Vocalizer Expressive 2.0

User’s Guide and
Programmer’s Reference
Revision 1

Copyright

Trademarks

Table of Contents

Vocalizer Expressive 2.0

Chapter 1

Getting started

User's Guide and
Programmer's Reference
Revision 1

Chapter I

Getting Started

Introduction

Thank you for using Nuance Vocalizer Expressive, our Text-To-Speech software designed for automotive and personal navigation devices. With the software and the documentation included in this package, you will be able to develop applications equipped with Nuance's state of the art automotive Text-To-Speech technology.

Installing the SDK

The Vocalizer Expressive SDK packages can be downloaded from Nuance support website, http://network.nuance. com.

First make sure to remove the voices and the engine of any previous version of the Vocalizer Expressive SDK. Then install the Vocalizer Expressive engine, and one or more voices.

The engine components are available from an SDK engine package, e.g. ve_engine_vM.m.r_target-platform.zip. This is a .zip file that you extract under an installation directory of your choice preserving the path of the files. A voice package like ve_enu_ava_embedded-pro_vM.m.r.zip contains the data components for one operating point of the voice. You have to extract these files under the same installation directory.

It's important that you download at least one voice package, and that you add the voice to the same installation directory. The installation directory should look like this, and in particular have the languages subdirectory:

```
installation_dir
+---common
+---doc
|   \---languages
+---inc
+---languages
+---lib
+---sample
\---test_sapi
```

If you fail to install a voice, Vocalizer Expressive won't find any language or voice data, and return an error to the test driver. For instance, vedemo.exe and the sample programs won't run, but report a message like "Can't initialize Nuance Vocalizer Expressive. Error code 0x80000012".

Chapter I

Refer to the VE SDK release note of your target platform (e.g. ve_release-note_vM.m.r_target-platform.htm) for specific details.

About the product

Text-To-Speech, from a general perspective, could be defined as the conversion of written text into spoken text.

When narrowed down to the context of electronics and computer science, it could be described as the process of speech synthesis by which machines translate a conventional orthographic representation of language into its spoken equivalent, using complex systems of linguistic rules and dictionaries, so as to achieve the most natural sounding speech output possible. The input text may be typed on a keyboard or may be read from various types of sources: files, web pages, data base records, SMS messages, etc.

Nuance Vocalizer Expressive is Text-To-Speech software designed for speech solutions embedded in automotive devices and in personal navigation devices. With its best-of-breed synthesis technologies it brings very high-quality 22 kHz audio with several footprints, and efficient usage of processing and memory resources.

Vocalizer Expressive also supports a variety of languages and voices. It features a flexible software architecture, and supports languages and voices as data-only components. For detailed information about the languages available, refer to the Product Release Note or contact Nuance.

Functional components

The main functional components of the Vocalizer Expressive software fall in 2 categories:

- Common code components.
  These components are language independent and they are responsible for handling API, managing internal system resources and processing all steps of Text-to-Speech system by accessing language or voice dependent data.

- Language and voice data components.
  The language and voice-specific data are bundled in a voice pack. This contains all the data required to work with one operating point of a voice.

Chapter I

Features

With the Nuance Vocalizer Expressive software you can perform the following tasks:

- Open and close one Text-To-Speech instance.
- Select a voice in one of the available languages.
- Synthesize speech from text. The input text is in the native language of the voice, but may contain fragments in the foreign languages that the voice supports.
- Stop reading out the input text.
- Change and inspect several control parameters such as volume level, rate level and pitch level.
- Load/unload a user dictionary
- Load/unload ruleset files
- Load/unload ActivePrompt databases

Chapter I

Navigating the documentation

User's Guide and Programmer's Reference

This volume contains information about the general aspects of the Text-To-Speech system, regardless of the language with which it is combined. The overview below outlines the contents of this volume.

Chapter I 'Getting started' introduces you to Text-To-Speech in general.

Chapter II 'Working with the Text-To-Speech System' consists of a Text-To-Speech primer. It contains definitions of concepts and characteristics of the Nuance TTS system. It also explains about application development.

Chapter III 'Text-To-Speech System Reference' provides operational instructions for the Nuance Text-To-Speech system. It reviews the functionality of the system, and describes the way in which the user can customize the pronunciation of input texts.

Chapter IV 'Text-To-Speech Function Reference' contains a detailed list and explanation of all the function calls, data types, data structures and error codes of the Application Programming Interface (API).

Appendix I 'Copyright and licensing of third-party software' gives the copyright and licensing information of third-party packages used by Vocalizer Expressive.

Language and voice documentation

The language and voice documentation provides specific instructions on the usage of Text-To-Speech with the languages and voices you have purchased.

The language and voice documentation is a set of .htm pages organized by language, voice and language data configuration under under installation_dir>\doc\languages. The start page is ve-language-index.html under installation_dir>\doc.

