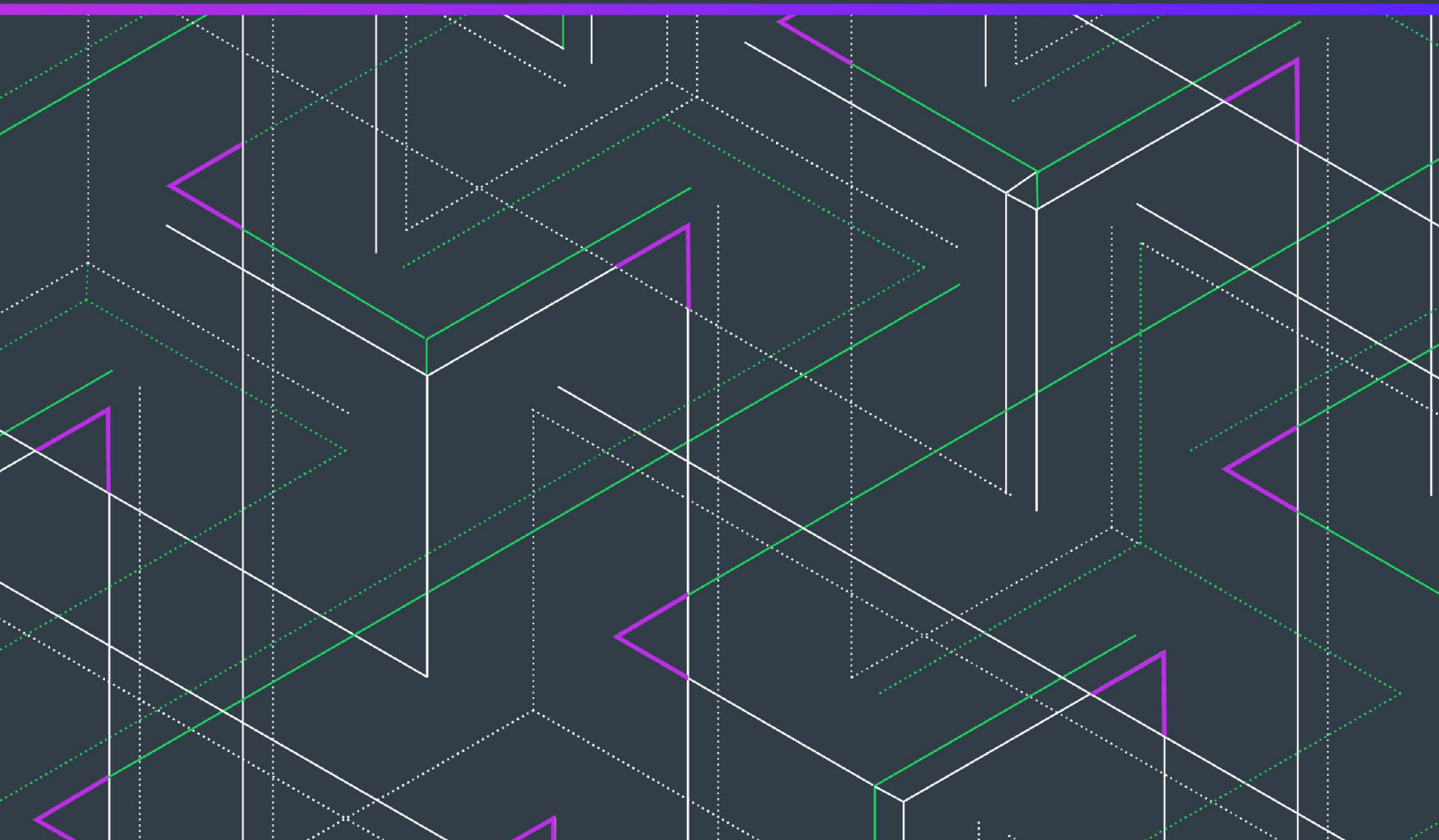
revenera™

Usage Intelligence 5.6.1

C++ for Windows SDK Developer Guide



Legal Information

Book Name: Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide

Part Number: FUI-0561-CPPUG

Product Release Date: 15 September 2023

Copyright Notice

Intellectual Property

Restricted Rights Legend

Contents

1 Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide 5

Release Notes 6

Product Support Resources 6

Contact Us 7

2 Getting Started with the Usage Intelligence C++ for Windows SDK 9

System Requirements. 9

Registering Your Product 9

Importing the SDK Files 10

Basic Integration Steps 10

Text Encoding 13

Next Steps 13

Advanced Features 13

3 SDK Initialization and Configuration. 15

Creating the Usage Intelligence SDK Instance 16

Destroying the Usage Intelligence SDK Instance 16

Getting SDK Version Information 17

Getting the Client ID 18

Initializing the Configuration 18

Single vs. Multiple Session Modes 22

Opt-Out Mechanism 22

Providing Further Data 23

Product Details. 23

Setting Product Data 24

Setting Product Edition 25

Setting Product Language 26

Setting Product Version 28
Setting Product Build Number 29
License Management 30
Changing ReachOut on Autosync Setting 32
Proxy Support 34

4 Basic SDK Controls 37
Starting the SDK 37
Stopping the SDK 38
Starting a Session 40
Stopping a Session 42
Caching and Synchronizing 43
Forced Synchronization 44

5 Feature / Event Tracking 47
Tracking an Event 48
Logging a Normal Event with a Numeric Field 50
Logging a Normal Event with a String Field 52
Logging a Custom Event 55

6 ReachOut Direct-to-Desktop Messaging Service 59
Automated Message Retrieval 60
Manual Message Retrieval 62
Checking for Manual ReachOut Messages of Any Type 62
Checking for Manual ReachOut Messages of a Specified Type 64
Freeing the Memory Allocated by the SDK 66

7 Exception Tracking 67

8 License Management 71
Client vs. Server Managed Licensing 71
Checking the License Data of the Supplied License Key 72
Setting the Current License to the Supplied License Key 75

9 Custom Properties 81
Setting Custom Property Data 81

10 SDK Status Checks 83
Getting the State of the RUI Instance 83
Testing the Connection Between the SDK and the Server 85

11 Common Function Return Values 87

1

Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide

Usage Intelligence 5.6.1—a software usage analytics solution designed for distributed C/C++, .NET, Obj-C and native Java applications on Windows, Macintosh, and Linux—provides deep insight into application usage. It enables you to see which of your application's features are used most and least often. Advanced reporting lets you filter by properties including region, version, OS platform, and architecture to focus your roadmap development.

The Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide explains how to implement the C++ for Windows SDK.

Table 1-1 ▪ Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide

Section	Description
Getting Started with the Usage Intelligence C++ for Windows SDK	Explains how to get started using Usage Intelligence C/C++ for Windows SDK.
SDK Initialization and Configuration	Explains how to create the Usage Intelligence SDK instance, initialize the configuration, and other configuration tasks.
Basic SDK Controls	Describes how to start and stop the SDK, and start and stop a session.
Feature / Event Tracking	Explains how to track and log events.
ReachOut Direct-to-Desktop Messaging Service	Explains how to create ReachOut messaging campaigns to deliver messages or surveys directly to the desktop of users who are running your software.
Exception Tracking	Describes how to collect runtime exceptions from your application.
License Management	Explains how to maintain a license key registry on the Usage Intelligence server in order to track license key usage and verify the status/validity of license keys used on your clients.

Table 1-1 ▪ Usage Intelligence 5.6.1 C++ for Windows SDK Developer Guide (cont.)

Section	Description
Custom Properties	Explains how to collect any custom value that is relevant to your specific application.
SDK Status Checks	Describes how to collect custom values that are relevant to your specific application.
Common Function Return Values	Lists common return values for Usage Intelligence functions.

Release Notes

Usage Intelligence 5.1.6 C++ for Windows SDK was released on September 15, 2023.

Usage Intelligence 5.6.1 C++ for Windows SDK

The following issue was resolved in the Usage Intelligence 5.6.1 C++ for Windows SDK:

Table 1-2 ▪ Resolved Issues in Usage Intelligence 5.1.1 C++ for Windows SDK

Issue	Description
RUI-2374	Vulnerabilities were discovered in the `ruiSDK_5.6.0.x86.dll` file. These vulnerabilities have been fixed in the 5.6.1 release of the Windows C++ RUI SDK.

Product Support Resources

The following resources are available to assist you with using this product:

- Revenera Product Documentation
- Revenera Community
- Revenera Learning Center
- Revenera Support

Revenera Product Documentation

You can find documentation for all Revenera products on the Revenera Product Documentation site:

https://docs.revenera.com

Revenera Community

On the Revenera Community site, you can quickly find answers to your questions by searching content from other customers, product experts, and thought leaders. You can also post questions on discussion forums for experts to answer. For each of Revenera's product solutions, you can access forums, blog posts, and knowledge base articles.

https://community.revenera.com

Revenera Learning Center

The Revenera Learning Center offers free, self-guided, online videos to help you quickly get the most out of your Revenera products. You can find a complete list of these training videos in the Learning Center.

https://learning.revenera.com

Revenera Support

For customers who have purchased a maintenance contract for their product(s), you can submit a support case or check the status of an existing case by making selections on the **Get Support** menu of the Revenera Community.

https://community.revenera.com

Contact Us

Revenera is headquartered in Itasca, Illinois, and has offices worldwide. To contact us or to learn more about our products, visit our website at:

http://www.revenera.com

You can also follow us on social media:

- Twitter
- Facebook
- LinkedIn
- YouTube
- Instagram

2

Getting Started with the Usage Intelligence C++ for Windows SDK

This section explains how to get started using Usage Intelligence C++ for Windows SDK:

- System Requirements
- Registering Your Product
- Importing the SDK Files
- Basic Integration Steps
- Text Encoding
- Next Steps

System Requirements

The Usage Intelligence C++ for Windows SDK has been tested on Windows versions from Windows XP through Windows 10. Both 32-bit and 64-bit builds of the Usage Intelligence DLLs are provided. The Usage Intelligence DLL loads in any dependent libraries automatically.

Registering Your Product

Before you can use the Usage Intelligence Software Analytics service or integrate the Usage Intelligence SDK with your software, you must first create an account by visiting https://info.revenera.com/SWM-EVAL-Usage-Intelligence.

Once you have a user name and register a new product account for tracking your application, you can get your Product ID, CallHome URL, and AES Key from the Administration page (within the Usage Intelligence dashboard). From here you can also download the latest version of the SDK.

Importing the SDK Files

Upon downloading the Usage Intelligence C++ for Windows SDK, you will find two header files, `ruiSDKC.h` and `ruiSDKDefines.h`, plus two DLLs: `ruiSDK_<version>.x64.dll` and `ruiSDK_<version>.x86.dll`. The `<version>` element will reflect the current shipping version of the libraries.

The `.h` files and the DLL of choice (x64 for 64-bit or x86 for 32-bit) should be copied into your source directory. You must also do an `#include` to include the header files. The DLL (x86 or x64) must be copied to your binary path, depending on the architecture that you are targeting.

Note ▪ *If you plan to integrate the Usage Intelligence SDK into a plug-in framework and expect to have plug-ins that are also using Usage Intelligence , there are special considerations on how this integration is to be done. Please contact Usage Intelligence support (support@revenera.com) for more information.*

In order to download the Usage Intelligence SDK, go to the following page in the Revenera Community:

Usage Intelligence SDK Download Links and API Documentation

Basic Integration Steps

The most basic Usage Intelligence (RUI) integration can be accomplished by following the steps below. It is however recommended to read the more advanced documentation as Usage Intelligence can do much more than the basic functionality that can be achieved by following these steps.

Task ***To perform basic integration:***

1. Download the latest SDK from the following page in the Revenera Community, and extract it to your preferred project location:

   Usage Intelligence SDK Download Links and API Documentation

2. Create a Usage Intelligence SDK object. The object returned from the call will be used in all subsequent Usage Intelligence API calls.

   ```
   bool registerAutoReachOut = true;
   RUIINSTANCE* mySDK = ruiCreateInstance(registerAutoReachOut);
   ```

3. Create the configuration point to the directory where the Usage Intelligence SDK will create and update files. The application using Usage Intelligence will need read and write access rights to this directory.

   ```
   RUIRESULT rc = RUI_OK;
   char* myPath = "<path to directory for SDK logging>";
   char* myProductId = "<Product ID>";
   char* myAppName = "<Your App Name>";
   char* myURL = "<CallHome URL>";
   char* myKey = "<Your AES HEX Key>";
   int32_t myProtocol = RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION;
   bool myMultiSessionSetting = false;
   bool myReachOutAutoSyncSetting = true;
   ```

```
rc = ruiCreateConfig(mySDK, myPath, myProductId, myAppName, myURL, myProtocol, myKey,
myMultiSessionSetting, myReachOutAutoSyncSetting);
if (rc != RUI_OK) {
//Your program logic to handle error...
}
```

Note the following:

- The Call Home URL, the Product ID and the AES Key can be retrieved from the RUI Dashboard via your registered account.
- The protocol choice is based on the application and environment needs. Normally, HTTP protocol (port 80) will give applications the greatest chance of success in most environments.
- The Multiple Session flag is a boolean value where you specify whether your application can have multiple user sessions per runtime session. This is normally false.

  Note ▪ *For further details, refer to Single vs. Multiple Session Modes.*

- The ReachOut Auto Sync flag indicates whether or not a ReachOut should be requested as part of each SDK Automatic Sync. A ReachOut request will be made only if a ReachOut handler has been set by registering the default graphical handler, `ruiDestroyInstance()`, or a custom handler, `ruiSetReachOutHandler()`.

4. Initialize the SDK with your product information. This is most conveniently done via the `ruiSetProductData()` call. This must be done BEFORE calling `ruiStartSDK()`.

