发送短信流程，关键LOG标红，关键节点代码标黄

1. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/ui/conversation/ComposeMessageView.java

mSendButton.setOnClickListener(**new** OnClickListener() {  
 @Override  
 **public void** onClick(**final** View clickView) {  
 sendMessageInternal(**true** */\* checkMessageSize \*/*);  
 }  
});

**private void** sendMessageInternal(**final boolean** checkMessageSize) {  
 LogUtil.i(LogUtil.BUGLE\_TAG, **"UI initiated message sending in conversation "** +  
 mBinding.getData().getConversationId());  
 ...  
 *// Check the host for pre-conditions about any action.*  
**if** (mHost.isReadyForAction()) {  
 ...  
 *// Asynchronously check the draft against various requirements before sending.*  
mBinding.getData().checkDraftForAction(checkMessageSize,  
 mHost.getConversationSelfSubId(), **new** CheckDraftTaskCallback() {  
 @Override  
 **public void** onDraftChecked(DraftMessageData data, **int** result) {  
 mBinding.ensureBound(data);  
 **switch** (result) {  
 **case** CheckDraftForSendTask.RESULT\_PASSED:  
 *// Continue sending after check succeeded.*  
**final** MessageData message = mBinding.getData()  
 .prepareMessageForSending(mBinding);  
 **if** (message != **null** && message.hasContent()) {  
 playSentSound();  
 mHost.sendMessage(message);  
 hideSubjectEditor();  
 **if** (AccessibilityUtil.isTouchExplorationEnabled(getContext())) {  
 AccessibilityUtil.announceForAccessibilityCompat(  
 ComposeMessageView.**this**, **null**,  
 R.string.sending\_message);  
 }  
 }  
 **break**;  
  
 …  
 }  
 }  
 }, mBinding);  
 } **else** {  
 ...  
 }  
}

2. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/ui/conversation/ConversationFragment.java

**public void** sendMessage(**final** MessageData message) {  
 **if** (isReadyForAction()) {  
 **if** (ensureKnownRecipients()) {  
 *// Merge the caption text from attachments into the text body of the messages*  
message.consolidateText();  
  
 mBinding.getData().sendMessage(mBinding, message);  
 mComposeMessageView.resetMediaPickerState();  
 } **else** {  
 LogUtil.w(LogUtil.BUGLE\_TAG, **"Message can't be sent: conv participants not loaded"**);  
 }  
 } **else** {  
 ...  
 }  
}

3. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/datamodel/data/ConversationData.java

**public void** sendMessage(**final** BindingBase<ConversationData> binding,  
 **final** MessageData message) {  
 ...  
  
 **if** (!OsUtil.isAtLeastL\_MR1() || message.getSelfId() == **null**) {  
 InsertNewMessageAction.insertNewMessage(message);  
 } **else** {  
 **final int** systemDefaultSubId = PhoneUtils.getDefault().getDefaultSmsSubscriptionId();  
 **if** (systemDefaultSubId != ParticipantData.DEFAULT\_SELF\_SUB\_ID &&  
 mSelfParticipantsData.isDefaultSelf(message.getSelfId())) {  
 *// Lock the sub selection to the system default SIM as soon as the user clicks on*  
 *// the send button to avoid races between this and when InsertNewMessageAction is*  
 *// actually executed on the data model thread, during which the user can potentially*  
 *// change the system default SIM in Settings.*  
InsertNewMessageAction.insertNewMessage(message, systemDefaultSubId);  
 } **else** {  
 InsertNewMessageAction.insertNewMessage(message);  
 }  
 }  
 *// Update contacts so Frequents will reflect messaging activity.*  
...  
}

4. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/datamodel/action/InsertNewMessageAction.java

*/\*\**  
 *\* Add message to database in pending state and queue actual sending*  
 *\*/*  
@Override  
**protected** Object executeAction() {  
 MessageData message = actionParameters.getParcelable(KEY\_MESSAGE);  
 ...  
 **final int** subId = self.getSubId();  
 LogUtil.i(TAG, **"InsertNewMessageAction: inserting new message for subId "** + subId);  
 actionParameters.putInt(KEY\_SUB\_ID, subId);  
  
 ...  
 MessagingContentProvider.notifyConversationListChanged();  
 ProcessPendingMessagesAction.scheduleProcessPendingMessagesAction(**false**, **this**);  
  
