Switchblade SDK
API Description

VERSION: 2.0

Razer Switchblade SDK

Table of Contents

SWITCHBLADE OVERVIEW **4**

Switchblade Software Components 5
Application Execution Flow 5
Critical Changes from Prior Releases 6

API DEFINITIONS AND CONVENTIONS **7**

INSTALLING THE SDK **8**

Dynamic Key and Static Trackpad Images 9
Application Event Callback 9
Dynamic Key Events 10
Gestures 10
Keyboard Events 11
Manual Rendering 11
Notes on Window Rendering 11
Managed Code 12

DEBUGGING **13**

SAMPLE APPLICATIONS **14**

CamStreamer – Basic Rendering 14
FunctionTest - Dynamic Key and Gesture Processing 14

SDK REFERENCE **15**

RzSBStart 15
RzSBStop 16
RzSBQueryCapabilities 16
RzSBRenderBuffer 16
RzSBSetImageDynamicKey 17
RzSBSetImageTouchpad 18
RzSBAppEventSetCallback 18
RzSBDynamicKeySetCallback 20
RzSBCaptureKeyboard 21
RzSBCaptureKeyboardSetCallback 21
RzSBGestureSetCallback 23
RzSBEnableGesture 25
RzSBEnableOSGesture 25

APPENDIX A – SWITCHBLADE SDK RETURN VALUES **26**

Standard Errors 26
File I/O Errors 26
Callback Errors 27
Dynamic Key Errors 27
Touchpad Errors 27
Interface-Specific Errors 27
Status Macros 28

APPENDIX B – GESTURE PARAMETERS **29**

TABLE OF FIGURES

FIGURE 1 - SWITCHBLADE IN RAZER BLADE 4
FIGURE 2 - SAMPLE FLOW 5

Switchblade Overview

The Razer Switchblade User Interface is a unique interface that includes 10 Dynamic Adaptive Tactile Keys and a Multi-touch LCD Track-panel. The following Razer products contain the Switchblade User Interface:

- Razer Blade
- Razer Starwars™: The Old Republic™ keyboard
- Razer DeathStalker Ultimate keyboard

The Switchblade hardware consists of a Razer home key, ten dynamic keys with a 16-bit (RGB565) graphic underlay area, and a Synaptics 2-button touchpad also with a 16-bit (RGB565) graphic underlay area. Currently, all Switchblade devices run on a USB 2.0 interface.

The dimensions of the dynamic key images in the graphic underlay area are 115 x 115 pixels. The dimensions of the touchpad graphic underlay area are 800 x 480 pixels. The supported image file formats for both the dynamic keys and touchpad underlays are BMP, GIF, JPG, and PNG.

Each dynamic key has a unique ID. These keys are arranged in two rows and are numbered one through ten, with DK1 being the key in the lower left corner and DK10 being the key in the upper right corner. The dynamic keys support the display of separate images for the dynamic keys in the "DOWN" and "UP" state, also sometimes referred to as "Make" or "Break" events, respectively.

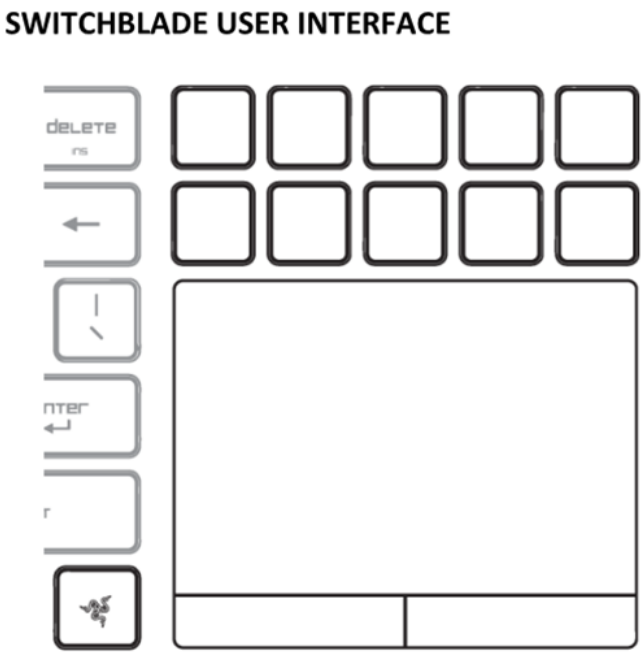

Figure 1 - Switchblade on the Razer Blade

On the Razer Blade, the Razer home key is located under the right shift key and is marked with the Razer logo. On the SWTOR and DeathStalker keyboards this key is located above the PgUp key. The home key is used to bring the user back to a default state when Synapse is running.

Switchblade Software Components

The Switchblade SDK framework main component is a DLL that applications either implicitly link to (using RzSwitchbladeSDK2.lib) or load and call. The DLL talks to the drivers and other framework components needed to access the Switchblade hardware.

