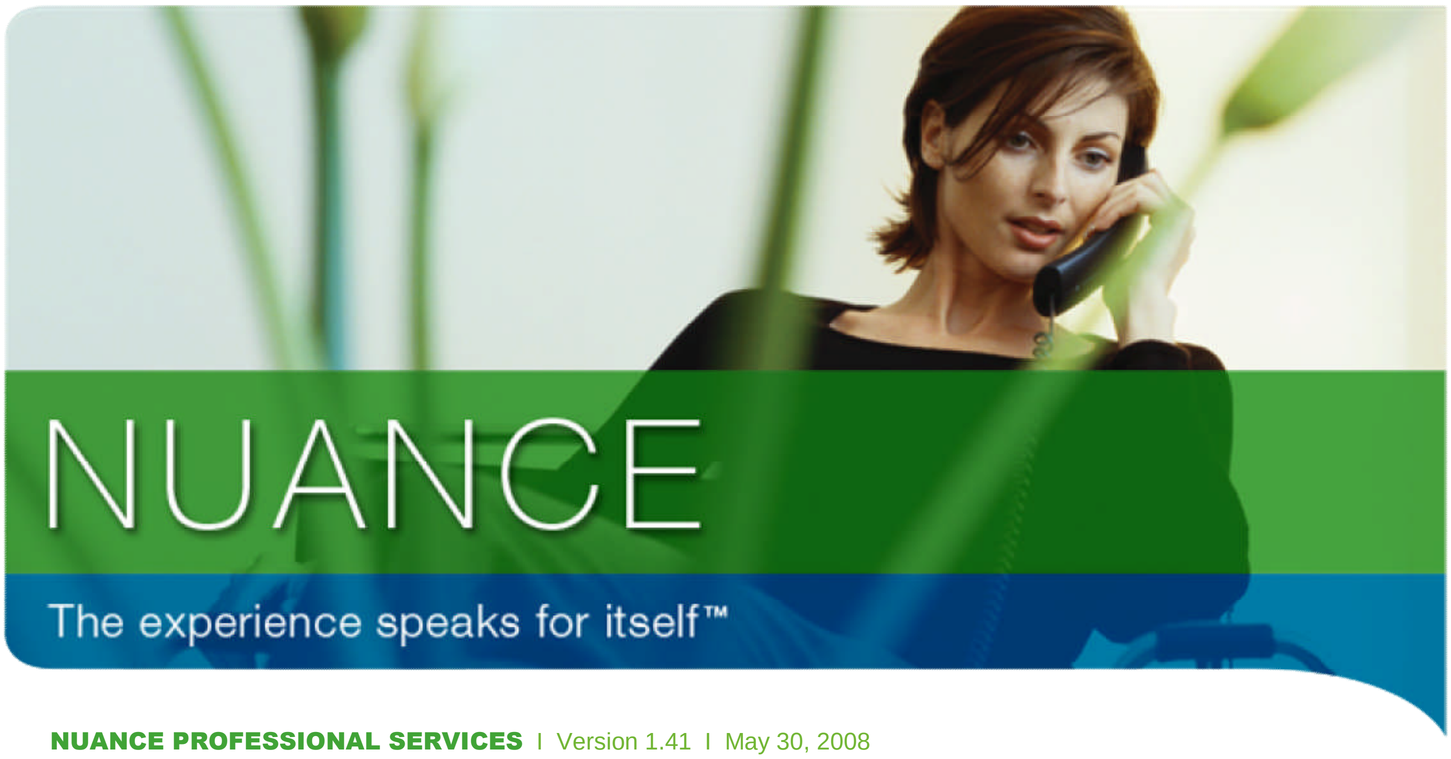

NUANCE PROFESSIONAL SERVICES I Version 1.41 I May 30, 2008

# SREC User Guide
for Android

NUANCE

## DOCUMENT HISTORY

| Date | Revised by | Version | Summary of Changes |
|---|---|---|---|
| 12/18/2007 | Jean Dahan, Dennis Velasco, Andy Wyatt | 1.0 | For delivery with SREC RC-1 for Android. |
| 01/11/2008 | Jean Dahan | 1.1 | Added description of dictionary lookup |
| 02/12/2008 | Rabih Majzoub | 1.2 | Added dynamic word addition documentation |
| 03/07/2008 | Jean Dahan Dennis Velasco | 1.3 | Updated for dynamic slot allocation, replaced NR_ with SR_ functions.<br>Corrected typographical errors in TCP section. |
| 03/14/2008 | Jean Dahan | 1.4 | Added min/max information for parameters |
| 05/30/2008 | Jean Dahan | 1.41 | Added information on in-utterance channel norm |

**TABLE OF CONTENTS**

# 1 INTRODUCTION

This document introduces the SREC embedded speech recognition engine. This gives a good introduction to a developer on the use of the SREC API and other related subjects. A comprehensive, HTML-based API reference is also available from source code

Chapter 2 discusses the SREC modules.

Chapter 3 walks through sample code, line by line, providing an example of how the API is meant to be used.

Chapter 4 provides an overview of creating grammars for SREC.

Chapter 5 details the various configuration parameters for SREC.

Chapter 6 describes the phonetic representation used by SREC.

Chapter 7 describes the SRecTest command file format used by the sample / test program.

# 2 SREC MODULES

The SREC API is built on top of two other libraries: ESR_Portable and ESR_Shared. SREC consists of eight modules:

- SR_AcousticModels
- SR_AcousticState
- SR_Grammar
- SR_Nametag
- SR_Nametags
- SR_Vocabulary
- SR_Recognizer
- SR_Session.

The following sections discuss them in greater detail.

## 2.1 ESR_Portable

The portable library abstracts I/O operations, memory management and other OS-dependent functionalities away from users. The library is used by SREC, as well as the SREC sample code and other Nuance products. .

Another major feature set provided by this library is LCHAR. LCHAR is a locale and hardware-independent character interface. On typical PCs, LCHAR translates to narrow characters (char) whereas in international distributions, LCHAR translates into wide characters (wchar_t). All string manipulation is abstracted away using LSTR*() macros which map to the correct implementation.

NOTE: This SREC release was only tested with narrow character builds.

## 2.2 ESR_Shared

The shared library provides utility classes used by the internal implementation. For example, ESR_Session (used to create SREC sessions) is implemented in this module.

## 2.3 SR_AcousticModels

This class represents a collection of acoustic models. Models may be loaded from or saved to disk or associated with a recognizer.

## 2.4 SR_AcousticState

This class represents the acoustic state of the caller and calling environment during a call in order to improve recognition accuracy. The base acoustic state may be loaded from or saved to disk and reset in between calls. Furthermore, this class may encapsulate multiple acoustic states such as those tuned for male or female models. As of version 1.0, the implementation of this feature is incomplete.

## 2.5 SR_Grammar

This class represents the recognition grammar, a collection of words and sentences the user may utter. Grammars may be loaded from or saved to disk.

Words may be dynamically added to predefined "slots". For example, given a grammar rule "lookup <name>" where <name> is a dynamic word slot, an application may populate the slot with names loaded from disk or names collected at runtime. Words are removed from slots using a reset function, which returns the grammar to its original state.

For performance reasons, the current implementation of SR_Grammar only supports offline grammar compilation with dynamic (onboard) slot manipulation.

### 2.5.1 SR_SemProc

SR_SemProc is a component used inside SR_Grammar, for semantic interpretation of recognition results. The "literal" of recognition results is passed through a parser to generate key/value associations which are easy for an application to extract. These key/value pairs allow for a language-independent and phrasing independent design of the application (i.e. no application changes needed to add synonyms or alternative phrase forms).

## 2.6 SR_Nametag, SR_Nametags

SR_Nametag represents phonetic Nametags, used for voice-enrolment. A nametag is created from a recognition result and may be inserted into dynamic grammar slots for subsequent recognitions.

SR_Nametags represents a collection of nametags and may be loaded from or saved to disk.

## 2.7 SR_Vocabulary

A vocabulary maps words to their phonetic representation, and is sometimes referred to as a dictionary. This mapping it backed up by a dictionary (basically a lookup table loaded from disk) or a TTP engine for unknown entries. Vocabularies are language-dependent. In the current version of the engine, there is no functionality to add words or pronunciations to the dictionary, other than by externally editing the input files.

