内容目录

CarPower休眠唤醒后，偶现关机/重启选择的界面，这个问题需要调查一下

# 调查一、了解input的大概数据流

kernel/include/dt-bindings/input/linux-event-codes.h

#define KEY\_POWER 116 /\* SC System Power Down \*/

device/chehejia/M01\_AE/qpnp\_pon.kl

key 116 POWER

frameworks/base/services/java/com/android/server/SystemServer.java

import com.android.server.input.InputManagerService;

import com.android.server.wm.WindowManagerService;

import com.android.server.policy.PhoneWindowManager;

InputManagerService inputManager = null;

//阶段一、初始化输入服务

inputManager = new InputManagerService(context);

wm = WindowManagerService.main(context, inputManager,

mFactoryTestMode != FactoryTest.FACTORY\_TEST\_LOW\_LEVEL, //工厂模式否

!mFirstBoot, mOnlyCore, new PhoneWindowManager()); //是否是首次开机，是否只是解析CoreApp

ServiceManager.addService(Context.INPUT\_SERVICE, inputManager);//“input”

//阶段二、启动输入服务

inputManager.start();

**阶段一、InputManager的初始化**

frameworks/base/services/core/java/com/android/server/input/InputManagerService.java

// Pointer to native input manager service object.

private final long mPtr;

private static native long nativeInit(InputManagerService service, Context context, MessageQueue messageQueue);

mPtr = nativeInit(this, mContext, mHandler.getLooper().getQueue());//通过JNI调用来启动native层的input系统，然后把返回值存放在mPtr中

frameworks/base/services/core/jni/com\_android\_server\_input\_InputManagerService.cpp

//这里实例化了NativeInputManagerService的一个对象，使用的Java层的MessageQueue的Looper，意味着Java层消息和Native消息是在同一个MessageQueue中的

static jlong nativeInit(JNIEnv\* env, jclass /\* clazz \*/,

jobject serviceObj, jobject contextObj, jobject messageQueueObj) {

sp<MessageQueue> messageQueue = android\_os\_MessageQueue\_getMessageQueue(env, messageQueueObj);

NativeInputManager\* im = new NativeInputManager(contextObj, serviceObj, messageQueue->getLooper());

im->incStrong(0);//增加对象引用计数

return reinterpret\_cast<jlong>(im);

}

NativeInputManager::NativeInputManager(jobject contextObj, jobject serviceObj, const sp<Looper>& looper) : mLooper(looper), mInteractive(true) {

sp<EventHub> eventHub = new EventHub();

mInputManager = new InputManager(eventHub, this, this);

}

frameworks/native/services/inputflinger/InputManager.cpp

InputManager::InputManager(

const sp<EventHubInterface>& eventHub,

const sp<InputReaderPolicyInterface>& readerPolicy,

const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {

mDispatcher = new InputDispatcher(dispatcherPolicy);

mReader = new InputReader(eventHub, readerPolicy, mDispatcher);

initialize();

}

void InputManager::initialize() {

mReaderThread = new InputReaderThread(mReader);//InputReaderThread线程，负责input事件的获取

mDispatcherThread = new InputDispatcherThread(mDispatcher);//InputDispatcherThread线程，负责input消息的发送

}

**阶段二、InputManager的启动**

切回InputManagerServer.java的start方法

public void start() {

nativeStart(mPtr);

}

com\_android\_server\_input\_InputManagerService.cpp

static void nativeStart(JNIEnv\* env, jclass /\* clazz \*/, jlong ptr) {

NativeInputManager\* im = reinterpret\_cast<NativeInputManager\*>(ptr);

status\_t result = im->getInputManager()->start();

if (result) {

jniThrowRuntimeException(env, "Input manager could not be started.");

}

}

InputManager.cpp

status\_t InputManager::start() {//同时给了PRIORITY\_URGENT\_DISPLAY这个级别跟SurfaceFlinger的priority是一样，可见是及时响应的

status\_t result = mDispatcherThread->run("InputDispatcher", PRIORITY\_URGENT\_DISPLAY);

if (result) {

ALOGE("Could not start InputDispatcher thread due to error %d.", result);

return result;

