Keyboard Manager Common

This project contains any code that is to be shared between the backend and UI projects. This file covers any functionality in this project which hasn't been covered along with the other modules.

Table of Contents

KeyboardManagerState

This class stores all the data related to remappings and is also used in the sense of a View Model as it used to communicate common data that is shared between the KBM UI and the backend. They are accessed on the UI controls using static class members of `SingleKeyRemapControl` and `ShortcutControl`.

UI States

UI states are used to keep track in which step of the UI flow is the user at, such as which Remap window they are on, or if they have one of the Type windows open. This is required because the hook needs to suppress input and update UI in some cases, and in some cases remappings have to be disabled altogether.

DetectSingleRemapKeyUIBackend and DetectShortcutUIBackend

These methods are called on the low level hook in the main keyboard event handler. When the user opens any UI window the UI states are updated and in this case some remappings have to be disabled. On the Remap keys window, all remappings are disabled, while on the Remap shortcuts window, shortcut remappings are disabled.

In addition to this, if the user has opened the Type window, and the window is in focus, whenever a key event is received we have to update the set of selected keys in the TextBlock in the ContentDialog. These methods also call the

`KeyDelay` handlers to check if the input is Esc/Enter and accordingly handle the Accessibility events for the Type window. When the user clicks the Type button, variables in the KeyboardManagerState store the corresponding TextBlocks which appear in the ContentDialog, so that these UI controls can be updated from the hook on the dispatcher thread.

HandleKeyDelayEvent

This method checks if the UI is in the foreground, and if so runs the key delay handlers that have been registered.

Saving remappings to file

The `SaveConfigToFile` method is called on clicking the OK button on the EditKeyboardWindow or EditShortcutsWindow. Since PowerToys Settings also reads the config JSON file, a named mutex is used before accessing the file, with a 1 second timeout. If the mutex is obtained the settings are written to the default.json file.

Concurrent Access to remap tables

To prevent the UI thread and low level hook thread from concurrently accessing the remap tables we use an `atomic bool` variable, which is set to `true` while the tables are getting updated. When this is `true` the hook will skip all remappings. Use of mutexes in the hook were removed to prevent re-entrant mutex bugs.

KeyDelay

This class implements a queue based approach for processing key events and based on the time difference between key down and key up events executes separate methods for `ShortPress`, `LongPress` or `LongPressReleased`. The class is used for the hold Enter/Esc functionality required for making the Type window accessible and prevent keyboard traps (see this for an example of it's usage). The `KeyEvents` are added to the queue from the hook thread of KBM, and a separate `DelayThread` is used to process the key events by checking the `time` member in the key event. The thresholds for short vs long press and hold wait timeouts are `static` constants, but if the module is extended for other purposes these could be made into arguments.

Note: Deletion of the `KeyDelay` object should never be called from the `DelayThread` i.e. from within one of the 3 handlers, as it can re-enter the mutex and would lead to a deadlock. This can be avoided by either deleting it on a separate thread or as done in the KBM UI, on the dispatcher thread. See this PR for more details on this issue.

Shortcut and RemapShortcut classes

The `Shortcut` class is a data structure for storing key combinations which are valid shortcuts and it contains several methods which are used for shortcut specific operations. `RemapShortcut` consists of a shortcut/key union (`std::variant`), along with other boolean flags which are required on the hook side for storing any relevant keyboard states mid-execution.

IsKeyboardStateClearExceptShortcut

This method is used by the `HandleShortcutRemapEvent` to check if any other keys on the keyboard have been pressed apart from the keys in the shortcut. This is required because shortcut to shortcut remaps should not be applied if the shortcut is pressed with other keys. The method iterates over all the possible key codes, except any keys that are considered reserved, unassigned, OEM-specific or undefined, as well as mouse buttons (see list here).

CheckModifiersKeyboardState

This method uses `GetVirtualKeyState` (internally calls `GetAsyncKeyState` in production code), to check if all the modifiers of the current shortcut are being pressed. Since Win doesn't have a non-L/R key code we check this by checking both LWIN and RWIN.

Tests

Tests for some methods in the `Shortcut` class can be found here.

Helpers

This namespace has any methods which are used across either UI or the backend which aren't specific to either. Some of these methods have tests here.

Foreground App Detection

`GetCurrentApplication` is used for detecting the foreground process for App-specific shortcuts. The logic is very similar to that used for FZ's app exception feature, involving `GetForegroundWindow` and `get_process_path`. The one additional case which has been added is for full-screen UWP apps, where the above method fails and returns `ApplicationFrameHost.exe`. The `GetFullscreenUWPWindowHandle` uses `GetGUIThreadInfo` API to find the window linked to the GUI thread. This logic is based on this stackoverflow answer.

Note: The `GetForegroundProcess` method performs string allocation in a weird way because of exceptions that were occurring while running tests as a result of memory being allocated or deallocated across dll boundaries. Here's the comment from the PR where this was added > To make app-specific logic test-able, a

GetForegroundProcess was added to the input interface which internally calls GetCurrentApplication. This allows us to mock this method in the test project by just setting some process name as the foreground process for that function. When I set this to just return the string name, it would goes runtime errors on the test project in debug_heap with `__acrt_first_block == header`. Based on this stackoverflow answer, this would happen if allocation happens in one dll's code space and deallocation happens in another. One way to avoid this is to change both the projects to MD (multi threaded dll) instead of MT(multi threaded), however that results in many compile-time errors since all the PT projects are configured as MT. To solve this, the GetForegroundProcess was rewritten such that its argument is the output variable, and we allocate memory for that string within the AppSpecificHandler method rather than in that function.