   ```
   char* myProductEdition = "Professional";
   char* myLanguage = "US English";
   char* myVersion = "5.0.0";
   char* myBuildNumber = "17393";

   rc = ruiSetProductData(mySDK, myProductEdition, myLanguage, myVersion, myBuildNumber);
   ```

5. Initialize the SDK with any optional custom properties by calling the function `ruiSetCustomProperty()`.
6. Call the function `ruiStartSDK()`.

   Note ▪ *You must set all known values for product data and custom properties BEFORE calling* `ruiStartSDK()` *otherwise you risk having null values for fields not specified. Once these calls are completed, you can safely call* `ruiStartSDK()`.

   Before making any other Usage Intelligence API tracking calls, you **MUST** call `ruiStartSDK()`. It is recommended that you place this call at the entry point of your application so the SDK knows exactly at what time your application runtime session was started. If using multi-session mode, you also need to call `ruiStartSession()` when a user session is started, and also provide a unique user session ID that you will then also use for closing the session or for Feature / Event Tracking.

7. Call `ruiStopSDK()` when closing your application so the SDK knows when your application runtime session has been closed.

Important ▪ *You must allow at least 5 seconds of application runtime to allow event data to be written to the log file and synchronized with the Server. If necessary, add a sleep of 5 seconds before calling* `ruiStopSDK()`.

If using multi-session mode, when user sessions are closed, you should call `ruiStopSession()` and send the ID of the session that is being closed as a parameter.

8. All of the other functions in the Usage Intelligence API can be called at any point in your application as long as the Usage Intelligence SDK has been initialized by calling `ruiStartSDK()`.

9. Call `ruiDestroyInstance()` to shut down the SDK and free the memory.

The following is an example of the basic integration outlined below. This example uses single-session mode.

```
//Initialize the Usage Intelligence Configuration
char* myURL="CALLHOME-URL-WITHOUT-PROTOCOL-PREFIX";
char* myProductId="INSERT-YOUR-PROD-ID";
char* myPath="INSERT-YOUR-ABSOLUTE-OR-RELATIVE-FILEPATH";
char* myVersion="1.2.3";
char* myBuildVersion="4567";
bool myMultiSessionSetting = false;
bool myReachOutAutoSyncSetting = true;
char* myKey = "<Your AES HEX Key>";

bool registerAutoReachOut = true;
RUIINSTANCE* mySDK = ruiCreateInstance(registerAutoReachOut);

RUIRESULT rc = RUI_OK;

rc = ruiCreateConfig(mySDK, myPath, myProductId, myAppName, myURL, RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION,
myKey, myMultiSessionSetting, myReachOutAutoSyncSetting);

if (rc != RUI_OK) {
    //Your program logic to handle error...
}

// Set your product data information

char* myProductEdition = "Professional";
char* myLanguage = "US English";
char* myVersion = "5.0.0";
char* myBuildNumber = "17393";

rc = ruiSetProductData(mySDK, myProductEdition, myLanguage, myVersion, myBuildNumber);

if (rc != RUI_OK) {
    //Your program logic to handle error...
}

// If you have any custom properties set them here.

//Inform Usage Intelligence that a new runtime session has been started.
rc = ruiStartSDK(mySDK);

if (rc != RUI_OK) {
    // Your program logic to handle error....
```

```
}

//Your program logic...

//Program closing - inform Usage Intelligence that this runtime session is closing down and sync.
//Must allow at least 5 seconds between ruiStartSDK() and this call. If less than that, add Sleep() call
//to provide enough time to send captured events to Server

rc = ruiStopSDK(mySDK, RUI_SDK_STOP_SYNC_INDEFINITE_WAIT);

if (rc != RUI_OK) {
    // Your program logic to handle error....
}

rc = ruiDestroyInstance(mySDK);

    if (rc != RUI_OK) {
    // Your program logic to handle error....
}

//Your program logic...
```

Text Encoding

In all cases where a text value is used, `char*` is the data type that is used.

Also, if non-ASCII values are returned, they are encoded by the SDK in UTF-8. If a non-ASCII text value needs to be passed as a parameter, it should be encoded in UTF-8 by the user application.

Next Steps

In the above section, we covered the basic integration steps. While these steps would work for most software products, it is recommended to do some further reading in order to get the most of what Usage Intelligence has to offer. It is recommended to go into more detail by reading the pages SDK Initialization and Configuration and Basic SDK Controls. Once you are familiar with the SDK, you may look at the advanced features.

Advanced Features

By following the Basic Integration Steps above, the SDK will be able to collect information about how often users run your product, how long they are engaged with your software as well as which versions and builds they are running. The SDK also collects information on what platforms and architectures your software is being run (i.e. OS versions, language, screen resolution, etc.). Once you have implemented the basic features, you may choose to use Usage Intelligence for more advanced features that include:

- Feature / Event Tracking
- ReachOut Direct-to-Desktop Messaging Service
- Exception Tracking
- License Management

- Custom Properties

3

SDK Initialization and Configuration

The Usage Intelligence V5 SDK is built to support running Usage Intelligence in plug-in type environments. To accomplish this goal, applications must first create an instance of the Usage Intelligence SDK by creating a new Usage Intelligence SDK object and using that object in subsequent calls.

The call to `ruiCreateInstance()` must be paired with a call to `ruiDestroyInstance()` when the client application is done using the SDK. The call to `ruiCreateInstance()` does not configure the SDK, `ruiCreateConfig()`, nor start the SDK, `ruiStartSDK()`.

A typical client will create only a single instance of the SDK. Creating more than one SDK instance is allowed and is used to support clients that are plug-ins or other scenarios whereby multiple independent clients may co-exist in the same executable. Multiple SDK instances perform independently of one another except when sharing a configuration file: `ruiCreateConfig()`.

`ruiCreateInstance()` is a synchronous function, returning when all functionality is completed.

Before an application can start reporting usage to the Usage Intelligence SDK, it must first provide some basic information such as the Product ID and the CallHome URL.

You should always fill in as much accurate and specific detail as possible since this data will be used by the Usage Intelligence Analytics Server to generate the relevant reports. The more (optional) details you fill in about your product and its licensing state, the more filtering and reporting options will be available to you inside the Usage Intelligence dashboard.

- Creating the Usage Intelligence SDK Instance
- Destroying the Usage Intelligence SDK Instance
- Getting SDK Version Information
- Getting the Client ID
- Initializing the Configuration
- Single vs. Multiple Session Modes
- Opt-Out Mechanism
- Providing Further Data

- Changing ReachOut on Autosync Setting
- Proxy Support

Creating the Usage Intelligence SDK Instance

The ruiCreateInstance() function creates the Usage Intelligence SDK instance to be used by the application. All subsequent Usage Intelligence SDK calls use the created instance as a parameter. This function also allocates the internal resources for the Usage Intelligence SDK instance. Calls to ruiCreateConfig() and ruiStartSDK() follow the creation of this object.

ruiCreateInstance()

```
RUIINSTANCE* ruiCreateInstance(bool registerDefaultGraphicalReachOutHandler)
```

Parameters

The ruiCreateInstance() function has the following parameters:

Table 3-1 ▪ ruiCreateInstance() Parameters

Parameter	Description
registerDefaultGraphicalReachOutHandler	(Boolean) On platforms that contain an SDK default graphical ReachOut handler (Windows and macOS), automatically register that handler via ruiSetReachOutHandler().

Returns

The opaque handle to the SDK instance, that is used in other API calls. NULL is returned if there is a creation error.

Destroying the Usage Intelligence SDK Instance

The ruiDestroyInstance() function destroys an instance of the SDK, created with ruiCreateInstance().

The destructor is a synchronous function, returning when all functionality is completed.

ruiDestroyInstance()

```
RUIRESULT ruiDestroyInstance(RUIINSTANCE* ruiInstance)
```

Returns

The ruiDestroyInstance() function returns one of the return status constants below.

Table 3-2 ▪ ruiDestroyInstance() Returns

Return	Description
RUI_OK	Function successful.

Table 3-2 ▪ ruiDestroyInstance() Returns

Return	Description
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.

Getting SDK Version Information

ruiGetSDKVersion() will always allocate memory (regardless of return code), and the client application is responsible for freeing that memory via ruiFree() when the memory is no longer needed.

ruiGetSDKVersion() can be called between ruiCreateInstance() and ruiDestroyInstance(), and can be called more than once.

ruiGetSDKVersion() is a synchronous function, returning when all functionality is completed.

ruiGetSDKVersion()

```
RUIRESULT ruiGetSDKVersion(RUIINSTANCE* ruiInstance, char** sdkVersion)
```

Parameters

The ruiGetSDKVersion() function has the following parameters.

Table 3-3 ▪ ruiGetSDKVersion() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
sdkVersion (char**)	Receives the version string allocated by the SDK; must be freed via ruiFree().

Returns

The ruiGetSDKVersion() function returns one of the return status constants below.

Table 3-4 ▪ ruiGetSDKVersion() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_INVALID_PARAMETER_EXPECTED_NON_NULL	Some API parameter is expected to be non-NULL, and is not.

Getting the Client ID

ruiGetClientId() will always allocate memory (regardless of return code) and the client application is responsible for freeing that memory via ruiFree() when the memory is no longer needed.

ruiGetClientId() can be called between ruiCreateInstance() and ruiDestroyInstance() and can be called more than once.

ruiGetClientId() is a synchronous function returning when all functionality is completed.

ruiGetClientId()

```
RUIRESULT ruiGetClientId(RUIINSTANCE* ruiInstance, char** clientID)
```

Parameters

The ruiGetClientId() function has the following parameters.

Table 3-5 ▪ ruiGetClientId() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
clientId (char**)	Receives the client ID string allocation by the SDK. Must be freed via ruiFree(). Returns the string “0”, if the client ID is not available.

Returns

The ruiGetClientId() function returns one of the return status constants below.

Table 3-6 ▪ ruiGetClientId() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_INVALID_PARAMETER_EXPECTED_NON_NULL	Some API parameter is expected to be non-NULL and is not.

Initializing the Configuration

Before calling any other function other than the ruiCreateInstance() and ruiGetState(), the ruiCreateConfig() function must be called in order to initialize the configuration.

`ruiCreateConfig()` creates a configuration for the SDK instance. A configuration is passed in a file, specified by `configFilePath`, `productID`, and `appName`. If two or more SDK instances, in the same process or in different processes, use the same values for these three parameters, then those SDK instances are bound together through a shared configuration. When multiple different application executables are being used, such usage is generally not desirable nor recommended. Instead, each application executable should use a different `appName` value.

The SDK has two communications modes: HTTPS or HTTP + AES-128 encryption. The mode is configured by protocol. When protocol is `RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION` or `RUI_PROTOCOL_HTTPS_WITH_FALLBACK`, the AES key must be supplied as a 128-bit (32 hex characters) hex-encoded string (aesKeyHex). When protocol is `RUI_PROTOCOL_HTTPS`, then `aesKeyHex` must be empty.

On first execution of a client application, no SDK configuration file will exist. This situation is detected by the SDK and will result in a New Registration message to the Server at `ruiStartSDK()`. Once the configuration is received from the Server, the SDK writes the configuration file, that is then used for subsequent client application executions.

`ruiCreateConfig()` must be called before most other APIs and must only be successfully called once.

`ruiCreateConfig()` is a synchronous function, returning when all functionality is completed.

ruiCreateConfig()

```
RUIRESULT ruiCreateConfig(RUIINSTANCE* ruiInstance, const char* configFilePath, const char* productID,
const char* appName, const char* serverURL, int32_t protocol, const char* aesKeyHex,
bool multiSessionEnabled, bool reachOutOnAutoSync)
```

Parameters

The `ruiCreateConfig()` function has the following parameters.

Table 3-7 ▪ ruiCreateConfig() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`
`configFilePath (const char*)`	The directory to use for the SDK instance's configuration file. Cannot be empty; must exist and be writable.
`productID (const char*)`	The Revenera-supplied product ID for this client; 10 digits.
`appName (const char*)`	The customer-supplied application name for this client, to distinguish suites with the same `productID`. Cannot be empty or contain white space; at most 16 UTF-8 characters. More information about the purpose of the `appName` parameter can be found in the knowledge base article: What is the purpose of the appName parameter when creating config in the SDK?

Table 3-7 ▪ ruiCreateConfig() Parameters

Parameter	Description
`serverURL (const char*)`	The URL to use for New Registrations. Subsequent communications from the SDK will either use this URL or the URL supplied by the Server. Cannot be empty. Cannot have a protocol prefix (e.g., no `http://` or `https://`). This URL is generated automatically on account creation and is used by the SDK to communicate with the Usage Intelligence server. You can get this URL from the Developer Zone once you login to the Usage Intelligence dashboard.
`protocol (int32_t)`	Indicates whether HTTP + AES, HTTPS with fall-back to HTTP + AES, or HTTPS only is used to communicate with the Server. Valid values are: • `RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION` • `RUI_PROTOCOL_HTTPS_WITH_FALLBACK` • `RUI_PROTOCOL_HTTPS`
`aesKeyHex (const char*)`	AES Key to use when protocol includes encryption (`RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION` or `RUI_PROTOCOL_HTTPS_WITH_FALLBACK`); 32 hex chars (16 bytes, 128 bit) key.
`multiSessionEnabled (bool)`	Indicates whether or not the client will explicitly manage `sessionIDs` via `ruiStartSession()` and `ruiStopSession()`, and supply those `sessionIDs` to the various event tracking APIs.
`reachOutOnAutoSync (bool)`	Indicates whether or not a ReachOut should be requested as part of each SDK Automatic Sync. A ReachOut request will be made only if a ReachOut handler has been set by registering the default graphical handler (`ruiCreateInstance()`) or a custom handler (`ruiSetReachOutHandler()`). This value may be changed at runtime using the call `ruiSetReachOutOnAutoSync()`.

