Vestec Automatic Speech Recognition Engine
Standard Edition
Version 1.1.1

Application Developer's Guide

Vestec Automatic Speech Recognition Engine
Standard Edition
Version 1.1.1
Application Developer's Guide

145 Columbia Street West, Suite 1, Waterloo, Ontario, Canada N2L 3L2

Table of Contents

About This Document

Vestec's Automatic Speech Recognition Engine (VASRE) is a speaker-independent speech recognition engine that supports a distributed architecture of servers and clients. VASRE works for Windows, GNU/Linux, and Open Solaris platforms to process an audio file or stream from external sources, such as telephone systems. The grammar can be built simply, with a list of keywords or pronunciations to be recognized, or with a more sophisticated industry standard.

This guide presents C data structure and Application Programming Interface (API) functions used to develop speech client applications.

Audience

This guide is intended for speech application developers who are using VASRE API to create client programs based on C or C++ programming languages.

Organization

This guide is organized as follows:

- Section 1 introduces the server-client architecture of VASRE and the general procedures of speech client applications.
- Section 2 describes the data structures defined for VASRE API.
- Section 3 explains the API functions for server port handling.
- Section 4 explains the API functions for grammar handling.
- Section 5 explains the API functions for audio processing.
- Section 6 explains the API functions for recognition result processing.
- Section 7 explains other API functions.

Conventions

Guides for VASRE use the following conventions:

- **Bold Arial** represents user utterances, recognized strings, and semantic results.
- `Courier New` represents file names, directory names, command line strings, and file contents.
- *Italic text* represents types, tokens, keywords, variables, and functions.
- Underlined text represents menu strings or texts in graphical user interface.
- *`Italic Courier New`* represents values replaced by you. For example, *`YYYY-MM-DD`* may represent a date in the year-month-day format.
- A paragraph starting with **N.B.** represents critical information or warning.
- For abbreviated terms, both singular and plural are spelled the same. For example, RM represents both resource manager and resource managers.

1 Introduction

The two main components of the VASRE distributed architecture are recognition servers and clients. The recognition server (henceforth server) is a permanent program unit dedicated to speech recognition. The client is a volatile program unit that bridges the server and external program units requiring speech recognition. A VASRE system generally has multiple servers. Under the VASRE architecture, each server can work for a single client. That is, it shouldn't be possible for there to be more active clients than there are servers.

For example, if you are going to develop a speech-enabled Interactive Voice Response (IVR) system, the client will be a program unit that activates the speech grammar on the server side according to the current context of a call session, forwards the speech data incoming from the caller to the server, and sends back the recognition results obtained from the server to the IVR. If the IVR system handles four simultaneous call sessions, you might have up to four active clients and hence should have four recognition servers.

The communication session between a server and a client is called a port. The client operation starts by opening a port towards an idle server. Since the server just connected to the client has no speech grammars, the first task of the client is generally adding grammars to the server.

The format of the VASRE grammar can be either text or binary. A grammar developer starts grammar development by writing a text grammar. The text grammar describes which words, phrases, or sentences VASRE should recognize from the given speech. If the text grammar is syntactically correct, the binary grammar can be generated via the compilation of the text grammar. Both text and binary grammars can be added to the server, but it is a better strategy to only add binary grammars, because adding text grammars to the server makes the server busier, and because you may lose the chance to fix any errors in the text grammar.

Each grammar added to the server has two fields, a tag and an activation token. The server may contain multiple grammars; all active grammars are used for speech recognition. For instance, if three grammars, `date`, `time`, and `money` were added to the server and `time` was set inactive, then both `date` and `money` will be used to recognize a given speech. The activeness of each grammar is controlled by the client, which should change the active grammars according to the current context of the call session.

Based on active grammars, the server processes the audio file, audio segment, or audio stream containing utterances to find the best words, phrases, or sentences matched. Multiple n-best alternatives, each of which is a set composed of: raw text recognized; the tag of the grammar used; confidence scores; and logical parsing, are available as members of the recognition result. The Semantic Interpretation (SI) processor can use the logical parsing to output the corresponding semantic result. If the client no longer requires speech recognition, it should close the port to release the busy server. When returning to the idle state, the server will internally clear out all the grammars it has.

All in all, the pseudo code of a typical client program can be outlined as follows (Note that the following is just a pseudo code and the actual function names are different):

```
port = Open_Port();
Add_Grammars(port);
WHILE audio processing is required DO
       Activate_Grammars(port);
       results = Process_Audio(port, audio);
       Output(results);
END_WHILE
Close_Port(port);
```

You can develop your own client programs using C or C++ programming languages and VASRE API, which provides a C header file `VasrLib.h` and two library files `libvasrlib.a` and `libVasrLib.so` for GNU/Linux and `VasrLib.lib` and `VasrLib.dll` for Windows. The location of the header file and library files are as follows:

- GNU/Linux:
  - `/usr/include/VestecASRE/VasrLib.h`
  - `/usr/lib/libvasrlib.a`
  - `/usr/lib/libVasrLib.so`
- Windows:
  - `VestecASRE\Include\VasrLib.h`
  - `VestecASRE\Lib\VasrLib.lib`
  - `VestecASRE\Lib\VasrLib.dll`

For your reference, the source code of two sample client applications is contained in the VASRE Standard Edition:

- GNU/Linux:
  - `/opt/VestecASRE/Samples/ConsoleApp/Src`
  - `/opt/VestecASRE/Samples/SAP/Src`
- Windows:
  - `VestecASRE\Samples\ConsoleApp\Src`
  - `VestecASRE\Samples\SAP\Src`

The rest of this document is dedicated to the explanation of the data structures and API functions. For further details, you may check the comments in `VasrLib.h` or `index.html` under `/usr/share/doc/VestecASRE/Api_reference/` for GNU/Linux and `VestecASRE\Doc\Api_reference\` for Windows. The VASRE Standard Edition also comes with the source code of two sample client applications for your reference.

N.B. Upon successful completion, most of the API functions presented in this guide return zero; otherwise, a non-zero error code will be returned. The full list of error codes is given in Appendix: List of VASRE API Error Codes. Pass the error code to *VA_ErrorCode2Msg* function to compose the associated error message.

2 Data Structures

Five C structure types are defined in `VasrLib.h`:

1. *HPORT*: Port information of a recognition server
2. *SAP*: Child structure of *HPORT* for streamed audio processing
3. *REC_RESULT*: Recognition results composed of one or more n-best alternatives
4. *NBEST_ALTER*: N-best alternative
5. *SEM_RESULT*: Semantic result of n-best alternative
6. *SRT_NODE*: Node of semantic result tree

This section describes the details on each structure type.

2.1 Type for Port Information

To acquire permission to use a particular server, declare a variable of the type *HPORT* and pass it to the *VA_OpenPort* function. If *VA_OpenPort* succeeds, the variable will have the valid port information of the connected server. If the address of the Resource Manager (RM) was wrongly specified or all the recognition servers are busy, *VA_OpenPort* will fail, returning an error code. *HPORT* is declared as follows:

```
struct HPORT
{
    int sd;                 /**< Variable for internal use */
    int sd0:                /**< Variable for internal use */
    int sd2;                /**< Variable for internal use */
    char ip_address[256];   /**< IP address of server */
    int port;               /**< Port number of server */
    int sfreq;              /**< Acceptable sampling frequency of audio */
    int samp_per_msec;      /**< Samples per msec (= sfreq/1000) */
    int min_speech_len;     /**< Acceptable minimum speech length in msec */
    int max_speech_len;     /**< Acceptable maximum speech length in msec */
    struct SAP sap;         /**< Structure for streamed audio processing */
};
```

sd , *sd0*, and *sd2* are used for internal purposes and you must not change the values of these variables from the client program. *ip_address* and *port* indicate the IP address and port number of the recognition server the client is being connected to. The values of these variables are decided by the RM while *VA_OpenPort* is running and you must not change the values. *sfreq*, *samp_per_msec*, *min_speech_len*, and *max_speech_len* represent sampling frequency, samples per millisecond, minimum speech length, and maximum speech length of audio the server will accept, respectively. The current version of VASRE supports the sampling frequency of 8 kHz only.