Chapter I

Document conventions

The following types of formatting in the text are used throughout the manuals to identify special information.

Convention	Type of information
Bold type	Used to refer to titles of chapters or sections. In the *Function Reference*, it denotes a term or a character to be typed literally, such as function names, type definitions.
Italic type	Used to refer to titles of manuals and to emphasize certain words, such as new terms. In the Function Reference, it denotes a placeholder or a variable for which you must supply a value,
[ ]	Encloses optional statements.
|	Denotes an either/or choice.
…	Indicates that the preceding item may be repeated.
`Monospaced` type	Sets off code examples and shows syntax spacing. Also indicates directory paths.
KEYCAPS	Indicates a key on your keyboard. Example: "Press *DEL* to remove the word.
Menu | Choice	Indicates menu commands. Example: "File | Save" points to the Save command on the File menu.

Vocalizer Expressive 2.0

Chapter II

Working with the Text-To-Speech System

User's Guide and
Programmer's Reference
Revision 1

Chapter II

Working with the Text-To-Speech System

A Text-To-Speech Primer

Introducing Text-To-Speech

Purpose of Text-To-Speech

Text-To-Speech can be defined in many ways. However, the most relevant description would probably be the one that describes it as a way of having a computer *audibly* communicate information to the user.

In situations where visual feedback is inadequate or even impossible, audible feedback may be an essential feature; in other situations it may add extra value to a product.

In general, Text-To-Speech provides a very valuable and flexible alternative for digital audio recordings in the following cases:

- Professional recordings are too expensive.
- Disk storage is insufficient to store recordings.
- The application does not know in advance what it will need to speak.
- The information varies too much to record and store all the alternatives.

Vocalizer Expressive also supports mixing digital audio recordings with Text-To-Speech for applications where a mixed approach is desired.

Main processing steps

Different implementations of Text-To-Speech systems exist. This section discusses some of the concepts on which these systems are built.

Generally, a Text-To-Speech conversion can be broken down into three main parts: a linguistic, a phonetic and an acoustic part.

First, an ordinary text is entered into the system. Linguistic processing converts this text into a phonetic transcription, which basically represents the sequence of phonemes of the spoken version of the text. From this representation, the phonetic processing calculates a stream of speech parameters; these model the speech signal. Finally, acoustic processing uses these parameters to synthesize the speech signal.

Linguistic processing

The linguistic processing of a Text-To-Speech system performs several tasks: text normalization, orthographics-to-phonetics conversion (i.e. grapheme-to-phoneme conversion and stress assignment), lexical and morphological analysis, syntactic analysis, and, to a lesser extent, semantic analysis.

Text preprocessing

Text preprocessing breaks the input text into individual sentences. For specific application domains additional intelligence can be built into a text preprocessing module.

Text normalization

A Text-To-Speech system should be able to read aloud any written text, even if it contains a miscellany of abbreviations, dates, currency indications, time indications, addresses, telephone numbers, bank account numbers and various other symbols such as quotation marks, parentheses, apostrophes and other punctuation marks.

For example, to solve the abbreviation problem, an abbreviation dictionary can be used. Abbreviations that do not occur in the dictionary are then pronounced as single words or are spelled out depending on the graphotactic structure of the abbreviation.

Another example of text normalization is the processing of digits. Digits are handled according to the syntactic and semantic context in which they appear. In English (as in Dutch and German) digit strings such as 1991 are pronounced differently according to the context (number or year). This is not the case in Spanish or French. In Spanish for example, the conversion of digit strings also needs lexical information because the pronunciation of the digit string sometimes changes depending on the gender of the noun or on the following abbreviation.

To handle text normalization, Text-To-Speech systems use a lot of orthographic knowledge, frequently phrased by linguistic context-dependent rules, in combination with dictionary lookup.

Orthographics-to-phonetics

This conversion is one of the main tasks of the linguistic processing part.

Chapter II

A Text-To-Speech system needs a lot of pronunciation knowledge to perform this task, which includes *grapheme-to-phoneme* conversion, *syllabification* and *stress assignment*.

Different ways of orthographic-to-phonetic conversion are possible:

- Consulting dictionaries containing full word forms or morphemes
- Using a set of pronunciation rules
- Using techniques such as neural nets or classification trees. Most (commercial) Text-To-Speech systems use a hybrid strategy combining word dictionaries, morpheme dictionaries and pronunciation rules. Although the same strategy can be used for the development of all language versions, it is obvious that each language has its own particularities.

Lexical, morphological and syntactic analysis

Lexical, morphological and syntactic analysis is needed to solve pronunciation ambiguities.

The English verb re'cord for example, can also be pronounced as the noun 'record. In French, the character string président is pronounced differently depending on its part-of-speech (noun or verb).

Lexical, morphological and syntactic information is also very important to create a correct prosodic pattern for each sentence. For instance, important syntactic boundaries entail intonational changes and vowel lengthening.

