RIL & HIDL: <https://blog.csdn.net/hwmagicok/article/details/113917658>

关键LOG标红，关键代码标黄

======================MT SMS======================

接收短信流程，ELT关键LOG:

[NW->MS] [MT] SMS\_\_RP\_DATA (rl\_id=3)

GSM SMS TPDU (GSM 03.40) SMS-DELIVER

radio LOG:

RILJ : [UNSL]< UNSOL\_RESPONSE\_NEW\_SMS [PHONE0]

11-12 10:32:58.034477 1362 1561 D MtkGsmInboundSmsHandler-0: IdleState.processMessage: processing EVENT\_NEW\_SMS

11-12 10:32:58.036659 1362 1561 D MtkGsmInboundSmsHandler-0: IdleState.exit: acquired wakelock, leaving IdleState

11-12 10:32:58.036848 1362 1561 D MtkGsmInboundSmsHandler-0: DeliveringState.enter: entering DeliveringState

11-12 10:32:58.037002 1362 1561 D MtkGsmInboundSmsHandler-0: DeliveringState.processMessage: processing EVENT\_NEW\_SMS

11-12 10:32:58.095261 1362 1561 D MtkGsmInboundSmsHandler-0: addTrackerToRawTable: URI of new row: content://sms/raw/1, id: 4698691570883853257

main LOG:

SmsReceiver.deliverSmsMessages

1. /v790/LA.QSSI.12.0.r1/LINUX/android/hardware/ril/libril/ril\_unsol\_commands.h

{RIL\_UNSOL\_RESPONSE\_NEW\_SMS, radio::newSmsInd, WAKE\_PARTIAL},

2. /v790/LA.QSSI.12.0.r1/LINUX/android/hardware/ril/libril/ril\_service.cpp

int radio::newSmsInd(int slotId, int indicationType,

int token, RIL\_Errno e, void \*response, size\_t responseLen) {

if (radioService[slotId] != NULL && radioService[slotId]->mRadioIndication != NULL) {

if (response == NULL || responseLen == 0) {

RLOGE("newSmsInd: invalid response");

return 0;

}

uint8\_t \*bytes = convertHexStringToBytes(response, responseLen);

if (bytes == NULL) {

RLOGE("newSmsInd: convertHexStringToBytes failed");

return 0;

}

hidl\_vec<uint8\_t> pdu;

pdu.setToExternal(bytes, responseLen/2);

#if VDBG

RLOGD("newSmsInd");

#endif

Return<void> retStatus = radioService[slotId]->mRadioIndication->newSms(

convertIntToRadioIndicationType(indicationType), pdu);

radioService[slotId]->checkReturnStatus(retStatus);

free(bytes);

} else {

RLOGE("newSmsInd: radioService[%d]->mRadioIndication == NULL", slotId);

}

return 0;

}

3. /v790/LA.QSSI.12.0.r1/LINUX/android/hardware/interfaces/radio/1.0/IradioIndication.hal

oneway newSms(RadioIndicationType type, vec<uint8\_t> pdu);

4. HIDL /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/RadioIndication.java

public class RadioIndication extends IRadioIndication.Stub {}

**public void** newSms(**int** indicationType, ArrayList<Byte> pdu) {  
 mRil.processIndication(indicationType);  
  
 **byte**[] pduArray = RIL.arrayListToPrimitiveArray(pdu);  
 **if** (RIL.RILJ\_LOGD) mRil.unsljLog(RIL\_UNSOL\_RESPONSE\_NEW\_SMS);  
  
 SmsMessageBase smsb = com.android.internal.telephony.gsm.SmsMessage.createFromPdu(pduArray);  
 **if** (mRil.mGsmSmsRegistrant != **null**) {  
 mRil.mGsmSmsRegistrant.notifyRegistrant(  
 **new** AsyncResult(**null**, smsb == **null** ? **null** : **new** SmsMessage(smsb), **null**));  
 }  
}

notifyRegistrant跳转到第8步

5. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/RIL.java

@UnsupportedAppUsage  
**void** unsljLog(**int** response) {  
 riljLog(**"[UNSL]< "** + responseToString(response));  
}