The default values of *min_speech_len* and *max_speech_len* are 300 and 10000, which respectively represent 0.3 seconds and 10 seconds. If you expect speech whose length is beyond this range, change the values of *min_speech_len* and *max_speech_len* using the function explained in Section 3.2. The allowed ranges of *min_speech_len* and *max_speech_len* are 0–1000 and 1000–20000, respectively.

If the length of input speech is shorter than *min_speech_len* or longer than *max_speech_len*, the audio

processing function will not perform speech processing, and will output the error code indicating the utterance is too short or too long. When the length of speech input in the streamed audio is longer than *max_speech_len*, the streamed audio processing function will return an error code indicating the speech is too long, but the recorded speech will be processed and the corresponding results will be output. See Section 5.2 for further details.

The last variable *sap* of type *SAP* is used to handle streamed audio. Details on *SAP* are described in the following subsection.

The address of the *HPORT* variable is used as the first argument of most API functions. To close the port and release the recognition server, invoke *VA_ClosePort* function.

N.B. Make sure your *HPORT* variable has been initialized by *VA_OpenPort* before you pass it to other API functions; an *HPORT* variable initially has a garbage value and the effect of calling the API functions with the *HPORT* variable uninitialized is unpredictable. Note also that once you call *VA_ClosePort* with the *HPORT* variable, the *HPORT* variable is no longer usable.

2.2 Type for Streamed Audio Processing

VASRE can take speech input from an audio file, from a buffer, or from streamed audio. For the first two cases, writing the client program is straightforward. After opening the port and adding grammars to the server, pass the audio file or buffer to audio processing API functions and the recognition results will be returned. To process an utterance in an audio stream, on the other hand, you should start streamed audio processing mode and keep sending segments of audio to the recognition server. After receiving each segment, the server will process it and let you know if the beginning or ending of speech is detected. This process is called voice activation detection or voice end detection. When voice end is detected, the server processes the speech recorded so far and returns the recognition result.

As a sub-structure of *HPORT*, *SAP* is defined to represent the parameters and state variables related to streamed audio processing. *SAP* is declared as follows:

```
enum SAP_STATE
{
    SAP_NO_SPEECH,      /**< No speech detected yet */
    SAP_BEGIN_SPEECH,   /**< Beginning of speech detected */
    SAP_RECORDING,      /**< Speech being recorded */
    SAP_END_SPEECH      /**< End of speech detected */
};

struct SAP
{
    int sap_on;             /**< if streamed audio processing is running */
    enum SAP_STATE state;   /**< State of streamed audio processing */
    int vad_amp_thres;      /**< Energy threshold for VAD */
    int vad_zc_thres;       /**< Zero-cross threshold for VAD */
    int vad_pta_thres;      /**< Pitch threshold of VAD */
    int vad_ptc_thres;      /**< Pitch count threshold of VAD */
    int vad_head_margin;    /**< Head margin of VAD in msec */
    int ved_tail_margin;    /**< Tail margin for VED in msec */
    int comp_max_amp;       /**< if computing maximum amplitude of speech */
    ...                     /**< Variables for internal use */
};
```

All the member variables of *SAP* are also managed by VASRE and you must not modify the values of

those variables directly from your program. *sap_on* indicates whether the streamed audio processor is running while state represents the state of the processor. See Section 5.2 for further description on these variables.

The seven tunable parameters of *SAP* are *vad_amp_thres*, *vad_zc_thres*, *vad_pta_thres*, *vad_ptc_thres*, *vad_head_margin*, *ved_tail_margin*, and *comp_max_amp*. You may change the values of these variables via the API functions introduced in Section 3.2. These variables are related to voice activation detection and voice end detection algorithms for streamed audio processing and may affect the recognition performance of your program. Other variables of *SAP* are used for internal purposes.

vad_amp_thres and *vad_zc_thres* represent the amplitude threshold and zero-cross threshold of voice activation detection, respectively. *vad_head_margin* is the time margin for voice activation detection. The audio signal can be represented by a waveform whose amplitude ranges from -32,768 to 32,767. If the absolute amplitude of the audio exceeds *vad_amp_thres*, then the streamed audio processor sets the internal timer to zero and starts counting the number of zero crosses, which are defined as the events that the sign of audio signal switches between positive and negative. If the number of zero crosses reaches *vad_zc_thres* before *vad_head_margin* elapses, the streamed audio processor starts recording audio as voice input. The past audio samples of *vad_head_margin* length are prepended to the recorded voice input.

The default values of *vad_amp_thres* and *vad_zc_thres* are 800 and 10. These default values should work well for a clean environment, where the noise level is very low. If the noise level is high, you should make the streamed audio process less sensitive to noise by increasing the values of *vad_amp_thres* and *vad_zc_thres*. The allowed ranges of *vad_amp_thres* and *vad_zc_thres* are 200–30000 and 3–200, respectively. The default value and allowed range of *vad_head_margin* are 300 and 150–500, respectively.

To avoid detecting high-amplitude noise as speech input, the streamed audio processor has another voice activation detection mechanism using pitch. The pitch threshold and pitch count threshold, *vad_pta_thres* and *vad_ptc_thres*, are used for this mechanism. The default values of these variables are 400 and 7, which work well for most cases. The allowed ranges of *vad_pta_thres* and *vad_ptc_thres* are 100–500 and 1–10, respectively. Try to increase the values of these two variables if the streamed audio processor is too sensitive to noise.

ved_tail_margin indicates how long the void recording will go on even when the amplitude of speech signal goes below *vad_amp_thres* and the number of zero crosses for past *vad_head_margin* is less than *vad_zc_thres*. As *ved_tail_margin* increases, the streamed audio processor will wait longer for the subsequent speech segment. At the same time, the processing time of the streamed audio processor will get slow a little, because the recorded speech is processed as soon as the recording is done. The default value 0.5 seconds of *ved_tail_margin* should be fine for most cases. The allowed range is 200–1200.

comp_max_amp is a flag representing whether the streamed audio process should calculate the maximum amplitude of the recorded speech while processing it. The default value of *comp_max_amp* is one. If the value is non-zero, the maximum amplitude is calculated and returned to the client as a part of recognition results. The maximum amplitude is generally used to reject the echoing of TTS voice. By setting *vad_amp_thres* higher than the maximum amplitude of echo, it can be effectively rejected. See Asterisk Integration Guide for further details.

2.3 Types for Recognition Results

After processing audio, the server returns zero or more recognized texts sorted according to their scores. The maximum allowed number of the recognized texts is called n-best (this parameter can be set when starting recognition servers) and the set of details regarding a recognized text is called an n-best

alternative. So, a recognition result can be regarded as a list of zero or more n-best alternatives. VASRE adopts two structure types *REC_RESULT* and *NBEST_ALTER* to represent a recognition result and an n-best alternative, respectively.

REC_RESULT is declared as follows:

```
struct REC_RESULT
{
    double cpu_sec;              /**< CPU seconds taken */
    int max_amp;                 /**< Maximum amplitude of speech */
    struct NBEST_ALTER* nbest;   /**< Linked list of n-best alternatives */
};
```

cpu_sec represents the time spent to process the speech input. *max_amp* is the maximum amplitude of the processed speech. This variable is used to report the maximum amplitude of echo falsely accepted as speech input during streamed audio processing. If *comp_max_amp* of *SAP* is set to zero, the maximum amplitude computation of the speech will be skipped and *max_amp* will be zero. See Section 2.2 for further details.

nbest is the header element of the linked list representing zero or more n-best alternatives. The server sorts the elements of the linked list in the descending order of the confidence score; that is, the first element has the highest confidence, the second the second highest, and so on. You can take each n-best alternative by traversing the linked list yourself or using the API functions mentioned in Section 6.2. The server may set the value of *nbest* to NULL when it fails to find raw texts matched to the grammar.

