NUANCE PROFESSIONAL SERVICES I Version 1.0 I December 20, 2007

UAPI User Guide

for SREC RC-1 for Android

DOCUMENT HISTORY

Date	Revised by	Version	Summary of Changes
12/20/2007	K. Evdokimov Andy Wyatt	1.0	For delivery with SREC RC-1 for Android.

TABLE OF CONTENTS

1 AUDIENCE

This document is intended for application developers writing speech enabled applications using UAPI SREC recognizer on Android Platform. An understanding of the Java programming language and the core Java APIs is assumed.

2 REFERENCES

1. SREC User Guide, Version 1.0, December 20, 2007

3 GLOSSARY

UAPI

Unified API

JNI

Java Native Interface

4 TABLES AND FIGURES

4.1 Tables

4.2 Figures

5 OVERVIEW

The Unified API (UAPI) is a common interface by which mobile applications may access device-resident and network-resident media resources. The UAPI has been designed to allow applications to access both in a transparent manner and to adjust usage as network connectivity warrants.

The term “unified” in “Unified API” refers to the unification of embedded and network speech technologies under the same API.

This document introduces the UAPI interface to SREC embedded speech recognition engine on Android Platform. The UAPI is implemented in the `android.speech.recognition` Java API package.

The UAPI User Guide is a programmer’s guide to developing speech applications using the `android.speech.recognition` package on Android Platform. A comprehensive API reference (javadoc) for `android.speech.recognition` is available.

In this guide the terms UAPI, SREC and Recognizer are used interchangeably.

5.1 UAPI highlights

- Java API
- Multithreaded
- Asynchronous
- Platform independent
- Implementation independent
- Language independent
- Unified interface for network and embedded (local) recognizers.

5.2 SREC Implementation capabilities

- SREC is a *continuous speech* recognizer. This means that the speaker doesn’t have to pause between the words when giving complex commands.
- SREC is a *speaker independent* recognizer.
- SREC is a *phoneme-based* recognizer. This allows any word to be recognized without previous training. Despite its phoneme-based nature, SREC also uses some whole word models to maximize accuracy for specific categories of words such as digits.

- SREC is a *constrained speech* (grammar based) recognizer

- SREC supports *dynamic word addition* to the grammar. Online Grapheme-to-Phoneme is supported. This allows the application to add new words to the grammar and perform the conversion from standard orthography to the appropriate phonetic dictionary.

- SREC supports *voice enrollment*. SREC can learn new words “on the fly” from a given speaker. This means that one SREC-based application can train online, store and later recognize user specific words (also known as voice tags or speaker-dependent words). Training requires only one utterance of the user word (more than one is possible).

- SREC supports a *simple semantic interpretation language* that allows grammar developers to associate grammar-specific orthographies and/or synonyms to application actions.

- US English *language support* only (in this release)

- Push-to-talk (no support for *echo cancellation*)

- *Prompting* (Speaker interface) not supported

- *End point detection*

- *Native implementation* (JNI)

5.3 System Requirements

The application and recognition engine will reside on the Android platform, whose *minimum* specifications are detailed in the table below:

Platform Name	Google development platform “sooner”
Processor model	TI OMAP, Qualcomm or similar with ARM9 or ARM11 core.
Processor clock	190 MHz minimum
RAM	Minimum: 32 MB SDRAM, 32 MB Flash
Audio Input	16 bit, PCM format, 8 kHz
Operating System	Google Open Handset Distribution, now known as “Android”, based on Linux OS and Java 2 SE.
Debugger	Tools part of Ubuntu Linux 6.06 and Mac OS X.
C Compiler	Tools part of Ubuntu Linux 6.06 and Mac OS X.

Table 1 Minimum System Requirements

6 RECOGNIZER: android.speech.recognition

The `javax.speech.recognition` package defines the `EmbeddedRecognizer` interface and well as a set of supporting classes and interfaces.

The functionality of the Unified API is grouped into 5 modules:

- `EmbeddedRecognizer`
- `Microphone`
- `DeviceSpeaker`
- `MediaFileReader`
- `MediaFileWriter`

A typical application would only use a Recognizer and a Microphone. The other modules are available for convenience and testability.

This section begins with a simple code example, and then reviews the capabilities of the UAPI in more detail through the following sub-sections.