@UnsupportedAppUsage  
**static** String responseToString(**int** request) {  
 **switch**(request) {  
 **case** RIL\_UNSOL\_RESPONSE\_RADIO\_STATE\_CHANGED:  
 **return "UNSOL\_RESPONSE\_RADIO\_STATE\_CHANGED"**;  
 **case** RIL\_UNSOL\_RESPONSE\_CALL\_STATE\_CHANGED:  
 **return "UNSOL\_RESPONSE\_CALL\_STATE\_CHANGED"**;  
 **case** RIL\_UNSOL\_RESPONSE\_NETWORK\_STATE\_CHANGED:  
 **return "UNSOL\_RESPONSE\_NETWORK\_STATE\_CHANGED"**;  
 **case** RIL\_UNSOL\_RESPONSE\_NEW\_SMS:  
 **return "UNSOL\_RESPONSE\_NEW\_SMS"**;  
 **case** RIL\_UNSOL\_RESPONSE\_NEW\_SMS\_STATUS\_REPORT:  
 **return "UNSOL\_RESPONSE\_NEW\_SMS\_STATUS\_REPORT"**;  
 **case** RIL\_UNSOL\_RESPONSE\_NEW\_SMS\_ON\_SIM:  
 **return "UNSOL\_RESPONSE\_NEW\_SMS\_ON\_SIM"**;

...

}

}

6. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/gsm/GsmInboundSmsHandler.java

*/\*\**  
 *\* Create a new GSM inbound SMS handler.*  
 *\*/*  
**private** GsmInboundSmsHandler(Context context, SmsStorageMonitor storageMonitor,  
 Phone phone) {  
 **super**(**"GsmInboundSmsHandler"**, context, storageMonitor, phone);  
 phone.mCi.setOnNewGsmSms(getHandler(), EVENT\_NEW\_SMS, **null**);  
 mDataDownloadHandler = **new** UsimDataDownloadHandler(phone.mCi, phone.getPhoneId());  
 mCellBroadcastServiceManager.enable();  
  
 **if** (TEST\_MODE) {  
 **if** (sTestBroadcastReceiver == **null**) {  
 sTestBroadcastReceiver = **new** GsmCbTestBroadcastReceiver();  
 IntentFilter filter = **new** IntentFilter();  
 filter.addAction(TEST\_ACTION);  
 context.registerReceiver(sTestBroadcastReceiver, filter);  
 }  
 }  
}

7. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/BaseCommands.java

@Override  
**public void** setOnNewGsmSms(Handler h, **int** what, Object obj) {  
 mGsmSmsRegistrant = **new** Registrant (h, what, obj);  
}

8. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/base/core/java/android/os/Registrant.java

*/\*\**  
 *\* This makes a copy of @param ar*  
 *\*/*  
@UnsupportedAppUsage  
**public void**  
notifyRegistrant(AsyncResult ar)  
{  
 internalNotifyRegistrant (ar.result, ar.exception);  
}  
  
*/\*package\*/* **void**  
internalNotifyRegistrant (Object result, Throwable exception)  
{  
 Handler h = getHandler();  
  
 **if** (h == **null**) {  
 clear();  
 } **else** {  
 Message msg = Message.obtain();  
  
 msg.what = what;  
 msg.obj = **new** AsyncResult(userObj, result, exception);  
 h.sendMessage(msg);  
 }  
}

此处的h和what在第６步设置．

9. 处理 EVENT\_NEW\_SMS消息　/v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/InboundSmsHandler.java

**private class** DeliveringState **extends** State {  
 ．．．  
 @Override  
 **public boolean** processMessage(Message msg) {  
 **if** (DBG) log(**"DeliveringState.processMessage: processing "** + getWhatToString(msg.what));  
 **switch** (msg.what) {  
 **case** EVENT\_NEW\_SMS:  
 *// handle new SMS from RIL*  
handleNewSms((AsyncResult) msg.obj);  
 sendMessage(EVENT\_RETURN\_TO\_IDLE);  
 **return** HANDLED;  
 ．．．  
 }  
 }  
}