N.B. The server of VASRE SE v1.1 and higher performs memory allocation to build the *nbest* link list. It is your responsibility to release the memory by calling API functions explained in Section 6.1.

The type of *nbest* is the pointer of *NBEST_ALTER*, which is defined as:

```
Enum TAG_FORMAT
{
    TAG_FORMAT_SL,  /**< SI String Literals */
    TAG_FORMAT_SC,  /**< SI Script */
    TAG_FORMAT_SE10 /**< Binary grammar generated by VASRE SE v1.0.x */
};

struct NBEST_ALTER
{
    char* gram_tag;              /**< Sentence string */
    unsigned short num_words;    /**< Number of recognized words */
    char** words;                /**< Word string */
    short* score;                /**< Confidence score of each word */
    enum TAG_FORMAT tf;          /**< Tag format */
    char* logic_parse;           /**< Logical parse for semantic result */
    struct NBEST_ALTER* next;    /**< Next n-best alternative */
};
```

Basically, an n-best alternative is a set of grammar tag (*gram_tag*), raw text (*words*), score (*score*), and logical parsing (*logic_parse*). The logical parsing is a formal syntax for describing the sequence and relation of tags and rule references to the tokens that are input to the grammar processor. Based on the tag statements given in the grammar and recognized raw text, the server outputs the logical parsing and corresponding tag format. The tag format is the way to interpret the logical parsing to generate a semantic result as described in Section 6.4.

Three tag formats may be output by the server:

- *TAG_FORMAT_SL* corresponds to SI string literals format defined in Semantic Interpretation for Speech Recognition (SISR) version 1.0, a W3C standard for semantic interpretation.
- *TAG_FORMAT_SC* represents SI script format defined in SISR version 1.0.
- *TAG_FORMAT_SE10* means the tag format used for VASRE SE version1.0. See the guide of SE version 1.0 for further details on this format.

Note that *TAG_FORMAT_SE10* is not output when you add text grammars to the server. The available tag formats for text grammars are only SI string literals and SI script. *TAG_FORMAT_SE10* is supported for backward compatibility only when you add binary grammars compiled by `GramGen` of VASRE SE version 1.0. See Grammar Developer's Guide for further details on tag formats.

Don't confuse *gram_tag* with the tags for SI. *gram_tag* simply represents the identification of the grammar, which is the path name of the grammar file by default. The server allocates memory for the NULL-terminated string *gram_tag*. You may release the memory for *gram_tag* manually. But, it is highly recommended to use the API function described in Section 6.1. to release the whole memory allocated for *REC_RESULT*.

num_words represents the number of words comprising raw recognized text. *words[0]*, ..., *words[num_words-1]* points to NULL-terminated strings representing the words in the order of appearance. *score[0]*, ..., *score[num_words-1]* are the corresponding confidence score of words. Use the API function described in Section 6.1. to release the memory.

next points to the address of the next n-best alternative.

2.4 Types for Semantic Results

You may call the SI processing function described in Section 6.4 with an n-best alternative to get the associated semantic result. To represent the semantic result, use *SEM_RESULT* declared as follows:

```
struct SEM_RESULT
{
    struct SRT_NODE *root;  /**< Root node */
};
```

SEM_RESULT has only one member *root*, which points to the root node of the tree representing the semantic result.

The structure type declared for the semantic result tree node is *SRT_NODE*:

```
enum SRT_NODE_TYPE
{
    SRT_NODE_TYPE_STR = 1,  /**< String type */
    SRT_NODE_TYPE_INT,      /**< Integer type */
    SRT_NODE_TYPE_DOUBLE,   /**< Double type */
    SRT_NODE_TYPE_BOOL,     /**< Boolean type */
    SRT_NODE_TYPE_OBJECT,   /**< Object type */
    SRT_NODE_TYPE_ARRAY     /**< Array type */
};

struct SRT_NODE
{
    char* name;                 /**< Name */
    enum SRT_NODE_TYPE type;    /**< Type */
    char* value;                /**< Value */
    struct SRT_NODE* child;     /**< Child node */
    struct SRT_NODE* sib;       /**< Sibling node */
};
```

The main properties of *SRT_NODE* are *name*, *type*, and *value*. *child* and *sib* are used to describe the tree structure of multiple *SRT_NODE* variables. The type of SRT_NODE can be one of the following: string, integer, double, bool, object, and array.

If the type of the node A is neither object nor array, *value* of A is a nonempty string and *child* of A is NULL. If the type of the node B is object or array, *value* of B is NULL and *child* of B points to other node.

For example, consider the following speech input ordering pizzas and drinks:

I'd like a large coke and two small pizzas with anchovies and pepperoni

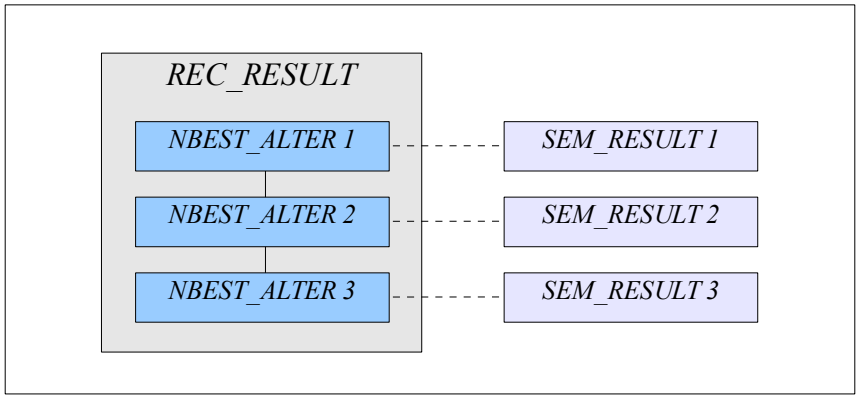
The semantic result tree would have the following hierarchy (*name*, *type*, and *value* of nodes are separated by a vertical bar within square brackets):

```
+["order" | Object | ""]
|
+---+["drink" | Object | ""]
|   |
|   +----["liquid" | String | "coke"]
|   +----["drinksize" | String | "large"]
|
+---+["pizza" | Object | ""]
    |
    +----["pizzasize" | String | "small"]
    +----["number" | Integer | "2"]
    +---+["topping" | Array | ""]
        |
        +----["0" | String | "anchovies"]
        +----["1" | String | "pepperoni"]
```

In this example, the root node “order” has two child nodes “drink” and “pizza”. To describe this relationship, *child* of “order”points to “drink” and *sib* of “drink” points to “pizza”. *sib* of “order” and “pizza” are NULL. Again, *child* of “drink” points to “liquid” and *sib* of “liquid” points to “drinksize”. *child* of pizza points to “pizzasize”, *sib* of “pizzasize” to “number”, and *sib* of “number” to “topping”. *child* of “toppings” points to the first element “0” and *sib* of “0” to “1”.

Even if the type of a node is integer, double, and bool, its value is represented by a NULL-terminated string *value*. To obtain actual integer or double value, you should process *value*. *value* of bool variable is either “true” or “false”.

Note again that *SEM_RESULT* object is paired with *NBEST_ALTER* object, not with the *REC_RESULT* object. The relationship amongst these three structure types are illustrated below:



3 Port Handling Functions

All the tasks performed by the client program use the port connected to the recognition server. This section presents the API functions related to port handling. Section 3.1 explains the API functions to control the port while Section 3.2 describes the way to change the values of port parameters. Section 3.3 reviews a function to get information of VASRE system.

