Chapter 9 Text To Speech (TTS) and Audio

9.1 Introduction

Android provides the text-to-Speech (TTS) feature for applications to synthesize speech from text in different languages.

Typically, a text-to-speech system, or referred to as an *engine*, is composed of two parts, a front-end and a back-end. The front-end performs some preprocessing of a raw text, converting symbols such as numbers and abbreviations to spelled-out words. It then performs text-to-phoneme conversion, assigning phonetic transcriptions to each word, and dividing the text into prosodic units, such as phrases, clauses, and sentences. The front-end output consists of the symbolic linguistic representation of prosody information and phonetic transcriptions. The back-end is the synthesizer that converts phonemes into sound waves. Figure 9-1 below shows such a system.

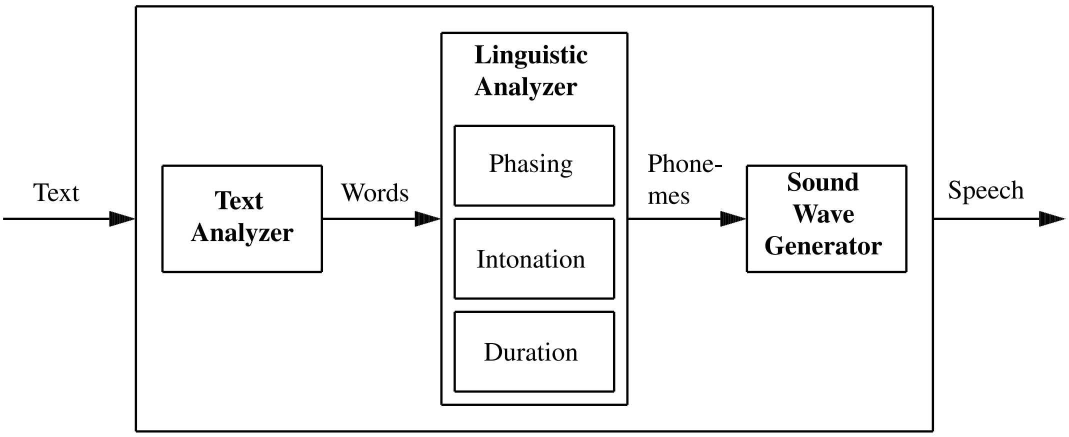


Figure 9-1 A Typical Text-to-Speech System

Researchers created the first generation computer-based speech synthesizers in the late 1950s. Noriko Umeda et al. developed the first general English text-to-speech system in 1968 at the Electrotechnical Laboratory of Japan. Since then significant advancement has been made in developing TTS systems in various languages. Sun Electronic first incorporated speech synthesis features in a video game in 1980. Milton Bradly Company first debuted a multi-player electronic game that has voice synthesis features in 1980.

Nowadays, a number of open-source TTS systems, usually written in C/C++ or Java are available and free for download. In particular, *FreeTTS* is a speech synthesizer written entirely in Java and is based upon *Flite*, which is a small run-time speech synthesis engine developed at Carnegie Mellon University (CMU). Flite is written completely in C, not using any C++ or scheme features for the reasons of portability, size and speed. It is derived from the Festival Speech Synthesis System from the University of Edinburgh and the Festvox project from CMU. FreeTTS supports a subset of the JSAPI 1.0 java speech synthesis specification.

Android provides a default text-to-speech (TTS) engine (Pico) with limited APIs. Other third party TTS engines are also available in the market. For instance, Samsung and LG preload their text-to-speech sets in their Android products. Google TTS that comes with some devices, and Ivona TTS HQ that has high-quality sounding voices are well-known free engines. Popular low cost engines include *Classic TTS Engine* (which covers 40+ languages), CereProc, and Loquendo TTS Susan.

9.2 The *TextToSpeech* Class

Android provides the class *TextToSpeech* for synthesizing speech from text for immediate playback or creating a sound file. Using the class, an application can incoporate rich speech features such as speaking different languages, setting the pitch level and the speaking speed.

To utilize a *TextToSpeech* object to synthesize speech from a text, the application must first complete an initialization stage of the object. To know when the initialization is complete, we have to implement the **OnInitListner** of *TextToSpeech* to obtain a notification of completing the initialization. If the object is no longer in use, the application should call the **shutdown**() method to release any resources used by the speech engine. One can refer to Android's developer site for the detailed usage and all the methods of this class. Here we discuss a few simple features and present an example to illustrate its basic usage.

To use the class *TextToSpeech*, we need to implement its interface *OnInitListener*, which defines a callback to be invoked to indicate the completion of the *TextToSpeech* engine initialization; we have to override the abstract method **onInit**, which is called to signal the completion. The following is an example implementation:

```
public class MainActivity extends Activity
                      implements View.OnClickListener, OnInitListener
{
  ImageButton ttsButton;
  EditText editText1;
  TextToSpeech tts;
  .....
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    .....
    ttsButton.setOnClickListener ( this );
    tts = new TextToSpeech(this, this);
  }

  public void onInit(int state) {
    if (state == TextToSpeech.SUCCESS) {
       int result = tts.setLanguage(Locale.US);

      if (result == TextToSpeech.LANG_MISSING_DATA
             || result == TextToSpeech.LANG_NOT_SUPPORTED) {
         .....
      } else {
        ttsButton.setEnabled(true);
        speak();
      }
    }
  }
}
```

In this example, we have used the constructor

TextToSpeech (Context *context*, TextToSpeech.OnInitListener *listener*)

of the *TextToSpeech* class, which uses the default TTS engine. It will also initialize the associated *TextToSpeech* engine if it is not already running. The variable *context* is the context that the instance is running in. The *listener* is the *TextToSpeech.OnInitListener* that will be called when the

TTS engine has initialized. The listener may be called immediately, in the case of a failure, before the *TextToSpeech* instance is fully constructed.

The method **setLanguage**() takes a *Locale* object as parameter, and sets the engine to speak the specified language. The class *Locale* represents a **language/country/variant** combination. Locale objects can be used to alter the presentation of information such as numbers or dates to suit the conventions in the region an application describes. The following table shows some available locales.