**private void** handleNewSms(AsyncResult ar) {  
 **if** (ar.exception != **null**) {  
 loge(**"Exception processing incoming SMS: "** + ar.exception);  
 **return**;  
 }  
  
 **int** result;  
 **try** {  
 SmsMessage sms = (SmsMessage) ar.result;  
 result = dispatchMessage(sms.mWrappedSmsMessage, SOURCE\_NOT\_INJECTED);  
 } **catch** (RuntimeException ex) {  
 loge(**"Exception dispatching message"**, ex);  
 result = RESULT\_SMS\_DISPATCH\_FAILURE;  
 }  
  
 ．．．  
}

**private int** dispatchMessage(SmsMessageBase smsb, @SmsSource **int** smsSource) {  
 *// If sms is null, there was a parsing error.*  
**if** (smsb == **null**) {  
 loge(**"dispatchSmsMessage: message is null"**);  
 **return** RESULT\_SMS\_NULL\_MESSAGE;  
 }  
  
 **if** (mSmsReceiveDisabled) {  
 *// Device doesn't support receiving SMS,*  
log(**"Received short message on device which doesn't support "**  
+ **"receiving SMS. Ignored."**);  
 **return** Intents.RESULT\_SMS\_HANDLED;  
 }  
  
 *// onlyCore indicates if the device is in cryptkeeper*  
**boolean** onlyCore = **false**;  
 **try** {  
 onlyCore = IPackageManager.Stub.asInterface(ServiceManager.getService(**"package"**))  
 .isOnlyCoreApps();  
 } **catch** (RemoteException e) {  
 }  
 **if** (onlyCore) {  
 *// Device is unable to receive SMS in encrypted state*  
log(**"Received a short message in encrypted state. Rejecting."**);  
 **return** Intents.RESULT\_SMS\_RECEIVED\_WHILE\_ENCRYPTED;  
 }  
  
 **int** result = dispatchMessageRadioSpecific(smsb, smsSource);  
  
 *// In case of error, add to metrics. This is not required in case of success, as the*  
 *// data will be tracked when the message is processed (processMessagePart).*  
**if** (result != Intents.RESULT\_SMS\_HANDLED && result != Activity.RESULT\_OK) {  
 mMetrics.writeIncomingSmsError(mPhone.getPhoneId(), is3gpp2(), smsSource, result);  
 mPhone.getSmsStats().onIncomingSmsError(is3gpp2(), smsSource, result);  
 }  
 **return** result;  
}

10. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/gsm/GsmInboundSmsHandler.java

*/\*\**  
 *\* Handle type zero, SMS-PP data download, and 3GPP/CPHS MWI type SMS. Normal SMS messages*  
 *\* are handled by {****@link*** *#dispatchNormalMessage} in parent class.*  
 *\**  
 *\** ***@param*** *smsb the SmsMessageBase object from the RIL*  
 *\** ***@param*** *smsSource the source of the SMS message*  
 *\** ***@return*** *a result code from {****@link*** *android.provider.Telephony.Sms.Intents},*  
 *\* or {****@link*** *Activity#RESULT\_OK} for delayed acknowledgment to SMSC*  
 *\*/*  
@Override  
**protected int** dispatchMessageRadioSpecific(SmsMessageBase smsb, @SmsSource **int** smsSource) {  
 SmsMessage sms = (SmsMessage) smsb;  
  
 **if** (sms.isTypeZero()) {  
 *// Some carriers will send visual voicemail SMS as type zero.*  
**int** destPort = -1;  
 SmsHeader smsHeader = sms.getUserDataHeader();  
 **if** (smsHeader != **null** && smsHeader.portAddrs != **null**) {  
 *// The message was sent to a port.*  
destPort = smsHeader.portAddrs.destPort;  
 }  
 VisualVoicemailSmsFilter  
 .filter(mContext, **new byte**[][]{sms.getPdu()}, SmsConstants.FORMAT\_3GPP,  
 destPort, mPhone.getSubId());  
 *// As per 3GPP TS 23.040 9.2.3.9, Type Zero messages should not be*  
 *// Displayed/Stored/Notified. They should only be acknowledged.*  
log(**"Received short message type 0, Don't display or store it. Send Ack"**);  
 addSmsTypeZeroToMetrics(smsSource);  
 **return** Intents.RESULT\_SMS\_HANDLED;  
 }  
  
