked) by default. Tapping Restore Default Settings also restores this default value by unchecking the Adjust automatically check box under Projector brightness.</td>
</tr>
<tr>
<td>Projector brightness</td>
<td>This option applies solely to Sprout hardware. This slider permits manual adjustment of the projector brightness. The check box Adjust automatically permits automatic adjustment based on the ambient light level.</td>
</tr>
<tr>
<td>Note:</td>
<td>The ambient light sensor can also be used to control the brightness of the monitor. This is outside of Sprout Platform control. By default, Adjust my screen brightness automatically is on in Windows. To turn this setting off, go to Settings &gt; Change PC settings &gt; PC and devices &gt; Power and sleep, and then tap the On/Off control below Adjust my screen brightness automatically.</td>
</tr>
<tr>
<td>External monitor</td>
<td>This option applies solely to Sprout hardware. If you attach an external monitor to a Sprout immersive computer using the HDMI output port, you have a choice of duplicating the monitor screen or the mat screen on the external monitor.</td>
</tr>
</tbody>
</table>
### User option

<table>
<thead>
<tr>
<th>User option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calibration</td>
<td>This option applies <em>solely</em> to Sprout hardware. If you have a Sprout immersive computer, you can calibrate the immersive computer. Calibration ensures optimal alignment of the mat screen with the work area of the touch mat.</td>
</tr>
<tr>
<td>About</td>
<td>About provides information about the versions of software and device drivers that are installed on the Sprout immersive computer.</td>
</tr>
</tbody>
</table>

### Developer options in the Sprout Settings Panel

In addition to the options in the Sprout Settings Panel that are intended for use by consumers, the Sprout Settings Panel has a developer mode. Press **Alt-Z** to enter developer mode.
This is a screen capture of the Sprout Settings Panel in developer mode:

Following are the developer options. Most of the developer options in the Sprout Settings Panel can be used both on Sprout hardware and on dual-display development systems.

<table>
<thead>
<tr>
<th>Developer option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit mouse on the mat.</td>
<td>This is intended as a convenience for developers. The Sprout experience is a touch-centric experience, especially for the touch mat. Therefore on a consumer Sprout immersive computer, the mouse is not allowed on the mat screen.</td>
</tr>
<tr>
<td>Permit all non-Sprout applications on the mat.</td>
<td>This is intended as a convenience for developers. The Sprout experience controls which applications have access to the touch mat. By default, only Sprout apps and Modern apps can appear on the touch mat. This developer option permits all Desktop apps to appear on the touch mat.</td>
</tr>
<tr>
<td>Developer option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Save captured data</td>
<td>If you have a Sprout immersive computer, you can save captured data acquired during a moment capture for use on dual-display development systems, and for testing purposes.</td>
</tr>
</tbody>
</table>
| Use saved capture data  | On dual-display development systems, you cannot capture moments using the Sprout cameras. Use this option to use saved capture data when capturing moments. The Sprout SDK contains saved data from three moment captures. You can also use saved capture data from a Sprout immersive computer. Captured moments are created from the saved capture data—the moments contain data from all of the Sprout cameras.  

**Note:** Following installation of the Sprout Platform, Use saved capture data is disabled on a Sprout immersive computer and enabled on a dual-display development system. |

**Diagnostics**

The **Diagnostics** tab in the Sprout Settings Panel provides easy access to the Sprout diagnostics log files.

For information about the Sprout diagnostics log files, see “Sprout Platform diagnostics log files.”

**Note:** The **Diagnostics** tab also has a section for starting, stopping, and restarting the Sprout Service. Do not use this functionality, Use the Windows Task Manager instead.

**Approved Applications**

The **Approved Applications** tab in the Sprout Settings Panel allows you to manage a list of Desktop applications that are permitted to appear on the touch mat.

Lists of approved applications are per-user.
Saving capture data and using saved capture data

Using the Sprout Settings Panel, you can instruct a Sprout immersive computer to save capture data. The saved data consists of images from the cameras and some other data. The data is pre-moment data from which moments can be constructed. You can also use the Sprout Settings Panel to instruct a Sprout immersive computer or a developer system to use saved capture data.
The following diagram shows the decision that the Sprout Platform makes regarding saving capture data and using saved capture data.

A moment created from saved capture data contains the same content as a captured moment. Because the moments have the same content, all subsequent processing is identical.

If you have a Sprout immersive computer, as well as dual-display development systems, you could:

1. Save captured data on the Sprout immersive computer.
2. Copy the saved capture data to the dual-display development systems.
3. Develop and test on the dual-display development systems, using the saved data.

There is no reason to both use saved capture data and to save capture data, at the same time on a Sprout immersive computer. All that does is to create extra copies of the data that you already have.

Following installation of the Sprout Platform and SDK, use of saved capture data is enabled if you are running a dual-display development system and disabled if you have a Sprout immersive computer.
**Saving captured data**

The Sprout Platform contains several sets of saved capture data for use when Sprout hardware is not available, that is, for use on dual-display development systems. The Sprout Platform processes this data as if it had just been captured, and then returns the requested information to the application.

While the Sprout Platform is in the mode in which it saves capture data, it does so for every moment capture. By default, data from each moment capture is saved in a new subfolder of this folder:

```
C:\ProgramData\Hewlett-Packard\Sprout\SDK\MockData
```

This location is *not* a per-user location. All data is saved in the same location for all users. You can change the location for saving the data in the Sprout Settings Panel.

To enable or disable saving of captured data:

1. Start the Windows Control Panel. On the Windows Start screen, type “Control Panel,” and then tap or click **Control Panel**.
2. Tap or click the **Sprout Settings** item. The Sprout Settings Panel opens.
3. Press **Alt-Z**.
4. Respond **Yes** on the User Account Control dialog box.
5. Tap or click the **Developer Options** tab.
6. Place a check mark in front of **Save captured data** to enable saving of captured data. Remove the check mark to disable saving of captured data. The default location is shown.
7. Optionally, change the location in which captured data is saved. Tap **Browse** and browse to the new location or enter a location in the text box. To use the default location again, delete the location, remove the check mark, and then tap the check mark again.
8. Tap or click **X** to close the Sprout Settings Panel.
9. Tap the **X** to close the Windows Control Panel.

**Using saved capture data**

While the Sprout Platform is in the mode in which it uses saved capture data, it does so for every moment capture. By default, data for each moment capture is obtained from subfolders of this folder, the same default folder in which capture data is saved, if it is saved:

```
C:\ProgramData\Hewlett-Packard\Sprout\SDK\MockData
```

This location is *not* a per-user location. All data is obtained from the same location for all users. You can change the location for obtaining the data in the Sprout Settings Panel. Capture data is read from the folders in sequence.
To enable or disable use of saved capture data:

1. Start the Windows Control Panel. On the Windows Start screen, type “Control Panel,” and then tap or click Control Panel.
2. Tap or click the Sprout Settings item. The Sprout Settings Panel opens.
3. Press Alt-Z.
4. Respond Yes on the User Account Control dialog box.
5. Tap or click the Developer Options tab.
6. Place a check mark in front of Use saved capture data to enable use of saved capture data. Remove the check mark to disable use of saved capture data.
7. Optionally, change the location from which saved capture data is obtained. Tap Browse and browse to the new location or enter a location in the text box. To use the default location again, delete the location, remove the check mark, and then tap the check mark again.
8. Tap or click X to close the Sprout Settings Panel.
9. Tap the X to close the Windows Control Panel.
Part 4:

Developing Sprout apps

In this part of the guide, we explain how to develop Sprout apps.
The Sprout Software Development Kit (SDK) contains language bindings that allow Sprout apps to use Sprout Platform features.

**Sprout language bindings**

Sprout language bindings are installed by the Sprout Platform installer. To use a Sprout immersive computer as a consumer, only the Sprout Platform installation is required. To develop Sprout apps on a Sprout immersive computer or on other hardware that is supported for software development, both the Sprout Platform and the Sprout SDK must be installed. The Sprout SDK installer installs additional files to assist with software development: sample app source code and binary files, saved capture data, and developer documentation.

The Sprout SDK supports access through these language bindings:

- **C++ binding**: A native C++ binding is available for apps written in C++.
- **WPF binding**: A Windows Presentation Foundation (WPF) binding is available for apps written in Common Language Runtime (CLR) languages such as C#.

The Sprout Platform and SDK:

- Save developers from having to consider the messy details for each Sprout camera and the touch mat, and from having to use multiple image-processing and optical character recognition (OCR) libraries, instead presenting a unified and user-friendly set of APIs.
- Ensure that Sprout functionality is available reliably by providing the functionality through a system service (the Sprout Service) and per-user background processes (Sprout Process instances).
- Provide features that allow Sprout apps to enhance the user experience, for example, by restricting the keyboard to the mat screen and by bypassing operating system touch handling.

A Sprout app is an app that you write, or a sample app, that uses the Sprout SDK to provide Sprout Platform features.

**Note**: In this release, development of Desktop applications is supported. Development of Modern apps is not supported.
A high-level view of the C++ binding

From a high-level view, the C++ binding consists of the following:

- **The public class HPPC**: This class is a factory for creating platform links (instances of the abstract class IPcLink). The method for creating a platform link is `CreateLink`.

- **Instances of the interface IPcLink**: Each instance of the interface IPcLink provides access to functionality of the Sprout Platform. Important data (including moments) is managed by the platform and only persists while the platform link exists. Handlers (for example, a communication handler) and controllers (for example, a touch controller) are also tied to a specific platform link.

- **Synchronous methods in an instance of the interface IPcLink**: These methods are the primary methods for doing anything with the Sprout Platform. For example, methods are available for capturing or creating a moment (`CaptureMoment`), for extracting pictures from a moment (`ExtractPicture`), and for classifying the contents of a moment (`Classify`).

  Note the following about these methods:

  - All methods are synchronous and block the application execution until the method returns.
  - Method overloading is used for many but not all of the methods.
  - These methods mostly return class instances nested into a `PcResult<Type>`. When a method fails, the smart pointer will be null and the code will represent the error.

- **The PcResult template**: The `PcResult` template that is returned from all methods in the instance of the interface IPcLink will nest the actual result of the method, and will control the nested raw pointer using an `std::shared_ptr`. The `PcResult` also holds the `Code` returned by the Sprout Platform. This code works as the error code if something goes wrong during execution.

- **While interacting with a moment**: While interacting with a moment, all methods (such as the ones mentioned above) are present inside an instance of the interface IPcLink.

- **Handler methods in an instance of the interface IPcLink**: These methods are the methods for creating handlers. For example, methods are available for creating a communication handler (`CreateCommunicationHandler`) and for creating a tracking handler (`CreateTrackingHandler`).

  Note the following about these methods:

  - The methods are synchronous.
  - Method overloading is used in some cases.

  These methods return an object capable of communicating with the Sprout Process, so the instance of the interface IPcLink that created this method must be kept alive until this object is not going to be used.
Interfaces that are returned by synchronous methods in an instance of the interface IPcLink:
Most of these interfaces provide access to data. For example, the interface IPcCamera provides information about a specific camera and the interface IPcPicture contains the pictures that were extracted from a moment. Two of the interfaces contain controllers: IPcTouch contains a touch controller and IPcWindowRegistration contains a window controller.

Specification interfaces: Specification interfaces provide access to information that includes the version of the Sprout Platform and of the language binding used by the app; the cameras and screens on the system; Sprout Platform options for pictures, outlines, and text; and classification tags that classify the contents of a moment. The method AccessSpecification in the interface IPcLink returns the top-level specification interface IPcSpecification. Other specification interfaces are accessed through the interface IPcSpecification.

A high-level view of the WPF binding
From a high-level view, the WPF binding consists of the following:

• The public class HPPC: This class is a factory for creating platform links (instances of the interface IPcLink). The method for creating a platform link is CreateLink.

• Instances of the interface IPcLink. Each instance of the interface IPcLink provides access to functionality of the Sprout Platform. Important data (including moments) is managed by the platform and only persists while the platform link exists. Handlers (for example, a communication handler) and controllers (for example, a touch controller) are also tied to a specific platform link.

• Asynchronous methods in an instance of the interface IPcLink: These methods are the primary methods for doing anything with the Sprout Platform. For example, methods are available for capturing or creating a moment (CaptureMomentAsync), for extracting pictures from a moment (ExtractPictureAsync), and for classifying the contents of a moment (ClassifyAsync).

  Note the following about these methods:
  o The method name includes the word Async, which indicates that the method is asynchronous.
  o Method overloading is used for many but not all of the methods.
  o These methods return instances of interfaces as tasks.

• Synchronous methods in an instance of the interface IPcLink: These methods are the methods for creating handlers. For example, methods are available for creating a communication handler (CreateCommunicationHandler) and for creating a tracking handler (CreateTrackingHandler).

  Note the following about these methods:
The methods are synchronous, so the method name does not include the word \texttt{Async}.

- Method overloading is used in some cases.
- These methods return instances of interfaces.

- **Interfaces that are returned as tasks by asynchronous methods in an instance of the interface \texttt{IPcLink}:** Most of these interfaces provide access to data. For example, the interface \texttt{IPcCamera} provides information about a specific camera and the interface \texttt{IPcPicture} contains the pictures that were extracted from a moment. Two of the interfaces contain controllers: \texttt{IPcTouch} contains a touch controller and \texttt{IPcWindowRegistration} contains a window controller.

- **Specification interfaces:** Specification interfaces provide access to information that includes the version of the Sprout Platform and of the language binding used by the app; the cameras and screens on the system; Sprout Platform options for pictures, outlines, and text; and classification tags that classify the contents of a moment. The method \texttt{AccessSpecificationAsync} in the interface \texttt{IPcLink} returns the top-level specification interface \texttt{IPcSpecification}. Other specification interfaces are accessed through the interface \texttt{IPcSpecification}.

- **Interfaces that are returned by synchronous methods in an instance of the \texttt{IPcLink} interface:** The synchronous methods in the interface \texttt{IPcLink} return interfaces that are handlers. For example, \texttt{CreateCommunicationHandler} returns an instance of the interface \texttt{IPcCommunicationHandler}, which is a communication handler.

Note the following about these methods:

- The methods are synchronous, so the method name does not include the word \texttt{Async}.
- Method overloading is used in some cases.

- **What data interfaces contain:** With the exception of specification interfaces, interfaces for accessing data include one or more of the following:

  - **Properties with get and/or set methods:** These provide the means for getting and setting data. Get methods return the requested data as a task, or in some cases a \texttt{bool} (to indicate the state of something). Set methods return a task. For example, the interface \texttt{IPcTouchLayer}, which provides access to a single touch layer, has the following property:

    ```csharp
    bool IsEnabled { get; }
    ```

  - **Enumerable children for the interface, with a get accessor:** Some data structures are tree structures. An enumeration provides access to the children. For example, in the interface \texttt{IPcPicture}, this is the enumeration of children:

    ```csharp
    IEnumerable<IPcPicture> Children { get; }
    ```
Methods: These provide the means for controlling things related to the interface. Similar to the methods in the interface IPcLink, these methods include asynchronous methods (which return a task or an instance of a class or interface as a task) and synchronous methods (which return an interface).

For example, the interface IPcTouchLayer has the following methods:

- `Task<Geometry> GetGeometryAsync();`
- `Task SetGeometryAsync(Geometry geometry);`
- `Task EnableAsync();`
- `Task DisableAsync();`

Event declarations: Events provide the means to inform your app regarding user actions (for example, touching the touch mat) and state changes (for example, when a touch layer is enabled or disabled). For example, the interface IPcTouchLayer declares the following events:

- `event EventHandler<PcTouchUpEventArgs> LayerTouchUp;`
- `event EventHandler<PcTouchDownEventArgs> LayerTouchDown;`
- `event EventHandler<PcTouchMoveEventArgs> LayerTouchMove;`
- `event EventHandler<PcTouchLayerStateEventArgs> LayerStateChanged;`

What handler interfaces contain:

- Properties with get accessors, if needed: These provide the means for getting data. Get methods return the requested data (not as a task). For example, the interface IPcCommunicationHandler has these properties with get accessors:

  ```csharp
  String Identifier { get; }
  String Name { get; }
  ```

  The properties are set when calling the method `CreateCommunicationHandler`.

- Methods: These provide the means for controlling things related to the interface. These methods are asynchronous methods that return a task.

  For example, the interface IPcCommunicationHandler has the following methods:

  ```csharp
  Task StartAsync();
  Task StopAsync();
  Task SendAsync(string destination, Stream message);
  ```
Event declarations: Events provide the means to inform your app regarding user actions (for example, touching the touch mat) and state changes (for example, when a touch layer is enabled or disabled). For example, the interface `IPcCommunicationHandler` declares the following event:

```c
event EventHandler<PcMessageEventArgs> MessageReceived;
```

Exception handling: Every method that returns a task can throw an aggregate exception. The aggregate exception encapsulates an instance of the class `PcException`. Functions that are synchronous return an instance of the class `PcException` without encapsulation. Every instance of the class `PcException` contains an `HRESULT`, which is an integer that identifies the facility and error code. The facility and error code can be used to find out more about the exception.

**Overview of Sprout Platform features**

The Sprout Platform and Sprout SDK exist to give Sprout apps access to some of the hardware of the Sprout immersive computer and to Sprout Platform software features.

The language bindings in the Sprout SDK provide access to the following Sprout Platform features:

- **Platform links**: Calling the method `CreateLink` in the class `HPPC` creates a platform link, which gives a Sprout app access to Sprout Platform features. When a platform link exists between a Sprout app and the Sprout Platform, the platform manages a session for the app. The platform also manages some application data (for example, moments), which only persists as long as the platform link persists.

- **Results and errors**: In the WPF binding, a method call returns either a result or an exception. In the C++ binding, a method call always returns a result, a result code, and a message. Outside of the context of a specific method call, the Sprout Platform can also notify a Sprout app about errors.

- **Specifications**: Specifications provide information about the Sprout immersive computer and about platform options.

- **Information about devices and software**: Specifications provide information about the Sprout immersive computer. The interface `IPcSpecification` provides interfaces for getting:
  - The version of the Sprout Platform and of the binding used by the app
  - Information about the cameras and screens on the system

- **Coordinates and conversions**: Sprout has coordinate systems that can be used to describe the physical location and dimensions of an object on the touch mat, as well as the location and dimensions in pixels. The Sprout Platform provides conversion functions for converting between physical and pixel coordinates.
• **Platform options**: Specifications provide information about platform options. The interface `IPcSpecification` provides interfaces for getting:
  o Sprout Platform options for pictures, outlines, and text
  o Classification tags that can be used to classify the contents of a moment
  o The supported touch layers

• **Intents**: Intents are identifiers of specific implementations of features in the Sprout Platform. For example, the platform has an extract option `Document` for pictures and outlines, as well as a classification tag `Flat`. The processing performed when extracting pictures and outlines of documents or the basis for the classification `Flat` might change. Intents permit the app developer to select a specific implementation.

• **Mat window management**: An app can register with the Sprout Platform an app window that will be displayed on the touch mat. Window registration is **required** to display a window on the mat screen and to use touch-management features. The Sprout Platform manages the registered window.

• **Touch management**: After registering an app window with the Sprout Platform, a Sprout app can use the platform’s touch-management features for the touch mat. When the Sprout Platform manages user touches, some touch events can be received **directly** by the app, **bypassing** the operating system. This is especially useful for indirect touch, when user touches on the touch mat do not steal focus from the monitor screen. Sprout touch management can also manage a keyboard (the Windows touch keyboard), although a Sprout app cannot specify the location or size of the keyboard.

• **Moments**: Sprout captures **moments**. A moment is a moment in time. It is like a snapshot with a camera, but Sprout has multiple cameras. The moment contains multiple unprocessed images and metadata related to the images. A moment is opaque, exposing only a GUID that identifies it. To obtain images and other data from a moment, you must use extract methods that are declared on the interface `IPcLink`.

• **Pictures and outlines**: Sprout apps can extract pictures from moments. Which pictures are extracted depends on what was on the mat when the moment was captured, as well as on the specified extract option.

Sprout apps can also extract outlines from moments. Which outlines are extracted depends on what was on the mat when the moment was captured, as well as on the specified extract option.

For example, when using the extract option `Object`, method calls to extract pictures and outlines will perform **automatic segmentation** (which distinguishes the objects on the mat) and return a tree structure that includes pictures of the individual objects and the corresponding outlines.

A possible use of outlines is to indicate when pictures of specific objects are selected by the user. They are also used for assisted segmentation and background removal.
Sprout Developer Guide

- **Assisted segmentation and background removal**: The Sprout Platform provides functions that can be used to refine outlines and pictures to accomplish assisted segmentation and background removal. Assisted segmentation permits the user to refine an outline and the corresponding image that were extracted from a captured moment. Background removal is corresponding functionality for a moment that was created from an image.

- **Object classification**: When a Sprout app captures a moment, it captures images from several cameras of the objects the user has placed on the touch mat. Sprout can distinguish individual objects, segment them, and provide separate pictures and outlines. It can also classify objects. This classification relies on the ability of the Sprout Platform to consider spatial information in three dimensions. Object recognition is different, depending on training Sprout to recognize specific objects.

  Sprout can classify objects with three classification tags: **Flat**, **Rectangular**, and **ThreeDimensional**. The number of classification tags might increase.

  Consider an example. A user has placed on the touch mat a flower, a postcard, and a square box. The classification tags for these would be:

  Flower: **ThreeDimensional**; postcard: **Flat** and **Rectangular**; square box: **Rectangular**

- **Text extraction**: Sprout can extract text from one or more of the pictures in a moment.

- **Object recognition and tracking**: Sprout apps can recognize and track 2D objects and 2D representations of 3D objects that are placed on or held above the touch mat.

- **Communication**: Sprout apps can communicate pair-wise using a communication handler provided by the Sprout Platform. For C++, a communication observer is also needed. If desired, the communication mechanism can also be used for intra-app communication.
Running the C++ sample app

The Sprout SDK has a C++ sample app, CppQtSample, which demonstrates features provided by the Sprout Platform. An executable file and source code files are provided.

About the sample app

The goals of the sample app CppQtSample are:

- To demonstrate use of some of the Sprout Platform features by applications
- To reveal C++ coding practices when using the C++ binding

This is the main window of the sample app:
Note the following about the user interface for the sample app:

- On the left side, a main menu lets you navigate through the Sprout Platform features.
- When you click an item in the menu, the user interface for that item is displayed. At the top of the right pane is a brief description of the feature. Below the description is a tabbed interface:
  - Click the **Test Function** tab to display the user interface for the feature, and to test the feature.
  - Click the **Code Details** tab to view the code that demonstrates the feature.

### Running the sample app

The sample app *CppQtSample* is supplied as an executable file.

**Note:** The Sprout Service and Sprout Process must be running to run the sample app.

To run the sample app, double-click the executable file *CppQtSample.exe* here:

%ProgramFiles%\Hewlett-Packard\Sprout\SDK\CppQtSample\

### Building and running the C++ sample app using Qt Creator

To use Qt Creator to build the C++ sample app *CppQtSample*:

1. Start Qt Creator, and then tap **File > Open File or Project**.
2. Navigate to the folder:
   
   C:\ProgramData\Hewlett-Packard\Sprout\SDK\Sample Programs\CppQtSample

3. Tap the file *CppQtSample.pro*, and then tap **Open**.
4. Tap **Configure Project**: Make sure that **Desktop Qt 5.3 MSVC2013 64-bit** is selected. Other targets should not be selected.
5. From the menu ribbon on the left side of the window, under **CppQtSample**, choose **Debug** or **Release**.
6. In the main menu, tap **Build > Run qmake**.
7. Tap **Build > Build Project "CppQtSample"**.

   The build creates one of these directories under **Sample Programs**:

   build-CppQtSample-Desktop_Qt_5_3_MSVC2013_64bit-Debug
   build-CppQtSample-Desktop_Qt_5_3_MSVC2013_64bit-Release

   **Note:** If the folder already exists, it is overwritten. The executable file is in the **bin** subfolder.
8. To run the sample app, type **Ctrl+R** or tap **Run** on the left menu ribbon.
Running the WPF sample app

The Sprout SDK has a WPF sample app written in C#, **WpfCsSample**, which demonstrates features provided by the Sprout Platform. An executable file and source code files are provided.

**About the sample app**

The goals of the sample app are:

- To demonstrate use of some of the Sprout Platform features by applications
- To reveal WPF and C# coding practices when using the WPF binding

This is the main window of the sample app:
Note the following about the user interface for the sample app:

- On the left side, a main menu lets you navigate through the Sprout Platform features. The code for the main window of the sample app populates the list of sample code files that demonstrate the features.

- When you click an item in the menu, the user interface for that item is displayed. At the top of the right pane is a brief description of how to demonstrate the feature. Below the description is a tabbed interface:
  - Click the **Test Function** tab to display the user interface for the feature, and to test the feature.
  - Click the **Code Details** tab to view the code that demonstrates the feature.

### Running the sample app

The sample app **WpfCsSample** is supplied as an executable file.