The Switchblade Framework processes *RzHome*, *RzAppManager*, and *RzSBHelper* to facilitate communication between applications and the Switchblade device. If a connection cannot be established with the Switchblade device due to the abnormal termination of an application, it may be necessary to end these processes manually although an application restart usually rectifies the problem.

The Switchblade SDK currently supports the Razer Blade, SWTOR keyboard and the Deathstalker Ultimate Keyboard. The DLL and library are usable in both 32-bit and 64-bit enviroments.

Application Execution Flow

The following diagram illustrates the data and flow control between the application and the SDK. **`RzSBStart`** must be called, with success, before other Switchblade API calls can be made. **`RzSBStop`** must be called when the application terminates – this releases the application's lock on the Switchblade device and cleans up framework connections.

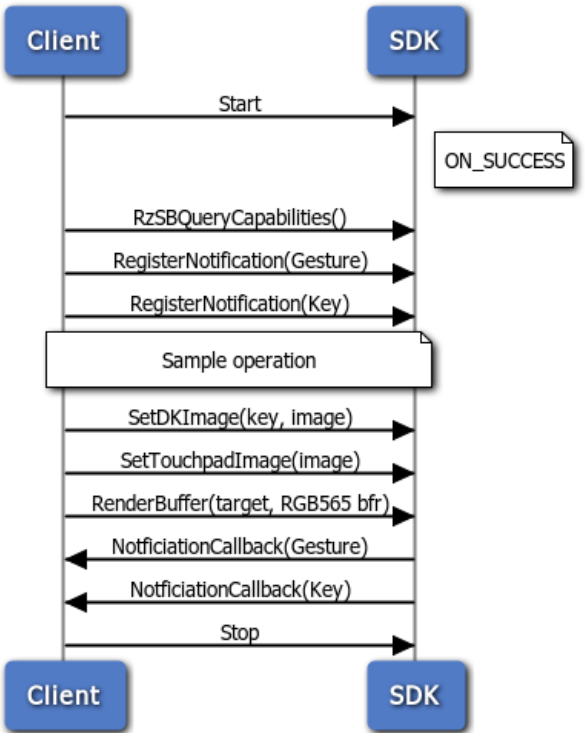

Figure 2 - Sample Flow

Critical Changes from Prior Releases

The SDK has the following critical changes that developers using the earlier versions of the SDK (prior to SDK 2.0) should be aware of:

- The SDK dll is now named **RzSwitchbladeSDK2.dll**. Previous versions uses **SwitchBladeSDK32.dll** which contains different interfaces. All succeeding development should make use of RzSwitchbladeSDK2.dll instead.

API Definitions and Conventions

Switchblade SDK applications run on the host machine and interact with the Switchblade and other compatible hardware. The infrastructure includes the SDK, the client application, and the Switchblade COM servers which are part of the overall Switchblade Framework.

Applications request access to Switchblade resources to render images in the available display types, e.g. the Dynamic Keys or LCD track-panel.

SDK API routine names are boldface, e.g. **`RzSBStart`**. All functions are __cdecl calls. All SDK APIs begin with the prefix "**`RzSB`**". Framework-specific types and constants are in **`blue`**, and most type definitions are found *`SwitchBladeSDK_types.h`*.

Windows API routines and constants are italic, e.g. *GetLastError()*.

All Switchblade Framework routines return HRESULTs, displayed in **`Red`**, which correspond to errors defined in *`SwitchBladeSDK_errors.h`*. Most return codes match the basic return codes defined in *`WinError.h`*.

The default success return result is **`RZSB_OK`**. The APIs return **`RZSB_NOT_STARTED`** when an application has not previously established a connection with the hardware. In many cases, *GetLastError()* provides additional error information.

All libraries and binaries use Unicode; the SDK does not support other character sets.

The current maximum length of the filepath plus the filename is 260 characters.

Installing the SDK

The Switchblade SDK installation package includes the SDK (lib/dll/doc) and prebuilt sample applications along with source code.

To install the Switchblade SDK on computer with a compatible Switchblade device, ensure that Synapse has completed all pending updates before running the installer.

The installer places the SDK in *C:\Razer\SwitchBladeSDK*.

The Switchblade SDK header files are placed in the *\include* folder of the current version number. Switchblade SDK library files are placed in the *\lib* folder. These folders must be included in any application that wishes to use a Switchblade device.

Precompiled sample applications are in *\bin*, with source located in *\source\apps*. Sample applications built from the included projects put binaries in the *\source\apps\bin* subdirectory (they do not replace the prebuilt binaries).

To uninstall the Switchblade SDK, run UninstallSDK.exe located in the root folder for the installed version.

Do not copy files from one system to another, as you may still be missing the necessary drivers and DLLs to properly use the Switchblade SDK. Always use the installer instead.

COM Initialization and Models

The SDK requires interaction with COM services in the Switchblade framework.
It is recommended that application developers explicitly initialize the COM to their intended use before calling any of the SDK APIs.