Returns

The `ruiCreateConfig()` function returns one of the return status constants below.

Table 3-8 ▪ ruiCreateConfig() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.

Table 3-8 ▪ ruiCreateConfig() Returns

Return	Description
RUI_CONFIG_ALREADY_CREATED	Configuration has already been successfully created.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
RUI_INVALID_PARAMETER_EXPECTED_NO_WHITESPACE	Some API parameter is expected to be free of white space, and is not.
RUI_INVALID_PARAMETER_TOO_LONG	Some API parameter violates its allowable maximum length.
RUI_INVALID_CONFIG_PATH	The configFilePath is not a well-formed directory name.
RUI_INVALID_CONFIG_PATH_NONEXISTENT_DIR	The configFilePath identifies a directory that does not exist.
RUI_INVALID_CONFIG_PATH_NOT_WRITABLE	The configFilePath identifies a directory that is not writable.
RUI_INVALID_PRODUCT_ID	The productID is not a well-formed Product ID.
RUI_INVALID_SERVER_URL	The serverURL is not a well-formed URL.
RUI_INVALID_PROTOCOL	The protocol is not a legal value.
RUI_INVALID_AES_KEY_EXPECTED_EMPTY	The AES Key is expected to be NULL/empty, and it's not.
RUI_INVALID_AES_KEY_LENGTH	The AES Key is not the expected length (32 hex chars).
RUI_INVALID_AES_KEY_FORMAT	The AES Key is not valid hex encoding.

Code Example

The following example shows how to initialize the Usage Intelligence configuration:

```
// ruiCreateConfig example
bool useDefaultReachOutHandler = false;

RUIINSTANCE* mySDK = ruiCreateInstance(useDefaultReachOutHandler);

char* myPath = "PATH-TO-YOUR-CONFIG-FILE";
char* myURL = "CALLHOME-URL-WITHOUT-PROTOCOL-PREFIX";
char* myProductId = "INSERT-YOUR-PROD-ID";
char* myAppName = "MyApplication";
char* myKey = "0123456789abcdeffedcba9876543210";
bool multiSessionEnabled = false;
bool reachOutOnAutoSync = false;
```

```
RUIRESULT result = ruiCreateConfig(mySDK, myPath, myProductId, myAppName, myURL,
RUN_PROTOCOL_HTTP_PLUS_ENCRYPTION, myKey, multiSessionEnabled, reachOutOnAutoSync);
```

Single vs. Multiple Session Modes

In desktop software, a single application instance would normally have only one single user session. This means that such an application would only show one window (or set of windows) to a single user and interaction is done with that single user. If the user would like to use two different sessions, two instances of the application would have to be loaded that would not affect each other. In such cases, you should use the single session mode, which handles user sessions automatically and assumes that one process (instance) means one user session.

The multiple session mode needs to be used in multi-user applications, especially applications that have web interfaces. In such applications, a number of users might be using the same application process simultaneously. In such cases, you need to manually tell the Usage Intelligence SDK must be notified when user sessions start and stop, and also how to link events (see Feature / Event Tracking) to user sessions.

To do this, when starting or stopping a user session, the methods `ruiStartSession()` and `ruiStopSession()` should be used, and when tracking events on a per user basis, a session ID needs to be passed as a parameter.

Opt-Out Mechanism

Starting from version 5.1.0, a new opt-out mechanism was introduced. Using this mechanism, if a user does not want to send tracking information to Revenera, the function `ruiOptOut()` must be called after calling `ruiCreateConfig()` and before `ruiStartSDK()`.

The `ruiOptOut()` function Instructs the SDK to send a message to the server to indicate that this user is opting-out (if not already sent in previous sessions) and disables all further functionality and communication with the server.

When `ruiOptOut()` is called, the SDK sends a message to the server after startup (`ruiStartSDK()`). This message informs the server that this client has opted-out and the server will register the opt-out. This message is only sent to the server once. The opt-out flag on the server will be used for reporting opt-out statistics only. The SDK will send no further information to the server as long as the user is opted-out.

If `ruiOptOut()` is called before a new registration, the server will never have any data about that installation. If `ruiOptOut()` is called for an installation that was already being tracked, the server will still contain the data that had been collected in the past and no past data is deleted.

The application must keep calling `ruiOptOut()` before every startup as long as the user wants to stay opted-out. If this function is not called, then the SDK assumes that the user is opting-in again and will start tracking normally.

Note that when an installation is not opted-out, it communicates with the server immediately on calling `ruiStartSDK()`. At this point, the SDK attempts to sync data regarding past application and event usage that had not been synced yet, and also system and product information such as OS version, CPU, GPU, product version, product edition, etc.

ruiOptOut()

```
RUIRESULT ruiOptOut(RUIINSTANCE* ruiInstance)
```

Parameters

The `ruiOptOut()` function has the following parameters.

Table 3-9 ▪ ruiOptOut()Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.

Returns

The `ruiOptOut()` function returns one of the return status constants below.

Table 3-10 ▪ ruiOptOut() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_ALREADY_STARTED`	SDK has already been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Providing Further Data

The Usage Intelligence SDK V5 requires that the application provide product data every time the SDK instance is run. In addition you can optionally set License data for the application. Finally, if you are using proxies to access the Internet, there is a function to set up the required information for connecting through that proxy.

- Product Details
- License Management

Product Details

The following functions are available to set product data:

- Setting Product Data

- Setting Product Edition
- Setting Product Language
- Setting Product Version
- Setting Product Build Number

Setting Product Data

The `ruiSetProductData()` function sets or clears the product data.

***Note** ▪ The product data must be set every time the SDK instance is run.*

***Note** ▪ This is different than V4 of the RUI (Trackerbird) SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`ruiSetProductData()` can be called between `ruiCreateConfig()` and `ruiStopSDK()` and can be called zero or more times.

`ruiSetProductData()` is a synchronous function returning when all functionality is completed.

ruiSetProductData()

```
RUIRESULT ruiSetProductData(RUIINSTANCE* ruiInstance, const char* productEdition, const
char* productLanguage, const char* productVersion, const char* productBuildNumber)
```

Parameters

The `ruiSetProductData()` function has the following parameters.

Table 3-11 ▪ `ruiSetProductData()` Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`productEdition (const char*)`	The product edition that is to be set. Maximum length of 128 characters.  ***Note** ▪ A NULL value removes any previous value*
`productLanguage (const char*)`	The product language that is to be set. Maximum length of 128 characters.  ***Note** ▪ A NULL value removes any previous value*

Table 3-11 ▪ `ruiSetProductData()` Parameters

Parameter	Description
`productVersion (const char*)`	The product version that is to be set. Maximum length of 128 characters.  ***Note ▪*** *A NULL value removes any previous value*
`productBuildNumber (const char*)`	The product build number that is to be set. Maximum length of 128 characters.  ***Note ▪*** *A NULL value removes any previous value*

Returns

The `ruiSetProductData()` function returns one of the return status constants below.

Table 3-12 ▪ `ruiSetProductData()` Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Setting Product Edition

The `ruiSetProductEdition()` function allows you to set the edition of your product. An example of this would be when a single product can be licensed/run in different modes such as “Home” and “Business”.

ruiSetProductEdition()

```
RUIRESULT ruiSetProductEdition(RUIINSTANCE* ruiInstance, const char* productEdition)
```

Parameters

The ruiSetProductEdition() function has the following parameters.

Table 3-13 ▪ ruiSetProductEdition()Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
productEdition (const char*)	The product edition that is to be set. Maximum length of 128 characters.  ***Note*** ▪ *A NULL value removes any previous value*

Returns

The ruiSetProductEdition() function returns one of the return status constants below.

Table 3-14 ▪ ruiSetProductEdition() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Language

The ruiSetProductLanguage() function allows you to set the language that the client is viewing your product. This is useful for products that have been internationalized, so you can determine how many installations are running your software in a particular language.

***Note** ▪ This is different than the OS language that is collected automatically by the Usage Intelligence SDK.*

ruiSetProductLanguage()

```
RUIRESULT ruiSetProductLanguage(RUIINSTANCE* ruiInstance, const char* productLanguage)
```

Parameters

The `ruiSetProductLanguage()` function has the following parameters.

Table 3-15 ▪ ruiSetProductLanguage() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`productLanguage (const char*)`	The product language that is to be set. Maximum length of 128 characters.  ***Note** ▪ A NULL value removes any previous value.*

Returns

The `ruiSetProductLanguage()` function returns one of the return status constants below.

Table 3-16 ▪ ruiSetProductLanguage() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Setting Product Version

The ruiSetProductVersion() function is used to set the version of the application being run.

ruiSetProductVersion()

```
RUIRESULT ruiSetProductVersion(RUIINSTANCE* ruiInstance, const char* productVersion)
```

Parameters

The ruiSetProductVersion() function has the following parameters.

Table 3-17 ▪ ruiSetProductVersion() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
productVersion (const char*)	The product version that is to be set. Maximum length of 128 characters.  ***Note ▪*** *A NULL value removes any previous value.*

Returns

The ruiSetProductVersion() function returns one of the return status constants below.

Table 3-18 ▪ ruiSetProductVersion() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Setting Product Build Number

The ruiSetProductBuildNumber() function is used to set the build number of the application being run.

ruiSetProductBuildNumber()

```
RUIRESULT ruiSetProductBuildNumber(RUIINSTANCE* ruiInstance, const char* productBuildNumber)
```

Parameters

The ruiSetProductBuildNumber() function has the following parameters.

Table 3-19 ▪ ruiSetProductBuildNumber() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
productBuildNumber (const char*)	The product build number that is to be set. Maximum length of 128 characters.  ***Note ▪*** *A NULL value removes any previous value*

Returns

The ruiSetProductBuildNumber() function returns one of the return status constants below.

Table 3-20 ▪ ruiSetProductBuildNumber() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

License Management

The ruiSetLicenseData() function sets or clears the license data. The legal parameter values include RUI_KEY_STATUS_UNCHANGED (-1).

Note ▪ *Different from the V4 of the Usage Intelligence SDK, a sessionID parameter can be supplied.*

The ruiSetLicenseData() function can be called between ruiCreateConfig() and ruiStopSDK() and can be called zero or more times. However, the usage requirements of the sessionID parameter are different if ruiSetLicenseData() is called before ruiStartSDK() or called after ruiStartSDK(). See sessionId for more information.

The ruiSetLicenseData() function can be called while a New Registration is being performed (ruiCreateConfig(), ruiStartSDK()). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The ruiSetLicenseData() function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

ruiSetLicenseData()

```
RUIRESULT ruiSetLicenseData(RUIINSTANCE* ruiInstance, int32_t keyType, int32_t keyExpired,
int32_t keyActivated, int32_t keyBlocked, int32_t keyAllowed, const char* sessionID)
```

Parameters

The ruiSetLicenseData() function has the following parameters.

Table 3-21 ▪ ruiSetLicenseData() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance()
keyType (int32_t)	The keyType parameter can be set to any of the constant types below. RUI_KEY_TYPE_UNCHANGED (-1) RUI_KEY_TYPE_EVALUATION (0) RUI_KEY_TYPE_PURCHASED (1) RUI_KEY_TYPE_FREEWARE (2) RUI_KEY_TYPE_UNKNOWN (3) RUI_KEY_TYPE_NFR (4) RUI_KEY_TYPE_CUSTOM1 (5) RUI_KEY_TYPE_CUSTOM2 (6) RUI_KEY_TYPE_CUSTOM3 (7)  ***Note*** ▪ *The three custom values may be used freely to denote your own custom license types.*

Table 3-21 ▪ ruiSetLicenseData() Parameters

Parameter	Description

`keyExpired (int32_t)`	Indicates whether the client license has expired. One of the values below:
	
	`RUI_KEY_STATUS_UNCHANGED (-1)`
	`RUI_KEY_STATUS_NO (0)`
	`RUI_KEY_STATUS_YES (1)`

`keyActivated (int32_t)`	Indicates whether the client license has been activated. One of the values below:
	
	`RUI_KEY_STATUS_UNCHANGED (-1)`
	`RUI_KEY_STATUS_NO (0)`
	`RUI_KEY_STATUS_YES (1)`

`keyBlocked (int32_t)`	Indicates whether the client license key has been blocked. One of the values below:
	
	`RUI_KEY_STATUS_UNCHANGED (-1)`
	`RUI_KEY_STATUS_NO (0)`
	`RUI_KEY_STATUS_YES (1)`

`keyAllowed (int32_t)`	Indicates whether the client license key has been allowed. One of the values below:
	
	`RUI_KEY_STATUS_UNCHANGED (-1)`
	`RUI_KEY_STATUS_NO (0)`
	`RUI_KEY_STATUS_YES (1)`

`sessionId (const char*)`	Optional `SessionId` parameter to associate keys to a specific session. NULL if not used. The usage requirements of the `sessionID` parameter are different if `ruiSetLicenseData()` is called before `ruiStartSDK()` or called after `ruiStartSDK()`. • **Called before startSDK (regardless of multiSessionEnabled)**—`sessionID` must be empty. • **Called after startSDK and multiSessionEnabled is set to false**—`sessionID` must be empty. This is similar to event tracking APIs. • **Called after startSDK and multiSessionEnabled is set to true**—`sessionID` must be a current valid value used in `ruiStartSession()`, or it can be empty. This is different than normal event tracking APIs, whereby a empty value is not allowed.

Returns

The ruiSetLicenseData() function returns one of the return status constants below.

Table 3-22 ▪ ruiSetLicenseData() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_SESSION_ID_EXPECTED_EMPTY	The sessionID is expected to be empty, and it was not.
RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_NOT_ACTIVE	The sessionID is not currently in use.

