Hard keys

Contents

Definition

Hardkeys: Also known as Hardware keys; Any controls in the car system connected to the Head unit. Examples of hardkeys are volume controls, fixed function buttons (back, forward, home screen, play pause), rotary controls etc.

Traditionally hardkeys referred only to physical controls (e.g. switches, rotary dails etc), hence hardware keys. But current systems are far more flexible and due to that this design can also refer to software buttons rendered by the system UI outside of the applications control or touch sensitive areas with software controlled functionality.

As a simple guideline to determine if a control falls under this design concept is to sample ask the question: “Could this functionality have provided by a physical key/knob”. If the answer is yes, it fits into this design otherwise it doesn’t.

Out of scope

For the current design the following aspect our defined as out of scope. They are either addressed belong to other designs or could be addressed in future iterations.

- Haptic feedback
- Application controlled buttons (e.g. application configurable icons)
- short vs. long key presses handling (part of the UI framework/toolkit)
- Display systems other then wayland
- Any requirements about the type of controls that need to be available.
- Implementation design; This document explains the high-level concepts not the implementation

Types of usage & example use-cases

We recognize three different types of controls and the effect that they have on the system:

- System controls: Effect the system as a whole (volume controls, home key).
- Role controls: Effect the current system context (pause/play)
- Application controls: Effect the current foreground application

The following sections provide more detail for these various types of controls.

System keys

Buttons handled by "the system". Regardless of application state. Example volume controls (always adjust the system volume), home screen buttons (always navigate back to the home screen), application shortcut buttons e.g. navigation (always open the specific application).

example use-cases

- Bob is using a webbrowser, presses the "home button". The system goes back to the homescreen.
- Alice presses the "navigation"button. The system opens the navigation application

Role specific keys

Buttons that should be handled by an agent or application furfilling a specific role. An example here are play/pause buttons which should get handled by the program currently playing audio (e.g. internet radio application, local media player etc).

Example use-case

- Simon starts playing music in the spotify application, switches to the web-browser application while the music streams keeps playing in the background. Simon doesn't like the current song and presses the "next song"

button, the spotify agent running in the background switches to the next song.

Application keys

Buttons that should always be handled by the currently active application. For example up/down/left/right select buttons.

Example use-case

- Harry is browsing through a list of potential destinations in the navigation application. He turns the rotary dail on the center console, the focus moves to the next potential destination on the list.
- Barry looks up a new radio station in the radio application. After listening a while he decides he likes the current station. Barry hold the "favourites" button for a while (long press), the radio application bookmarks the current station.

Non-functional requirements

Apart from the use-cases mentioned above, there are several requirements on the design that don't directly impact the functionality but are important from e.g. a security point of view.

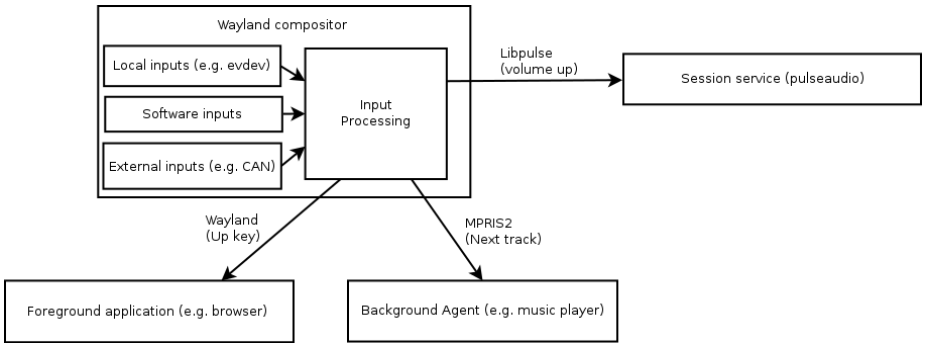
- Applications should not be apple to eavesdrop on the input send to other processes
- Applications should not be able to injects inputs into other processes (Synthesized input[1])

A more complete overview of issues surrounding input security (integrity, confidentiality) be found on the Compositor security[2] page.

Design

On wayland systems the design of inputs is relatively straightforward, the compositor is responsible for gathering all the inputs from the various sources and chooses how to handle them. The diagram below has a high-level overview of some example input sources and examples of the various types of controls.

[1] https://www.apertis.org/concepts/compositor_security/#Synthesize_input
[2] https://www.apertis.org/concepts/compositor_security/



Thanks to the Wayland input flow only having two actors (the compositor and the receiver) (as opposed to that of X11) that there is no way for applications to either spy on the inputs of other applications or to inject inputs into other applications.

Compositor Input sources

Various example inputs are shown in the diagram (though others could be defined as well):

- Local inputs: evdev is the kernel input subsystem, directly attached controls will emit key events using that subsystem (e.g. i2c attached buttons)
- External inputs: Any input sources that aren't directly attached, e.g. inputs via the CAN network or even an IP network
- Software inputs: Software defined input sources, e.g. onscreen buttons drawn by the compositor or pre-defined touchscreen areas

Note that the exact implementation of gathering input is left up to the implementer. E.g. for CAN inputs it's not recommended for the compositor to have direct access to the CAN bus, however that design is part the implementation of the generic CAN handling.

Each of these events input will feed into the input processing internally into the compositor.

Compositor Input processing

All input events are gathered into one consistent way of input processing in the compositor. From both the compositor internals and the applications/agents point of view the exact technology to gather the inputs is not relevant (Note that the device could be, e.g. steering wheel controls vs center console controls).

The task of the input processing into the compositor is to determine where the key event should be delivered and via which method. Following the classification outlined earlier.

System keys

Keys meant for the system are processed directly by the compositor. Resulting in the compositor taking an action either purely by itself (e.g. switching to an overview screen) or by calling methods on external services. In the example given the diagram, the compositor processes the “volume up”key and as a result of that uses the libpulse API to ask pulseaudio to increase the system volume.

Application keys

Keys meant for the current foreground application are simply forward to the application using the wayland protocol. All further handling is up to the applications and its toolkit, this includes but is not limited to the recognition of short-press vs. long-press, application specific interpretation of keys (e.g. opening the bookmarks if the favourites key is pressed) etc.

Note that the current foreground application might well be the compositor itself. For example if the compositor is responsible of rendering the status bar, it could be possible to use key navigation to navigate the statusbar.

Role keys

Keys for a specific role are forwarded to the application or agent current furfilling that role. It is expected that each role implemements a role-specific D-Bus interface either for direct key handling or commands.

For example for music players, assuming the MPRIS2 is mandated for media agents (including Radio), the compositor would determine which agent or application currently has the Media role and call the MPRIS2 method corresponding to the key on it (e.g. Next).

The reason for requiring role-specific D-Bus interfaces rather than simply forwarding keys via e.g. the wayland protocol is that an agent doesn't have to connect to the display server, only to the D-Bus session bus. It also means there is a clean separation between the actual key handling and the action taken. For example an OEM may define a short press on a “forward”key meaning a seek, while a long-press means “next track”, in this design that can purely be policy in the compositor.

Key handling recommendations

Even though it is out of scope for this design specifically. Some general recommendations about key handling:

- Keys should be specific to one category even when used in long vs. short press or in combinations. As undoing key events is impossible.

- Any combination keys or keys with long-press abilities should only be handled on key release (key down ignored), for the same reasoning. Cancelling an in-progress action is either confusing to the user or not possible.