 **return** message;  
}

5. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/datamodel/action/ProcessPendingMessagesAction.java

**public static void** scheduleProcessPendingMessagesAction(**final boolean** failed,  
 **final** Action processingAction) {  
 **final int** subId = processingAction.actionParameters  
 .getInt(KEY\_SUB\_ID, ParticipantData.DEFAULT\_SELF\_SUB\_ID);  
 LogUtil.i(TAG, **"ProcessPendingMessagesAction: Scheduling pending messages"**  
+ (failed ? **"(message failed)"** : **""**) + **" for subId "** + subId);  
 *// Can safely clear any pending alarms or connectivity events as either an action*  
 *// is currently running or we will run now or register if pending actions possible.*  
unregister(subId);  
  
 **final boolean** isDefaultSmsApp = PhoneUtils.getDefault().isDefaultSmsApp();  
 **boolean** scheduleAlarm = **false**;  
 *// If message succeeded and if Bugle is default SMS app just carry on with next message*  
**if** (!failed && isDefaultSmsApp) {  
 *// Clear retry attempt count as something just succeeded*  
setRetry(0, subId);  
  
 *// Lookup and queue next message for each sending/downloading for immediate processing*  
 *// by background worker. If there are no pending messages, this will do nothing and*  
 *// return true.*  
**final** ProcessPendingMessagesAction action = **new** ProcessPendingMessagesAction();  
 **if** (action.queueActions(processingAction)) {  
 **if** (LogUtil.isLoggable(TAG, LogUtil.VERBOSE)) {  
 **if** (processingAction.hasBackgroundActions()) {  
 LogUtil.v(TAG, **"ProcessPendingMessagesAction: Action queued"**);  
 } **else** {  
 LogUtil.v(TAG, **"ProcessPendingMessagesAction: No actions to queue"**);  
 }  
 }  
 *// Have queued next action if needed, nothing more to do*  
**return**;  
 }  
 *// In case of error queuing schedule a retry*  
scheduleAlarm = **true**;  
 LogUtil.w(TAG, **"ProcessPendingMessagesAction: Action failed to queue; retrying"**);  
 }  
 ...  
}

*/\*\**  
 *\* Queue any pending actions*  
 *\**  
 *\** ***@param*** *actionState*  
 *\** ***@return*** *true if action queued (or no actions to queue) else false*  
 *\*/*  
**private boolean** queueActions(**final** Action processingAction) {  
 **final** DatabaseWrapper db = DataModel.get().getDatabase();  
 **final long** now = System.currentTimeMillis();  
 **boolean** succeeded = **true**;  
 **final int** subId = processingAction.actionParameters  
 .getInt(KEY\_SUB\_ID, ParticipantData.DEFAULT\_SELF\_SUB\_ID);  
  
 LogUtil.i(TAG, **"ProcessPendingMessagesAction: Start queueing for subId "** + subId);  
  
 **final** String selfId = ParticipantData.getParticipantId(db, subId);  
 **if** (selfId == **null**) {  
 *// This could be happened before refreshing participant.*  
LogUtil.w(TAG, **"ProcessPendingMessagesAction: selfId is null"**);  
 **return false**;  
 }  
  
 *// Will queue no more than one message to send plus one message to download*  
 *// This keeps outgoing messages "in order" but allow downloads to happen even if sending*  
 *// gets blocked until messages time out. Manual resend bumps messages to head of queue.*  
**final** String toSendMessageId = findNextMessageToSend(db, now, selfId);  
 **final** String toDownloadMessageId = findNextMessageToDownload(db, now, selfId);  
 **if** (toSendMessageId != **null**) {  
 LogUtil.i(TAG, **"ProcessPendingMessagesAction: Queueing message "** + toSendMessageId  
 + **" for sending"**);  
 *// This could queue nothing*  
**if** (!SendMessageAction.queueForSendInBackground(toSendMessageId, processingAction)) {  
 LogUtil.w(TAG, **"ProcessPendingMessagesAction: Failed to queue message "**  
+ toSendMessageId + **" for sending"**);  
 succeeded = **false**;  
 }  
 }  
 ...  
 **return** succeeded;  
}

6. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/datamodel/action/SendMessageAction.java

*/\*\**  
 *\* Queue sending of existing message (can only be called during execute of action)*  
 *\*/*  
**static boolean** queueForSendInBackground(**final** String messageId,  
 **final** Action processingAction) {  
 **final** SendMessageAction action = **new** SendMessageAction();  
 **return** action.queueAction(messageId, processingAction);  
}