3.1 Opening and Closing Port

Two functions are used to open and close a server port:

- *VA_OpenPort*: Open port toward recognition server
- *VA_ClosePort*: Close port from recognition server

```
int VA_OpenPort(
    const char* rm_ip,  /**< [in] IP address of resource manager */
    struct HPORT* hport /**< [out] Pointer to HPORT variable */
    );
int VA_ClosePort(
    struct HPORT* hport /**< [in] Pointer to HPORT variable */
    );
```

To open a port, call *VA_OpenPort* with a string representing the IP address of the RM, and the address of a *HPORT* variable. If the *HPORT* variable is successfully initialized with the valid port information, this function will return zero. Otherwise, a non-zero error code will be returned.

To close the port and release the connected server, call *VA_ClosePort*. This function internally removes all the grammars added so far from the server to reinitialize the server ready for future client connection. The following example demonstrates the typical usage of *VA_OpenPort* and *VA_ClosePort*.

```
#include <VestecASRE/VasrLib.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
    char* rm_ip = "127.0.0.1";
    struct HPORT hport;
    int ret = VA_OpenPort(rm_ip, &hport);
    if(ret != 0)
    {
        printf("  %s\n\n", VA_ErrorCode2Msg(ret));
        return ret;
    }
    // do something with hport
    // ...
    VA_ClosePort(&hport);
}
```

Once *VA_ClosePort* has been invoked, the client program should call *VA_OpenPort* again to work with the recognition server.

N.B. If VASRE API functions experience internal errors, they may call *VA_ClosePort* internally to avoid

system crash. If the API functions fail because of an error from the client program, they do not call *VA_ClosePort*. To see if VA_ClosePort has been called upon failure, check the value of *sd* or *sd2* of the *HPORT* variable. If the value is -1, *VA_ClosePort* has been invoked internally. You may simply call *VA_ClosePort* whenever the API function fails, because subsequent calls of *VA_ClosePort* are permitted and have no effect.

3.2 Setting Port Parameters

You may change the values of *HPORT* member variables using the following functions:

- *VA_SetMinSpeechLen*: Change the value of *min_speech_len* of *HPORT*, the minimum speech length accepted by the server
- *VA_SetMaxSpeechLen*: Change the value of *max_speech_len* of *HPORT*, the maximum speech length accepted by the server
- *VA_SetVadAmpThres*: Change the value of *vad_amp_thres* of *SAP*, the energy threshold for voice activation detection
- *VA_SetVadZcThres*: Change the value of *vad_zc_thres* of *SAP*, the zero-cross threshold for voice activation detection
- *VA_SetPtaThres*: Change the value of *vad_pta_thres* of *SAP*, the pitch threshold for voice activation detection
- *VA_SetPtcThres*: Change the value of *vad_ptc_thres* of *SAP*, the pitch count threshold for voice activation detection
- *VA_SetVadHeadMargin*: Change the value of *vad_head_margin* of *SAP*, the head margin of audio for voice activation detection
- *VA_SetVedTailMargin*: Change the value of *ved_tail_margin* of *SAP*, the tail margin of audio for voice end detection
- *VA_SetCompMaxAmp*: Set or reset *comp_max_amp* of *SAP*, the flag indicating whether maximum amplitude of processed audio will be output

```
int VA_SetMinSpeechLen(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Minimum speech length in msec */
    );
int VA_SetMaxSpeechLen(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Maximum speech length in msec */
    );
int VA_SetVadAmpThres(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Amplitude threshold for VAD */
    );
int VA_SetVadZcThres(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Zero-cross threshold for VAD */
    );
int VA_SetVadPtaThres(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Pitch threshold for VAD */
    );
int VA_SetVadPtcThres(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Pitch count threshold for VAD */
    );
int VA_SetVadHeadMargin(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Head margin of audio for VAD in msec */
    );
int VA_SetVedTailMargin(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] Tail margin of audio for VED in msec */
    );
int VA_SetCompMaxAmp(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    int value               /**< [in] If maximum amplitude will be output */
    );
```

You must not change the values of *HPORT* variables directly from your program. The default values and allowed ranges of these tunable variables are listed in the following table:

Variable	Default Value	Minimum Value	Maximum Value
min_speech_len	300	0	1000
max_speech_len	10000	1000	20000
vad_amp_thres	1000	200	30000
vad_zc_thres	10	3	200
vad_pta_thres	400	100	500
vad_ptc_thres	7	1	10
vad_head_margin	300	150	500
ved_tail_margin	500	200	1200
comp_max_amp	0 (OFF)	0 (OFF)	1 (ON)

More details on each variable are described in Section 2.1 and 2.2.

3.3 Getting System Information

You may obtain version and license information and server status using *VA_GetInfo* function.

```
enum VINFO
{
    INFO_VERSION = 1,    /**< Version of VASRE */
    INFO_LICENSE,        /**< Number of licenses purchased and being used */
    INFO_SERVERS         /**< Server status */
};

int VA_GetInfo(
    const char* rm_ip,           /**< [in] IP address of resource manager */
    const enum VINFO info_type, /**< [in] Type of information */
    char* info_str,              /**< [out] String representing information */
    const size_t info_str_size  /**< [in] Size of <em>info_str</em> */
    );
```

Prepare an array of *char* and pass it as the third argument of *VA_GetInfo*. The size of the array must be passed as the fourth argument. If *info_str_size* is too small compared to the actual information string, this function outputs the truncated string.

VA_GetInfo returns zero if it successfully obtained the *info_str*; otherwise, a non-zero error code is returned.

4 Grammar Handling Functions

The recognition server maintains a list of grammars. When the client opens the server port, this list is empty. The client can add grammars to this list as long as the sum of the grammar vocabulary sizes is less than the limit encrypted in the license file (by default, this limit is 500). See Section 14 of Grammar Developer's Guide for further details. Each grammar in the list has two fields, tag and activeness. The tag is used as identification and all the active grammars in the list are used for speech recognition. The grammars are automatically deleted when the client program closes the port. Since adding each grammar takes some time, adding all the grammars required for a speech application and activating a portion of them is a good strategy.

This section presents the API functions that add grammars to the server, delete grammars from the server, replace existing grammars with new ones, and activate or deactivate existing grammars. Section 4.1 explains the API functions related to grammar addition while Section 4.2 and 4.3 describe the API functions for grammar deletion and replacement, respectively. Section 4.4 lists the functions for grammar activation and deactivation.

4.1 Adding Grammars

The three grammar formats acceptable to the recognition server are a text file, a binary file, and a Speech Recognition Grammar Specification (SRGS) text. The client program adds grammars to the server using the following functions:

- *VA_AddGramGout*: Add .gout grammar to the recognition server
- *VA_AddGramGout2*: Add `.gout` grammar to the recognition server with tag
- *VA_AddGramGrm*: Add `.grm` grammar to the recognition server
- *VA_AddGramGrm2*: Add `.grm` grammar to the recognition server with tag
- *VA_AddGramStr*: Add SRGS text to the recognition server

```
int VA_AddGramGout(
    struct HPORT* hport,        /**< [in] Pointer to HPORT variable */
    const char* gout_path_name  /**< [in] Path name of binary grammar file (.gout) */
    );
int VA_AddGramGout2(
    struct HPORT* hport,        /**< [in] Pointer to HPORT variable */
    const char* tag,            /**< [in] Grammar tag */
    const char* gout_path_name  /**< [in] Path name of binary grammar file (.gout) */
    );
int VA_AddGramGrm(
    struct HPORT* hport,        /**< [in] Pointer to HPORT variable */
    const char* grm_path_name,  /**< [in] Path name of text grammar file (.grm) */
    char* srgs_err_msg,         /**< [out] Error message from grammar compiler */
    const size_t em_size        /**< [in] Size of srgs_err_msg */
    );
int VA_AddGramGrm2(
    struct HPORT* hport,        /**< [in] Pointer to HPORT variable */
    const char* tag,            /**< [in] Grammar tag */
    const char* grm_path_name,  /**< [in] Path name of text grammar file (.grm) */
    char* srgs_err_msg,         /**< [out] Error message from grammar compiler */
    const size_t em_size        /**< [in] Size of srgs_err_msg */
    );
int VA_AddGramStr(
    struct HPORT* hport,        /**< [in] Pointer to HPORT variable */
    const char* tag,            /**< [in] Grammar tag */
    const char* srgs_str,       /**< [in] Pointer to null terminated SRGS text */
    char* srgs_err_msg,         /**< [out] Error message from grammar compiler */
    const size_t em_size        /**< [in] Size of srgs_err_msg */
    );
```