No.	**Locale**
1	US
2	CANADA_FRENCH
3	GERMANY
4	ITALY
5	JAPAN
6	CHINA

After we have specified the language, we can call the **speak** method of the class to produce speech from the text like the following code:

```
private void speak() {
  String text1 = editText1.getText().toString();
  tts.speak(text1, TextToSpeech.QUEUE_FLUSH, null);
}
```

The TTS engine manages a global queue of all the entries, also known as *utterances*, to synthesize speech. The **speak**() method of *TextToSpeech* produces speech from the text string using the specified queuing strategy and speech parameters. The method, which is asynchronous, adds the request to the TTS request queue and then returns. Note that at the time this method returns, the synthesis might not have finished or even started! Users are recommended to set an utterance progress listener and use the KEY_PARAM_UTTERANCE_ID parameter to reliably detect errors during synthesis.

The constant QUEUE_FLUSH represents the queue mode where all entries in the playback queue are dropped and replaced by the new entry. Thus in the example, *text1* is spoken immediately. Note that queues are flushed with respect to a given calling app but entries in the queue from other callees are not discarded.

If we want the engine to finish the utterances in the queue before playing the current one, we can use set the queue mode QUEUE_ADD, which adds the new entry at the end of the queue. For example, the statements

```
tts.speak(text1, TextToSpeech.QUEUE_FLUSH, null);
tts.speak(text2, TextToSpeech.QUEUE_ADD, null);
```

will first finish playing *text1* before playing *text2*.

Besides the **speak** method, the following table lists some available useful methods of the class.

Method	Description
addSpeech(String *text*, String *file*)	Adds a mapping between a text and a sound file.
getLanguage()	Returns a *Locale* object that describes the language.
isSpeaking()	Checks if *TextToSpeech* engine is busy speaking.
setPitch(float *pitch*)	Sets the speech pitch for the *TextToSpeech* engine.
setSpeechRate(float *speechRate*)	Sets the speech speed.
shutdown	Releases resources used by the *TextToSpeech* engine.
stop()	Stops speaking.

9.3 A Simple TTS Example

We present in this section a very simple example of using the class *TextToSpeech* to generate speech from the text entered in an *EditText* field. Suppose we call the project and the application *TtsDemo*, and the package *tts.ttsdemo*. As usual, we first use Eclipse IDE to create the default files. We modify the file *MainActivity.java* to implement *OnInitListener* as discussed above. We also declare a *TextView* for displaying related messages, an *EditText* for the user to enter text, and an *ImageButton* for the user to click on to convert the text to speech. The following Listing shows the modified code.

Program Listing 9-1 *MainActivity.java* of *TtsDemo*

```
package tts.ttsdemo;

import java.util.Locale;
import android.app.Activity;
import android.os.*;
import android.view.*;
import android.widget.*;
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.OnInitListener;

public class MainActivity extends Activity
             implements View.OnClickListener, OnInitListener
{
   ImageButton ttsButton;
   TextView msg;
   EditText editText1;
   TextToSpeech tts;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       ttsButton = (ImageButton) findViewById(R.id.ttsButton);
       editText1 = (EditText) findViewById(R.id.editText1);
       msg = (TextView) findViewById(R.id.msg);
       ttsButton.setOnClickListener ( this );
       tts = new TextToSpeech(this, this);
   }

   public void onClick(View view) {
    speak();
   }
   @Override
   public void onInit(int state) {
     if (state == TextToSpeech.SUCCESS) {
       msg.setText("TTS initialized successfully!\n " +
            "Enter a sentence and click TTS.");
       int result = tts.setLanguage(Locale.US);
       if (result == TextToSpeech.LANG_MISSING_DATA
                || result == TextToSpeech.LANG_NOT_SUPPORTED) {
         msg.setText("TTS: Language not supported!");
       } else {
          ttsButton.setEnabled(true);
          speak();
        }
      } else {
         msg.setText("TTS: Initilization Failed!");
      }
    }

    private void speak() {
      String text = editText1.getText().toString();
      tts.speak(text, TextToSpeech.QUEUE_FLUSH, null);
    }
    @Override
    public void onDestroy() {
        //  Shut down tts to release resources!
        if (tts != null) {
            tts.stop();
            tts.shutdown();
        }
        super.onDestroy();
    }
}
```

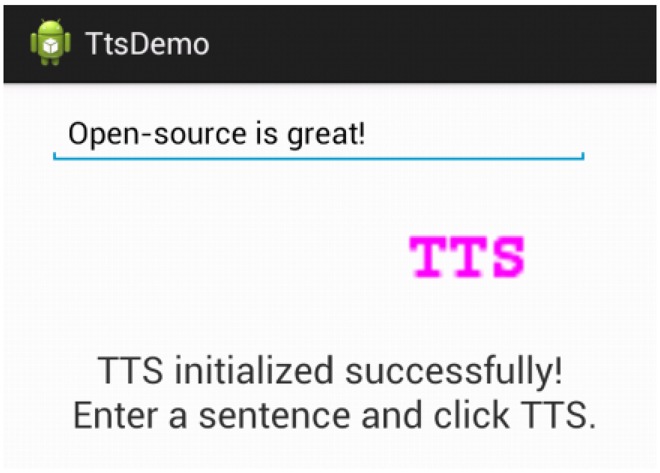
The code is very simple and most of its features have been explained in the previous section.
We also need to modify the layout file, *res/layout/activity_main.xml* that defines the UI of our project. The modified file is listed in Listing 9-2 below.

Program Listing 9-2 *activity_main.xml* of Project *TtsDemo*

```
 <?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <LinearLayout
        android:id="@+id/linearLayout1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="12pt"
        android:layout_marginRight="12pt"
        android:layout_marginTop="4pt" >
        <EditText
            android:id="@+id/editText1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginRight="6pt"
            android:layout_weight="1"
            android:inputType="text" >
        </EditText>
    </LinearLayout>
    <RelativeLayout
        android:id="@+id/linearLayout2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="6pt"
        android:layout_marginRight="6pt"
        android:layout_marginTop="4pt" >
        <ImageButton
            android:id="@+id/ttsButton"
            android:layout_width="90dp"
            android:layout_height="90dp"
            android:layout_alignParentRight="true"
            android:layout_alignParentTop="true"
            android:layout_marginRight="60dp"
            android:adjustViewBounds="true"
            android:background="@drawable/tts"
            android:scaleType="centerCrop"
            android:src="@drawable/tts" />
    </RelativeLayout>
    <TextView
        android:id="@+id/msg"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="4pt"
        android:layout_marginRight="4pt"
        android:layout_marginTop="2pt"
        android:gravity="center_horizontal"
        android:textSize="10pt" >
    </TextView>
 </LinearLayout>
------------------------------------------------------------------------
```

We also need to put a PNG image file in a drawable directory (we put it in *res/drawable-xhdpi*) to represent the TTS button that we click on to speak the entered text.