If the COM is not initialized upon using the SDK, the DLL initializes the COM libraries using a Multi-Threaded Apartment Model (MTA).

See section "SDK Reference – RzSBStart" for additional details.

Dynamic Key and Static Trackpad Images

The SDK provides simple interface functions to modify the dynamic key images as well as the LCD touchscreen graphic. Two interface functions provide static rendering capabilities:

> Call **`RzSBSetImageDynamicKey`** to modify the images associated with each dynamic key state. This call supports image assignment for each key in one of two states – up or down; the SDK automatically switches the up/down images to reflect the state of each dynamic key.
>
> Call **`RzSBSetImageTouchpad`** to set a static LCD touchpad image. This call replaces any previously set graphic, however is not suitable for dynamically changing images.

To render dynamic images, see Manual Rendering to learn more about the different approaches in the SDK.

Razer Key on Switchblade Devices

`RzHome`, the application that manages the main launcher display on system startup, takes over the Switchblade module when a user presses the Home key on a Razer keyboard.

An event will be triggered signaling the SDK application whenever this occurs, allowing it to perform cleanup (i.e. closing down of application). Normally, this indicates that the application should perform cleanup and close down.

Application Event Callback

The Switchblade framework supports running multiple applications at the same time. However, only one application is active at one time. With regards to this, an application event scheme is provided to allow the SDK application to react depending on its state with the Switchblade UI.

Call **`RzSBAppEventSetCallback`** to register for notification that Switchblade ownership has been changed.

The events received in the assigned callback are summarized by the following table:

ACTIVATED	The application is activated (the SB framework may run multiple apps at the same time). Since the application is now the active app, the application can perform a refresh of its UI.
DEACTIVATED	The application has been put to the background, and another application is actively using the Switchblade UI. This indicates that the SDK application can temporarily stop its UI updates and major operations (since it won't be visible and usable for the user until it gets reactivated). This is normally triggered when the user run another application or pressed the Razer Home key. *
CLOSE	The SB framework has initiated a request to close the application. The application should perform cleanup, and as much as possible terminate on its own.
EXIT	The SB framework will forcefully close the application. This event is always preceeded by the **RZSBSDK_EVENTTYPE_CLOSE** event. Cleanup should be done there. Right after this event is received, the application will be forcibly closed. This serves as a final chance for the application to release any resources it may still be holding.

*Upon receiving the **DEACTIVATE** event, if the application does not need to run in the background – it may opt to perform cleanup and close its own.

Dynamic Key Events

The SDK provides callback notification for dynamic key events. The callback function provides granular control over each key's events.

Call **RzSBDynamicKeySetCallback** to register an event handler for dynamic key events.

Gestures

The SDK provides event notification for many gestures and can filter transmission of these gestures to the operating system. Gestures include Press, Tap, Flick, Zoom, and Rotate.

Enable gesture event notification with a call to **RzSBEnableGesture**, specifying gestures that should generate events to an application-defined callback.

For each gesture, determine if it should be propagated to the operating system for further processing. Call **RzSBEnableOSGesture** to adjust these filters.

Finally, register the gesture event handler with a call to **RzSBGestureSetNotification**. The application-defined event handler is subsequently called with each individual gesture event.

Keyboard Events

Since the Switchblade devices come with keyboard sets, the SDK provides event notifications to capture the key presses. This only affects the keyboard of the Switchblade device and not any other keyboard attached to the user's system.

Call **`RzSBCaptureKeyboard`** to enable the keyboard event capturing in the SDK application. Once enabled, only the SDK application will receive keyboard input from the device where the application is running. Other keyboards attached to the system will work normally.

Call **`RzSBKeyboardCaptureSetCallback`** to register an event handler for dynamic key events. It works the same as the other functions to set a callback.

When the application no longer needs the keyboard capture, call **`RzSBCaptureKeyboard`** again to release the device and allow it for normal keyboard operations.

Manual Rendering

The **`RzSBRenderBuffer`** function allows for rendering images on the LCD trackpad using raw bitmap data.

For very specific control of images displayed on the Switchblade LCD Track-panel or Dynamic Keys, use **`RzSBRenderBuffer`**. This function takes a pointer to an RGB565 bitmap image.

When rendering to the LCD Track-panel, this bitmap should be 800x480 pixels in size. When rendering to Dynamic Keys, this buffer should be 115x155 pixels in size.

Note that this buffer is presented "upside down" from *Bit(Stretch)Blt* function formats. Adjust y coordinates when sharing buffers with **Blt* functions.

Call **`RzSBRenderBuffer`** for each required display update. Note however that buffer rendering capabilities are not incremental; this call updates the entire target area each time.

With this method, the Switchblade module can render dynamic content to the LCD trackpad display at between 20 and 30fps, and often higher, depending on system load.

Notes on Window Rendering

Previous SDK versions included automatic window rendering (functions prefixed with **`RzSBWin`**). These are no longer supported in the SDK 2.0 to streamline the core SDK library which do not require them.