6.1 Hello World application

The following example shows a simple application that uses SREC recognizer. In this example, we define a grammar that allows a user to say either “yes” or “no”. The grammar is defined using the W3C Speech Recognition Grammar Specification Version 1 grammar format. This format is documented by the W3C grammar specification at http://www.w3.org/TR/grammar-spec. Please see “UAPI User Guide” for specific details on SREC grammar format.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<grammar version="1.0" xml:lang="en-US" mode="voice" root="_boolean">

 <rule id="_boolean" scope="public">
    <one-of>
      <item>yes <tag>MEANING='1'</tag></item>
      <item>no <tag>MEANING='0'</tag></item>
      <item> <ruleref uri="#ROOT"/> </item>
    </one-of>
 </rule>

<rule id="ROOT" scope="public">
<item>__ROOT__</item>
</rule>

</grammar>
```

The following code shows how to obtain a recognizer, create and load the grammar, and then to process microphone based speech using the grammar. After the applications processes the audio input, it performs some cleanup and exits.

```
/*-----------------------------------------------------------------------------*
 *   HelloWorld.java                                                           *
 /*----------------------------------------------------------------------------*
```

```
 *  HelloWorld.java                                                           *
 *                                                                            *
 *  Copyright 2007 Nuance Communciations, Inc.                                *
 *                                                                            *
 *  Licensed under the Apache License, Version 2.0 (the 'License');           *
 *  you may not use this file except in compliance with the License.          *
 *                                                                            *
 *  You may obtain a copy of the License at                                   *
 *      http://www.apache.org/licenses/LICENSE-2.0                            *
 *                                                                            *
 *  Unless required by applicable law or agreed to in writing, software       *
 *  distributed under the License is distributed on an 'AS IS' BASIS,         *
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  *
 *  See the License for the specific language governing permissions and       *
 *  limitations under the License.                                            *
 *                                                                            *
 *---------------------------------------------------------------------------*/

package android.speech.recognition.examples;

import java.util.Hashtable;

import android.speech.recognition.EmbeddedRecognizer;
import android.speech.recognition.AbstractRecognizerListener;
import android.speech.recognition.RecognizerListener;

import android.speech.recognition.SrecGrammar;
import android.speech.recognition.G2GConfiguration;
import android.speech.recognition.SrecGrammarListener;
import android.speech.recognition.AbstractSrecGrammarListener;
import android.speech.recognition.GrammarListener;

import android.speech.recognition.Microphone;
import android.speech.recognition.AudioSource;
import android.speech.recognition.AudioStream;
import android.speech.recognition.Codec;

import android.speech.recognition.NBestRecognitionResult;
import android.speech.recognition.NBestRecognitionResult.Entry;


/**
 * Hello World application for UAPI Users Guide.
 *
 * @author kman
 */

public class HelloWorld extends AbstractRecognizerListener
  implements RecognizerListener
{
  static final String ESRSDK =
    (System.getenv("ESRSDK") != null) ? System.getenv("ESRSDK") :
      "/system/usr/srec";

  EmbeddedRecognizer rec;
  SrecGrammar grammar;

  boolean bDone = false;


  public static void main(String[] args)
```

```
  {
    try {
      HelloWorld helloWorld = new HelloWorld();

      helloWorld.recognize();

    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }


  public void recognize()
  {

    try {

      //obtain the recognizer
      rec = EmbeddedRecognizer.getInstance();

      //configure the recognizer
      String recConfig = ESRSDK + "/config/en.us/baseline11k.par";
      System.out.println("configuring recognizer with " + recConfig);
      rec.configure(recConfig);

      //set listener
      rec.setListener(this);

      //create grammar
      String grammarPath = ESRSDK + "/config/en.us/grammars/boolean.g2g";
      System.out.println("creating grammar " + grammarPath);
      grammar =
       (SrecGrammar) rec.createGrammar(grammarPath,
                                  new HelloWorldGrammarConfig());

      //kick-off grammar load
      System.out.println("loading grammar");
      grammar.load();

      synchronized (this) {
       while (!bDone) {
         wait();
       }
      }

    }
    catch (Exception e) {
      e.printStackTrace();
    }

    System.out.println("good bye.");
  }



  private class HelloWorldGrammarListener extends AbstractSrecGrammarListener
    implements SrecGrammarListener
  {
    @Override
    public void onError(Exception e)
```

```
    {
      System.out.println("Error during grammar load: " + e.toString());
    }

    @Override
    public void onLoaded()
    {
      System.out.println("grammar loaded");