*/\*\**  
 *\* Read message from database and queue actual sending*  
 *\*/*  
**private boolean** queueAction(**final** String messageId, **final** Action processingAction) {  
 actionParameters.putString(KEY\_MESSAGE\_ID, messageId);  
  
 **final** DatabaseWrapper db = DataModel.get().getDatabase();  
  
 **final** MessageData message = BugleDatabaseOperations.readMessage(db, messageId);  
 *// Check message can be resent*  
**if** (message != **null** && message.canSendMessage()) {  
 ...  
  
 **if** (isSms) {  
 **final** String smsc = BugleDatabaseOperations.getSmsServiceCenterForConversation(  
 db, conversationId);  
 actionParameters.putString(KEY\_SMS\_SERVICE\_CENTER, smsc);  
  
 **if** (recipients.size() == 1) {  
 **final** String recipient = recipients.get(0);  
  
 actionParameters.putString(KEY\_RECIPIENT, recipient);  
 *// Queue actual sending for SMS*  
processingAction.requestBackgroundWork(**this**);  
  
 **if** (LogUtil.isLoggable(TAG, LogUtil.DEBUG)) {  
 LogUtil.d(TAG, **"SendMessageAction: Queued SMS message "** + messageId  
 + **" for sending"**);  
 }  
 **return true**;  
 } **else** {  
 LogUtil.wtf(TAG, **"Trying to resend a broadcast SMS - not allowed"**);  
 }  
 } **else** {  
 *// Queue actual sending for MMS*  
processingAction.requestBackgroundWork(**this**);  
  
 **if** (LogUtil.isLoggable(TAG, LogUtil.DEBUG)) {  
 LogUtil.d(TAG, **"SendMessageAction: Queued MMS message "** + messageId  
 + **" for sending"**);  
 }  
 **return true**;  
 }  
 }  
  
 **return false**;  
}

*/\*\**  
 *\* Send message on background worker thread*  
 *\*/*  
@Override  
**protected** Bundle doBackgroundWork() {  
 **final** MessageData message = actionParameters.getParcelable(KEY\_MESSAGE);  
 **final** String messageId = actionParameters.getString(KEY\_MESSAGE\_ID);  
 Uri messageUri = actionParameters.getParcelable(KEY\_MESSAGE\_URI);  
 Uri updatedMessageUri = **null**;  
 **final boolean** isSms = message.getProtocol() == MessageData.PROTOCOL\_SMS;  
 **final int** subId = actionParameters.getInt(KEY\_SUB\_ID, ParticipantData.DEFAULT\_SELF\_SUB\_ID);  
 **final** String subPhoneNumber = actionParameters.getString(KEY\_SUB\_PHONE\_NUMBER);  
  
 LogUtil.i(TAG, **"SendMessageAction: Sending "** + (isSms ? **"SMS"** : **"MMS"**) + **" message "**  
+ messageId + **" in conversation "** + message.getConversationId());  
  
 **int** status;  
 **int** rawStatus = MessageData.RAW\_TELEPHONY\_STATUS\_UNDEFINED;  
 **int** resultCode = MessageData.UNKNOWN\_RESULT\_CODE;  
 **if** (isSms) {  
 Assert.notNull(messageUri);  
 **final** String recipient = actionParameters.getString(KEY\_RECIPIENT);  
 **final** String messageText = message.getMessageText();  
 **final** String smsServiceCenter = actionParameters.getString(KEY\_SMS\_SERVICE\_CENTER);  
 **final boolean** deliveryReportRequired = MmsUtils.isDeliveryReportRequired(subId);  
  
 status = MmsUtils.sendSmsMessage(recipient, messageText, messageUri, subId,  
 smsServiceCenter, deliveryReportRequired);  
 } **else** {  
 ...  
 }  
  
 *// When we fast-fail before calling the MMS lib APIs (e.g. airplane mode,*  
 *// sending message is deleted).*  
ProcessSentMessageAction.processMessageSentFastFailed(messageId, messageUri,  
 updatedMessageUri, subId, isSms, status, rawStatus, resultCode);  
 **return null**;  
}

7. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/sms/MmsUtils.java

**public static int** sendSmsMessage(**final** String recipient, **final** String messageText,  
 **final** Uri requestUri, **final int** subId,  
 **final** String smsServiceCenter, **final boolean** requireDeliveryReport) {  
 **if** (!isSmsDataAvailable(subId)) {  
 LogUtil.w(TAG, **"MmsUtils: can't send SMS without radio"**);  
 **return** MMS\_REQUEST\_MANUAL\_RETRY;  
 }  
 **final** Context context = Factory.get().getApplicationContext();  
 **int** status = MMS\_REQUEST\_MANUAL\_RETRY;  
 **try** {  
 *// Send a single message*  
**final** SendResult result = SmsSender.sendMessage(  
 context,  
 subId,  
 recipient,  
 messageText,  
 smsServiceCenter,  
 requireDeliveryReport,  
 requestUri);  
 ...  
 } **catch** (**final** Exception e) {  
 LogUtil.e(TAG, **"MmsUtils: failed to send SMS "** + e, e);  
 }  
 **return** status;  
}

8. /v790/LA.QSSI.12.0.r1/LINUX/android/packages/apps/Messaging/src/com/android/messaging/sms/SmsSender.java

*// This should be called from a RequestWriter queue thread*  
**public static** SendResult sendMessage(**final** Context context, **final int** subId, String dest,  
 String message, **final** String serviceCenter, **final boolean** requireDeliveryReport,  
 **final** Uri messageUri) **throws** SmsException {  
 **if** (LogUtil.isLoggable(TAG, LogUtil.VERBOSE)) {  
 LogUtil.v(TAG, **"SmsSender: sending message. "** +  
 **"dest="** + dest + **" message="** + message +  
 **" serviceCenter="** + serviceCenter +  
 **" requireDeliveryReport="** + requireDeliveryReport +  
 **" requestId="** + messageUri);  
 }  
 **if** (TextUtils.isEmpty(message)) {  
 **throw new** SmsException(**"SmsSender: empty text message"**);  
 }  
 *// Get the real dest and message for email or alias if dest is email or alias*  
 *// Or sanitize the dest if dest is a number*  
**if** (!TextUtils.isEmpty(MmsConfig.get(subId).getEmailGateway()) &&  
 (MmsSmsUtils.isEmailAddress(dest) || MmsSmsUtils.isAlias(dest, subId))) {  
 *// The original destination (email address) goes with the message*  
message = dest + **" "** + message;  
 *// the new address is the email gateway #*  
dest = MmsConfig.get(subId).getEmailGateway();  
 } **else** {  
 *// remove spaces and dashes from destination number*  
 *// (e.g. "801 555 1212" -> "8015551212")*  
 *// (e.g. "+8211-123-4567" -> "+82111234567")*  
dest = PhoneNumberUtils.stripSeparators(dest);  
 }  
 **if** (TextUtils.isEmpty(dest)) {  
 **throw new** SmsException(**"SmsSender: empty destination address"**);  
 }  
 *// Divide the input message by SMS length limit*  
**final** SmsManager smsManager = PhoneUtils.get(subId).getSmsManager();  
 **final** ArrayList<String> messages = smsManager.divideMessage(message);  
 **if** (messages == **null** || messages.size() < 1) {  
 **throw new** SmsException(**"SmsSender: fails to divide message"**);  
 }  
 *// Prepare the send result, which collects the send status for each part*  
**final** SendResult pendingResult = **new** SendResult(messages.size());  
 sPendingMessageMap.put(messageUri, pendingResult);  
 *// Actually send the sms*  
sendInternal(  
 context, subId, dest, messages, serviceCenter, requireDeliveryReport, messageUri);  
 *// Wait for pending intent to come back*  
...  
 **return** pendingResult;  
}