A frequently used method for tagging isolated words with their parts of speech is a combination of morphological rules and dictionary look-up. For example, particular word endings help predict the part-of-speech of words.

The syntactic analysis can be performed with different parsing techniques. Some of these techniques are developed within the field of Natural Language Processing (NLP) and adapted to the special needs of Text-To-Speech synthesis. For example, parsing techniques for Text-To-Speech, much more than for NLP applications such as text translation, should meet the real-time requirement.

Most of the current commercially available Text-To-Speech systems do not perform a full syntactic analysis, i.e. they do not construct a full syntax tree, but rather perform a phrase level parsing. For instance, context-dependent rules can be used to solve part-of-speech ambiguities and divide a sentence in word groups and prosodic phrases.

Phonetic processing

The phonetic module performs two main tasks to produce an adequate sequence of speech parameters:

- Segmental synthesis
- Creation of good prosodic patterns

Segmental synthesis

This part of the Text-To-Speech system is responsible for the synthesis of the spectral characteristics of synthetic speech. In most systems, the segmental synthesis module also handles amplitude (loudness).

Prosody

To synthesize intelligible and natural sounding speech, it is essential to create good prosodic characteristics.

The synthesis of prosody involves two steps:

- The production of a good intonation contour
- The assignment of a correct duration to each phoneme

As already mentioned, the creation of a correct amplitude (loudness) contour is frequently handled as a part of the segmental synthesis module.

With respect to the *intonation,* some important principles have to be taken into account.

Each sentence contains at least one or more important or dominant words.

In a lot of languages, an important word is marked by means of an intonation accent realized as a pitch movement on the lexically accented syllable of the important word.

Intonation is not only used to emphasize words but also to mark the sentence type (e.g. declarative versus interrogative, WH-questions versus yes/no-questions) and to mark important syntactic boundaries (e.g. with phrase final continuation rises).

In tone languages such as Chinese, word meanings and/or grammatical contrasts can be conveyed by variations in pitch. In pitch-accent languages such as Swedish and Japanese, a particular syllable in a word is pronounced with a certain tone. This is in contrast to languages such as English where each word has a fixed lexical stress position, though there is less restriction on the use of pitch.

Apart from all the intonation effects just described, some segmental effects (such as the influence of the post-vocalic consonant on the pitch of the preceding vowel) can also be observed in natural intonation contours.

A Text-To-Speech system should include a language-specific intonation module that models the perceptually relevant intonation effects of the target language. Such an intonation model should at least take into account the number, location and stress level of the important words, the location of the major syntactic boundaries and the sentence type.

Among the different approaches possible, an approach applicable to a lot of languages (such as English and Dutch) is to describe pitch contours by means of standardized pitch movements (rises and falls). Rules specify how these elementary pitch movements can be combined to create intonation contours for entire messages.

Assigning a correct duration to each phoneme is essential. Measurements on speech data as well as perceptual experiments prove the relevance and the importance of good duration models.

Phoneme durations are influenced by a lot of factors. Without being exhaustive, the list below shows some of the factors a duration model should take into account, as they influence the intrinsic duration of the phonemes:

- The phonetic context
- The stress level
- The position within the word
- The syntactic structure of the sentence
- The opposition between content and function words

Phoneme models can be developed and implemented in different ways resulting, for example, in rule models, neural net models or decision tree models.

Some of the models are phoneme-oriented while others predict the duration of syllables before assigning durations to phonemes.

Although the prosody models in Text-To-Speech systems have become increasingly sophisticated, synthetic prosody is still one of the main causes of the quality difference between synthetic and human speech.

Acoustic processing

The last part of a Text-To-Speech conversion performs the acoustic processing.

At this stage, the speech data created in the previous stage of the processing are converted into a speech signal. The synthesis model used should allow the independent manipulation of spectral characteristics, phoneme duration and intonation.

The Nuance Text-To-Speech system uses one of a set of proprietary speech synthesizers to create the speech output.

Voice operating points

The Nuance Text-To-Speech system incorporates different approaches to phonetic and acoustic processing, called different back-end technologies:

- Back-end technology 1 uses a speechbase of encoded speech units taken from recordings of natural speech, selects the appropriate units and concatenates them to realize a phonetic transcription.
- Back-end technology 3 has a speech parameter generator that has been trained on a corpus of recordings and their transcription, and a parametric synthesizer.

The different back-end technologies implement different models of natural speech, and typically trade off voice quality for processing resources as footprint and CPU load:

- Back-end technology 1 is able to produce as good as natural sounding speech at the cost of large speechbases (tens to hundreds of megabytes). The speech quality degrades notably when the speechbase grows smaller than a threshold number of speech units.
- Back-end technology 3 on the other hand is good to reach small footprints (around 2 megabyte) while remaining able to produce smooth speech, albeit sounding more synthetic.

