Application note Hue EDK connection flow

1. Introduction

The EDK connection flow implements all logic which is needed to discover and connect to a Hue system. However, there are a few cases where user interaction is required which needs to be handled by the application user interface. This document describes in detail how to use the connection flow.

2. When to connect

We distinguish two different types of applications.

1. An application which requires a Hue system to function
    - At startup of application, connect with regular method `ConnectBridgeAsync()`
2. An application which doesn't require a Hue system to function
    - Where appropriate search for bridges in the background using `ConnectBridgeBackgroundAsync()`, and notify the user about Hue if a bridge is found
    - Somewhere in the options/settings menu, provide a way to connect with regular method `ConnectBridgeAsync()`

3. Connecting to a bridge

Main

The main way of connecting to a bridge is using `ConnectBridgeAsync()`.

If a valid bridge is already configured, it will connect to that bridge directly, else it will first search for new bridges on the network and connect if a bridge is found.

In more detail, it will try the following options to connect to a bridge:

- If no known bridges are stored, search for a new bridge and connect if any bridge is found
- If known bridges are stored, connect to the last used bridge on the known IP address
- If that fails, search for the last used bridge and connect if rediscovered on another IP address
- If that fails, connect to the most recently used known bridge which is available on the network
- If that fails, search for a new bridge and connect if any bridge is found
- If that fails, give up

Note that when a bridge is found, there are a few cases where the user needs to perform an action (such as pressing the pushlink button, selecting an entertainment group or updating his bridge firmware) before the connection can be fully completed. See section 5 for details

In the background

If instead of the user explicitly requesting to connect to a bridge, the application wants to discover bridges in the background, there is the option to use `ConnectBridgeBackgroundAsync()`. This does the same as `ConnectBridgeAsync()`, except for the for the following differences:

- The search strategy does not include a brute force retry if the first search cycle didn't deliver any results
- Bridge models without streaming capability will be ignored
- When a new bridge is found, instead of immediately start scanning for the user pressing the pushlink button, it will just finish and remember the discovered bridge. When the application is ready to show the UI to the user, `ConnectBridgeAsync()` can be called to 'resume'.

Note: only use this method for a bridge search in the background, i.e. a search that is not requested by the user. A user requested bridge search should use the standard connect method `ConnectBridgeAsync()`.

Manual IP

In case the bridge discovery doesn't work even though the user has a bridge on the same network, there is a last resort in which the user enters the bridge ip address manually. In this case `ConnectBridgeManualIpAsync(const string &ipAddress)` can be called which skips both loading of any already configured bridge and new bridge discovery. Instead it will immediately try to connect to the bridge on the given ip address. The rest of the procedure is the same as `ConnectBridgeAsync()`.

4. Getting feedback

Via asynchronous callback or handler

Before connecting, the application can use `RegisterFeedbackCallback(FeedbackMessageCallback callback)` to get feedback during the connection procedure via a callback. The callback provides a `FeedbackMessage` which indicates:

- The type of ongoing request: `GetRequestType()`
- An enum identifier: `GetId()`
- A tag (or 'string id') used for looking up the user message translation: `GetTag()`
- A message type: `GetType()`, can be `USER` (needs user interaction) or `INFO`
- If the message type is USER, a user message string explaining the user what to do in the language the EDK is configured in: `GetUserMessage()`
- A debug message string: `GetDebugMessage()`
- The current active bridge: `GetBridge()`
- In case of loading (all stored bridges) or searching (all discovered bridges) a list of bridges: `GetBridgeList()`

Note that instead of a callback, the application can choose to implement the `IFeedbackMessageHandler` interface and set it via `RegisterFeedbackHandler(FeedbackMessageHandlerPtr handler)`.

Via synchronous request

By calling `GetConnectionResult()` the application can request the result of the last connection procedure at any time. If the result is `Completed` or `ActionRequired`, the procedure has delivered a bridge object. In case of `ActionRequired`, the type of required action can be found via `GetLoadedBridge()->GetStatus()` and the user can be informed what to do via `GetLoadedBridge()->GetUserStatus()`.