 *// Send SMS-PP data download messages to UICC. See 3GPP TS 31.111 section 7.1.1.*  
**if** (sms.isUsimDataDownload()) {  
 UsimServiceTable ust = mPhone.getUsimServiceTable();  
 **return** mDataDownloadHandler.handleUsimDataDownload(ust, sms, smsSource);  
 }  
  
 **boolean** handled = **false**;  
 **if** (sms.isMWISetMessage()) {  
 updateMessageWaitingIndicator(sms.getNumOfVoicemails());  
 handled = sms.isMwiDontStore();  
 **if** (DBG) log(**"Received voice mail indicator set SMS shouldStore="** + !handled);  
 } **else if** (sms.isMWIClearMessage()) {  
 updateMessageWaitingIndicator(0);  
 handled = sms.isMwiDontStore();  
 **if** (DBG) log(**"Received voice mail indicator clear SMS shouldStore="** + !handled);  
 }  
 **if** (handled) {  
 addVoicemailSmsToMetrics(smsSource);  
 **return** Intents.RESULT\_SMS\_HANDLED;  
 }  
  
 **if** (!mStorageMonitor.isStorageAvailable() &&  
 sms.getMessageClass() != SmsConstants.MessageClass.CLASS\_0) {  
 *// It's a storable message and there's no storage available. Bail.*  
 *// (See TS 23.038 for a description of class 0 messages.)*  
**return** Intents.RESULT\_SMS\_OUT\_OF\_MEMORY;  
 }  
  
 **return** dispatchNormalMessage(smsb, smsSource);  
}

11. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/opt/telephony/src/java/com/android/internal/telephony/InboundSmsHandler.java

*/\*\**  
 *\* Dispatch a normal incoming SMS. This is called from {****@link*** *#dispatchMessageRadioSpecific}*  
 *\* if no format-specific handling was required. Saves the PDU to the SMS provider raw table,*  
 *\* creates an {****@link*** *InboundSmsTracker}, then sends it to the state machine as an*  
 *\* {****@link*** *#EVENT\_BROADCAST\_SMS}. Returns {****@link*** *Intents#RESULT\_SMS\_HANDLED} or an error value.*  
 *\**  
 *\** ***@param*** *sms the message to dispatch*  
 *\** ***@param*** *smsSource the source of the SMS message*  
 *\** ***@return*** *{****@link*** *Intents#RESULT\_SMS\_HANDLED} if the message was accepted, or an error status*  
 *\*/*  
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.R, trackingBug = 170729553)  
**protected int** dispatchNormalMessage(SmsMessageBase sms, @SmsSource **int** smsSource) {  
 SmsHeader smsHeader = sms.getUserDataHeader();  
 InboundSmsTracker tracker;  
  
 **if** ((smsHeader == **null**) || (smsHeader.concatRef == **null**)) {  
 *// Message is not concatenated.*  
**int** destPort = -1;  
 **if** (smsHeader != **null** && smsHeader.portAddrs != **null**) {  
 *// The message was sent to a port.*  
destPort = smsHeader.portAddrs.destPort;  
 **if** (DBG) log(**"destination port: "** + destPort);  
 }  
 tracker = TelephonyComponentFactory.getInstance()  
 .inject(InboundSmsTracker.**class**.getName())  
 .makeInboundSmsTracker(mContext, sms.getPdu(),  
 sms.getTimestampMillis(), destPort, is3gpp2(), **false**,  
 sms.getOriginatingAddress(), sms.getDisplayOriginatingAddress(),  
 sms.getMessageBody(), sms.getMessageClass() == MessageClass.CLASS\_0,  
 mPhone.getSubId(), smsSource);  
 } **else** {  
 *// Create a tracker for this message segment.*  
SmsHeader.ConcatRef concatRef = smsHeader.concatRef;  
 SmsHeader.PortAddrs portAddrs = smsHeader.portAddrs;  
 **int** destPort = (portAddrs != **null** ? portAddrs.destPort : -1);  
 tracker = TelephonyComponentFactory.getInstance()  
 .inject(InboundSmsTracker.**class**.getName())  
 .makeInboundSmsTracker(mContext, sms.getPdu(),  
 sms.getTimestampMillis(), destPort, is3gpp2(),  
 sms.getOriginatingAddress(), sms.getDisplayOriginatingAddress(),  
 concatRef.refNumber, concatRef.seqNumber, concatRef.msgCount, **false**,  
 sms.getMessageBody(), sms.getMessageClass() == MessageClass.CLASS\_0,  
 mPhone.getSubId(), smsSource);  
 }  
  