      System.out.println("setting up microphone for 16bit 11KHz");
      Microphone mic = Microphone.getInstance();
      mic.setCodec(Codec.PCM_16BIT_11K);
      AudioStream audioStream = mic.createAudio();
      mic.start();

      System.out.println("Please say \"yes\" or \"no\"");

      rec.recognize(audioStream, grammar);
    }
  }


  private class HelloWorldGrammarConfig implements G2GConfiguration
  {
    SrecGrammarListener grammarListener = new HelloWorldGrammarListener();

    public GrammarListener getListener()
    {
      return grammarListener;
    }

    public Object grammarToMeaning(String semanticMeaning,
                               Hashtable<String, String> parameters)
    {
      return semanticMeaning;
    }
  }


  /* RecognizerListener overrides */
  @Override
  public void onRecognitionFailure(RecognizerListener.FailureReason reason)
  {
    System.out.println("recognition failed: " + reason.toString());
  }

  @Override
  public void onRecognitionSuccess(NBestRecognitionResult result)
  {
    int numResults = result.getSize();
    System.out.println("RECOGNITION SUCCESS: got " + numResults + " results.");

    NBestRecognitionResult.Entry entry;
    for (int i = 0; i < numResults; i++)
    {
      entry = result.getEntry(i);
      System.out.println("result "  + (i + 1) + ": '"  + entry.getLiteralMeaning() +
"'");
    }
  }
```

```
  @Override
  public void onStopped()
  {
    System.out.println("done recognizing");

    Microphone.getInstance().stop();

    synchronized (this) {
      bDone = true;
      notifyAll();
    }
  }

}
```

This example illustrates the basic steps which UAPI interface should be used:

- Instantiate the Recognizer: `EmbeddedRrcognizer.getInstance()` is used to obtain an instance of the Recognizer.
- Configure Recognizer: `configure()` call is used to configure recognizer.
- Create and load grammars: `EmbeddedRecongizer.createGrammar()` and `SrecGrammar.load()` are used to create the grammar object and load the grammar from file.
- Attach the recognizer listener to the recognizer
- Create `AudioStream` and start `AudioSource`
- Start recognition
- Listen for and process results
- Stop `AudioSource`
- Cleanup

6.2 Instantiation of the Recognizer

EmbeddedRecognizer instantiation follows the singleton design pattern. There is only one instance of EmbeddedRecognizer per application process. This single instance of the recognizer is obtained via a call to `EmbeddedRecognizer.getInstance()`.

6.3 Configuration of the Recognizer

Before the recognizer is used to process speech it needs to be configured using `EmbeddedRecognizer.configure()`method:

```
public abstract void configure(java.lang.String config)
                        throws java.lang.IllegalArgumentException,
```