Using one of the available back-end technologies Vocalizer Expressive can synthesize one and the same voice in different ways. We say that Vocalizer Expressive uses a particular voice operating point to synthesize speech for the voice. The following voice operating points are offered:

- Premium High voice operating point:
  back-end technology 1, 22 kHz
  (parameter value "premium-high").
- Embedded High voice operating point:
  back-end technology 1, 22 kHz
  (parameter value "embedded-high").
- Embedded Pro voice operating point:
  back-end technology 1, 22 kHz
  (parameter value "embedded-pro").

- Embedded Compact voice operating point:
  back-end technology 3, 22 kHz
  (parameter value "embedded-compact").

Note that the different back-end technologies have their own configuration settings. Back-end technology 1 has different modes of operation according to

- the sampling frequency: 22 kHz for the Vocalizer Expressive SDK voice packages.
- the size of the speechbase (in number of speech units): Embedded Pro < Embedded High ~ Premium High (where the order represents overall voice quality), and
- the type of encoding of the speech units: Embedded Pro ~ Embedded High < Premium High.

Back-end technology 3 has a single operation mode for the sampling frequency: 22 kHz.

Definition of concepts

Phonetic transcription and phonemes

A phonetic transcription consists of a sequence of phonemes. A *phoneme* is the most elementary building block in the sound system of a language. In essence, a phoneme constitutes a family of sound variants, which a language treats as being "the same". Its concept allows establishing patterns of organization in the indefinitely large range of sounds heard in a language. Typically, a specific language contains approximately 50 different phonemes.

Nuance has established its own specifications for the representation of phonemes: the *L&H+ phonetic alphabet.* It associates each phoneme to a sequence of one or more characters. The phonemes of the supported languages with their associated L&H+ representation are described in the **Language and voice documentation.**

User Dictionaries

User dictionaries allow you to specify special pronunciations for particular words or strings of characters (e.g. abbreviations) and can contain orthographic as well as phonetic information. They make it possible to customize the output of the Text-To-Speech system.

See the **User Dictionaries** section below for more information.

Language Codes

The system uses a Nuance proprietary language code and this code is used as a part of filename of user dictionary and other purpose. The language code is as below:

Language name	Language code
Arabic	ARW
American English	ENU
Argentinian Spanish	SPA
Australian English	ENA
Basque	BAE
Belgian Dutch	DUB
Brazilian Portuguese	PTB
British English	ENG
Canadian French	FRC
Catalan	CAE
Chinese Mandarin	MNC
Colombian Spanish	SPC
Czech	CZC
Danish	DAD
Dutch	DUN
Finnish	FIF
French	FRF
Galician	GLE
German	GED
Greek	GRG
Hebrew	HEI
Hindi	HII
Hong Kong Cantonese	CAH
Hungarian	HUH
Indian English	ENI
Irish English	ENE
Indonesian	IDI
Italian	ITI
Japanese	JPJ
Korean	KOK
Mexican Spanish	SPM
Norwegian	NON
Polish	PLP
Portuguese	PTP
Romanian	ROR
Russian	RUR
Slovak	SKS
Scottish English	ENS
South African English	ENZ
Spanish	SPE
Swedish	SWS
Taiwanese Mandarin	MNT

Language name	Language code
Thai	THT
Turkish	TRT
Valencian	VAE

Chapter II

Supplying external services

Vocalizer Expressive relies on a number of services that the user needs to implement, and therefore are called external services. These external services are abstractions of platform resources, and they allow the user to select an implementation that best suits the target application and platform.

An external service basically is a collection of callback functions, called an interface, and a handle of the service. The TTS class and/or its instances use the service by calling an interface function on the supplied service handle, and thus pass control to the user-defined implementation.

These are the different external services:

- the Heap service:
  lets Vocalizer Expressive allocate and free memory blocks. It is a required service.

- the Critical Sections service:
  allows Vocalizer Expressive to run thread-safe. If the application does not require this, it can omit this service.

- the data access services Data Streams and Data Mappings:
  provide Vocalizer Expressive with the language and voice-specific data. The Data Streams service is required, the Data Mappings service is optional.

- the User Log service:
  lets Vocalizer Expressive transfer the raw data of error and diagnostic messages to the client so the user can decide about the log format and location. This is an optional service.

- the Output Delivery service:
  lets Vocalizer Expressive transfer the synthesized audio and marker stream to the client.

It is up to the client to collect the appropriate services and pass them into the function that creates the TTS class. This implies that the client can't create a TTS class or TTS instances unless it supplies the required services to Vocalizer Expressive.

The user has the full freedom to select the implementation of a service interface that best suits her needs. But this freedom also puts the responsibility on the user to provide Vocalizer Expressive with a correct implementation that is fast enough to let Vocalizer Expressive synthesize speech in real time. The package includes a reference implementation of the services as part of the sample program *read_file*;

this will help you getting started, and may already be good enough for most users.