 **if** (VDBG) log(**"created tracker: "** + tracker);  
  
 *// de-duping is done only for text messages*  
 *// destPort = -1 indicates text messages, otherwise it's a data sms*  
**return** addTrackerToRawTableAndSendMessage(tracker,  
 tracker.getDestPort() == -1 */\* de-dup if text message \*/*);  
}

**protected int** addTrackerToRawTableAndSendMessage(InboundSmsTracker tracker, **boolean** deDup) {  
 **int** result = addTrackerToRawTable(tracker, deDup); //将消息添加到数据库中  
 **switch**(result) {  
 **case** Intents.RESULT\_SMS\_HANDLED:  
 sendMessage(EVENT\_BROADCAST\_SMS, tracker);  
 **return** Intents.RESULT\_SMS\_HANDLED;  
  
 **case** Intents.RESULT\_SMS\_DUPLICATED:  
 **return** Intents.RESULT\_SMS\_HANDLED;  
  
 **default**:  
 **return** result;  
 }  
}

**private class** DeliveringState **extends** State {  
 ．．．  
 @Override  
 **public boolean** processMessage(Message msg) {  
 **if** (DBG) log(**"DeliveringState.processMessage: processing "** + getWhatToString(msg.what));  
 **switch** (msg.what) {  
 ．．．  
 **case** EVENT\_BROADCAST\_SMS:  
 *// if any broadcasts were sent, transition to waiting state*  
InboundSmsTracker inboundSmsTracker = (InboundSmsTracker) msg.obj;  
 **if** (processMessagePart(inboundSmsTracker)) {  
 sendMessage(obtainMessage(EVENT\_UPDATE\_TRACKER, msg.obj));  
 transitionTo(mWaitingState);  
 } **else** {  
 *// if event is sent from SmsBroadcastUndelivered.broadcastSms(), and*  
 *// processMessagePart() returns false, the state machine will be stuck in*  
 *// DeliveringState until next message is received. Send message to*  
 *// transition to idle to avoid that so that wakelock can be released*  
log(**"DeliveringState.processMessage: EVENT\_BROADCAST\_SMS: No broadcast "**  
+ **"sent. Return to IdleState"**);  
 sendMessage(EVENT\_RETURN\_TO\_IDLE);  
 }  
 **return** HANDLED;  
 ．．．  
 }  
 }  
}

*/\*\**  
 *\* Process the inbound SMS segment. If the message is complete, send it as an ordered*  
 *\* broadcast to interested receivers and return true. If the message is a segment of an*  
 *\* incomplete multi-part SMS, return false.*  
 *\** ***@param*** *tracker the tracker containing the message segment to process*  
 *\** ***@return*** *true if an ordered broadcast was sent; false if waiting for more message segments*  
 *\*/*  
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.R, trackingBug = 170729553)  
**private boolean** processMessagePart(InboundSmsTracker tracker) {  
 ．．．  
 **if** (messageCount == 1) {  
 *// single-part message单条短信*  
pdus = **new byte**[][]{tracker.getPdu()};  
 timestamps = **new long**[]{tracker.getTimestamp()};  
 block = BlockChecker.isBlocked(mContext, tracker.getDisplayAddress(), **null**);  
 } **else** {  
 *// multi-part message长短信*  
．．．  
 }  
// 彩信的情况  
 **final boolean** isWapPush = (destPort == SmsHeader.PORT\_WAP\_PUSH);  
 ．．．  
// 创建广播接收者  
 SmsBroadcastReceiver resultReceiver = tracker.getSmsBroadcastReceiver(**this**);  
  