```
                                      java.io.FileNotFoundException,
                                      java.io.IOException,
                                      java.lang.UnsatisfiedLinkError,
                                      java.lang.ClassNotFoundException
```

`EmbeddedRecognizer` configuration is the process during which the system resources necessary for operation of the recognizer are obtained. `EmbeddedRecognizer` is not automatically configured at the system start-up time because it requires a substantial CPU and memory resources. Also the process of configuration is a relatively slow procedure, when compared to typical recognizer response times.

6.3.1 Configuration State System

The behavior of an `EmbeddedRecognizer` with respect to configuration can be described by the state system in Figure 1. Each state defines a particular mode of operation of an `EmbeddedRecognizer`. The `EmbeddedRecognizer` behaves differently depending on its current state. UAPI does not provide for an explicit way to query or modify the state of `EmbeddedRecognizer`. The `EmbeddedRecognizer` state system detailed below is meant as an aid to application developer, to illustrate applicability and behavior of different methods as different times.

In some cases applications can monitor the `EmbeddedRecognizer` state through the event/listener system, e.g. by using `EmbeddedRecognzierListener.onStopped()`. However, there are no events associated with the completion of the `configure()` operation.

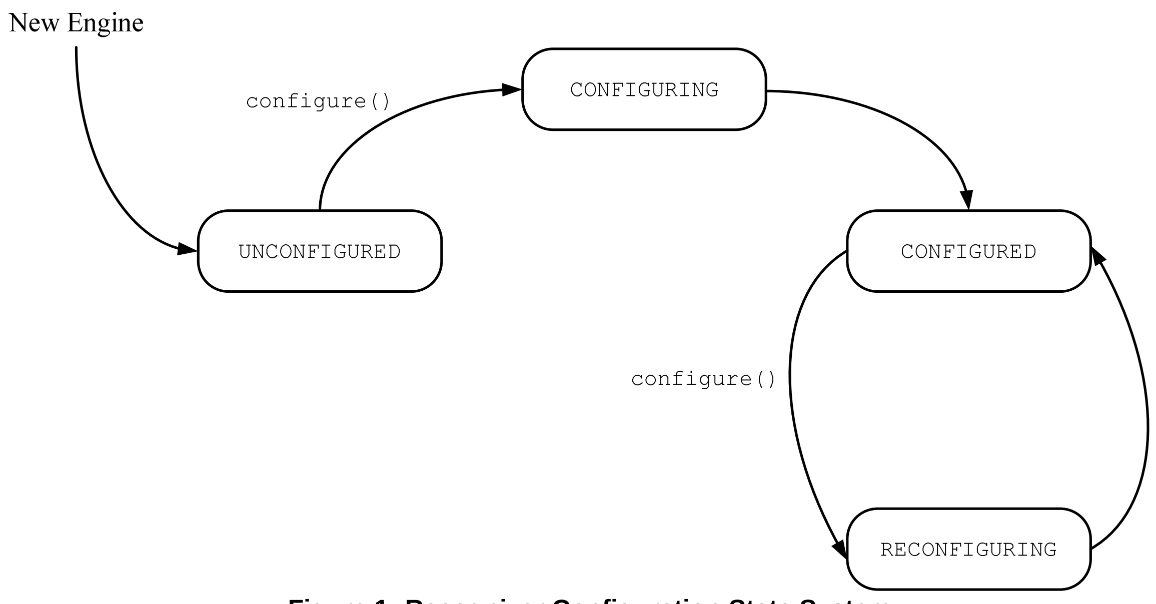

Figure 1: Recognizer Configuration State System

Each block represents a state of the `EmbeddedRecognizer`. The `EmbeddedRecognizer` is always in one of the four specified states. There are no events issued as the recognizer transitions between these states. The current configuration state of the recognizer cannot be queried.

The normal operational state of the `EmbeddedRecognizer` is `CONFIGURED`. While in the `CONFIGURED` state, the `EmbeddedRecognizer` will be either `BUSY` or `IDLE`. The `BUSY-IDLE` state subsystem of the recognizer's `CONFIGURED` state is described in section 6.4.

It is important to note that once the recognizer is configured, it cannot return to the `UNCONFIGURED` state. There is no way for the application developer to cause the recognizer to free the resources obtained during the initial configuration process.

The recognizer can be reconfigured. The reconfiguration resets all recognizer state, parameters to the defaults specified by the `config` parameter of `configure()` method, which transitions the recognizer into the `RECONFGURING` state. If no errors occur during reconfiguring, the recognizer then returns to the `CONFIGURED` state, just as it had after being configured initially.

The reconfiguration process invalidates any grammars that have been created. The grammar objects created in the context of the previous configurations should not be used in the context of the new configuration. The old grammar objects should be dereferenced and the new grammar object should be created in the context of the current recognizer configuration. Any operations on invalid (old) grammar object will fail.

6.3.2 Configuration is Synchronous

Unlike most of the `android.speech.recognition` API methods, the `configure()` method is synchronous (blocking). For advanced applications, it is often desirable to start up the configuration of a recognizer in a background thread while other parts of the application are being initialized. For GUI applications, it is often necessary to process user interface related events concurrently with configuration of the recognizer. This can be achieved by calling the `configure()`method in a separate thread. The following code shows an example of this using an inner class implementation of the `Runnable` interface, as in the following example.

```
new Thread(new Runnable() {
public void run() {
try {
…configure()
}
catch (Exception e) {
e.printStackTrace();
}
}
}).start();