Changing ReachOut on Autosync Setting

The flag to determine whether or not a ReachOut should be requested as part of each SDK Automatic Sync is initially set in the ruiCreateConfig() call. There may be certain cases when the application wants to either enable or disable this functionality during the application lifetime.

The ruiSetReachOutOnAutoSync() function allows the application to enable or disable this capability after ruiCreateConfig() has been called.

The ruiSetReachOutOnAutoSync() function enables (true) or disables (false) the ReachOut on Autosync capability. Note if the call does not change the existing setting, the API will still return RUI_OK.

The ruiSetReachOutOnAutoSync() function can be called between ruiCreateConfig() and ruiStopSDK() and can be called zero or more times.

ruiSetReachOutOnAutoSync()

```
RUIRESULT ruiSetReachOutOnAutoSync(RUIINSTANCE* ruiInstance, bool reachOutOnAutoSyncSetting)
```

Parameters

The ruiSetReachOutOnAutoSync() function has the following parameters.

Table 3-23 ▪ ruiSetReachOutOnAutoSync() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
reachOutOnAutoSyncSetting (bool*)	Enable (true) or disable (false) the ReachOut on Autosync capability.

Returns

The ruiSetReachOutOnAutoSync() function returns one of the return status constants below.

Table 3-24 ▪ ruiSetReachOutOnAutoSync() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Proxy Support

The Usage Intelligence SDK V5 library supports communications through HTTP proxy servers on all major operating system types: Windows, Linux, and macOS. Application developers are responsible for obtaining proxy credentials (if the proxy requires it) and setting those credentials in the SDK so communications can use the credentials for the proxy. The function `ruiSetProxy()` handles setting and clearing the proxy related information.

The `ruiSetProxy()` function sets or clears the data to be used with a proxy. If there is no proxy between the SDK and the Server, there is no need to use this function. The address can be either empty (for transparent proxy servers) or non-empty. The username and password must both be empty (non-authenticating proxy) or both be non-empty (authenticating proxy). The port is only used for non-transparent proxy servers, hence port must be zero if address is empty, otherwise port must be non-zero.

`ruiSetProxy()` can be called between `ruiCreateConfig()` and `ruiStopSDK()`, and can be called zero or more times.

`ruiSetProxy()` is a synchronous function, returning when all functionality is completed.

ruiSetProxy()

```
RUIRESULT ruiSetProxy(RUIINSTANCE* ruiInstance, const char* address, uint16_t port, const
char* username, const char* password)
```

Permitted Parameter Combinations

The SDK uses the proxy data in multiple ways to attempt to communicate via a proxy. The allowed parameter combinations and their usage are as follows:

Table 3-25 ▪ ruiSetProxy() Permitted Parameter Combinations

address	port	username	password	Description
empty	0	empty	empty	Resets the proxy data to its initial state, no proxy server is used, and the Server is contacted directly.
non-empty	not 0	empty	empty	Identifies a non-authenticating non-transparent proxy that will be used unless communications fails, then falling back to using no proxy.
empty	0	non-empty	non-empty	Identifies an authenticating transparent proxy that will be used unless communications fails, then falling back to using no proxy.
non-empty	not 0	non-empty	non-empty	Identifies an non-transparent authenticating proxy that will be used unless communications fails, then falling back to using an authenticating transparent proxy, then falling back to using no proxy.

Parameters

The ruiSetProxy() function has the following parameters.

Table 3-26 ▪ ruiSetProxy() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
address (const char*)	The server name or IP address (dot notation) for the proxy server.
port (uint16_t)	The port for the proxy server; only used with non-transparent proxy, port != 0 if and only if address non-empty.
username (const char*)	The proxy username; username and password must both be empty or both be non-empty.
password (const char*)	The proxy password; username and password must both be empty or both be non-empty.

Returns

The ruiSetProxy() function returns one of the return status constants below.

Table 3-27 ▪ ruiSetProxy() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_INVALID_PROXY_CREDENTIALS	The proxy username and password are not an allowable combination.
RUI_INVALID_PROXY_PORT	The proxy port was not valid.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

4

Basic SDK Controls

Once the required configuration is initialized (explained in SDK Initialization and Configuration) and set according to the needs of your application, you may inform the SDK that the application has started. This will allow you to use further functions that expect the application to be running such as `ruiCheckLicenseKey()` and `ruiCheckForReachOut()`.

- Starting the SDK
- Stopping the SDK
- Starting a Session
- Stopping a Session
- Caching and Synchronizing

Starting the SDK

The ruiStartSDK() function starts the SDK. `ruiStartSDK()` must be paired with a call to `ruiStopSDK()`.

After the SDK is started, the various event tracking APIs are available to be used. If `ruiCreateConfig()` did not detect a configuration file, `ruiStartSDK()` will perform a New Registration with the Server. Until a New Registration is complete, the SDK will not be able to save event data to a log file or perform synchronization with the Server. A successful New Registration (or presence of a configuration file) will put the SDK into a normal running state, whereby events are saved to a log file, automatic and manual synchronizations with the Server are possible, and getting ReachOut campaigns from the Server are possible. A failed New Registration will put the SDK into an aborted state, not allowing further activity.

`ruiStartSDK()` must be called after `ruiCreateConfig()`, and must be called only once.

`ruiStartSDK()` is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

If `ruiOptOut()` is called before a new registration has been done for a user, the SDK will not sync any system and product information and no data is recorded for the user. The SDK will inform the server once that there is an opted out user for reporting opt-out statistics only.

ruiStartSDK()

```
RUIRESULT ruiStartSDK(RUIINSTANCE* ruiInstance)
```

Parameters

The ruiStartSDK() function has the following parameters.

Table 4-1 ▪ ruiStartSDK() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().

Returns

The ruiStartSDK() function returns one of the return status constants below.

Table 4-2 ▪ ruiStartSDK() Returns

Return	Description
RUI_OK	Synchronous functionality successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STARTED	SDK has already been successfully started.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Stopping the SDK

The ruiStopSDK() function stops the SDK that was started with ruiStartSDK(). This function should always be called at the exit point of your application.

If explicit sessions are allowed (multiSessionsEnabled = true in ruiCreateConfig()), then any sessions that have been started with ruiStartSession() that have not been stopped with ruiStopSession() are automatically stopped. A manual synchronization with the Server, ruiSync(), will be performed at stop depending on the value of doSync (as described in Parameters).

ruiStopSDK() must be called after ruiStartSDK() and must be called only once. After ruiStopSDK() is called, the various event tracking APIs are no longer available. The only APIs available are ruiGetState() and the ruiDestroyInstance(). The SDK cannot be re-started with a subsequent call to ruiStartSDK().

`ruiStopSDK()` is a synchronous function, including the manual synchronization with the Server (if requested), returning when all functionality is completed.

ruiStopSDK()

```
RUIRESULT ruiStopSDK(RUIINSTANCE* ruiInstance, int32_t doSync)
```

Parameters

The `ruiStopSDK()` function has the following parameters.

Table 4-3 ▪ ruiStopSDK() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`
`doSync (int32_t)`	Indicates whether to do a manual synchronization as part of the stop, and if so, the wait limit. A manual synchronization with the Server, `ruiSync()`, will be performed at stop depending on the value of `doSync`: • **1**—Do not perform a manual synchronization with the Server as part of the stop. • **0**—Perform a manual synchronization with the Server as part of the stop; wait indefinitely for completion. • **>0**—Perform a manual synchronization with the Server as part of the stop; wait only `doSync` seconds for completion.

Returns

The `ruiStopSDK()` function returns one of the return status constants below.

Table 4-4 ▪ ruiStopSDK() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_NOT_STARTED`	SDK has not been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Table 4-4 ▪ ruiStopSDK() Returns

Return	Description
RUI_INVALID_DO_SYNC_VALUE	The doSync manual sync flag/limit violates its allowable range.

Starting a Session

The ruiStartSession() function starts an explicit session for event tracking in the SDK. It must be paired with a call to ruiStopSession().

Explicit sessions are allowed only if ruiCreateConfig() was called with multiSessionEnabled = true. When explicit sessions are enabled, a valid sessionID becomes a required parameter to the event tracking APIs, as described in Parameters.

ruiStartSession() can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

ruiStartSession() is a synchronous function, returning when all functionality is completed.

ruiStartSession()

```
RUIRESULT ruiStartSession(RUIINSTANCE* ruiInstance, const char* sessionID)
```

Parameters

The ruiStartSession() function has the following parameters.

Table 4-5 ▪ ruiStartSession() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().

Table 4-5 ▪ ruiStartSession() Parameters

Parameter	Description
`sessionID (const char*)`	This parameter should contain a unique ID that refers to the user session that is being started. This same ID should later be used for event tracking. The content of a `sessionID` is conditioned and validated (after conditioning) with the following rules: ● **Conditioning**—All leading white space is removed. ● **Conditioning**—All trailing white space is removed. ● **Conditioning**—All internal white spaces other than space characters (' ') are removed. ● **Validation**—Cannot be shorter than 10 UTF-8 characters. ● **Validation**—Cannot be longer than 64 UTF-8 characters. The resulting conditioned and validated `sessionID` must be unique (i.e. not already in use).  ***Note*** ▪ *With the above conditioning, two `sessionIDs` that differ only by white space or after the 64th character, will not be unique. A `sessionID` should not be re-used for different sessions.*

Returns

The `ruiStartSession()` function returns one of the return status constants below.

Table 4-6 ▪ ruiStartSession() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.

Table 4-6 ▪ ruiStartSession() Returns

Return	Description
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_NOT_STARTED	SDK has not been successfully started.
RUI_FUNCTION_NOT_AVAIL	Function is not available.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_ALREADY_ACTIVE	The sessionID is already currently in use.

Stopping a Session

The ruiStopSession() function stops an explicit session started with ruiStartSession().

Explicit sessions are allowed only if ruiCreateConfig() was called with multiSessionEnabled = true. Any explicit sessions not ended with a call to ruiStopSession() are automatically ended when ruiStopSDK() is called. In case this method is never called, eventually this session will be considered as “timed-out”, and the time of the last recorded event will be assumed to be the time when the last event was recorded.

ruiStopSession() can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

ruiStopSession() is a synchronous function, returning when all functionality is completed.

ruiStopSession()

```
RUIRESULT ruiStopSession(RUIINSTANCE* ruiInstance, const char* sessionID)
```

Parameters

The ruiStopSession() function has the following parameters.

Table 4-7 ▪ ruiStopSession() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
sessionID (const char*)	A current valid session ID to be stopped (content conditioning and validation rules in ruiStartSession()).

Returns

The `ruiStopSession()` function returns one of the return status constants below.

Table 4-8 ▪ ruiStopSession() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_NOT_STARTED`	SDK has not been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY`	The `sessionID` is expected to be non-empty, and it was not.
`RUI_INVALID_SESSION_ID_TOO_SHORT`	The `sessionID` violates its allowable minimum length.
`RUI_INVALID_SESSION_ID_TOO_LONG`	The `sessionID` violates its allowable maximum length.
`RUI_INVALID_SESSION_ID_NOT_ACTIVE`	The `sessionID` is not currently in use.

Caching and Synchronizing

The Usage Intelligence SDK was designed to minimize network traffic and load on the end user's machine. In order to do this, all the collected architecture info and runtime tracking data is cached locally and then compressed and sent to the Usage Intelligence server in batches at various intervals whenever appropriate. By default, log data is sent once for every runtime session (during `ruiStartSDK()`) and every 20 minutes after that while the application is running.

Data may be sent via HTTP (port 80) or HTTPS (port 443) depending on application preference. When data is sent over HTTP, AES encryption is used to encrypt the data payload. When data is sent on HTTPS, normal HTTPS security measures are used. The application may also choose to start with HTTPS communication and if blocked or unsuccessful the SDK will fall back to using encrypted HTTP.

- Forced Synchronization

Forced Synchronization

Under normal conditions, you do not need to instruct the Usage Intelligence SDK when to synchronize with the cloud server, since this happens automatically and is triggered by application interaction with the API. In a typical runtime session, the SDK will always attempt to synchronize with the server at least once whenever the application calls `ruiStartSDK()`. For long running applications, the SDK will periodically sync with the server every 20 minutes.

For applications that require a more customized synchronization, the API also provides an option to request manual synchronization of all cached data. This is done by calling the `ruiSync()` function.

The ruiSync() function performs a manual synchronization with the Server. In normal operation, the SDK periodically performs automatic synchronizations with the Server. `ruiSync()` provides the client an ability to explicitly synchronize with the Server on demand. The manual synchronization can request a ReachOut with `getReachOut`.

Note ▪ *Similar to the parameter `reachOutOnAutoSync` (on function `ruiCreateConfig()`), the ReachOut will not be requested if there is no registered handler (`ruiCreateInstance()` and `ruiSetReachOutHandler()`).*

`ruiSync()` can be called between `ruiStartSDK()` and `ruiStopSDK()` and can be called zero or more times.

Note ▪ *`ruiSync()` will not be successful if a New Registration is in progress (i.e. `ruiCreateConfig()` and `ruiStartSDK()`). A manual synchronization with the Server can be associated with `ruiStopSDK()`.*

`ruiSync()` is an asynchronous function returning immediately with further functionality executed on separate thread(s).

As explained in the previous section, this function is not normally required and should be avoided in most cases. Both the SDK and the Server can reject a `ruiSync()` request from occurring even if this is requested by the developer. This is done to prevent abuse and unnecessary server load if this function is called too frequently.

ruiSync()

```
RUIRESULT ruiSync(RUIINSTANCE* ruiInstance, bool getReachOut)
```

Parameters

The `ruiSync()` function has the following parameters.

Table 4-9 ▪ ruiSync() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`getReachout (bool)`	This optional parameter instructs the server whether to send a ReachOut message during this particular sync if available.