In the future releases of the SDK, the functionalities would be bundled in separate libraries so only applications that require them would need to load the functionalities.

However, the application developer can achieve the same amount of functionality using the current SDK. The developer can implement their own processing to render a window's visible area (e.g.

MFC dialog window) onto a bitmap and use the **`RzSBRenderBuffer`** function to render the image on the trackpad. The updating of the display can either be done on a periodic interval (e.g. polling the display) or by interpreting window messages (e.g. *WM_PAINT* messages, or client redraw due to Windows themes) that is triggered whenever the view of the window changes.

Mouse input and control selection for the dialog can be emulated by using the gesture data received from the gesture events (see section "" for details). Gestures can then be interpreted to send *WM_LBUTTONDOWN* and *WM_LBUTTONUP* messages to the window or control.

Keyboard input can be processed using the standard Windows API (e.g. RawInput, SendInput) or through the **`RzSBKeyboardCaptureSetCallback`** event.

Managed Code

The Switchblade SDK supports calls from managed code (C++, C#, Visual Basic, etc). Each technology provides import and marshaling capabilities to call across managed/unmanaged boundaries.

A sample C# wrapper is provided with the SDK installer (as an archive file).

Debugging

During application development, there will be times that the application does not run to completion and shuts down properly. In these cases, there are a few systemic events to consider.

While debugging an application, the API calls particularly **`RzSBStart`** and **`RzSBStop`** might fail preventing an application from being run or further debugged. This scenario is often triggered when applications are forcibly terminated (application did not have a chance to properly release Switchblade resources) or the application failed to do the proper cleanup.

Often, simply pressing the Razer Home key would reset the environment for the next application run. However if that fails, follow these steps to attempt to restart the Switchblade framework (without the need to restart the whole system):

1. Start the **Task Manager**
2. Navigate to the **Processes** tab
3. Right-click on *RzAppManager* and select **End Process**
   (accept the confirmation dialog)
4. Repeat for *RzHome*, *RzSBHelper*, and *RzSynapse* – in that order
5. Restart *RzSynapse* (from the **Start Menu** or from
   `C:\Program Files (x86)\Razer\Synapse\RzSynapse.exe`)

Sample Applications

The SDK includes several sample applications designed to illustrate the different SDK capabilities. One Microsoft Visual Studio solution file contains all sample application projects:

C:\Razer\SwitchBladeSDK\v2.0\source\apps\RzSDKSampleApps.sln

CamStreamer – Basic Rendering

The *CamStreamer* sample application makes use of **RzSBRenderBuffer** to redirect camera input to the Switchblade LCD trackpad. While much of this code deals with the camera interface, the reusable SDK portions include:

1. Translating an RGB buffer into RGB565 format
2. Creating a proper **RZSB_BUFFERPARAMS** structure
3. Calling **RzSBRenderBuffer** periodically, using a Windows timer

FunctionTest - Dynamic Key and Gesture Processing

The *FunctionTest* sample application illustrates dynamic key and gesture event processing. Reusable SDK portions include:

1. Setting and updating dynamic key up/down images (**RzSBSetImageDynamicKey**)
2. Setting trackpad display image (**RzSBSetImageTouchpad**)
3. Registering for and processing dynamic key events (**RzSBDynamicKeySetCallback**)
4. Enabling gestures (**RzSBEnableGesture**)
5. Filtering gestures (**RzSBEnableOSGesture**)
6. Registering for and processing gesture events (**RzSBGestureSetCallback**)
7. Responding to application event callbacks (**RzSBAppEventSetCallback**)
8. Responding to keyboard event callbacks (**RzSBKeyboardCaptureSetCallback**)

SDK Reference

RzSBStart

Grants access to the Switchblade device, establishing application connections.

```
HRESULT RzSBStart();
```

Remarks

RzSBStart sets up the connections that allow an application to access the Switchblade hardware device. This routine returns **RZSB_OK** on success, granting the calling application control of the device. Subsequent calls to this routine prior to a matching **RzSBStop** call are ignored.

RzSBStart must be called before other Switchblade SDK routines will succeed.
RzSBStart must always be accompanied by an **RzSBStop**.

COM initialization should be called prior to calling **RzSBStart**.

If the application developer intends to use Single-Threaded Apartment model (STA) and call the SDK functions within the same thread where the COM was initialized, then CoInitialize() should be called before **RzSBStart**. Note that some MFC controls automatically initializes to STA.

If the application developer intends to call the SDK functions on different threads, then the CoInitializeEx() should be called before **RzSBStart**.

Note: When the **RzSBStart()** is called without the COM being initialized (e.g. thru calling *CoInitializeEx*) the SDK initializes the COM to Multi-Threaded Apartment (MTA) model. As such, callers must invoke SDK functions from an MTA thread.

