Push Button Service:

A service for Android

V 1.0

Stephan Greto-McGrath

August 09, 2016

Contents

Overview

The Push Button Service is a simple, portable, service APK for Android that serves as an interface between a push button and the android device. The service detects logic level changes on GPIO pins through the sysfs interface and determines whether the button is up or down, and whether it has received a short, long or double press. This information is then broadcast as an intent so that other applications may use the input as they see fit. The broadcasts for button state will be one of two possibilities, either `com.google.hal.BUTTON_UP_X` or `com.google.hal.BUTTON_DOWN_X`. The type of press will be broadcast as `com.google.hal.SHORT_X`, `com.google.hal.LONG_X` or `com.google.hal.DOUBLE_X`, where `X` corresponds to the button number the even occurred on.

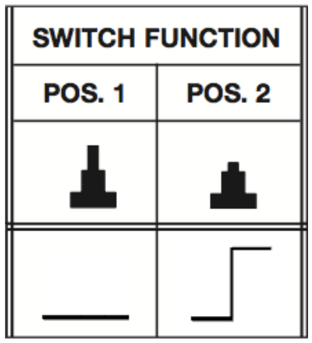
The service runs on boot, using a `BroadcastReceiver` that receives the system's intent: `android.intent.action.BOOT_COMPLETED`. It can also be manually initiated using `ServiceLauncher.java` which is simply a launcher that will start the service running.

Push Button Functionality

Types of presses

The service is capable of detecting three distinct types of presses: a short press, a long press, and a double press. The waveforms corresponding to button presses can be seen in Figure 1.

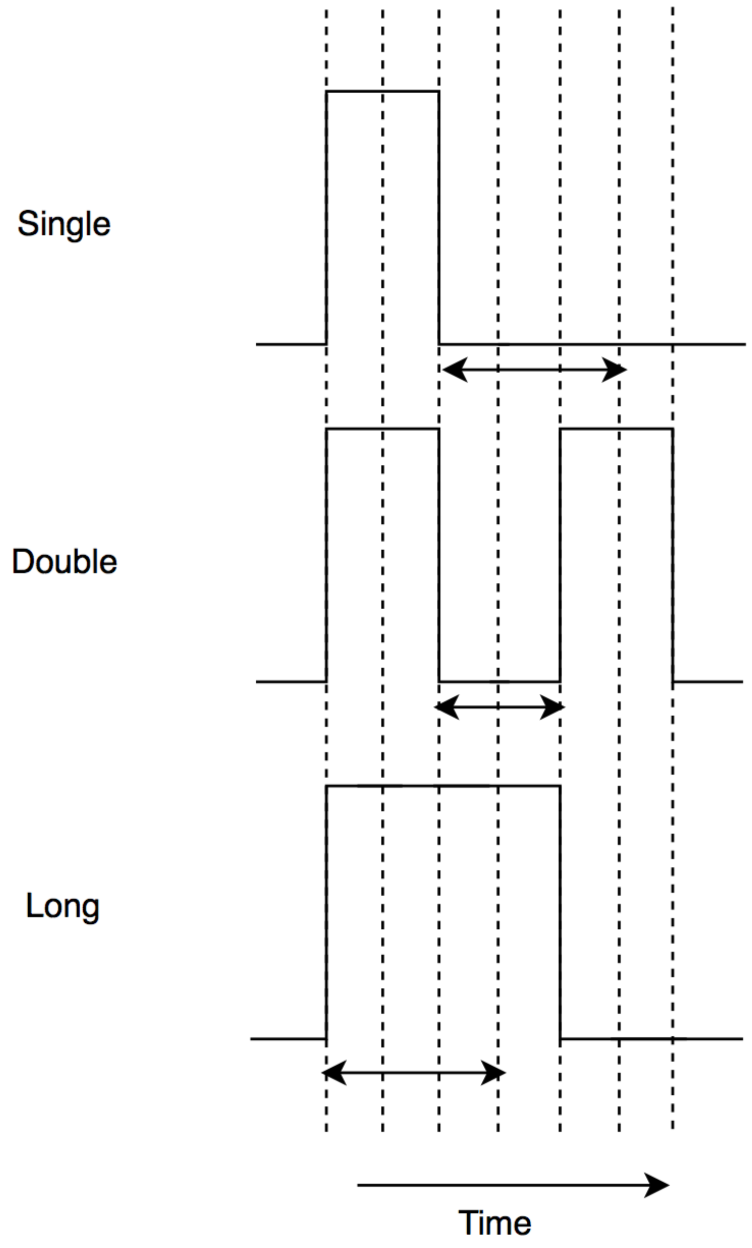
Figure 1: Push Button Logic Levels



NOTE: The software recognizes a logic 1 as the button being pressed down. If a pulldown switch is being used, the logic level of the GPIO pin can be inverted using `active_low`. For more details, please consult GPIO Setup under Platform Requirements.

