

Introducing SMS

If you own a mobile phone that's less than two decades old, chances are you're familiar with SMS messaging. SMS (short messaging service) is now one of the most-used features on mobile phones, with many people favoring it over making phone calls.

SMS technology is designed to send short text messages between mobile phones. It provides support for sending both text messages (designed to be read by people) and data messages (meant to be consumed by applications).

As a mature mobile technology, there's a lot of information out there that describes the technical details of how an SMS message is constructed and transmitted over the air. Rather than rehash that here, the following sections focus on the practicalities of sending and receiving text and data messages within Android.

Using SMS in Your Application

Android offers full access to SMS functionality from within your applications with the `SmsManager`. Using the SMS Manager, you can replace the native SMS application or create new applications that send text messages, react to incoming texts, or use SMS as a data transport layer.

SMS message delivery is not timely, so SMS is not really suitable for anything that requires real-time responsiveness. That said, the widespread adoption and resiliency of SMS networks make it a particularly good tool for delivering content to non-Android users and reducing the dependency on third-party servers.

As a ubiquitous technology, SMS offers a mechanism you can use to send text messages to other mobile phone users, irrespective of whether they have Android phones.

Compared to the instant messaging mechanism available through the GTalk Service, using SMS to pass data messages between applications is slow, possibly expensive, and suffers from high latency. On the other hand, SMS is supported by almost every phone on the planet, so where latency is not an issue, and updates are infrequent, SMS data messages are an excellent alternative.

Sending SMS Messages

SMS messaging in Android is handled by the `SmsManager`. You can get a reference to the SMS Manager using the static method `SmsManager.getDefault()`, as shown in the snippet below.

```
SmsManager smsManager = SmsManager.getDefault();
```

To send SMS messages, your applications require the `SEND_SMS` permission. To request this permission, add it to the manifest using a `uses-permission` tag, as shown below:

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

Sending Text Messages

To send a text message, use `sendTextMessage` from the SMS Manager, passing in the address (phone number) of your recipient and the text message you want to send, as shown in the snippet below:

```
String sendTo = "5551234";  
String myMessage = "Android supports programmatic SMS messaging!";  
smsManager.sendTextMessage(sendTo, null, myMessage, null, null);
```

The second parameter can be used to specify the SMS service center to use; entering null as shown in the previous snippet uses the default service center for your carrier.

The final two parameters let you specify Intents to track the transmission and successful delivery of your messages.

To react to these Intents, create and register Broadcast Receivers as shown in the next section.

Tracking and Confirming SMS Message Delivery

To track the transmission and delivery success of your outgoing SMS messages, implement and register Broadcast Receivers that listen for the actions you specify when creating the Pending Intents you pass in to the `sendTextMessage` method.

The first Pending Intent parameter, `sentIntent`, is fired when the message is either successfully sent or fails to send. The result code for the Broadcast Receiver that receives this Intent will be one of:

- ❑ `Activity.RESULT_OK` To indicate a successful transmission.
- ❑ `SmsManager.RESULT_ERROR_GENERIC_FAILURE` To indicate a nonspecific failure.
- ❑ `SmsManager.RESULT_ERROR_RADIO_OFF` When the connection radio is turned off.
- ❑ `SmsManager.RESULT_ERROR_NULL_PDU` To indicate a PDU failure.

The second Pending Intent parameter, `deliveryIntent`, is fired only after the destination recipient receives your SMS message.

The following code snippet shows a typical pattern for sending an SMS and monitoring the success of its transmission and delivery:

```
String SENT_SMS_ACTION = "SENT_SMS_ACTION";
String DELIVERED_SMS_ACTION = "DELIVERED_SMS_ACTION";
// Create the sentIntent parameter
Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPI = PendingIntent.getBroadcast(getApplicationContext(), 0, sentIntent, 0);
// Create the deliveryIntent parameter
Intent deliveryIntent = new Intent(DELIVERED_SMS_ACTION);
PendingIntent deliverPI = PendingIntent.getBroadcast(getApplicationContext(), 0, deliveryIntent, 0);
// Register the Broadcast Receivers
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context _context, Intent _intent) {
        switch (getResultCode()) {
            case Activity.RESULT_OK:
                [... send success actions ... ]; break;
            case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
                [... generic failure actions ... ]; break;
            case SmsManager.RESULT_ERROR_RADIO_OFF:
                [... radio off failure actions ... ]; break;
            case SmsManager.RESULT_ERROR_NULL_PDU:
                [... null PDU failure actions ... ]; break;
        }
    }
},
new IntentFilter(SENT_SMS_ACTION));
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context _context, Intent _intent) {
        [... SMS delivered actions ... ]
    }
},
new IntentFilter(DELIVERED_SMS_ACTION));
// Send the message
smsManager.sendTextMessage(sendTo, null, myMessage, sentPI, deliverPI);
```

Monitoring Outgoing SMS Messages

The Android debugging bridge supports sending SMS messages between multiple emulator instances. To send an SMS from one emulator to another, specify the port number of the target emulator as the “to” address when sending a new message.

Android will automatically route your message to the target emulator instance, where it’ll be handled as a normal SMS.