The binary grammar files, whose extension is `.gout`, can be added using *VA_AddGramGout* and *VA_AddGramGout2*. Either the absolute or the relative path name of a `.gout` file can be used as an argument of these functions. The text grammar files whose extension is `.grm` can be added using *VA_AddGramGrm* and *VA_AddGramGrm2*. The SRGS grammar text can be added using *VA_AddGramStr*. Note that the recognition server will compile the text grammars added by *VA_AddGramGrm*, *VA_AddGramGrm2*, and *VA_AddGramStr*.

If a syntax error is found in SRGS text, *VA_AddGramGrm*, *VA_AddGramGrm2*, and *VA_AddGramStr* return a non-zero error code and copy the associated NULL-terminated error message to *srgs_err_msg*. To get the error message, pass an array of *char* and its size as the last two parameters of those functions. If the size is less than the actual error message, it will be truncated.

The grammar tag can be specified when adding the grammars using and *VA_AddGramGout2*, *VA_AddGramGrm2*, and *VA_AddGramStr*. When adding grammars using *VA_AddGramGout* and *VA_AddGramGrm*, the tags will be set to the path name of the grammar files. The tag of a grammar is used as the identifier when deleting or activating the grammar.

The following table shows which function you should use for specific purposes:

	Adding .gout	Adding .grm	Adding SRGS text
Adding without tag	*VA_AddGramGout*	*VA_AddGramGrm*	N/A
Adding with tag	*VA_AddGramGout2*	*VA_AddGramGrm2*	*VA_AddGramStr*

The added grammars will be active by default. To activate some of the added grammars selectively, use

the API functions explained in Section 4.4. You may remove the added grammars from the recognition server using the API functions explained in Section 4.2.

Upon successful completion, these functions return zero; otherwise, a non-zero error code will be returned.

4.2 Deleting Grammars

Delete the added grammars using the following API functions:

- *VA_DelGram*: Delete a grammar from recognition server
- *VA_DelAllGram*: Delete all the grammars from recognition server

```
int VA_DelGram(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    const char* tag         /**< [in] Tag of grammar to be removed */
    );
int VA_DelAllGram(
    struct HPORT* hport     /**< [in] Pointer to HPORT variable */
    );
```

Call *VA_DelGram* to delete a single grammar using its tag name. To delete all the grammars from the server, use *VA_DelAllGram*. Upon successful completion, these functions return zero; otherwise, a non-zero error code will be returned.

In general, the use of grammar deletion functions is very limited since the added grammars are automatically deleted during grammar replacement or port closing, which are respectively described in Section 4.3 and 3.1.

4.3 Replacing Grammars

To delete all the existing grammars from the server and add a new grammar, use the grammar replacement function listed below:

- *VA_ReplaceGramGout*: Replace existing grammars with .gout grammar
- *VA_ReplaceGramGout2*: Replace existing grammars with `.gout` grammar and tag
- *VA_ReplaceGramGrm*: Replace existing grammars with `.grm` grammar
- *VA_ReplaceGramGrm2*: Replace existing grammars with `.grm` grammar and tag
- *VA_ReplaceGramStr*: Replace existing grammars with SRGS grammar

```
int VA_ReplaceGramGout(
    struct HPORT* hport,         /**< [in] Pointer to HPORT variable */
    const char* gout_path_name   /**< [in] Path name of binary grammar file (.gout) */
    );
int VA_ReplaceGramGout2(
    struct HPORT* hport,         /**< [in] Pointer to HPORT variable */
    const char* tag,             /**< [in] Grammar tag */
    const char* gout_path_name   /**< [in] Path name of binary grammar file (.gout) */
    );
int VA_ReplaceGramGrm(
    struct HPORT* hport,         /**< [in] Pointer to HPORT variable */
    const char* grm_path_name,   /**< [in] Path name of text grammar file (.grm) */
    char* srgs_err_msg,          /**< [out] Error message from grammar compiler */
    const size_t em_size         /**< [in] Size of srgs_err_msg */
    );
int VA_ReplaceGramGrm2(
    struct HPORT* hport,         /**< [in] Pointer to HPORT variable */
    const char* tag,             /**< [in] Grammar tag */
    const char* grm_path_name,   /**< [in] Path name of text grammar file (.grm) */
    char* srgs_err_msg,          /**< [out] Error message from grammar compiler */
    const size_t em_size         /**< [in] Size of srgs_err_msg */
    );
int VA_ReplaceGramStr(
    struct HPORT* hport,         /**< [in] Pointer to HPORT variable */
    const char* tag,             /**< [in] Grammar tag */
    const char* srgs_str,        /**< [in] Pointer to null terminated SRGS text */
    char* srgs_err_msg,          /**< [out] Error message from grammar compiler */
    const size_t em_size         /**< [in] Size of srgs_err_msg */
    );
```

The grammar replacement functions can be regarded as a combination of the grammar deletion function and grammar addition functions. For example, the call of *VA_ReplaceGramGout* is identical to the subsequent calls of *VA_DelAllGram* and *VA_AddGramGout*.

4.4 Activating Grammars

Use the following functions to activate or deactivate the added grammar:

- *VA_ActivateThisGramOnly*: Activate specified grammar and deactivate all other grammars
- *VA_ActivateGram*: Activate specified grammar
- *VA_DeactivateGram*: Deactivate specified grammar

```
int VA_ActivateThisGramOnly(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    const char* tag         /**< [in] Tag of grammar to be solely activated */
    );
int VA_ActivateGram(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    const char* tag         /**< [in] Tag of grammar to be activated */
    );
int VA_DeactivateGram(
    struct HPORT* hport,    /**< [in] Pointer to HPORT variable */
    const char* tag         /**< [in] Tag of grammar to be deactivated */
    );
```

Call *VA_ActivateGram* and *VA_DeactivateGram* to activate or deactivate a grammar in the server list. If the server has many grammars, switching activeness of each grammar can be a tedious task. To avoid this, VASRE API also provides *VA_ActivateThisGramOnly*, which activates the specified grammar while deactivating any others in the server grammar list.

5 Audio Processing Functions

With VASRE, you can process either batch audio or streamed audio. Batch audio is an audio file or segment containing one speech segment ready to be processed by the server. Streamed audio is a continuous signal that may contain multiple speech segments intermittently. For your convenience, voice activation detection algorithms are implemented inside the streamed audio processor of VASRE.