Heap service

The Heap service offers functions to allocate, reallocate and free blocks of memory.

The reference implementation simply delegates to the ANSI/C functions *malloc()*, *calloc()*, *realloc()* and *free()*.

Critical Sections service

The Critical Sections service interface defines functions to create critical sections (mutexes), to synchronize different threads on entering and leaving blocks of code.

The reference implementation is built on the critical section library available on one of the target platforms Windows, Windows Mobile and Unix.

Data access

Vocalizer Expressive uses code components that work with language and voice-specific data components to convert text into speech. The data components required for one operating point of a voice are bundled in a single voice pack. The client configures a TTS instance in terms of language, voice, voice operating point, frequency, etc. and from those parameter settings the code components derive which voice pack they need.

Vocalizer Expressive does not make any assumptions about the location of its voice packs in a deployed application. Therefore the data components are identified by a logical name, and a Vocalizer Expressive component that needs a data component, queries a data access service for it by name.

Reference deployment configuration

In the Vocalizer Expressive software the voice packs are available from a reference deployment configuration based on a file system:

- Each voice pack is stored in a separate .dat file, where the filename is easily derived from its logical name: it is equal to the logical name with '/' converted to '_', [^a-z0-9] converted to '-' and ".dat" appended. For instance, the voice pack named 'enu/ava/embedded-pro/1-0-0' corresponds with the file 'enu_ava_embedded-pro_1-0-0.dat'

- There are separate directories for each language. This means that all the .dat files for a language and its voices are located in a separate directory.

Note that Vocalizer Expressive components may request one and the same data component in a voice pack several times, and may even request data components that don't exist, as they may look for specific data components first, then fall back to more generic ones.

Types of data access services

Vocalizer Expressive accepts two different types of data access services: a required Data Streams service, and an optional Data Mappings service. The Data Mappings service is stricter to work with than the Data Streams service as the caller does not own the data, but only gets a read-only pointer to a data block owned by a data mapping; hence the caller must not touch the data. In contrast, the Data Streams service copies a block of data in a buffer owned by the caller.

The Vocalizer Expressive components access their data according to the data mapping model using an internal data mapping wrapper. This wrapper either emulates data mappings using the Data Streams service, or it merely delegates to the Data Mappings service. This design makes Vocalizer Expressive suited to work with a true implementation of a data mapping interface that exploits the power of memory-mapped files.

The Data Mappings service allows the client to optimize resource usage and performance for platforms like Windows Mobile that have native OS support for memory-mapped file access, or for applications and platforms where data should reside in ROM (minimize RAM use at the cost of performance) or where data should be loaded into RAM in their entirety at startup (maximize performance).

The reference implementation of both data access services compile the filename of a data component using the rule above, then look for that file in a common directory and a list of language directories. It's up to the client to pass these directories as arguments to the function that retrieves the data access service.

Data Streams service

The Data Streams service offers functions to create and work with data streams. The primary purpose is to read data from a particular position in a data stream. It also includes a function to write data to a data stream, but Vocalizer Expressive only calls this when it is built with extra logging.

The reference implementation delegates to the file I/O functions from the standard library.

Data Mappings service

The Data Mappings service is a bundle of functions to create and work with data mappings to acquire read-only access to blocks of data.

The reference implementation is built on the file mapping library available on one of the target platforms Windows and Windows Mobile.

User Log service

The User Log service defines a logging interface for reporting diagnostic and error messages. It is optional, but when supplied, it is added as a log subscriber to the log system of Vocalizer Expressive, next to other built-in log subscribers such as the diagnostic logger.

The User Log service receives the raw data of error messages (the error ID and a list of key-value pairs), and this lets the user choose her own the log format and where to put the log information.

The reference implementation simply uses *printf()* to write the error ID and its key-value pairs to stdout.

Chapter II

Preparing a text for Text-To-Speech

Introduction

Vocalizer Expressive is designed to pronounce any written text. The Text-To-Speech conversion is based on state-of-the-art technology from Nuance. For the pronunciation of the input text, the Nuance Text-To-Speech system applies linguistic rules and dictionaries, so as to achieve the best possible speech output.

Vocalizer Expressive offers a set of additional mechanisms to intervene in the automatic pronunciation process by means of control sequences specified within the input text or by loading tuning data that overrule and complement the internal system behavior.

The different types of tuning data (User Dictionaries, User Rulesets and ActivePrompt databases) are covered in the following chapters. This chapter describes the basic controls to intervene in the pronunciation of text:

- Rewriting the orthography
- Using control sequences
- Entering phonetic input

Input text encoding

By default Vocalizer Expressive expects the input text to be encoded in platform endian UTF-16. Vocalizer Expressive does not require a specific Unicode version per language.