## 2.8 SR_Recognizer, SR_RecognizerResult

The speech recognizer binds SR_AcousticModels and SR_Grammars and uses them for recognition. The recognizer takes in audio samples, processes them and returns recognition results. In case of successful recognition, the results contain a list of semantic results (multiple semantic results per nbest-list entry).

The following semantic keys are guaranteed to be defined:

| Key name (case sensitive) | Description |
|---|---|
| meaning | Equal to the literal, unless overridden by the semantic script. |
| literal | What the speaker said |
| conf | Confidence score |
| raws | Raw score |

## 2.9 SR_Session

The SREC session binds all other components. Aside from holding configuration parameters, the session initializes global objects used by all other SREC components. In order to use SREC API, one must first initialize the SR_Session and close it on shutdown.

The properties contained within the SREC session denote defaults. For example, when a new Grammar is loaded from disk, it inherits default values from the SREC session. In order to override configuration on that specific grammar instance, call SR_GrammarSetParameter() whereas if one wishes to override SREC-wide grammar parameters call ESR_SessionSetProperty() instead.

## 2.10 SR_EventLog

The SREC engine can be configured to log events and waveforms. See “Logging parameters” for how to control this logging. It should be used only for debugging, not at run-time.

## 2.11 Mandatory Arguments

Upon initializing the SREC_Session, most of the mandatory configuration parameters are read from the PAR file, however there are some parameters that the application is responsible for. The following parameters (in the SREC_Session) must be initialized before the SREC API may be used:

```
      cmdline.arbfile
      cmdline.argfile
      cmdline.bgsniff
      cmdline.channel
      cmdline.datapath
      cmdline.detail_res
      cmdline.lda
      cmdline.models
      cmdline.multable
      cmdline.parfile
      cmdline.rejfile
      cmdline.results
      cmdline.rules
      cmdline.tcp
      cmdline.use_image
      cmdline.vocabulary
```

These values usually come from the program command-line arguments or from the ARG file.

## 3 SAMPLE CODE

SREC includes a set of sample code and grammars which are used for testing SREC. Refer to these files for more examples on using the SREC functionality. [Note: some of these paths could change for different devices.]

Test programs are in:

```
device/out/target/product/sooner/system/bin
```

Source code for the test programs are in:

```
device/extlibs/srec/srec/test/SrecTest*
```

Libraries are in:

```
device/out/target/product/sooner/system/lib
```

Shell scripts are in:

```
device/extlibs/srec/config/en.us
```

and get installed in

```
/system/usr/srec/config/en.us
```

### 3.1 SRecTest

This is review of a sample program called SRecTest which reads a collection of audio samples from disk, runs a recognition against them and outputs the results to the screen.

Some code within a described function may be omitted and replaced with "`...`", especially error handling code, for the sake of readability. Also note that the LCHAR macro only works when defined as char.

Not all functions are described below. To help visualize the nesting of the functions explored below, use the indentation and coloring in the following list:

```
main()
    srec_test_init_system()
    srec_test_run_test()

        srec_test_run_test_init()
            srec_test_run_test_init_session()
                InitSession()
            srec_test_run_test_init_models()
            srec_test_run_test_init_vocab_grammar()

        srec_test_run_test_execute()
            srec_test_process_commands()
                srec_test_use_context()
                srec_test_recognize_nist()

        srec_test_run_test_shutdown()

    srec_test_shutdown_system()
```

### 3.1.1 main()

SRecTest is a highly factored program, with a structure that allows looping with varying amounts of resource destruction and allocation between the different levels of loops. This allows very thorough memory leak testing.

```
int main(int argc, LCHAR* argv [] )
  {
  int             test_status;
  unsigned int    num_shutdown_loops;
  unsigned int    current_shutdown_loop;
  unsigned int    num_continuous_run_loops;
  unsigned int    current_continuous_run_loop;
  unsigned int    srec_test_heap_size;
...
  ApplicationData applicationData;
  PLogger* logger;

  srec_test_heap_size = ( 4 * 1024 * 1024 );
  logger = NULL;
...
  test_status = srec_test_get_run_params ( &num_shutdown_loops,
      &num_continuous_run_loops );

  if ( test_status == 0 )
    {
    current_shutdown_loop = 0;

    while ( ( current_shutdown_loop < num_shutdown_loops ) && ( test_status == 0 ) )
      {
      test_status = srec_test_init_system ( srec_test_heap_size, logger, argc, argv );

      if ( test_status == 0 )
        {
        current_continuous_run_loop = 0;

        while ( ( current_continuous_run_loop < num_continuous_run_loops )
              && ( test_status == 0 ) )
          {
          test_status = srec_test_init_application_data ( &applicationData,
                argc, argv );

          if ( test_status == 0 )
            {
            test_status = srec_test_run_test ( &applicationData );
            srec_test_shutdown_application_data ( &applicationData );
            }
          current_continuous_run_loop++;
          }
        test_status = srec_test_shutdown_system ( logger );
        }
      current_shutdown_loop++;
      }
    }
```

- The main() function has two while loops, a “shutdown loop” and “continuous run loop”.
- A shutdown loop is one where the recognizer is shutdown between repeats of the loop.
- There are matching “init” and “shutdown” function call pairs around the loops for the system, and application data. Similar pairs of functions will be evident when deeper levels of the code are discussed.

- The function srec_test_init_system(), described in the next section, calls several functions to initialize system modules.
- The key function to step into is srec_test_run_test(), which calls srec_test_run_test_init() and srec_test_run_test_execute(), both described in more detail in the following sections.

### 3.1.2 Initialization

#### 3.1.2.1 srec_test_init_system()

```
static int srec_test_init_system ( unsigned int srec_test_heap_size, PLogger* logger,
int arg_count, LCHAR *arg_vals [] )
    {
    int init_status;
...
    init_status = srec_test_init_memory_system ( srec_test_heap_size );

    if ( init_status == 0 )
        {
        init_status = srec_test_init_file_system ( arg_count, arg_vals );

        if ( init_status == 0 )
            {
            init_status = srec_test_init_logging_system ( arg_count, arg_vals,
                    logger );

            if ( init_status != 0 )
                {
                srec_test_shutdown_file_system ( );
                srec_test_shutdown_memory_system ( );
                }
            }
        else
            {
            srec_test_shutdown_memory_system ( );
            }
        }
    return ( init_status );
    }
```

- This function calls three key initialization functions for key subsystems: the memory system, file system, and logging system.
- See the implementation of those functions for details.

#### 3.1.2.2 srec_test_run_test_init()

```
static int srec_test_run_test_init ( ApplicationData *applicationData )
    {
    int run_status;

    run_status = srec_test_run_test_init_session ( applicationData );

    if ( run_status == 0 )
        {
        run_status = srec_test_run_test_init_models ( applicationData );
        if ( run_status == 0 )
            {
            run_status = srec_test_run_test_init_vocab_grammar ( applicationData );

            if ( run_status != 0 )
                {
                srec_test_run_test_shutdown_models ( applicationData );
                srec_test_run_test_shutdown_session ( applicationData );
                }
            }
        else
            {
            srec_test_run_test_shutdown_session ( applicationData );
            }
        }
    return ( run_status );
    }
```

- This function sets up everything needed for a run through a test: the session, acoustic models, and grammars.
- Each of these functions is described in more detail below.

#### 3.1.2.3 srec_test_run_test_init_session()

```
static int srec_test_run_test_init_session ( ApplicationData *applicationData )
    {
    int             run_status;
    ESR_ReturnCode  esr_status;

    run_status = 0;
    LPRINTF(L("\nCreate recognizer:\n"));
    LPRINTF(L("    InitSession()\n"));
    esr_status = InitSession ( applicationData->argc, applicationData->argv );

    if ( esr_status == ESR_SUCCESS )
        {
        LPRINTF(L("    SR_RecognizerCreate()\n"));
        esr_status = SR_RecognizerCreate ( &applicationData->recognizer );

        if ( esr_status != ESR_SUCCESS )
            ...
```

- The function InitSession() is described in the next section.
- If it returns successfully, a new Recognizer object is created.