To run the sample app, double-click the executable file **WpfCsSample.exe** here:

```
%ProgramFiles%\Hewlett-Packard\Sprout\SDK\WpfCsSample\n
```

### Building the sample app WpfCsSample

To use Visual Studio to build and run the WPF sample app **WpfCsSample**:

1. Run Visual Studio.
2. Open the project file. Tap **File > Open > Project/Solution**, and then navigate to this location:

   C:\ProgramData\Hewlett-Packard\Sprout\SDK\Sample Programs\WpfCsSample

   Open the project file **WpfCsSample.csproj**.
3. Build and run the application (press **Ctrl+F5**).
Developing apps

Use Visual Studio to develop Windows Presentation Framework (WPF) apps that use the WPF binding of the Sprout SDK. The apps can be written in C# or other Common Language Runtime (CLR) languages. Use Visual Studio or Qt Creator to develop C++ apps that use the C++ binding of the Sprout SDK.

Note: The language bindings in this release of the Sprout SDK support development of Windows Presentation Framework (WPF) and C++ Desktop applications. Use of the Sprout SDK to develop Modern apps is not supported in this release. You can use the Sprout SDK to develop 64-bit or 32-bit apps. However, because of the memory consumption by the Sprout Platform when capturing moments and performing other tasks, we recommend that you develop 64-bit apps.

Development prerequisites

Before you can develop apps that use the Sprout SDK to provide Sprout Platform features, the following must be true:

- You must have either a Sprout immersive computer or one of the dual-display development systems that are described in “Supported hardware configurations.”
- You must use an account that has Administrator privileges.
- (WPF) Visual Studio must be installed.
- (C++) Qt, Qt, Visual Studio, and the Visual Studio Add-in 1.2.3 for Qt5 must be installed.
- The Sprout Platform and Sprout SDK must be installed.
- The Sprout Service and Sprout Process must be running. For information about these processes and how to start them, see “Running the Sprout Service and Sprout Process.”
- If you are developing on a dual-display development system, you must use saved capture data. For that procedure, see “Using saved capture data.”

Developing on a dual-display development system

The Sprout Platform and SDK were designed for use on Sprout immersive computers. To get one hundred percent of the Sprout experience, you need a Sprout immersive computer. That said, the
Sprout Platform and SDK have been designed to work on specific dual-display development systems. On these development systems, the Sprout Platform simulates some aspects of Sprout operation. In some cases, the experience criterion for simulation is to provide an identical or similar experience. In others, the experience criterion is to not return exceptions or errors.

The following table compares the developer experiences when using a Sprout immersive computer and when using a dual-display development system.

**Note:** In no cases should you write code for a developer system, or differently. For example, Sprout has a touch mat, and touch-related APIs rely on that fact. Even though a developer system lacks a touch mat, you write code for the Sprout immersive computer, which has a touch mat. The Sprout Platform does the best it can in these circumstances. In some cases, the best it can do is to return success without doing what was asked of it.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Sprout immersive computer</th>
<th>Developer system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating and disposing of a platform link</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Exception and error handling</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Obtaining information about the system</td>
<td>Returns genuine information.</td>
<td>Returns genuine information for everything except the cameras, which are not present. Data returned for the cameras is mock data.</td>
</tr>
<tr>
<td>Coordinate systems and conversions</td>
<td>Touch coordinates use physical coordinates. Coordinate conversions are identical.</td>
<td>User touches are not processed by the Sprout Platform and no touch events occur. Coordinate conversions are identical.</td>
</tr>
<tr>
<td>Getting and using options</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Registering and unregistering mat windows</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Displayability and displaying mat windows</td>
<td>Relies on information about the projector and the touch mat.</td>
<td>Simulated. Displayability is assumed to be <strong>MatScreenOn</strong> and there are no <strong>DisplayabilityChanged</strong> events.</td>
</tr>
<tr>
<td>Touch on the touch mat</td>
<td>All aspects function.</td>
<td>User touches are not processed by the Sprout Platform and no touch events occur. Method calls return success.</td>
</tr>
<tr>
<td>Aspect</td>
<td>Sprout immersive computer</td>
<td>Developer system</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------------------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Capturing moments</td>
<td>Primarily uses the Sprout cameras.</td>
<td>Must use saved capture data. Capture-progress monitoring does not apply.</td>
</tr>
<tr>
<td>Can use saved capture data.</td>
<td>Can use saved capture data. Capture-progress monitoring is possible.</td>
<td></td>
</tr>
<tr>
<td>Creating moments from images</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Extracting pictures and outlines from moments</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Assisted segmentation and background removal</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Classifying the objects in moments</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Extracting text from moments</td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td>Object recognition and tracking</td>
<td>Possible</td>
<td>Not possible, because this feature relies on the Sprout cameras. API methods return success and no object-tracking events occur.</td>
</tr>
<tr>
<td>Communication</td>
<td>Identical</td>
<td>Identical</td>
</tr>
</tbody>
</table>

### Key points

Following are some key points about the Sprout Platform:

- **Support and reporting defects**: You get support and report defects through the Sprout Developer Center. For information, see “Support.”

- **You might not have Sprout hardware**: You can develop apps that use the Sprout Platform and SDK on specific dual-display development systems. If using a dual-display development system, use saved capture data.

- **Before your app can use Sprout Platform features, it must create a platform link**: The platform link exposes Sprout SDK APIs that provide Sprout Platform features to your app. For WPF, dispose of the platform link when you no longer need Sprout Platform features. For information, see “Platform links.”
• **Moments can be captured:** Sprout captures moments. A moment is a moment in time. It is like a snapshot with a camera, but Sprout has multiple cameras. The moment contains multiple images and metadata related to the images. For information, see “Capturing a moment.”

• **Moments can be created from images:** The Sprout Platform can create a moment from an image. For information, see “Creating a moment from an image.”

• **Captured moments consume a lot of memory:** Manage their creation and disposal with that in mind. For information, see “Disposing of moments and links.”

• **Moments are opaque:** To get anything out of them, you must use extraction methods. For information, see “Pictures and outlines” and “Text extraction.”

• **When you extract something from a moment, it is always returned in a tree structure with a parent, and sometimes but not always with children:** For pictures and outlines, what is extracted depends on what is on the mat and on the extract option used. For text, what is extracted depends on what is on the mat and on the extract option in a prior call to extract pictures or outlines, if any. For example, when using the extract option `Object` when extracting pictures, the top-level picture is a picture of the entire work area of the touch mat. Child pictures are pictures of each object on the touch mat when the moment was captured. If the moment was created from an image, only the parent is present. For information, see “Pictures and outlines” and “Text extraction.”

• **Users can refine the outlines of pictures to improve them and to remove unwanted elements such as the background:** The Sprout SDK has methods for refining outlines and images in a moment. For information, see “Assisted segmentation,” “Refining outlines,” and “Refining pictures.”

• **(WPF) Most methods are asynchronous:** Methods that are meant to be used asynchronously include `Async` in the method name. You should use them asynchronously to provide the best user experience. For more information, see “Using a task-based asynchronous design pattern.”

• **(C++) Most methods are synchronous:** Methods that are meant to be used synchronously do not include `Async` in the method name.

• **Sample apps demonstrate Sprout features and have commented code:** For C++, the sample app `CppQtSample` demonstrates important features and has commented code. For WPF, the sample app `WpfCsSample` demonstrates important features and has commented code.

• **The public APIs in each binding are documented on a website:** The API reference provides information about the APIs. For more information, see “API reference.”

• **Coordinate systems and measurements:** Coordinate systems for the work area on the touch mat, the mat screen, and the monitor screen begin at the point `(X=0, Y=0)` at the upper-left corner of the touch mat work area or screen. The positive X-axis runs down along the left edge of the touch mat work area or screen. The positive Y-axis runs across the top of the touch mat work area or screen. Mat measurements are in millimeters. The work area of the touch mat measures 12 inches high by 16 inches wide, thus the height and width are 304.8 mm and 406.4 mm respectively. Screen measurements are in pixels. We provide utility classes for converting back and forth. For more information, see “Sprout coordinate systems” and “Converting between physical and pixel measurements.”
If you are looking for something, **IPcLink is the place to start**: IPcLink is the interface for the platform link. It exposes methods for the most important features of the Sprout Platform. For more information, view the information about IPcLink in the *API Reference*.

**Coding recommendations for C#**

Following are coding recommendations.

**Using a task-based asynchronous design pattern**

For most methods, we recommend that you use a task-based asynchronous design pattern. These methods have "Async" in the method name, for example, `CaptureMomentAsync`.

Reasons for using a task-based asynchronous design pattern are:

- Using the platform link involves communication between your application and the Sprout Service.
- Capturing a moment and some calculations can take nontrivial amounts of time.
- Asynchronous designs are best for responsiveness of the user interface.

That the method names contain "Async" does not mean that the Sprout Platform takes care of making the method call asynchronous. You must do so, using the `await` keyword. Here are several examples of the approach, taken from the WpfCsSample sample app file `CaptureMatControl.xaml.cs`:

```csharp
// Capture a moment. All captured entities (for example, 
// pictures and outlines) will be extracted from the moment.
using (var moment = await sdk.CaptureMomentAsync())
{
    // Extract the top-level picture and child images.
    var picture = await sdk.ExtractPictureAsync(moment);

    // Extract the top-level outline and child outlines.
    var outline = await sdk.ExtractOutlineAsync(moment);
}
```

The method signatures for these methods in the interface IPcLink all return other interfaces as tasks:

- `Task<IPcMoment> CaptureMomentAsync();`
- `Task<IPcPicture> ExtractPictureAsync(IPcMoment moment);`
- `Task<IPcOutline> ExtractOutlineAsync(IPcMoment moment);`

...
Disposing of moments and links

Moments and links can consume considerable amounts of memory. For this reason, they are disposable (they implement the interface `IDisposable`) and they should be disposed of when they are no longer needed.

**Note:** Dispose of moments before disposing of the platform link, because the moments are defined relative to the platform link. Dispose of the platform link only when you are certain that the user has finished using all of the Sprout Platform features that rely on the platform link.

Moments and links can be disposed of in two ways:

- **Explicitly:** The moment or link can be disposed of with the Dispose method, for example:
  ```csharp
  pcMoment.Dispose();
  pcLink.Dispose();
  ```

- **Implicitly:** By placing the creation of the platform link or moment in the scope of a using statement, the platform link or moment is disposed of automatically when the application control exits the scope of the using statement. For example:
  ```csharp
  using (var sdk = HPPC.CreateLink())
  {
    // Code to accomplish things while the platform link exists
    // Create a moment using the embedded bitmap.
    using (var moment = await sdk.CaptureMomentAsync(bmpSrc))
    {
      // Perform tasks with the moment, for example,
      // extract pictures and outlines and refine them.

      // Close the using statement context, which disposes of
      // the moment.
    }
    // Close the using statement context, which disposes of
    // the platform link.
  }
  ```

Be careful when using `await` statements

When an app is being terminated, make sure that you correctly dispose of the `PCLink` object. If the `PCLink` object is not disposed of, then its worker threads can remain active and that will prevent the app from closing.
A common scenario when this occurs is:

1. An app uses an `await` statement with a method that is being executed on the main thread.
2. The code inside the `await` statement is currently being executed (for example, a moment capture or object tracking is in progress).
3. The app shuts down, for example, when the user taps the X to close the app.

In this case, `await` statements will never return. If you are relying on some logic below the `await` statements to dispose of the `PCLink` object, then disposal of the `PCLink` object will not happen. This is a common caveat with `using` statements—the disposal block won’t be executed in the scenario described here.

**Development approach for C++ using Qt Creator**

To use Qt Creator to develop a C++ app that uses the Sprout Platform:

1. Start Qt Creator, and then click **New Project**.
2. Choose a template for the project, and then tap **Choose**.
3. Create the project using the Qt Creator wizard. Make sure that for **Kit**: Desktop Qt 5.3 MSVC2013 64-bit is selected. Other kits should **not** be selected.
4. Add your code to the project.
5. From the menu ribbon on the left side of the window, under the name of your project, choose **Debug** or **Release**.
6. In the main menu, tap **Build > Run qmake**.
7. Tap **Build > Build Project “Project_name”**.
   The build creates one of these directories under **Sample Programs**:
   ```
   build-Project_name-Desktop_Qt_5_3_MSVC2013_64bit-Debug
   build- Project_name-Desktop_Qt_5_3_MSVC2013_64bit-Release
   ```
   **Note**: If the folder already exists, it is overwritten. The executable file is in the bin subfolder.
8. To run the sample app, type **Ctrl+R** or tap **Run** on the left menu ribbon.

**Development approach for C++ using Visual Studio**

**Note**: Following installation of Qt and the Visual Studio Add-in 1.2.3 for Qt5, perform the configuration steps in “Required software for building the C++ sample app CppQtSample.”

To use Visual Studio to develop a C++ app that uses the Sprout Platform:

1. Run Visual Studio.
2. Create a new project:
   a. Tap or click File > New > Project.
   b. Tap or click a template category. For example, tap Visual C++.
   c. Select your desired template. For example, select Qt5 Projects > Qt Application.
   d. In Name, enter a Project Name.
   e. Optionally, change the Location or Solution name.
   f. Tap or click OK.

3. Tap Build > Configuration Manager.
   a. In the Active solution configuration drop-down menu, choose Release or Debug.
   b. In the Active solution platform drop-down menu, choose Debug or Release. In the Active solution platform menu, choose x64 or Win32. Build should be checked. Tap Close.

4. In Windows Explorer, navigate to the location of the Sprout binding library binary files. You will need the version number of the binding (the folder name) for the steps below. The bindings are in subfolders of this folder:

   %ProgramData%\Hewlett-Packard\Sprout\SDK\Bindings\CPP\

   The folder might contain multiple folders. You generally want the most recent one (the one with the largest number).

   In the steps that follow, you will need the full path names of folders in this directory and the filename of a library file.

5. In your project, add the headers, libraries, and additional dependencies for the C++ binding:
   a. In Solution Explorer, navigate to your project.
   b. Open Configuration Properties:
      (Mouse) Right-click the project name, and then select Properties.
      (Touch) Tap-and-hold the project name, raise your finger to display the right-click menu, and then tap Properties.
   c. Under Configuration Properties, click C/C++.
   d. Add the directory where the C++ binding headers are installed in Additional Include Directories. For example, the end of the line should look like this:

      prior-items;%ProgramData%\Hewlett-Packard\Sprout\SDK\Bindings\CPP\version\include;%\(AdditionalIncludeDirectories\)

      Tap Apply.
   e. Under Configuration Properties, click Linker.
f. Add the directory where the C++ binding libraries are installed in Additional Library
Directories. For example, the end of the line should look like this:

```
prior-items;%ProgramData%Hewlett-Packard\Sprout\SDK\Bindings\CPP\version\v120\x64\lib;
%(AdditionalLibraryDirectories)
```

Tap Apply.

g. Tap the triangle next to Linker to expand the category, and then tap Input.

h. In Additional Dependencies, add:

- For a release version: `Sprout-version.lib`, where version is the version number of the library in the location in step f. For example, the end of the line should look like this:

```
prior-items;Sprout-version.lib;%(AdditionalDependencies)
```

- For a debug version: `Sprout-versionD.lib`, where version is the version number of the library in the location in step f. For example, the end of the line should look like this:

```
prior-items;Sprout-versionD.lib;%(AdditionalDependencies)
```

Tap Apply, and then tap OK.


7. You can now code and build, but your app cannot execute without the Sprout Platform DLLs. Copy all DLLs in this folder:

```
%ProgramData%Hewlett-Packard\Sprout\SDK\Bindings\CPP\version\v120\x64\bin
```

To the folder where your application is, $(OutDir).

To check the location of $(OutDir), tap Project > Properties, select Configuration Properties > General, select Output Directory, and from the drop-down menu on the right, select <Edit...>. Tap the Macros>> button. Browse to $(OutDir) and view the value.

### Development approach for C#

To develop a WPF app in C# that uses the Sprout Platform:

1. Run Visual Studio.

2. Create a new project:
   a. Tap or click File > New > Project.
   b. Select the WPF Application template. Navigate to Installed > Templates > Visual C#, and then tap or click WPF Application.
   c. In Name, enter a Project Name.
d. Optionally, change the Location or Solution name. The check box Create directory for solution should be checked.

e. Tap or click OK.

3. From the drop-down menus at the top of the screen, choose Debug or Release; and choose Any CPU or add the platform x64 or x86.

4. In your project, add a reference to HP.PC.Presentation.dll for the current binding version:

a. In Solution Explorer, navigate to your project and tap or click the triangle to display its contents.

b. Open Reference Manager:
   (Mouse) Right-click References, and then select Add Reference.
   (Touch) Tap-and-hold Reference, raise your finger to display the right-click menu, and then tap Add Reference.

c. Click Assemblies, and then click Extensions.

d. Select the HP.PC.Presentation that has the largest version number (the most recent version), and then tap or click the checkbox so that it is selected.

e. Tap or click OK to close Reference Manager.

5. HP.PC.Presentation.dll is deployed to global assembly cache (GAC), so there is no need to make local copies. Set the property Copy Local for the HP.PC.Presentation reference to False:

a. In your project in Solution Explorer, navigate to the reference HP.PC.Presentation, and then tap or click the reference to display the reference’s properties.

b. Tap or click the property Copy Local, and then choose False from the drop-down menu.

6. Add your code to the project to make use of Sprout SDK classes and functions.

7. Build and run your solution (press Ctrl+F5).
Part 5:
Using Sprout features

Through the language bindings of the Sprout SDK, the Sprout Platform provides many features that make Sprout experiences possible. In this part of the guide, we take you on a feature-by-feature tour.
Platform links

Creating a platform link is the starting point for providing Sprout functionality in your app.

About the platform link

To communicate with the Sprout Process and to provide functionality from the Sprout Platform, an application must create an instance of the platform link as a member of the calling application’s application class. While the platform link instance exists, your app can provide Sprout features.

The platform link:

- Creates and manages a session between your application and the Sprout Service.
- Ensures that resources are initialized.
- Provides Sprout Platform features to your application through the public APIs in a language binding.
- Disposes of the session and associated resources when the platform link is disposed of.

An application only needs a single platform link, which then creates a single session. For testing purposes, it is possible to create multiple instances of the platform link in one app.

Note: Most methods in the Sprout SDK require that a platform link exist. Dispose of the platform link only when Sprout Platform features are no longer needed.

It can be reasonable to use a series of platform links, for example:

1. Create a platform link.
2. Use the platform link while the user does things that require the platform link (for example, while the user is capturing moments).
3. Dispose of the platform link when the user does something else.
4. Create a new platform link if the user returns to the part of the program that has features that depend on the platform link.

The advantage of this serial-link approach is freeing resources. However, this can be a disadvantage if the user needs something in those resources (for example, moments). No significant time penalty is associated with creating or disposing of a platform link.
Different Sprout apps get separate platform links. This allows different Sprout apps to use their own settings as the user changes focus among them.

A platform link does not require a mat window. For example, a small add-on designed to access Sprout Platform features might not need a mat window. If a mat window is needed, create the mat window and register the window with the platform by calling the method `RegisterWindow` | `RegisterWindowAsync` in the interface `IPcLink`. The method `RegisterWindow` | `RegisterWindowAsync` returns an instance of the interface `IPcWindowRegistration` to serve as a window handler for the window. For more information, see “Windows.”

When to create and dispose of the platform link

Three scenarios are possible:

- **Multiple, interacting Sprout features:** In this scenario, a Sprout app uses APIs from the Sprout SDK to provide multiple Sprout features in the app. Data from earlier user actions, such as capturing a moment, must persist so that the user can perform other tasks, for example, extracting images from the moment, annotating the images, and so forth.

  In this case, the platform link should persist from the time that the user can first use a Sprout feature until after it is certain that the user will not use Sprout features. In practice, this probably means creating the platform link when the app starts and disposing of the platform link during the process of shutting down the app.

- **A section of the app, or a mode, provides Sprout features:** This is similar to the prior scenario, but the Sprout features in the app are restricted to a specific section of the app or mode.

  In this case, the platform link should persist from the time that the user first enters the Sprout section or mode until the user leaves the Sprout section or mode, and until after the actions taken there have been finished (for example, by saving captured images).

- **A single, point-of-use Sprout feature:** In this scenario, a Sprout app uses APIs from the Sprout SDK to provide a single Sprout feature in the app. The user action is a simple one, such as capturing a moment, extracting an image from the moment, and pasting the image into a document—possibly all done by tapping a single button, Capture Image.

  In this case, the platform link does not need to persist after the time when the app has finished providing the Sprout feature. In this example, the platform link can be created when the user taps Capture Image and, when the image has been pasted into the document, the platform link can be disposed of.

More about disposing of the platform link

You should dispose of the instance of the platform link for your application. It is important to dispose of the platform link instance, to prevent memory leaks.

When you dispose of the instance of the platform link for your application, all session data, including user data, are disposed of. In most cases, you probably want to dispose of the platform link only when closing a Sprout app.
It is important to not dispose of the platform link prematurely. The moment, which is the collection of all Sprout camera data captured at a specific moment, depends on the session and platform link. When the platform link is gone, the session and moment are gone. When the platform link is gone, the user has some limited ability to continue to manipulate what has already been extracted from a moment, but the user will not be able to go back to the moment for anything. Attempting to use a moment after the platform link has been disposed of will result in runtime exceptions (in WPF) or in PcResult instances that contain error codes (in C++).

**Note:** Before you dispose of the platform link instance, you should ensure that all user data has been saved, for example, by giving the user the opportunity to save unsaved data.

**Coding styles for platform links in C#**

This section describes two coding approaches for creating and disposing of the platform link:

- **Larger scope:** When the scope of Sprout feature use is large, the platform link should be closed explicitly when Sprout features are no longer needed, and in exception handling. Disposing of the platform link in all cases, including the “happy path” and in exception cases, prevents memory leaks. This approach is described in “Coding style for a larger scope of Sprout feature use.”

- **Smaller scope:** When the scope of Sprout feature use is small, it is useful to use a using statement to constrain the scope of the platform link. When the using statement is closed, the platform link is automatically disposed of. This approach is described in “Coding style for a smaller scope of Sprout feature use.”

**Coding style for a larger scope of Sprout feature use**

This code snippet shows how to create and dispose of the platform link when the Sprout app has a large scope of Sprout feature use. You can do so by using a private, read-only attribute of a class, without a using statement to constrain the scope of the platform link. For an example of this approach, see the Communication sample in the sample app WpfCsSample.

**Coding style for a smaller scope of Sprout feature use**

This code snippet shows how to create and dispose of the platform link when the Sprout app has a small scope of Sprout feature use. You can do so by using a using statement to constrain the scope of the platform link. For an example of this approach, see the Versions sample in the sample app WpfCsSample.

You do not need to dispose of the platform link in the catch clauses. If they are reached, the platform link has already been disposed of.
Events

A number of features provided by the Sprout Platform rely on platform events. The features include window management, touch management, and inter-app (or intra-app) communication.

Note: The focus here is on events that the platform raises. In some cases, Windows events are also useful. We will also mention these cases.

Displayability of mat windows

When an app wants to display a window on the touch mat, it might not be able to do so. The touch mat might not be connected, or the touch mat might be connected but the projector might be off.

To handle the displayability of the mat window, an app might want to do one or both of these:

- Query the displayability state of a registered window before displaying it. If the displayability state is MatScreenNotAvailable, an app could instruct the user to connect the touch mat.
- Subscribe to or observe the displayability-change event to be apprised about changes in the displayability of the mat window.

Overview for C++

The Sprout SDK provides the following functionality related to the displayability of mat windows:

- The virtual method Displayability is declared on the interface IPCWindowRegistration. The method returns an instance of the class PcDisplayability.
- The method Updated that is declared on the interface IPCDisplayabilityObserver received a displayability-change event, which is raised each time the displayability of the active registered window changes.
- The event arguments for the displayability-change event are an instance of the interface IPCDisplayabilityChangeEventArgs, which declares virtual methods for obtaining the prior and new values of the enumeration PcDisplayability.
- The enumeration PcDisplayability enumerates the displayability states.
Overview for WPF

The Sprout SDK provides the following functionality related to the displayability of mat windows:

- The property `Displayability` is declared on the interface `IPcWindowRegistration`. The property has a get accessor that returns an instance of the interface `PcDisplayability`.

- The event `DisplayabilityChanged` is declared on the interface `IPcWindowRegistration`. The event communicates to an app that a change in the displayability state of a registered window has occurred.

- The event arguments for the event `DisplayabilityChanged` are an instance of the class `PcDisplayabilityEventArgs`, which specifies the prior and new values of the displayability state, using values of the enumeration `PcDisplayability`.

- The enumeration `PcDisplayability` enumerates the displayability states.

Displayability states

Following are the displayability states.

<table>
<thead>
<tr>
<th>State</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatScreenNotAvailable</td>
<td>The mat screen is turned off because the touch mat is not connected.</td>
</tr>
<tr>
<td></td>
<td>The platform cannot turn on the mat screen when the method <code>Display</code> or <code>DisplayAsync</code> that is declared on the interface <code>IPcWindowRegistration</code> is called. Even though the mat screen is turned off, the platform places the window on the mat screen, which will allow the window to be displayed when the touch mat is connected and the mat screen is turned on.</td>
</tr>
<tr>
<td>MatScreenOn</td>
<td>The mat screen is turned on and is visible.</td>
</tr>
<tr>
<td></td>
<td>A user can see and interact with the window after the app calls the method <code>Display</code> or <code>DisplayAsync</code> that is declared on the interface <code>IPcWindowRegistration</code>.</td>
</tr>
<tr>
<td>State</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MatScreenOff</td>
<td>The mat screen is turned off and is not visible. The Sprout Platform might try to turn on the mat screen, depending on the user’s current interactions with the app. Turning on the mat screen will allow the user to see and interact with the window after the method <code>Display</code> or <code>DisplayAsync</code> that is declared on the interface <code>IPcWindowRegistration</code> is called. When the mat screen is turned on, the displayability state changes to <code>MatScreenOn</code>. Even though the mat screen is turned off, the platform places the window on the mat screen, which will allow the window to be displayed when the mat screen is turned on.</td>
</tr>
</tbody>
</table>

**User touches**

The platform uses events to convey information about user touches to a Sprout app. This touch handling bypasses the usual route for user touch events, which would flow from the touch mat to the operating system, and then to the app. In the case when a registered app is using the touch mat through the Sprout Platform, and the layer `Bypass` is enabled, the platform intercepts user touch data from touch-handling areas of the layer `Bypass` before the data goes to the operating system. The platform processes the data and sends touch events, which a registered Sprout app can subscribe to. In some cases, the platform sends data related to touch events to the operating system, for example when the user is using a keyboard (which is displayed in the layer `Input` when the layer is enabled).