// Do other stuff while allocation takes place
...
// Now wait until configuration is completed
//
```

A `configure() call during CONFIGURING or RECONFIGURING states will block until the previous configuration is completed and then run normally.`

A `configure()` call will fail with if the recognizer is not in the `IDLE` state.

6.4 Busy-Idle State System

An `EmbeddedRecognizer` in `CONFIGURED` state has `BUSY` and `IDLE` sub-states. Once an `EmbeddedRecognizer` reaches `CONFIGURED` state, it also enters the `IDLE` state.

The `IDLE/BUSY` state indicates whether the recognizer is busy performing a tack such as processing input audio or performing a grammar related operation.

Figure 2 shows the `BUSY-IDLE` state sub-system of an `EmbeddedRecognizer`.

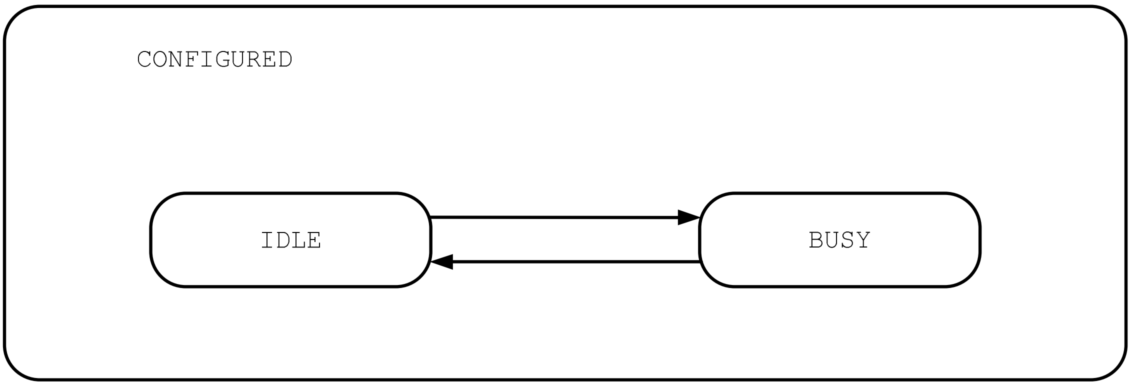

Figure 2: IDLE and BUSY Recognizer states

The following methods represent the requests to the recognizer and cause the transition from IDLE to BUSY state:

`EmbeddedRecognizer.resetAcousticState`

`EmbeddedRecognizer.recognize`

`EmbeddedRecognzier.getParameters`

`EmbeddedRecognizer.setParameters`

`SrecGrammar.addItemList`

`SrecGrammar.compileAllSlotsve`

`SrecGrammar.resetAllSlots`

`SrecGrammar.save`

`SrecGrammar.load`

`SrecGrammar.unload`

The `EmbeddedRecognizer` interface does not provide any explicit methods to test or monitor the `IDLE/BUSY` state directly. Instead, the application developer should monitor recognizer and grammar related events to determine when the current operation is completed and the recognizer returns to the IDLE state.

The implicit `IDLE/BUSY` state sub-system determines how the recognizer and grammar operations are handled.

Calling the methods from the above list while the recognizer is in the `BUSY` state is illegal. The `UAPI` contact does not guarantee the behaviour or the order of execution of the methods listed above if called while the recognizer is in the `BUSY` state. Application developer must wait for the current asynchronous operation to complete and the recognizer enter the `IDLE` state before invoking the next call on the recognizer.

One exception to the rule above is the `EmbeddedRecognizer.stop()` method which is intended to be called while the recognizer is in the BUSY state.

6.5 Recognizer Events

During recognition process the recognizer generates several events. The typical sequence of events during recognition is as follows:

- **`onStarted`** – indicates the start of the recognition. This is the first event during the recognition. It is always issued.
- **`onBeginningOfSpeech` –** begin of speech detected. Issued only if the recognizer detects the beginning of speech. May not occur.
- **`onEndOfSpeech` –** end of speech detected. Issued only if the recognizer detects the end of speech in the input audio stream. May not occur.
- **`onRecognitionSuccess` or `onRecognitionFailure` –** one and only one of these is guaranteed to occur.
- **`onStopped` –** indicated the end of the recognition. This is the last event during the recognition sequence of events. It is guaranteed to occur.

6.6 Recognition Results