#### 3.1.2.4 InitSession()

```
ESR_ReturnCode InitSession(int argc, LCHAR *argv [])
{
    ESR_ReturnCode  init_status;
    LCHAR           path[P_PATH_MAX];
    size_t          len;
    len = P_PATH_MAX;
    init_status = ESR_CommandLineGetValue ( argc, argv, L("parfile"), path, &len );

    if ( init_status == ESR_SUCCESS )
        {
        init_status = SR_SessionCreate ( path );

        if ( init_status == ESR_SUCCESS )
            {

  /* Command-line options always override PAR file options */
            init_status = ESR_SessionImportCommandLine ( argc, argv );

            if ( init_status != ESR_SUCCESS )
                {
                SR_SessionDestroy ( );
                ...
```

- The SREC session is initialized using the mandatory 'parfile' command-line argument from the application-session, using ESR_CommandLineGetValue() and SR_SessionCreate().
- The command-line arguments are imported from the application session into the SREC session.

#### 3.1.2.5 srec_test_run_test_init_models()

```
static int srec_test_run_test_init_models ( ApplicationData *applicationData )
    {
    int             run_status;
    ESR_ReturnCode  esr_status;
    LCHAR           filename[P_PATH_MAX];
    size_t          len;

    run_status = 0;
    LPRINTF(L("Load acoustic models:\n"));
    len = P_PATH_MAX;
    esr_status = ESR_SessionGetLCHAR ( L("cmdline.models"), filename, &len );

    if ( esr_status == ESR_SUCCESS )
        {
        LPRINTF(L("    SR_AcousticModelsLoad()\n"));
        esr_status = SR_AcousticModelsLoad ( filename, &applicationData->models );

        if ( esr_status == ESR_SUCCESS )
            {
            LPRINTF(L("    SR_RecognizerSetup()\n"));
            esr_status = SR_RecognizerSetup ( applicationData->recognizer,
                    applicationData->models );

            if ( esr_status != ESR_SUCCESS )
                ...
```

- Retrieve the 'models' command-line argument, which denotes the acoustic models filename, and load the acoustic models from disk into the 'models' object.
- ESR_SessionGetLCHAR() returns a copy of an LCHAR*-type value.
- SR_RecognizerSetup() associates acoustic models with the recognizer.

#### 3.1.2.6 srec_test_run_test_init_vocab_grammar()

```
static int srec_test_run_test_init_vocab_grammar ( ApplicationData *applicationData )
      {
      int              run_status;
      ESR_ReturnCode   esr_status;
      LCHAR            filename[P_PATH_MAX];
      size_t           len;

      run_status = 0;
     /* Create vocabulary object and associate with grammar */
      LPRINTF(L("Create vocabulary object and associate with grammar:\n"));
      len = P_PATH_MAX;
      esr_status = ESR_SessionGetLCHAR ( L("cmdline.vocabulary"), filename, &len );

      if ( esr_status == ESR_SUCCESS )
          {
          LPRINTF(L("    SR_VocabularyLoad()\n"));
          esr_status = SR_VocabularyLoad ( filename, &applicationData->vocabulary );

          if ( esr_status == ESR_SUCCESS )
              {
              LPRINTF(L("    SR_VocabularyGetLanguage()\n"));
              esr_status =  SR_VocabularyGetLanguage ( applicationData->vocabulary,
                      &applicationData->locale );

              if ( esr_status == ESR_SUCCESS )
                  {
                  /* start a new log session */
                  LPRINTF( L("Start a new log session:\n") );
                  LPRINTF( L("    SR_RecognizerLogSessionStart()\n") );

                  esr_status = SR_RecognizerLogSessionStart (
                       applicationData->recognizer, L("SRecTest.session1") );

                  if ( esr_status != ESR_SUCCESS )
                      {
                      SR_VocabularyDestroy ( applicationData->vocabulary );
                      applicationData->vocabulary = NULL;
                      run_status = -1;
                           ...
```

- The program gets the filename of the vocabulary file, then uses SR_VocabularyLoad() to load the vocabulary.
- Then set the locale based on the language of the vocabulary.
- If this succeeds, start a new log session with SR_RecognizerLogSessionStart()

### 3.1.3 Execution

#### 3.1.3.1 srec_test_run_test_execute()

```
  static int srec_test_run_test_execute ( ApplicationData *applicationData )
      int              run_status;
      ESR_ReturnCode   esr_status;

      run_status = 0;
      applicationData->nametag = NULL;

      LPRINTF(L("Recognize:\n"));
      LPRINTF(L("    SR_NametagsCreate()\n"));

      esr_status = SR_NametagsCreate ( &applicationData->nametags );

      if ( esr_status == ESR_SUCCESS )
          {
          run_status = srec_test_process_commands ( applicationData );
          SR_NametagsDestroy ( applicationData->nametags );
          applicationData->nametags = NULL;

          if ( run_status != 0 )
              ...
```

- Now the program is out of srec_test_run_test_init(), and almost ready to start recognition.
- SR_NametagsCreate() is called to setup a collection of nametags (empty).
- Then srec_test_process_commands() is called to process command files, as described below.

#### 3.1.3.2 srec_test_process_commands()

```
int srec_test_process_commands ( ApplicationData *data )
    {
    int             process_status;
    PFile           *command_file;
    FILE            *results_file;
    LCHAR           *got_line_ok;
    LCHAR           linebuffer [MAX_LINE_LENGTH];
    size_t          recognition_count;

    recognition_count = 0;
    process_status = srec_test_open_command_file ( &command_file );

    if ( process_status == 0 )
        {
        process_status = srec_test_open_results_file ( &results_file );

        if ( process_status == 0 )
            {
            do
                {
                got_line_ok = pfgets ( linebuffer, MAX_LINE_LENGTH, command_file );

                if ( got_line_ok != NULL )
                    srec_test_execute_command ( data, linebuffer, results_file,
                                                         &recognition_count );
                }

            while ( ( got_line_ok != NULL ) && ( process_status == 0 ) );

            srec_test_close_results_file ( results_file );
            }
        srec_test_close_command_file ( command_file );
        }
    return ( process_status );
    }
```

- Open both a results file and TCP file from disk. The results file will be used to log events to disk. The TCP file is an index of audio samples to be read from disk.
- pfgets() is a portable implementation of fgets(); reading one line of text from the TCP file.
- If the line is read successfully, srec_test_execute_command() is called (see next section), passing the line from the TCP file and a pointer to the results file.

#### 3.1.3.3 srec_test_execute_command()

```
int srec_test_execute_command ( ApplicationData *data, LCHAR *text,
                                   FILE *results_file, size_t *recognition_count )
    {
    int         execute_status;
    LCHAR       *current_command_start;
    LCHAR       *current_end_command;
    LCHAR       command [MAX_LINE_LENGTH];
    SR_Grammar  *active_grammar;
    LCHAR       log_buffer [LOG_BUFFER_SIZE];

    srec_test_get_active_grammar ( data, &active_grammar );
    current_command_start = text;

    execute_status = srec_test_get_one_command_item ( current_command_start,
                             MAX_LINE_LENGTH, command, &current_end_command );

    if ( execute_status == 0 )
        {
        if ( LSTRCMP ( command, L("recognize_nist") ) == 0 )
            execute_status = srec_test_recognize_nist_file ( active_grammar, data,
                         results_file, current_end_command, recognition_count );

        else if ( LSTRCMP ( command, L("recognize_pcm") ) == 0 )
            execute_status = srec_test_recognize_pcm_file ( active_grammar, data,
                         results_file, current_end_command, recognition_count );

        else if ( LSTRCMP ( command, L("context_load") ) == 0 )
            execute_status = srec_test_load_context ( data, current_end_command );

        else if ( LSTRCMP ( command, L("context_use") ) == 0 )
            execute_status = srec_test_use_context ( active_grammar, data,
                         current_end_command );
        ...
```

- This function has a much longer sequence of if/else if/else if… than reproduced above. There are a 20 commands matched by this function, which dispatches to the appropriate command-handling routines.
- The syntax of the command file is described in section 7. The following section shows an example command file, bothtags5.tcp.