The Sprout SDK provides the following functionality related to touch handling:

- The method `AccessTouch` or `AccessTouchAsync` is declared on the interface `IPcLink`. This method is used to create a touch controller (an instance of the interface `IPcTouch`) for a specific registered window.

- The method `GetTouchLayer` is declared on the interface `IPcTouch`. This method is synchronous in both C++ and WPF. The method returns an instance of the interface `IPcTouchLayer` for the requested touch layer.

**Inter-app and intra-app communication in C++ apps**

The platform provides functionality for inter-app and intra-app communication.

**Note:** A platform link is needed to create a communication handler, but communication is not local to the link.
The Sprout SDK provides the following functionality related to communication for Sprout apps that use the C++ binding:

- The method `CreateCommunicationHandler` is declared on the interface `IPcLink`. The method is used to create a specific communication handler. An identifier identifies the communication handler. Other parts of the same app or other apps send messages to a specific destination by referring to the identifier as the destination when sending messages.

- A communication handler can send messages to other communication handlers. Communication observers are needed to receive notifications about the events that contain the messages.

- The following virtual methods are declared on the interface `IPcCommunicationHandler`:
  - `Start`, which turns on the communication channel
  - `Stop`, which turns off the communication channel
  - `Send`, which sends a message

- The virtual method `Updated` is declared on the interface `IPcCommunicationObserver`. A communication observer must implement the method `Updated` to receive notifications about events.

- The notifications contain an instance of the interface `IPcMessageEventArgs`, which contains the message and related data.

- Virtual methods in the interface `IPcCommunicationHandler` allow adding communication observers to a list of communication observers, and removing them from the list.

To send a message, an app must:

1. Create a platform link.
2. Create a communication handler that will send events that carry the messages to named destinations (identifiers of communication handlers).
3. Turn on the communication channel.
4. Construct the message.
5. Send the message.

To receive a message, an app must:

1. Create a platform link.
2. Create a communication handler that will be a named destination for receiving events that carry the messages.
3. Create a communication observer, and add the communication observer to the list of communication observers maintained by the communication handler.
4. Turn on the communication channel.
5. Use the virtual method `Updated` to receive the events.
6. Parse the data streams that the events contain to extract the messages.

**Note:** To parse the message, the receiving app must know the structure of data stream. The only information about the data stream that is contained in the instance of the class `IPcMessageEventArgs` is the length of the data stream in bytes.

### Inter-app and intra-app communication in WPF apps

The platform provides functionality for inter-app and intra-app communication.

**Note:** A platform link is needed to create a communication handler, but communication is not local to the link.

The Sprout SDK provides the following functionality related to communication:

- The method `CreateCommunicationHandler` is declared on the interface `IPcLink`. The method is used to create a specific communication handler. An identifier identifies the communication handler. Other parts of the same app or other apps send messages to a specific destination by referring to the identifier as the destination when sending messages.
- A communication handler can send messages to other communication handlers (if the other communication handlers subscribe to the `MessageReceived` event for the source communication handler).
- The following methods are declared on the interface `IPcCommunicationHandler`:
  - `StartAsync`, which turns on the communication channel
  - `StopAsync`, which turns off the communication channel
  - `SendAsync`, which sends a message
- The event `MessageReceived` is declared on the interface `IPcCommunicationHandler`. The event communicates a message from a sender to all subscribing receivers.
- The event arguments for the event `MessageReceived` are an instance of the class `PcMessageEventArgs`, which contains a constructor for the message.

To send a message, an app must:

1. Create a platform link.
2. Create a communication handler that will send events that carry the messages to named destinations (communication handlers that subscribe to the `MessageReceived` events that carry the messages).
3. Turn on the communication channel.
4. Construct the message.
5. Send the message.

To receive a message, an app must:
1. Create a platform link.
2. Create a communication handler that will be a named destination for receiving events that carry the messages.
3. Subscribe to `MessageReceived` events sent by one or more specific communication handlers.
4. Turn on the communication channel.
5. Receive the events.
6. Parse the data streams that the events contain to extract the messages.

**Note:** To parse the message, the receiving app must know the structure of data stream. The only information about the data stream that is contained in the instance of the class `PcMessageEventArgs` is the length of the data stream in bytes.
Specifications

Specifications provide information about the Sprout immersive computer and about Sprout Platform options. The interface `IPcSpecification` is the starting point.

**Specification interfaces**

You use *specifications* to get information about the Sprout immersive computer, as well as to get information about Sprout Platform options.

The reason for using APIs to obtain information about the Sprout immersive computer is that things change. Today, the pixel density of the monitor screen is 3.78 pixels per millimeter. Tomorrow, it might be something else. Today, we have four extract options for pictures and outlines. Tomorrow, we might have six.

The starting point for accessing specifications is the interface `IPcSpecification`. You call the method `AccessSpecification | AccessSpecificationAsync`, which is available through an instance of the interface `IPcLink`, to get an instance of the interface `IPcSpecification`, for example:

```
C++
PcResult<IPcSpecification> spec = link->AccessSpecification();

WPF
IPcSpecification spec = await link.AccessSpecificationAsync();
```

The interface `IPcSpecification` contains properties with get accessors (WPF) or virtual methods (C++) for getting eight other specification interfaces. Using these subordinate interfaces through the instance of the interface `IPcSpecification`, you can obtain the information and use the options mentioned here:

- **Version**: Version of the Sprout Platform and of the binding used by the app
- **Camera**: Cameras available on the system
Overview of specification interfaces (C++)

Following is a diagram for the C++ binding of the interfaces that are available through virtual functions that are declared on the interface `IPcSpecification`. Numbers in the diagram correspond to explanations that follow the diagram.

In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`. 

```cpp
PcResult<IPcLink> link = HPPC::CreateLink();
```

2. 

```cpp
PcResult<IPcSpecification> spec = link->AccessSpecification();
```

3. 

```cpp
PcResult<IPcVersionSpecification> version = spec->GetVersion();
```

In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`. 

```cpp
PcResult<IPcLink> link = HPPC::CreateLink();
```

2. 

```cpp
PcResult<IPcSpecification> spec = link->AccessSpecification();
```

3. 

```cpp
PcResult<IPcVersionSpecification> version = spec->GetVersion();
```
2. The method `AccessSpecification` is available through the instance of the platform link. It returns an instance of the interface `IPcSpecification`. In this code example, the instance of the interface `IPcSpecification` is placed in the variable `spec`.

3. Beginning with the interface `IPcSpecification`, virtual methods are declared on the interfaces. For example, the virtual method `Picture` returns an instance of the interface `IPcPictureSpecification`. Follow the chain of virtual methods until you get what you need. For a complete example, see “Using options for extracting and refining pictures.”

**Overview of specification interfaces (WPF binding)**

Following is a diagram for the WPF binding of the interfaces that are available through properties that are declared on the interface `IPcSpecification`. Numbers in the diagram correspond to explanations that follow the diagram.

In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`. 
2. The method `AccessSpecificationAsync` is available through the instance of the platform link. It returns an instance of the interface `IPcSpecification`. In this code example, the instance of the interface `IPcSpecification` is placed in the variable `spec`.

3. Beginning with the interface `IPcSpecification`, interface properties with get accessors are declared on the interfaces. For example, the property `Picture` gets an instance of the interface `IPcPictureSpecification`. Follow the chain of interface properties until you get what you need. For a complete example, see “Using options for extracting and refining pictures.”

**Using options for extracting and refining pictures (C++)**

We work through in detail one example of using a specification: how to specify the extract option `Mat` when extracting pictures from a moment.

The short answer is “like this” (the part in bold):

```cpp
PcResult<IPcPicture> fastPicture =
    link->ExtractPicture(moment,
    spec->Picture()->Extract()->Mat());
```

Here is the longer answer. Following is a diagram of how four interfaces supply the extract option `Mat` for extracting pictures from a moment. Numbers in the diagram correspond to explanations that follow the diagram.
In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`.

2. The method `AccessSpecification` is available through the instance of the platform link. It returns an instance of the interface `IPcSpecification`. In this code example, the instance of the interface `IPcSpecification` is placed in the variable `spec`.


PcResult<IPcLink> link = MPPC::CreateLink();

PcResult<IPcSpecification> spec = link->AccessSpecification();

virtual specification::IPcPictureSpecification* Picture() = 0;

virtual IPCPictureExtractSpecification* Extract() = 0;

virtual IPCPictureExtractOption* BasicProcessing() = 0;

virtual IPCPictureExtractOption* Object() = 0;

virtual IPCPictureExtractOption* Mat() = 0;

virtual IPCPictureExtractOption* Document() = 0;

PcResult<IPcPictureExtractOption> Find(PcGuid identifier) = 0;

PcResult<IPcPicture> fastPicture =

link->ExtractPicture(moment, spec->Picture()->Extract()->Mat());
5. The virtual method `Mat` is declared on the interface `IPcPictureExtractSpecification`. The method `Mat` returns an instance of the interface `IPcPictureExtractOption`. In addition to providing the extract options for pictures, the interface `IPcPictureExtractSpecification` provides two functions. `Find` can be used to find a specific extract option for pictures. `Enumerate` can be used to obtain an enumeration of all extract options for pictures.

6. The interface `IPcPictureExtractOption` implements the interface `IPcIntent`. This can be used to obtain a specific implementation of functionality.

7. In the code to extract a picture, the method `ExtractPicture` takes as input parameters the moment and the extract option for pictures:

   ```csharp
   PcResult<IPcPicture> fastPicture = link->ExtractPicture(moment, spec->Picture()->Extract()->Mat());
   ```

   **Using options for extracting and refining pictures (WPF)**

   We work through in detail one example of using a specification: how to specify the extract option `Mat` when extracting pictures from a moment.

   The short answer is “like this” (the part in bold):

   ```csharp
   var fastPicture = await link.ExtractPictureAsync(moment, spec.Picture.Extract.Mat);
   ```

   Here is the longer answer. Following is a diagram of how four interfaces supply the extract option `Mat` for extracting pictures from a moment. Numbers in the diagram correspond to explanations that follow the diagram.
Sprout Developer Guide

In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`.

2. The method `AccessSpecificationAsync` is available through the instance of the platform link. It returns an instance of the interface `IPcSpecification`. In this code example, the instance of the interface `IPcSpecification` is placed in the variable `spec`.

3. The interface property `Picture` is declared on the interface `IPcSpecification`. The property `Picture` has a get accessor that returns an instance of the interface `IPcPictureSpecification`.

4. The interface property `Extract` is declared on the interface `IPcPictureSpecification`. The property `Extract` has a get accessor that returns an instance of the interface `IPcPictureExtractSpecification`.

```csharp
using (var link = IPPC.CreateLink())
{
    var link = IPpcLink();

    IPCSpecification spec = await link.AccessSpecificationAsync();
    spec

    IPCPictureSpecification Picture = spec; 

    IPCPictureExtractSpecification Extract = spec;

    IPCPictureExtractSpecification Option = Extract;

    var extractOption = spec.Picture.Extract.Get();
    var fastPicture = await link.ExtractPictureAsync(moment, extractOption);
}
```

In the diagram above:

1. Creating a platform link is the starting point for accessing Sprout Platform features. The variable `link` is an instance of the interface `IPcLink`.

2. The method `AccessSpecificationAsync` is available through the instance of the platform link. It returns an instance of the interface `IPcSpecification`. In this code example, the instance of the interface `IPcSpecification` is placed in the variable `spec`.

3. The interface property `Picture` is declared on the interface `IPcSpecification`. The property `Picture` has a get accessor that returns an instance of the interface `IPcPictureSpecification`.

4. The interface property `Extract` is declared on the interface `IPcPictureSpecification`. The property `Extract` has a get accessor that returns an instance of the interface `IPcPictureExtractSpecification`. 
5. The interface property `Mat` is declared on the interface `IPcPictureExtractSpecification`. The property `Mat` has a get accessor that returns an instance of the interface `IPcPictureExtractOption`. In addition to providing the extract options for pictures, the interface `IPcPictureExtractSpecification` provides two functions. `FindAsync` can be used to find a specific extract option for pictures. `EnumerateAsync` can be used to obtain an enumeration of all extract options for pictures.

6. The interface `IPcPictureExtractOption` implements the interface `IPcIntent`. This can be used to obtain a specific implementation of functionality.

7. In the code to extract a picture, the method `ExtractPictureAsync` takes as input parameters the moment and the extract option for pictures:

   ```csharp
   var fastPicture = await link.ExtractPictureAsync(moment, spec.Picture.Extract.Mat);
   ```

**Using options for extracting and refining outlines**

Options for extracting and refining outlines work exactly like options for extracting and refining pictures.

**Software versions (WPF)**

You can use the interfaces `IPcSpecification` and `IPcVersionSpecification` to get the version of the Sprout Platform and of the binding used by the app. For software versions:

- It is possible to get interface properties that contain the versions of the Sprout Platform and language binding.
- The interface `IPcVersionSpecification` does contain the methods `FindAsync` and `EnumerateAsync`.
- The interface `IPcVersionSpecification` does implement the interface `IPcIntent`.

To obtain information about the software versions of the Sprout Platform and binding, use the interface `IPcVersionSpecification` through the interface `IPcSpecification`. Proceed as follows:

1. Create an instance of the interface `IPcLink`. For example, place the instance in the variable `link`:

   ```csharp
   using (var link = HPPC.CreateLink())
   ```

2. Through the instance of the platform link, use the method `AccessSpecificationAsync` to return an instance of the interface `IPcSpecification`:

   ```csharp
   IPcSpecification spec = await link.AccessSpecificationAsync();
   ```
3. The interface property `Version` is declared on the interface `IPcSpecification`. The property `Version` has a get accessor that returns an instance of the interface `IPcVersionSpecification`. Following is the declaration of the property `Version`:

```csharp
IPcVersionSpecification Version { get; }
```

4. The interface properties `SDK` and `Binding` are declared on the interface `IPcVersionSpecification`. The interface property `SDK` is the version of the Sprout Platform. The properties have get accessors that return the respective versions. Following are the declarations of the properties `SDK` and `Binding`:

```csharp
Version SDK { get; }
Version Binding { get; }
```

5. To obtain the version of the Sprout Platform, the code can be written this way. Interface properties are accessed using dot notation:

```csharp
var platformVersion = spec.Version.SDK;
```

### Cameras (WPF)

**Note:** In this release, `IPcCameraSpecification` and `IPcCamera` cannot be used to obtain information about the cameras.

You can use the interfaces `IPcSpecification` and `IPcCameraSpecification` to get information about the cameras on a system. For cameras:

- It is possible to get interface properties that specify the kinds of cameras on a system.
- The interface `IPcCameraSpecification` does contain the methods `FindAsync` and `EnumerateAsync`.
- The interface `IPcCamera` does implement the interface `IPcIntent`.

To obtain information about the cameras on a system, use the interface `IPCCameraSpecification` through the interface `IPcSpecification`. Proceed as follows:

1. Create an instance of the interface `IPcLink`. For example, place the instance in the variable `link`:

   ```csharp
   using (var link = HPPC.CreateLink())
   ```

2. Through the instance of the platform link, use the method `AccessSpecificationAsync` to return an instance of the interface `IPcSpecification`:

   ```csharp
   IPCSpecification spec = await link.AccessSpecificationAsync();
   ```

3. The interface property `Camera` is declared on the interface `IPcSpecification`. The property `Camera` has a get accessor that returns an instance of the interface `IPCCameraSpecification`. Following is the declaration of the property `Camera`:

   ```csharp
   IPCCameraSpecification Camera { get; }
   ```
4. The interface properties that represent types of cameras (HighDefinition, LowDefinition, Infrared, and Depth) are declared on the interface IPCCameraSpecification. The properties have get accessors that return the respective camera types. Following are the declarations of the properties:

```
IPcCamera HighDefinition { get; }
IPcCamera LowDefinition { get; }
IPcCamera Infrared { get; }
IPcCamera Depth { get; }
```

5. To enumerate all of the cameras on the system and add the camera names to a list box, the code can be written this way (this code is from the sample app WpfCsSample). Interface properties are accessed using dot notation:

```
IPcSpecification spec = await pcLink.AccessSpecificationAsync();
var cameras = await spec.Camera.EnumerateAsync();
```

**Screens (WPF)**

You can use the interfaces IPcSpecification and IPcScreenSpecification to get information about the screens on a system. For screens:

- It is possible to get interface properties that specify the kinds of screens on a system.
- The interface IPcScreenSpecification does contain the methods FindAsync and EnumerateAsync.
- The interface IPcScreen does implement the interface IPcIntent.

To obtain information about the screens on a system, use the interface IPcScreenSpecification through the interface IPcSpecification. Proceed as follows:

1. Create an instance of the interface IPcLink. For example, place the instance in the variable link:

   ```
   using (var link = HPPC.CreateLink())
   ```

2. Through the instance of the platform link, use the method AccessSpecificationAsync to return an instance of the interface IPcSpecification:

   ```
   IPcSpecification spec = await link.AccessSpecificationAsync();
   ```

3. The interface property Screen is declared on the interface IPcSpecification. The property Screen has a get accessor that returns an instance of the interface IPcScreenSpecification. Following is the declaration of the property Camera:

```
IPcScreenSpecification Screen { get; }
```
4. The interface properties that represent types of screens (**Mat** and **Monitor**) are declared on the interface **IPcScreenSpecification**. The properties have get accessors that return the respective screen types. Following are the declarations of the properties **Mat** and **Monitor**:

```csharp
IPcScreen Mat { get; }
IPcScreen Monitor { get; }
```

5. To enumerate all of the screens on the system and display information about the screens, the code can be written this way (this code is from the sample app **WpfCsSample**). Interface properties are accessed using dot notation:

```csharp
foreach (var src in spec.Screen.EnumerateAsync().Result)
{
    var screenTypeInfo = new ScreenTypeInfo(
        src.Identifier,
        src.PixelDensity,
        src.PixelExtent,
        src.WorkArea.Location);
    AvailableDisplays.Add(screenTypeInfo);
}
```

**Text (WPF)**

You can use the interfaces **IPcSpecification** and **IPcTextSpecification**, as well as additional subordinate interfaces, to get options for extracting and refining text. For the texts:

- It is possible to get interface properties that specify the options for extracting and refining texts. There are also options for languages.
- The interfaces **IPcTextExtractSpecification**, **IPcTextRefineSpecification**, and **IPcTextLanguageSpecification** do contain the methods **FindAsync** and **EnumerateAsync**.
- The interfaces **IPcTextExtractOption**, **IPcTextRefineOption**, and **IPcTextLanguageOption** do implement the interface **IPcIntent**.

To use the extract option **BasicProcessing** when extracting text from a moment, use the interface **IPcTextSpecification** through the interface **IPcSpecification**. Proceed as follows:

1. Create an instance of the interface **IPcLink**. For example, place the instance in the variable **link**:

```csharp
using (var link = HPPC.CreateLink())
```
2. Through the instance of the platform link, use the method \texttt{AccessSpecificationAsync} to return an instance of the interface \texttt{IPcSpecification}:

\[
\text{IPcSpecification } \text{spec} = \text{await link.AccessSpecificationAsync();}
\]

3. The interface property \texttt{Text} is declared on the interface \texttt{IPcSpecification}. The property \texttt{Text} has a get accessor that returns an instance of the interface \texttt{IPcTextSpecification}. Following is the declaration of the property \texttt{Text}:

\[
\text{IPcTextSpecification } \text{Text} \{ \text{get; } \}
\]

4. The interface properties \texttt{Extract}, \texttt{Refine}, and \texttt{Language} are declared on the interface \texttt{IPcTextSpecification}. These properties have get accessors that returns instances of the interfaces \texttt{IPcTextSpecification}, \texttt{IPcTextRefineSpecification}, and \texttt{IPcTextLanguageSpecification} respectively. Following are the declarations of the properties \texttt{Extract}, \texttt{Refine}, and \texttt{Language}:

\[
\text{IPcTextExtractSpecification } \text{Extract} \{ \text{get; } \}
\]
\[
\text{IPcTextRefineSpecification } \text{Refine} \{ \text{get; } \}
\]
\[
\text{IPcTextLanguageSpecification } \text{Language} \{ \text{get; } \}
\]

5. The interface properties that represent options for extracting and refining text, and for language, are declared on the interfaces \texttt{IPcTextSpecification}, \texttt{IPcTextRefineSpecification}, and \texttt{IPcTextLanguageSpecification}. The properties have get accessors that return the options. For example, following are the declarations of the properties \texttt{Default} and \texttt{BasicProcessing}:

\[
\text{IPcTextExtractOption } \text{Default} \{ \text{get; } \}
\]
\[
\text{IPcTextExtractOption } \text{BasicProcessing} \{ \text{get; } \}
\]

6. In the code to extract text, the method \texttt{ExtractTextAsync} takes as input parameters the moment and the extract option for text:

\[
\text{var basicText} = \text{await link.ExtractTextAsync(moment, spec.Text.Extract.BasicProcessing);}
\]

**Classification tags (WPF)**

You can use the interfaces \texttt{IPcSpecification} and \texttt{IPcClassificationSpecification} to get tags (instances of the interface \texttt{IPcTag}) that classify the contents of a moment.

The method \texttt{ClassifyAsync} in the interface \texttt{IPcLink} places the tags that apply to a moment in a tree structure in the moment and returns the tags in an instance of the interface \texttt{IPcClassification}.

For classification tags:

- It is possible to get interface properties that specify the classification tags.
The interface `IPcClassificationSpecification` does contain the methods `FindAsync` and `EnumerateAsync`.

The interface `IPcTag` does implement the interface `IPcIntent`.

To obtain information about the tags that are used when classifying the contents of a captured moment, use the interface `IPcClassificationSpecification` through the interface `IPcSpecification`. Proceed as follows:

1. Create an instance of the interface `IPcLink`. For example, place the instance in the variable `link`:
   ```csharp
   using (var link = HPPC.CreateLink())
   ```
2. Through the instance of the platform link, use the method `AccessSpecificationAsync` to return an instance of the interface `IPcSpecification`:
   ```csharp
   IPcSpecification spec = await link.AccessSpecificationAsync();
   ```
3. The interface property `Classification` is declared on the interface `IPcSpecification`. The property `Classification` has a get accessor that returns an instance of the interface `IPcClassificationSpecification`. Following is the declaration of the property `Classification`:
   ```csharp
   IPcClassificationSpecification Classification { get; }
   ```
4. The interface properties that represent classification tags (`Flat` and `Rectangular`) are declared on the interface `IPcClassificationSpecification`. The properties have get accessors that return the respective classification tags. Following are the declarations of the properties `Flat` and `Rectangular`:
   ```csharp
   IPcTag Flat { get; }
   IPcTag Rectangular { get; }
   ```
5. To enumerate all of the classification tags available in the Sprout Platform and to add the classification tags to a list box, the code can be written this way. Interface properties are accessed using dot notation:
   ```csharp
   foreach (var src in spec.Classification.EnumerateAsync().Result)
   {
   }
   ```

**Touch-layer identifiers (C++ and WPF)**

You can use the interfaces `IPcSpecification` and `IPcTouchSpecification` to get identifiers of touch layers (instances of the interface `IPcTouchLayerOption`) that are available in the Sprout Platform.
The method `AccessTouch | AccessTouchAsync` in the interface `IPcLink` creates a touch controller, an instance of the interface `IPcTouch`, for a registered window. In the interface `IPcTouch`, the method `GetTouchLayer` uses this identifier to retrieve a specific touch layer.