5.1 Processing Audio in File or Buffer

VASRE provides the following API functions for batch audio processing:

- *VA_ProcessWave*: Process a specified wave file
- *VA_ProcessBuffer*: Process audio in a buffer

```
int VA_ProcessWave(
    struct HPORT* hport,            /**< [in] Pointer to HPORT variable */
    const char* wav_path_name,      /**< [in] Path name of wave file to be processed */
    struct REC_RESULT* rec_result   /**< [out] Pointer to REC_RESULT variable */
    );
int VA_ProcessBuffer(
    struct HPORT* hport,            /**< [in] Pointer to HPORT variable */
    short* const buffer,            /**< [in] Pointer to 16-bit audio samples */
    const size_t buffer_size,       /**< [in] Size of buffer */
    struct REC_RESULT* rec_result   /**< [out] Pointer to REC_RESULT variable */
    );
```

To process the wave file with a header, use *VA_ProcessWave* function. You can pass the absolute or relative path name of an 8 kHz `.wav` file as the second argument of *VA_ProcessWave*. This function reads the header of the wave file to check the sampling frequency. Therefore, if the wave file is header-less or its sampling frequency is something other than 8 kHz, this function will return an error code without performing speech recognition. If you need to process a header-less audio file, copy its content to a buffer and pass it to *VA_ProcessBuffer*, which processes the audio stored in the array of 16-bit audio samples. *VA_ProcessBuffer* never checks the sampling frequency of the audio and it is your responsibility to make sure the buffer contains 8 kHz audio.

Declare a variable of type *REC_RESULT* and pass its address as the last parameter of *VA_ProcessWave* and *VA_ProcessBuffer*. Once these functions returns zero, *REC_RESULT* will contain the recognition results. Note that even when this function returns zero, *REC_RESULT* may contain no n-best alternatives, which means the server found no recognition result.

For your reference, the source code of the sample application performing batch audio recognition is available at `/opt/VestecASRE/Samples/ConsoleApp/Src/` for GNU/Linux and `\VestecASRE\Samples\ConsoleApp\Src\` for Windows.

5.2 Processing Audio in Stream

VASRE provides the following API functions for streamed audio processing:

- *VA_StartStream*: Start streamed audio processing mode

- *VA_StopStream*: Stop streamed audio processing mode
- *VA_ProcessStream*: Process audio samples during streamed audio processing mode
- *VA_ProcessStream1k*: Process one thousand samples during streamed audio processing mode

```
int VA_StartStream(
    struct HPORT* hport         /**< [in] Pointer to HPORT variable */
    );
int VA_StopStream(
    struct HPORT* hport         /**< [in] Pointer to HPORT variable */
    );
int VA_ProcessStream(
    struct HPORT* hport,            /**< [in] Pointer to HPORT variable */
    short* const buffer,            /**< [in] Pointer to 16-bit audio samples */
    const size_t buf_size,          /**< [in] Number of audio samples */
    struct REC_RESULT* rec_result   /**< [out] Pointer to REC_RESULT variable */
    );
int VA_ProcessStream1k(
    struct HPORT* hport,            /**< [in] Pointer to HPORT variable */
    short* const buffer,            /**< [in] Pointer to 1000 audio samples */
    struct REC_RESULT* rec_result   /**< [out] Pointer to REC_RESULT variable */
    );
```

To process streamed audio data coming from an external source, use the combination of the above functions. The first two functions start and stop the Streamed Audio Processor (SAP), which is a state machine managed by the recognition server. The last two functions pass the segment of streamed audio to the SAP.

VA_StartStream initializes the SAP by allocating memory for *sap* of *HPORT* variable and assigning default values. *VA_StopStream* closes the SAP by freeing the memory allocated for *sap* of *HPORT* variable. To avoid memory leakage, the ordered pair of *VA_StartStream* and *VA_StopStream* should be called within the client program.

Between the two function calls, the client program iteratively calls *VA_ProcessStream* or *VA_ProcessStream1k* to pass the segments of streamed audio to the SAP. The two functions are basically the same except that you may specify the audio segment size for the former. Each call of *VA_ProcessStream* or *VA_ProcessStream1k* switches the state of the SAP amongst *SAP_NO_SPEECH*, *SAP_BEGIN_SPEECH*, *SAP_RECORDING*, and *SAP_END_SPEECH*, which are defined as members of *enum SAP_STATE*. The client program should monitor the state of the SAP with *sap.state* of the *HPORT* variable. See Section 2.2 for the details of *SAP* structure type.

The SAP starts with *SAP_NO_SPEECH*, which means no speech is detected yet. As soon as the SAP detects the beginning of a speech from the segment subsequently given by *VA_ProcessStream* or *VA_ProcessStream1k*, the state changes to *SAP_BEGIN_SPEECH* and the SAP starts passing the segment to the recognition server. The state will then change to *SAP_RECORDING* and the SAP will keep forwarding upcoming audio segments to the server until the end of speech is detected. As soon as the SAP detects the end of the speech, the state changes to *SAP_END_SPEECH* and the recognition results found by the server are copied to *REC_RESULT* variable. And then, the state returns to *SAP_NO_SPEECH* or *SAP_BEGIN_SPEECH* for the next call of *VA_ProcessStream* or *VA_ProcessStream1k*.

All in all, the following C code fragment shows the typical usage of streamed audio processing functions:

```
#include <VestecASRE/VasrLib.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
    char* rm_ip = "127.0.0.1";
    struct HPORT hport;
    struct REC_RESULT result;

    int ret = VA_OpenPort(rm_ip, &hport);
    if(ret != 0)
    {
        printf("  %s\n\n", VA_ErrorCode2Msg(ret));
        return ret;
    }
    VA_StartStream(hport);
    while(1)
    {
        int16_t* rcvd = read_1000_samples_from_audio_source();
        if(rcvd == NULL)
            break;
        ret = VA_ProcessStream1k(hport, rcvd, &result);
        if(ret != 0)
            printf("  %s\n\n", VA_ErrorCode2Msg(ret));
        if(hport->sap.state == SAP_END_SPEECH)
        {
            output_result(result);
            VA_FreeRecResult(&result);
        }
    }
    VA_StopStream(hport);
    VA_ClosePort(&hport);
}
```

The above example uses *VA_ProcessStream1k*, which passes 1000 samples to the server at a time. If you need to process an audio segment whose size is less than 1000, use *VA_ProcessStream* specifying the segment size. If you have an audio segment whose size is bigger than 1000, you should chop it into several pieces and call *VA_ProcessStream1k* or *VA_ProcessStream* multiple times for each piece. The minimum allowed segment size for *VA_ProcessStream* is one, but try to pass a segment of at least several hundred, because sending too-small segments to the SAP will degrade recognition accuracy.

You can control the sensitivity of voice activation detection and voice end detection using the functions explained in Section 3.2.

For your reference, the sample application performing streamed audio processing is available at `/opt/VestecASRE/Samples/SAP/` for GNU/Linux and `\VestecASRE\Samples\SAP\` for Windows.

6 Recognition Results Processing Functions

After obtaining the *REC_RESULT* object from the audio processing function, you should process it to take n-best alternatives and transform them into semantic results. This section reviews the functions related to the processing of *REC_RESULT* and *SEM_RESULT* objects.

6.1 Initializing Recognition Result Structure

VASRE provides the following API functions for construction and destruction of *REC_RESULT* variables:

- *VA_InitRecResult*: Initialize recognition result structure
- *VA_FreeRecResult*: Release memory allocated for recognition result structure

```
int VA_InitRecResult(
    struct REC_RESULT* rec_result   /**< [in] Pointer to recognition result */
    );
int VA_FreeRecResult(
    struct REC_RESULT* rec_result   /**< [in] Pointer to recognition result */
    );
```

Call *VA_InitRecResult* to initialize a *REC_RESULT* variable. This function assigns zero to all the members of the *REC_RESULT* variable. *VA_FreeRecResult* releases the memory allocated for the *REC_RESULT* variable. *REC_RESULT* variable output by audio processing functions must be released by *VA_FreeRecResult* to avoid memory leakage.

These functions return an error code if *res_result* is NULL; otherwise, zero is returned.