When we run the application, we will have a UI similar to the one shown in Figure 9-2. Clicking on the TTS image button, the text entered in the *EditText* field will be converted to audible speech.



Open-source is great!

TTS initialized successfully!
Enter a sentence and click TTS.

Figure 9-2 UI of *TtsDemo* Project

If we want to change the **language of the speech**, we can set the new language using the method **setLanguage** like

tts.setLanguage(Locale.CHINESE);
which sets the speaking language to Chinese.

To set the **pitch rate** to another value other than the default value of 1, we can use the method **setPitch** like

tts.setPitch (0.8);
which sets the pitch level to 0.8.

The **speech speed** can be also changed from its default value of 1.0. For example,

tts.setSpeechRate (1.5);
increases the speech speed by 50%.

9.4 Writing Speech to a File

9.4.1 Playback Options

We can always associate an audio stream that is played with a stream type, defined in *android.media.AudioManager*. The **speak**() method actually takes three parameters:

speak(String text, int queueMode, HashMap<String, String> params)
The class *HashMap<String, String>*, the data type of the last parameter of **speak**, is a hash table based implementation of the *Map* interface, which associates a value with a key. The first parameter type (*String* here) inside the angular brackets represents the type of keys maintained by the map and the second parameter type (also *String* here) represents the type of mapped values. This parameter of **speak** allows us to pass to the TTS engine an optional *HashMap* parameter, specified as a key/value pair. (If no such pair is needed in our application, we can pass in **null** for the parameter.)

We can make use of this parameter to customize our playback. For example, consider an alarm clock that speaks text and the user can choose its settings. We would then like the text to be played on the stream type of *AudioManager*.STREAM_ALARM. We can use the *HashMap* parameter to change the stream type of our utterances:

```
HashMap<String, String> hashMap = new HashMap();
hashMap.put ( TextToSpeech.Engine.KEY_PARAM_STREAM,
             String.valueOf(AudioManager.STREAM_ALARM) );
tts.speak( text1, TextToSpeech.QUEUE_FLUSH, hashMap );
tts.speak( text2, TextToSpeech.QUEUE_ADD, hashMap );
```

The **put**() method of *HashMap* in the code associates the *String*

TextToSpeech.Engine.KEY_PARAM_STREAM,
which is the key, with the STREAM_ALARM *String*, which is the key value. We use this key pair to identify the utterance we need for some other purposes. For this to work, we also need to implement the *TextToSpeech.OnUtteranceCompletedListener* interface in our activity:

```
tts.setOnUtteranceCompletedListener(
                        (OnUtteranceCompletedListener) this);
hashMap.put(TextToSpeech.Engine.KEY_PARAM_STREAM,
                  String.valueOf(AudioManager.STREAM_ALARM));
tts.speak( text1, TextToSpeech.QUEUE_FLUSH, hashMap );
// Add another optional parameter to hashMap
hashMap.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,
                                          "Alarm message ID");
tts.speak( text2, TextToSpeech.QUEUE_ADD, hashMap );
```

This code notifies the activity of the application when it has finished synthesizing *text2*. The application can then do some desired task such as playing some music or displaying a message on the screen:

```
public void onUtteranceCompleted ( String utteranceId ) {
  if ( utteranceId.equals("Alarm message ID") )
    doSomething();
}
```

Note that the *OnUtteranceCompletedListener* can be assigned to the *TextToSpeech* object only after the TTS **onInit** has been called. However, this listener has been deprecated, and developers are recommended to use *UtteranceProgressListener*, which is added in API level 15. *UtteranceProgressListener* is a listener for events relating to the progress of an utterance through the synthesis queue. Each utterance is associated with a call to the method **speak**(String, int, HashMap) or **synthesizeToFile**(String, HashMap, String) with an associated utterance identifier, as per KEY_PARAM_UTTERANCE_ID. The **speak** method allows multiple threads to call its specified callbacks. *UtteranceProgressListener* is an abstract class, so we have to implement it, and we can do that using an inner class:

```
tts.setOnUtteranceProgressListener ( new
  UtteranceProgressListener()
  {
    @Override
    public void onDone(String utteranceId){
      if ( utteranceId == "Alarm message ID")
        onUtteranceCompleted ( utteranceId );
    }

    @Override
    public void onError(String utteranceId){
    }

    @Override
    public void onStart(String utteranceId){
    }
  });
```

We should replace the statement *tts.setOnUtteranceCompletedListener(this);* by the above code.

9.4.2 Saving Speech

We have discussed how to convert text to speech using the **speak**() method of the class *TextToSpeech*. There can be situations in which we would like the synthesized speech to be recorded in an audio file so that next time when we want to play the speech, we do not have to go through the synthesis process again. The class *TexttToSpeech* provides the method **synthesizeToFile**() to record the synthesized data in an audio format. The following code shows how this is done:

```
HashMap<String, String> hashMap = new HashMap();
String keyText = "A string for key";
String outFile = "sample1.wav";
hashMap.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, keyText);
tts.synthesizeToFile(keyText, hashMap, outFile);
tts.speak( text1, TextToSpeech.QUEUE_FLUSH, hashMap );
tts.speak( text2, TextToSpeech.QUEUE_ADD, hashMap );
```

The code saves the synthesized audio data in *sample1.wav* in the *.wav* format after the synthesis process has finished. (The completion of the synthesis can be notified by an optional utterance identifier as discussed above.) The *.wav* file can be played just like any other audio resource with *android.media.MediaPlayer*.

The **put**() method of *HashMap* in the example associates the String *keyText* with
TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,
which is of type *String* and is a parameter key to identify an utterance in
TextToSpeech.OnUtteranceCompletedListener or *UtteranceProgressListener*
after the text has been spoken, or a file has been played back or a silence duration has elapsed.