**C++**

In the C++ binding, this is the declaration of the method `GetTouchLayer`:

```cpp
virtual PcResult<IPcTouchLayer> GetTouchLayer(specification::IPcTouchLayerOption* touchLayerOption) = 0;
```

And this is an example of getting the touch layer Bypass (from the file `touchbypass.cpp` in the sample app `CppQtSample`):

```cpp
PcResult<IPcTouchLayer> layerHandler = touch->GetTouchLayer(spec->Touch()->Bypass());
```

**WPF**

In the WPF binding, this is the declaration of the method `GetTouchLayer`:

```csharp
IPcTouchLayer GetTouchLayer(IPcTouchLayerOption, touchLayerOption)
```

And this is an example of getting the touch layer `Bypass` (from the file `TouchAreaSampleControl.xaml.Display.cs` in the sample app `WpfCsSample`); some code is omitted (indicated by `...`):

```csharp
_bypassLayer = await PrepareTouchLayer(_touchController, specification.Touch.Bypass, touchArea);
...
private async Task<IPcTouchLayer> PrepareTouchLayer(IPcTouch touch, IPcTouchLayerOption touchLayerOption, UIElement control)
...

var layer = touch.GetTouchLayer(touchLayerOption);
```

For touch-layer types:

- It is possible to get interface properties that specify the identifiers of the touch layers.
- The interface `IPcTouchSpecification` does contain the methods `FindAsync` and `EnumerateAsync`. 
The interface `IPcTouchLayerOption` does implement the interface `IPcIntent`. To identify a touch layer to get, use the interface `IPcTouchSpecification` through the interface `IPcSpecification`. Proceed as follows:

1. Create an instance of the interface `IPcLink`. For example, place the instance in the variable `link`:
   ```csharp
   using (var link = HPPC.CreateLink())
   ```
2. Through the instance of the platform link, use the method `AccessSpecificationAsync` to return an instance of the interface `IPcSpecification`:
   ```csharp
   IPcSpecification spec = await link.AccessSpecificationAsync();
   ```
3. The interface property `Touch` is declared on the interface `IPcSpecification`. The property `Touch` has a get accessor that returns an instance of the interface `IPcTouchSpecification`. Following is the declaration of the property `Touch`:
   ```csharp
   IPcTouchSpecification Touch { get; }
   ```
4. The interface properties that represent touch-layer identifiers (`Bypass`, and `Input`) are declared on the interface `IPcTouchSpecification`. The properties have get accessors that return the respective touch-layer identifiers. Following are the declarations of the properties:
   ```csharp
   IPcTouchLayerOption Bypass { get; }
   IPcTouchLayerOption Input { get; }
   ```
5. To retrieve the `Rejection` touch layer, the code can be written this way. Interface properties are accessed using dot notation:
   ```csharp
   var layer = touch.GetTouchLayer(spec.Touch.Rejection);
   ```
Coordinates and conversions

Sprout senses user touches on the touch mat and displays pictures on the mat screen and the monitor. To make this possible, the Sprout Platform works with both physical and pixel coordinates. The Sprout SDK provides the class `ConversionUtilities`, which contains functions for converting between the coordinate systems.

**Sprout coordinate systems**

Sprout uses two (X, Y) coordinate systems. Coordinates are used when referencing the positions of points and other geometric elements in space, and by reference the positions of real objects (physical coordinates, measured in millimeters) or of pixels (screen coordinates, measured in pixels).

The Sprout Platform uses both physical and pixel coordinate systems:

- **The touch mat**: The touch mat has a physical coordinate system with the origin (0, 0) in the physical upper-left corner of the work area of the touch mat (the white area). The positive X axis runs across the upper edge of the work area. The positive Y axis runs down the left edge of the work area. The touch point and touch size in the arguments for a touch event use physical coordinates. The work area of the touch mat measures 12 inches high by 16 inches wide, thus the height and width are 304.8 mm and 406.4 mm respectively.

- **The monitor screen and the mat screen**: The monitor screen and the mat screen have pixel coordinate systems. Each pixel coordinate system has (0, 0) in the center of the pixel in the upper-left corner of the mat screen. The positive X axis access runs across the centers of the upper row of pixels on the screen. The positive Y axis runs down the centers of the leftmost row of pixels on the screen. For both screens, the available area might be the entire screen, but it might not be. And there is the complication that Windows considers the monitor screen to be extended onto the mat screen. These complications are explained below.

- **Images**: Images consist of pixels, so they can be displayed using a pixel coordinate system. Images also often have desired physical sizes, for example, an image of a 4-by-6 inch photograph could be displayed so that it appears to be 4 by 6 inches. Images extracted from moments contain information about the pixel density. Images from other sources such as the Web or cameras might or might not contain information about the pixel density, and the pixel density information might not be appropriate for the Sprout screens.
**Physical coordinates**

The following diagram illustrates the physical coordinate system of the touch mat.

![Physical coordinate system](image)

**Pixel coordinates**

Both the monitor screen and the mat screen have pixel coordinate systems. The positive X axis access runs across the centers of the upper row of pixels on the screen. The positive Y axis runs down the centers of the leftmost row of pixels on the screen.

**The work area and the offset of the screens**

The *work area* and the *offset* affect where you can display things such as pictures and outlines. Both apply to the monitor screen and the mat screen.

There are *two sets* of work areas and offsets, one for Windows and one for Sprout. Windows work areas and offsets reflect the current location or locations of the Windows task bar. Sprout work areas and offsets reflect use of full-screen windows that cover task bars if they are present.

**Note:** Use the Sprout work areas and offsets, which can be obtained from an instance of the interface *IPcScreen*, which you obtain from the specification *IPcScreenSpecification*.

Following are descriptions of the Sprout work areas and Sprout screen offsets:

- **Sprout work areas:** The *work area* of a screen is the area that an app can use to display things. For Sprout apps, this is currently the full screen for both the monitor screen and the mat screen. It is important to note that this might change, for example, if a Sprout task bar is added. Obtain information about the work areas from *IPcScreen*. Do not hard code values.
- **Sprout screen offsets**: The offset is the change in pixel coordinates from (0, 0) at the upper-left corner of a display (where the pixel would be without the offset) to the upper-left pixel of the screen. For Sprout apps, the offset for the monitor screen is currently (0, 0) and the offset for the mat screen is (448, 1080). The offset for the mat screen reflects the fact that Windows considers the mat display to be an extension of the monitor display. It is important to note that these offsets might change, for example, if a Sprout task bar is added. Obtain information about the offsets from `IPcScreen`. Do not hard code values.

This diagram shows examples of work areas and offsets, given one arrangement of task bars:
Whether an app needs to explicitly consider the work area and the offset depends on the circumstances:

- **Monitor screen**: The Sprout Platform does not manage app windows on the monitor screen. Use WPF to manage windows on the monitor screen and to ensure that an app respects the work area. That is, references to specific pixels must consider the offset and work area. Optionally, you can obtain information about the work area and offset of the monitor screen from the Sprout Platform.

- **Mat screen**: The Sprout Platform does manage app windows on the mat screen if you register the mat window with the Sprout Platform and use the method `Display` or `DisplayAsync` to display the window (which is the recommended approach). When using the Sprout Platform to display a window, the Sprout Platform considers the offset from (0, 0) of the monitor screen to (0, 0) of the mat screen automatically, so you can use the (0, 0) point of the mat screen as the upper-left corner of the screen.

**Note**: Windows for Sprout apps are currently designed to be presented full screen, thus not respecting any additional offset on the mat screen or the work area of the mat screen, and covering a task bar if one is present.

### Obtaining an instance of `IPcScreen` for the monitor screen or the mat screen

Use the specification interface `IPcScreenSpecification` to obtain an instance of the interface `IPcScreen` for a specific screen (Monitor or Mat). From the instance of `IPcScreen`, you can obtain information about the work area. That information includes the offset.

**Note**: If you address pixels on the mat screen, and you use information from `IPcScreen` as the starting point, then the correct offset and dimensions of the work area are taken into account.

#### C++

Using the virtual method `WorkArea`, you can obtain a rectangle (an instance of the class `PcPixelRectangle`) that describes the work area. The method `PcPixelRectangle::Location` returns an instance of the class `PcPixelPoint`, which declares methods for obtaining the X and Y coordinates of the top-left corner of the work area. The X and Y coordinates in the instance of `PcPixelPoint` include the offset if there is one. The method `PcPixelRectangle::Size` returns an instance of the class `PcPixelSize`, which declares methods for obtaining the width and height of the work area.
Using the get accessor for the property `WorkArea`, you can obtain a rectangle (`Rect`) that describes the work area. The `Rect` contains the X coordinate of the left side of the rectangle, the Y coordinate of the top of the rectangle, the width of the rectangle, and the height of the rectangle. The X and Y coordinates include the offset if there is one.

### Converting between physical and pixel measurements

The Sprout Platform operates with both physical dimensions, because physical objects and the touch mat are physical, and with pixel dimensions, because images are displayed on the monitor and on the mat screen. In some places, you provide or get physical dimensions. In other places, you provide or get pixel dimensions.

In the class `ConversionUtilities`, we provide functions for converting between physical and pixel measurements.

### Scaling things for display and method calls

Sprout captures moments. From the moments, an app can extract pictures and outlines. Pictures and outlines both have bounding rectangles. An outline is a contour. Outline refinement uses a bitmap created from a stroke collection. To display things correctly, and for outline refinement to work correctly, pictures, bounding rectangles, outline contours, and stroke collections must be scaled correctly.

Following are several key points about scaling things:

- Regarding the match between physical and pixel coordinates, for both the monitor and the touch mat, the upper-left corner of the upper-left-most pixel is placed on the upper-left-most physical point of the display.
- Some things use pixel coordinates and others use physical coordinates. For example, a picture consists of pixels, but the bounding box for a picture has a physical location and dimensions.
- For pixel density, the Sprout Platform works with pixels per millimeter.
- In some cases, WPF works with pixels per inch (or dots per inch, DPI). WPF also sometimes assumes that the pixel density of a screen is 96 DPI. Conversions are needed for these cases.
• Images from other sources than Sprout cameras might contain information about the pixel density. That information is often in pixels per inch, and it should be considered unreliable. When a moment is created from an image, the pixels-per-inch information is translated into pixels per millimeter. When pictures or outlines are extracted from the moment, the millimeters-per-inch information is available. That information is should also be considered unreliable. Consider the pixel dimensions of the image, the desired display size, and the pixels per millimeter of the target screen.

• Always obtain information about the pixel density of the screens from an instance of IPCScreen obtained through the interface IPCScreenSpecification.

• Sprout cameras do not have pixel densities, just pixel extents.
Exceptions (WPF)

Sometimes software is not able to do what is asked of it. The Sprout Platform throws an exception when it cannot fulfill a request.

Platform exceptions

The Sprout Platform can throw exceptions. Every Sprout SDK method does one or the other of these, never both:

- It succeeds and returns a result.
- It fails and throws an exception.

What an exception contains

The Sprout Platform class for exceptions is `PcException`, which extends and specializes the WPF base class `Exception`:

```csharp
public class PcException : Exception
```

Every method in the Sprout SDK is either asynchronous or synchronous. The exceptions that the platform throws differ for these two cases:

- **Asynchronous functions**: Every method that returns a task (with or without a `TResult`) can throw an aggregate exception. The aggregate exception encapsulates the `PcException`.
- **Synchronous functions**: Functions that are synchronous return a `PcException` without encapsulation.

Every `PcException` contains an `HRESULT` which is an integer that identifies the facility and error code. The facility and error code can be used to find out more about the exception.

Handling an exception

**Note**: In this release, only one part of the `WpfCsSample` sample app uses this approach for exceptions: `VersionSample.xaml.Display.cs`. Refer to that file.
The WpfCsSample sample app file `VersionSample.xaml.Display.cs` contains an example of the code for exception handling. The first part is for synchronous method calls. The second part is for asynchronous calls.
Windows on the touch mat

For a Sprout app to use the screen that is projected on the touch mat, the app must register with the platform any app window that will appear on the touch mat. A registered window is also required for the platform to manage user touches on the touch mat.

Note: The Sprout Platform does not manage windows on the monitor. Use Windows APIs for that.

About mat windows

Sprout’s mat screen is projected onto the touch mat. For any window that will use the touch mat and the mat screen, a Sprout app must use the Sprout SDK to register the window and to get a window handler (an instance of the interface `IPcWindowRegistration`). To create a window handler for a mat window, call the method `RegisterWindow | RegisterWindowAsync` that is declared on the interface `IPcLink`.

Note: The Sprout Platform tries to ensure that the optimal screen resolution is used on the touch mat (that is, 1024x768 pixels). Applications must use this resolution for the mat window.

The window handler does five things:

- It provides the method `Display | DisplayAsync` for displaying the window. This method takes care of moving the window from the monitor to the mat screen if necessary. The method does its best to ensure that the window is displayed on the mat screen, but there is no guarantee and the method does not return an exception if it was not able to display the window on the mat screen. Depending on the displayability state of the window, the Sprout Platform might try to turn on the mat screen (that is, the projector).

- It provides the ability to get the displayability of the window, and to receive events that convey information about a change in displayability.

- It enforces UI requirements for mat windows for Sprout apps. A mat window in a Sprout app must be displayed as a full-screen, uppermost, active window without borders.

- It makes it possible to use touch functionality for the touch mat that the Sprout Platform provides.

- It provides `Unregister | UnregisterAsync` functions for registering and unregistering the window.
You don't use the Sprout Platform to create the window. You use Windows APIs to create the window. In the method `RegisterWindow | RegisterWindowAsync`, you pass the window handle as a parameter to a new instance of the window-handler interface `IPcWindowRegistration`.

A platform link does not require a window. A small add-on designed to access Sprout Platform features (but not features related to the touch mat and mat screen) might not need a window. Many Sprout Platform features are provided through the interface `IPcLink`. For example, captured moments are related to a platform link. An app can capture a moment, extract a picture or pictures from the moment, and display the picture or pictures in a window on the monitor without having a Sprout Platform window handler for the window.

**Key points**

Note the following key points about managing windows on the mat screen:

- The Sprout SDK provides a window-handler interface, `IPcWindowRegistration`, which can be instantiated to provide window handlers for application windows that will use Sprout Platform features on the touch mat. Instantiate the interface `IPcWindowRegistration` by calling the method `RegisterWindow | RegisterWindowAsync` in an instance of the interface `IPcLink`.

- An application can have one or more platform links. Each platform link can have zero registered windows, one registered window, or more than one registered window.

- A platform link does not require a mat window. For example, a small add-on designed to access Sprout Platform features might not need a window. However, using the Sprout Platform to manage user touches on the touch mat does require a window that is registered with the Sprout Platform.

- An application with multiple windows in the same platform link has access to the platform link's moments across the windows. (Accessing moments does not involve the window handlers.)

- You don't use the Sprout SDK to create the window. You have to provide the window. The Sprout SDK provides a window-handler interface, to which you pass the window when you create an instance of the window handler to handle a specific window.

- A window that is registered with a window handler must be displayed full screen on the mat screen. This is a UI requirement for Sprout apps.

- You display a registered window on the mat screen by calling the method `Display | DisplayAsync` in an instance of the interface `IPcWindowRegistration`.

- You control touch functionality, that is, regions of touch sensitivity and displaying and hiding a virtual keyboard and/or virtual trackpad, by calling functions in an instance of the touch-controller interface `IPcTouch` and by using functions, properties, and events in related instances of the interface `IPcTouchLayer`.

- You control whatever else happens in the window yourself, without using the Sprout Platform.
• If your Sprout app has more than one mat window, calling the method `Display` | `DisplayAsync` to display a specific mat window hides the window that was previously displayed on the mat screen. Calling the method `Display` | `DisplayAsync` for the first window displays that window again and hides the second window.

• (WPF) To close a mat window that you plan to display again, you can use the WPF method `Close`.

• To close a mat window when it is no longer needed, unregister the window by calling the method `Unregister` | `UnregisterAsync` in the instance of the interface `IPcWindowRegistration` for the window.

  **Note:** (WPF) Before unregistering the window, unsubscribe from touch events used by the window and dispose of the interface `IPcTouch` and all of the `IPcTouchLayer` interfaces that were used by the window.

• The Sprout Platform manages the use of the touch mat and mat screen by different Sprout apps. For example, if a user switches from Sprout app A to Sprout app B, the mat window for app B replaces that for app A.

• If two or more Sprout apps use the touch mat and mat screen, then the settings for the apps might differ. For example, two Sprout apps might define different touch-handling areas. In this case, the settings that are active are for the active application session.

### Registering a window (WPF)

To register a window with the Sprout Platform, call the method `RegisterWindowAsync` in an instance of the interface `IPcLink`. This method returns an instance of the window-handler interface `IPcWindowRegistration`.

When registering a window with the Sprout Platform, the window must have been created with a valid window handle. To ensure that the window can be registered correctly, we recommend that do the following:

1. Create a WPF window.
2. Create the window handle.
3. Register the window with the Sprout Platform by calling the method `RegisterWindowAsync` in the interface `IPcLink`.
4. Call the method `DisplayAsync` to display the window.

An example of this approach, see the `WpfCsSample` sample app file `Present.xaml.Display.cs`.  

---

`sprout` by `hp`
Displaying a window on the mat screen (WPF)

To display a window on the mat screen, call the method `DisplayAsync` in an instance of the interface `IPcLink`, for example:

```csharp
await window.DisplayAsync();
```

Using touch functionality in a window (WPF)

To use touch functionality that is provided by the Sprout Platform in the window, you must create an instance of the touch controller `IPcTouch` by calling the method `AccessTouchAsync` in an instance of the interface `IPcLink`. When you call the method `AccessTouchAsync`, you reference the window handler for the registered window. This is the method declaration for `AccessTouchAsync`:

```csharp
Task<IPcTouch> AccessTouchAsync(IPcWindowRegistration window);
```

This is an example of the method call:

```csharp
_touchController = await _sdk.AccessTouchAsync(pcwindow);
```

Unregistering a window (WPF)

To unregister a window, call the method `UnregisterWindow` in the instance of the interface `IPcWindowRegistration` for the window.

Following is an example of a command for unregistering a window:

```csharp
await window.UnregisterAsync();
```

Sample app files (WPF)

Regarding the sample application:

For examples of window management, see the files `Present.xaml.Display.cs`, `Keyboard.xaml.Display.cs`, and `TrackPadSampleControl.xaml.Display.cs`.

Steps (WPF)

If an app will use Sprout Platform features on the touch mat and mat screen, perform these steps:

1. Use `HPPC.CreateLink` to create a platform link.
2. Use Windows APIs to create a window for the app, and a handle for the window. When you create an instance of the Sprout Platform window handler, you pass the window to the handler.
3. Register the window with the Sprout Platform by calling the method `RegisterWindowAsync` in the interface `IPcLink`. This returns the window handler for the window, an instance of the interface `IPcWindowRegistration`.

4. Display the window by calling the method `DisplayAsync` in the instance of the interface `IPcWindowRegistration` for the window.

5. Call the method `AccessTouchAsync` to create a touch controller for the window. Identify the window with the parameter `window`. Also call other touch-related methods to set the geometry of touch-handling areas and to enable touch layers as needed.

**Multiple windows from one app on the mat screen**

To display multiple app windows on the mat screen, an app can instantiate multiple window handlers. A window that is managed by the Sprout Platform can only be presented on the mat screen topmost on the full screen. So, if an app uses multiple windows on the mat screen (in sequence), application logic should ensure that the correct window is topmost.

**Windows from different Sprout apps on the mat screen**

If two or more Sprout apps use the touch mat and mat screen, then the settings for the apps might differ. For example, two Sprout apps might define different touch-handling areas on the touch mat. In this case, the settings that are active are for the active application session.
Touch

Sprout has a touch mat. Regarding user touches and touch events, you have a choice: you can leave management of touch events to the operating system, you can have the Sprout Platform manage touch events, or you can do a combination of both.

About the touch mat and mat screen

Following are several important points about the touch mat and mat screen:

- The Sprout Platform must be running for the touch mat and the mat screen to work.
- The entire touch mat is not touch-sensitive. The white area in the middle of the mat is touch-sensitive. The mat screen is projected onto that area.
- The process of aligning the mat screen with the work area is called calibration. Calibration can be performed from the Sprout Settings Panel or the Sprout Workspace. Calibration might give completely accurate alignment, but some small misalignment can remain after calibration.
- To use Sprout as an immersive computer, the touch mat must be attached. But users can remove the touch mat. Removing the touch mat removes touch input to your app from the touch mat (obviously). It also stops projection of the mat screen. Your app should use displayability and error handling to manage removal of the touch mat.

User control of mat screen projection and the touch mat

Before presenting touch management, it is worth remembering that a user has some control over both screen projection and touch.

A user can use the Projector/Touch Mat button (the rightmost button) on the Sprout Connector to control the projector and the touch mat. The user can:

- Tap the Projector/Touch Mat button to turn off the touch mat.
- Touch-and-hold the Projector/Touch Mat button to turn off the projector.
- Tap the Projector/Touch Mat button to turn both the projector and the touch mat back on.

Because the Sprout Platform performs these actions, the platform can inform a Sprout app about the current state of the projector-plus-touch mat system. This is referred to as displayability.
Approaches for managing touch events for user touches on the touch mat

There are three approaches for managing touch events, which are events that are generated when a user touches the touch mat:

- **Use the operating system to manage touch events**: In this case, a Sprout app does not use the Sprout Platform to create a touch controller and to manage touch events. All touch events go from the touch mat to the operating system. The Sprout app would use operating system APIs to get information about touch events.

  **Note**: In this case, user touches on the touch mat will move window focus to the touch mat. This is referred to as *direct touch*.

- **Use the Sprout platform to manage touch events**: A Sprout app can use the Sprout Platform to manage touch events during the entire lifetime of the mat window of the app. The Sprout app would use Sprout APIs to get information about touch events.

  **Note**: In this case, user touches on the touch mat will not move window focus to the touch mat if they occur in touch-handling areas of the enabled touch layer Input or Bypass. This is referred to as *indirect touch*.

The reason for using indirect touch can be illustrated with a game controller. A user touches controls on the touch mat, but window focus remains on the monitor.

If any areas of the mat window do not have enabled touch-handling areas, then user touches on those areas are sent to the operating system and are direct touches, which move window focus from the monitor to the touch mat.

Touch events contain information about the coordinates of the touch point in millimeters and the width and height of the bounding rectangle that bounds the touch. For more information, see the *API Reference*.

- **Use both approaches at different times**: It is possible for a sprout app to use both approaches for managing touch events.

The following table summarizes the event paths and window focus for the different approaches for managing user touches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Event path</th>
<th>Window focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the operating system (no Sprout Platform touch controller)</td>
<td>Touch events go from the touch mat to the operating system. An app can obtain touch events from the operating system.</td>
<td>The operating system controls window focus, which moves to the mat window when the user touches the touch mat.</td>
</tr>
<tr>
<td>Approach</td>
<td>Event path</td>
<td>Window focus</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Touch layer <strong>Bypass</strong></td>
<td>Touch events go from the touch mat to the Sprout Platform. An app can obtain touch events from the Sprout Platform.</td>
<td>The Sprout Platform controls window focus, which remains on the monitor window when the user touches the touch mat.</td>
</tr>
<tr>
<td>Touch layer <strong>Input</strong></td>
<td>Touch events go from the touch mat to the operating system, where they are interpreted as key strokes. An app can obtain keyboard events from touches the touch mat.</td>
<td>The operating system controls window focus, which remains on the monitor window when the user touches the touch mat.</td>
</tr>
<tr>
<td>Using the Sprout Platform touch controller, with touches occurring on areas that are outside of the areas of enabled touch layers.</td>
<td>Touch events go from the touch mat to the operating system. An app can obtain touch events from the operating system.</td>
<td>The operating system controls window focus, which moves to the mat window when the user touches the touch mat.</td>
</tr>
</tbody>
</table>

**Overview of touch management using the operating system**

In this case, a Sprout app registers a window that it wants to display on the touch mat. But the Sprout app does not use the Sprout Platform to create a touch controller. All touch events go from the touch mat to the operating system. The Sprout app would use operating system APIs to get information about touch events.

The diagram that follows shows use of the operating system for touch management.
Overview of touch management using the platform

A Sprout app can use the Sprout Platform to manage touch events during all or some of the lifetime of the mat window of the app. When the platform is not managing touch events, the operating system is. In some cases, both the platform and the operating system manage touch events.

Following are key points about touch management by the Sprout Platform:

- As with most things, the first step is to obtain an instance of the platform link `IPcLink`. The second step is to register a window. An instance of the touch controller `IPcTouch` must know which window the touch events are intended for.

- For your app to use the Sprout Platform’s touch management, it must create a single instance of the touch controller `IPcTouch`. From the touch controller, the app must get an instance of the touch-layer controller `IPcTouchLayer` for each touch layer that it plans to use. The touch-layer controller is the most important interface for touch management.
• User touches are recognized by the touch mat, but the touch mat does not immediately send touch events to the operating system. Instead, the Sprout Platform evaluates the user touches with respect to touch layers that can be thought of as floating above the touch mat. Touch layers are illustrated in this diagram:

A user touches the work area of the touch mat. The touch propagates through the touch layers of a Sprout Platform touch controller.

• What a touch layer does depends on four things:
  o **Whether your app has an instance of the layer controller for the layer.** If your app doesn’t, then the touch layer does not exist.
  o **Which layer it is.** The two layers have different purposes, as indicated in the diagram above.
  o **Whether the layer is enabled.** User touches are evaluated for an enabled layer with a defined touch-handling area. User touches are not evaluated for a disabled layer, or for an enabled layer without a defined touch-handling area.
  o **The geometry of the touch-handling area of the layer.** User touches inside the geometry defined for the layer are handled by the layer. User touches outside the geometry pass through to the next layer, or to the touch mat. Based on the geometry of the touch-handling area, the platform generates a *layer mask* for the touch layer. The layer mask has the number 1 for each pixel where the layer will handle a user touch, and the number 0 for each pixel where the layer will pass the touch to the layer below it or to the touch mat.
Following is an example of touch-handling areas on the touch mat:

- The touch-handling area of an enabled layer has two effects:
  - It generates touch events for user touches in the area that are not handled by higher layers. Touch events generated by the touch layer **Input** go to the operating system. Touch events generated by the touch layer **Bypass** can be received by the app.
  - It blocks the touch events from being evaluated by lower layers and from being handled by the touch mat.