}

result = mReaderThread->run("InputReader", PRIORITY\_URGENT\_DISPLAY);

return OK;

}

**InputDispatcherThread**

InputDispatcherThread继承的是Thread类，由于它是Thread子类，于是继承它的run方法，进入run方法后会调用threadLoop()，在Thread类中它是虚函数，得由子类来复写，如下所示

bool InputDispatcherThread::threadLoop() {

mDispatcher->dispatchOnce();

return true;

}

void InputDispatcher::dispatchOnce() {

nsecs\_t nextWakeupTime = LONG\_LONG\_MAX;

{ // acquire lock

AutoMutex \_l(mLock);

mDispatcherIsAliveCondition.broadcast();

// Run a dispatch loop if there are no pending commands.

// The dispatch loop might enqueue commands to run afterwards.

if (!haveCommandsLocked()) {

dispatchOnceInnerLocked(&nextWakeupTime);//事件分发

}

// Run all pending commands if there are any.

// If any commands were run then force the next poll to wake up immediately.

if (runCommandsLockedInterruptible()) {//事件执行

nextWakeupTime = LONG\_LONG\_MIN;

}

} // release lock

// Wait for callback or timeout or wake. (make sure we round up, not down)

nsecs\_t currentTime = now();

int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);

mLooper->pollOnce(timeoutMillis);//获取键盘事件，如果没有消息的话就是阻塞epoll，底层的实现是基于pipe机制的

}

InputDispatch会调用dispatchOnceInnerLocked去分发消息；在消息分完完成后，又使用mLooper继续获取消息。当mLooper.pollOnce有读取到消息的时候就会往管道里头写新的内容，唤醒正在等待键盘事件的线程；当没有消息的时候，就无限的阻塞着，线程进入空闲等待状态

**InputReaderThread**

InputReader.cpp

bool InputReaderThread::threadLoop() {

mReader->loopOnce();

return true;

}

void InputReader::loopOnce() {

int32\_t oldGeneration;

int32\_t timeoutMillis;

bool inputDevicesChanged = false;

Vector<InputDeviceInfo> inputDevices;

{ // acquire lock

AutoMutex \_l(mLock);

oldGeneration = mGeneration;

timeoutMillis = -1;

uint32\_t changes = mConfigurationChangesToRefresh;

if (changes) {

mConfigurationChangesToRefresh = 0;

timeoutMillis = 0;

refreshConfigurationLocked(changes);

} else if (mNextTimeout != LLONG\_MAX) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

timeoutMillis = toMillisecondTimeoutDelay(now, mNextTimeout);

}

} // release lock

//阶段一，通过EventHub的getEvents方法来获取input事件，/dev/input/event\*

size\_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT\_BUFFER\_SIZE);

{ // acquire lock

AutoMutex \_l(mLock);

mReaderIsAliveCondition.broadcast();

if (count) {

processEventsLocked(mEventBuffer, count);//阶段二，开始处理读取出来的元事件

}

if (mNextTimeout != LLONG\_MAX) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

if (now >= mNextTimeout) {

#if DEBUG\_RAW\_EVENTS

ALOGD("Timeout expired, latency=%0.3fms", (now - mNextTimeout) \* 0.000001f);

#endif

mNextTimeout = LLONG\_MAX;

timeoutExpiredLocked(now);

}

}

if (oldGeneration != mGeneration) {

inputDevicesChanged = true;

getInputDevicesLocked(inputDevices);

}

} // release lock

// Send out a message that the describes the changed input devices.

if (inputDevicesChanged) {

mPolicy->notifyInputDevicesChanged(inputDevices);

}

mQueuedListener->flush();//阶段三，把QueuedInputListener中的消息全部都开始处理

}

InputListener.cpp

void QueuedInputListener::flush() {

size\_t count = mArgsQueue.size();

for (size\_t i = 0; i < count; i++) {

NotifyArgs\* args = mArgsQueue[i];

args->notify(mInnerListener);//mlnnerListener是InputDispatcher对象

delete args;