*// Actually sending the message using SmsManager*  
**private static void** sendInternal(**final** Context context, **final int** subId, String dest,  
 **final** ArrayList<String> messages, **final** String serviceCenter,  
 **final boolean** requireDeliveryReport, **final** Uri messageUri) **throws** SmsException {  
 Assert.notNull(context);  
 **final** SmsManager smsManager = PhoneUtils.get(subId).getSmsManager();  
 **final int** messageCount = messages.size();  
 **final** ArrayList<PendingIntent> deliveryIntents = **new** ArrayList<PendingIntent>(messageCount);  
 **final** ArrayList<PendingIntent> sentIntents = **new** ArrayList<PendingIntent>(messageCount);  
 **for** (**int** i = 0; i < messageCount; i++) {  
 *// Make pending intents different for each message part*  
**final int** partId = (messageCount <= 1 ? 0 : i + 1);  
 **if** (requireDeliveryReport && (i == (messageCount - 1))) {  
 *// TODO we only care about the delivery status of the last part*  
 *// Shall we have better tracking of delivery status of all parts?*  
deliveryIntents.add(PendingIntent.getBroadcast(  
 context,  
 partId,  
 getSendStatusIntent(context, SendStatusReceiver.MESSAGE\_DELIVERED\_ACTION,  
 messageUri, partId, subId),  
 0*/\*flag\*/*));  
 } **else** {  
 deliveryIntents.add(**null**);  
 }  
 sentIntents.add(PendingIntent.getBroadcast(  
 context,  
 partId,  
 getSendStatusIntent(context, SendStatusReceiver.MESSAGE\_SENT\_ACTION,  
 messageUri, partId, subId),  
 0*/\*flag\*/*));  
 }  
 **try** {  
 **if** (MmsConfig.get(subId).getSendMultipartSmsAsSeparateMessages()) {  
 *// If multipart sms is not supported, send them as separate messages*  
**for** (**int** i = 0; i < messageCount; i++) {  
 smsManager.sendTextMessage(dest,  
 serviceCenter,  
 messages.get(i),  
 sentIntents.get(i),  
 deliveryIntents.get(i));  
 }  
 } **else** {//长短信  
 smsManager.sendMultipartTextMessage(  
 dest, serviceCenter, messages, sentIntents, deliveryIntents);  
 }  
 } **catch** (**final** Exception e) {  
 **throw new** SmsException(**"SmsSender: caught exception in sending "** + e);  
 }  
}

9. /v790/LA.QSSI.12.0.r1/LINUX/android/frameworks/base/telephony/java/android/telephony/SmsManager.java

**public void** sendTextMessage(  
 String destinationAddress, String scAddress, String text,  
 PendingIntent sentIntent, PendingIntent deliveryIntent) {  
 android.util.SeempLog.record\_str(75, destinationAddress);  
 sendTextMessageInternal(destinationAddress, scAddress, text, sentIntent, deliveryIntent,  
 **true** */\* persistMessage\*/*, getOpPackageName(), getAttributionTag(),  
 0L */\* messageId \*/*);  
}

**private void** sendTextMessageInternal(String destinationAddress, String scAddress,  
 String text, PendingIntent sentIntent, PendingIntent deliveryIntent,  
 **boolean** persistMessage, String packageName, String attributionTag, **long** messageId) {  
 ...  
**if** (persistMessage) {  
 resolveSubscriptionForOperation(**new** SubscriptionResolverResult() {  
 @Override  
 **public void** onSuccess(**int** subId) {  
 ISms iSms = getISmsServiceOrThrow();  
 **try** {  
 iSms.sendTextForSubscriber(subId, packageName, attributionTag,  
 destinationAddress, scAddress, text, sentIntent, deliveryIntent,  
 persistMessage, messageId);  
 } **catch** (RemoteException e) {  
 Log.e(TAG, **"sendTextMessageInternal: Couldn't send SMS, exception - "**  
+ e.getMessage() + **" "** + formatCrossStackMessageId(messageId));  
 notifySmsError(sentIntent, RESULT\_REMOTE\_EXCEPTION);  
 }  
 }  
  
 @Override  
 **public void** onFailure() {  
 notifySmsError(sentIntent, RESULT\_NO\_DEFAULT\_SMS\_APP);  
 }  
 });  
 } **else** {  
 ...  
 }  
}

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

@Override  
**public void** sendTextForSubscriber(**int** subId, String callingPackage,  
 String callingAttributionTag, String destAddr, String scAddr, String text,  
 PendingIntent sentIntent, PendingIntent deliveryIntent,  
 **boolean** persistMessageForNonDefaultSmsApp, **long** messageId) {  
 ...  
 **if** (isBluetoothSubscription(info)) {  
 sendBluetoothText(info, destAddr, text, sentIntent, deliveryIntent);  
 } **else** {  
 sendIccText(subId, callingPackage, destAddr, scAddr, text, sentIntent, deliveryIntent,  
 persistMessageForNonDefaultSmsApp, messageId);  
 }  
}