There are other ways to associate audio resources with speech. Continuing the above example, suppose we have a *.wav* file that contains the synthesized data associated with *keyText*. We can also associate *keyText* with an audio resource, which can be accessed with one of the two **addSpeech**() methods:
tts.addSpeech(keyText, outFile);
This way any call to **speak**() using the key string “A string for key” will result in the playback of *outFile*. If the file is missing, then **speak**() will ignore the file and simply synthesize the given string to produce speech. We can take advantage of this feature to provide an option to the user to customize how “A string for key” sounds: she can record her own version or use the synthesized sound by removing the sound file. Regardless of the option, we can use the same code to add the text to the speech queue:
tts.speak(keyText, TextToSpeech.QUEUE_ADD, hashAlarm);

9.5 Speak From a Text File with Pause/Resume

9.5.1 Synchronization by Condition Variable

We have discussed how to convert a short text to speech using the class *TextToSpeech*. We may apply the same technique to speak the text from a file. This works fine if the file is small as we can simply read the whole file into a *String* object and speak the text of the *String* as we did in the above sections. However, if the file is large, this won’t work because a *String* object cannot be infinitely large to hold any size of text. Neither can we keep adding text to the utterance queue, because this may make the queue overflow as a queue’s capacity is also finite.

A simple solution to this problem is to let the engine finish speaking a certain text segment and notify the activity when it is done, before reading in new text from the file. This can be accomplished by reading the file line by line and using a condition variable to force the reading activity to wait after the utterance queue has accumulated a certain number of lines of text. We can use an integer variable *count* to keep track of the lines in the queue. By making use of the third parameter of the **speak** method that we discussed in previous sections, we can notify the reading activity and decrement *count*:

```
final Lock mutex = new ReentrantLock();
final Condition textSpoken = mutex.newCondition();
int count = 0;

private void speak() throws InterruptedException {
  String str = null;
  InputStream is = null;
  HashMap<String, String> hashMap = new HashMap();
  String keyText = "Text Spoken ID";
  hashMap.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,keyText);
  MyUtteranceProgressListener listener = new
                          MyUtteranceProgressListener();
  tts.setOnUtteranceProgressListener ( listener );
  try {
    is = getResources().openRawResource(R.raw.myfile);
    BufferedReader reader = new
              BufferedReader(new InputStreamReader(is));
    while ((str = reader.readLine()) != null) {
      mutex.lock();
      count++;
      while ( count > 50 )
        textSpoken.await();   // Condition wait
      mutex.unlock();
      tts.speak( str, TextToSpeech.QUEUE_ADD, hashMap );
    }
  }  catch(IOException e) {
           Log.e(LOG_APP_TAG, e.getMessage());
  }
}

public void onUtteranceCompleted ( String utteranceId ) {
  if ( utteranceId.equals("Text Spoken ID")) {
    mutex.lock();
    count--;
    mutex.unlock();
    textSpoken.signal();
  }
}
```

In this code, *textSpoken* is the condition variable we mentioned. The reading activity waits (sleeps) when the counter *count* is larger than 50. When the engine has completed speaking one line of text, it calls **onUtteranceCompleted**(), passing to it the utterance ID, “Text Spoken ID”; the method decrements the *count*, and wakes up any activity that waits on the condition variable *textSpoken*.

9.5.2 Speak Using a Different Thread

We can also add a pause button to the application, so that the app stops synthesizing when the button is clicked and resumes the task when the resume button is clicked. We can call the **stop**() method of *TextToSpeech* to stop synthesizing. However, the method will also discard all utterances in the queue. So we need to buffer the lines we have read from the file and keeps track of the current line the engine is speaking. When we resume the synthesis, we can add the text lines saved in the buffer to the utterance queue again to the engine, which has discarded them in the queue when executing **stop**().

Since the UI has to interact with the user all the time, we have to start the speech engine with a different thread so that the main activity will not be mostly consumed by the speech task. Suppose we again use Eclipse IDE to develop such an application. We call the names of the project and the application, *FileToSpeech* and the name of the package, *tts.filetospeech*. We create another class file *Speech.java*, which contains the code of the thread that manages the engine and our main program, *MainActivity.java* mainly manages the UI and the initialization of the speech engine. Listing 9-3 below shows the complete code of the activity class *MainActivity*.

Program Listing 9-3 *MainActivity.java* of Project *FileToSpeech*

```
package tts.filetospeech;
```

```
import java.io.*;
import java.util.*;
import android.app.Activity;
import android.os.*;
import android.util.Log;
import android.view.View;
import android.widget.*;
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.*;

public class MainActivity extends Activity
                                implements  OnInitListener
{
  ImageButton ttsButton;
  Button pauseButton;
  Button resumeButton;
  TextView msg;
  TextToSpeech tts;
  InputStream is = null;
  BufferedReader reader;
  boolean firstTime = true;  //First time reading text to synthesize
  Speech speech = null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ttsButton = (ImageButton) findViewById(R.id.ttsButton);
    msg = (TextView) findViewById(R.id.msg);
    pauseButton = (Button) findViewById(R.id.pauseButton);
    resumeButton = (Button) findViewById(R.id.resumeButton);
    tts = new TextToSpeech(this, this);
  }

  // Start button clicked
  public void onStart(View view) {
    if ( firstTime ) {
      is = getResources().openRawResource(R.raw.myfile);
      speech = new Speech ( tts, is );
      Thread speechThread = new Thread ( speech );
      speechThread.start();   // start thread to speak
      firstTime = false;
    }
  }

  // Pause button clicked
  public void onPause ( View view ) {
    if ( speech != null )
      speech.speechPause();
  }

  // Resume button clicked
  public void onResume ( View view ){
    if ( speech != null )
      speech.speechResume();
  }

  @Override
  public void onInit(int state) {
    if (state == TextToSpeech.SUCCESS) {
      msg.setText("TTS initialized successfully!\n " +
            "Click TTS to start speaking.");
      int result = tts.setLanguage(Locale.US);
      if (result == TextToSpeech.LANG_MISSING_DATA
             || result == TextToSpeech.LANG_NOT_SUPPORTED) {
        msg.setText("TTS: Language not supported!");
      } else {
        ttsButton.setEnabled(true);
      }
    } else {
      msg.setText("TTS: Initilization Failed!");
    }
  }

  @Override
  public void onDestroy() {
    if (tts != null) {
      tts.stop();
      tts.shutdown();
    }
    super.onDestroy();
  }
}
----------------------------------------------------------------------
```