Recognition results are provided by the recognizer to an application when the recognizer processes the incoming speech that matches the current recognition grammar. The recognition results provide an application with information about what the speaker said. The recognizer may not be correct about what the speaker said every time. Never the less, this situation of *misrecognition* is still referred to as *recognition success*. The situation when the recognizer is unable to make any guesses as to what the speaker said is called *recognition failure*. Recognition failure should not be confused with unexpected API errors, exceptions and failures. Recognition failure is an expected outcome of speech recognition.

The recognizer notifies an application of the recognition results by the `onRecognitionSuccess` and `onRecognitionFailure` events issued to the `RecognizerListener`.

6.7 Audio Management

The input audio for the recognizer is specified by means of `AudioSream` objects. Each `AudioStream` object represents a sequence of audio samples associated with an audio source. The `AudioStream` interface does not have any public members. The application writer does not manipulate the audio data directly. Audio manipulation is performed by the underlying implementation.

The audio data originates from `AudioSource` objects. There are two types of `AudioSource` objects currently available: `Microphone` and `MediaFileReader`. `Microphone` represents the microphone on the Android platform. The `MediaFileReader` objects allow application developer to work with audio data stored in files.

Each AudioStream object is associated with one and only one AudioSource. There may be multiple AudioStream objects associated with a single AudioSource object.

From the logical perspective, the application developer may view AudioStream objects as containers for audio data. The AudioSource object associated with the AudioStream object writes audio data to the AudioStream object and the EmbeddedRecognizer and MediaFileWriter objects read audio data from the AudioStream.

6.7.1 Microphone

Microphone instantiation follows the singleton design pattern. There is only one instance of Microphone per application process. This single instance of the microphone is obtained via a call to Microphone.getInstance().

The behavior of the Microphone can be described by the state system in Figure 3: Microphone State System.

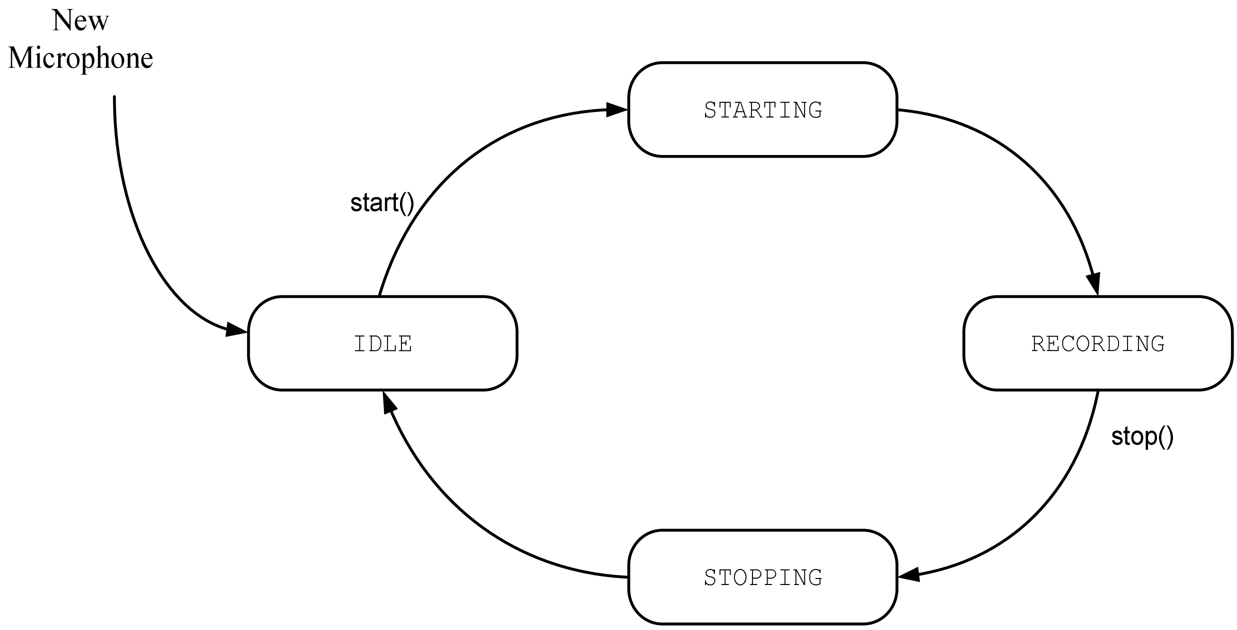

Figure 3: Microphone State System

In the IDLE state the Microphone does nothing. While in the RECORDING state the Microphone sends audio data from the underlying microphone system device into all AudioStream objects associated with the Microphone.

Microphone.start() method transitions the Microphone from the IDLE state to the STARTING state. Microphone.stop() method transitions the Microphone from the RECODRING state to the STOPPING state.

When the Microphone completes the IDLE-STARTING-RECORDING state transition, the Microphone issues onStarted() event to the associated MicrophoneListener. The RECORDING-STOPPING-IDLE state transition is indicated by the onStopped() event. Unexpected errors during the state machine traversal are indicated by the onError() event sent to the MicrophoneListener.

6.7.2 MediaFileReader

`MediaFilerReader` supports reading audio from a file. Speech application developers will typically use this interface for off-line debugging and testing.

Below is the section from `HelloWorld` application modified to use audio stored in a file rather than microphone based audio.