Future SDK versions will move these calls into an isolated STA, giving application developers additional freedom to use COM in any fashion.

However, application developers may already implement their own processing to isolate the SDK initialization and calls to avoid the issues for COM in different threading models.

RzSBStop

Cleans up the Switchblade device connections and releases it for other applications.

```
void RzSBStop();
```

Remarks

RzSBStop cleans up the connections made by **RzSBStart**. This routine releases an application's control of the Switchblade hardware device, allowing other applications to take control. Subsequent calls to this routine prior to a matching **RzSBStart** are ignored.

If an application terminates after calling **RzSBStart** without a matching call to **RzSBStop**, other applications may fail to acquire control of the Switchblade device. In this case, manually kill the framework processes. See section "Debugging" for more details.

RzSBQueryCapabilities

Collects information about the SDK and the hardware supported.

```
HRESULT RzSBQueryCapabilities(
      __out PRZSBSDK_QUERYCAPABILITIES pSBSDKCap
);
```

Parameters

pSBSDKCap [out]

A pointer to a previously allocated structure of type **RZSBSDK_QUERYCAPABILITIES**, defined in *SwitchBladeSDK_types.h*. On successful execution, this routine fills the parameters in pSBSDKCap with the proper information about the SDK and supported hardware.

RzSBRenderBuffer

Controls output to the Switchblade display.
The application can send bitmap data buffer directly to the Switchblade trackpad display thru this function providing are more direct and faster way of updating the display.

```
HRESULT RzSBRenderBuffer (
      __in  RZSBSDK_DISPLAY dwTarget,
      __in  RZSBSDK_BUFFERPARAMS *bufferParams
 );
```

Parameters

dwTarget [in]

Specifies the target location on the Switchblade display – the main display or one of the dynamic key areas. Please refer to the definition for **RZSBSDK_DISPLAY** in *SwitchBladeSDK_types.h* for accepted values.

**bufferParams* [in]

A pointer to a buffer parameter structure of type **RZSBSDK_BUFFERPARAMS** that must be filled with the appropriate information for the image being sent to the render buffer.
This input parameter is an RGB565 bitmap image buffer with a bottom-up orientation.
Please refer to the definition for **RZSBSDK_BUFFERPARAMS** in *SwitchBladeSDK_types.h* for further detail.

Note: Since the function accepts the buffer for bottom-up bitmap, the application should invert the original image along its vertical axis prior to calling the function. This can be done easily with BitBlit and StretchBlt APIs.

RzSBSetImageDynamicKey

Set images on the Switchblade UI's Dynamic Keys.

```
HRESULT RzSBSetImageDynamicKey(
      __in RZSBSDK_DKTYPE DynamicKey,
      __in RZSBSDK_KEYSTATETYPE state,
      __in LPWSTR lpszImageFilename
);
```

Parameters

DynamicKey [in]

The dynamic key rendering target. See the Switchblade SDK header file *SwitchBladeSDK_types.h* for the dynamic key definitions.

state [in]

The desired dynamic key state (up, down) for the specified image. See the Switchblade SDK header file *SwitchBladeSDK_types.h* for accepted values.

lpszImageFilename [in]

The image filepath for the given *state*. This image should be 115 x 115 pixels in dimension. Accepted file formats are BMP, GIF, JPG, and PNG.

Note: Animation in GIF files are not supported.

RzSBSetImageTouchpad

Places an image on the main Switchblade display.

```
HRESULT RzSBSetImageTouchpad(
      __in LPWSTR lpszImageFilename
);
```

Parameters

lpszImageFilename [in]

Filepath to the image to be placed on the main Switchblade display. This image should be 800 x 480 pixels in dimension. Accepted file formats are BMP, GIF, JPG, and PNG.

Note: Animation in GIF files are not supported.

RzSBAppEventSetCallback

Sets the callback function for application event callbacks.

```
HRESULT RzSBAppEventSetCallback (
      __in AppEventCallbackType pFn
 );
```

Parameters

pFn [in]

Pointer to a callback function. If this argument is set to NULL, the routine clears the previously set callback function.

Remarks

RzSBAppEventSetCallback sets the callback function for Switchblade ownership change notification and requires a pointer to a valid callback function. The event handling function must take three arguments: **RZSBSDK_EVENTTYPE** rzEvent, **DWORD** wParam, and **DWORD** lParam, in this order. rzEvent determines the type of event, and the other two parameters are reserved for additional information specific to the type of event.

rzEvent can be one of three values:

RZSBSDK_EVENTTYPE_ACTIVATED
RZSBSDK_EVENTTYPE_DEACTIVATED
RZSBSDK_EVENTTYPE_CLOSE
RZSBSDK_EVENTTYPE_EXIT

The SDK application can react to the above events depending on their intended operation:

RZSBSDK_EVENTTYPE_ACTIVATED	The Switchblade framework has activated the SDK application. The application can resume its

	operations and update the Switchblade UI display.
RZSBSDK_EVENTTYPE_DEACTIVATED	The application has been deactivated to make way for another application. In this state, the SDK application will not receive any Dynamic Key or Gesture events, nor will it be able to update the Switchblade displays.
RZSBSDK_EVENTTYPE_CLOSE	The Switchblade framework has initiated a request to close the application. The application should perform cleanup and can terminate on its own when done.
RZSBSDK_EVENTTYPE_EXIT	The Switchblade framework will forcibly close the application. This event is always preceeded by the **RZSBSDK_EVENTTYPE_CLOSE** event. Cleanup should be done there.