Vocalizer Expressive can also be configured to accept the character encoding UTF-8. This is done by calling **ve_ttsSetParamList()** and setting the **VE_PARAM_TYPE_OF_CHAR** parameter to the value **VE_TYPE_OF_CHAR_UTF8**.

Rewriting the orthography

As the Text-To-Speech system has limitations not all messages will come out equally well.

By experiment with different ways to phrase the same message (e.g. using synonyms or changing word order), often a better result can be obtained.

This can most easily be done by re-writing static input text, but even dynamically generated text can be rewritten using search and replace patterns via user rulesets. See the **User Rulesets** section below for more information.

Using control sequences

Overview

A control sequence is a piece of text that is not to be read out, but instead offers the possibility to intervene in the automatic pronunciation process. In this way the user can alter the way in which a text will be read, and acquire full control over the pronunciation of the input text. Control sequences can also be used to insert bookmarks in the text.

Vocalizer Expressive supports a number of control sequences which are covered in the following sections:

- Activating implicit matching for an ActivePrompt domain
- Controlling end-of-sentence detection
- Setting the language of the input text
- Marking a multi-word string for lookup in the user dictionary
- Setting the type of prosodic boundary
- Setting the word prominence level
- Inserting a pause
- Changing the pitch
- Changing the speaking rate
- Controlling the read mode
- Resetting control sequences to the default
- Setting the spelling pause duration
- Inserting phonetic text, Pinyin text for Chinese languages or diacritized text for Arabic
- Guiding text normalization
- Changing the voice
- Changing the volume
- Setting the end-of-sentence pause duration
- Inserting a digital audio recording
- Inserting a bookmark

- Inserting an ActivePrompt, which is either a tuned Text-To-Speech segment or a compressed digital audio recording stored in an Nuance ActivePrompt database

All control sequences follow this general syntax notation:

*<ESC> \ <parameter> = <value> *

where

- *<ESC>* represents the escape character `"\x1B"`
- *<parameter>* is the name of the control parameter that the control sequence affects
- *<value>* is the value you want to assign to the control parameter.

A value that is set with a control sequence, remains active until another control sequence sets a new value, or until the end of the input text. Note that control sequences should be located outside of words; when entered inside a word the effect is left unspecified.

Activating implicit matching for an ActivePrompt domain

This control sequence activates implicit matching for an ActivePrompt domain starting at a specific location in the text.

For example:

<ESC>\domain=banking\Did you say your account number is 238773?

ActivePrompts are explained in the **ActivePrompts** section below.

Controlling end-of-sentence detection

The control sequences *<ESC>\eos=1* and *<ESC>\eos=0* control end of sentence detection, with *<ESC>\eos=1* forcing a sentence break and *<ESC>\eos=0* suppressing a sentence break. To suppress a sentence break, the *<ESC>\eos=0* must appear immediately after the symbol that triggers the break (such as after a period). To disable automatic end-of-sentence detection for a block of text, use *<ESC>\readmode=explicit_eos* as described below.

Some examples:

Tom lives in the U.S. <ESC>\eos=1\ So does John. 180 Park Ave. <ESC>\eos=0\ Room 24

Setting the language of the text

Use the control sequence *<ESC>\lang=<lng_code>* to indicate that the input text starting at that location is in the language *<lng_code>*. The value *<lng_code>* is a 3-letter language code.

Example:

Follow <ESC>\lang=frf\ <ESC>\toi=lhp\ 'Ry_d$_la_vjE.jaR.'djER <ESC>\toi=orth\ <ESC>\lang=enu\ for 100 meter.

Note that it depends on the multilingual capabilities of the voice whether the voice can take the language of the text into account.

Marking a multi-word string for lookup in the user dictionary

Use this control sequence to mark the beginning and the end of a multi-word string that you want Vocalizer Expressive to look up as a single entry in the user dictionary.

For example:

Alternatively use the <ESC>\mw\ IP address <ESC>\mw\ to connect.

This is explained in the **User Dictionaries** section below.

Setting the type of prosodic boundary

Insert *<ESC>\nlu=BND:<strength>* to set the type of prosodic boundary inserted after the following word.

Prosodic boundary	Description
<ESC>\nlu=BND:W	Weak phrase boundary (no silence in speech)
<ESC>\nlu=BND:S	Strong phrase boundary (silence in speech)
<ESC>\nlu=BND:N	No boundary

For example:

Ich sehe <ESC>\nlu=BND:S\ Hans morgen im Kino.

Setting the word prominence level

Insert *<ESC>\nlu=PRM:<level>* to set the prominence level on the following word.

Prominence	Description
<ESC>\nlu=PRM:0	Reduced
<ESC>\nlu=PRM:1	Stressed
<ESC>\nlu=PRM:2	Accented
<ESC>\nlu=PRM:3	Emphasized

For example

Ich sehe <ESC>\nlu=PRM:3\ Hans morgen im Kino.