Returns

The `ruiSync()` function returns one of the return status constants below.

Table 4-10 ▪ ruiSync() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_NOT_STARTED`	SDK has not been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`RUI_TIME_THRESHOLD_NOT_REACHED`	The API call frequency threshold (set by the Server) has not been reached. Sync not allowed at this time.

5

Feature / Event Tracking

Through event tracking, Usage Intelligence allows you keep track of how your clients are interacting with the various features within your application, potentially identifying how often every single feature is being used by various user groups. Apart from monitoring feature usage, you can also keep track of how often an event happens - such as how often an auto save has been made on average for every hour your application was running. This is accomplished through `ruiTrackEvent()`.

You may also keep a numeric value, a text value, or a collection of name/value String pairs every time an event is reported. This can be used, for example in the case of `ruiTrackEventNumeric()`, to keep track of the length of time it took to save a file, or the file size that was saved, etc. These events can be recorded using the functions `ruiTrackEventNumeric()`, `ruiTrackEventText()`, and `ruiTrackEventCustom()` respectively.

Once event-related data has been collected, you will be able to identify trends of the features that are most used during evaluation and whether this trend changes once users switch to a freeware or purchased license or once they update to a different version/product build. You will also be able to compare whether any UI tweaks in a particular version or build number had any effect on exposing a particular feature or whether changes in the actual functionality make a feature more or less popular with users. This tool provides excellent insight for A/B testing whereas you can compare the outcome from different builds to improve the end user experience.

- Tracking an Event
- Logging a Normal Event with a Numeric Field
- Logging a Normal Event with a String Field
- Logging a Custom Event

Note ▪ *Event Tracking should NOT be used to track the occurrence of exceptions since there is another specific API call for this purpose. If you need to track exceptions, refer to Exception Tracking.*

Tracking an Event

The ruiTrackEvent() feature enables you keep track of how your clients are interacting with the various features within your application, potentially identifying how often every single feature is being used by various user groups. Apart from monitoring feature usage, you can also keep track of how often an event happens - such as how often an auto save has been made on average for every hour your application was running.

The ruiTrackEvent() feature logs a normal event with the supplied data.

ruiTrackEvent() can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

ruiTrackEvent() can be called while a New Registration is being performed (ruiCreateConfig(), ruiStartSDK()). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

ruiTrackEvent() is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

ruiTrackEvent()

```
RUIRESULT ruiTrackEvent(RUIINSTANCE* ruiInstance, const char* eventCategory, const char* eventName,
const char* sessionID)
```

Parameters

The ruiTrackEvent() function has the following parameters.

Table 5-1 ▪ ruiTrackEvent() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
eventCategory (const char*)	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for eventCategory and eventName). The content of eventCategory and eventName is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—eventCategory can be empty; eventName cannot be empty.

Table 5-1 ▪ ruiTrackEvent() Parameters

Parameter	Description
`eventName (const char*)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: ● **Conditioning**—All leading white space is removed. ● **Conditioning**—All trailing white space is removed. ● **Conditioning**—All internal white spaces other than space characters (' ') are removed. ● **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. ● **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`sessionID (const char*)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `ruiTrackEvent()` function returns one of the return status constants below.

Table 5-2 ▪ ruiTrackEvent() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.

Table 5-2 ▪ ruiTrackEvent() Returns

Return	Description
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_NOT_STARTED	SDK has not been successfully started.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_SESSION_ID_EXPECTED_EMPTY	The sessionID is expected to be empty, and it was not.
RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_NOT_ACTIVE	Parameter validation: The sessionID is not currently in use.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Parameter validation: Some API parameter is expected to be non-empty, and is not.

Logging a Normal Event with a Numeric Field

You may keep a numeric value, a text value, or a collection of name/value string pairs every time an event is reported. This can be used, for example in the case of ruiTrackEventNumeric(), to keep track of the length of time it took to save a file, or the file size that was saved, etc. These events can be recorded using the functions ruiTrackEventNumeric(), ruiTrackEventText(), and ruiTrackEventCustom() respectively.

The ruiTrackEventNumeric() function logs a normal event with the supplied data, including a custom numeric field.

The ruiTrackEventNumeric() function can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

The ruiTrackEventNumeric() function can be called while a New Registration is being performed (ruiCreateConfig(), ruiStartSDK()). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The ruiTrackEventNumeric() function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

ruiTrackEventNumeric()

```
RUIRESULT ruiTrackEventNumeric(RUIINSTANCE* ruiInstance, const char* eventCategory, const
char* eventName, double customValue, const char* sessionID)
```

Parameters

The `ruiTrackEventNumeric()` function has the following parameters.

Table 5-3 ▪ ruiTrackEventNumeric() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`eventCategory (const char*)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`eventName (const char*)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`customValue (double)`	Custom numeric data associated with the event.
`sessionID (const char*)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `ruiTrackEventNumeric()` function returns one of the return status constants below.

Table 5-4 ▪ ruiTrackEventNumeric() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_NOT_STARTED`	SDK has not been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`RUI_INVALID_SESSION_ID_EXPECTED_EMPTY`	The `sessionID` is expected to be empty, and it was not.
`RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY`	The `sessionID` is expected to be non-empty, and it was not.
`RUI_INVALID_SESSION_ID_TOO_SHORT`	The `sessionID` violates its allowable minimum length.
`RUI_INVALID_SESSION_ID_TOO_LONG`	The `sessionID` violates its allowable maximum length.
`RUI_INVALID_SESSION_ID_NOT_ACTIVE`	Parameter validation: The `sessionID` is not currently in use.
`RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY`	Parameter validation: Some API parameter is expected to be non-empty, and is not.

Logging a Normal Event with a String Field

You may keep a numeric value, a text value, or a collection of name/value string pairs every time an event is reported. This can be used, for example in the case of `ruiTrackEventNumeric()`, to keep track of the length of time it took to save a file, or the file size that was saved, etc. These events can be recorded using the functions `ruiTrackEventNumeric()`, `ruiTrackEventText()`, and `ruiTrackEventCustom()` respectively.

The `ruiTrackEventText()` function logs a normal event with the supplied data, including a custom string field.

The `ruiTrackEventText()` function can be called between `ruiStartSDK()` and `ruiStopSDK()`, and can be called zero or more times.

The `ruiTrackEventText()` function can be called while a New Registration is being performed (`ruiCreateConfig()`, `ruiStartSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The `ruiTrackEventText()` function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

ruiTrackEventText()

```
RUIRESULT ruiTrackEventText(RUIINSTANCE* ruiInstance, const char* eventCategory, const char* eventName,
const char* customValue, const char* sessionID)
```

Parameters

The `ruiTrackEventText()` function has the following parameters.

Table 5-5 ▪ ruiTrackEventText() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`eventCategory (const char*)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.

Table 5-5 ▪ ruiTrackEventText() Parameters

Parameter	Description
`eventName (const char*)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: • **Conditioning**—All leading white space is removed. • **Conditioning**—All trailing white space is removed. • **Conditioning**—All internal white spaces other than space characters (' ') are removed. • **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. • **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`customValue (const char*)`	Custom text data associated with the event, cannot be empty. Trimmed to a maximum length determined by the Server. Current default maximum is 4096 UTF-8 characters.
`sessionID (const char*)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `ruiTrackEventText()` function returns one of the return status constants below.

Table 5-6 ▪ ruiTrackEventText() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.

Table 5-6 ▪ ruiTrackEventText() Returns

Return	Description
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_NOT_STARTED	SDK has not been successfully started.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_SESSION_ID_EXPECTED_EMPTY	The sessionID is expected to be empty, and it was not.
RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_NOT_ACTIVE	Parameter validation: The sessionID is not currently in use.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Parameter validation: Some API parameter is expected to be non-empty, and is not.

Logging a Custom Event

You may keep a numeric value, a text value, or a collection of name/value string pairs every time an event is reported. This can be used, for example in the case of ruiTrackEventNumeric(), to keep track of the length of time it took to save a file, or the file size that was saved, etc. These events can be recorded using the functions ruiTrackEventNumeric(), ruiTrackEventText(), and ruiTrackEventCustom() respectively.

The ruiTrackEventCustom() function logs a normal event with the supplied data, including an array of custom name/value pairs.

The ruiTrackEventCustom() function can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

The ruiTrackEventCustom() function can be called while a New Registration is being performed (ruiCreateConfig(), ruiStartSDK()). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

The ruiTrackEventCustom() function is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

The name/value pairs are supplied in a struct of type RUINAMEVALUEPAIR. The struct contains two fields:

```
const char* name;
const char* value;
```

Note ▪ *Custom data will be logged in the format* `(Key1,Value1)&&(Key2,Value2)…&&(KeyN,ValueN)`.

ruiTrackEventCustom()

```
RUIRESULT ruiTrackEventCustom(RUIINSTANCE* ruiInstance, const char* eventCategory, const
char* eventName, RUINAMEVALUEPAIR* customValue, uint32_t numValues, const char* sessionID)
```

Parameters

The `ruiTrackEventCustom()` function has the following parameters.

Table 5-7 ▪ ruiTrackEventCustom() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*`	Pointer to the RUI instance created via `ruiCreateInstance()`.
`eventCategory (const char*)`	The name of the category that this event forms part of. This parameter is optional (send empty string if not required). Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: ● **Conditioning**—All leading white space is removed. ● **Conditioning**—All trailing white space is removed. ● **Conditioning**—All internal white spaces other than space characters (' ') are removed. ● **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. ● **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.
`eventName (const char*)`	The name of the event to be tracked. Unlike V4 of the Usage Intelligence SDK, there is no concept of extended names (for `eventCategory` and `eventName`). The content of `eventCategory` and `eventName` is conditioned and validated (after conditioning) with the following rules: ● **Conditioning**—All leading white space is removed. ● **Conditioning**—All trailing white space is removed. ● **Conditioning**—All internal white spaces other than space characters (' ') are removed. ● **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters. ● **Validation**—`eventCategory` can be empty; `eventName` cannot be empty.

Table 5-7 ▪ ruiTrackEventCustom() Parameters

Parameter	Description
`customValues (RUINAMEVALUEPAIR*)`	Custom data associated with the event. A given name and/or value can be empty. A given name cannot contain white space. All names and values are trimmed to a maximum length determined by the Server. Both names and values have a default maximum of 128 UTF-8 characters.
`numValues (uint32_t)`	The size of the customValues array (i.e., the number of name/value pairs). Cannot be 0.
`sessionID (const char*)`	If multiple user sessions are supported within the application (`multiSessionEnabled = true`), this should contain the unique ID that refers to the user session in which the event occurred. It must be a current valid value used in `ruiStartSession`. If the application supports only a single session (`multiSessionEnabled = false`), then this value should be empty.

Returns

The `ruiTrackEventCustom()` function returns one of the return status constants below.

Table 5-8 ▪ ruiTrackEventCustom() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_NOT_STARTED`	SDK has not been successfully started.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.

Table 5-8 ▪ ruiTrackEventCustom() Returns

Return	Description
RUI_INVALID_SESSION_ID_EXPECTED_EMPTY	The sessionID is expected to be empty, and it was not.
RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY	The sessionID is expected to be non-empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_NOT_ACTIVE	Parameter validation: The sessionID is not currently in use.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Parameter validation: Some API parameter is expected to be non-empty, and is not.
RUI_INVALID_PARAMETER_EXPECTED_NO_WHITESPACE	Some API parameter is expected to be free of white space, and is not.

6

ReachOut Direct-to-Desktop Messaging Service

From the Usage Intelligence dashboard, you can create ReachOut messaging campaigns that are used to deliver messages or surveys directly to the desktop of users who are running your software. You may choose a specific target audience for your message by defining a set of delivery filters so that each message will be delivered only to those users who match the specified criteria (such as geographical region, edition, version, build, language, OS, license status, runtime duration, days since install, etc.)

When building a ReachOut campaign you can choose between two message delivery options.

- **Automated HTML pop-up messages** (handled entirely by the Usage Intelligence library and requires absolutely NO coding.) Currently this is only available on Windows and macOS. For more information, see Automated Message Retrieval.
- **Manually retrieving the message** (plain text or URL) through code by using the `ruiCheckForReachOut()` or `ruiCheckForReachOutOfType()` functions. For more information, see Manual Message Retrieval.

Automated Message Retrieval

The RUI V5 SDK provides a default, automated ReachOut handler that works on Windows and macOS. Developers can override this handler by implementing the ReachOut handler functions with their own code and providing these handler functions to the the `ruiSetReachOutHandler()` function. See the `ruiSDKC.h` file for details on the functions required to support custom ReachOut. The three functions include:

Table 6-1 ▪ Functions Required to Support Custom ReachOut

Function	Description
`void (*RUIReachOutHandler)(void* arg, const char* width, const char* height, int32_t position, const char* message)`	Handles the ReachOut message. The parameters are: • **width**—Width of the window as configured in the ReachOut campaign. Value will be suffixed by P for pixels or % for percentage. • **height**—Height of the window as configured in the ReachOut campaign. Value will be suffixed by P for pixels or % for percentage. • **position**—Position where the window should be displayed (1 = top-left, 2 = top-center, 3 = top-right, 4 = middle-left, 5 = middle-center, 6 = middle right, 7 = bottom-left, 8 = bottom-center, 9 = bottom-right). • **message**—URL to display.
`int (*RUIReachOutGetReadyForNext)(void* arg)`	To handle getting the status of whether the handler is ready for the next ReachOut message. Returns 0 for not ready, non-zero for ready.
`void (RUI_CALLBACK_LINKAGE *RUIReachOutCloser)(void* arg)`	Called when the SDK is stopping or shutting down.

The `ruiSetReachOutHandler()` function sets a custom ReachOut handler. Any previously registered handler, including the default graphical ReachOut handler that may have been registered (`ruiCreateInstance()`). If handler is NULL, then all parameters are considered to be NULL. Setting a handler to NULL effectively removes the current handler, if any, without setting a new handler.