Example

```
// MyAppEventCallback gets the gesture callback and closes the application.
// Init() is called near the start of the application, right after
// making a call to RzSBStart().

HRESULT STDMETHODCALLTYPE MyAppEventCallback(
       RZSBSDK_EVENTTYPE eventtype,
       DWORD dwParam1,
       DWORD dwParam2)
{
       HRESULT hReturn = S_OK;

       // Close the app the moment we deactivate or receive the close request.
       if (eventtype == RZSBSDK_EVENTTYPE_DEACTIVATED ||
           eventtype == RZSBSDK_EVENTTYPE_CLOSE)
              PostQuitMessage(0);

       return hReturn;
}

HRESULT Init()
{
       HRESULT hResult = S_OK;

       // Set MyDynamicKeyCallback as the callback function.
       hResult = RzSBAppEventSetCallback(
            reinterpret_cast<AppEventCallbackType*>(MyAppEventCallback));

       return hResult;
}
```

RzSBDynamicKeySetCallback

Sets the callback function for dynamic key events.

```
HRESULT RzSBDynamicKeySetCallback(
      __in DynamicKeyCallbackFunctionType pFn
 );
```

Parameters

pFn [in]

Pointer to a callback function. If this argument is set to NULL, the routine clears the previously set callback function.

Remarks

RzSBDynamicKeySetCallback sets the callback function for dynamic key events and requires a pointer to a valid callback function. The dynamic key event handling function must take two arguments of type **RZSBSDK_DKTYPE** DynamicKey and **RZSBSDK_KEYSTATETYPE** DynamicKeyState, in this order. DynamicKey indicates the pressed dynamic key, while DynamicKeyState indicates the state of the dynamic key when the callback was sent.

Example

```
// MyDynamicKeyCallback gets the gesture callback and forwards it to
// the appropriate handler if dynamic key 1 is pressed.
// Init() is called near the start of the application, right after
// making a call to RzSBStart().

HRESULT STDMETHODCALLTYPE MyDynamicKeyCallback(
       RZSBSDK_DKTYPE DynamicKey,
       RZSBSDK_KEYSTATETYPE DynamicKeyState)
{
       HRESULT hReturn = S_OK;

       // If it is dynamic key 1 and the key is in the down state
       if(DynamicKey == RZSBSDK_DK1 && DynamicKeyState == RZSBSDK_KEYSTATE_DOWN)
              DK1Handler();

       return hReturn;
}

HRESULT Init()
{
       HRESULT hResult = S_OK;

       // Set MyDynamicKeyCallback as the callback function.
       hResult = RzSBDynamicKeySetCallback(
            reinterpret_cast<DynamicKeyCallbackFunctionType*>(MyDynamicKeyCallback));

       return hResult;
}
```

RzSBCaptureKeyboard

Enables or disables the keyboard capture functionality.

```
HRESULT RzSBEnableGesture(
      __in bool bEnable
);
```

Parameters
bEnable [in]
The enable state. *true* enables the capture while *false* disables it.

Remarks
When the capture is enabled, the SDK application can receive keyboard input events through the callback assigned using **RzSBKeyboardCaptureSetCallback**. The OS will not receive any keyboard input from the Switchblade device as long as the capture is active. Hence, applications must release the capture when no longer in use (call **RzSBEnableGesture** with *false* as parameter).

The function only affects the keyboard device where the application is running. Other keyboard devices will work normally.

RzSBCaptureKeyboardSetCallback

Sets the callback function for dynamic key events.

```
HRESULT RzSBKeyboardCaptureSetCallback(
      __in KeyboardCallbackFunctionType pFn
 );
```

Parameters
pFn [in]
Pointer to a callback function. If this argument is set to NULL, the routine clears the previously set callback function.

Remarks
RzSBKeyboardCaptureSetCallback sets the callback function for keyboard events and requires a pointer to a valid callback function.

The key event handling function must take three arguments of type `UINT uMsg, WPARAM wParam` and `LPARAM lParam`, in this order.

The uMsg indicates the keyboard event (WM_KEYDOWN, WM_KEYUP, WM_CHAR). The wParam indicates the key pressed (virtual key value). The lParam indicates the key modifiers (i.e. control key).

The application will only receive this event if the capture was enabled using the **`RzSBEnableGesture`** function.