#### 3.1.3.4 bothtags5.tcp

```
# this test describes operation of voicetags and texttags
#
# to run this script please be sure to prepare the input grammars
#
# grxmlcompile -par baseline11k.par -grxml bothtags5.grxml
# make_g2g -base bothtags5,addWords=100 -out bothtags5.g2g
# grxmlcompile -par baseline11k.par -grxml enroll.grxml
# make_g2g -base enroll -out enroll.g2g
#
# now run the script with the following command line
# /system/bin/SRecTest -parfile baseline11k.par -tcp tcp/bothtags5.tcp \
#                          -datapath audio/ >out_SHIP_bothtags5.txt 2>&1
#
# VOICETAGS PREPARATION
# let's load up the voice-enrollment "grammar" and refer to it as "ve" later
# ROOT is the name of the rule we activate in that grammar,
# no other rule should work anyways

context_load  grammars/bothtags5.g2g  BothTags   trash  not_ve
context_load  grammars/enroll.g2g VoiceEnroll ROOT ve

# VOICETAGS
# the pattern for voicetags is :
# (1) the we loadup the voice-enrollment grammar
# (2) the user utters the training token for the "voicetag"
#                          (sometimes loosely called nametag)
# (3) the voicetag "recognition result" from that training token
#                          is then added to a list of tags (for saving to disk)
# (4) the voicetag is also added to the primary recognition grammar

context_use  VoiceEnroll
recognize_nist  v139/v139_024.nwv 0 0 VCE_Pete_Gonzalez
context_free  VoiceEnroll
context_use  BothTags
addword_from_last_nametag  @Names VCE_Pete_Gonzalez 0
add_to_nametags  VCE_Pete_Gonzalez
context_free  BothTags
...
```

- See section 7 for details of the command file format.
- This script starts by explaining (in the comments) how to build binary grammars for the test and run the test.
- It then makes context_load commands with the grammars, then shows a sequence of context_use, recognize_nist, and context_free calls for setting up voicetags.
- See below to see how context_use and recognize_nist are implemented when handled by srec_test_execute_command() in srec_test_use_context() and srec_test_recognize_nist_file()

#### 3.1.3.5 srec_test_use_context()

```
int srec_test_use_context ( SR_Grammar *active_grammar, ApplicationData *data,
                                  LCHAR *command_text )
    {
```

```
    int             use_status;
    ESR_ReturnCode  esr_status;
    int             grammar_num;
    BOOL            found_grammar;
    BOOL            grammar_is_active;
    BOOL            grammar_is_ve;
    LCHAR           grammar_id [P_PATH_MAX];

    if ( active_grammar == NULL )
        {
        use_status = srec_test_get_one_command_item ( command_text, P_PATH_MAX,
                        grammar_id, NULL );

        if ( use_status == 0 )
            {
            found_grammar = srec_test_get_grammar_from_id ( data, grammar_id,
                        &grammar_num, &grammar_is_active, &grammar_is_ve );

            if ( found_grammar == TRUE )
                {
                esr_status = SR_RecognizerSetupRule ( data->recognizer,
                        data->grammars [grammar_num].grammar,
                        data->grammars [grammar_num].ruleName );

                if ( esr_status == ESR_SUCCESS )
                    {
                    esr_status = SR_RecognizerActivateRule ( data->recognizer,
                            data->grammars [grammar_num].grammar,
                            data->grammars [grammar_num].ruleName, 1 );
                    if ( esr_status == ESR_SUCCESS )
                        {
                        if ( data->grammars [grammar_num].is_ve_grammar == TRUE )
                            {
                            esr_status = SR_RecognizerSetBoolParameter (
                                    data->recognizer, L("enableGetWaveform"), TRUE );

                            if ( esr_status == ESR_SUCCESS )
                                {
                                data->active_grammar_num = (int)grammar_num;
                                }
                            else
                                {
                                use_status = -1;
                                ...
```

- This function processes a context_use command from a TCP command file.
- The ApplicationData structure includes a grammars array that contains a pointer to all loaded grammars.
- srec_test_get_grammar_from_id() is called to locate a loaded grammar (from that array).
- SR_RecognizerActivateRule() associates the grammar with the recognizer.

#### 3.1.3.6 srec_test_recognize_nist_file()

```

static int srec_test_recognize_nist_file ( SR_Grammar *active_grammar,
               ApplicationData *data, FILE *results_file
               LCHAR *command_text, size_t *recognition_count )
    {
    int                     recognize_status;
    ESR_ReturnCode          esr_status;
    SR_RecognizerStatus     esr_recog_status;
    SR_RecognizerResultType result_type;
    LCHAR                   *transcription;
    LCHAR                   waveform [MAX_LINE_LENGTH];
    LCHAR                   bos [MAX_LINE_LENGTH];
    LCHAR                   eos [MAX_LINE_LENGTH];
    PFile                   *waveform_file;
    BOOL                    hit_eof;

    if ( active_grammar != NULL )
        {
        recognize_status = srec_test_get_three_command_items ( command_text,
                MAX_LINE_LENGTH, waveform,
                MAX_LINE_LENGTH, bos, MAX_LINE_LENGTH,
                eos, NULL, &transcription );
        if ( recognize_status == 0 )
            {
            recognize_status = srec_test_log_reco_from_file_data ( active_grammar,
                                    data, waveform, bos, eos, transcription );

            if ( recognize_status == 0 )
                {
                recognize_status = srec_test_open_nist_file ( waveform,
                                    &waveform_file );

                if ( recognize_status == 0 )
                    {
                    if ( ( data->forced_rec_mode == ForcedRecModeOn ) ||
                              ( data->forced_rec_mode == ForcedRecModeOneTime ) )
                        SR_GrammarAllowOnly ( active_grammar, data->transcription );
                    esr_status = SR_RecognizerStart ( data->recognizer );

                    if ( esr_status == ESR_SUCCESS )
                        {
                        ( *recognition_count )++;
                        hit_eof = FALSE;

                        do
                            {
                            recognize_status = srec_test_get_audio_from_file (
                                    waveform_file, data, &hit_eof );

                            if ( recognize_status == 0 )
                                recognize_status = srec_test_feed_recognizer (
                                    data, hit_eof, &esr_recog_status, &result_type  );
                            }
                        while ( ( hit_eof == FALSE ) && ( result_type !=
                                    SR_RECOGNIZER_RESULT_TYPE_COMPLETE ) &&
                                     ( recognize_status == 0 ) );

                        if ( recognize_status == 0 )
                            {
                            recognize_status = srec_test_flush_audio ( data,
                                    &esr_recog_status, &result_type );

                            if ( recognize_status == 0 )
                                {
                                recognize_status = srec_test_process_results ( data,
                                        esr_recog_status,
                                        results_file, *recognition_count );
                                }
                            }
                        }
                        esr_status = SR_RecognizerStop ( data->recognizer );
                        ...
```

- srec_test_recognize_nist_file() opens an audio test file in the nist format and sends the audio to the recognizer.
- The grammar is expected to have already been loaded and activated with the use_context command.
- The audio is read in parts and passed to the recognizer within the do / while loop using srec_test_get_audio_from_file() and srec_test_feed_recognizer() until there is no more audio or SR_RECOGNIZER_RESULT_TYPE_COMPLETE or the recognition has been aborted.

### 3.1.4 Shutdown

#### 3.1.4.1 srec_test_run_test_shutdown

```
static int srec_test_run_test_shutdown ( ApplicationData *applicationData )
    {
    int shutdown_status;

    shutdown_status = srec_test_run_test_shutdown_vocab_grammar ( applicationData );

    if ( shutdown_status == 0 )
        {
        shutdown_status = srec_test_run_test_shutdown_models ( applicationData );

        if ( shutdown_status == 0 )
            shutdown_status = srec_test_run_test_shutdown_session ( applicationData );
        }
    return ( shutdown_status );
    }
```

- This is the counterpart to srec_test_run_test_init(), and makes calls to similar functions to release the resources used by the vocabulary, grammar, acoustic models, and session.
- These functions are not discussed in detail, but always bear one rule in mind: tear things down in the opposite order from when they were created.

#### 3.1.4.2 srec_test_shutdown_system()

```
static int srec_test_shutdown_system ( PLogger *logger )
    {
    int shutdown_status;

    shutdown_status = srec_test_shutdown_logging_system ( logger );

    if ( shutdown_status == 0 )
        {
        shutdown_status = srec_test_shutdown_file_system ( );

        if ( shutdown_status == 0 )
            shutdown_status = srec_test_shutdown_memory_system ( );
        }
    return ( shutdown_status );
    }
```

- This is the counterpart to srec_test_init_system(), and does the expected release of system resources.