### The touch layers

Following is more information about using the touch layers:

#### Using the touch layer **Input**

Using a single instance of the touch controller **IPCTouch**, a Sprout app can get the touch layer **Input** (an instance of **IPCTouchLayer** for the layer **Input**). For the touch layer **Input**:

- The touch layer **Input** is the uppermost touch layer.
• When the layer has been gotten and is enabled, the Windows touch keyboard appears on the touch mat.

• When the keyboard appears on the mat, user touches on the keyboard generate touch events that are sent to the operating system, where they are interpreted as key presses.

• User touches on other areas of the touch mat pass through the Input layer, either to the Bypass layer (if it was gotten and is enabled, and if the touches fall on regions inside the geometry of the layer’s mask) or to the touch mat (where the touch events are sent to the operating system).

• When the Input layer is disabled, the keyboard is hidden.

**Using the touch layer Bypass**

Using a single instance of the touch controller IPCTouch, a Sprout app can get the touch layer Bypass (an instance of IPCTouchLayer for the layer Bypass). For the touch layer Bypass:

• The touch layer Bypass is the second and lowermost touch layer.

• When *only* the touch layer Bypass has been gotten and is enabled, user touches on the touch mat inside active areas of the touch layer (inside the geometry of the mask defined for the layer) generate touch events that can be received by the Sprout app, *bypassing* the operating system.

• When *only* the touch layer Bypass has been gotten and is enabled, user touches on other areas of the touch mat (outside the geometry of the mask defined for the layer) pass through the Bypass layer to the touch mat (where the touch events are sent to the operating system).

• If *both* the touch layers Input and Bypass have been gotten and enabled, then the touch layer Input is above the touch layer Bypass. The keyboard blocks the area of the touch layer Bypass that is below it. Otherwise, the touch layer Bypass works in the way described in the previous two bullets.

• When the Bypass layer is disabled, no touch events are sent directly to the app.

**User touches that are not intercepted by a touch layer**

When neither layer is enabled, or when one or both layers are enabled, but user touches are outside of active areas for the layer or layers, user touches result in touch events that are sent to the operating system. The touch events are the same as those that would occur if the Sprout Platform touch management was not in use.

**Examples**

Following are three diagrams that indicate how user touches are handled by the touch layer Input, the touch layer Bypass, and by the touch mat after a touch passes through both of the touch layers. This is not all of the possibilities. For example, cases when only one touch layer is present are not shown.
This is a diagram of a user touch being handled by the touch layer **Input**:

A Sprout app can obtain information about touch events from the operating system.

A user touches the work area of the touch mat. The touch propagates through the touch layers of a Sprout Platform touch controller.

**Touch layers**

**Input layer:**
Enabled; touch occurs on an area of the mask with value 1; touch is handled by this layer.

**Bypass layer:**
Enabled; touch has already been handled by a higher layer, so no evaluation occurs for this layer.

**Touch mat**
The touch mat does not send touch events directly to the operating system.

This is a diagram of a user touch being handled by the touch layer **Bypass**:

A user touches the work area of the touch mat. The touch propagates through the touch layers of a Sprout Platform touch controller.

**Touch layers**

**Input layer:**
Enabled; touch occurs on an area of the mask with value 0; touch is not handled by this layer.

**Bypass layer:**
Enabled; touch occurs on an area of the mask with value 1; touch is handled by this layer.

**Touch mat**
The touch mat does not send touch events to the operating system.
This is a diagram of a user touch being handled by the touch mat, after evaluation by the touch layers:

**Sprout Developer Guide**

**Touch events**

The Sprout app will use Sprout APIs to get information about touch events:

- Using the touch layer **Input**, a Sprout app can show and hide the keyboard. When user touches result in the layer generating touch events, the events are sent to the operating system (they are input to the operating system; hence the name of the layer), where the events are interpreted as key presses. Advantages for using the Sprout Platform to control the keyboard are explained below.

- Using the touch layer **Bypass**, when user touches result in the layer generating touch events, the Sprout app can receive information about the events directly from the Sprout Platform. The events are not sent to the operating system (they bypass the operating system; hence the name of the layer).

Touch events contain information about the coordinates of the touch point in millimeters and the width and height of the bounding rectangle that bounds the touch. For more information, see the API Reference.
Step-by-step
Following are step-by-step instructions for using touch features of the Sprout Platform.

Start with a registered window
To manage touch events on the touch mat in a Sprout app, you start by registering an app window with the platform. This allows the platform to display the window on the touch mat, and to associate touch events with a specific app window.

Remember that the touch mat is not a touch screen. The platform must associate the touch mat with the projected mat screen. And, because all touch events are not just sent to the operating system, the platform must know which app window should receive which touch events.

Create a touch controller
Create a touch controller to manage touch layers and touch events for an app window on the touch mat. When creating the touch controller, you pass the name of a mat window to the touch controller. The window must already exist.

Get touch layers
The touch mat has physical layers, but you do not need to concern yourself with them. The Sprout Platform provides a stack of virtual touch-input layers (or more simply touch layers).

Set the geometry of the touch-handling areas of the touch layer Bypass
If the touch layer Bypass is used, set the geometry of the layer.

Enable and disable layers as needed
For a touch layer to work, it must both have a defined geometry and be enabled. For the touch layer Input, the geometry is pre-defined, so enabling the touch layer is enough. For the touch layer Bypass, your app must do both of these. They can be done in either order. It makes sense to do them together and for error handling to encompass the combined method calls. When the layer Bypass is enabled without the geometry of the layer mask being set, the layer behaves as if it were disabled—no touch events are available to the app.

Redefine the geometry of the touch layer Bypass as needed
The touch-handling area of the touch layer Input has a static geometry. The keyboard is always in the same place.

On the other hand, there is no reason that the touch-handling areas of the Bypass layer need to be static. User interaction with the user interface could result in the desire to change which areas of the touch mat generate touch events that the app can receive directly. The operative word here is “could.” The geometry could be static, for example, designed to receive touch events from the entire touch mat.
Redefining the geometry is accomplished using the method `SetGeometry` or `SetGeometryAsync` that is declared on the interface `IPcTouchLayer`.

**Note:** You do not need to disable and then re-enable a touch layer to set the geometry of the layer. Just set the geometry.

**Dispose of touch-management resources when they are no longer needed**

When touch-management resources are no longer needed, dispose of them.

**The reliability of user touches**

Before leaving the topic of touch management, it is worth mentioning that users to not always touch where they intend to. For example, whether using a physical or touch keyboard, I might touch the letter S instead of the letter A. If I touch precisely half way between the S and the A, it is hard to know which letter I intended to type. User interface designers must consider the realities of user dexterity when designing the size, placement, and interactivity of user-interface controls.

With Sprout, **six** factors might affect the reliability of user touches. The first three of these are ones where user familiarity with Sprout will affect reliability. The final two are small nuances.

- A user will be using a large, horizontal touch surface (the touch mat), which lacks tactile feedback.
- A user will also be using a vertical touch surface (the monitor), which lacks tactile feedback. The novelty here might be use of a vertical touch surface, and will almost certainly be use of both a vertical and horizontal touch surface.
- When using any user interface control, when a user reaches for it, at some point the user’s hand or finger or both hide the control that he or she intends to touch. How you reach, and how and when the controls are hidden, differ for the vertical and horizontal touch surfaces.
- The mat screen is a *projected* screen, so when a user’s finger gets close to a control, the control will be projected onto the user’s finger. But the location of the control on the finger is not exactly the same as the location of the control on the mat, for two reasons. First, a finger is not flat. Second, the projector mirror is not located directly above the screen, so a UI control that is projected onto a finger will appear to be closer to the far edge of the touch mat than it would appear if the projected UI control reached the mat.
- User touches are detected using very precise physical coordinates and interpreted with respect to touch layers that are precisely aligned with the work area of the touch mat and of the same dimensions. But the projected screen might *not* be precisely aligned with the work area on the touch mat. Calibration aligns the projected mat screen with the work area as closely as possible for a specific Sprout immersive computer. In some cases, the alignment will be exact or very close. In other cases, there might be a slight difference (on the order of a millimeter or two) between the position of the work area of the touch mat and the position of the projected mat screen. The mat screen might of a slightly different size vertically and/or horizontally, or it might be displaced, rotated, or warped.
(WPF only) In the WPF binding, the geometry of the touch layer Bypass is defined using the class Geometry, which can define a touch-handling area with an outline that does not precisely include or exclude some pixels. If the touch-handling area is a circle, then (1) some pixels will be entirely inside the circle, (2) some pixels might have the outermost vertex on the circle, (3) some pixels will have some of the pixel inside the circle and some outside, and (4) some pixels will be entirely outside the circle. The touch-handling area of the layer mask includes only the pixels for cases (1) and (2). In practice, the tip of a user’s finger is much larger than a pixel, so this consideration is mostly of theoretical interest.
Moments

Sprout captures moments. A moment is a moment in time. It is like a snapshot with a camera, but with Sprout there are multiple cameras. Sprout can also create moments from images, for example, from pictures taken with a camera or obtained from the Web.

Capturing a moment

Sprout captures a moment using its cameras that face the touch mat. This diagram illustrates capturing a moment:
For each camera, the Sprout Platform stores data in the moment for the entire work area on the touch mat (for example, a high-resolution image of the work area) and for the individual objects (for example, a high-resolution image of each object).

Most Sprout Platform features depend on having a moment. For example, extracting pictures, extracting outlines, refining pictures, and refining outlines all require that the moment has been captured (or created in the case of a moment created from an image) and that the moment still exists (it has not been disposed of).

To obtain data from the moment, including pictures and outlines, you must extract the data from the moment using API methods. For example, to extract pictures from a moment, you call `ExtractPicture` | `ExtractPictureAsync`.

**What a captured moment contains**

A captured moment contains the following data:

- *(Immediately after the capture completes):* **Image data from the Sprout cameras is added to the moment.** This content in the moment never changes. You cannot access the data directly. The data is not ready for presentation to users.

  To retrieve images and other data that are useful for presentation to users, use extract methods in the interface `IPcLink`, for example, `ExtractPicture` | `ExtractPictureAsync`. You can specify an extract option when calling an extract method. The extract option instructs the Sprout Platform to perform specific additional processing and to return specific results.

- *(After a request is made to extract pictures or outlines):* **A tree structure of information that includes the requested data, and possibly other data, is added to the moment, and the requested data is returned by the extraction method.** Exactly what is stored in the moment and returned in the tree structure depends on the extract option that was used in the request. Calling an extract method for pictures or outlines adds data about both pictures and outlines to the moment, but returns only the requested data (about pictures or outlines). Calling the other method returns the other data, and it can do so more quickly, because the data has already been calculated and added to the moment.

- *(After a request is made to extract text):* **A tree structure of information that includes the requested information, and possibly other information, is added to the moment and returned by the extract method.** Exactly what is stored in the moment and returned in the tree structure depends on the extract option that was used in a prior request to extract pictures or outlines, if one was made. If there was no prior request to extract pictures or outlines from this moment, then the Sprout Platform automatically adds picture and outline data to the moment as if an extract method for pictures or outlines had been called, using the extract option `Object`.

If a moment already contains the requested data, the data is not created again. If the moment does not contain the requested data, then the data is created and is added to the moment, replacing the tree-structure of data that was already there.
The tree structure

Data in the moment are organized in tree structures, and when you extract the data, you get these tree structures.

For a captured moment, the following are true:

- The parent is the entire work area.
- The individual objects are children.
- Pictures have a hierarchy of two levels—the parent (a picture of all of the objects) at the top and all children (pictures of the individual objects) immediately below.
- Outlines also have a two-level hierarchy. For outlines, the parent is not an outline of the objects, but rather an outline of the work area.
- Text hierarchies can have more levels.

For a created moment, the following are true:

- The parent is the entire work area.
- There are no children.

Dispose of moments

A moment consumes considerable memory resources, which might affect system performance, stability, or both. For this reason, we recommend that:

- Your app should manage the workflow with respect to capturing moments.
- When your app is finished with a moment, dispose of the moment to free up the resources and to maintain system performance.

For more information, see “Disposing of moments and links.”

Monitoring the progress of a moment capture

Capturing a moment can take a small amount of time. The amount of time that it takes is not a fixed value. The time depends on camera-specific details such as whether the camera has been initialized. So when the flash occurs relative to when a user tapped a user-interface control also varies.

By using a capture-progress monitor when capturing a moment, a Sprout app can find out when the flash occurs. This allows an app to synchronize user-interface feedback to the user (for example, a shutter sound) with the flash. The moment that is returned by this method overload is identical to the moment returned by the method overload that does not use the capture-progress monitor, an instance of the interface `IPcMoment`. 
Binding interfaces and classes
Following are the interfaces and classes that pertain to capture-progress monitoring:

IPcLink, IPCaptureProgressMonitor, IPcCaptureProgressObserver, and
IPcCaptureFlashEventArgs

Following is the method overload in the interface IPcLink for capturing a moment that uses a
progress monitor:

virtual PcResult<IPcMoment>
CaptureMoment(IPcCaptureProgressMonitor* captureProgressMonitor) = 0;

Sample app source code (CppQtSample)
For relevant source code in the sample app WpfCsSample, tap Show Source Code for the sample
Pictures and Outlines.

Summary of approach
To monitor the progress of a moment capture:

1. Create a platform link, an instance of the interface IPcLink. Captured moments will be
   associated with the link.
2. Create an instance of the capture-progress monitor interface
   IPcCaptureProgressMonitor.
3. Create an instance of the capture-progress observer interface
   IPcCaptureProgressObserver.
4. Add the capture-progress observer to the capture-progress monitor.
5. Use the capture-progress observer to observe the event FlashEvent. In response to receipt
   of the event, do something, for example, play a shutter sound.

WPF
Binding interfaces and classes
Following are the interfaces and classes that pertain to capture-progress monitoring:

IPcLink, IPCaptureProgressMonitor, and PcCaptureFlashEventArgs
Following is the method overload in the interface IPcLink for capturing a moment that uses a progress monitor:

```csharp
Task<IPcMoment> CaptureMomentAsync(IPcCaptureProgressMonitor captureProgressMonitor);
```

Sample app source code (WpfCsSample)

For relevant source code in the sample app WpfCsSample, tap Show Source Code for the sample Capture Audio.

Summary of approach

To monitor the progress of a moment capture:

1. Create a platform link, an instance of the interface IPcLink. Captured moments are associated with the link.
2. Create an instance of the progress-monitor interface IPcCaptureProgressMonitor.
3. Subscribe to the event Flash.
4. Capture a moment.
5. In response to receipt of the event Flash, do something, for example, play a shutter sound.
6. Unsubscribe from the event Flash.

Creating a moment from an image

Sprout can also create a moment from an image, for example, an image from the Web.

A moment that is created from an image contains:

- **The image**: This is the original image. You cannot access the image directly. Instead, use the various “Extract” methods in the interface IPcLink, for example, ExtractPictureAsync. When extracting this image, it is at the top level of the hierarchy, and there are no children.

- **A bounding rectangle and an outline of the entire image**: The Sprout Platform does not perform an automatic segmentation. It creates a bounding rectangle and outline for the entire image, and the bounding rectangle and the outline are the same.
This diagram illustrates creating a moment from an image:

An image in a file, for example, taken with a camera or from the Web

Command to create a moment from an image

```
01011010101
00101010100
01001110101
11010101100
10100100000
11101110100
```

Moment

The moment we have all been waiting for. At the moment, it contains just data. To get useful things from the moment, you must extract them.

How an alpha channel is handled

When a moment is created from an image file, for example a file from the Web, if the file contains an alpha channel, then the image data in the moment retains the alpha channel, and an image extracted from the moment retains the alpha channel.

For information about how an alpha channel is handled during assisted segmentation, see “How an alpha channel is handled during assisted segmentation.”
Pictures and outlines

Captured moments contain camera data—images and other data, but not yet images that are presentable to a user. A moment that is created from an image contains the image. The SDK provides APIs for extracting pictures and outlines from both of these kinds of moments, as well as for refining the pictures and outlines.

Extract options for pictures and outlines

Note: The method overloads of the methods `ExtractPicture` | `ExtractPictureAsync` and `ExtractOutline` | `ExtractOutlineAsync` that take an extract option can be used for both captured and created moments. However, for created moments, the extract option is ignored. Extracting pictures and outlines from a created moment always returns the original image as the top-level image in the tree (with no children) and the top-level outline that corresponds to it.

When extracting pictures or outlines from a moment, you can specify extract options. Extract options give the Sprout Platform guidance regarding what the moment contains and what you intend to do with the extracted pictures and outlines.

Alternatively, you can use a method overload that does not specify an extract option (in which case the Sprout Platform chooses an extract option for you). And you can do a combination of both of these approaches.

You access extract options for pictures through the specification interface `IPcPictureExtractSpecification`. You access extract options for outlines through the specification interface `IPcOutlineExtractSpecification`.

True for all extract options

Following are true for all extract options:

- An instance of `IPcPicture` or `IPcOutline` always contains a tree structure. In some cases, the tree structure only has a top level; there are no children.
- For pictures and outlines, the bounding box of the top level picture or outline is always of the entire mat.
- For outlines, the contour of the top-level outline is always of the entire mat.
• The goal is to have the position and dimensions of a bounding box exactly match the horizontal and vertical positions and dimensions of the contour. This is sometimes the case, but it is also possible for the bounding box to be slightly larger in one or more directions. In all directions, the bounding box will never be inside of the contour.

• Small objects (smaller than about 7 x 7 mm square or 8 mm in diameter) are not segmented and do not result in children in the tree structure. The objects are present in pictures of the entire mat.

**Extract options**

Following are the extract options for pictures and outlines:

<table>
<thead>
<tr>
<th>Extract option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BasicProcessing</strong></td>
<td>When using the extract option <strong>BasicProcessing</strong>, extraction algorithms return only information that corresponds to the entire work area on the touch mat. Segmentation is not performed. One goal of the extract option <strong>BasicProcessing</strong> is to return information as quickly as possible. No enhancement of the picture is performed, and the picture might have a lower resolution (smaller pixel extents) than the picture of the mat that is returned when using the extract option <strong>Mat</strong>. The tree structures returned in <strong>IPcPicture</strong> and <strong>IPcOutline</strong> have one level. The top level contains an image or outline of the entire work area of the touch mat.</td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td>When using the extract option <strong>Object</strong>, extraction algorithms are optimized for extracting information from a captured moment that contains objects. The platform performs segmentation and returns all of the information about the requested item. The tree structures returned in <strong>IPcPicture</strong> and <strong>IPcOutline</strong> have two levels. The top level contains an image or outline of the entire work area of the touch mat. The second-level images or outlines are of objects on the touch mat.</td>
</tr>
<tr>
<td><strong>Mat</strong></td>
<td>When using the extract option <strong>Mat</strong>, extraction algorithms return only information that corresponds to the entire work area on the touch mat. Segmentation is not performed. One goal of the extract option <strong>Mat</strong> is to return information more quickly than is the case when using the extract option <strong>Object</strong> (but not as quickly as when using <strong>BasicProcessing</strong>). The tree structures returned in <strong>IPcPicture</strong> and <strong>IPcOutline</strong> have one level. The top level contains an image or outline of the entire work area of the touch mat.</td>
</tr>
<tr>
<td>Extract option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| Document       | When using the extract option `Document`, extraction algorithms are optimized for extracting information from a captured moment that contains a single, rectangular document and only the document. The platform performs segmentation and returns all of the information about the requested item. If one rectangular document is present on the mat, then:  
  - The tree contains two levels. The top level contains the picture or outline of the entire mat. The second level contains the picture or outline of the document.  
  - In the outline, the contour of the rectangular document is forced to be more exactly rectangular.  
If anything else is present on the touch mat (for example, a non-rectangular object, multiple documents, or a combination of these), then the tree has one level, which contains the picture or outline of the entire mat. |

**Note:** When you use the extract option `Object`, extraction algorithms return a picture or an outline of the work area of the touch mat, the same picture or outline that would be returned when using the extract option `Mat`. These are at the top level of the tree that is returned in instances of the interfaces `IPcPicture` and `IPcOutline` respectively. So, there is no need to call the extract method for a picture or outline twice, once with the extract option `Object` and once with the extract option `Mat`. If you need pictures and outlines for objects on the touch mat and a picture and outline of the work area of the touch mat, just use the extract option `Object` and traverse the tree to get what you need.

**Specifying an extract option**

One of the method overloads for the methods `ExtractPicture` | `ExtractPictureAsync` and `ExtractOutline` | `ExtractOutlineAsync` permits you to specify an extract option. The extract options are explained in the prior section.

The reasons for specifying an extract option are:

- You have more control over the results of the extraction, and ultimately of the user experience.
- You can indicate that the Sprout Platform can omit some calculations that are not needed, which improves the time required to extract pictures or outlines.
- If necessary, you can change the pictures and outlines that are available (for a captured moment).
Following are the method declarations:

**C++**

```cpp
virtual PcResult<IPcPicture> ExtractPicture(IPcMoment* moment, specification::IPcPictureExtractOption* option) = 0;

virtual PcResult<IPcOutline> ExtractOutline(IPcMoment* moment, specification::IPcOutlineExtractOption* option) = 0;
```

**WPF**

```csharp
Task<IPcPicture> ExtractPictureAsync(IPcMoment moment, IPcPictureExtractOption option);

Task<IPcOutline> ExtractOutlineAsync(IPcMoment moment, IPcOutlineExtractOption option);
```

**Letting the Sprout Platform decide the extract option**

One of the method overloads for the methods `ExtractPicture` | `ExtractPictureAsync` and `ExtractOutline` | `ExtractOutlineAsync` permits you to allow the Sprout Platform to choose the extract option.

Following are the method declarations:

**C++**

```cpp
virtual PcResult<IPcPicture> ExtractPicture(IPcMoment* moment) = 0;

virtual PcResult<IPcOutline> ExtractOutline(IPcMoment* moment) = 0;
```

**WPF**

```csharp
Task<IPcPicture> ExtractPictureAsync(IPcMoment moment);

Task<IPcOutline> ExtractOutlineAsync(IPcMoment moment);
```
Letting the Sprout Platform decide the extract option has these outcomes:

<table>
<thead>
<tr>
<th>Extraction method call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First call to extract pictures or outlines</strong></td>
<td>If no extract option is specified for the first call of a method to extract pictures or outlines from a moment, then in the current implementation the Sprout Platform uses the extract option <strong>Object</strong>, and the method call extracts high resolution pictures of the work area and of objects on the work area.</td>
</tr>
<tr>
<td><strong>Subsequent call to extract pictures or outlines from the same moment</strong></td>
<td>For a subsequent call of a method to extract pictures or outlines from a moment, in the current implementation the Sprout Platform uses the same extract option (explicit or decided by the platform) as the immediately prior method call. This permits the platform to use data that has already been extracted and to perform fewer computations.</td>
</tr>
</tbody>
</table>

**A series of calls to extract methods for pictures and outlines**

When a series of method calls extract pictures or outlines, the Sprout Platform does the following for method calls after the first one:

- If the specified extract option is the same as the extract option used in the immediately prior method call to extract pictures or outlines (whether explicitly or through a decision by the platform), then the data needed has already been added to the moment. That data is used to return the requested pictures or outlines.

- If the specified extract option differs from the extract option used in the immediately prior method call to extract pictures or outlines (whether explicitly or through a decision by the platform), then the Sprout Platform performs new calculations using the new extract option and replaces the data in the moment with new data. That new data is used to return the requested pictures or outlines.

Why is this important? One of the goals of extract options is to give you what your app needs. A second is to do it as quickly as possible. If your app only needs an image of the work area of the touch mat, then specifying the extract option **BasicProcessing** or **Mat** will save time and make the app more responsive. And if the moment already contains the data that you need, then using the same extract option that was used previously to extract it will save time. Actually, saving everything that you will need after one method call to extract pictures or outlines will save more time.

**Also extracting text**

When you extract text from a moment, the method call to extract text relies on a prior extraction of pictures, or performs one for you. For more information, see “Impact of prior picture or outline extraction.”
Outlines

To work with outlines in a captured moment (or the single outline in a moment that was created from an image), you must extract the outlines from the moment using `ExtractOutline` or `ExtractOutlineAsync`.

**Note:** You are responsible for persisting and keeping track of everything that you extract from a moment, and of new versions of things such as refined pictures and outlines.
Extracting outlines from a captured moment using the extract option `Mat`

This diagram illustrates extracting outlines from a captured moment using the extract option `Mat`. Using the extract option `BasicProcessing` produces a similar result.

**Moment**
An instance of the interface `IPcMoment`.

After capture, the moment contains camera data. To get outlines from the moment, you must extract them.

**Command to extract outlines from the moment using the extract option `Mat`**

Automatic segmentation will not be done.

**Outlines**
The variable `outline` is an instance of the interface `IPcOutline`.

- `outline.Contour` is the contour of the mat image. It does not include the contours of the objects.
- `outline.PixelDensity` gives the number of pixels per millimeter for both the X and Y axes.
- Child outlines are not present in the tree structure, because automatic segmentation was not performed.
- `outline.PhysicalBoundaries` is a bounding rectangle.
Extracting outlines from a captured moment using the extract option Object

This diagram illustrates extracting outlines from a *captured* moment using the extract option Object:

- **Outlines**: The variable `outline` is an instance of the interface `IPcOutline`.
  - `outline.ConTour` is the contour of the mat image. It does not include the contours of the objects.
  - `outline.PixelDensity` gives the number of pixels per millimeter for both the X and Y axes.
  - `outline.PhysicalBoundaries` is a bounding rectangle.