6.2 Obtaining N-best Alternatives

VASRE provides the following API functions for direct access of n-best alternatives:

- *VA_GetNumAlterFromRecResult*: Get number of n-best alternatives
- *VA_GetNbestAlterFromRecResult*: Get an n-best alternative

```
int VA_GetNumAlterFromRecResult(
    struct REC_RESULT* rec_result   /**< [in] Pointer to REC_RESULT variable */
    );
struct NBEST_ALTER* VA_GetNbestAlterFromRecResult(
    struct REC_RESULT* rec_result,  /**< [in] Pointer to REC_RESULT variable */
    size_t index                    /**< [in] Index of n-best alternative */
    );
```

Call *VA_GetNumAlterFromRecResult* to get the number of n-best alternatives of a recognition result. This functions counts the elements in the linked list representing n-best alternatives in *REC_RESULT* structure and returns the count. If *rec_result* is NULL or corrupted, -1 is returned.

To get an n-best alternative from the recognition result, call *VA_GetNbestAlterFromRecResult* with the zero-based index of the n-best alternative to be taken. For example, if the index is zero, this function

returns the n-best alternative with the highest confidence score; index one represents the n-best alternative with the second highest confidence and so on. If the given REC_RESULT variable is corrupted or *index* is no less than the number of the n-best alternatives, *VA_GetNbestAlterFromRecResult* returns NULL. Note that this function never allocates memory for *NBEST_ALTER* and you may not free the returned pointer. *VA_GetNbestAlterFromRecResult* simply returns the address of existing NBEST_ALTER structure in the REC_RESULT structure. This function returns NULL if *rec_result* is NULL or corrupted.

6.3 Initializing Semantic Result Structure

VASRE provides the following API functions for construction and destruction of *SEM_RESULT* variables:

- *VA_InitSemResult*: Initialize semantic result structure
- *VA_FreeSemResult*: Release memory allocated for semantic result structure

```
int VA_InitSemResult(
    struct SEM_RESULT* sem_result   /**< [in] Pointer to semantic result */
    );
int VA_FreeSemResult(
    struct SEM_RESULT* sem_result   /**< [in] Pointer to semantic result */
    );
```

Call *VA_InitSemResult* to initialize a *SEM_RESULT* variable. This function assigns zero to the member variable *root* of the *SEM_RESULT* variable. *VA_FreeSemResult* releases the memory allocated for the *SEM_RESULT* variable. If the *SEM_RESULT* variable contains semantic tree structure and you don't need it any more, you should call *VA_FreeSemResult* to release the memory.

These functions return an error code if *sem_result* is NULL; otherwise, zero is returned.

6.4 Obtaining Semantic Results

VASRE provides the following API functions for getting a semantic result, and for accessing it:

- *VA_GetSemResult*: Generate semantic result tree from n-best alternative
- *VA_GetSrtNodeTypeStr*: Get string representing type of semantic result from enum value

```
int VA_GetSemResult(
    struct NBEST_ALTER* nbest_alter,    /**< [in] Pointer to n-best alternative */
    struct SEM_RESULT* sr,  /**< [out] Semantic result */
    char* script_err_msg,   /**< [out] Error message from SI processor */
    const size_t em_size    /**< [in] Size of script_err_msg */
    );
const char* VA_GetSrtNodeTypeStr(
    enum SRT_NODE_TYPE type /**< [in] Type of semantic result tree node */
    );
```

Call *VA_GetSemResult* to generate a semantic result from an n-best alternative. The semantic result is represented as a tree structure whose node is of *SRT_NODE* type. The semantic result output by this function must be released by *VA_FreeSemResult*.

If *VA_GetSemResult* fails to generate the semantic result due to syntax errors from tag statements or internal errors of the semantic interpretation processor, it returns a non-zero error code. The corresponding error message is copied to *script_err_msg*. If *em_size* is too small to cover the error message, this function outputs the truncated error message.

Call *VA_GetSrtNodeTypeStr* to get a string representing type of semantic result from the enum value. The returned string can be (1) "String", (2) "Integer", (3) "Double", (4) "Boolean", (5) "Object", or (6) "Array". Since a string literal is returned, do not attempt to release the memory of the returned character pointer. If *type* is unknown, *VA_GetSrtNodeTypeStr* returns an empty string.

6.5 Abstracting N-best Alternatives

VASRE provides the following API functions for streamed audio processing:

- *VA_GetRawTextOfNbestAlter*: Get raw text of n-best alternative
- *VA_GetTextOfNbestAlter*: Get text of n-best alternative
- *VA_GetScoreOfNbestAlter*: Get score of n-best alternative

```
int VA_GetRawTextOfNbestAlter(
    struct NBEST_ALTER* nbest_alter,    /**< [in] Pointer to n-best alternative */
    char* raw_text,                     /**< [out] Raw text */
    const size_t raw_text_size          /**< [in] Size of raw_text */
    );
int VA_GetTextOfNbestAlter(
    struct NBEST_ALTER* nbest_alter,    /**< [in] Pointer to n-best alternative */
    char* text,                         /**< [out] Output text */
    const size_t text_size              /**< [in] Size of text */
    );
short VA_GetScoreOfNbestAlter(
    struct NBEST_ALTER* nbest_alter     /**< [in] Pointer to n-best alternative */
    );
```

Call *VA_GetRawTextOfNbestAlter* to obtain the raw text of an n-best alternative. If *raw_text_size* is too small to cover the raw text, this function outputs the truncated text.

Call *VA_GetTextOfNbestAlter* to get the text representing an n-best alternative. The text basically represents a semantic result, which can be typed String, Integer, Double, or Boolean. If the semantic result is typed Object or Array and has multiple properties, the text representing the first element will be output. If this function fails to generate the semantic result due to the errors in the logical parsing or the semantic interpretation processor, this function returns a non-zero error code and outputs the raw text it recognized. If *text_size* is too small to cover the text, this function outputs the truncated text.

Upon successful operation, both *VA_GetRawTextOfNbestAlter* and *VA_GetTextOfNbestAlter* return zero; otherwise a non-zero error code will be returned. Note that if *raw_text* or *text* are NULL or *raw_text_size* or *text_size* is zero, then the two functions return zero but output nothing.

Call *VA_GetScoreOfNbestAlter* to get the confidence score of an n-best alternative. This function just takes the average of word-wise confidence scores. Access *nbest_alter* directly to get the confidence score of each word composing the recognized phrase or sentence. This function returns confidence score ranging from 0 to 1000. If *nbest_alter* is NULL or corrupted, this function returns zero.

6.6 Generating XML

VASRE provides the following API functions to generate an Extensible Markup Language (XML) file or string from the recognition result:

- *VA_RecResult2XmlFile*: Generate XML file from recognition result
- *VA_RecResult2XmlStr*: Generate XML string from recognition result

```
int VA_RecResult2XmlFile(
    REC_RESULT* rec_result,     /**< [in] Pointer to REC_RESULT variable */
    const char* xml_path_name,  /**< [in] Path name of XML file to be created */
    char* err_msg,              /**< [out] Error message from SI processor */
    const size_t em_size        /**< [in] Size of err_msg */
    );
int VA_RecResult2XmlStr(
    REC_RESULT* rec_result, /**< [in] Pointer to REC_RESULT variable */
    char* xml_str,          /**< [out] XML string */
    const size_t xs_size,   /**< [in] Size of xml_str */
    char* err_msg,          /**< [out] Error message from SI processor */
    const size_t em_size    /**< [in] Size of err_msg */
    );
```

Call *VA_RecResult2XmlFile* or *VA_RecResult2XmlStr* to generate an XML file or string from the *REC_RESULT* variable. The XML string represents the hierarchy of the *REC_RESULT* structure and the associated semantic results.

Because both *VA_RecResult2XmlFile* and *VA_RecResult2XmlStr* invoke the SI processor internally, they fail if the SI script contains syntax errors, or if the SI processor itself fails. If this is the case, the corresponding message will be copied to *err_msg*. If *em_size* is too small to cover the error message, this function outputs the truncated error message.