`ruiSetReachOutHandler()` can be called more than once.

`ruiSetReachOutHandler()` is a synchronous function, returning when all functionality is completed.

ruiSetReachOutHandler()

```
RUIRESULT ruiSetReachOutHandler(RUIINSTANCE* ruiInstance, RUIREACHOUTHANDLER handler,
RUIREACHOUTCLOSER closer, RUIREACHOUTREADYFORNEXT getReadyForNext, void* arg)
```

Parameters

The ruiSetReachOutHandler() function has the following parameters.

Table 6-2 ▪ ruiSetReachOutHandler() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
handler (RUIREACHOUTHANDLER)	The function to be called when a ReachOut is received from the Server.
closer (RUIREACHOUTCLOSER)	The function to be called when the SDK is shutting down.
getReadyForNext (RUIREACHOUTREADYFORNEXT)	The function to be called when the SDK is checking to see if the handler is ready for a new ReachOut message.
arg (void*)	Data to be passed to handler function and closer function; provides the client a self-managed handle.

Returns

The ruiSetReachOutHandler() function returns one of the return status constants below.

Table 6-3 ▪ ruiSetReachOutHandler() Returns

Parameter	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_SDK_ALREADY_STARTED	The SDK has already been successfully started.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.

Manual Message Retrieval

When you want full control on when and where in your application to display a ReachOut message to your users, you can define ReachOut messages of the type plain text or URL. From within the application call one of the below functions to check with the Usage Intelligence server whether there are any pending messages (of this type) waiting to be delivered.

You may choose to display plain text messages anywhere in the application such as in a status bar or information box. URL type messages can either be opened in a browser or else rendered it in a HTML previewer embedded within the application.

The difference between `ruiCheckForReachOut()` and `ruiCheckForReachOutOfType()` is that `ruiCheckForReachOut()` takes an 'empty' `messageType` parameter and fills it with the type of message that is sent by the server. In the case of `ruiCheckForReachOutOfType()`, the message type is specified by the developer and the server would then only send messages of that type.

The message type can be one of the following constants:

```
RUI_MESSAGE_TYPE_ANY  (0)
RUI_MESSAGE_TYPE_TEXT (1)
RUI_MESSAGE_TYPE_URL  (2)
```

For more information, see the following:

- Checking for Manual ReachOut Messages of Any Type
- Checking for Manual ReachOut Messages of a Specified Type
- Freeing the Memory Allocated by the SDK

Checking for Manual ReachOut Messages of Any Type

The `ruiCheckForReachOut()` function explicitly checks for manual ReachOut messages on the Server. `ruiCheckForReachOut()` will check for any manual ReachOut message type, whereas `ruiCheckForReachOutOfType()` will check for ReachOut messages of a specified type.

`ruiCheckForReachOut()` can be called between `ruiStartSDK()` and `ruiStopSDK()`, and can be called zero or more times.

`ruiCheckForReachOut()` is a synchronous function, returning when all functionality is completed.

ruiCheckForReachOut()

```
RUIRESULT ruiCheckForReachOut(RUIINSTANCE* ruiInstance, char** message, int32_t* messageCount,
int32_t* messageType)
```

Parameters

The `ruiCheckForReachOut()` function has the following parameters.

Table 6-4 ▪ ruiCheckForReachOut() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.

Table 6-4 ▪ ruiCheckForReachOut() Parameters

Parameter	Description
`message (char**)`	Receives the ReachOut string allocated by the SDK; must be freed via `ruiFree()`.
`messageCount (int32_t*)`	Receives the message count (including returned message). A zero (0) return means no message is available on the server. If `messageCount` is greater than zero, it indicates the number of messages remaining on the server. No allocation, no `ruiFree()`.
`messageType (int32_t*)`	This value is filled by the SDK and contains the type of message that is received (can be either `RUI_MESSAGE_TYPE_TEXT` or `RUI_MESSAGE_TYPE_URL`.)

Returns

The `ruiCheckForReachOut()` function returns one of the return status constants below.

Table 6-5 ▪ ruiCheckForReachOut() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`RUI_TIME_THRESHOLD_NOT_REACHED`	The API call frequency threshold (set by the Server) has not been reached.
`RUI_NETWORK_CONNECTION_ERROR`	Not able to reach the Server.
`RUI_NETWORK_SERVER_ERROR`	Error while communicating with the Server.
`RUI_NETWORK_RESPONSE_INVALID`	Message format error while communicating with the Server.

Code Example

```
bool useDefaultReachOutHandler = true;
RUIINSTANCE* mySDK = ruiCreateInstance(useDefaultReachOutHandler); //...; //Creation and initialization shown in other snippets.

int32_t message_type;
char* message;
int32_t message_count;

RUIRESULT messageRet = ruiCheckForReachOut(mySDK, &message, &message_count, &message_type);

if(messageRet == RUI_OK && message_count > 0){
    cout << "This is your message:----" <<  message << endl;
} else {
    cout << "No messages" << endl;
}
ruiFree(message);
```

Checking for Manual ReachOut Messages of a Specified Type

The ruiCheckForReachOutOfType() function requests a manual ReachOut message from the server while specifying the type of message that is needed. The message type needed is to be sent in the messageTypeExpected parameter, and can be one of the message type constants described above.

ruiCheckForReachOut() will check for any manual ReachOut message type, whereas ruiCheckForReachOutOfType() will check for ReachOut messages of a specified type.

ruiCheckForReachOutOfType()

```
RUIRESULT ruiCheckForReachOutOfType(RUIINSTANCE* ruiInstance, char** message, int32_t* messageCount, int32_t messageTypeExpected)
```

Parameters

The ruiCheckForReachOutOfType() function has the following parameters.

Table 6-6 ▪ ruiCheckForReachOutOfType() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
message (char**)	Receives the ReachOut string allocated by the SDK; must be freed via ruiFree().
messageCount (int32_t*)	Receives the message count (including returned message). A zero (0) return means no message is available on the server. If messageCount is greater than zero, it indicates the number of messages remaining on the server.

Table 6-6 ▪ ruiCheckForReachOutOfType() Parameters

Parameter	Description
`messageType (int32_t)`	This value is provided by the application and dictates the type of message that is expected to be received (can be either `RUI_MESSAGE_TYPE_TEXT` or `RUI_MESSAGE_TYPE_URL`).

Returns

The `ruiCheckForReachOutOfType()` function returns one of the return status constants below.

Table 6-7 ▪ ruiCheckForReachOutOfType() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_SUSPENDED`	The Server has instructed a temporary back-off.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration has not been successfully created.
`RUI_SDK_ALREADY_STOPPED`	SDK has already been successfully stopped.
`RUI_TIME_THRESHOLD_NOT_REACHED`	The API call frequency threshold (set by the Server) has not been reached.
`RUI_NETWORK_CONNECTION_ERROR`	Not able to reach the Server.
`RUI_NETWORK_SERVER_ERROR`	Error while communicating with the Server.
`RUI_NETWORK_RESPONSE_INVALID`	Message format error while communicating with the Server.

Code Example

```
bool useDefaultReachOutHandler = true;
RUIINSTANCE* mySDK = ruiCreateInstance(useDefaultReachOutHandler); //...; //Creation and initialization
shown in other snippets.

int32_t message_type_expected = RUI_MESSAGE_TYPE_TEXT;
char* message;
```

```
int32_t message_count;

RUIRESULT messageRet = ruiCheckForReachOutOfType(mySDK, &message, &message_count,
message_type_expected);

if(messageRet == RUI_OK && message_count > 0){
    cout << "This is your message:----" <<  message << endl;
} else {
    cout << "No messages" << endl;
}
ruiFree(message);
```

Freeing the Memory Allocated by the SDK

The `ruiFree()` function frees the memory allocated by the SDK. The APIs that return allocated memory are: `ruiGetSDKVersion()`, `ruiCheckForReachOut()`, `ruiCheckForReachOutOfType()`, `ruiCheckLicenseKey()`, and `ruiSetLicenseKey()`.

`ruiFree()` can be called any time and can be called more than once.

`ruiFree()` is a synchronous function, returning when all functionality is completed.

ruiFree()

```
void ruiFree(void* arg)
```

Parameters

The `ruiFree()` function has the following parameters.

Table 6-8 ▪ ruiFree() Parameters

Parameter	Description
`arg (void*)`	Pointer to allocated memory returned by the SDK in one of the above APIs.

7

Exception Tracking

Usage Intelligence is able to collect runtime exceptions from your application and then produce reports on the exceptions that were collected. Once an exception is tracked, Usage Intelligence will also save a snapshot of the current machine architecture so that you can later (through the on-line exception browser within the Usage Intelligence dashboard) investigate the exception details and pinpoint any specific OS or architecture related information that are the cause of common exceptions.

The `ruiTrackException()` function logs an exception event with the supplied data.

`ruiTrackException()` can be called between `ruiStartSDK()` and `ruiStopSDK()`, and can be called zero or more times.

`ruiTrackException()` can be called while a New Registration is being performed (`ruiCreateConfig()`, `ruiStartSDK()`). However, the event data is not written to the log file until the New Registration completes, and if the New Registration fails, the data will be lost.

`ruiTrackException()` is an asynchronous function, returning immediately with further functionality executed on separate thread(s).

ruiTrackException()

```
RUIRESULT ruiTrackException(RUIINSTANCE* ruiInstance, RUIEXCEPTIONEVENT exceptionData, const
char* sessionID)
```

Parameters

The `ruiTrackException()` function has the following parameters.

Table 7-1 ▪ ruiTrackException() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.

Table 7-1 ▪ ruiTrackException() Parameters

Parameter	Description

`event (RUIExecptionEvent)`	A struct containing the exception event data to be logged.
	
	The `RUIExecptionEvent` class contains the following fields:
	
	`const char* className;`
	`const char* methodName;`
	`const char* exceptionMessage;`
	`const char* stackTrace;`
	
	The content of `exceptionData.className` and `exceptionData.methodName` are conditioned and validated (after conditioning) with the following rules:
	
	• **Conditioning**—All leading white space is removed.
	• **Conditioning**—All trailing white space is removed.
	• **Conditioning**—All internal white spaces other than space characters (‘ ‘) are removed.
	• **Conditioning**—Trimmed to a maximum of 128 UTF-8 characters.
	• **Validation**—Cannot be empty.

`sessionID (const char*)`	An optional session ID complying with above usage (content conditioning and validation rules in `ruiStartSession()`). Empty if not used. Different than V4 of the RUI (Trackerbird) SDK, `ruiTrackException()` accepts a `sessionID` parameter. The usage requirements of the `sessionID` parameter are the following: • If multiple user sessions are supported within the application (`multiSessionEnabled = true`), `sessionID` must be a current valid value used in `ruiStartSession()`, or it can be empty. This is different than normal event tracking APIs, whereby an empty value is not allowed. • If the application supports only a single session (`multiSessionEnabled = false`), then sessionID must be empty.

Returns

The `ruiTrackException()` function returns one of the return status constants below.

Table 7-2 ▪ ruiTrackException() Returns

Return	Description
`RUI_OK`	Synchronous functionality successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.

Table 7-2 ▪ ruiTrackException() Returns

Return	Description
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_NOT_STARTED	SDK has not been successfully started.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_SESSION_ID_EXPECTED_EMPTY	The sessionID is expected to be empty, and it was not.
RUI_INVALID_SESSION_ID_TOO_SHORT	The sessionID violates its allowable minimum length.
RUI_INVALID_SESSION_ID_TOO_LONG	The sessionID violates its allowable maximum length.
RUI_INVALID_SESSION_ID_NOT_ACTIVE	The sessionID is not currently in use.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.

8

License Management

Usage Intelligence allows you to maintain your own license key registry on the Usage Intelligence server in order to track license key usage and verify the status/validity of license keys used on your clients.

There are multiple ways that the key registry is populated with license keys:

- Keys are collected automatically from your clients whenever you call the `ruiSetLicenseKey()` function.
- You can add/edit keys manually via the Usage Intelligence dashboard.
- You can add/edit keys directly from your CRM by using the Usage Intelligence Web API.

For more information, see:

- Client vs. Server Managed Licensing
- Checking the License Data of the Supplied License Key
- Setting the Current License to the Supplied License Key

Client vs. Server Managed Licensing

Usage Intelligence gives you the option to choose between managing your license key status (i.e. Blocked, Allowed, Expired or Activated) and key type on the server (server managed) or managing this status through the application (client managed). Applications can individually set whether each license status or license type is either Sever Managed or Client Managed by visiting the **License Key Management Settings** page on the Usage Intelligence dashboard. The major difference is outlined below:

Client Managed

The server licensing mechanism works in reporting-only mode and your application is expected to notify the server that the license status has changed through the use of `ruiSetLicenseData()`.

When to Use

Use client managed when you have implemented your own licensing module/mechanism within your application that can identify whether the license key used by this client is blocked, allowed, expired or activated. In this case you do not need to query the Usage Intelligence server to get this license status. However you can simply use this function to passively inform Usage Intelligence about the license status used by the client. In this case:

- Usage Intelligence will use this info to filter and report the different key types and statuses and their activity.
- Usage Intelligence licensing server will operate in passive mode (i.e. reporting only).
- Calling `ruiCheckLicenseKey()` will return the license type and flags as Unknown (-1).

Server Managed

You manage the key status on the server side and your application queries the server to determine the status of a particular license key by calling `ruiCheckLicenseKey()` or `ruiSetLicenseKey()`.

When to Use

Use server managed if you do not have your own licensing module/mechanism within your application and thus you have no way to identify the license status at the client side.