How the service determines what type of press has been received can be seen through the waveform diagrams in Figure 2. The two-headed arrows illustrate where the elapsed time is used to determine what type of button action has occurred. For a single press, the time after the falling edge is important. If a long enough period has elapsed, the service determines that the delay has been too long to justify the next input as being the latter half of a double press. Conversely, for the double press, if a second rising edge is received after the falling edge within a small interval, the service concludes that a double press has been input. For the long press, the time after a rising edge and before a falling edge is most important. If the delay is long enough, the service will conclude that the button is being held down.

Figure 2: Push Button Waveforms

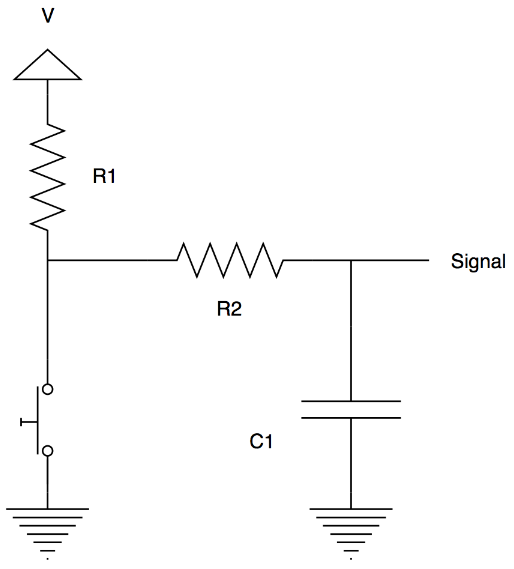


Connecting the Push Button

Standard Test Circuitry

Typically, a push button is connected to a microcontroller (MCU) as a pull-down switch to GND with a lowpass filter to handle of switch bounce. This can be seen below in Figure 3.

Figure 3: Push Button Circuit



The values for R1, R2 and C1 have been left unspecified as they should be set according to the button and the connected MCU. Many MCUs have the means to deal with switch bounce without including the lowpass filter. Depending on the platform being used, the lowpass may be unnecessary as the MCU will most likely contain a Schmitt Trigger as well as some sort of of configurable debounce implemented in the firmware. Therefore, the specifics of connecting the rotary encoder will be at the discretion of the developer and based on their platform.

Platform Requirements

GPIO Setup

The service requires as many exposed GPIO pins as desired buttons, accessible through the `sysfs` interface. Currently, the service is set to use two buttons, although this can be expanded to N buttons by simply changing the size of the necessary data structures as well as specifying more GPIO pins in the source code. Below is a summary of what needs to be done through the `adb shell` in order to export and configure the GPIO pins. For each pin (XX is the pin number):

```
echo XX > /sys/class/export
echo in > /sys/class/gpioXX/direction
echo both > /sys/class/gpio/gpioXX/edge
```

If the switch is a pulldown to GND:

```
echo 1 > /sys/class/gpio/gpioXX/active_low
```

These pin numbers must be specified accordingly in the source, in the C code: `button_service.c` via the `static const int` array `gpios[]`.

Root Access

If the developer would like to have the GPIOs exported and their edges configured automatically upon boot, the app will require root access. In order to enable this functionality, the developer simply has to remove the commenting surrounding the `setup_gpios();` function call in the C code made available through export to the JNI.