5. Handling the different results

After starting a bridge connection procedure, you can end up in five different situations in terms of what action is needed. Below table describes these. It assumes using `ConnectBridgeAsync()` but would be similar for the other connection methods.

Situation	FeedbackMessage id	Suggested action
No bridge found	`ID_DONE_NO_BRIDGE_FOUND`	• Show the user message. • Offer a retry option, calling `ConnectBridgeAsync()` again. • Potentially provide the manual IP backup option `ConnectBridgeManualIpAsync(const string &ipAddress)` • A way to skip.
New bridge found (or authorization lost on the current bridge)	`ID_PRESS_PUSH_LINK`	• Show the user message • Show an image of the bridge to guide the user to press the button on the bridge.
User needs to select which entertainment setup to use (in application)	`ID_SELECT_GROUP`	• Show the user message. • Show the list of available groups via `GetLoadedBridge()->GetGroups()` and using `GetName()`. • When the user selects a group call `SelectGroupAsync(GroupPtr group)`.

User needs to perform action external of application (e.g. create entertainment setup or upgrade bridge using the mobile Hue app)	`ID_BRIDGE_NOT_FOUND`
	`ID_INVALID_MODEL`
	`ID_INVALID_VERSION`
	`ID_NO_GROUP_AVAILABLE`
	`ID_BUSY_STREAMING`
	`(ID_INTERNAL_ERROR)`
		• Show the user message.
		• Offer a retry option, calling `ConnectBridgeAsync()` again.
		• A way to skip.
		Note: in case of `NO_GROUP_AVAILABLE`, the application can detect that the user has created a group via `GROUPLIST_UPDATED` to proceed automatically.

Connected to bridge	`DONE_COMPLETED`	• Possibly show a small connect confirmation.

In case of the background search, no action is needed at `ID_DONE_NO_BRIDGE_FOUND`. In case of `ID_DONE_BRIDGE_FOUND`, the user should get notified that there is an option to enable the Hue integration, after which the bridge can be connected to with the regular `ConnectBridgeAsync()`. If a user doesn't want to use the integration, this preference should probably be saved to not automatically search again next time the user starts the application. In that case it would be good to inform the user how to still connect manually later.

6. When the connection flow is in progress

When the user has actively requested to connect to a bridge, the UI should probably show that the connection procedure is in progress and offer a way to cancel.

Only one connection procedure can be active at any time. For any procedure it is always guaranteed that if it is started `PROCEDURE_STARTED` is called, and if it is finished (whether successful or not) `PROCEDURE_FINISHED` is called. Similarly when the procedure is ongoing a call to `GetConnectionResult()` will return `Busy`.

This can be used to e.g. show a spinner and/or change the text of the connect button into a cancel button. To cancel call `AbortConnecting()`.

7. Start streaming

Before creating the HueStream instance, the library can be configured to automatically start streaming at connection: `config->GetAppSettings()->SetAutoStartAtConnection(true/false)`. If the application wants to connect to the bridge already before showing any light effects this should be `false`. If streaming should be started immediately then this can be `true`, note that this takes over the light control so actual effects are needed for the user not to be in the dark.

Start and stop streaming can be controlled via `StartAsync()` and `StopAsync()`.

8. Getting the current bridge(s)

If needed, the active bridge can be retrieved from every feedbackmessage or via `GetLoadedBridge()`. Loading the stored bridges is the first thing which is done at a `ConnectBridgeAsync()` call, but can also done separately via `LoadBridgeInfo()`.

The list of all known bridges can be retrieved from the feedback message `ID_FINISH_LOADING_BRIDGE_CONFIGURED` or via `GetAllKnownBridges()`.

9. Changing to a different bridge

Optionally, a user may want to connect to another bridge than the one currently connected to. Switching to a known bridge can be done with `ConnectManualBridgeInfoAsync(BridgePtr bridge)`. And searching for a totally new bridge with `ConnectNewBridgeAsync()`.

10. Resetting the bridge configuration

When the application wants to fully erase all persistent bridge data, `ResetAllPersistentDataAsync()` can be used. For only resetting the current active bridge: `ResetBridgeInfoAsync()`.