Example

```
// MyKeyboardCallback gets the gesture callback and forwards it to
// the appropriate handler if the escape key is pressed.
// Init() is called near the start of the application, right after
// making a call to RzSBStart().
// Cleanup() is called before closing the application, before calling RzSBStop().


HRESULT STDMETHODCALLTYPE MyKeyboardCallback(
	UINT uMsg,
	WPARAM wParam,
	LPARAM lParam)
{
	HRESULT hReturn = S_OK;

	if (uMsg == WM_KEYDOWN && wParam == VK_ESCAPE)
		EscapeKeyHandler();

	return hReturn;
}

HRESULT Init()
{
	HRESULT hResult = S_OK;

	// Enable the keyboard capture
	RzSBCaptureKeyboard(true);

	// Set MyKeyboardCallback as the callback function.
	hResult = RzSBKeyboardCaptureSetCallback(
	    reinterpret_cast<KeyboardCallbackFunctionType*> MyKeyboardCallback));

	return hResult;
}

void Cleanup()
{
	// Release the keyboard capture
	RzSBCaptureKeyboard(false);
}
```

RzSBGestureSetCallback

Sets the callback function for gesture events.

```
HRESULT RzSBGestureSetCallback(
      __in TouchpadGestureCallbackFunctionType pFn
);
```

Parameters

pFn [in]

Pointer to a callback function. If this argument is set to NULL, the routine clears the previously set callback function.

Remarks

RzSBGestureSetCallback sets the callback function for gesture events and requires a pointer to a valid callback function. Only one valid gesture may be specified per callback request, however the SDK supports one callback for each supported gesture. The callbacks can be separate routines, or a single handler.

The gesture event handling function must take five arguments: **RZSBSDK_GESTURETYPE** gesture, **DWORD** dwParameters, **WORD** wXPos, **WORD** wYPos and **WORD** wZPos, in this order. The gesture value determines the type of received gesture. The parameters wXPos, wYPos and wZPos provide additional information for the received gesture. Refer to the Appendix for detailed gesture parameters.

Example

```
// MyGestureCallback gets the gesture event and forwards it to the
// appropriate handler for taps and release of a press.
// Init() is called near the start of the application, right after
// making a call to RzSBStart().

HRESULT STDMETHODCALLTYPE MyGestureCallback(
       RZSBSDK_GESTURETYPE gesture,
       DWORD dwParameters,
       WORD wXPos,
       WORD wYPos,
       WORD wZPos)
{
       HRESULT hReturn = S_OK;
       switch(gesture) {
       case RZSBSDK_GESTURE_PRESS:
              hReturn = PressHandler(wXPos, wYPos);
              break;
       case RZSBSDK_GESTURE_TAP:
              hReturn = TapHandler(wXPos, wYPos);
              break;
       default:
              // We don't care about other gestures, so break
              break;
       }
       return hReturn;
}

HRESULT Init()
{
       HRESULT hResult = S_OK;

       // Set MyGestureCallback as the callback function.
       hResult = RzSBGestureSetCallback(
           reinterpret_cast<TouchpadGestureCallbackFunctionType*>(MyGestureCallback));

       return hResult;
}
```

RzSBEnableGesture

Enables or disables gesture events.

```
HRESULT RzSBEnableGesture(
    __in RZSBSDK_GESTURETYPE gesture,
    __in bool bEnable
);
```

Parameters

gesture [in]

Gesture to be enabled or disabled. Valid values consist of gesture bitwise combination found in the file *SwitchBladeSDK_types.h*.

bEnable [in]

The enable state. *true* enables the gesture while *false* disables it.

Remarks

In nearly all cases, gestural events are preceded by a **RZSBSDK_GESTURE_PRESS** event. With multiple finger gestures, the first finger contact registers as a press, and the touchpad reports subsequent contacts as the appropriate compound gesture (tap, flick, zoom or rotate).

RzSBEnableOSGesture

Enables or disables gesture event forwarding to the OS.

```
HRESULT RzSBEnableOSGesture(
    __in RZSBSDK_GESTURETYPE gesture,
    __in bool bEnable
);
```

Parameters

gesture [in]

Gesture to be enabled or disabled. Valid values consist of gesture bitwise combination found in the file *SwitchBladeSDK_types.h*.

bEnable [in]

The enable state. *true* enables the gesture while *false* disables it.

Remarks

Setting the **RZSBSDK_GESTURE_PRESS** for OS gesture is equivalent to **RZSBSDK_GESTURE_PRESS**, **RZSBSDK_GESTURE_MOVE** and **RZSBSDK_GESTURE_RELEASE**.

Appendix A – Switchblade SDK Return Values

The Switchblade SDK uses HRESULTS for function return values. These definitions are described in the text that follows. Use *GetLastError()* to obtain additional information.

SwitchBladeSD_errors.h contains the full list of errors. Utilize these results for compatibility with previous and subsequent versions of the SDK.

Standard Errors

RZSB_OK
Generic success return value. Defined as **S_OK**.