}

mArgsQueue.clear();

}

小结一下InputReader 这三个阶段

1、通过EventHub的getEvents方法来获取input事件。

EventHub负责打开/dev/input/目录下的所有设备，然后为每一个设备创建一个Device，并把这个Device放入EventHub所定义的数组们Device中。之后，就是把这个设备纳入监视范围。然后就是开始等待事件的发生，一旦有事件发生，就从产生事件的设备中读取出这些设备，把这些事件转化为RawEvent类型放入InputReader提供的事件数组中，之后返回。到这里，从EventHub获取事件就结束了。

2、从EventHub中读取出若干事件，然会对这些事件进行预处理。

3、把QueuedInputListener中的事件分发出去。

# 调查二、long press key debug log

EventHub的数据流里未提供是谁触发的长按，所以现在先在framework层加一下long press key debug log，问题复现是有log可查

device/chehejia/M01\_AE/qpnp\_pon.kl

key 116 POWER

frameworks/native/include/input/InputEventLabels.h

DEFINE\_KEYCODE(POWER),

frameworks/native/include/android/keycodes.h

AKEYCODE\_POWER = 26,

framework/base/services/core/java/com/android/server/policy/PhoneWindowManager.java

case KeyEvent.KEYCODE\_POWER: {

Slog.w(TAG, "interceptKeyBeforeQueueing: KeyEvent.KEYCODE\_POWER debug for PowerStateMachineImpl test!!!");

/\*

// Any activity on the power button stops the accessibility shortcut

cancelPendingAccessibilityShortcutAction();

result &= ~ACTION\_PASS\_TO\_USER;

isWakeKey = false; // wake-up will be handled separately

if (down) {

interceptPowerKeyDown(event, interactive);

} else {

interceptPowerKeyUp(event, interactive, canceled);

}

\*/

break;

}

10-19 04:43:56.161 1286 1465 D PowerStateMachineImpl: recv MCU\_PWR\_QUERY(2)

10-19 04:44:41.900 1286 1465 D PowerStateMachineImpl: Recv NM mode 0

10-19 04:44:41.927 1286 1796 D PowerStateMachineImpl: Transition from PRESLEEP to SLEEP

10-19 04:45:04.268 562 860 W WindowManager: interceptKeyBeforeQueueing: KeyEvent.KEYCODE\_POWER mask for PowerStateMachineImpl!!!

10-19 04:45:04.315 562 860 W WindowManager: interceptKeyBeforeQueueing: KeyEvent.KEYCODE\_POWER mask for PowerStateMachineImpl!!!

10-19 04:45:04.410 1286 1796 D PowerStateMachineImpl: Transition from SLEEP to PARTLYWORK

10-19 04:45:04.937 1286 1465 D PowerStateMachineImpl: recv wakeup reason (51)

10-19 04:45:21.673 1286 1465 D PowerStateMachineImpl: Recv NM mode 3

# 调查三、关机界面触发

frameworks/base/services/core/java/com/android/server/policy/PhoneWindowManager.java

private void powerLongPress() {//长按处理的接口

final int behavior = getResolvedLongPressOnPowerBehavior();

Slog.w(TAG, "powerLongPress mask for PowerStateMachineImpl, behavior = " + behavior);

switch (behavior) {

case LONG\_PRESS\_POWER\_NOTHING:

break;

case LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS:

Slog.w(TAG, "LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS mask for PowerStateMachineImpl");

mPowerKeyHandled = true;

performHapticFeedbackLw(null, HapticFeedbackConstants.LONG\_PRESS, false);

showGlobalActionsInternal();//触发关机界面

break;

case LONG\_PRESS\_POWER\_SHUT\_OFF:

case LONG\_PRESS\_POWER\_SHUT\_OFF\_NO\_CONFIRM:

...

break;

}

}