- **Top-level outline**
- **Children** `outline.Children`
  - Each child outline is also an instance of the interface `IPcOutline`, providing:
    - `child.ConTour`
    - `child.PhysicalBoundaries`
    - `child.PictureDensity`

For outlines, the children do not have children.
**Extracting outlines from a created moment**

For a *created* moment, extracting outlines gives the same result as extracting outlines from a captured moment, when the extract option is Mat. For more information, see “Extracting outlines from a captured moment using the extract option Mat.”

**Refining outlines**

A user can refine the initial outline or outlines present in a moment. This is the first step in assisted segmentation. The second step is refining pictures based on the refined outlines.

There are differences between the initial outline or outlines for captured and created moments:

- For a *captured* moment, the Sprout Platform uses data from multiple cameras to create the outlines in the moment. It starts the process with physical objects, and will probably separate the objects correctly and produce reasonable outlines. The result is a tree-structure with a parent outline and child outlines.

- For a *created* moment, the Sprout Platform has only a single image from which to create outlines. It does not start with objects. The initial outline (the parent) is just an outline of the image, and there are no child outlines.

Refined outlines are not used to replace the outlines in the moment. Instead, new outlines are created. *A moment is never changed.*

To refine an outline, use `RefineOutlineAsync`. For examples of usage, see these sample app files:

- AssistedSegmentationControl.xaml.Display.cs
- BitmapAssistedSegmentationControl.xaml.Display.cs

**Pictures**

To work with pictures in a captured moment (or the picture in a moment that was created from an image), you must extract the pictures from the moment using `ExtractPicture` | `ExtractPictureAsync`.

**Note:** You are responsible for persisting and keeping track of everything that you extract from a moment, and of new versions of things such as refined pictures and outlines.
Extracting pictures from a captured moment using the extract option Mat

This diagram illustrates extracting pictures from a captured moment using the extract option Mat. Using the extract option BasicProcessing produces a similar result.

```
01011010101
00101010100
01001110101
11010101100
10100100000
11101010000
```

**Moment**
An instance of the interface IPCMoment

After capture, the moment contains raw camera data. To get pictures from the moment, you must extract them.

**Command to extract pictures from the moment using the extract option Mat**

Automatic segmentation will not be done. If segmented data are needed, an extraction request that uses the extract option Object must be used.

**Pictures**
The variable picture is an instance of the interface IPCPicture.

```
picture
```

picture.Image is an image of the entire mat.

picture.PixelDensity gives the number of pixels per millimeter for both the X and Y axes.

Child pictures are not present in the tree structure, because automatic segmentation was not performed.

```
picture.PhysicalBoundaries is a bounding rectangle.
```
Extracting pictures from a captured moment using the extract option Object

This diagram illustrates extracting pictures from a captured moment using the extract option Object:

**Top-level picture**

- **picture**
  - `picture.Image` is an image of the entire mat.
  - `picture.PixelDensity` gives the number of pixels per millimeter for both the X and Y axes.
  - `picture.PhysicalBoundaries` is a bounding rectangle.

**Children**

- `picture.Children` Each child picture is also an instance of the `IPcPicture` interface, providing:
  - `child.Image`
  - `child.PhysicalBoundaries`
  - `child.PixelDensity`

For pictures, the children do not have children.
Each child image has an alpha channel mask applied when extracted, so that it consists of pixels that have color values (for example, the spots on a tiger) and pixels that are transparent (transparent pixels replace the background, which otherwise would be an opaque image of the touch mat). This background removal makes it easier for the user of an application to create a composition that contains multiple images.

**Extracting pictures from a created moment**

For a *created* moment, extracting pictures gives the same result as extracting pictures from a captured moment, when the extract option is **Mat**. For more information, see “Extracting pictures from a captured moment using the extract option **Mat**.”

**Refining pictures**

After you have a refined outline, you can refine a picture based on the outline.

Refined pictures are not used to replace the pictures in the moment. Instead, new pictures are created. *A moment is never changed.* If desired, you could create a new moment from the refined picture.

To refine an outline, use **RefinePictureAsync**. For examples of usage, see these sample app files:

- AssistedSegmentationControl.xaml.Display.cs
- BitmapAssistedSegmentationControl.xaml.Display.cs

**Using the skew angle to make a picture or outline upright**

The interface **IPcOutline** provides a *skew angle* for each outline in the **IPcOutline** tree structure. Similarly, the **IPcPicture** interface provides a skew angle for each picture in the **IPcPicture** tree structure.

```csharp
double SkewAngle { get; }
```

The skew angle is the rotation needed to make the axes of the minimum area rectangle around the outline or picture align with the X and Y axes.

The above information might be all you need to know. Get the angle from the interface **IPcOutline** or **IPcPicture**, apply the angle to the outline or image, display the result, and you are done. In the event that you want more information, it is below.

The skew angle is in radians. Positive angles are clockwise rotations. Negative angles are counterclockwise rotations.
In what follows, we consider two angles:

Φ  The angle between the positive X axis and the axis of the minimum area rectangle surrounding the outline or picture that falls in the range of angles that are less than or equal to zero and greater than \(-\pi/2\).

Θ  The skew angle, that is, the rotation needed to make the axes of the minimum area rectangle around the outline or picture align with the X and Y axes.

The following diagram illustrates the case when the skew angle is positive.
The following diagram illustrates the case when the skew angle is negative.

Values of the minimum area rectangle axis rotation angle \( \phi \) and the corresponding skew angles \( \theta \) are as follows:

<table>
<thead>
<tr>
<th>Rectangle axis rotation angle ( \phi )</th>
<th>Skew angle ( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi = 0 ) radians</td>
<td>( \theta = 0 ) radians. No rotation is needed, because the axes of the minimum area rectangle already align with the X and Y axes.</td>
</tr>
<tr>
<td>(-\pi/4 &lt; \phi &lt; 0)</td>
<td>( \theta = -\phi ) radians. Relative to the positive X axis, the minimum area rectangle axes are rotated ( \phi ) radians (counterclockwise; ( \phi ) is negative). The angle of rotation ( \phi ) is less than (-\pi/4). The skew angle ( \theta ) to correct for that rotation is (-\phi) radians (clockwise; (-\phi) is positive). For example, if ( \phi = -\pi/12 ) radians, then ( \theta = \pi/12 ) radians.</td>
</tr>
</tbody>
</table>
**Rectangle axis rotation angle φ**  

<table>
<thead>
<tr>
<th>Rectangular axis rotation angle φ</th>
<th>Skew angle Θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\pi/2 &lt; \phi \leq -\pi/4$</td>
<td>$\Theta = -\pi/2 - \phi$ radians. Relative to the positive X axis, the minimum area rectangle axes are rotated $\phi$ radians (counterclockwise; $\phi$ is negative). The angle of rotation $\phi$ is greater than or equal to $-\pi/4$ and less than $-\pi/2$. The skew angle $\Theta$ to correct for that rotation is $-\pi/2 - \phi$ radians (counterclockwise; $-\pi/2 - \phi$ is negative). For example, if $\phi = -\pi/3$ radians, then $\Theta = -\pi/6$ radians.</td>
</tr>
<tr>
<td>$\phi = -\pi/2$ radians</td>
<td>$\Theta = 0$ radians. This is identical to the case of $\Theta = 0$ radians.</td>
</tr>
</tbody>
</table>

**Note:** The skew angles $\Theta$ of the top-level outline and top-level picture extracted from a moment are 0 radians, because the axes of the minimum area rectangle are already aligned with the X and Y axes. There is no need to rotate the top-level outline or top-level picture by the skew angle.
Assisted segmentation

Assisted segmentation allows users to refine the edges of pictures of segmented objects in captured moments, for example, to remove shadows. Assisted segmentation also allows users to remove undesired content from a picture extracted from a created moment. We sometimes refer to the latter as “background removal,” which is a common use case.

Assisted segmentation of an object in a captured moment

Following are examples of assisted segmentation of an object in a captured moment, using the Assisted Segmentation tab in the sample app WpfCsSample.

Before the assisted segmentation:
The strokes that indicate what parts of the image to include and exclude:

The result of assisted segmentation:
Assisted segmentation of the image in a moment created from an image

Following are examples of assisted segmentation of a captured moment, using the Background Removal tab in the sample app WpfCsSample.

Before the assisted segmentation:
The strokes that indicate what parts of the image to include and exclude:

The result of assisted segmentation:
How an alpha channel is handled during assisted segmentation

If a moment is created from an image that contains an alpha channel, then the image data in the moment retains the alpha channel, and a picture extracted from the moment retains the alpha channel.

If the outline or picture extracted from the moment is refined using assisted segmentation, then the original alpha channel is respected as follows:

- The area masked by the alpha channel in the original image is never revealed. The alpha channel will always cover those areas.
- During assisted segmentation, a user can specify exclusion strokes on areas of the image that are not masked by the alpha channel. The outline or picture is adjusted accordingly. If the picture is saved as an image, then the excluded areas will be added to the alpha channel.
- Before the image is saved, a user can specify inclusion strokes that re-include previously excluded areas on the image, but only on areas that he or she previously excluded, not on the area that is masked by the original alpha channel.
Which areas can and cannot be included during assisted segmentation is illustrated in the following diagram:

1. A butterfly. The transparent area around it is masked by an alpha channel.

2. The alpha channel

3. What remains after assisted segmentation (*simulated*)

4. The user can restore some or all of the areas in green by additional assisted segmentation.

   The user cannot add the area in pink, which remains hidden by the alpha mask.
Object classification

Objects have physical characteristics that Sprout can identify. Some objects are flat. Others are three dimensional. Some objects are rectangular. The Sprout Platform can return classification tags for objects in a captured moment.

Objects and classification

Sprout captures color, infrared, and depth information when it captures a moment. It uses this information to locate objects and to perform an automatic segmentation. Sprout is also able to use the information to classify the objects in a captured moment. Sprout classifies objects when you call the method `ClassifyAsync` in the interface `IPcLink`. This is the method declaration:

```csharp
Task<IPcClassification> ClassifyAsync(IPcMoment moment);
```

This is an example of the method call:

```csharp
var classification = await sdk.ClassifyAsync(moment);
```

Available classification tags

You can refer to a specific classification tag or obtain all supported classification tags by using the interface `IPcClassificationSpecification` through an instance of the interface `IPcClassification`.

The instance of the interface `IPcClassification` that the method `ClassifyAsync` returns contains a tree structure that matches the tree structure for text (in the interface `IPcText`). If none of the objects that are detected contain text, then the tree structure for `IPcClassification` also matches the tree structures for pictures (in the interface `IPcPicture`) and outlines (in the interface `IPcOutline`).

The top-level node of the tree structure in the interface `IPcClassification` contains all of the tags that apply to one or more nodes in the entire tree structure. Nodes at lower levels contain tags that apply to the specific nodes.
Captured and created moments can contain text. The SDK provides APIs for extracting text from moments.

**Extracting text from a moment**

Sprout can capture and create moments that contain text elements, for example, post cards, recipes, receipts, and so forth. The Sprout Platform can recognize text elements in image data in a moment. You can use the method `ExtractText` | `ExtractTextAsync` to extract text from a moment.

Following is an example of text extraction using the WPF sample app `WpfCsSample`. The mouseover shows the extracted text, which exactly matches the actual text.

**Impact of prior picture or outline extraction**

To extract text, a picture of the text needs to have been extracted first (or outlines need to have been extracted), which includes processed images in the moment. You can take either of these approaches:

- **Do not extract pictures first**: In this case, the Sprout Platform extracts the pictures automatically, using the extract option `Object`. Text extraction uses the top-level mat image.
Sprout Developer Guide

- **Extract pictures first, not specifying an extract option**: In this case, the extract option `Object` is used. Text extraction uses the top-level mat image.

- **Extract pictures first, specifying an extract option**: In this case, the specified extract option is used. In all cases, the top-level mat image is used for text extraction. The value of specifying the extract option `Document` is that the second-level image is of the document, and it includes a skew angle that can be used to straighten the document before extracting text.

**Text extraction functionality has known limitations**

Text extraction functionality has known limitations that affect whether the Sprout Platform is able to extract text from moments, and the quality of the text extraction.

Note the following current limitations:

- Only English text is recognized.
- Dark text on a light background is recognized; light text on a dark background is not. Higher contrast works better than lower contrast.
- Some font sizes and font faces result in reduced extraction accuracy.
- In some cases, the Sprout Platform does not perceive text to be text. A method call to extract text can return no text, even though text is present.
- In some cases, the Sprout Platform recognizes text, but letters are identified incorrectly. Extracted data can contain invalid characters.
- Extracted text can be incomplete.
- The order in which extracted texts are concatenated in the top-level tree node and/or in intermediate-level tree nodes might not match what the user expects.

Here are some guidelines regarding which documents work better for text extraction:

- Sprout only extracts darker texts on lighter backgrounds.
- Texts are only extracted accurately if the texts are *upright* (they run straight across the work area of the touch mat, and are oriented for you to read the texts). Text that runs at an angle, either because it is that way on a document or because a user places a document on the touch mat at an angle, is extracted less accurately. Suggestions to the user, program logic, and possible use of the skew angle to make documents upright can mitigate this issue.
- Extraction works better when there is a moderate to large amount of text, for example, for the text on a letter-sized document. Extraction is less accurate for smaller amounts of text.
- Some font sizes and font faces can result in reduced extraction accuracy.
App logic

Because the quality of text extraction varies depending on the quality of the original text, the user workflow of an app that uses Sprout's text extraction should do the following:

- Instruct the user to place the document on the touch mat such that the text is in reading orientation, with lines of text parallel to the near edge of the touch mat. If the text on a document or object is at an angle, then the text should be placed so it is level, not the document or object.
- Consider using the skew angle to make a document more upright. If the user has been guided to place a document on the touch mat carefully, this might not be needed.
- Provide accept/reject logic and capture-again logic.
- Provide the ability to compare the original and the text extraction.
- Provide the ability to edit the extracted text, to correct any mistakes.

Using the skew angle to make text upright

The following example shows what happens when text is not upright. The text is the same as that shown above, for which the extraction accuracy was 100%.

Here, the accuracy is about 81%. Enough errors have been introduced by the rotation of the text that the result would be unusable.

If a captured image contains text that is not upright because the image is not upright, you might be able to use the skew angle to straighten the image before text extraction is attempted. The skew angle can be used to align the axes of the minimum-area rectangle that surrounds a text to correspond with the X and Y coordinates of the touch mat. This is only possible for captured moments.

The skew angle can be used if the text runs across a rectangular document. The skew angle cannot be used if the text is at an angle relative to the edges of a document.
Because both cases where adjusting for the skew angle will help and others where adjusting for it will hurt will apply, an app should not assume that it must use the skew angle. If used, the control should be given to the user.

To use the skew angle:

1. Extract pictures from the captured moment, specifying the extract option Document.
2. Use the skew angle for the second-level picture, which is of the document, to create an image of that picture that is upright.
3. Create a new moment using the upright image.
4. (Optional) Extract pictures from the created moment.
5. Extract text from the created moment.

For information about the skew angle, see “Using the skew angle to make a picture or outline upright.”
Object recognition and tracking

Sprout can recognize and track one or more 2D objects, or 2D representations of 3D objects (such as photographs), that are placed on or held above the touch mat. You train Sprout to recognize the objects by providing the object-tracking handler with training images.

Overview

During object tracking, Sprout takes video of the touch mat, and of everything placed on or held above the touch mat. In these video frames, Sprout can recognize specific objects (object recognition) that it has been trained to recognize (training), and then track the movements of the objects (object tracking).

Sprout can recognize and track 2D objects and 2D representations of 3D objects (for example, printed photographs).

When an instance of the object-tracking handler has been created and started, the platform looks for the objects to be tracked in video frames and raises events when the objects are recognized in the video frames. In each event, the payload contains information about the objects recognized and their locations in physical coordinates.

In C++, one or more object tracking observers are also needed to observe the events.

The object-tracking handler manages the video. You do not need to do so.
This drawing portrays object recognition and tracking:

**Representing on the monitor when recognition occurs**

These poppies were recognized in video frames.

**Work area of the touch mat**

- Rotation around the Z axis and movement in the X-Y plane
- Rotation around a line in the X-Y plane

This is the training image, but it has been tilted up too much to be recognized.

Although also poppies, these images were not training images, so they are not recognized.
**Training**

Training uses a set of one or more bitmap images of specific objects. The bitmaps can be created from pictures extracted from one or more moments; or from other images, for example ones from the Web or taken with a digital camera.

Training is instructing Sprout to look for *this exact object or these exact objects*. In fact, because you use images to train the object-tracking handler, training is instructing Sprout to *look for images that are identical to one or more of the training images, or that match a training image closely enough*.

Sprout *cannot* be trained using multiple examples of a class of object (for example, lilies) to recognize that a specific object belongs to the class of objects.

You train an object-tracking handler when you create it, by providing a set of training images. For as long as that instance of the object tracking handler exists, it can recognize and track only those objects. To track a different object or set of objects, you must create a new instance of the object-tracking handler. Only one instance can exist at a time.

**Suitable objects for training and recognition**

Sprout can recognize and track *2D objects* and *2D representations of 3D objects* (for example, printed photographs).

Object recognition and tracking rely on the recognition of unique features in images of 2D objects. Because of this:

- Objects that have too few features or too many features can present challenges for the object-tracking algorithms.
- The accuracy of object recognition and tracking differ from object to object.
- Some objects are not recognized.

Here are some notes about objects that can be used for training and about how well Sprout will recognize them. Different objects will give different user experiences. App planners and developers should consider these nuances.

**Why not 3D objects?**

Because Sprout recognizes objects by comparing features in 2D video frames with features in 2D training images, Sprout does not know whether an object that is being asked to recognize and track is a 2D object or a 3D object. So, Sprout will try to recognize *all* objects that it has been trained to recognize, and that are placed on or held above the touch mat while object tracking is in progress.
Sprout can’t do a good job recognizing and tracking 3D objects, because the features that are present change when a 3D object is rotated, and the relationships between the features in a 2D projection (the training image or video frame) change too. Consider this comparison:

- If I obtain a training image from a moment capture of a 2D object, for example, a post card lying on the touch mat, then the object tracking handler will manage well. When I move the post card around on the mat, rotate it in two dimensions, and even (to some degree) rotate it in three dimensions, then these changes will not cause problems with object recognition and tracking.

- If I obtain a training image from a moment capture of a 3D object, for example, a stuffed bear facing the Sprout cameras, then the object-tracking handler will do moderately well as I move the bear around on the mat (keeping it facing the cameras), or rotate the bear around the Z-axis (the depth axis), or lift the bear up a bit toward the cameras. It will not do as well for these movements as it would with a 2D object. The object-tracking handler will do poorly if I rotate the stuffed bear around a different axis. First, the video images used for recognition and tracking will contain images of a bear with two eyes and a nose and a mouth. A bear from the side has one eye and one nose and one mouth, and these features all look different from face on. From the back, a bear has no eyes, no nose, and no mouth, but instead has a tail.

In summary, Sprout will try to recognize and track 3D objects (because it is just working with images, and because users can try 3D objects), but it will not do as good a job recognizing and tracking 3D objects.

What about almost-2D objects?

A book or a flat box is a 3D object, but each has a significant 2D aspect (the book cover and the top of the box). Sprout will do reasonably well recognizing and tracking objects such as these, if the large 2D surfaces have enough features.

Features

Sprout recognizes and tracks features in video frames, based on the same features in training images. Features can be thought of as changes in the lightness and darkness of adjacent areas. A photograph of a beige wall has few features. A photograph of orchids and a toucan in a rain forest has more features.

Objects with more features are recognized and tracked more reliably than objects with few features.

Edges of a plain object are not good features

A pattern cut from a plain piece of paper has edges with a specific shape, but those edges do not make good features. Drawing lines near the edges increases the precision of recognition and tracking, though even in this case, the number of features might not be enough for object recognition and tracking.

Colors

During object recognition and tracking, Sprout does not consider color.
Different sizes of the same object

Sprout is able to recognize different sizes of the same object as the same object. For example, if you print a 5 x 7 inch photograph and a 2.5 x 3.5 inch copy of the same photograph, and use one or the other for training, then Sprout will recognize both sizes as the same object.

Similar objects

If two training images are similar, then Sprout might consider that an object presented to it matches both. Sprout will report both matches.

Orientation of objects

Sprout assumes that that starting point for object recognition and tracking is a 2D object placed flat on the touch mat, or an arbitrary 2D image. Each of these is perpendicular to a line of sight, and thus is "maximally spread out" feature-wise.

If I capture a moment with a playing card lying on the touch mat, extract the picture, create a bitmap, and use the bitmap for training, then when I again place the playing card on the touch mat during object tracking, the same features will be present in the same locations. If I rotate the playing card around the X or Y axis or both, then the features will get more and more compressed, until the card is edge on and all of the features are gone.

If I capture a playing card held at a large angle to the mat, then that will be the training image, and Sprout will not recognize the playing card lying flat on the mat, because its ability to recognize bending and elastic deformations is limited.

Deformations of 2D objects such as bending and stretching

Sprout will recognize 2D objects that are deformed, for example by bending or stretching, to some extent, but if the deformations exceed some level, recognition will stop. Before recognition stops, tracking data can become unreliable (erratic and jumpy). This is a side effect of the reduction in the number of recognized features.

Unions of two objects

If Sprout is asked to recognize and track two objects, and during tracking the two objects are placed adjacent to one another, Sprout will recognize the individual objects. It has no concept of union.

Objects and backgrounds

If training occurs with an object on a background that has features, then during object tracking, Sprout does not distinguish between the object and the background. Sprout has been instructed to look for both object and background, so it might or might not recognize just the object.

On the other hand, if training occurs without a background (this is the case for an object in a captured moment), then Sprout will be able to recognize the object during object tracking, irrespective of whether it is on a background or not.
Other objects

Having other objects present during object tracking, either objects used for training or not, does not alter recognition of a specific object for which training was done, except in the event that these other objects obscure too much of the object of interest. Obscuring an object by placing other objects in front of it reduces Sprout’s ability to recognize an object.

Hands

Hands can play a role in the success of object recognition and tracking:

- **Hands during training:** If training uses a bitmap image created from a picture extracted from a captured moment, and the picture includes a person’s hands (someone was holding the object), then the object that Sprout is being asked to recognize and track is the intended object plus the hands. So, the object alone might not be recognized.

- **Hands during tracking:** If Sprout was trained to recognize an object, then it will recognize and track the object when the object is placed on the touch mat. If the object is held above the touch mat, then Sprout will recognize and track the object, provided that the hands and fingers of the person holding the object do not obscure too much of the object.

Multiple copies of an object

Sprout really shouldn’t be trained with multiple identical copies of images of the same object. But it does reasonable things if it is. Here are some edge cases:

- **Multiple copies of the same image that are given the same names during training:** If training uses multiple copies of the same image of an object, and the objects are given the same names, then when Sprout recognizes the object in video frames, the event that is raised indicates the presence of multiple objects in the same location with the single name.

- **Multiple copies of the same image that are given different names during training:** If training uses multiple copies of the same image of an object, and the objects are given different names, then when Sprout recognizes the object in video frames, the event that is raised indicates the presence of multiple objects in the same location with different names.

Wholes and parts

If Sprout is asked to recognize and track two objects, and one of the objects is a significant part of the other (for example, the objects are a post card and half of an identical post card), then during tracking the Sprout will recognize the whole as the whole and as the half, and the half as the half and as the whole.

Different objects that are given the same name

Sprout will consider these to be different objects. If a user names a fork, a spoon, and a knife “kitchen utensil,” then Sprout will report the name “kitchen utensil” when it sees 2D images of these objects.
False negatives and false positives

Given everything written above, you realize that Sprout can fail to recognize an object, for example, if it has been rotated too much relative to the touch mat, or if too much of the object is obscured. This is a false negative. The object is there, but Sprout does not recognize it.

Sprout can also mistake an object that is not present for one that is. This is a false positive. An example would be that I train Sprout with a photograph of Uncle Bob, taken in my garden at 3:00 PM on Saturday, June 21st. Bob is wearing tan pants and a blue shirt, and he is resting his left hand on a sundial.

Then, I show Sprout a photo of Uncle Frank (I did not train Sprout with this photograph), and Sprout says “Bob.” What happened? Well, Bob and Frank are twins, they dress identically, Frank also rested his hand on the sundial, and I took that photo two minutes after I took the photo of Bob. It is no wonder that Sprout is confused. I have trouble telling my uncles apart.

Controlling object tracking

After an instance of the tracking handler has been created, which includes training, you can start and stop object tracking, and can determine whether object tracking is in progress, using methods (and properties for WPF) that are declared on the interface IPCObjectTrackingHandler.

Starting object tracking

Start object tracking by calling the method Start | StartAsync.

Stopping object tracking

Stop object tracking by calling the method Stop | StopAsync. The object-tracking handler is not disposed of and it remains trained to recognize the set of objects that it was trained with. Starting object tracking again will track the same objects.

Determining whether object tracking is in progress

Determine whether tracking is in progress by getting the property (for WPF) or calling the method (for C++) IsTrackingInProgress.

Object-tracking events

When tracking is in progress, an instance of the tracking handler raises Updated | TrackUpdated events that contain event data about the tracked objects, in an instance of the interface IPCTrackEventArgs (for C++) or class PcTrackEventArgs (for WPF).

