Rotary Encoder Service:

A service for Android

V 1.3

Stephan Greto-McGrath

August 08, 2016

Contents

Overview 3

Rotary Encoder Functionality 3
Quadrature 3
Finite State Machine 4

Connecting the Encoder 5
Standard Test Circuitry 5

Platform Requirements 6
GPIOs 6
Root Access 6

Under the Hood 7
Code Structure 7

Troubleshooting 8
Installing Manually in `/system/app` 8
`poll()` Function Defective on Certain Devices 8

Overview

The Rotary Encoder Service is a simple, portable, service APK for Android that serves as an interface between a rotary encoder and the android device. The service detects logic level changes on GPIO pins through the sysfs interface and uses a Finite State Machine (FSM) in order to detect which direction the encoder is turning in. This direction is then broadcast as an intent so that other applications may use the input as they see fit. The broadcast will be either `com.google.hal.CLOCKWISE` or `com.google.hal.COUNTER_CLOCKWISE`

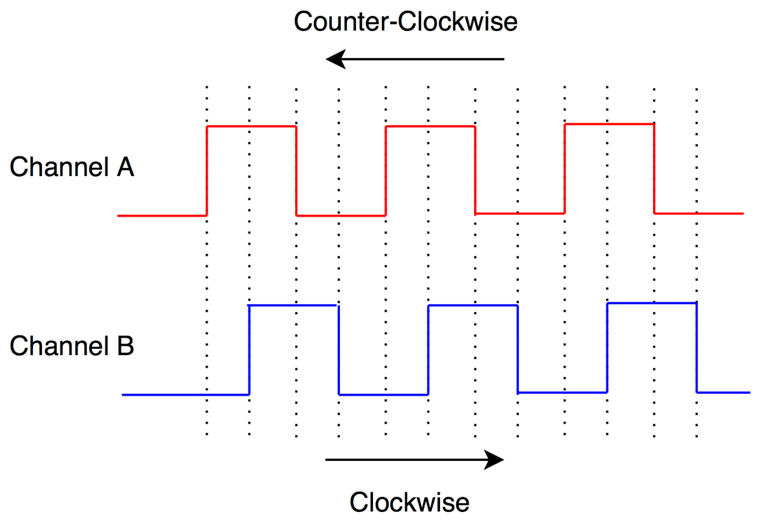
The service runs on boot, using a `BroadcastReceiver` that receives the system's intent: `android.intent.action.BOOT_COMPLETED`. It can also be manually initiated using `ServiceLauncher.java` which is simply a launcher that will start the service running.

Rotary Encoder Functionality

Quadrature

The rotary encoder for which this service was designed was one with a quadrature output. This means that the two signals (A & B) are 90° out of phase. Reading the two channels at subsequent intervals, we can determine whether the encoder is being turned clockwise or counter-clockwise.

Figure 1: Rotary Encoder Waveform



Finite State Machine

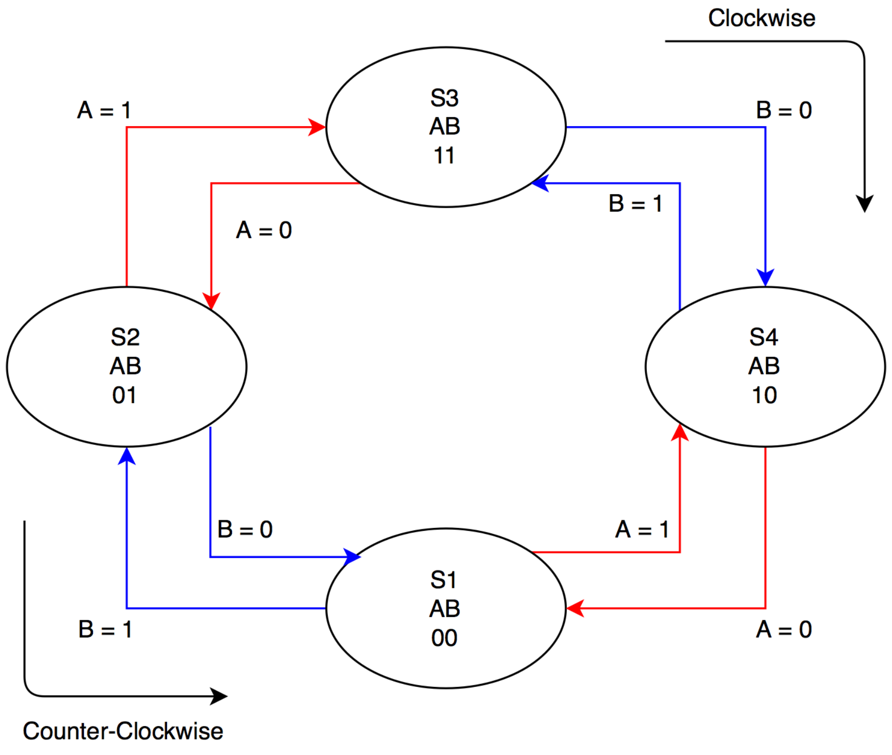
The above waveforms can be put into a state transition table. Below "/" refers to an invalid transition while "CW" and "CCW" refer to clockwise and counter-clockwise respectively.