In this mode, whenever a client changes their license key your application can call `ruiSetLicenseKey()` to register the new license key. In reply to this API call, the server will check if the license key exists on the key register and in the reply it will specify to your application whether this key is flagged as blocked, allowed, expired or activated, along with the type of key submitted. If you want to verify a key without actually registering a key change for this client you can use `ruiCheckLicenseKey()` which returns the same values but does not register this key with the server. In this case:

- The key register is maintained manually on the server by the software owner
- Usage Intelligence licensing server will operate in active mode so apart from using this key info for filtering and reporting, it will also report back the key status (validity) to the SDK whenever requested through the API.
- Calling `ruiCheckLicenseKey()` or `ruiSetLicenseKey()` will return the 4 status flags denoting whether a registered key is: Blocked, Allowed, Expired and Activated and the key type.
- If the key does not exist on the server, all 4 status flags will be returned as false (0).

Checking the License Data of the Supplied License Key

The `ruiCheckLicenseKey()` function checks the Server for the license data for the supplied `licenseKey`. Whereas `ruiCheckLicenseKey()` is a passive check, `ruiSetLicenseKey()` changes the license key. The license array has size, indexes and values as specified in RUISDKDefines.h.

Note ▪ *The order of the license array data has changed from the RUI (Trackerbird) SDK V4.*

The `ruiCheckLicenseKey()` function can be called between `ruiStartSDK()` and `ruiStopSDK()`, and can be called zero or more times.

The ruiCheckLicenseKey() function is a synchronous function returning when all functionality is completed.

ruiCheckLicenseKey()

```
RUIRESULT ruiCheckLicenseKey(RUIINSTANCE* ruiInstance, const char* licenseKey, int32_t** licenseArray)
```

Parameters

The ruiCheckLicenseKey() function has the following parameters.

Table 8-1 ▪ ruiCheckLicenseKey() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
licenseKey (const char*)	The license key to be checked. This value cannot be empty. The function accepts a const char* parameter that is the license key itself and an int32_t pointer to an array of length 5 that it fills with the returned result. You may use the following constants to refer to the required value by its index: RUI_LICENSE_ARRAY_INDEX_KEY_TYPE (0) RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED (1) RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE (2) RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED (3) RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED (4) Each of the values 1 through 4 will be set to either 0 or 1 that refers to false or true respectively. The first value (RUI_LICENSE_ARRAY_INDEX_KEY_TYPE) will be set to a number between 0 and 7 (inclusive) that refers to the 8 possible license types listed below. The values may also be -1 that means “Unknown”. The following are the possible license types: RUI_KEY_TYPE_UNCHANGED (-1) RUI_KEY_TYPE_EVALUATION (0) RUI_KEY_TYPE_PURCHASED (1) RUI_KEY_TYPE_FREEWARE (2) RUI_KEY_TYPE_UNKNOWN (3) RUI_KEY_TYPE_NFR (4) - Key Type is Not For Resale RUI_KEY_TYPE_CUSTOM1 (5) RUI_KEY_TYPE_CUSTOM2 (6) RUI_KEY_TYPE_CUSTOM3 (7) The following are the possible key status values: • RUI_KEY_STATUS_UNCHANGED (-1)—Key Status is Unchanged (when in parameter) or Unknown (when out parameter). • RUI_KEY_STATUS_NO (0)—Key Status is No. • RUI_KEY_STATUS_YES (1)—Key Status is Yes.
licenseArray (int32_t**)	The array that will be filled to contain the license status flags.

Returns

The `ruiCheckLicenseKey()` function returns one of the return status constants below.

Table 8-2 ▪ ruiCheckLicenseKey() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
RUI_TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.
RUI_NETWORK_CONNECTION_ERROR	Not able to reach the Server.
RUI_NETWORK_SERVER_ERROR	Error while communicating with the Server.
RUI_NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

```
//Test a license key
bool useDefaultReachOutHandler = true;
RUIINSTANCE* mySDK = ruiCreateInstance(useDefaultReachOutHandler); //...; //Creation and initialization
shown in other snippets.
char* myProductKey = "xyz";
int32_t* licenseResult = NULL;
RUIRESULT rc = ruiCheckLicenseKey(mySDK, myProductKey, &licenseResult);
if(rc == RUI_OK)
{
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_TYPE] == RUI_KEY_TYPE_UNCHANGED) {
        cout << "License key information is unknown\n" << endl;
```

```
    } else {
        cout << "License key type is " << licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_TYPE] << endl;
    }
    //Check if the license key is activated
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE] == RUI_KEY_STATUS_YES){
        cout << "License Active" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE] == RUI_KEY_STATUS_NO) {
        cout << "License Inactive" << endl;
    } else {
        cout << "License key information is unknown" << endl;
    }

    //check if license key is blocked
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED] == RUI_KEY_STATUS_YES){
        cout << "License is blocked" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED] == RUI_KEY_STATUS_NO) {
        cout << "License is NOT blocked" << endl;
    } else {
        cout << "License blocked status unknown" << endl;
    }

    //Check if license key is expired
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED] == RUI_KEY_STATUS_YES){
        cout << "License is expired" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED] == RUI_KEY_STATUS_NO) {
        cout << "License is NOT expired" << endl;
    } else {
        cout << "License expiration status unknown" << endl;
    }

    //Check if license key is allowed
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED] == RUI_KEY_STATUS_YES){
        cout << "Key is allowed" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED] == RUI_KEY_STATUS_NO) {
        cout << "Key is NOT on allowed list" << endl;
    } else {
        cout << "Key allowed status unknown" << endl;
    }
} else {
    cout << "Failed to invoke function ruiCheckLicenseKey()" << endl;
}

ruiFree(licenseResult);
```

Setting the Current License to the Supplied License Key

The `ruiSetLicenseKey()` function checks the Server for the license data for the supplied `licenseKey` and sets the current license to `licenseKey`.

Whereas `ruiCheckLicenseKey()` is a passive check, `ruiCheckLicenseKey()` changes the license key. The Server always registers the `licenseKey` even if the Server knows nothing about the `licenseKey`.

When a new (unknown) licenseKey is registered, the Server sets the license data to keyType RUI_KEY_TYPE_UNKNOWN and the four status flags (blocked, allowed, expired, activated) to RUI_KEY_STATUS_NO. The license array has size, indexes and values as specified in RUISDKDefines.h.

The order of the license array data has changed from the Usage Intelligence SDK V4. The ruiSetLicenseKey() function can be called between ruiStartSDK() and ruiStopSDK(), and can be called zero or more times.

The ruiSetLicenseKey() function is primarily a synchronous function, returning once the check with Server has completed. Some post- processing functionality is performed asynchronously, executed on separate thread(s).

The ruiSetLicenseKey() function should be called when an end user is trying to enter a new license key into your application and you would like to confirm that the key is in fact valid (i.e. blocked or allowed), active, or expired. The function is very similar to the ruiCheckLicenseKey() function, however rather than just being a passive license check, it also registers the new key with the server and associates it with this particular client installation.

ruiSetLicenseKey()

```
RUIRESULT ruiSetLicenseKey(RUIINSTANCE* ruiInstance, const char* licenseKey, int32_t** licenseArray,
const char* sessionID)
```

Parameters

The ruiSetLicenseKey() function has the following parameters.

Table 8-3 ▪ ruiSetLicenseKey() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().

Table 8-3 ▪ ruiSetLicenseKey() Parameters

Parameter	Description

`licenseKey (const char*)`	The license key to be checked. This value cannot be empty.
	
	The function accepts a `const char*` parameter that is the license key itself and an `int32_t` pointer to an array of length 5 that it fills with the returned result. You may use the following constants to refer to the required value by its index:
	
	`RUI_LICENSE_ARRAY_INDEX_KEY_TYPE (0)`
	`RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED (1)`
	`RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE (2)`
	`RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED (3)`
	`RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED (4)`
	
	Each of the values 1 through 4 will be set to either 0 or 1 that refers to false or true respectively. The first value (`RUI_LICENSE_ARRAY_INDEX_KEY_TYPE`) will be set to a number between 0 and 7 (inclusive) that refers to the 8 possible license types listed below. The values may also be -1 that means "Unknown". The following are the possible license types:
	
	`RUI_KEY_TYPE_UNCHANGED (-1)`
	`RUI_KEY_TYPE_EVALUATION (0)`
	`RUI_KEY_TYPE_PURCHASED (1)`
	`RUI_KEY_TYPE_FREEWARE (2)`
	`RUI_KEY_TYPE_UNKNOWN (3)`
	`RUI_KEY_TYPE_NFR (4) - Key Type is Not For Resale`
	`RUI_KEY_TYPE_CUSTOM1 (5)`
	`RUI_KEY_TYPE_CUSTOM2 (6)`
	`RUI_KEY_TYPE_CUSTOM3 (7)`
	
	The following are the possible key status values:
	
	• `RUI_KEY_STATUS_UNCHANGED (-1)`—Key Status is Unchanged (when in parameter) or Unknown (when out parameter).
	• `RUI_KEY_STATUS_NO (0)`—Key Status is No.
	• `RUI_KEY_STATUS_YES (1)`—Key Status is Yes.

`licenseArray (int32_t**)`	Pointer to the array that will be filled to contain the license status flags.
`sessionID (const char*)`	An optional session ID complying with above usage (content conditioning and validation rules in `ruiStartSession()`). Use empty if not used. Different from the V4 of the Usage Intelligence SDK, a `sessionID` parameter can be supplied (based on `ruiCreateConfig()` multi session value): • **If multiSessionEnabled is set to false**—`sessionID` must be empty. This is similar to event tracking APIs. • **If multiSessionEnabled is set to true**—`sessionID` must be a current valid value used in `ruiStartSession()`, or it can be empty. This is different than normal event tracking APIs, whereby a empty value is not be allowed.

Returns

The ruiSetLicenseKey() function returns one of the return status constants below.

Table 8-4 ▪ ruiSetLicenseKey() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY	Some API parameter is expected to be non-empty, and is not.
RUI_TIME_THRESHOLD_NOT_REACHED	The API call frequency threshold (set by the Server) has not been reached.
RUI_NETWORK_CONNECTION_ERROR	Not able to reach the Server.
RUI_NETWORK_SERVER_ERROR	Error while communicating with the Server.
RUI_NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.

Code Example

```
//Register a new license key
bool useDefaultReachOutHandler = true;
RUIINSTANCE* mySDK = ruiCreateInstance(useDefaultReachOutHandler); //...; //Creation and initialization
shown in other snippets.
char* myProductKey = "xyz";
int32_t* licenseResult = NULL;

if(ruiSetLicenseKey(mySDK, myProductKey, &licenseResult, NULL)==RUI_OK)
{
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_TYPE] == RUI_KEY_TYPE_UNCHANGED) {
        cout << "License key information is unknown" << endl;
```

```
    } else {
        cout << "License key type is " << licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_TYPE] << endl;
    }
    //Check if the license key is activated
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE] == RUI_KEY_STATUS_YES){
        cout << "License Active" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ACTIVE] == RUI_KEY_STATUS_NO) {
        cout << "License Inactive" << endl;
    } else {
        cout << "License status unknown" << endl;
    }

    //check if license key is blocked
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED] == RUI_KEY_STATUS_YES){
        cout << "License is blocked" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_BLOCKED] == RUI_KEY_STATUS_NO) {
        cout << "License is NOT blocked" << endl;
    } else {
        cout << "License blocked status unknown" << endl;
    }

    //Check if license key is expired
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED] == RUI_KEY_STATUS_YES){
        cout << "License is expired" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_EXPIRED] == RUI_KEY_STATUS_NO) {
        cout << "License is NOT expired" << endl;
    } else {
        cout << "License expiration status unknown" << endl;
    }

    //Check if license key is allowed
    if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED] == RUI_KEY_STATUS_YES){
        cout << "Key is allowed" << endl;
    } else if (licenseResult[RUI_LICENSE_ARRAY_INDEX_KEY_ALLOWED] == RUI_KEY_STATUS_NO) {
        cout << "Key is NOT on allowed list" << endl;
    } else {
        cout << "Key allowed status unknown" << endl;
    }
} else {
    cout << "Failed to invoke function ruiSetLicenseKey()" << endl;
}

ruiFree(licenseResult);
```

9

Custom Properties

Apart from the pre-set values that Usage Intelligence collects, such as OS version, product version, edition, language, and license type, you also have the ability to collect any custom value that is relevant to your specific application.

Typical examples where you can benefit from custom properties include storing the download source or marketing campaign from where the user downloaded your software or some other status in your application. These custom properties will then be available inside the filters panel on every report so you may use them as part of your report filtering criteria.

The `ruiSetCustomProperty()` function sets or clears the custom property data. For more information, see Setting Custom Property Data.

***Note** ▪ By default, you can store up to 1000 unique values inside every custom property. Please contact Usage Intelligence support (support@revenera.com) if you want to discuss this limit or alternative uses.*

Setting Custom Property Data

The `ruiSetCustomProperty()` function sets or clears the custom property data.

***Note** ▪ The custom property data must be set every time the SDK instance is run. This is different than V4 of the Usage Intelligence SDK where the supplied product data was stored in the SDK configuration file and if it was not supplied, the values in the configuration file were used.*

`ruiSetCustomProperty()` can be called between `ruiCreateConfig()` and `ruiStopSDK()`, and can be called zero or more times.

`ruiSetCustomProperty()` is a synchronous function, returning when all functionality is completed.

ruiSetCustomProperty()

```
RUIRESULT ruiSetCustomProperty(RUIINSTANCE* ruiInstance, uint32_t customPropertyID, const
char* customValue)
```

Parameters

The ruiSetCustomProperty() function has the following parameters.