**private void** sendIccText(**int** subId, String callingPackage, String destAddr,  
 String scAddr, String text, PendingIntent sentIntent, PendingIntent deliveryIntent,  
 **boolean** persistMessageForNonDefaultSmsApp, **long** messageId) {  
 Rlog.d(LOG\_TAG, **"sendTextForSubscriber iccSmsIntMgr"**  
+ **" Subscription: "** + subId + **" id: "** + messageId);  
 IccSmsInterfaceManager iccSmsIntMgr = getIccSmsInterfaceManager(subId);  
 **if** (iccSmsIntMgr != **null**) {  
 iccSmsIntMgr.sendText(callingPackage, destAddr, scAddr, text, sentIntent,  
 deliveryIntent, persistMessageForNonDefaultSmsApp, messageId);  
 } **else** {  
 Rlog.e(LOG\_TAG, **"sendTextForSubscriber iccSmsIntMgr is null for"**  
+ **" Subscription: "** + subId + **" id: "** + messageId);  
 sendErrorInPendingIntent(sentIntent, SmsManager.RESULT\_ERROR\_GENERIC\_FAILURE);  
 }  
}

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

**public void** sendText(String callingPackage, String destAddr, String scAddr,  
 String text, PendingIntent sentIntent, PendingIntent deliveryIntent,  
 **boolean** persistMessageForNonDefaultSmsApp, **long** messageId) {  
 sendTextInternal(callingPackage, destAddr, scAddr, text, sentIntent, deliveryIntent,  
 persistMessageForNonDefaultSmsApp, SMS\_MESSAGE\_PRIORITY\_NOT\_SPECIFIED,  
 **false** */\* expectMore \*/*, SMS\_MESSAGE\_PERIOD\_NOT\_SPECIFIED, **false** */\* isForVvm \*/*,  
 messageId);  
}

**private void** sendTextInternal(String callingPackage, String destAddr, String scAddr,  
 String text, PendingIntent sentIntent, PendingIntent deliveryIntent,  
 **boolean** persistMessageForNonDefaultSmsApp, **int** priority, **boolean** expectMore,  
 **int** validityPeriod, **boolean** isForVvm, **long** messageId) {  
 **if** (Rlog.isLoggable(**"SMS"**, Log.VERBOSE)) {  
 log(**"sendText: destAddr="** + destAddr + **" scAddr="** + scAddr  
 + **" text='"** + text + **"' sentIntent="** + sentIntent + **" deliveryIntent="**  
+ deliveryIntent + **" priority="** + priority + **" expectMore="** + expectMore  
 + **" validityPeriod="** + validityPeriod + **" isForVVM="** + isForVvm  
 + **" "** + SmsController.formatCrossStackMessageId(messageId));  
 }  
 notifyIfOutgoingEmergencySms(destAddr);  
 destAddr = filterDestAddress(destAddr);  
 mDispatchersController.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent,  
 **null***/\*messageUri\*/*, callingPackage, persistMessageForNonDefaultSmsApp,  
 priority, expectMore, validityPeriod, isForVvm, messageId);  
}

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

**public void** sendText(String destAddr, String scAddr, String text, PendingIntent sentIntent,  
 PendingIntent deliveryIntent, Uri messageUri, String callingPkg, **boolean** persistMessage,  
 **int** priority, **boolean** expectMore, **int** validityPeriod, **boolean** isForVvm,  
 **long** messageId) {  
 **if** (mImsSmsDispatcher.isAvailable() || mImsSmsDispatcher.isEmergencySmsSupport(destAddr)) {  
 mImsSmsDispatcher.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent,  
 messageUri, callingPkg, persistMessage, priority, **false** */\*expectMore\*/*,  
 validityPeriod, isForVvm, messageId);  
 } **else** {  
 **if** (isCdmaMo()) {  
 mCdmaDispatcher.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent,  
 messageUri, callingPkg, persistMessage, priority, expectMore,  
 validityPeriod, isForVvm, messageId);  
 } **else** {  
 mGsmDispatcher.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent,  
 messageUri, callingPkg, persistMessage, priority, expectMore,  
 validityPeriod, isForVvm, messageId);  
 }  
 }  
}