# 4 CREATING GRAMMARS FOR SREC

## 4.1 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
- there is no support for language specifications inside rules or individual items
- there is no support for rule imports or rule exports, scope specifications such as "public" or "private" are largely ignored
- SREC uses the tinyxml xml document representation which does not link to a conformance checker
  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. This parameter is overridden by the parfile specified on the grxml command line.

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

## 4.2 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
- .params … parameters to be used at grammar load time

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.

### 4.3 Pronunciation dictionaries at grammar compilation time

At grammar compilation time, the words in the grxml grammar are looked up the text-format user dictionary, which is specified in the file that the “cmdline.vocabulary” parameter points to. This dictionary will typically be much larger than the dictionary that is packaged for deployment. The default pronunciation dictionary is located at:

```
device/extlibs/srec/config/en.us/dictionary/large.ok
```

This dictionary is used by all grammar compilations. :

When words are found in the dictionary, all pronunciations for that word are used, and the word is not looked up in any other dictionary. The dictionary is case-sensitive, but the same word in different casing should not be specified in the dictionary, rather a case-convention should be adopted for you particular application.

When words are not found in the (.ok) dictionary, then the word is split into items on spaces and underscores. Each item is looked up separately in the (.ok) dictionary, and words that are not found there are passed to the G2P engine. The SETI G2P engine is case-insensitive and ignores all non-alphanumeric characters except for “’” (forward apostrophe, as in can’t). It is strongly advised that orthographies be text normalized before passing them to the SETI G2P engine. Normalization may include stripping/substituting out or translating non-alphanumeric characters, replacing spaces with underscores, spelling out large numbers, etc. It is impossible to anticipate all possible orthographies, but the types of orthographies should be anticipated and tested. A good developer should look at the trends in a particular source of words (e.g. mp3 titles containing creative spellings, mixed digits and alpha, contact names that can contain extra spaces or that may be preceded with titles or abbreviations, etc.

The **dictTest** tool can be used to test words in the dictionary and/or to test the formatting of the dictionary that you've created.

Adding words to the dictionary should follow the phonological conventions documented in this guide. Similar words in the **large.ok** dictionary can also be used as inspiration.

```
(1) % export ESRSDK=`pwd`/extlibs/srec
% export ESRLANG=en.us
% dictTest -par /device/extlibs/srec/config/en.us/baseline.par
Dictation Test Program for esr (Nuance Communications, 2007)
'qqq' to quit
> hello
hello : helO
>
```

## 4.4 Metas

### 4.4.1 word_penalty

“word_penalty” is used to balance insertions and deletions produced by the recognizer. To reduce the number of insertions, word_penalty should be increased. To reduce the number of deletions, word_penalty should be decreased. A default value of 40 is used when the “word_penalty” is not specified.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<grammar xml:lang="en-us" version="1.0" root="ROOT">
<meta name="word_penalty" content="40"/>
```

### 4.4.2 do_skip_interword_silence

This is used only with the voice enrollment “enroll” grammar. During typical grammar compilation we add optional silence after the pronunciation of each word, which allows the user to pause between words. In the enrollment grammar, words are phonemes, and there is already a word associated with “silence”. As such we don’t need to allow optional silence after for each word (phoneme). The voice-enrollment grammar is carefully designed, developers should be take caution if editting it.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<grammar xml:lang="en-us" version="1.0" root="ROOT">
<meta name="do_skip_interword_silence" content="true"/>
```

## 4.5 Dynamic Grammars and slot addition

All recognition contexts must have a grammar and all grammars must be compiled offline. For recognition contexts that have dynamic content the engine supports slot-based dynamic word addition, but even if there are no static words, an “empty” grammar must still be created and compiled.

SREC implementation allows that the number of words to be added to a grammar be specified at grammar compilation time:

```
(3)% make_g2g –base mygrammar[,addWords=$N] –out
```

Note that the $N is a soft-limit on the number of words to be added, and the space required for these words is pre-allocated in the file and in memory at grammar load-time. This obviously assumes certain limits on the number of pronunciations per word and number of phonemes per pronunciation. Thus, the space required is an estimate because averages are used to calculate it:

- Average number of characters per word: 18
- Average number of arcs & nodes (~phonemes) per word: 10 & 7
- Average number of characters per semantic script of word: 45

SREC is able to dynamically allocate slot memory at runtime, so the “,addWords=$N” is really intended for platforms on which we prefer NOT to use allocations that way.

The file size of the g2g file is nearly identical to the memory requirement for this grammar.

The “slot” is the grammar element into which words will be added. The grammar in question may consist of a carrier phrase such as:

```
lookup $Names
```

.. where **$Names** is the slot name.

The grammar specification must be such that the slot is specified as a rule unto itself in the grammar, and must be empty except for a single marker word

The grxml file for this example must look as follows:

```
…
<rule id="ROOT" scope="public">
        <item>
```

```
         <ruleref uri="#LOOKUP"/>
         <tag>M_N=LOOKUP.NAME?LOOKUP.NAME:'';
                M_P=LOOKUP.PLACE?LOOKUP.PLACE:'';
                M_T=LOOKUP.TELN?LOOKUP.TELN:'';</tag>
       </item>
</rule>

   <rule id="LOOKUP">
     <item>lookup</item>
     <one-of>
       <item>
          <ruleref uri="#NAMES"/>
          <item repeat="0-1">
              <item>at</item>
              <ruleref uri="#PLACES"/>
              <tag>PLACE=PLACES.P</tag>
          </item>
          <tag>NAME=NAMES.N</tag>
       </item>
       <item>
          <ruleref uri="#S"/>
          <tag>TELN=S.X;</tag>
        </item>
     </one-of>
   </rule>
…
```

Note the use of the special __Names__ marker. This word does not need to be in the pronunciation dictionary, it will be handled in a special way.

The API functions to be used for dynamic word addition are summarized in the table below:

```
/**
 * Adds word to rule slot.
 *
 * @param self SR_Grammar handle
 * @param slot Slot name, eg “@Names”
 * @param word Word to be added to the slot
 * @param pronunciation Word pronunciation (optional). Pass NULL to omit.
 * @param weight value to associate the word; a positive integer penalty (0-300)
 * @param tag eScript semantic expression for the word. In other words, eScript will execute
 *            "MEANING=<tag>"
 */
ESR_ReturnCode SR_GrammarAddWordToSlot(SR_Grammar* self, const LCHAR* slot, const LCHAR* word,
const LCHAR* pronunciation, int weight, const LCHAR* tag);
/**
 * Removes all elements from rule slot.
 *
 * @param self SR_Grammar handle
 * @param slot Slot name, eg “@Names”
 */
ESR_ReturnCode SR_GrammarResetSlot(SR_Grammar* self, const LCHAR* slot);
/**
 * Adds word to rule slot.
 *
 * @param self SR_Grammar handle
 * @param slot Slot name, eg “@Names”
 * @param nametag Nametag to be added to the slot
 * @param weight value to associate the word; a positive integer penalty (0-300)
 * @param nametag Nametag to be added to the slot
 */
ESR_ReturnCode SR_GrammarAddNametagToSlot(SR_Grammar* self, const LCHAR* slot, const SR_Nametag*
nametag, int weight, const LCHAR* tag);
```

### 4.5.1 Dynamic word addition

SREC has now the ability to add words dynamically to a grammar. As a consequence, you don’t have to specify the maximum number of words that can be added to a grammar a priori (during the compilation time), but instead you can strip out the “addWords” key from the make_g2g command (or you can set it to zero), and the engine will dynamically allocate the appropriate memory space needed to add all of the words you need, hence saving some memory space.

```
(3)% make_g2g –base mygrammar,addWords=0 –out
```

Or

```
(3)% make_g2g –base mygrammar –out
```

Please note that once you decided to reset the slots of the grammar, you will be offered dynamic word addition mode only, even if you had pre-allocated some space during grammar compilation time. In other words, the pre-allocated space will be lost once the grammar is reset.