AB	A A=0	 A=1	 Dir	B B=0	 B=1	 Dir
S1/00	/	**S4**/10	CCW	/	**S2**/01	CW
S2/01	/	**S3**/11	CW	**S1**/00	/	CCW
S3/11	**S2**/01	/	CCW	**S4**/10	/	CW
S4/10	**S1**/00	/	CCW	/	**S3**/11	CW

Table 1: State Transition Table

From the state transition table, we can construct a state transition diagram. This visual reference describes the behavior implemented in the service.

Figure 2: State Transition Diagram

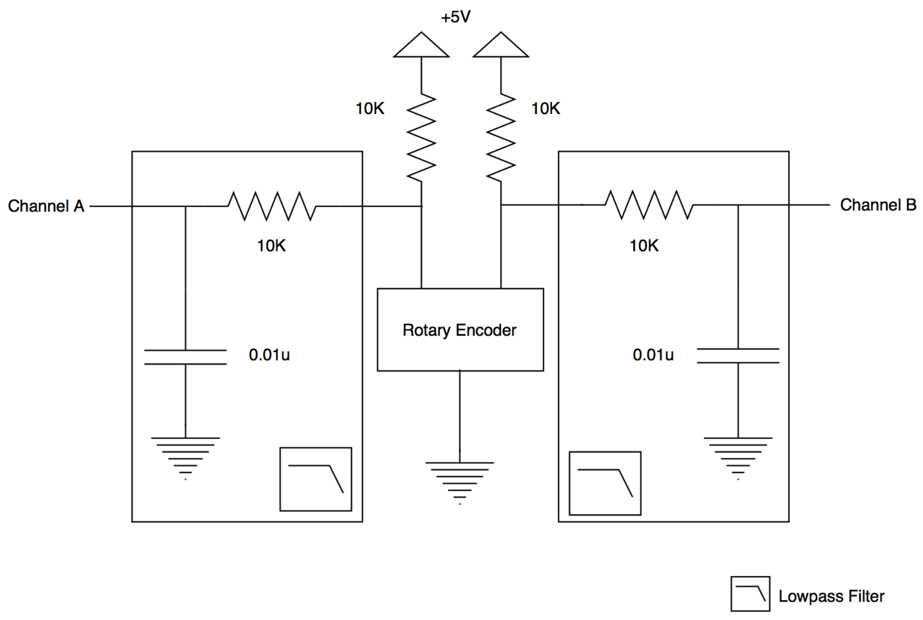


Connecting the Encoder

Standard Test Circuitry

Typically, a rotary encoder is connected to a microcontroller (MCU) as a pull-down switch to GND with a lowpass filter to take care of switch bounce. This can be seen in Figure 3.

Figure 3: Rotary Encoder Circuit



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.

In the case of the platform that was used to develop the service, the lowpass filter was not necessary, and the 5V was actually only 1.8V (the maximum for the device). Introducing a lowpass filter on this device was unnecessary.

Platform Requirements

GPIOs

The encoder requires two exposed GPIO pins accessible through the `sysfs` interface. 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
```

These pin numbers must be specified accordingly in the source, in the C code: `rotary-encoder-service.c` via the `static const int` provided for both `gpio1` and `gpio2`

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(gpio1, gpio2);` 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(gpio1, gpio2);` function call must be un-commented, 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.

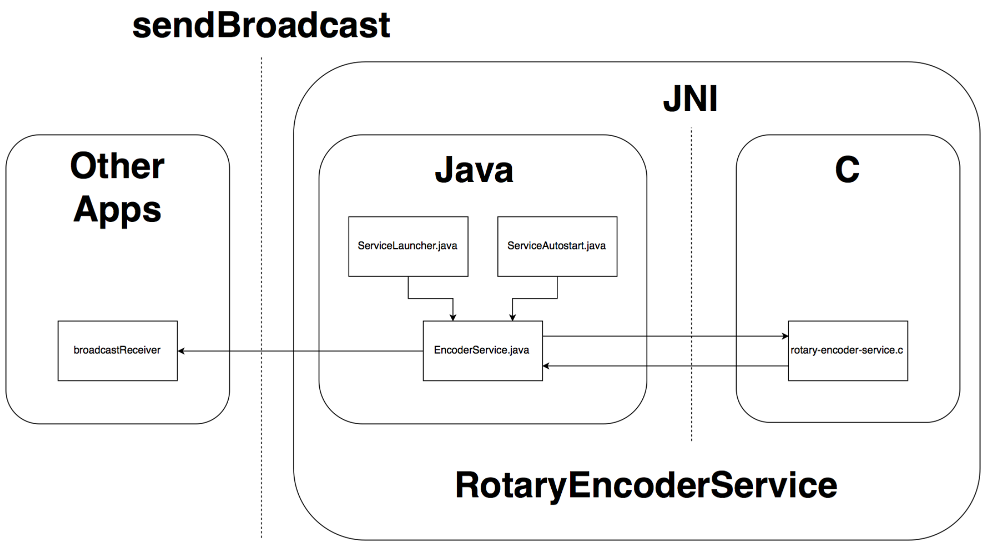
Under the Hood

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).

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, `EncoderService.java` begins to run. This, in turn, makes a call through JNI to `rotary-encoder-service.c` which begins a new thread and attaches it to the JVM.

Figure 4: Software Component Overview



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 puts the past and current state through the finite state machine to determine what direction the encoder has been turned. A call to Java is then made with this information through the JNI. Java then broadcasts one of two intents: either `com.google.hal.CLOCKWISE` or `com.google.hal.COUNTER_CLOCKWISE`. These can be picked up by other applications using a `broadcastReceiver` in order to act upon inputs.

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 `librotary-encoder-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.