```
String audioPath = ESRSDK + "/config/en.us/audio/v139/v139_113.nwv";
int headerLength = 1024;
MediaFileReader mediaFileReader =
  MediaFileReader.create(audioPath, headerLength, Codec.PCM_16BIT_11K,
                     null);

//make MediaFileReader file look more like microphone
mediaFileReader.setMode(MediaFileReader.Mode.REAL_TIME);
AudioStream audioStream = mediaFileReader.createAudio();
mediaFileReader.start();

rec.recognize(audioStream, grammar);
```

6.7.3 MediaFileWriter

MediaFileWriter interface allows programmers to save AudioStream audio data into a file. Application developers will typically use this interface for saving recognized audio input for offline processing and debugging. For more information please see UAPI Reference on `android.speech.recognition.MediaFileWriter`.

6.7.4 DeviceSpeaker

DeviceSpeaker is an interface for audio output. It can be used to implement audio prompts during speech recognition application. See UAPI reference on `android.speech.recognition.DeviceSpeaker` for more details. This interface is not implemented on Android Platform with this release.

6.8 Error Handling

This section describes how the errors from UAPI methods are reported to the applciation. The terms *error* and *failure* in this section refer to the unexpected errors (exceptions) arising from execution of various methods of UAPI. These API function errors and failures should not be confused with the *recognition failures* which are also sometimes referred to as *recognition errors*. The recognition failure (or recognition error) is an **expected** outcome of speech recognition process.

UAPI features both synchronous and asynchronous functionality. Synchronous methods of UAPI follow a typical Java design pattern under which a method either succeeds or throws an exception.

6.8.1 Recognizer Errors

The outcomes of asynchronous grammar operation are indicated by events delivered to the `GrammarListener` specified by the `GrammarConfiguration`. The table below summarizes the asynchronous grammar operations and the corresponding asynchronous outcomes.

Grammar method	success	failure
`addItemList`	`onAddItemList`	`onAddItemListFailure`
`compileAllSlots`	`onCompileAllSlots`	`onError`
`resetAllSlots`	`onResetAllSlots`	`onError`
`save`	`onSaved`	`onEror`
`load`	`onLoaded`	`onError`
`unload`	`onUnloaded`	`onError`

Table 2: Asynchronous Grammar Methods and Events

The outcomes of asynchronous recognizer operations are indicated by events delivered to the `RecognizerListener` specified by the EmbeddedRecognizer.setListener(…) method. The table below summarizes the asynchronous recognizer operations and the corresponding asynchronous outcomes.

Recognizer method	success	failure
`resetAcousticState`	`onAcousticStateReset`	`onError`
`getParameters`	`onParametersGet`	`onParametersGetError`
`setParameters`	`onParametersSet`	`onParametersSetError`
`recognize`	`onStopped`	`onError`

Table 3: Asynchronous Recognizer Methods and Events

6.9 Other Recognizer Functionality

6.9.1 Acoustic Adaptation

The recognizer may adapt to the speaker and speaker environment in order to improve recognition accuracy. This adaptation process relies on the recognizer preserving some amount of acoustic state information between the recognitions.

In order to achieve the best accuracy, the application developer should clear the accumulated acoustic state information when the speaker or the speaker environment is known to have changed.

It is also important to explicitly control recognizer adaptation during automated accuracy tests in order to obtain consistent accuracy results.

The acoustic state information is cleared during configuration or reconfiguration process as the recognizer transitions through the `CONFIGURING` or `RECONFIGURING` states.

The application developer can control the acoustic state information of the recognizer by calling `EmbeddedRecognizer.resetAcousticState()` method. Upon successful completion of the call, the recognizer acoustic state information is reset and the recognizer is returned to the same state acoustic adaptation state it was in right after it entered `CONFIGURED` state but before any calls to `recognize(…).`

The `EmbeddedRecognizer.resetAcousticState()` is an asynchronous (non-blocking) method. A successful completion is indicated via `RecognizerListener.onAcousticStateReset()` event. An error is indicated by `RecognizerListener.onError(…)` event.

6.9.2 Recognizer Parameters

The recognizer parameters that can be changed dynamically are listed in Table 4. For a complete list of recognizer parameters, please refer to the SREC User Guide.

Parameter Name	Description	Typical Values	Min	Max
`CREC.Frontend.swicms.cmn`	Channel normalization values in string form. These values have no logical value to the application. They should normally only be set after having been get previously.			
`SREC.Recognizer.utterance_timeout`	maximum number of frames to wait for declaring start of speech (ms)	400		
`CREC.Frontend.samplerate`	Sample rate of the audio data (samples per second); this is an indication on the input audio such that audio can be a frequency higher than the minimum required by the acoustic model (high_cut), in such a case some high frequency content is ignored	8000, 11025, 16000, 22050	8000	22050
`CREC.Recognizer.terminal_timeout`	Default end of utterance timeout when the search is at the end of the grammar (Number of 20ms frames, ie see do_skip_even_frames).	20		
`CREC.Recognizer.optional_terminal_timeout`	End of utterance timeout when the search is optionally at the end of the grammar, eg. after any digit in an unconstrained digit recognition (Number of 20ms frames, ie see do_skip_even_frames).	40		
`CREC.Recognizer.non_terminal_timeout`	End of utterance timeout for words that do not occur at the end of the utterance. (Number of 20ms frames,	200		

	ie see do_skip_even_frames).			
`CREC.Recognizer.eou_threshold`	Score delta, by which this search state needs to be best before starting to count frames for timeouts below.	150		
`enableGetWaveform`	Used only during voice enrollment process. If set to "1" before the recognition, the voicetag produced during the recognition will also contain the audio data produced during the recognition process. The user can retrieve the audio data with the `VoicetagItem.getAudio()` function or it can be saved to a file at the same time the voicetag item is saved using the VoicetagItem.save(…) function. The audio is in PCM format and is start-pointed and end-pointed.	"0" "1"		

Table 4: Dynamic Recognizer Parameters

The parameters summarized above can be manipulated during runtime via `EmbeddedRecognizer.setParameters()` and `EmbeddedRecognizer.getParametres()`.

7 SREC GRAMMARS

7.1 Editing grammars

SREC grammars are defined in the W3C XML format and possibly extended at run-time through dynamic word addition and for a different tag interpretation language. For details of the grammar formalism, developers should refer the to W3C grammar specification at http://www.w3.org/TR/grammar-spec, with the following exceptions:

- support for <item repeat="$N" … $N can any number
- support for <item repeat="$N-" … $N can any number
- support for <item repeat="$N-$M" … but $M>$N

The important parameters that are looked for in the grammar are near the top of the file:

```
<?xml version="1.0" encoding="ISO8859-1"?>
<grammar xml:lang="en-US" version="1.0" mode="speech" root="myRoot">
```

xml:lang … indicates the language of the grammar, the specified language will trigger use of the right dictionaries and acoustic models to compile the grammar. The engine supports an extensive but limited set of languages. Language encoding conventions are detailed in the Phonology chapter.

encoding … for European language in which accents must be used, the use of ISO Latin-1 encoding is supported

7.2 Compiling grammars

Grammars must always be compiled off-line on desktop Linux. The command line instructions are as follows:

```
(1) % grxmlcompile -par /device/extlibs/srec/config/en.us/baseline.par -grxml
test.grxml
(2) % make_g2g –base test –out test.g2g
```

In Step 1, we create AT&T text format fsms (http://www.research.att.com/~fsmtools/fsm/man4/fsm.5.html). The required files are:

- .map … the list of words
- .PCLG.txt … the finite-state transducer to be used for the search
- .Grev2.det.txt … the transducer to be used for nbest processing
- .P.txt … the semantic interpretation graph
- .script … the semantic interpretation scripts

These text files should not be edited; they are dumped for diagnostic purposes only.

In Step 2, we package these 5 files into a single binary format file to be used on the target platform.

7.3 Dynamic Grammar modification at runtime

The SREC recognizer supports the ability for an application to modify grammars at runtime. There are two ways to modify the grammars as runtime: dynamic word addition and voice enrolment.

SREC supports *dynamic word addition* to the grammar. Online Grapheme-to-Phoneme is supported. This allows the application to add new words to the grammar and perform the conversion from standard orthography to the appropriate phonetic dictionary.

SREC supports *voice enrollment*. SREC can learn new words “on the fly” from a given speaker. This means that one SREC-based application can train online, store and later recognize user specific words (also known as voice tags or speaker-dependent words). Training requires only one utterance of the user word (more than one is possible).

7.4 Additional Grammar Concepts

For additional information regarding grammars, please refer to the SREC User Guide.