This class mainly handles the UI and initializes the speech engine. The code that speaks the text from the file is listed below:

Program Listing 9-4 *Speech.java* of Project *FileToSpeech*

```
package tts.filetospeech;

import java.io.*;
import java.util.*;
import android.util.Log;
import java.util.concurrent.locks.*;
import android.speech.tts.*;
import android.speech.tts.TextToSpeech.OnInitListener;

public class Speech  implements Runnable
{
  TextToSpeech tts;
  boolean firstTime = true;
  HashMap<String, String> hashMap;
  InputStream is = null;
  BufferedReader reader = null;
  final Lock mutex = new ReentrantLock();
  final Condition textSpoken = mutex.newCondition();
  final Lock mutex1 = new ReentrantLock();
  final Condition pauseCond = mutex1.newCondition();
```

```
int count = 0;
int position = 0;
final int LEN = 20;
String buffer[] = new String[LEN];
boolean pausing = false;
boolean starting = true;
boolean resuming = false;
// Constructor
public Speech ( TextToSpeech tts0, InputStream is0 )
{
  tts = tts0;
  is = is0;
}

@Override
public void run() {
  try {
    speak();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}

private void speak() throws InterruptedException {
  String str = null;
  if ( firstTime ){  // Initialization only once
    String keyText = "Text Spoken ID";
    hashMap = new HashMap();
    hashMap.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,keyText);
    MyUtteranceProgressListene listener =
                         new MyUtteranceProgressListene();
    tts.setOnUtteranceProgressListener ( listener );
    firstTime = false;
  }
  try {
    if ( reader == null  )
      reader = new BufferedReader(new InputStreamReader(is));
    while ((str = reader.readLine()) != null) {
      mutex1.lock();
      while ( pausing )
        pauseCond.await();  // Condition wait
      mutex1.unlock();
      mutex.lock();
      count++;
      if ( starting ){
        tts.speak( str, TextToSpeech.QUEUE_FLUSH, hashMap );
        starting = false;     // starts once only
      } else {
        if ( resuming ) {
          // copy buffer string back
          int j =  position - 1;
          if ( j < 0 ) j = j + LEN;
          for ( int i = 1; i < count; i++){
            if ( i == 1 )
              tts.speak( buffer[j],
                     TextToSpeech.QUEUE_FLUSH, hashMap );
            else
              tts.speak( buffer[j], TextToSpeech.QUEUE_ADD,
                                                   hashMap );
            j = ( j + 1) % LEN;
          }
        }  // if ( resuming )
        tts.speak( str, TextToSpeech.QUEUE_ADD, hashMap );
      } // else
      while ( count > LEN )
        textSpoken.await();   // Condition wait
      mutex.unlock();
      buffer[position] = str;
      position++;
      if ( position == LEN )
        position = 0;         // Circular buffer
    }  // while reader
  }  catch(IOException e) {
     Log.e("speak", e.getMessage());
  }
}

public void onUtteranceCompleted ( String utteranceId ) {
  if ( utteranceId.equals("Text Spoken ID")) {
    mutex.lock();
    count--;
    textSpoken.signal();
    mutex.unlock();
  }
}

// Innner class for UtteranceProgressListener
class MyUtteranceProgressListener extends UtteranceProgressListener
{
  @Override
  public void onDone(String utteranceId)
  {
    onUtteranceCompleted ( utteranceId );
  }
  @Override
  public void onError(String utteranceId){
  }
  @Override
  public void onStart(String arg0) {
  }
}

public void speechPause()
{
  if ( pausing ) return;  // engine already stopped
  mutex1.lock();
  pausing = true;
  mutex1.unlock();
  tts.stop();     // stop engine, clears utterance queue
}
```

```
   public void speechResume ()
   {
     if ( !pausing ) return;  // engine already running
     mutex1.lock();
     pausing = false;
     resuming = true;
     pauseCond.signal();   // Wake up the waiting method
     mutex1.unlock();
   }
 }
--------------------------------------------------------------------------
```

In this class, **speak**() is the method that does the reading of file and synthesizing. It uses the two condition variables, *textSpoken*, and *pauseCond* to block the engine. It blocks (waits) on *pauseCond* when the user clicks the *Pause* button, setting the variable *pausing* to true. It is woken up by the signal of *pauseCond*, when someone clicks *Resume*, setting *pausing* to false and executing the statement *pauseCond.signal();*. and making *pauseCond* to signal.

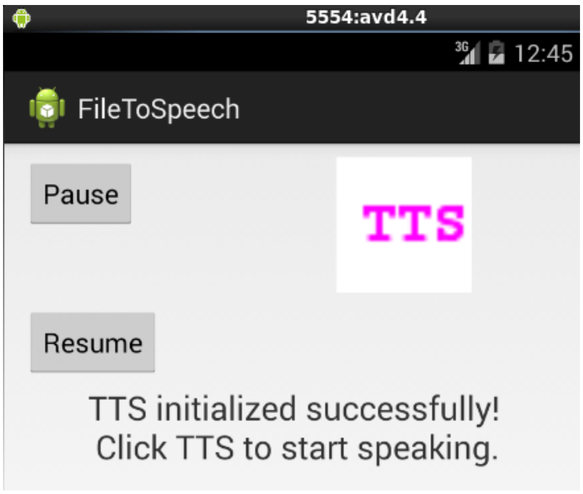
In this program, each utterance is a line of text. When the *Pause* button is clicked, the rest of the line is discarded. When it resumes, it starts from the next line. You can modify this program to handle this situation differently. It can start from the current line rather than the next, or it can finish speaking the current line before it stops. Or you can make an utterance to be a word instead of a line.

The UI layout, including the button listeners, is defined in *res/layout/activity_main.xml*, which is listed below.