．．．  
  
 **if** (!filterInvoked) {  
 *// Block now if the filter wasn't invoked. Otherwise, it will be the responsibility of*  
 *// the filter to delete the SMS once processing completes.*  
**if** (block) {  
 deleteFromRawTable(tracker.getDeleteWhere(), tracker.getDeleteWhereArgs(),  
 DELETE\_PERMANENTLY);  
 log(**"processMessagePart: returning false as the phone number is blocked"**,  
 tracker.getMessageId());  
 **return false**;  
 }  
  
 dispatchSmsDeliveryIntent(pdus, format, destPort, resultReceiver,  
 tracker.isClass0(), tracker.getSubId(), tracker.getMessageId());  
 }  
  
 **return true**;  
}

*/\*\**  
 *\* Creates and dispatches the intent to the default SMS app, appropriate port or via the {****@link***  
*\* AppSmsManager}.*  
 *\**  
 *\** ***@param*** *pdus message pdus*  
 *\** ***@param*** *format the message format, typically "3gpp" or "3gpp2"*  
 *\** ***@param*** *destPort the destination port*  
 *\** ***@param*** *resultReceiver the receiver handling the delivery result*  
 *\*/*  
**private void** dispatchSmsDeliveryIntent(**byte**[][] pdus, String format, **int** destPort,  
 SmsBroadcastReceiver resultReceiver, **boolean** isClass0, **int** subId, **long** messageId) {  
 Intent intent = **new** Intent();  
 intent.putExtra(**"pdus"**, pdus);  
 intent.putExtra(**"format"**, format);  
 **if** (messageId != 0L) {  
 intent.putExtra(**"messageId"**, messageId);  
 }  
  
 **if** (destPort == -1) {  
 intent.setAction(Intents.SMS\_DELIVER\_ACTION);  
 *// Direct the intent to only the default SMS app. If we can't find a default SMS app*  
 *// then sent it to all broadcast receivers.*  
 *// We are deliberately delivering to the primary user's default SMS App.*  
ComponentName componentName = SmsApplication.getDefaultSmsApplication(mContext, **true**);  
 **if** (componentName != **null**) {  
 *// Deliver SMS message only to this receiver.*  
intent.setComponent(componentName);  
 logWithLocalLog(**"Delivering SMS to: "** + componentName.getPackageName()  
 + **" "** + componentName.getClassName(), messageId);  
 } **else** {  
 intent.setComponent(**null**);  
 }  
  
 *// Handle app specific sms messages.*  
AppSmsManager appManager = mPhone.getAppSmsManager();  
 **if** (appManager.handleSmsReceivedIntent(intent)) {  
 *// The AppSmsManager handled this intent, we're done.*  
dropSms(resultReceiver);  
 **return**;  
 }  
 } **else** {  
 intent.setAction(Intents.DATA\_SMS\_RECEIVED\_ACTION);  
 Uri uri = Uri.parse(**"sms://localhost:"** + destPort);  
 intent.setData(uri);  
 intent.setComponent(**null**);  
 }  
  
 Bundle options = handleSmsWhitelisting(intent.getComponent(), isClass0);  
 dispatchIntent(intent, android.Manifest.permission.RECEIVE\_SMS,  
 AppOpsManager.OPSTR\_RECEIVE\_SMS, options, resultReceiver, UserHandle.SYSTEM, subId);  
}

12. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/receiver/SmsDeliverReceiver.java

**public final class** SmsDeliverReceiver **extends** BroadcastReceiver {  
 @Override  
 **public void** onReceive(**final** Context context, **final** Intent intent) {  
 SmsReceiver.deliverSmsIntent(context, intent);  
 }  
}