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

**public void** sendText(String destAddr, String scAddr, String text,  
 PendingIntent sentIntent, PendingIntent deliveryIntent, Uri messageUri,  
 String callingPkg, **boolean** persistMessage, **int** priority,  
 **boolean** expectMore, **int** validityPeriod, **boolean** isForVvm,  
 **long** messageId) {  
 Rlog.d(TAG, **"sendText id: "** + messageId);  
 SmsMessageBase.SubmitPduBase pdu = getSubmitPdu(  
 scAddr, destAddr, text, (deliveryIntent != **null**), **null**, priority, validityPeriod);  
 **if** (pdu != **null**) {  
 HashMap map = getSmsTrackerMap(destAddr, scAddr, text, pdu);  
 SmsTracker tracker = getSmsTracker(callingPkg, map, sentIntent, deliveryIntent,  
 getFormat(), messageUri, expectMore, text, **true** */\*isText\*/*,  
 persistMessage, priority, validityPeriod, isForVvm, messageId);  
  
 **if** (!sendSmsByCarrierApp(**false** */\* isDataSms \*/*, tracker)) {  
 sendSubmitPdu(tracker);  
 }  
 } **else** {  
 Rlog.e(TAG, **"SmsDispatcher.sendText(): getSubmitPdu() returned null"** + **" id: "**  
+ messageId);  
 triggerSentIntentForFailure(sentIntent);  
 }  
}

*/\*\* Send a single SMS PDU. \*/*  
@UnsupportedAppUsage(maxTargetSdk = Build.VERSION\_CODES.R, trackingBug = 170729553)  
**private void** sendSubmitPdu(SmsTracker tracker) {  
 sendSubmitPdu(**new** SmsTracker[] {tracker});  
}  
  
*/\*\* Send a multi-part SMS PDU. Usually just calls {****@link*** *sendRawPdu}. \*/*  
**private void** sendSubmitPdu(SmsTracker[] trackers) {  
 **if** (shouldBlockSmsForEcbm()) {  
 Rlog.d(TAG, **"Block SMS in Emergency Callback mode"**);  
 handleSmsTrackersFailure(trackers, SmsManager.RESULT\_SMS\_BLOCKED\_DURING\_EMERGENCY,  
 NO\_ERROR\_CODE);  
 } **else** {  
 sendRawPdu(trackers);  
 }  
}

@VisibleForTesting  
**public void** sendRawPdu(SmsTracker[] trackers) {  
 ...  
 *// checkDestination() returns true if the destination is not a premium short code or the*  
 *// sending app is approved to send to short codes. Otherwise, a message is sent to our*  
 *// handler with the SmsTracker to request user confirmation before sending.*  
**if** (checkDestination(trackers)) {  
 *// check for excessive outgoing SMS usage by this app*  
**if** (!mSmsDispatchersController  
 .getUsageMonitor()  
 .check(appInfo.packageName, trackers.length)) {  
 sendMessage(obtainMessage(EVENT\_SEND\_LIMIT\_REACHED\_CONFIRMATION, trackers));  
 **return**;  
 }  
  
 **for** (SmsTracker tracker : trackers) {  
 **if** (mSmsDispatchersController.getUsageMonitor().isSmsAuthorizationEnabled()) {  
 **final** SmsAuthorizationCallback callback = **new** SmsAuthorizationCallback() {  
 @Override  
 **public void** onAuthorizationResult(**final boolean** accepted) {  
 **if** (accepted) {  
 sendSms(tracker);  
 } **else** {  
 tracker.onFailed(mContext, SmsManager.RESULT\_ERROR\_GENERIC\_FAILURE,  
 SmsUsageMonitor.ERROR\_CODE\_BLOCKED);  
 }  
 }  
 };  
 mSmsDispatchersController.getUsageMonitor().authorizeOutgoingSms(tracker.mAppInfo,  
 tracker.mDestAddress,tracker.mFullMessageText, callback, **this**);  
 } **else** {  
 sendSms(tracker);  
 }  
 }  
 }  
  
 **if** (mTelephonyManager.isEmergencyNumber(trackers[0].mDestAddress)) {  
 **new** AsyncEmergencyContactNotifier(mContext).execute();  
 }  
}

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

**protected void** sendSms(SmsTracker tracker) {  
 **int** ss = mPhone.getServiceState().getState();  
  
 Rlog.d(TAG, **"sendSms: "**  
+ **" isIms()="** + isIms()  
 + **" mRetryCount="** + tracker.mRetryCount  
 + **" mImsRetry="** + tracker.mImsRetry  
 + **" mMessageRef="** + tracker.mMessageRef  
 + **" mUsesImsServiceForIms="** + tracker.mUsesImsServiceForIms  
 + **" SS="** + ss  
 + **" "** + SmsController.formatCrossStackMessageId(tracker.mMessageId));  
 …