Program Listing 9-5 *activity_main.xml* of Project *FileToSpeech*

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <RelativeLayout
        android:id="@+id/linearLayout1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="6pt"
        android:layout_marginRight="6pt"
        android:layout_marginTop="4pt" >
        <Button
        android:id="@+id/pauseButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onPause"
        android:text="Pause" />
        <ImageButton
            android:id="@+id/ttsButton"
            android:layout_width="90dp"
            android:layout_height="90dp"
            android:layout_alignParentRight="true"
            android:layout_alignParentTop="true"
            android:layout_marginRight="60dp"
            android:adjustViewBounds="true"
            android:background="@drawable/tts"
            android:scaleType="centerCrop"
            android:onClick="onStart"
            android:src="@drawable/tts" />
    </RelativeLayout>
    <RelativeLayout
        android:id="@+id/linearLayout2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="6pt"
        android:layout_marginRight="6pt"
        android:layout_marginTop="4pt" >
        <Button
          android:id="@+id/resumeButton"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:onClick="onResume"
          android:text="Resume" />
    </RelativeLayout>
    <TextView
        android:id="@+id/msg"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="4pt"
        android:layout_marginRight="4pt"
        android:layout_marginTop="2pt"
        android:gravity="center_horizontal"
        android:textSize="10pt" >
    </TextView>
</LinearLayout>
------------------------------------------------------------------------
```

When we run the program, we will see a UI like the one shown in Figure 9-3 below.



TTS initialized successfully!
Click TTS to start speaking.

Figure 9-3 UI of *FileToSpeech* Project

9.6 Playing Audio Clips

9.6.1 Playing Sound

Android provides resources for managing audio and video media. The Android multimedia framework includes support for playing a variety of common media types.

Android supports various audio streams for different purposes. For example, a phone volume button can be configured to control the sound volume of a specific audio stream, increasing or decreasing the volume according to the need. We can also configure a button to control the sound media stream type in our application.

There are two main API's for audio playback, the *SoundPool* class for playing small audio clips, and the more commonly used *MediaPlayer* class for playing any kind of audio clips. We can use *SoundPool*, which loads files asynchronously, to play several sounds at the same time and repeat the play. However, The sound files to be played must not exceed 1 MB. The *OnLoadCompleteListener* can be used to check whether the loading of a file is complete.

We can use the *MediaPlayer* class, which also loads files asynchronously, to control playback of audio/video files and streams. The player is is managed as a state machine. Using this class, we can easily integrate audio, video and images into our applications. We can play video and/or audio from media files stored in our application's resources (raw resources), from standalone files in the filesystem, or from a data stream of a remote server through a network connection. Often we also need the *MediaController* class to control playback, and use a *Service* instance to play audio when the user is not interacting with the app directly. The *ContentResolver* class is often used to retrieve tracks on the device.

The *MediaPlayer* class may be the most important component of the Android media framework. The class is particularly easy to use for simple applications. A *MediaPlayer* object can fetch, decode, and play both audio and video with minimal setup. It supports several different media sources including

1. Local resources
2. Internal URIs, such as one that can be obtained from a *ContentResolver*
3. External URLs (streaming)

The details of this class and its usage can be found at the web sites,
http://developer.android.com/reference/android/media/MediaPlayer.html
and
http://developer.android.com/guide/topics/media/mediaplayer.html

9.6.2 Simple Examples of Using *MediaPlayer*

As a simple example, we show how to play a local audio file from the main activity (which is not desirable as we normally play an audio in the background). The following code is the complete program of *MainActivity.java* that plays an audio file located in the local raw resource directory *res/raw*:

```
package tts.audiodemo;

import android.os.*;
import android.app.Activity;
import android.media.MediaPlayer;

public class MainActivity extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    MediaPlayer mp = MediaPlayer.create(this, R.raw.sound1_wav);
    // no need to call prepare(); create() has done that already
    mp.start();
  }
}
```

The audio file name in this example is *sound1_wav*. The system does not try to parse a raw file in any particular way. Also, a raw filename cannot contain any upper case letter or a dot '.'. Thus a filename cannot have the two conventional parts, a primary part followed by an extension after a dot. However, the content of this resource should not be raw audio. It should be a properly formatted and encoded media file in one of the formats that Android supports. For clarity, we use the underscore '_' to replace the role of the dot. Thus the file *sound1_wav* actually means *sound1.wav*, which is encoded in the WAV format. When you run the program, you should hear the sound generated from the encoded data of the file.

The code of the above example is simple enough. We modify it to play a file on the Internet and show that *MediaPlayer* loads file asynchronously. In our next simple example, we show that the **start**() method of *MediaPlayer* returns immediately, not waiting for the playback to finish. In this example, the app flashes a message on the screen after it has started and another message after it has finished using the *Toast* class. The example will play a source file residing on a Web site. So we need to add the Internet access permission statement in the file *AndroidManifest.xml*:

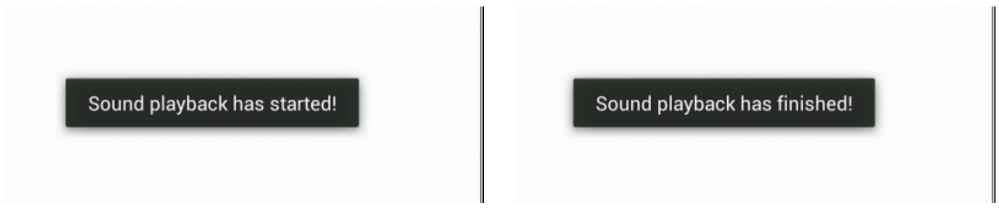
```
<?xml version="1.0" encoding="utf-8"?>
 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 .....
 <uses-permission android:name="android.permission.INTERNET"/>
 <application>
 .....
 </application>
</manifest>
```

Since we need to toast a message at the end, we need to know when the play completes. To accomplish this, we can add to the code a completion listener, which will call a method to perform the task when the player finishes playing. The following is the complete program (*MainActivity.java*) of this example:

```
package tts.audiodemo1;

import android.os.*;
import java.io.IOException;
import android.widget.Toast;
import android.app.Activity;
import android.media.MediaPlayer;
import android.media.AudioManager;
import android.media.MediaPlayer.OnCompletionListener;

public class MainActivity extends Activity
{
  MediaPlayer mp = null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mp = new MediaPlayer();
    String url = "http://www.forejune.com/android/files/sound1.wav";
    mp.setAudioStreamType(AudioManager.STREAM_MUSIC);
    try {
      mp.setDataSource(url);
      mp.prepare();    // might take long! (for buffering, etc)
    } catch (IllegalArgumentException | SecurityException
       | IllegalStateException | IOException e) {
      e.printStackTrace();
    }
    mp.start();
    mp.setOnCompletionListener(new
               OnCompletionListener() {
         @Override
         public void onCompletion(MediaPlayer mp) {
           onDone();
         }
       });
    if ( mp.isPlaying() )
      Toast.makeText(this, "Sound playback has started!",
                                        Toast.LENGTH_LONG).show();
  }

  protected void onDone()
  {
    Toast.makeText(this, "Sound playback has finished!",
                                        Toast.LENGTH_LONG).show();
    mp.release();
    mp = null;
  }
}
```

In the code, the complete listener is implemented as an anonymous inner class. The **setOnCompletionListener**() method register a callback to be invoked when the end of a media source has been reached during playback. In our example, when it finds the play is complete, it calls **onDone**(), which shows a *Toast* message, and calls **release**() to release the resources and then nullifies the *MediaPlayer*. It is important to always release the resources and not to hang to a *MediaPlayer* object that is no longer needed. This is because a *MediaPlayer* object can consume valuable system resources.

When we run the program, we should see a *Toast* message at the beginning of playing the sound (Figure 9-4 (a)), and another message at the end of the play (Figure 9-4 (b)).



(a) (b)

Figure 9-4 Toast Messages at Beginning and End of Play in Example

9.6.3 Asynchronous Playback of *MediaPlayer*

In most applications, we do not want to start a *MediaPlayer* in the main activity because most often we want the sound to be played in the background and leave the main activity to handle the UI for user interactions. Also, the call to *prepare*() may take a long time to execute as it might involve fetching and decoding media data. This may cause the UI to hang until the method returns, which may frustrate any user and may cause an ANR (Application Not Responding) error. We may manage and start a *MediaPlayer* object at the background with a new service as suggested by the official Android developers site. Here, we present an example that uses an alternative approach. We start the player using a worker thread running in the background, which is a traditional way of handling background tasks. Mutual exclusion locks and condition variables are used to achieve synchronization, sending the worker to sleep while the *MediaPlayer* is playing the sound and waking it up when the play has finished. The worker thread then calls the main thread, which handles the UI, informing it that the play is complete. The main thread displays a Toast message on the screen to notify the user.

We call this example project and application *AudioDemo2*, where the main activity class, *MainActivity*, handles only the UI, and the media player is managed by a class named *MyPlayer*. The UI in *MainActivity* defines four buttons, *Start*, *Pause*, *Stop* and *Resume*, to start, pause, stop and resume the media player respectively (see Figure 9-5). As we usually do, we define the UI layout in the file *res/layout/activity_main.xml*. The following is the complete listing of the code of *MainActivity*:

Program Listing 9-6 *MainActivity.java* of *AudioDemo2*

```
package tts.audiodemo2;

import android.os.*;
import android.widget.*;
import android.util.Log;
import android.view.View;
import android.app.Activity;

public class MainActivity extends Activity
{
  public  TextView textView;

  MyPlayer player = null;
  boolean started = false;   //player not started at beginning

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textView = (TextView) findViewById(R.id.status);
  }

  public  void playComplete ( View view )
  {
    showToast( "Play complete!");
    player = null;
    started = false;
  }
```

```
   public void onStart(View view) throws InterruptedException {
     if ( !started ){
       player = new MyPlayer ( this, view );
       Thread playerThread = new Thread ( player );
       playerThread.start();   // start player thread
       Toast.makeText(this,"Play started!",Toast.LENGTH_LONG).show();
       textView.setText("Play started!");
       started = true;
     }
   }

   public void pausePlay ( View view )
   {
     if ( player != null ) {
       if ( player.pausePlayer() )
         textView.setText("Player paused!");
     }
   }

   public void resumePlay ( View view )
   {
     if ( player != null ) {
       if ( player.resumePlayer() )
         textView.setText("Player resumed!");
     }
   }

   // stop player
   public void onStop ( View view )
   {
     if ( player != null ) {
       if ( player.stopPlayer() )
         textView.setText("Player stopped!");
     }
   }

   // display a Toast message on UI thread
   public void showToast(final String toast)
   {
     runOnUiThread(new Runnable() {
       public void run()
       {
          Toast.makeText(MainActivity.this, toast, i
                                   Toast.LENGTH_SHORT).show();
       }
     });
   }
 }
-------------------------------------------------------------------------
```

In the code, the method **playComplete**() will be called by the other class *MyPlayer*, when the media player detects that the playback is complete. It nullifies the current player (the zombie object will be handled by the Java garbage collector). To play the sound again, the user has to click the *Start* button again, which will create a new player object.

The method *Toast*.**makeText**() must be called from within the UI thread as do most methods that deal with the UI. If we call it from a worker thread, an exception error saying *Can't create handler inside thread that has not called Looper.prepare()* will occur. The function **showToast**() at the end of the code is to ensure that **makeText**() will be called from the UI thread. In this function, the method **runOnUiThread**(*Runnable action*) is a method of the class *Activity*. It runs the specified *action* on the UI thread.

The following listing shows the complete code of the other class, *MyPlayer*:

Program Listing 9-7 *MyPlayer.java* of *AudioDemo2*

```
package tts.audiodemo2;

 import java.io.*;
 import java.util.concurrent.locks.*;
 import android.util.Log;
 import android.view.View;
 import android.content.Intent;
 import android.media.MediaPlayer;
 import android.media.AudioManager;
 import android.media.MediaPlayer.OnCompletionListener;

 public class MyPlayer  implements Runnable
 {
   MediaPlayer mp = null;
   MyListener listener = null;
   final Lock mutex = new ReentrantLock();
   final Condition done = mutex.newCondition();
   Intent intent = null;   // not used in this demo
   boolean complete = false;
   MainActivity main;
   View view;         // not really needed

   public MyPlayer (){
      //super ("MyPlayer");
   }

   public MyPlayer ( MainActivity main0, View view0 ){
      main = main0;
      view = view0;
      listener = new MyListener();
   }

   public MyPlayer ( Intent i ) {
     //super ("MyPlayer");
     listener = new MyListener();
     intent = i;
   }

   @Override
   public void run() {
     onHandleIntent ( intent );
     main.playComplete( view );
   }
```