## 4.6 Testing for Semantic Results (parseStringTest)

After the grammar has been written in grxml format, it is often useful to check the validity of the semantic tags without having to activate the recognition engine. This is done by specifying phrases in the same form the recognizer produces the words sequence of recognized words. For a specified input phrase, the results in the form of key values pairs are as in this example:

```
Input Phrase: phone dial one two three
Semantic Result:
myRoot.N0 = “123”
myRoot.id = ‘PHONE2”
myRoot.meaning = “PHONE2 123”
```

To do this the parseStringTest tool can be used as follows:

```
parseStringTest –base mygrammar.g2g
parseStringTest –base mygrammar
```

The parseStringTest tool will prompt you to enter a recognition result such as “phone dial one two three”: The tool will respond by telling the user whether the sentence is valid according to the grammar and what the keys and values will be under the results API, as noted above.

## 4.7 Changes from previous grammar format specifications

1. The <count number=”optional” > formalism is no longer supported, replaced with <item repeat=”0-1”> formalism

2. The <item tag=”script” formalism is not longer supported, replaced with <item><tag>script</tag> formalism

## 4.8 Grammars description and parse examples

The date grammar includes popular expressions of year, month, and day. For example,
YYYY = four-digit year,
MM = two-digit month (01=January, etc.),
DD = two-digit day of month (01 through 31),
DAY = Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.
The date grammar also allows relative date such as "yesterday" and "tomorrow". Some parse examples are:
- friday eleventh october twenty thirty
- thursday the twenty second of october two thousand twenty eight
- saturday fifteenth november nineteen thirty eight

The time grammar is used to express clock time. For example,
hh = two digit hour (00 through 23) (am/pm is allowed),
mm = two digit minute (00 through 59),
ss = two digit second (00 through 59).
Some parse examples are:
- twelve o'clock PM
- ten forty two PM
- twelve hour

The digit grammar is a loop of digits (0~9), for example,
- zero nine one two
- three four five six eight oh.
0 could be "zero" or "oh".

The lookup grammar is a phone number lookup grammar; the keyword is "lookup". People names or phone numbers are followed by "lookup". For example,
- lookup Peter
- lookup David
- lookup one three four five six one one

# 5 CONFIGURATION PARAMETERS

## 5.1 Setting configuration parameters

Configuration parameters are set in different ways depending on the underlying operating system. In the SREC API documentation, the description of each configuration parameter indicates the appropriate configuration mechanisms.

How one sets parameters depends on how SREC is used. The mechanisms for setting configuration parameters are:

- Grammar files - many parameters can be controlled with the <meta> tag inside XML grammars. these parameters are dynamic in the sense that the values might from one recognition event to the next.
- API - many parameters can only be set with SR_RecognizerSetParameter(), SR_GrammarSetParameter() and ESR_SessionSetProperty() functions.
- TRC keys for diagnostic logging

There are no SREC parameters that are set via environment variables.

## 5.2 Rules of parameter precedence

Parameters can be set in various ways including via: API function calls, statements in grammar files, and default settings in user and baseline configuration files. Not all parameters can be set by all methods. Regardless of how a parameter is set, its value is resolved when recognition starts using the following order of precedence

- ESR_RecognizerSetParameter() takes the highest precedence.
- A grammar can define parameters via the <meta> tag.
- A user configuration file can provide application- or platform-specific defaults. These defaults are loaded into the ESR_Session.
- The baseline.xml configuration file from Nuance provides default values.
- In the absence of a needed configuration parameter, SREC either provides a hard-coded value or generates an error.

## 5.3 Parameters set in PAR file

The following list enumerates the various configuration parameters, their meaning and typical values.

### 5.3.1 Configuration Parameters

| Parameter Name | Description | Typical Value(s) | Minimum | Maximum |
|---|---|---|---|---|
| | | | | |
| `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 |

### 5.3.2 Non-Tunable Parameters

| Parameter Name | Description | Typical Value(s) | | |
|---|---|---|---|---|
| `CREC.Frontend.mel_dim` | Number of Cepstrum coefficients. | 12 | | |
| `CREC.Frontend.premel` | Pre-emphasis coefficient. | 0.9 | | |
| `CREC.Frontend.lowcut` | Lower cutoff frequency (Hz). | 125 | | |
| `CREC.Frontend.highcut` | Upper cutoff frequency (Hz); this parameter should be consistent with the acoustic model specified in the par file; 5500 for the 11kHz acoustic model; 4000 for the 8kHz acoustic model; see also "samplerate" | 5500 | | |
| `CREC.Frontend.window_factor` | Analysis window size. (Number of frame periods). | 2 | | |
| `CREC.Frontend.offset` | Offset value to be removed from input audio. | 0 | | |
| `CREC.Frontend.ddmel` | Turns delta delta Cepstrum calculation on or off. | YES | | |
| `CREC.Frontend.melA` | Cepstrum scaling coefficients. | See Note 1 | | |
| `CREC.Frontend.melB` | Cepstrum offset coefficients. | See Note 1 | | |
| `CREC.Frontend.dmelA` | Delta Cepstrum scaling coefficients. | See Note 1 | | |
| `CREC.Frontend.dmelB` | Delta Cepstrum offset coefficients. | See Note 1 | | |
| `CREC.Frontend.ddmelA` | Delta delta Cepstrum scaling coefficients. | See Note 1 | | |
| `CREC.Frontend.ddmelB` | Delta delta Cepstrum offset coefficients. | See Note 1 | | |
| `CREC.Frontend.peakdecayup` | Non-linear filtering co-efficient. | 0.3 | | |
| `CREC.Frontend.peakdecaydown` | Non-linear filtering co-efficient. | 0.7 | | |
| `CREC.Frontend.do_skip_even_frames` | When set to YES, every other front-end frame is not processed. | YES | | |
| `CREC.Frontend.peakdecayup` | Spectral Envelope Detection parameter. | 0.3 | | |
| `CREC.Frontend.peakdecaydown` | Spectral Envelope Detection parameter. | 0.7 | | |

| | | | | |
|---|---|---|---|---|
| CREC.Frontend.cuberoot | Replace Log function with a cube root in dB power calculations. | NO | | |
| CREC.Frontend.forgetfactor | Forget factor for channel normalisation. | 50 | | |
| CREC.Acoustic.dimen | Defines the number of dimensions in the whole word acoustic vector. | 36 | | |
| CREC.Acoustic.whole_skip | State skip penalty for whole words. | 40 | | |
| CREC.Acoustic.whole_stay | State loop penalty for whole words. | 40 | | |
| CREC.Acoustic.minvar | Floor value for variance. | 2860 | | |
| CREC.Acoustic.maxvar | Ceiling value for variance. | 2860 | | |
| CREC.Acoustic.frame_period | Acoustic matching frame period. | 20 | | |
| CREC.Pattern.imelda_scale | Covariance scaling in IMELDA. | 14 | | |
| CREC.Pattern.mix_score_scale | Scaling factor for subword distances. | 0.46 | | |
| CREC.Pattern.uni_score_scale | Scaling factor for wholeword distances. | 0.46 | | |

#### 5.3.2.1 Other Recognition Parameters

| Parameter Name | Description | Typical Value(s) | | |
|---|---|---|---|---|
| CREC.Pattern.dimen | Number of dimensions in the pattern vector; the acoustic models are packaged with 28 dimensions, only a value lower than 28 can be used | 26-36 | | |

### 5.3.3 Tunable Parameters

The following parameters may be adjusted during parameter tuning in order to optimize the performance of the recognizer on a particular test Corpus.