Conforming to the Maximum SMS Message Size

SMS text messages are normally limited to 160 characters, so longer messages need to be broken into a series of smaller parts. The SMS Manager includes the `divideMessage` method, which accepts a string as an input and breaks it into an `ArrayList` of messages wherein each is less than the allowable size. Use `sendMultipartTextMessage` to transmit the array of messages, as shown in the snippet below:

```

ArrayList<String> messageArray = smsManager.divideMessage(myMessage);
ArrayList<PendingIntent> sentIntents = new ArrayList<PendingIntent>();
for (int i = 0; i < messageArray.size(); i++)
    sentIntents.add(sentPI);
smsManager.sendMultipartTextMessage(sendTo,
    null,
    messageArray,
    sentIntents, null);

```

The `sentIntent` and `deliveryIntent` parameters in the `sendMultipartTextMessage` method are `ArrayLists` that can be used to specify different `Pending Intents` to fire for each message part.

Sending Data Messages

You can send binary data via SMS using the `sendDataMessage` method on an SMS Manager. The `sendDataMessage` method works much like `sendTextMessage`, but includes additional parameters for the destination port and an array of bytes that constitute the data you want to send.

The following skeleton code shows the basic structure of sending a data message:

```

Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPI = PendingIntent.getBroadcast(getApplicationContext(), 0, sentIntent, 0);
short destinationPort = 80;
byte[] data = [ ... your data ... ];
smsManager.sendDataMessage(sendTo, null, destinationPort, data, sentPI, null);

```

Listening for SMS Messages

When a new SMS message is received by the device, a new broadcast `Intent` is fired with the `android.provider.Telephony.SMS_RECEIVED` action. Note that this is a `String` literal, SDK 1.0 does not include a reference to this string so you must specify it explicitly when using it in your applications.

For an application to listen for SMS `Intent` broadcasts, it first needs to have the `RECEIVE_SMS` permission granted. Request this permission by adding a `uses-permission` tag to the application manifest, as shown in the following snippet:

```

<uses-permission
android:name="android.permission.RECEIVE_SMS"/>

```

The SMS broadcast `Intent` includes the incoming SMS details. To extract the array of `SmsMessage` objects packaged within the SMS broadcast `Intent` bundle, use the `pdu` key to extract an array of SMS pdus, each of which represents an SMS message. To convert each pdu byte array into an SMS Message object, call `SmsMessage.createFromPdu`, passing in each byte array as shown in the snippet below:

```

Bundle bundle = intent.getExtras();
if (bundle != null) {
    Object[] pdus = (Object[]) bundle.get("pdus");
    SmsMessage[] messages = new SmsMessage[pdus.length];
    for (int i = 0; i < pdus.length; i++)
        messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
}

```

Each `SmsMessage` object contains the SMS message details, including the originating address (phone number), time stamp, and the message body.

The following example shows a `Broadcast Receiver` implementation whose `onReceive` handler checks incoming SMS texts that start with the string `@echo`, and then sends the same text back to the phone that sent it:

```

public class IncomingSMSReceiver extends BroadcastReceiver {
    private static final String querystring = "@echo ";
    private static final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
    public void onReceive(Context _context, Intent _intent) {
        if (_intent.getAction().equals(SMS_RECEIVED)) {
            SmsManager sms = SmsManager.getDefault();
            Bundle bundle = _intent.getExtras();
            if (bundle != null) {
                Object[] pdus = (Object[]) bundle.get("pdus");
                SmsMessage[] messages = new SmsMessage[pdus.length];
                for (int i = 0; i < pdus.length; i++)

```

```

messages[i] = SmsMessage.createFromPdu((byte[]) pdu[i]);
for (SmsMessage message : messages) {
String msg = message.getMessageBody();
String to = message.getOriginatingAddress();
if (msg.toLowerCase().startsWith(queryString)) {
String out = msg.substring(queryString.length());
sms.sendTextMessage(to, null, out, null, null);
}
}
}
}
}
}

```

To listen for incoming messages, register the Broadcast Receiver using an Intent Filter that listens for the `android.provider.Telephony.SMS_RECEIVED` action String, as shown in the code snippet below:

```

final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
IntentFilter filter = new IntentFilter(SMS_RECEIVED);
BroadcastReceiver receiver = new IncomingSMSReceiver();
registerReceiver(receiver, filter);

```

Simulating Incoming SMS Messages

There are two techniques available for simulating incoming SMS messages in the emulator. The first was described previously in this section; you can send an SMS message from one emulator to another by using its port number as the destination address.

Alternatively, you can use the Android debug tools introduced in Chapter 2 to simulate incoming SMS messages from arbitrary numbers, as shown in Figure 9-2.

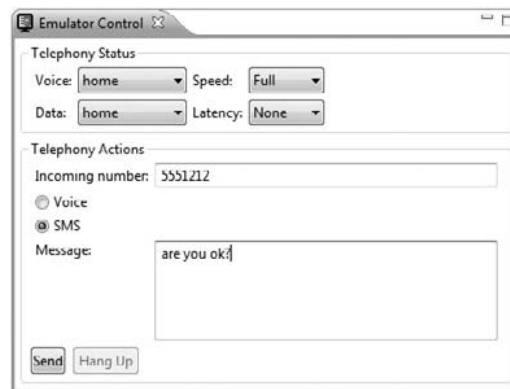


Figure 9-2

Handling Data SMS Messages

For security reasons, the version 1 release has restricted access to receiving data messages. The following section has been left to indicate how likely future functionality may be made available.

Data messages are received in the same way as a normal SMS text message and are extracted in the same way as shown in the above section.

To extract the data transmitted within a data SMS, use the `getUserData` and `getUserDataHeader` methods, as shown in the following snippet:

```

byte[] data = msg.getUserData();
SmsHeader header = msg.getUserDataHeader();

```

The `getUserData` method returns a byte array of the data included in the message, while `getUserDataHeader` returns an array of metadata elements used to describe the data contained in the message.