Upon successful operation, these functions return zero; otherwise, a non-zero error code will be issued. Note that if *xml_str* is NULL or *xs_size* is zero, *VA_RecResult2XmlStr* returns zero but outputs no XML string.

7 Other Functions

Use the following function to compose an error message based on the error code returned by any API function:

- *VA_ErrorCode2Msg*

```
char* VA_ErrorCode2Msg(
    const int err_code  /**< Error code */
    );
```

This function returns a string literal and you don't need to free the memory of the returned character pointer.

Appendix: List of VASRE API Error Codes

The VASRE API functions whose return type is *int* return zero upon successful completion. Upon failure, they return non-zero error codes. This section lists the error codes and the corresponding descriptions in the ascending order. You may pass the error code to *VA_ErrorCode2Msg* function to have the error message composed.

19: Wrong grammar file name

Path name of grammar file is wrong. It must end with `.grm` or `.gout`.

20: Wrong .grm file name

Path name of `.grm` file is wrong. It must end with `.grm`.

22: Failed to find .grm file

`.grm` file does not exist at the specified location.

30: Failed to open .grm file

The API function failed to open `.grm` file. Check the permission and make sure the file is not shared by other processes.

31: Wrong .grm file format

`.grm` file is a binary file. It must be a text file containing an SRGS text.

368: Damaged .gout file

The API function failed to decrypt `.gout` file to a loadable format. It is likely that the `.gout` file is not the output of the grammar compiler.

369: Wrong .gout file format

`.gout` file is a text file. It must be a binary file.

370: Failed to open .gout file

The API function failed to open `.gout` file. Check the permission and make sure the file is not shared by other processes.

371: Failed to process .gout file

The API function failed to interpret `.gout` file. It is likely the `.gout` file is not the output of the grammar compiler.

1500: Failed to initialize Windows socket

The API function failed to initialize the socket API (Windows only).

1501: Failed to connect to RM

The API function failed to connect to the RM. Check whether the RM has been started.

1502: Failed to communicate with RM

The API function failed to communicate with the RM. Check whether the RM has been started.

1503: No recognition server

No recognition server is currently available. This may happen if servers have not been started or you are attempting to launch more clients than the servers there are.

1509: Address of *HPORT* variable is zero

The address of HPORT variable is zero (*NULL*). Make sure the address of the *HPORT* variable correctly initialized was passed to the API function.

1520, 1521, 1522, and 1523: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1530: Invalid minimum speech length

The minimum speech length must be in the range of 0–1000 milliseconds.

1531: Invalid maximum speech length

The maximum speech length must be in the range of 1000–20000 milliseconds.

1532: Invalid amplitude threshold for voice activation detection

The amplitude threshold for voice activation detection must be in the range of 200–30000.

1533: Invalid zero-cross threshold for voice activation detection

The zero-cross threshold for voice activation detection must be in the range of 3–200.

1534: Invalid head margin for voice activation detection

The head margin for voice activation detection must be in the range of 150–500.

1535: Invalid tail margin for voice end detection

The tail margin for voice end detection must be in the range of 200–1200.

1536: pitch threshold for voice activation detection

The pitch threshold for voice activation detection must be in the range of 100–500.

1537: pitch count threshold for voice activation detection

The pitch count threshold for voice activation detection must be in the range of 1–10.

1700: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1705: No active grammar

No grammars are active on the server side. The audio cannot be processed.

1710: Vocabulary size overflowed

The vocabulary size of the added grammar(s) exceeds the license limit, which is 500 by default.

1720: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1731: Duplicate grammar tag

The API could not add the grammar since a grammar of the same tag already exists on the server side.

1732: Wrong grammar contents

The server failed to load the grammar because its contents have been corrupted.

1733: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1750: Grammar is too long

The grammar contents are too big to be added to the server.

1755: Failed to open XML file

The API function failed to open the XML file. Check the path name of XML and the permission of system directory.

1756: Wrong XML file name

Given XML file name is wrong or empty.

1757: Failed to save XML file

Error occurred while saving XML file.

1780, 1781, 1782, 1783, 1784, 1785, 1786: Failed to obtaining result of SI script processor

Error occurred while obtaining semantic results from SI script processor because of internal error of SI script processor.

1790, 1791, 1792, 1793: Failed to initialize SI script processor

Error occurred while initializing SI script processor, probably because system resources are insufficient.

1794: Syntax error in SI script

SI script processor found a syntax error from SI tag statements. See error message output by *VA_GetSemResult* for further details.

1795: Too long SI script

SI tag statement is too long (> 1,000,000 bytes).

1800: Failed to activate grammar

The API failed to activate the specified grammar because the grammar does not exist on the server side.

1801: Failed to deactivate grammar

The API failed to deactivate the specified grammar because the grammar does not exist on the server side.

1802: Failed to delete grammar

The API failed to delete the specified grammar because the grammar does not exist on the server side.

1804: Invalid grammar tag

The grammar tag is wrong. A grammar tag must comprise alphanumeric letters, hyphens, underscores, and periods.

1805: Too long grammar tag

The grammar tag is too long. The maximum allowed length of a grammar tag is 255.

1806: Invalid grammar tag

The grammar tag is wrong. A grammar tag must start with an alphabetical letter.

1807: Invalid grammar tag

The grammar tag is an empty string.

1900: Failed to find audio file

`.wav` file does not exist at the specified location.

1901: Failed to open audio file

The API function failed to open `.wav` file. Check the permission to ensure the file is not shared by other processes.

1910, 1911: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1920: Streamed audio processor is not ready

Streamed audio processor is not ready. Call *VA_StartStream* first. See Section 5.2 for further details.

1922: Invalid audio segment size

For streamed audio processing, each audio segment size must be no greater than 1000. See Section 5.2

for further details.

1923: Invalid audio segment size

For streamed audio processing, each audio segment size must be greater than zero. See Section 5.2 for further details.

1930: Too long speech

Speech is too long. Use *VA_SetMaxSpeechLen* to increase the maximum length of acceptable speech. See Section 3.2 for details.

1931: Too short speech

Speech is too short. Use *VA_SetMinSpeechLen* to decrease the minimum length of acceptable speech. See Section 3.2 for details.

1932: Invalid audio file format

The audio file format must be PCM in mono channel. This error is generally issued when the audio file format is ulaw or alaw.

1933: Invalid sampling frequency

The sampling frequency of the audio file must be 8 kHz.

1935: Invalid bytes per second

The bytes per second of the audio file must be 16000.

1937: Invalid bytes per sample

Bytes per sample of the audio file must be two.

1938, 2000: Failed to communicate with server

The API function failed to communicate with the server due to internal communication errors. Restart the server and try again.

1942: Pointer variable to *REC_RESULT* is NULL

The address of *REC_RESULT* variable is NULL and the API function couldn't access the recognition result.

1943: Pointer variable to *SEM_RESULT* is NULL

The address of *SEM_RESULT* variable is NULL and the API function couldn't access the semantic result.

2001: Invalid logical parsing

The first letter of logical parsing must be a square bracket “[”.

2002: Invalid logical parsing

The SI processor encountered the end of string while performing logical parsing.

2003: Invalid logical parsing

The SI processor encountered an unexpected letter while performing logical parsing.

2011: Invalid logical parsing

The SI processor failed to find a square bracket “[” following a rule name. In logical parsing, a square bracket must follow a rule name.

2012: Invalid logical parsing

Rule name in logical parsing is of length one: “$”.

2021: Invalid logical parsing

In logical parsing, the SI processor failed to find a brace “}” closing a tag.

2051: Invalid logical parsing

The SI processor found more words in logical parsing than *num_of_words* in *REC_RESULT* structure.

2052: Invalid logical parsing

The SI processor found mismatch between a word in logical parsing and the corresponding *words* element in *REC_RESULT* structure.

2081: Memory allocation failed

The API function failed to allocate memory.