**Beginning of Utterance detection**
The following parameters control the initial speech / silence detection within SREC.

| Parameter Name | Description | Typical Value(s) | Minimum | Maximum |
|---|---|---|---|---|
| CREC.Frontend.do_smooth_c0 | Controls whether C0 is smoothed prior to voicing detection. | NO | | |
| CREC.Frontend.speech_detect | Before any kind of processing begins, first detect that some speech has occurred. This parameter sets the threshold for this initial speech detection. | 14 | 0 | 80 |
| CREC.Frontend.start_windback | Once 'speech_detect' has been exceeded, rewind by 'start_windback' frames then begin frame by frame processing. Specifically, this means that each frame is marked as either speech, unsure or silence. | 50 | 0 | 60 |
| CREC.Frontend.speech_above | Threshold for marking an individual frame as speech. [not used] | 18 | | |
| CREC.Frontend.ambient_within | While marking each frame as either speech, unsure or silence, maintain an estimate of the background level. | 7 | 0 | 80 |

| | | | | |
|---|---|---|---|---|
| | Frames where C0 does not exceed the sum of the current background value and 'ambient_within' are used to update the background estimate. | | | |
| `CREC.Frontend.utterance_allowance` | Where a frame is marked as silence and the previous or next frame is marked as speech, up to 'utterance allowance' frames of the region marked as 'silence' will be changed to 'unsure' in order to pad out the speech section. This reduces the risk of missing part of the utterance. | 40 | 0 | 80 |

**Timeout and End-of-Utterance Detection Parameters**

| Parameter Name | Description | Typical Value(s) | Minimum | Maximum |
|---|---|---|---|---|
| `SREC.Recognizer.utterance_timeout` | maximum number of (10ms) frames to wait for declaring start of speech (ms); beyond this number we assume there no useful speech to come | 400 | 100 | 600 |
| `cmdline.bgsniff` | number of (10ms) frames to wait for background level estimation | 25 | 4 | 25 |
| `cmdline.bgsniff_min` | minimum number of (10ms) frames to wait for background level estimation | 4 | 4 | 4 |
| `cmdline.silence_duration_in_frames` | Wait for this many quiet (10ms) frames before declaring end of speech, counting starts *after* the hold_off period, see end_of_utterance_hold_off_in_frames , measured in 10ms frames | 100 | 0 | 999 |
| `cmdline.gatedmode` | whether start of speech pointing should be done at all | 1 | 0 | 1 |
| `cmdline.end_of_utterance_hold_off_in_frames` | number of (10ms) frames to skip before attempting end of speech detection | 30 | 30 | 30 |
| `CREC.Recognizer.max_frames` | Maximum number of (20ms) frames on which we can run recognition; after this number end-of-speech is automatically declared, measured in 20ms frames | 1000 | 100 | 2000 |
| `CREC.Recognizer.eou_thresold` | Score delta, by which the best search state needs to be best before starting to count frames for timeouts below. | 150 | 1 | 999 |
| `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 | 1 | 2000 |
| `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, ie see do_skip_even_frames). | 200 | 1 | 2000 |
| `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 | 1 | 2000 |

**Channel normalization Parameters**
ESR/SREC run channel normalization by estimating the channel mean and subtracting it from incoming frames before they are presented to the recognizer. The cross-utterance channel normalization is slow, the in-utterance channel normalization is fast.

| Parameter Name | Description | Typical Value(s) | Minimum | Maxiimum |
|---|---|---|---|---|
| `CREC.Frontend.swicms.forget_factor` | The weight given to the long-term average cmn vector, higher values imply learning will be slow and steady, low values mean learning is fast but possibly unsteady. The weight is used as a number of 20ms frames such that new_mean = (400*old_mean + new_data*amt_new_data)/(400+amt_new_data) | 400 | 1 | 65000 |
| `CREC.Frontend.swicms.sbindex` | Balances the frames to be used for channel estimation, 100 means use speech only, 0 means use background only, with in-between integers allowed | 100 | 0 | 100 |
| `CREC.Frontend.swicms.inutt.forget_factor2` | The weight given to the short-term average cmn vector, higher values means learning is slow, low values mean learning is fast. The same formula as above is used. | 40 | 1 | 65000 |
| `CREC.Frontend.swicms.inutt.disable_after` | Number of 20ms frames after which to disable further short-term cmn learning (although we continue to apply it); we restart learning after a channel reset | 200 | 0 | 65000 |
| `CREC.Frontend.swicms.inutt.enable_after` | Number of 20ms speech frames after which to apply short-term cmn; i.e. before which the estimate is unreliable; consideration of speech frames is done by consulting CREC.Frontend.start_windback | 10 | 0 | 65000 |

**Accuracy and N-Best Parameters**
ESR/SREC is capable of creating N-Best results. The cpu and memory usage, plus accuracy on top choice and nbest choices can be adjusted with these parameters.

| Parameter Name | Description | Typical Value(s) | Minimum | Maxiimum |
|---|---|---|---|---|
| `CREC.Recognizer.max_fsm_arcs` | Maximum number of grammar arcs the recognizer can support searching | 25000 | 100 | 65000 |
| `CREC.Recognizer.max_fsm_nodes` | Maximum number of grammar nodes the recognizer can searching | 14500 | 100 | 65000 |
| `CREC.Recognizer.max_model_states` | Number of states to be scored in the acoustic model; this should not be changed unless the acoustic model changes | 3600 | 3600 | 3600 |
| `CREC.Recognizer.max_frames` | Maximum number of frames on which we can run recognition; after this number end-of-speech is automatically declared | 1000 | 100 | 2000 |
| `CREC.Recognizer.viterbi_prune_thresh` | score based pruning threshold, higher means more accuracy and more cpu intensive | 400 | 100 | 999 |
| `CREC.Recognizer.max_hmm_tokens` | memory based pruning on active | 400 | 40 | 800 |

| | | | | |
|---|---|---|---|---|
| | search states, higher means more accuracy and more cpu/memory use | | | |
| `CREC.Recognizer.max_fsmnode_tokens` | memory based pruning on active search states, higher means more accuracy and more cpu/memory use | 400 | 40 | 800 |
| `CREC.Recognizer.max_altword_tokens` | memory based pruning for nbest, higher means more nbest accuracy but more memory | 400 | 40 | 800 |
| `CREC.Recognizer.max_word_tokens` | memory based pruning for nbest, higher means more memory use but more dense nbest | 2000 | 40 | 9000 |
| `CREC.Recognizer.num_wordends_per_frame` | memory based pruning for nbest, higher means more dense nbest, but too high will require re-pruning, ie max_word_tokens ~ nframes*num_wordends_per_frame, but too many frames will cause cpu intense repruning | 10 | 1 | 20 |

**Logging Parameters**
SREC is capable of logging events, words added to the grammar dynamically, and recoding waveforms at runtime. Such logging is useful for debugging and tuning.
SREC is capable of creating N-Best results. When in N-Best mode, SREC produces a list of the top N results rather than just the most likely result.

| Parameter Name | Description | Typical Value(s) |
|---|---|---|
| `SREC.Recognizer.osi_log_level` | score based pruning threshold, higher means more accuracy and more cpu intensiveIndicates the type of logging to perform at runtime. OSI Log levels (bit set indicates level is ON)<br>0 no logging<br>BIT 0 -> BASIC logging<br>BIT 1 -> AUDIO waveform logging<br>BIT 2 -> DYNAMIC ADD WORD logging<br>e.g. value is 3 = BASIC+AUDIO logging, no ADDWORD | 4007 |
| `cmdline.DataCaptureDirectory` | Set to the full path that is used by SREC to save all logged data. | ../logs |

**Nametag Parameters**

| Parameter Name | Description | Typical Value(s) |
|---|---|---|
| `cmdline.nametagPath` | Specifies the base path for all nametags. For example, on some platforms, all nametags are saved to Flash ROM, so their base path is /dev/flash. | /dev/flash |
| `enableGetWaveform` | Must be enabled prior to a recognition in order for SR_RecognizerResultGetWaveform() to return a value; otherwise the function will return NULL. | FALSE |