```
  //@Override ( need Override if started as an IntentSerive )
  protected void onHandleIntent(Intent intent) {
    mp = new MediaPlayer();
    String url = "http://www.forejune.com/android/files/sound1.wav";
    mp.setAudioStreamType(AudioManager.STREAM_MUSIC);
    try {
      mp.setDataSource(url);
      mp.prepare();    // might take long! (for buffering, etc)
    } catch (IllegalArgumentException | SecurityException
          | IllegalStateException | IOException e) {
      e.printStackTrace();
    }

    mp.start();
    mp.setOnCompletionListener ( listener );
    try {
      mutex.lock();
      while ( !complete )
        done.await();           //Condition wait
      mutex.unlock();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  // To detect when is the play complete
  class MyListener implements OnCompletionListener
  {
    @Override
    public void onCompletion ( MediaPlayer mp ) {
      complete = true;
      mutex.lock();
      done.signal();    //wake up sleeping thread
      mutex.unlock();
      mp.release();     //release resources
      mp = null;
    }
 }

  // stop the media player
  public boolean stopPlayer()
  {
    if( mp !=null && mp.isPlaying() ){
          mp.stop();
          return true;
    } else
      return false;
  }

  // pause the media player, which can be resumed by calling start()
  public boolean pausePlayer()
  {
    if( mp !=null && mp.isPlaying() ){
            mp.pause();
            return true;
    } else
      return false;
  }

  // resume the paused media player
  public boolean resumePlayer()
  {
    if( mp !=null && !mp.isPlaying() ){
            mp.start();
            return true;
    } else
      return false;
  }
}
--------------------------------------------------------------------------
```

After the worker thread has started to run, the method **onHandleIntent** is called. This method creates and prepares a *MediaPlayer*, which starts playing the sound from the hard-coded Web address in the example. It also sets the listener *OnCompletionListener*, which is implemented inside the class. It then waits on the condition variable *done* and goes to sleep. When the playback of the sound is complete, the method **onCompletion** of the listener is called; this method nullifies the *MediaPlayer* and releases its resources. It also wakes up the sleeping thread, which will return from **onHandleIntent** to **run**, which then calls **playComplete**() of *MainActivity* that will present messages about the completion to the user.

With minor modifications, the thread can also be started as an *IntentService* and the method, **onHandleIntent**() will be an *Override* and called asynchronously.

The variable *intent* is not used here. It is left in the code for the convenience of any reader who wants to make modifications to the code.

The xml code listed below is of the file *res/layout/activity_main.xml*, which defines the layout of the UI.

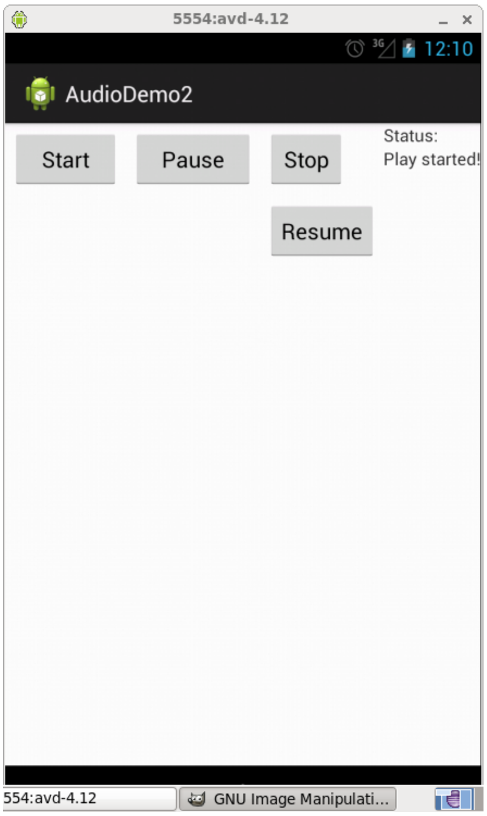
Program Listing 9-8 *activity_main.xml* of *AudioDemo2*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal" >

   <Button
       android:id="@+id/onstart"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_margin="5dip"
       android:layout_weight="1"
       android:onClick="onStart"
       android:text="Start" />
  <Button
       android:id="@+id/pauseplay"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_margin="5dip"
       android:layout_weight="1"
```

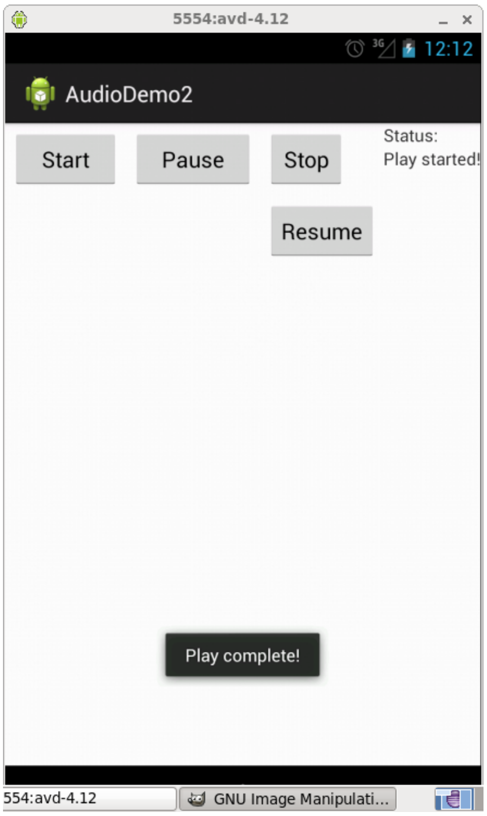
```
        android:onClick="pausePlay"
        android:text="Pause" />
   <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical" >
     <Button
        android:id="@+id/stop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="5dip"
        android:layout_weight="1"
        android:onClick="onStop"
        android:text="Stop" />

     <Button
        android:id="@+id/resume"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="5dip"
        android:onClick="resumePlay"
        android:text="Resume" />
   </LinearLayout>
   <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical" >

     <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Status: " />
     <TextView
        android:id="@+id/status"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Not started" />
   </LinearLayout>
 </LinearLayout>
------------------------------------------------------------------------
```

Since the source file is on the Internet. We have to set the permission in the *AndroidManifest.xml*:
<uses-permission android:name="android.permission.INTERNET"/>

Figure 9-5 shows the UI of this application when we run it. Figure 9-5(a) shows the buttons and the text layout at the beginning. The *Status* message shows "*Play started!*" implying that the button *Start* has been clicked and the audio is played in the background. Figure 9-5(b) shows the Toast message when the playback is complete.



(a)



(b)

Figure 9-5 UI of *Audiodemo2*