Table 9-1 ▪ ruiSetCustomProperty() Parameters

Parameter	Description
ruiInstance (RUIINSTANCE*)	Pointer to the RUI instance created via ruiCreateInstance().
customPropertyID (uint32_t)	This is a numeric index between 1 and 20. On the Usage Intelligence dashboard, custom values are given an ID ranging from C01 to C20. This ID is used to identify which of the 20 possible custom value is being set.
customValue (const char*)	The custom property value to use in Server reports; empty value clears the value.

Returns

The ruiSetCustomProperty() function returns one of the return status constants below.

Table 9-2 ▪ ruiSetCustomProperty() Returns

Return	Description
RUI_OK	Function successful.
RUI_INVALID_SDK_OBJECT	SDK Instance parameter is NULL or invalid.
RUI_SDK_INTERNAL_ERROR_FATAL	Irrecoverable internal fatal error. No further API calls should be made.
RUI_SDK_ABORTED	A required New Registration has failed, and hence the SDK is aborted. ruiStopSDK() and ruiDestroyInstance() are possible.
RUI_SDK_SUSPENDED	The Server has instructed a temporary back-off.
RUI_SDK_PERMANENTLY_DISABLED	The Server has instructed a permanent disable.
RUI_SDK_OPTED_OUT	Instance has been instructed by the application to opt-out.
RUI_CONFIG_NOT_CREATED	Configuration has not been successfully created.
RUI_SDK_ALREADY_STOPPED	SDK has already been successfully stopped.
RUI_INVALID_CUSTOM_PROPERTY_ID	The customPropertyID violates its allowable range.

10

SDK Status Checks

You can perform SDK status checks using the `ruiGetState()` and `ruiTestConnection()` functions.

- Getting the State of the RUI Instance
- Testing the Connection Between the SDK and the Server

Getting the State of the RUI Instance

The `ruiGetState()` function returns a `RUISTATE` value that contains the state of the RUI instance. The SDK state can change asynchronously.

This function can be called more than once.

This function is a synchronous function.

ruiGetState()

```
RUISTATE ruiGetState(RUIINSTANCE* ruiInstance)
```

Parameters

The `ruiGetState()` function has the following parameters.

Table 10-1 ▪ ruiGetState() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.

Returns

The ruiGetState() function returns one of the return status constants below.

Table 10-2 ▪ ruiGetState() Returns

Return	Description
RUI_SDK_STATE_FATAL_ERROR	Irrecoverable internal fatal error. No further API calls should be made
RUI_SDK_STATE_UNINITIALIZED	Instance successfully created (ruiCreateInstance()) but not yet successfully configured ruiCreateConfig().
RUI_SDK_STATE_CONFIG_INITIALIZED_NOT_STARTED	Successfully configured and not yet started via ruiStartSDK().
RUI_SDK_STATE_STARTED_NEW_REG_RUNNING	Running ruiStartSDK() with a New Registration in progress but not yet completed.
RUI_SDK_STATE_RUNNING	Running ruiStartSDK() with no need for New Registration or with successfully completed New Registration.
RUI_SDK_STATE_ABORTED_NEW_REG_PROXY_AUTH_FAILURE	Aborted run ruiStartSDK() due to failed New Registration.
RUI_SDK_STATE_ABORTED_NEW_REG_NETWORK_FAILURE	Aborted run ruiStartSDK() due to failed New Registration.
RUI_SDK_STATE_ABORTED_NEW_REG_FAILED	Aborted run ruiStartSDK() due to failed New Registration.
RUI_SDK_STATE_SUSPENDED	Instance has been instructed by Server to back-off. Will return to running once back-off cleared.
RUI_SDK_STATE_PERMANENTLY_DISABLED	Instance has been instructed by Server to disable. This is a permanent, irrecoverable state.
RUI_SDK_STATE_STOPPING_NON_SYNC	Stop in progress ruiStopSDK(). Stopping non-Sync-related threads.
RUI_SDK_STATE_STOPPING_ALL	Stop in progress ruiStopSDK(). Stopping Sync-related threads.
RUI_SDK_STATE_STOPPED	Stop completed ruiStopSDK()
RUI_SDK_STATE_OPTED_OUT	Instance has been instructed by the application to opt-out.

Testing the Connection Between the SDK and the Server

The `ruiTestConnection()` function tests the connection between the SDK and the Server. If a valid configuration file exists from `ruiCreateConfig()`, the URL used for the test will be the one in that file, set by the Server. Otherwise, the URL used for the test will be the one set by the client in the call to `ruiCreateConfig()` . The proxy is used during the test if set by calling `ruiSetProxy()`.

This function can be called between `ruiCreateConfig()` and `ruiStopSDK()` and can be called zero or more times.

This function is a synchronous function and only returns with all functionality is completed.

ruiTestConnection()

```
RUIRESULT ruiTestConnection(RUIINSTANCE* ruiInstance)
```

Parameters

The `ruiTestConnection()` function has the following parameters.

Table 10-3 ▪ ruiTestConnection() Parameters

Parameter	Description
`ruiInstance (RUIINSTANCE*)`	Pointer to the RUI instance created via `ruiCreateInstance()`.

Returns

The `ruiTestConnection()` function returns one of the return status constants below.

Table 10-4 ▪ ruiTestConnection() Returns

Return	Description
`RUI_OK`	Function successful.
`RUI_INVALID_SDK_OBJECT`	SDK Instance parameter is NULL or invalid.
`RUI_SDK_INTERNAL_ERROR_FATAL`	Irrecoverable internal fatal error. No further API calls should be made.
`RUI_SDK_ABORTED`	A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.
`RUI_SDK_PERMANENTLY_DISABLED`	The Server has instructed a permanent disable.
`RUI_SDK_OPTED_OUT`	Instance has been instructed by the application to opt-out.
`RUI_CONFIG_NOT_CREATED`	Configuration from `ruiCreateConfig()` has not been successfully created.

Table 10-4 ▪ ruiTestConnection() Returns

Return	Description
RUI_SDK_ALREADY_STOPPED	SDK has already been stopped.
RUI_NETWORK_COMMUNICATION_ERROR	Not able to reach the Server.
RUI_NETWORK_SERVER_ERROR	Error while communicating with the Server.
RUI_NETWORK_RESPONSE_INVALID	Message format error while communicating with the Server.
RUI_TEST_CONNECTION_INVALID_PRODUCT_ID	Invalid Product ID.
RUI_TEST_CONNECTION_MISMATCH	Mismatch between Product ID and URL.

11

Common Function Return Values

This section lists common return values for Usage Intellligence functions.

RUI_OK (0)

Function (which may be synchronous or asynchronous), fully successful during synchronous functionality.

RUI_SDK_INTERNAL_ERROR_FATAL (-999)

Irrecoverable internal fatal error. No further API calls should be made.

RUI_SDK_ABORTED (-998)

A required New Registration has failed, and hence the SDK is aborted. `ruiStopSDK()` and `ruiDestroyInstance()` are possible.

RUI_INVALID_SDK_OBJECT (-100)

SDK Instance parameter is NULL or invalid. Not used in CPP interface.

RUI_INVALID_PARAMETER_EXPECTED_NON_NULL (-110)

Some API parameter is expected to be non-NULL, and is not.

RUI_INVALID_PARAMETER_EXPECTED_NON_EMPTY (-111)

Some API parameter is expected to be non-empty, and is not.

RUI_INVALID_PARAMETER_EXPECTED_NO_WHITESPACE (-113)

Some API parameter is expected to be free of white space, and is not.

RUI_INVALID_PARAMETER_TOO_LONG (-114)

Some API parameter violates its allowable maximum length.

RUI_INVALID_CONFIG_PATH (-120)

The `configFilePath` is not a well-formed directory name.

RUI_INVALID_CONFIG_PATH_NONEXISTENT_DIR (-121)

The `configFilePath` identifies a directory that does not exist.

RUI_INVALID_CONFIG_PATH_NOT_WRITABLE (-122)

The `configFilePath` identifies a directory that is not writable.

RUI_INVALID_PRODUCT_ID (-130)

The `productID` is not a well-formed Product ID.

RUI_INVALID_SERVER_URL (-140)

The `serverURL` is not a well-formed URL.

RUI_INVALID_PROTOCOL (-144)

The protocol is not a legal value. Must be one of the following:

Table 11-1 ▪ Protocol Values

Protocol	Description
`RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION (1)`	Protocol to the Server is HTTP + AES-128 Encrypted payload.
`RUI_PROTOCOL_HTTPS_WITH_FALLBACK (2)`	Protocol to the Server is HTTPS, unless that doesn't work, falling back to HTTP + Encryption.
`RUI_PROTOCOL_HTTPS (3)`	Protocol to the Server is HTTPS with no fall-back.

RUI_INVALID_AES_KEY_EXPECTED_EMPTY (-145)

The AES Key is expected to be NULL/empty, and it is not. This occurs if `RUI_PROTOCOL_HTTPS` is used at the protocol selection and an AES Key is supplied.

RUI_INVALID_AES_KEY_LENGTH (-146)

The AES Key is not the expected length (32 hex chars). An AES key is required if using `RUI_PROTOCOL_HTTP_PLUS_ENCRYPTION` or `RUI_PROTOCOL_HTTPS_WITH_FALLBACK` as the protocol choice.

RUI_INVALID_AES_KEY_FORMAT (-147)

The AES Key is not valid hex encoding. String passed must only include hexadecimal characters.

RUI_INVALID_SESSION_ID_EXPECTED_EMPTY (-150)

The `sessionID` is expected to be empty, and it was not. This occurs if a session ID is passed to functions that accept a session ID but no `ruiStartSession()` is active.

RUI_INVALID_SESSION_ID_EXPECTED_NON_EMPTY (-151)

The `sessionID` is expected to be non-empty, and it was empty.

RUI_INVALID_SESSION_ID_TOO_SHORT (-152)

The `sessionID` violates its allowable minimum length. Minimum length is 10.

RUI_INVALID_SESSION_ID_TOO_LONG (-153)

The `sessionID` violates its allowable maximum length. Maximum length is 64.

RUI_INVALID_SESSION_ID_ALREADY_ACTIVE (-154)

The `sessionID` is already currently in use.

RUI_INVALID_SESSION_ID_NOT_ACTIVE (-155)

The `sessionID` is not currently in use.

RUI_INVALID_CUSTOM_PROPERTY_ID (-160)

The `customPropertyID` violates its allowable range. By default the range is 1 to 20.

RUI_INVALID_DO_SYNC_VALUE (-170)

The `doSync` manual sync flag/limit violates its allowable range.

RUI_INVALID_MESSAGE_TYPE (-180)

The `messageType` is not an allowable value.

RUI_INVALID_PROXY_CREDENTIALS (-190)

The proxy username and password failed proxy authentication.

RUI_INVALID_PROXY_PORT (-191)

The proxy port was not valid.

RUI_CONFIG_NOT_CREATED (-200)

Configuration has not been successfully created. The function `ruiCreateConfig()` must be called before performing this operation.

RUI_CONFIG_ALREADY_CREATED (-201)

Configuration has already been successfully created. A previous `ruiCreateConfig()` was successful and the subsequent calls to this function are not allowed.

RUI_SDK_NOT_STARTED (-210)

SDK has not been successfully started. The function `ruiStartSDK()` must be called before using this function.

RUI_SDK_ALREADY_STARTED (-211)

SDK has already been successfully started. A previous `ruiStartSDK()` was successful and subsequent calls to this function are not allowed.

RUI_SDK_ALREADY_STOPPED (-213)

SDK has already been successfully stopped. A previous `ruiStopSDK()` was successful and subsequent calls to this function are not allowed.

RUI_FUNCTION_NOT_AVAIL (-300)

This indicates that this particular API call is not currently available. Possible causes include:

- **This feature is disabled** from the server side. If this is an optional feature you might need to turn it on from the Usage Intelligence dashboard.
- **You have called this function too many times** in quick succession from the same client. In order to prevent abuse the server might impose a minimum interval (time threshold) before you can call this function again. This interval can vary from seconds to minutes.
- **There has been a time out on this request** to the Usage Intelligence server.

RUI_SYNC_ALREADY_RUNNING (-310)

A sync with the Server is already running. Only one sync operation may be running at a time.

RUI_TIME_THRESHOLD_NOT_REACHED (-320)

The API call time frequency threshold (set by the Server) has not been reached. In other words, the application is generating too many requests per time period.

RUI_SDK_SUSPENDED (-330)

The Server has instructed a temporary back-off. No events are logged but future communication with the Server is possible if the server allows it.

RUI_SDK_PERMANENTLY_DISABLED (-331)

The Server has instructed a permanent disable. No communication with the server is possible and events will not be logged.

RUI_SDK_OPTED_OUT (-332)**

Instance has been instructed by the application to opt-out.

RUI_NETWORK_CONNECTION_ERROR (-400)

Communication attempts were not able to reach the Server. This means there was a problem communicating with the Usage Intelligence server due to:

- Network connectivity problems
- Incorrect proxy settings
- HTTP or HTTPS traffic is blocked by a gateway or firewall

In some cases, you can use `ruiTestConnection()` to help diagnose the issue.

RUI_NETWORK_SERVER_ERROR (-401)

Error while communicating with the Server. Communication with the server was successful but the server response indicates a Server error.

Login to the Usage Intelligence dashboard to make sure your account is active and there are no critical warnings. Test using `ruiTestConnection()` function.

RUI_NETWORK_RESPONSE_INVALID (-402)

The response from the Server was returned with a message format error.

RUI_TEST_CONNECTION_INVALID_PRODUCT_ID (-420)

The `ruiTestConnection()` function had an invalid `ProductID` supplied. Check the Product ID provided to you for accuracy.

RUI_TEST_CONNECTION_MISMATCH (-421)

The `ruiTestConnection()` function had a mismatch between URL and Product ID. Check the Product ID and URL provided to you for accuracy.