## 6 PHONETIC REPRESENTATION

SREC uses two formats for phonemes. The short form, as stored in ".ok" dictionary files is used for on-platform representation and in grammar-compilation, with only a single character per phoneme. The long form is used to when editing in a text editor. The converter, pht_to_long.pl is used to convert short forms to long, the long form can be edited, then converted back to short using pht_to_short.pl. Alternatively, the ".ok" file can be edited directly. The converter is in: device/extlibs/srec/tools/cmd/

| SREC (long) | SREC (short) | Example |
|---|---|---|
| sil | # | silence (should not be used) |
| & | & | optional interword silence |
| AH | ) | c**ar**pet, h**ar**d (generally followed by /r/) |
| AE | , | c**a**re, sq**ua**re (generally followed by /r/) |
| ee | / | cit**y**, appr**e**c**i**ate (short) |
| ih | 6 | establ**i**sh, b**e**come |
| OY | < | **joi**n, t**o**ys |
| OW | ? | t**ow**n, all**ow**ed |
| uh | @ | **a**bout, **a**live, **a**rena (short) |
| EY | A | **a**ble, **a**ce, r**ai**se |
| ch | C | **ch**eat, **ch**air |
| dh | D | **th**e, bo**th**er |
| EE | E | b**ea**ch, t**ea**ch (long) |
| AY | I | n**i**ne, f**i**ve |
| ul | L | addition**al**, ab**le** |
| ng | N | taki**ng**, givi**ng** |
| OH | O | al**oh**a, th**ough** (stressed) |
| ur | P | amat**eur**, sist**er** (unstressed) |
| sh | S | **sh**ake |
| th | T | **th**in, bo**th** |
| OOH | U | ann**u**ity, b**oo**th |
| UR | V | w**or**th, iceb**er**g (stressed) |
| zh | Z | abra**s**ion |
| oh | ] | wind**ow**s, d**o**nation (unstressed) |
| AA | a | **a**valanche, **a**labaster |
| b | b | **b**rake |
| eh | c | **e**xpress, K**e**ntucky |

| SREC (long) | SREC (short) | Example |
|---|---|---|
| IH | i | cl**i**ff, p**i**nch |
| j | j | **j**ail, **ge**orge |
| k | k | **c**a**k**e |
| l | l | **l**ake, bow**l** |
| m | m | **m**onk, **m**e**m**es |
| n | n | **n**ame, ca**n** |
| AW | o | n**o**minal, b**o**mb |
| p | p | **p**an |
| OO | q | b**oo**k, g**oo**d |
| r | r | **r**ake |
| s | s | **s**ister |
| t | t | **t**ool |
| UH | u | beechn**u**t, bec**o**me (long) |
| v | v | **v**an |
| w | w | **w**e |
| y | y | **y**ou |
| z | z | bird**s**, bi**z**arre |
| AWH | { | bl**o**ssom, b**oa**rd |
| um | } | bott**om**, calci**um** |
| un | ~ | americ**an**, ambit**ion** |
| ENV | ^ | reserved for used as marker for slots |
| jnk | J | reserved for used as marker for slots |
| EH | e | obj**e**ct, **je**t, expr**e**ss |
| f | f | **f**reight |
| g | g | **g**oogol |
| h | h | **h**ead |
| d | d | **d**ad |

SREC uses special phonetic representations for digits and natural numbers. These should be checked in the dictionaries provided and should be re-used for words containing such words.

- Zero, oh, one, two, three, four, five, six, seven, eight, nine; 0, 1, 2, … 9
- Eleven, … nineteen; 11, 12, … 19
- Twenty, .. ninety; 20, 30, .. 90

**Using the phonetic representation conversion utilities (optional), from the 'device' directory:**

```
export ESRSDK=`pwd`/extlibs/srec
perl pht_to_long.pl -ok $ESRSDK/config/en.us/dictionary/large.ok -otxt $ESRSDK/config/en.us/dictionary/large.edit
perl pht_to_short.pl -i $ESRSDK/config/en.us/dictionary/large.edit -ok $ESRSDK/config/en.us/dictionary/large.ok
```

# 7 SRECTEST COMMAND FILE FORMAT

The command file (a.k.a. TCP file because of the .tcp extension) consists of lines containing commands to be executed by SRecTest. Each line consists of a command followed by zero or more parameters, or it contains a comment.

## Commands:

**context_load** *context_path* *context_id* *rule_name* *voice_enroll_indicator*
The context_load command loads a context into the recognizer for later use. This command is mandatory for executing any recognition commands. To activate this context, use the context_use command. The SRecTest program can store up to 4 contexts at a time.

The context path gives the location of the context.
The context_id is a unique identifier for this context. All context commands will refer to this id.
The rule_name is the rule that's activated when the context is activated.
The voice_enroll_indicator indicates whether or not this context is for voice enrollment or not. "ve" indicates a voice enrollment context. "not_ve" indicates the context is not for voice enrollment.

**context_use** *context_id*
The context_use command activates a previously loaded context. Any previously activated contexts must be de-activated with context_free before this context can be activated.

The context_id is a unique identifier for this context. It was set with context_load. All context commands will refer to this id.

**context_free** *context_id*
The context_free command de-activates the current active context. The context must be de-activated before another context can be activated.

The context_id is a unique identifier for this context. It was set with context_load. All context commands will refer to this id.

**context_unload** *context_id*
The context_unload command unloads a previously loaded context. The context must be de-activated, if it is active, before being unloaded.

The context_id is a unique identifier for this context. It was set with context_load. All context commands will refer to this id.

**recognize_nist** *file_name* *bos_time* *eos_time* *transcription(s)*
The recognize_nist command recognizes an audio file with a NIST header attached.

The file_name is the location of the audio file. This will be prepended with the -datapath that is passed on the command line.
The bos_time is the time in seconds for the beginning of speech. If not known, put 0. [NOT SUPPORTED]
The eos_time is the time in seconds for the end of speech. If not known, put 0. [NOT SUPPORTED]
The transcription(s) contains a space delimited set of words for the audio being recognized.

**recognize_pcm** *file_name* *bos_time* *eos_time* *transcription(s)*
The recognize_pcm command recognizes an audio file with straight pcm encoding and no header attached.

The file_name is the location of the audio file. This will be prepended with the -datapath that is passed on the command line.
The bos_time is the time in seconds for the beginning of speech. If not known, put 0. [NOT SUPPORTED]
The eos_time is the time in seconds for the end of speech. If not known, put 0. [NOT SUPPORTED]

The transcription(s) contains a space delimited set of words for the audio being recognized. This is the only place where spaces are allowed in a parameter.

**addwords_from_nametags** *slot*
This command enrolls the set of nametags for a slot into the currently active grammar.

The slot is the placeholder in the grammar where the grammar developer intends to add words or phrases.

**resetslots**
This command resets all of the slots in the active grammar to their default settings.

**addword** *slot* *word* *pronunciation* *weight* *sematic_tag*
This command adds a word to the active grammar.

The slot is the placeholder in the grammar where the grammar developer intends to add words or phrases.
The word is the text to be added.
The pronunciation is the phoneme string for the word.
The weight <Finish Me>. Set this to zero for no weight applied.
The semantic_tag is the key that will be returned from the recognizer for this word.

**context_compile**
This command compiles the currently active grammar. This command takes no parameters.

**context_save** *file_name*
This command saves the currently active context to a file.

The filename is the name of the file that the grammar will be saved to. I don’t know if any path will be prepended to this.

**addword_from_last_nametag** *slot* *nametag_id* *weight*
This command adds the specific nametag_id to a slot with the appropriate weight.

The slot is the placeholder in the grammar where the grammar developer intends to add words or phrases.
The nametag_id identifies the word being added.
The weight <Finish Me>. Set this to zero for no weight applied.

**load_nametags** *file_name*
This command loads a set of nametags that is stored in a file.

The filename is the name of the file that the nametags will be loaded from. I don’t know if any path will be prepended to this.

**save_nametags** *file_name*
This command saves a set of nametags to a file.

The filename is the name of the file that the nametags will be stored to. I don’t know if any path will be prepended to this.

**clear_nametags**
This command destroys the current set of nametags and creates a new empty set of nametags. This command takes no parameters.

**add_to_nametags** *nametag_id*
This command adds a nametag id to a set of nametags.

The nametag_id identifies the nametag set.

**acousticstate_load** *file_name*
This command loads the acoustic state that is stored in a file.

The file_name is the name of the file where the acoustic state data is loaded from.
.
**acousticstate_reset**
This command clears the acoustic state of the recognizer. This is mainly to run a test as if the waveforms that follow were presented at startup. This resets only the cepstral mean values, not the speech detection state values. This command takes no parameters.

**forced_rec** *mode*
This command sets the recognition mode in the recognizer.

The mode can be one of : on, meaning it is always on; off, meaning it is always off; one_time, meaning it is on once then turned off.

**#**
This is the comment indicator. It must begin the line. This line is skipped during processing.

**change_sample_rate** *new_rate*
This command sets the sample rate of the audio in the recognizer. This does not affect the audio device in any way.

The new_sample_rate can be one of 8000, 11025, 16000 or 22050.

**set_audio_size** *new_size*
This command sets the sample size of the audio in the recognizer. This affects how much audio is requested by the application from the file or audio device or audio file and how much data is passed to the recognizer.

The new_size can be in the range of 1 to 10240 bytes.