Message reply = obtainMessage(EVENT\_SEND\_SMS\_COMPLETE, tracker);

…  
 *// sms over gsm is used:*  
 *// if sms over IMS is not supported AND*  
 *// this is not a retry case after sms over IMS failed*  
 *// indicated by mImsRetry > 0 OR*  
 *// this tracker uses ImsSmsDispatcher to handle SMS over IMS. This dispatcher has received*  
 *// this message because the ImsSmsDispatcher has indicated that the message needs to*  
 *// fall back to sending over CS.*  
**if** (0 == tracker.mImsRetry && !isIms() || tracker.mUsesImsServiceForIms) {  
 **if** (tracker.mRetryCount == 0 && tracker.mExpectMore) {  
 mCi.sendSMSExpectMore(IccUtils.bytesToHexString(smsc),  
 IccUtils.bytesToHexString(pdu), reply);  
 } **else** {  
 mCi.sendSMS(IccUtils.bytesToHexString(smsc),  
 IccUtils.bytesToHexString(pdu), reply);  
 }  
 } **else** {  
 mCi.sendImsGsmSms(IccUtils.bytesToHexString(smsc),  
 IccUtils.bytesToHexString(pdu), tracker.mImsRetry,  
 tracker.mMessageRef, reply);  
 *// increment it here, so in case of SMS\_FAIL\_RETRY over IMS*  
 *// next retry will be sent using IMS request again.*  
tracker.mImsRetry++;  
 }  
}

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

@Override  
**public void** sendSMS(String smscPdu, String pdu, Message result) {  
 IRadio radioProxy = getRadioProxy(result);  
 **if** (radioProxy != **null**) {  
 RILRequest rr = obtainRequest(RIL\_REQUEST\_SEND\_SMS, result,  
 mRILDefaultWorkSource);  
  
 *// Do not log function args for privacy*  
**if** (RILJ\_LOGD) riljLog(rr.serialString() + **"> "** + requestToString(rr.mRequest));  
  
 GsmSmsMessage msg = constructGsmSendSmsRilRequest(smscPdu, pdu);  
 **if** (mRadioVersion.greaterOrEqual(RADIO\_HAL\_VERSION\_1\_6)) {  
 **try** {  
 android.hardware.radio.V1\_6.IRadio radioProxy16 =  
 (android.hardware.radio.V1\_6.IRadio) radioProxy;  
 radioProxy16.sendSms\_1\_6(rr.mSerial, msg);  
 mMetrics.writeRilSendSms(mPhoneId, rr.mSerial, SmsSession.Event.Tech.SMS\_GSM,  
 SmsSession.Event.Format.SMS\_FORMAT\_3GPP,  
 getOutgoingSmsMessageId(result));  
 } **catch** (RemoteException | RuntimeException e) {  
 handleRadioProxyExceptionForRR(rr, **"sendSMS"**, e);  
 }  
 } **else** {  
 **try** {  
 radioProxy.sendSms(rr.mSerial, msg);  
 mMetrics.writeRilSendSms(mPhoneId, rr.mSerial, SmsSession.Event.Tech.SMS\_GSM,  
 SmsSession.Event.Format.SMS\_FORMAT\_3GPP,  
 getOutgoingSmsMessageId(result));  
 } **catch** (RemoteException | RuntimeException e) {  
 handleRadioProxyExceptionForRR(rr, **"sendSMS"**, e);  
 }  
 }  
 }  
}

16. /v790/LA.QSSI.12.0.r1/LINUX/android/hardware/interfaces/radio/1.6/Iradio.hal

oneway sendSms\_1\_6(int32\_t serial, GsmSmsMessage message);

17. /v790/LA.QSSI.12.0.r1/LINUX/android/hardware/interfaces/radio/1.0/Iradio.hal

oneway sendSms(int32\_t serial, GsmSmsMessage message);

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

Return<void> RadioImpl::sendSms(int32\_t serial, const GsmSmsMessage& message) {

#if VDBG

RLOGD("sendSms: serial %d", serial);

#endif

dispatchStrings(serial, mSlotId, RIL\_REQUEST\_SEND\_SMS, false,

2, message.smscPdu.c\_str(), message.pdu.c\_str());

return Void();

}