RZSB_UNSUCCESSFUL
The application failed to establish a connection with the Switchblade device. Call *GetLastError()* for additional error information. Defined as **E_FAIL**.

RZSB_INVALID_PARAMETER
One or more of the parameters is invalid. Call *GetLastError()* for additional error information. Defined as **E_INVALIDARG**.

RZSB_INVALID_POINTER
One or more of the pointers point to data that is either not fully readable or writable. Defined as **E_POINTER**.

RZSB_FILE_NOT_FOUND
The referenced file could not be found. Defined as **ERROR_FILE_NOT_FOUND**.

File I/O Errors

RZSB_FILE_ZERO_SIZE
Zero-length files are not allowed.

RZSB_FILE_INVALID_NAME
The filepath points to a file that does not exist.

RZSB_FILE_INVALID_TYPE
Zero-sized images are not allowed.

RZSB_FILE_READ_ERROR
The number of bytes read was different from the number of bytes expected.

RZSB_FILE_INVALID_FORMAT
The image file is not a supported file format.

RZSB_FILE_INVALID_LENGTH
The file length does not match the expected length.

RZSB_FILE_NAMEPATH_TOO_LONG
The length of the filepath is greater than 260 characters.

RZSB_IMAGE_INVALID_SIZE
The dimensions of the image do not match the requirements for the display.

RZSB_IMAGE_INVALID_DATA
The image could not be verified as valid. Call *GetLastError()* for additional error information.

Callback Errors

RZSB_CALLBACK_NOT_SET
The application tried to call or clear a callback that was never set.

RZSB_CALLBACK_REMOTE_FAIL
Setting the callback failed on the server.

Dynamic Key Errors

RZSB_DK_INVALID_KEY
The dynamic key referenced is not a valid dynamic key value. Refer to *SwitchBladeSDK_types.h* for valid values.

RZSB_DK_INVALID_KEY_STATE
The dynamic key state referenced is not a valid dynamic key state. Refer to *SwitchBladeSDK_types.h* for valid states.

Touchpad Errors

RZSB_TOUCHPAD_INVALID_GESTURE
Invalid gesture input detected by the Switchblade touchpad.

Interface-Specific Errors

RZSB_NOT_STARTED
The internal callstack is in disorder. This is sometimes due to not having previously called **RzSBStart**.

Status Macros

Use the following status macros as a straightforward way to check if certain criteria have been met. All macros return a Boolean value.

`RZSB_SUCCESS(a) (S_OK == (a))`

Takes an HRESULT and checks if the function that returned the HRESULT succeeded, returning *true* if it did. Defined in *`SwitchBladeSDK_errors.h`*.

`RZSB_FAILED(a) (S_OK != (a))`

Takes an HRESULT and checks if the function that returned that HRESULT failed, returning *true* if it did. Defined in *`SwitchBladeSDK_errors.h`*.

`ValidGesture(a) ((a) & RZGESTURE_ALL)`

Takes a **`RZSBSDK_GESTURETYPE`** and checks to ensure that it is a valid gesture, returning *true* if it is. Defined in *`SwitchBladeSDK_types.h`*.

`SingleGesture(a) (0 == ((a-1) & (a)))`

Takes a **`RZSBSDK_GESTURETYPE`** and checks whether it is a single gesture, returning *true* if it is. Defined in *`SwitchBladeSDK_types.h`*.

Appendix B – Gesture Parameters

The values of the parameters for the `TouchpadGestureCallbackFunction` varies depending on the type of gesture. Please refer to the following table for each values:

RZSBSDK_GESTURE_PRESS	dwParameters: number of touch points wXPos: x-coordinate of the touch point wYPos: y-coordinate of the touch point wZPos: reserved
RZSBSDK_GESTURE_RELEASE	dwParameters: number of touch points wXPos: x-coordinate of the touch point wYPos: y-coordinate of the touch point wZPos: reserved
RZSBSDK_GESTURE_TAP	dwParameters: reserved wXPos: x-coordinate of the touch point wYPos: y-coordinate of the touch point wZPos: reserved
RZSBSDK_GESTURE_FLICK	dwParameters: number of touch points wXPos: reserved wYPos: reserved wZPos: direction of the flick RZSBSDK_DIRECTION_LEFT, RZSBSDK_DIRECTION_RIGHT, RZSBSDK_DIRECTION_UP, RZSBSDK_DIRECTION_DOWN
RZSBSDK_GESTURE_ZOOM	dwParameters: 1 if zoom in, 2 if zoom out wXPos: reserved wYPos: reserved wZPos: reserved
RZSBSDK_GESTURE_ROTATE	dwParameters: 1 if clockwise, 2 if counter-clockwise wXPos: reserved wYPos: reserved wZPos: reserved
RZSBSDK_GESTURE_MOVE	dwParameters: reserved wXPos: x-coordinate of the touch point wYPos: y-coordinate of the touch point wZPos: reserved