The event data contains an enumeration of PcTrackedObject instances, each of which contains the following data for each tracked object:

- An identifier of the object
• A bounding box for the object
• The rotation in radians of the bounding box for the object

An app would subscribe to the `TrackUpdated` events and perform actions based on the received event data.

**App logic (C++)**

This is the app logic for C++:

1. **Create a platform link:** To use Sprout Platform functionality, you need a platform link.
2. **Obtain images of objects:** Programmatically permit the user to obtain images of one or more objects to recognize and track, and to assign names to the objects. The images could be obtained from one or more captured moments, in which case you would need to extract pictures from the moment, and then create images from the instances of the class `IPcPicture`. Or the images could be obtained from other sources, such as the Web.
3. **Convert images to bitmaps:** If necessary, convert images to bitmaps (`PcImage` in C++).
4. **Create a set of training images:** Place the bitmap images and associated names in instances of the class `PcTrainingImage`, which is used to provide training images to the object-tracking handler.
5. **Create an instance of the object-tracking handler to track the objects:** Use the method `CreateObjectTrackingHandler` that is declared on the interface `IPcLink` to create an instance of the object-tracking handler `IPcObjectTrackingHandler`. When creating the object-tracking handler, you provide it with the set of training images.
6. **Create an object-tracking observer:** Create an object-tracking observer to receive object-tracking events by creating a concrete implementation of the class `IPcObjectTrackingHandler`. If needed, you can create multiple object-tracking observers.
7. **Add the object-tracking observer to the list of object-tracking observers:** Add the object-tracking observer to a list of object-tracking observers that the object-tracking handler knows about.
8. **Turn on object tracking:** When desired, turn on object tracking. Use the method `Start` that is declared on the interface `IPcObjectTrackingHandler`.
9. **Receive object-tracking events:** When object tracking is on, the object-tracking handler generates events when it recognizes in a video frame any of the tracked objects. The object-tracking observer receives the events.
10. **Extract event payloads and handle object-tracking events:** By extracting the event payloads, information can be obtained about the names of the objects that are recognized and about their physical locations relative to the touch mat. An app can use this information, for example, by presenting it to the user.
11. **Turn off object tracking:** When object tracking is no longer needed, turn off object tracking. Use the method `Stop` that is declared on the interface `IPcObjectTrackingHandler`.

**App logic (WPF)**

This is the app logic for WPF:

1. **Create a platform link:** To use Sprout Platform functionality, you need a platform link.

2. **Obtain images of objects:** Programmatically permit the user to obtain images of one or more objects to recognize and track, and to assign names to the objects. The images could be obtained from one or more captured moments, in which case you would need to extract pictures from the moment, and then create images from the instances of the class `IPcPicture`. Or the images could be obtained from other sources, such as the Web.

3. **Convert images to bitmaps:** If necessary, convert images to bitmaps (`BitmapSource` in WPF).

4. **Create a set of training images:** Place the bitmap images and associated names in instances of the class `PcTrainingImage`, which is used to provide training images to the object-tracking handler.

5. **Create an instance of the object-tracking handler to track the objects:** Use the method `CreateObjectTrackingHandler` that is declared on the interface `IPcLink` to create an instance of the object-tracking handler `IPcObjectTrackingHandler`. When creating the object-tracking handler, you provide it with the set of training images.

6. **Subscribe to object-tracking events:** Subscribe to the events that the object-tracking handler will generate when it recognizes an object in a video frame.

7. **Turn on object tracking:** When desired, turn on object tracking. Use the method `StartAsync` that is declared on the interface `IPcObjectTrackingHandler`.

8. **Extract event payloads and handle object-tracking events:** When object tracking is on, the object-tracking handler generates events when it recognizes in a video frame any of the tracked objects. The app has subscribed to the events. By extracting the event payloads, information can be obtained about the names of the objects that are recognized and about their physical locations relative to the touch mat. An app can use this information, for example, by presenting it to the user.

9. **Turn off object tracking:** When object tracking is no longer needed, turn off object tracking. Use the method `StopAsync` that is declared on the interface `IPcObjectTrackingHandler`.

10. **Dispose of the object-tracking handler:** When the object-tracking handler is no longer needed, dispose of it.

11. **Dispose of the platform link:** When the platform link is no longer needed, dispose of it.
A sample use case (WPF)

Following is an example scenario where this functionality is used (given an app that uses the platform functionality and that implements a suitable user interface):

1. A user has two post cards (one from New York and one from San Francisco) and two printed photographs
2. A user places three playing cards on the touch mat and captures a moment.
3. Pictures of the three cards (instances of the interface IPcPicture) can be extracted from the moment.
4. The pictures of the three cards are used as training images when creating an instance of the tracking handler IPcObjectTrackingHandler. This tracking handler instance recognizes and tracks these three cards.
5. The app subscribes to the TrackUpdated event for this instance of the tracking handler.
6. When any of the three cards are placed on the touch mat, or are held above the touch mat, the cards are recognized as objects to be tracked and are tracked. During tracking, position data for the objects is provided to the app through event arguments.
7. When other cards are placed on the touch mat, or are held above the touch mat, they are not recognized by this instance of the tracking handler as objects to be tracked and they are not tracked. One or more other tracking handler instances might have been created to track the other cards, but only a single tracking handler instance can be running at one time.
Inter-app and intra-app communication

Sprout apps can communicate with one another by exchanging messages as event arguments. The same mechanism can be used for intra-app communication.

To do so, any app that wants to communicate with other apps creates a communication handler that provides a named destination for messages and the ability to control the communication channel.

One example of when this might be useful is if separate apps run on the monitor and the touch mat, and the apps need to communicate with each other.

Overview

The Sprout Platform includes the ability to send and receive messages between Sprout apps, as well as within a single app. The following diagram illustrates communication. Numbers in the diagram correspond to explanations that follow the diagram.
In the diagram above:

1. A sprout app uses the method `CreateCommunicationHandler` to create an instance of the communication handler interface `IPcCommunicationHandler`. The method call specifies an identifier for the communication handler (this is the destination when sending messages) and a friendly name for the communication handler. A C++ app will also need to call the method `CreateCommunicationObserver` to create a communication observer, and to add the communication observer to the list of communication observers for the communication handler. In this example, the first Sprout app creates communication handlers with identifiers A and B.

2. A second Sprout app also creates a communication handler, with identifier C.

3. Using the method `Send` or `SendAsync`, the communication handler instances can exchange messages. Message exchanges between destinations A and B, and between A and C, are shown.

**Communication handler and observer (C++)**

To communicate with a different Sprout app, a Sprout app creates an instance of a communication handler, `IPcCommunicationHandler`, and at least one (and probably only one) communication observer, `IPcCommunicationObserver`. A Sprout app can use the communication handler to:

- Identify the communication channel that it will use to send messages, and that other apps will refer to as the destination when sending messages.
- Send messages to other Sprout apps
- Start and stop the communication channel
- Add communication observers to and remove them from the list of communication observers

The communication observer provides a virtual method for receiving messages.

An app developer must create a concrete implementation of the class `IPcCommunicationObserver` to receive events that contain messages. An application can subscribe to the events by implementing this class, and then using the method `AddCommunicationObserver` that is declared on the interface `IPcCommunicationHandler` to provide an instance of the class `IPcCommunicationObserver` to the interface `IPcCommunicationHandler`.

The `Updated` method, which is declared on the interface `IPcCommunicationObserver`, delivers the event payload (the message and related data in an instance of the interface `IPcMessageEventArgs`).
Communication handler (WPF)

To communicate with a different Sprout app, a Sprout app creates an instance of a communication handler, `IPcCommunicationHandler`. A Sprout app can use the communication handler to:

- Identify the communication channel that it will use to send messages, and that other apps will refer to as the destination when sending messages.
- Receive messages from other Sprout apps by triggering a `MessageReceived` event when a new message is available. To receive messages, a Sprout app subscribes to the `MessageReceived` event. Each `MessageReceived` event contains an instance of the class `PcMessageEventArgs`, which contains the message and related data.
- Send messages to other Sprout apps
- Start and stop the communication channel
Part 6:

Troubleshooting
Troubleshooting the Sprout Platform and SDK

This section will help you troubleshoot problems with the Sprout Platform and SDK.

If a problem persists, and especially if it recurs under reproducible circumstances, we want to hear the details. For information about contacting us for support or to report issues, see “Support.”

Sprout Platform diagnostics log files

The Sprout Platform has two diagnostics log files, one for the Sprout Service (one file) and one for the Sprout Process (per-user files).

The diagnostics log files for the Sprout Platform are located here:

Sprout Service log file:
C:\ProgramData\Hewlett-Packard\Sprout\SDK\SproutService.log

Sprout Process log file (per user):
C:\Users\$UserName$\AppData\Local\Hewlett-Packard\Sprout\SDK\SproutProcess.log

The diagnostics log files are intended for diagnosing major events, such as Sprout Service or Sprout Process crashes. The log files contain entries for the major events and their timings. If you encounter a Sprout Service or Sprout Process crash, attach these log files when reporting an issue. For the procedure for reporting issues, see “Reporting issues.”

The maximum size of a log file is 5 MB. When a diagnostics log file reaches that size, the contents of filename.log are copied to a log archive file (filename1.log, filename2.log, filename3.log, or filename4.log) and a new diagnostics log file is created. The maximum number of log and log archive files is five, counting filename.log. When the fifth file reaches 5 MB, the first log archive file is emptied and reused, and then the second, and so forth.
Part 7:

Appendices

This section contains information about interfaces and classes in the bindings, a list of open source software that is used in the Sprout Platform, and a glossary.
# Interfaces and classes

Following are lists of the interfaces and classes in the C++ and WPF bindings. More information is available in the *API Reference* websites.

## Overview of interfaces and classes in the C++ binding

Following are interfaces and classes in the C++ binding. For more information about each interface or class, including the methods that they contain, see the *API Reference*.

### Interfaces in the namespace `hp::pc::native`  

Following are interfaces (pure virtual classes) in the namespace `hp::pc::native`:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IPcCamera</code></td>
<td>The interface <em>IPcCamera</em> provides information about a camera on the system.</td>
</tr>
<tr>
<td><code>IPcClassification</code></td>
<td>The interface <em>IPcClassification</em> provides tag information that classifies the contents of a moment.</td>
</tr>
<tr>
<td><code>IPcClassificationSpecification</code></td>
<td>The interface <em>IPcClassificationSpecification</em> provides the classification tags that are available in the Sprout Platform.</td>
</tr>
<tr>
<td><code>IPcCommunicationHandler</code></td>
<td>The interface <em>IPcCommunicationHandler</em> provides the functionality for a communication channel.</td>
</tr>
<tr>
<td><code>IPcCommunicationObserver</code></td>
<td>The interface <em>IPcCommunicationObserver</em> provides the functionality for a communication observer. It specifies a contract to the app developer and provides a virtual method for receiving messages.</td>
</tr>
<tr>
<td><code>IPcDisplayabilityChangeEventArgs</code></td>
<td>The interface <em>IPcDisplayabilityChangeEventArgs</em> is a class for receiving information about a change in the displayability state of a registered window as arguments in an event.</td>
</tr>
<tr>
<td>Interface</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IPCDisplayabilityObserver</td>
<td>The interface <em>IPcDisplayabilityObserver</em> provides the functionality for an observer of displayability-change events. It specifies a contract to the app developer and provides a virtual method for receiving the displayability-change events.</td>
</tr>
<tr>
<td>IPCIntent</td>
<td>The interface <em>IPcIntent</em> represents an optional intent that can be used when calling some SDK methods, in order to obtain a specific implementation of the Sprout Platform functionality that the methods provide access to.</td>
</tr>
<tr>
<td>IPCLink</td>
<td>The interface <em>IPcLink</em> provides access to all functionality of the Sprout Platform.</td>
</tr>
<tr>
<td>IPCMessageEventArgs</td>
<td>The interface <em>IPcMessageEventArgs</em> represents a data structure that contains a message and related data.</td>
</tr>
<tr>
<td>IPCMoment</td>
<td>The interface <em>IPcMoment</em> contains a moment of life captured by Sprout.</td>
</tr>
<tr>
<td>IPCObjectTrackingHandler</td>
<td>The interface <em>IPcObjectTrackingHandler</em> provides a handler that allows an app to recognize and track 2D objects placed on or held above the touch mat, based on prior training to recognize images of the objects.</td>
</tr>
<tr>
<td>IPCObjectTrackingObserver</td>
<td>The class <em>IPcObjectTrackingObserver</em> provides the functionality for an object-tracking observer. It specifies a contract to the app developer and provides a virtual method for receiving object-tracking events.</td>
</tr>
<tr>
<td>IPCOutline</td>
<td>The interface <em>IPcOutline</em> provides read-only access to an outline, some properties of the outline, and to children of the outline.</td>
</tr>
<tr>
<td>IPCPicture</td>
<td>The interface <em>IPcPicture</em> provides read-only access to a picture, some properties of the picture, and to children of the picture.</td>
</tr>
<tr>
<td>IPCScreen</td>
<td>The interface <em>IPcScreen</em> provides information about a screen on the system.</td>
</tr>
<tr>
<td>Interface</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IPcSpecification</td>
<td>The interface <strong>IPcSpecification</strong> provides access to other interfaces used for getting information about the Sprout immersive computer and Sprout Platform, as well as access to interfaces that define options for the Sprout immersive computer and Sprout Platform.</td>
</tr>
<tr>
<td>IPcTag</td>
<td>The interface <strong>IPcTag</strong> represents a classification tag.</td>
</tr>
<tr>
<td>IPcText</td>
<td>The interface <strong>IPcText</strong> provides read-only access to the text in a moment, some properties of the text, and to children of the text.</td>
</tr>
<tr>
<td>IPcTouch</td>
<td>The interface <strong>IPcTouch</strong> provides access to all of the layers in the stack of touch layers that the Sprout Platform uses for managing touch events on the touch mat.</td>
</tr>
<tr>
<td>IPcTouchEventArgs</td>
<td>The <strong>IPcTouchEventArgs</strong> class defines arguments for the Sprout Platform event that is generated by a touch layer when a user touches the touch-handling area of an enabled touch layer (or moves or raises a finger or stylus), and when the layer state of a touch layer changes.</td>
</tr>
<tr>
<td>IPcTouchLayer</td>
<td>The interface <strong>IPcTouchLayer</strong> provides access to a specific layer in the stack of touch layers that the Sprout Platform uses for managing touch events on the touch mat.</td>
</tr>
<tr>
<td>IPcTouchLayerObserver</td>
<td>The interface <strong>IPcTouchLayerObserver</strong> provides the functionality for a touch-layer observer. It specifies a contract to the app developer and provides a virtual method for receiving touch events.</td>
</tr>
<tr>
<td>IPcTrackEventArgs</td>
<td>The interface <strong>IPcTrackEventArgs</strong> defines arguments for the event that is triggered when an object is recognized in a video frame by the object-tracking handler.</td>
</tr>
<tr>
<td>IPcWindowRegistration</td>
<td>The interface <strong>IPcWindowRegistration</strong> provides a window handler for an application window that needs to appear on the mat screen and that needs access to the touch mat.</td>
</tr>
</tbody>
</table>
## Classes in the namespace `hp::pc::native`

Following are other classes in the namespace `hp::pc::native`:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConversionUtilities</td>
<td>The class <code>ConversionUtilities</code> provides a set of helper methods for converting between the physical and screen coordinate systems.</td>
</tr>
<tr>
<td>HPPC</td>
<td>The class <code>HPPC</code> provides a factory for creating platform links.</td>
</tr>
<tr>
<td>PcGuid</td>
<td>The class <code>PcGuid</code> represents a globally unique identifier (GUID).</td>
</tr>
<tr>
<td>PcImage</td>
<td>The class <code>PcImage</code> contains constructors for creating bitmap images and methods for obtaining information about bitmap images.</td>
</tr>
<tr>
<td>PcPhysicalPoint</td>
<td>The class <code>PcPhysicalPoint</code> represents a 2D physical point, that is, a 2D representation of a physical point in the real world.</td>
</tr>
<tr>
<td>PcPhysicalPoint3D</td>
<td>The class <code>PcPhysicalPoint3D</code> represents a 3D physical point, that is, a 3D representation of a physical point in the real world.</td>
</tr>
<tr>
<td>PcPhysicalQuadrilateral</td>
<td>The class <code>PcPhysicalQuadrilateral</code> represents a 2D physical quadrilateral.</td>
</tr>
<tr>
<td>PcPhysicalRectangle</td>
<td>The class <code>PcPhysicalRectangle</code> is used to construct a physical rectangle, that is, a rectangle in physical coordinates (in millimeters).</td>
</tr>
<tr>
<td>PcPhysicalSize</td>
<td>The class <code>PcPhysicalSize</code> represents the size of a 2D object in physical coordinates (in millimeters).</td>
</tr>
<tr>
<td>PcPhysicalSize3D</td>
<td>The class <code>PcPhysicalSize3D</code> represents the size of a physical object in three dimensions (with physical coordinates in millimeters).</td>
</tr>
<tr>
<td>PcPixelDensity</td>
<td>The class <code>PcPixelDensity</code> represents a pixel density, that is, the number of pixels per physical unit of measurement (pixels per millimeter).</td>
</tr>
<tr>
<td>PcPixelPoint</td>
<td>The class <code>PcPixelPoint</code> represents a point in 2D pixel coordinates as an integer pair (x, y).</td>
</tr>
</tbody>
</table>
### Class

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PcPixelPoint3D</td>
<td>The class <strong>PcPixelPoint3D</strong> represents a point in 3D pixel coordinates as an integer triplet ( (x, y, z) ).</td>
</tr>
<tr>
<td>PcPixelRectangle</td>
<td>The class <strong>PixelRectangle</strong> represents a pixel rectangle.</td>
</tr>
<tr>
<td>PcPixelSize</td>
<td>The class <strong>PcPixelSize</strong> represents the size of a 2D object in pixel coordinates.</td>
</tr>
<tr>
<td>PcResult</td>
<td>The class <strong>PcResult</strong> is a template class for wrapping the result values and result codes that all SDK methods return.</td>
</tr>
<tr>
<td>PcTrackedObject</td>
<td>The class <strong>PcTrackedObject</strong> encapsulates information about an object that has been recognized and is being tracked by the object-tracking handler.</td>
</tr>
<tr>
<td>PcTrainingImage</td>
<td>The class <strong>PcTrainingImage</strong> represents an association between an image of an object and the name of the object in the image.</td>
</tr>
</tbody>
</table>

### Interfaces in the namespace \texttt{hp::pc::native::specification}

Following are interfaces (pure virtual classes) in the namespace \texttt{hp::pc::native::specification}:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPCCameraSpecification</td>
<td>The interface <strong>IPCameraSpecification</strong> provides information about the cameras that are supported by the Sprout Platform.</td>
</tr>
<tr>
<td>IPCOutlineExtractOption</td>
<td>The interface <strong>IPOutlineExtractOption</strong> represents an extract option for outlines.</td>
</tr>
<tr>
<td>IPCOutlineExtractSpecification</td>
<td>The interface <strong>IPOutlineExtractSpecification</strong> provides access to extract options, and to functions related to extract options, for outlines.</td>
</tr>
<tr>
<td>IPCOutlineRefineOption</td>
<td>The interface <strong>IPOutlineRefineOption</strong> represents a refine option for outlines.</td>
</tr>
<tr>
<td>Interface</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IPcOutlineRefineSpecification</td>
<td>The interface IPcOutlineRefineSpecification provides access to refine options, for outlines.</td>
</tr>
<tr>
<td>IPcOutlineSpecification</td>
<td>The interface IPcOutlineSpecification provides access to options for outlines.</td>
</tr>
<tr>
<td>IPcPictureExtractOption</td>
<td>The interface IPcPictureExtractOption represents an extract option for pictures.</td>
</tr>
<tr>
<td>IPcPictureExtractSpecification</td>
<td>The interface IPcPictureExtractSpecification provides access to extract options, for pictures.</td>
</tr>
<tr>
<td>IPcPictureRefineOption</td>
<td>The interface IPcPictureRefineOption represents a refine option for pictures.</td>
</tr>
<tr>
<td>IPcPictureRefineSpecification</td>
<td>The interface IPcPictureRefineSpecification provides access to refine options, for pictures.</td>
</tr>
<tr>
<td>IPcPictureSpecification</td>
<td>The interface IPcPictureSpecification provides access to options for pictures.</td>
</tr>
<tr>
<td>IPcScreenSpecification</td>
<td>The interface IPcScreenSpecification provides information about the screens that are supported by the Sprout Platform.</td>
</tr>
<tr>
<td>IPcTextExtractOption</td>
<td>The interface IPcTextExtractOption represents an extract option for text.</td>
</tr>
<tr>
<td>IPcTextExtractSpecification</td>
<td>The interface IPcTextExtractSpecification provides access to extract options, for text.</td>
</tr>
<tr>
<td>IPcTextLanguageOption</td>
<td>The interface IPcTextLanguageOption represents a language option for text.</td>
</tr>
<tr>
<td>IPcTextLanguageSpecification</td>
<td>The interface IPcTextLanguageSpecification provides access to language options, for text.</td>
</tr>
</tbody>
</table>
### Interface Description

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPCTextRefineOption</td>
<td>The interface <strong>IPcTextRefineOption</strong> represents a refine option for text.</td>
</tr>
<tr>
<td>IPCTextRefineSpecification</td>
<td>The interface <strong>IPcTextRefineSpecification</strong> provides access to refine options, and to functions related to refine options, for text.</td>
</tr>
<tr>
<td>IPCTextSpecification</td>
<td>The interface <strong>IPcTextSpecification</strong> provides access to options for text.</td>
</tr>
<tr>
<td>IPCTouchLayerOption</td>
<td>The interface <strong>IPcTouchLayerOption</strong> represents a supported touch layer type when using the interface IPCTouch.</td>
</tr>
<tr>
<td>IPCTouchSpecification</td>
<td>The interface <strong>IPcTouchSpecification</strong> provides information about the touch layers that are supported by the Sprout Platform.</td>
</tr>
<tr>
<td>IPCVersionSpecification</td>
<td>The interface <strong>IPcVersionSpecification</strong> provides information about the version of the Sprout Platform and of the binding used by the app.</td>
</tr>
</tbody>
</table>

### Interfaces and classes that are not used yet

Following are interfaces and classes that are not used yet:

- IPCCameraSpecification
- IPCCamera
- IPCOutlineRefineSpecification
- IPCPictureRefineSpecification
- IPCTextSpecification
- IPCTextExtractSpecification
- IPCTextLanguageSpecification
- IPCTextRefineSpecification
- IPCOutlineRefineOption
- IPCPictureRefineOption
- IPCTextSpecification
- IPCTextExtractOption
- IPCTextLanguageOption
- IPCTextRefineOption
- PcPhysicalPoint3D
- PcPhysicalSize3D
- PcPixelPoint3D

### Overview of interfaces and classes in the WPF binding

Following are interfaces and classes in the WPF binding. For more information about each interface or class, including the methods that they contain, see the API Reference.

### Interfaces in the namespace HP.PC.Presentation

Following are interfaces in the namespace **HP.PC.Presentation**:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
</table>

*Sprout Developer Guide*
<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPcCamera</td>
<td>The interface IPcCamera provides information about a camera on the system.</td>
</tr>
<tr>
<td>IPcClassification</td>
<td>The interface IPcClassification provides tag information that classifies the contents of a moment.</td>
</tr>
<tr>
<td>IPcCommunicationHandler</td>
<td>The interface IPcCommunicationHandler provides the functionality for a communication channel.</td>
</tr>
<tr>
<td>IPcIntent</td>
<td>The interface IPcIntent represents an optional intent that can be used when calling some SDK methods, in order to obtain a specific implementation of the Sprout Platform functionality that the methods provide access to.</td>
</tr>
<tr>
<td>IPcLink</td>
<td>The interface IPcLink provides access to all functionality of the Sprout Platform.</td>
</tr>
<tr>
<td>IPcMoment</td>
<td>The interface IPcMoment contains a moment of life captured by Sprout.</td>
</tr>
<tr>
<td>IPcObjectTrackingHandler</td>
<td>The interface IPcObjectTrackingHandler provides a handler that allows an app to recognize and track 2D objects placed on or held above the touch mat, based on prior training to recognize images of the objects.</td>
</tr>
<tr>
<td>IPcOutline</td>
<td>The interface IPcOutline provides read-only access to an outline, some properties of the outline, and to children of the outline.</td>
</tr>
<tr>
<td>IPcPicture</td>
<td>The interface IPcPicture provides read-only access to a picture, some properties of the picture, and to children of the picture.</td>
</tr>
<tr>
<td>IPcScreen</td>
<td>The interface IPcScreen provides information about a screen on the system.</td>
</tr>
<tr>
<td>IPcSpecification</td>
<td>The interface IPcSpecification provides access to other interfaces used for getting information about the Sprout immersive computer and Sprout Platform, as well as access to interfaces that define options for the Sprout immersive computer and Sprout Platform.</td>
</tr>
<tr>
<td>IPcTag</td>
<td>The interface IPcTag represents a classification tag.</td>
</tr>
<tr>
<td>IPcText</td>
<td>The interface IPcText provides read-only access to the text in a moment, some properties of the text, and to children of the text.</td>
</tr>
</tbody>
</table>
### Interface Description

**IPcTouch**
The interface **IPcTouch** provides access to all of the layers in the stack of touch layers that the Sprout Platform uses for managing touch events on the touch mat.