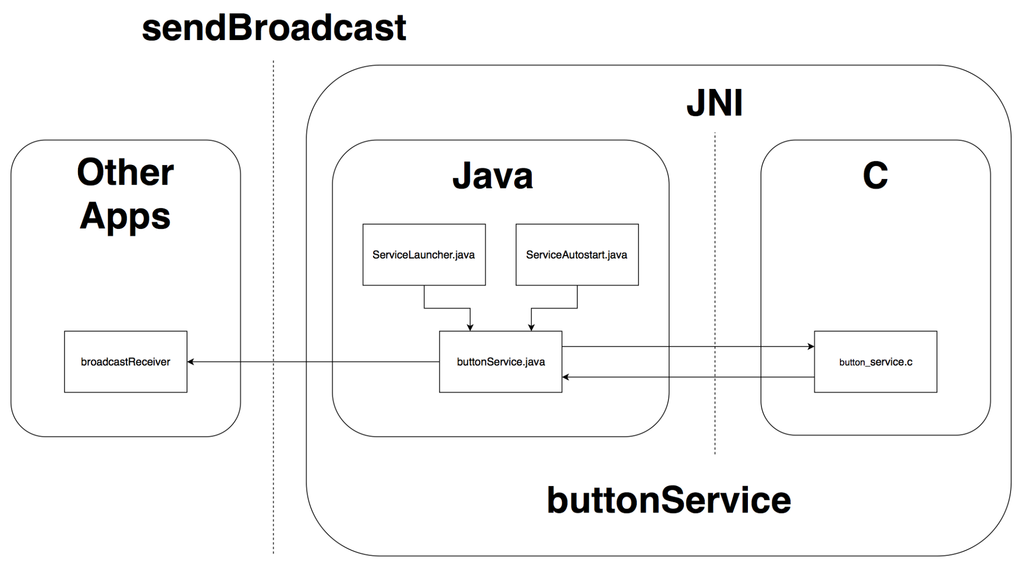
Note: If installed in `/system/app`, the launcher will no longer appear in the app drawer. This shouldn't be a problem as the service starts on boot using a `BroadcastReceiver`. Furthermore, the `setup_gpios();` function call must be uncommented, or the GPIOs must be exported and configured on boot by some other means such as with `init.rc` or the service will fail to run on boot.

Under the Hood

Code Structure

Figure 4 presents a high level overview of the service's components and how they interact with each other and the outside world (through GPIO) or other apps (using broadcasts).

Figure 4: Software Component Overview



The service can begin one of two ways, either it is initiated on boot when `ServiceAutostart.java` receives a signal that boot has completed using the `BroadcastReceiver`, or when `ServiceLauncher.java` is initiated by the user by clicking on the app's launcher. In both cases, `buttonService.java` begins to run. This, in turn, makes a call through JNI to `button_service.c` which begins a new thread and attaches it to the JVM.

This new thread uses the `poll()` function as interrupt detection on the `sysfs` GPIO values. When a change occurs, it begins its service routine. This routine uses the logic outlined in the Push Button Functionality section to determine which type of press has occurred. Then the native code calls java which broadcasts an intent. These can be picked up by other applications using a `broadcastReceiver` in order to act upon inputs. Broadcasts will be of the form: `com.google.hal.BUTTON_UP_X`, `com.google.hal.BUTTON_DOWN_X`, `com.google.hal.SHORT_X`, `com.google.hal.LONG_X` or `com.google.hal.DOUBLE_X`, where X corresponds to the position of the GPIO pin in the array `gpios[]` and serves as an indicator as to which button was pressed.

Troubleshooting

Installing Manually in `/system/app`

Assuming the app is signed with the necessary signature to run as a system app, it can be manually installed using the following procedure and `adb shell`.

1. The system must first be made writeable. In order to do this, use:
   `adb shell mount -o rw,remount,rw /system`
2. The APK must be moved to the correct location on the device using:
   `adb push /location/of/signed-app.apk /system/app/signed-app.apk`
3. If you would like to adjust the permissions of the app to correspond to those of the other system apps, navigate to `/system/app` and use `chmod`.
4. There is a chance the device will not detect the native library. In order to solve this issue:
   (a) Add `.zip` to the APK on the computer in order to be able to unzip it.
   (b) Once unzipped, navigate to `/lib*/*target architecture*` and move `libbutton_service.so` to `/system/lib` on the device using `adb push`
   (c) If you like, you can also use `chmod` here to match the permissions of the native library to those of the device.
5. Restart the device in order to let the system install the app. On the second restart, the app should be functional. Check the `logcat` in order to confirm that it has begun running and is not experiencing any issues.

`poll()` Function Defective on Certain Devices

This shouldn't normally present an issue, however, while developing the service, it was noted that on certain devices, `poll()` was either not returning at all with `POLLPRI` and delay set to `-1` or returning immediately with `POLLIN` no matter the delay.

If all is well with `POLLPRI`, this is ideal. If however the function refuses to return, it can simply be changed to `POLLIN`. In this case, an extra check already in the code will ensure a state change prior to any further processing. This fixes the issue, albeit with added overhead.