Inserting a pause

This control sequence inserts a pause of a specified duration at a specific location in the text.

For example:

His name is <ESC>\pause=300\ Michael.

The control sequence *<ESC>\pause=<dur_ms>* \ inserts a pause of *<dur_ms>* milliseconds; the supported range is 1..65535 milliseconds for Embedded Pro, Embedded High and Premium High voice operating points, 1..6553 for Embedded Compact voice operating points.

Changing the pitch

The control sequence *<ESC>\pitch=<level>* scales the inherent pitch of the voice with a factor *<level>*. The value *<level>* is between 50 (half the inherent pitch, i.e. one octave lower) and 200 (two times the inherent pitch, i.e. one octave higher). The default value is 100.

Example:

I can <ESC>\pitch=80\ speak lower <ESC>\rate=120\ or speak higher.

Changing the speaking rate

The control sequence *<ESC>\rate=<level>* sets the speaking rate to the specified value, where level is between 50 (half the default rate) and 400 (four times the default rate), where 100 is the default speaking rate.

Example:

I can <ESC>\rate=150\ speed up the rate <ESC>\rate=75\ or slow it down.

Controlling the read mode

The control sequence *<ESC>\readmode=mode* can change the reading mode from sentence mode (the default) to various specialized modes:

Read mode	**Description**
*<ESC>\readmode=sent *	Sentence mode (the default)
<ESC>\readmode=char	Character mode (similar to spelling)
<ESC>\readmode=word	Word-by-word mode
<ESC>\readmode=line	Line-by-line mode
<ESC>\readmode=explicit_eos	Explicit end-of-sentence mode (sentence breaks only where indicated by *<ESC>\eos=1*)

Example:

<ESC>\readmode=sent\ Please buy green apples. You can also get pears.
(This input will be read sentence by sentence.)

<ESC>\readmode=char\ Apples
(The word "Apples" will be spelled.)

<ESC>\readmode=line
Bananas
Low-fat milk
Whole wheat flour
(This input will be read as a list, with a pause at the end of each line.)

<ESC>\readmode=explicit_eos
Bananas.
Low-fat milk.
Whole wheat flour.
(This input will be read as one sentence.)

Resetting control sequences to the default

The control sequence *<ESC>\rst* resets all parameters to the original settings used at the start of synthesis.

For example:

<ESC>\vol=10\ The volume is set to a low value. <ESC>\rst\ Now it is reset to its default value.
<ESC>\rate=75\ The rate is set to a low value. <ESC>\rst\ Now it is reset to its default value.

Setting the spelling pause duration

The control sequence *<ESC>\spell=<duration>* sets the inter-character pause to the specified value in msec. For example:

The part code is
<ESC>\tn=spell\<ESC>\spell=200\a134b<ESC>\tn=normal

Note: The spelling pause duration does not affect the spelling done by *<ESC>\readmode=char* because that mode treats each character as a separate sentence. To adjust the spelling pause duration for *<ESC>\readmode=char*, set the end of sentence pause duration using *<ESC>\wait* instead.

Inserting phonetic text, Pinyin text for Chinese languages or diacritized text

By default Vocalizer Expressive considers the input as orthographic text, but it also supports other types of input:

- Phonetic text. Phonetic input is explained in the **Entering Phonetic Input** section.
- Pinyin text for Chinese languages. Pinyin is a Romanized form that represents Chinese ideographs using Latin letters and numbers.
- Diacritized orthographic text for languages like Arabic and Hebrew. In these languages regular written text may leave out

the vowels. The diacritized form is the counterpart with all vowels explicitly represented by diacritics.

The control sequence <*ESC*>*toi*=<*type*>\\ marks the type of the input starting after the control sequence:

Type of input	Starts
<*ESC*>*toi=lhp*\\	Phonetic text in the phonetic alphabet L&H+.
<*ESC*>*toi=nts*\\	Phonetic text in the phonetic alphabet NT-SAMPA.
<*ESC*>*toi=pyt*\\	Pinyin text in Chinese languages.
<*ESC*>*toi=diacritized*\\	Diacritized text
<*ESC*>*toi=orth*\\	Orthographic text (default)

The control sequences that start phonetic text can be extended as <*ESC*>*toi=*<*type_phon*>*:"*<*orth_text*>*"*\\: this defines <*orth_text*> as the orthographic counterpart of the phonetic fragment. Vocalizer Expressive uses such a phonetic + orthographic fragment similarly to a phonetic user dictionary entry. It may also entirely fall back to the orthographic alternative if it can't realize the phonetic fragment. This is elaborated in the section **How to work with native and foreign phonetic input**.