**IPcTouchLayer**
The interface **IPcTouchLayer** provides access to a specific layer in the stack of touch layers that the Sprout Platform uses for managing touch events on the touch mat.

**IPcWindowRegistration**
The interface **IPcWindowRegistration** provides a window handler for an application window that needs to appear on the mat screen and that needs access to the touch mat.

### Classes in the namespace `HP.PC.Presentation`

Following are classes in the namespace `HP.PC.Presentation`:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConversionUtilities</td>
<td>The class <strong>ConversionUtilities</strong> provides a set of helper methods for converting between the physical and screen coordinate systems.</td>
</tr>
<tr>
<td>HPPC</td>
<td>The class <strong>HPPC</strong> provides a factory for creating platform links.</td>
</tr>
<tr>
<td>PcDisplayabilityChangeEventArgs</td>
<td>The class <strong>PcDisplayabilityChangeEventArgs</strong> is a class for sending information about a change in the displayability state of a registered window as arguments in an event.</td>
</tr>
<tr>
<td>PcException</td>
<td>The class <strong>PcException</strong> is the class for base exceptions thrown by the Sprout Platform.</td>
</tr>
<tr>
<td>PcMessageEventArgs</td>
<td>The class <strong>PcMessageEventArgs</strong> is a used to construct a data structure that contains a message and related data.</td>
</tr>
<tr>
<td>PcPhysicalPoint</td>
<td>The class <strong>PcPhysicalPoint</strong> represents a 2D physical point, that is, a 2D representation of a physical point in the real world.</td>
</tr>
<tr>
<td>PcPhysicalPoint3D</td>
<td>The class <strong>PcPhysicalPoint3D</strong> represents a 3D physical point, that is, a 3D representation of a physical point in the real world.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PcPhysicalQuadrilateral</td>
<td>The class <code>PcPhysicalQuadrilateral</code> represents a 2D physical quadrilateral.</td>
</tr>
<tr>
<td>PcPhysicalRectangle</td>
<td>The class <code>PcPhysicalRectangle</code> represents a physical rectangle, that is, a rectangle in physical coordinates (in millimeters).</td>
</tr>
<tr>
<td>PcPhysicalSize</td>
<td>The class <code>PcPhysicalSize</code> represents the size of a 2D object in physical coordinates (in millimeters).</td>
</tr>
<tr>
<td>PcPhysicalSize3D</td>
<td>The class <code>PcPhysicalSize3D</code> represents the size of a physical object in three dimensions (with physical coordinates in millimeters).</td>
</tr>
<tr>
<td>PcPixelDensity</td>
<td>The class <code>PcPixelDensity</code> represents a pixel density, that is, the number of pixels per physical unit of measurement (pixels per millimeter).</td>
</tr>
<tr>
<td>PcPixelPoint3D</td>
<td>The class <code>PcPixelPoint3D</code> represents a point in 3D pixel coordinates as an integer triplet (x, y, z).</td>
</tr>
<tr>
<td>PcTouchDownEventArgs</td>
<td>The class <code>PcTouchDownEventArgs</code> defines arguments for the Sprout Platform event that is generated by a touch-input layer when a user touches the touch-handling area of the enabled layer.</td>
</tr>
<tr>
<td>PcTouchLayerStateEventArgs</td>
<td>The class <code>PcTouchLayerStateEventArgs</code> defines arguments for the Sprout Platform event that is triggered when a touch layer is either enabled or disabled.</td>
</tr>
<tr>
<td>PcTouchMoveEventArgs</td>
<td>The class <code>PcTouchMoveEventArgs</code> defines arguments for the Sprout Platform event that is generated by a touch-input layer when the touch point moves (when a user moves his or her finger while still touching the mat) on the touch-handling area of the enabled layer.</td>
</tr>
<tr>
<td>PcTouchUpEventArgs</td>
<td>The class <code>PcTouchUpEventArgs</code> defines arguments for the Sprout Platform event that is generated by a touch-input layer when touch is released (when a user raises his or her finger off of the mat) from the touch-handling area of the enabled layer.</td>
</tr>
</tbody>
</table>
### Class

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PcTrackedObject</td>
<td>The class <code>PcTrackedObject</code> encapsulates information about an object that has been recognized and is being tracked by the object-tracking handler.</td>
</tr>
<tr>
<td>PcTrackEventArgs</td>
<td>The class <code>PcTrackEventArgs</code> defines arguments for the event that is triggered when an object is recognized in a video frame by the object-tracking handler.</td>
</tr>
<tr>
<td>PcTrainingImage</td>
<td>The class <code>PcTrainingImage</code> represents an association between an image of an object and the name of the object in the image.</td>
</tr>
</tbody>
</table>

### Interfaces in the namespace `HP.PC.Presentation.Specification`

Following are interfaces in the namespace `HP.PC.Presentation.Specification`:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPcCameraSpecification</td>
<td>The interface <code>IPcCameraSpecification</code> provides information about the cameras that are supported by the Sprout Platform.</td>
</tr>
<tr>
<td>IPcClassificationSpecification</td>
<td>The interface <code>IPcClassificationSpecification</code> provides the classification tags that are supported by the Sprout Platform.</td>
</tr>
<tr>
<td>IPcOutlineExtractOption</td>
<td>The interface <code>IPcOutlineExtractOption</code> represents an option for outlines.</td>
</tr>
<tr>
<td>IPcOutlineExtractSpecification</td>
<td>The interface <code>IPcOutlineExtractSpecification</code> provides access to extract options, and to functions related to extract options, for outlines.</td>
</tr>
<tr>
<td>IPcOutlineRefineOption</td>
<td>The interface <code>IPcOutlineRefineOption</code> represents a refine option for outlines.</td>
</tr>
<tr>
<td>IPcOutlineRefineSpecification</td>
<td>The interface <code>IPcOutlineRefineSpecification</code> provides access to refine options, and to functions related to refine options, for outlines.</td>
</tr>
</tbody>
</table>
### Interface | Description
--- | ---
**IPcOutlineSpecification** | The interface `IPcOutlineSpecification` provides access to options for outlines.

**IPcPictureExtractOption** | The interface `IPcPictureExtractOption` represents an extract option for pictures.

**IPcPictureExtractSpecification** | The interface `IPcPictureExtractSpecification` provides access to extract options, and to functions related to extract options, for pictures.

**IPcPictureRefineOption** | The interface `IPcPictureRefineOption` represents a refine option for pictures.

**IPcPictureRefineSpecification** | The interface `IPcPictureRefineSpecification` provides access to refine options, and to functions related to refine options, for pictures.

**IPcPictureSpecification** | The interface `IPcPictureSpecification` provides access to options for pictures.

**IPcScreenSpecification** | The interface `IPcScreenSpecification` provides information about the screens that are supported by the Sprout Platform.

**IPcTextExtractOption** | The interface `IPcTextExtractOption` represents an extract option for text.

**IPcTextExtractSpecification** | The interface `IPcTextExtractSpecification` provides access to extract options, and to functions related to extract options, for text.

**IPcTextLanguageOption** | The interface `IPcTextLanguageOption` represents a language option for text.

**IPcTextLanguageSpecification** | The interface `IPcTextLanguageSpecification` provides access to language options, and to functions related to language options, for text.

**IPcTextRefineOption** | The interface `IPcTextRefineOption` represents a refine option for text.

**IPcTextRefineSpecification** | The interface `IPcTextRefineSpecification` provides access to refine options, and to functions related to refine options, for text.
### Interface

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPCrTextSpecification</td>
<td>The interface <code>IPcTextSpecification</code> provides access to options for text.</td>
</tr>
<tr>
<td>IPCrTouchLayerOption</td>
<td>The interface <code>IPcTouchLayerOption</code> represents a supported touch layer type when using the interface <code>IPcTouch</code>.</td>
</tr>
<tr>
<td>IPCrTouchSpecification</td>
<td>The interface <code>IPcTouchSpecification</code> provides information about the touch layers that are available in the Sprout Platform.</td>
</tr>
<tr>
<td>IPCrVersionSpecification</td>
<td>The interface <code>IPcVersionSpecification</code> provides information about the version of the Sprout Platform and of the binding used by the app.</td>
</tr>
</tbody>
</table>

### Interfaces and classes that are not used yet

Following are interfaces and classes that are not used yet:

Open Source Software

The Sprout Platform contains some open source software.

Open source software used by the platform

Following is the open source software that is included in the Sprout Platform. In the case of the tools to generate the *API Reference*, it is the output that is included. The names in the first column are short-form program names. They are also the first parts of the license filenames.

**Note:** The Sprout Platform incorporates what it needs of these packages. The platform installer does not install the complete packages, and you don’t need to install them either. This information is for reference.

<table>
<thead>
<tr>
<th>Software and Version</th>
<th>Notes and License URLs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boost</strong> 1.55.0</td>
<td>General purpose C++ library <a href="http://www.boost.org/users/license.html">http://www.boost.org/users/license.html</a></td>
</tr>
<tr>
<td><strong>clrzmq</strong> 3.2.4</td>
<td>Common Language Runtime (CLR) binding for ØMQ <a href="https://github.com/zeromq/clrzmq/blob/master/LICENSE">https://github.com/zeromq/clrzmq/blob/master/LICENSE</a></td>
</tr>
<tr>
<td><strong>cppzmq</strong> 3.2.4</td>
<td>C++ binding for ØMQ <a href="https://github.com/zeromq/cppzmq/blob/master/LICENSE">https://github.com/zeromq/cppzmq/blob/master/LICENSE</a></td>
</tr>
<tr>
<td><strong>doxygen</strong> 1.8.6</td>
<td>Tool for generating the <em>API Reference</em> <a href="http://www.stack.nl/~dimitri/doxygen/index.html">http://www.stack.nl/~dimitri/doxygen/index.html</a></td>
</tr>
<tr>
<td><strong>flan</strong> 1.8.4</td>
<td>Library for fast nearest-neighbor searches <a href="http://www.cs.ubc.ca/research/flann/#download">http://www.cs.ubc.ca/research/flann/#download</a></td>
</tr>
<tr>
<td><strong>graphviz</strong> 2.36</td>
<td>Graph visualization software used by doxygen <a href="http://www.graphviz.org/Download.php">http://www.graphviz.org/Download.php</a></td>
</tr>
<tr>
<td>Software and Version</td>
<td>Notes and License URLs</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td><strong>leptonica</strong> 1.69</td>
<td>C library for image processing and analysis</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.leptonica.com/about-the-license.html">http://www.leptonica.com/about-the-license.html</a></td>
</tr>
<tr>
<td><strong>log4cplus</strong> 1.1.1</td>
<td>C++ logging API</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.apache.org/licenses/LICENSE-2.0.html">http://www.apache.org/licenses/LICENSE-2.0.html</a></td>
</tr>
<tr>
<td><strong>mhook</strong> 2.3</td>
<td>API hooking library</td>
</tr>
<tr>
<td></td>
<td>No URL is available. Refer to the file.</td>
</tr>
<tr>
<td><strong>opencv</strong> 2.4.8</td>
<td>Open Source Computer Vision: Computer vision algorithms and imaging hardware control</td>
</tr>
<tr>
<td></td>
<td><a href="http://opensource.org/licenses/BSD-2-Clause">http://opensource.org/licenses/BSD-2-Clause</a></td>
</tr>
<tr>
<td><strong>pcl</strong> 1.7.1</td>
<td>Point Cloud Library: Image processing algorithms</td>
</tr>
<tr>
<td></td>
<td><a href="https://github.com/PointCloudLibrary/pcl/blob/master/LICENSE.txt">https://github.com/PointCloudLibrary/pcl/blob/master/LICENSE.txt</a></td>
</tr>
<tr>
<td><strong>qhull</strong> 2012.1</td>
<td>Convex hull triangulation library</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.qhull.org/COPYING.txt">http://www.qhull.org/COPYING.txt</a></td>
</tr>
<tr>
<td><strong>tesseract</strong> 3.02.02</td>
<td>Tesseract Optical Character Recognition engine</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.apache.org/licenses/LICENSE-2.0.html">http://www.apache.org/licenses/LICENSE-2.0.html</a></td>
</tr>
<tr>
<td><strong>viennacl</strong> 1.5.1</td>
<td>Linear algebra library for large systems of equations</td>
</tr>
<tr>
<td></td>
<td><a href="http://viennacl.sourceforge.net/viennacl-about.html">http://viennacl.sourceforge.net/viennacl-about.html</a></td>
</tr>
<tr>
<td><strong>vlfeat</strong> 0.9.17</td>
<td>Library of computer vision algorithms</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.vlfeat.org/license.html">http://www.vlfeat.org/license.html</a></td>
</tr>
<tr>
<td><strong>wix</strong> 3.8</td>
<td>Installer creation tool</td>
</tr>
<tr>
<td></td>
<td><a href="http://wixtoolset.org/about/license/">http://wixtoolset.org/about/license/</a></td>
</tr>
<tr>
<td><strong>xerces</strong> 3.1.1</td>
<td>Xerces C++ XML parsing library</td>
</tr>
<tr>
<td><strong>zeromq</strong> 3.2.4</td>
<td>ØMQ (also known as ZeroMQ): A communications library</td>
</tr>
<tr>
<td></td>
<td><a href="http://zeromq.org/area:licensing">http://zeromq.org/area:licensing</a></td>
</tr>
</tbody>
</table>
License files for open source software

The platform installer places some license files for open source software that is used by the Sprout Platform in this directory. The Sprout Platform and SDK Developer Documentation website has a link to the directory.

C:\Program Files\Hewlett-Packard\Sprout\SDK\Open_Source_Licenses
Glossary

Following are terms used by Sprout and the Sprout Platform and SDK, as well as some relevant Microsoft Windows terms.

All-in-One PC: A personal computer in which all of the components are built into the monitor

API: See application programming interface (API)

app: In Windows terminology, a Modern app that runs in the WinRT environment, uses the Modern UI, and is launched from the Start screen. Other names for a Modern app are a Windows Store app, a tailored application, and a WinRT app. The other kind of application is a Desktop application, which can be launched from the Desktop. Also, app is just an abbreviation for application. When we need to distinguish, we refer to a Modern app or a Desktop application.

application: A software program that runs on a personal computer. Also see app.

application programming interface (API): Application code that other applications can use to interact with an application

assisted segmentation: Segmentation assistance that the user can provide after automatic segmentation for a captured moment and in place of automatic segmentation for a created moment, to refine the boundaries of captured items. Also see segmentation and automatic segmentation.

automatic segmentation: Segmentation that occurs automatically, without user assistance. Automatic segmentation is performed in some cases for a captured moment when an extract method is used to extract data from the moment. Whether automatic segmentation is performed depends on the extract option that is used in the extract method call. Automatic segmentation is not performed for a moment that was created from an image. Also see segmentation and assisted segmentation.

capture: To take a set of data from each of the Sprout cameras at a specific point in time. Sprout stores the camera data and associated data in a moment.

capture area: See work area.

Common Language Runtime (CLR): A virtual machine component of Microsoft’s .NET framework. It is responsible for running .NET programs.
contour: A set of points that consists of the boundary points for an outline

coordinate system: A system that uses sets of numbers (coordinates) to determine the positions of points and other geometric elements in space, and by reference the positions of real objects (physical coordinates, measured in millimeters) or of pixels (screen coordinates, measured in pixels).

The physical coordinate system has (0, 0) in the physical upper-left corner of the work area. The positive X axis access runs across the upper edge of the work area. The positive Y axis runs down the left edge of the work area. The work area of the touch mat measures 12 inches high by 16 inches wide, thus the height and width are 304.8 mm and 406.4 mm respectively.

The pixel coordinate system has (0, 0) in the center of the pixel in the upper-left corner of the mat screen. The positive X axis access runs across the centers of the upper row of pixels on the mat screen. The positive Y axis runs down the centers of the leftmost row of pixels on the mat screen.

Desktop: Screen in Windows 8 from which users can run Desktop applications

Desktop application: A Windows application that is started from the Desktop, as opposed to a Modern app which is started from the Start screen. Also see app.

Dual-display development system: Specific dual-display computer configurations that can be used for development of Sprout apps.

global assembly cache (GAC): A computer-wide code cache that stores .NET assemblies that are shared by several applications on the computer

high-resolution camera: The Sprout camera that takes high-resolution images

high-resolution image: An image taken with the high-resolution camera

mat screen: The Sprout touchscreen that consists of the touch mat and the projected display

Sprout Platform: The core Sprout software that provides access to the Sprout hardware (for example, cameras and the touch mat), Sprout software features (for example, to capture a moment). The Sprout Platform installation includes the language bindings that are logically a part of the Sprout SDK.

Sprout Process (Sprout Process): A process that provides a binding library for the Sprout SDK. There is one Sprout Process per user of the Sprout immersive computer or dual-display development system.

Sprout Service (Sprout Service): A service that provides access to Sprout hardware and features of the Sprout Platform. There is one Sprout Service per Sprout immersive computer or dual-display development system.
Sprout Software Development Kit (SDK): Software that exposes language bindings that apps can use to incorporate Sprout features. The language bindings provide access to the Sprout Platform. The Sprout SDK also includes compiled sample apps, saved capture data for use on dual-display development systems, and developer documentation.

**image**: A graphical representation of an object or objects. See also **picture**.

**image dimensions**: The dimensions of an image in pixels

**link**: See **platform link**.

**mat**: The touch mat. Usually, when we refer to the mat, we are referring to the work area of the touch mat. The mat screen is projected on the mat.

**Sprout app**: An app that uses the Sprout SDK to provide the user with Sprout features

**Sprout immersive computer**: All of Sprout, including the All-in-One PC part, the column in the back, HP Illuminator (the part at the top that contains the projector mirror, Sprout cameras, and desk lamp; and the HP Touch Mat.

**Modern app**: A Windows app that is started from the Start screen, as opposed to a Desktop application which is started from the Desktop. Also see **app**.

**moment**: A moment in time, captured by Sprout. A moment is like a snapshot with a camera, but with Sprout there are multiple cameras. To obtain useful data from the moment, for example pictures and outlines, you must extract the data using extract methods.

A moment consumes considerable memory resources. When your app is finished with a moment, dispose of the moment.

**monitor**: A computer display. For Sprout, the monitor is a touch monitor that displays the vertical screen.

**non-touch monitor**: A monitor that is not sensitive to touch

**notification area of the taskbar**: Area to the right side of the task bar that has small icons that indicate the statuses of hardware and software systems, and that can often be used to start applications and provide limited access to features.

**object**: A physical object that you can place on the Sprout mat screen in order to capture a moment. The object might be something like a vase or watch, but it can also be a printed photograph or a document such as a newspaper clipping. In this regard, we regard your cat as an object. Remember to say “hold still.”

**physical dimensions (or physical size)**: The width (length along the X axis) and width (length along the Y axis) of an object. The physical dimensions of an object are the same as the physical dimensions of the object’s bounding rectangle.
physical location: The location of an object specified as the location (X and Y coordinates) of the upper-left corner of the object’s bounding rectangle.

picture: A “picture” is often synonymous with “an image.” In some cases and especially for digital images, the word “picture” implies that the image was taken with a camera (it is a synonym for “photo”). We also think of “pictures” as being things we paint or draw, or images like those. The word “image” is the more general term, but it is also a bit more technical.

Sprout captures moments, not pictures. Moments contain data from the Sprout cameras. In Sprout, we refer to images of physical objects as “pictures.” To get pictures, you have to extract the pictures from a moment.

pixel: For an image, a pixel is the smallest area that can have an assigned color value. An image is an array of pixels. For a monitor or other display, a pixel is the smallest area of the display in which the color and brightness can be adjusted, and the smallest area of the display on which a single image pixel can be displayed.

pixel density: A measurement of how many pixels there are per unit area on a monitor or other display. In this case, the measurement is of the number of pixels per millimeter. The pixel density is measured along both the X (left-to-right) and Y (top-to-bottom for the monitor screen and further-away-to-closer for the mat screen) axes. If the pixels are not square, the horizontal and vertical pixel densities will differ.

pixel extent: The width (number of pixels along the X axis) and width (number of pixels along the Y axis) of an area on a screen on which an image is displayed. For a camera, pixel extent is the number of pixels (for each dimension) in the highest resolution (pixel extent) image that is produced by the camera.

pixel location: The X and Y pixel coordinates of the pixel of the uppermost and leftmost pixel in an area on a screen on which an image is displayed.

projector: A part of the Sprout hardware that projects the mat screen onto the touch mat (creating a touch screen on the touch mat)

point: A location on the touch mat described by its (X, Y) physical coordinates or by its (X,Y) pixel coordinates. For example, the location of a bounding rectangle is given by the (X, Y) physical coordinates of the upper-left corner of the rectangle.

rectangle: A rectangular area on the touch mat described by (X, Y, width, height) in either physical coordinates or pixel coordinates. For example, a bounding rectangle uses physical coordinates.

resolution: See pixel density and pixel extent.

screen: What you see and interact with on the Sprout monitor (the monitor screen) and on the touch mat (the mat screen)

SDK: See Sprout Software Development Kit (SDK).
**platform link:** A link that is established between a Sprout app and the Sprout Platform. The platform link provides the features of the Sprout Platform to the Sprout app.

**segmentation:** The process of analyzing the images and other data in a captured moment and of extracting images, contours, and bounding rectangles of the captured items present. Also see *automatic segmentation* and *assisted segmentation*.

**session:** When a platform link has been created between an application and the Sprout Service, the Sprout Service creates a session, which it uses to manage application state information data that is received from and sent to the app. You do not need to worry about the session. Just create the platform link, keep the platform link around for as long as you need information from it (usually for as long as your app needs Sprout features), and then dispose of the platform link.

**Sprout Platform:** Software that provides Sprout features.

**Sprout Process:** Per-user process that, along with the Sprout Service, provides the features of the Sprout Platform.

**Sprout Service:** Windows service that, along with the Sprout Process, provides the features of the Sprout Platform.

**Start screen:** Screen in Windows from which users run Modern apps.

**tailored application:** Another name for a Modern app. Also see *app*.

**taskbar:** Bar at the bottom of the Desktop screen. On the left side, it shows open programs and files and allows you to switch between them. On the right side is a notification area that communicates status information regarding programs and computer settings.

**touch monitor:** A monitor that is sensitive to touch so that it can be used as an input device. Also see *monitor*.

**touch mat:** A touch mat that is sensitive to touch so that it can be used as an input device. With Sprout, projection of the mat screen onto the touch mat creates a touch screen.

**touch screen:** A screen that is sensitive to touch so that it can be used as an input device. Also see *screen*.

**vertical screen:** The Sprout touchscreen on the monitor.

**window:** A rectangular area of the screen that contains a graphical user interface that displays output and usually allows input.

**Windows Presentation Framework (WPF):** A presentation system for building Windows client applications.

**Windows Runtime (WinRT):** A software layer that includes a set of APIs that you use to build Modern apps.
**Windows Store app**: Another name for a Modern app, a Windows app that is started from the Start screen. Also see `app`.

**WinRT app**: Another name for a Modern app. Also see `app`.

**work area**: The central area of the touch mat, where a Sprout user can place objects and capture moments.

**WPF**: See *Windows Presentation Framework (WPF)*.

**WpfCsSample**: A Sprout Sample App that illustrates SDK features. Commented source code is provided for the parts of the sample app that use the SDK. The source code can be viewed from within the sample app.

**XAML**: A declarative markup language created by Microsoft that is used for application programming.
Index

alpha channel
  assisted segmentation .............................................. 148
API
  references .................................................................. 11
applications
  approved .................................................................. 45
approved applications ............................................. 45
architecture ............................................................... 30
assisted segmentation .............................................. 26, 144
  alpha channel .......................................................... 148
automatic segmentation ............................................. 25
background
  removal ......................................................................... 26
binding
  C++ ........................................................................... 52
  WPF ............................................................................ 53
C++ binding
  interfaces and classes ................................................. 171
  calibration ...................................................................... 41
camera information
  obtaining from specification ...................................... 91
capture data
  saving ........................................................................... 46
capture progress
  monitoring for moments ............................................. 125
capturing moments .................................................... 24, 123
classification
  objects ......................................................................... 150
classification tags
  obtaining from specification ....................................... 95
classifying objects ......................................................... 26
communication
  C++ apps ..................................................................... 79
  inter-app ...................................................................... 28, 166, 167
  inter-app and intra-app ................................................. 165
  WPF apps ..................................................................... 81
conversion between coordinate systems ......................... 102
coordinate systems ....................................................... 98
coordinates
  conversion .................................................................. 102
  CppQtSample ................................................................ 59
creating moments from images ...................................... 24, 127
Developer
  Forum .......................................................................... 15
development systems
  hardware ....................................................................... 37
diagnostics log files ..................................................... 169
displayability ................................................................... 77
  states ............................................................................. 78
displaying a window on the touch mat ......................... 109
documentation ............................................................... 10
  website .......................................................................... 10
downloading
  platform and SDK .......................................................... 40
dual screens and touch surfaces .................................... 24
events ............................................................................. 77
touch ............................................................................. 79
exceptions in WPF apps ............................................... 104
extract options
  pictures and outlines .................................................. 129
  extract options for outlines ......................................... 129
  extract options for pictures .......................................... 90
  C++ .............................................................................. 86
  WPF apps ..................................................................... 88