Example:

<*ESC*>*lang=iti*\\ <*ESC*>*toi=nts:"Romano Prodi"*\\ *ro|'ma|no prO|di* <*ESC*>*toi=orth*\\

Guiding text normalization

The control sequence <*ESC*>*tn=*<*type*>\\ is used to guide the text normalization processing step. This is the basic set of values that you can specify for <*type*>; you find a description of the full set in the **Language and voice documentation**.

Control sequence	Use
<*ESC*>*tn=spell* \\	Instruct text normalization to start spelling out the input text that follows.
<*ESC*>*tn=address* \\	Inform text normalization to expand the text that follows as an address.
<*ESC*>*tn=sms* \\	Inform text normalization to expand the text that follows as an SMS message.
<*ESC*>*tn=normal* \\	Reset to the regular text normalization.

The end of a text fragment that should be normalized in a special way is tagged with <*ESC*>*tn=normal*\\.

The initial text normalization mode for each input text depends on the value of the VE_PARAM_TEXTMODE parameter: SMS mode if VE_TEXTMODE_SMS or regular if VE_TEXTMODE_STANDARD. For more info on this parameter see **Chapter IV: Text-To-Speech Function Reference**.

Some examples:

<ESC>\tn=address\ 244 Perryn Rd
Ithaca, NY <ESC>\tn=normal

That's spelled <ESC>\tn=spell\Ithaca<ESC>\tn=normal

<ESC>\tn=sms\ Carlo, can u give me a lift 2 Helena's house 2nite? David <ESC>\tn=normal

Changing the voice

The control sequence *<ESC>\voice=<voice_name>* changes the speaking voice, which also forces a sentence break. For example:

<ESC>\voice=samantha\ Hello, this is Samantha.
<ESC>\voice=tom\ Hello, this is Tom.

Changing the volume

The control sequence *<ESC>\vol=<level>* sets the volume to the specified level, where level is a value between 0 (no volume) and 100 (the maximum volume), where 80 is the default volume. This control affects both the speech signal synthesized by Vocalizer Expressive and the audio that the user supplies through a recorded ActivePrompts database and by *<ESC>\audio="<path>"*.

For example:

<ESC>\vol=10\ I can speak rather quietly, <ESC>\vol=90\ but also very loudly.

Setting the end-of-sentence pause duration

The control sequence *<ESC>\wait=<value>* sets the end of sentence pause duration (wait period) to a value between 0 and 9, where the pause will be 200 msec multiplied by that number. Some examples:

<ESC>\wait=2\ There will be a short wait period after this sentence.
<ESC>\wait=9\ This sentence will be followed by a long wait period. Did you notice the difference?

Inserting a digital audio recording

This control sequence inserts a digital audio recording at a specific location in the text.

For example:

Say your name at the beep. <ESC>\audio="c:\recordings\beep.wav"

The control sequence *<ESC>\audio="<path>"* inserts the recording specified by *<path>*, a local file system path. Vocalizer Expressive only supports inserting WAV format audio files that contain linear

16-bit PCM samples, and the recording's sampling rate must match the current voice.

Inserting a bookmark

The control sequence *<ESC>\mrk=<name>* marks the position where it appears in the input text, and has Vocalizer Expressive track this position throughout the Text-To-Speech conversion. After synthesis it delivers a bookmark marker that refers to this position in the input text and the corresponding position in the audio output. For more information on the marker output mechanism, please refer to the topics on the VE_CBOUTNOTIFY call-back function and the VE_MSG_OUTBUFDONE message in **Chapter IV: Text-To-Speech Function Reference**.

The use of this control sequence does not affect the speech output process.

It is important to note that Vocalizer Expressive only supports bookmarks where the name is a number in the range [0, 2147483648].

Some examples:

This bookmark <ESC>\mrk=1111\ marks a reference point.
Another <ESC>\mrk=2222\ does the same.

Inserting an ActivePrompt

This control sequence explicitly inserts an ActivePrompt at a specific location in the text.

For example:

<ESC>\prompt=banking::confirm_account_number\ 238773?

ActivePrompts are explained in the **ActivePrompts** section below.

Entering phonetic input

Nuance Vocalizer Expressive supports phonetic input, so that words of which the spelling deviates from the pronunciation rules of a given language (e.g. foreign words or acronyms unknown to the system) can still be correctly pronounced.

The phonetic input is composed of symbols of a phonetic alphabet. Vocalizer Expressive supports 2 phonetic alphabets, both of which can conveniently be entered from a keyboard:

- L&H+ is a Nuance specific alphabet. In the **Language ad voice documentation** you will find the L&H+ Phonetic Alphabet of the language concerned.
- The NT-SAMPA phonetic alphabet is a proprietary standard of NavTeq modeled after SAMPA and X-SAMPA. The

NavTeq Voice Reference Guide defines the list of phonetic symbols per language.

Using the control sequence for phonetic text a possible phonetic input (as a replacement for the English word “zero”) can be:

<ESC>\toi=lhp *'zi*.R+*o&U* *<ESC>\toi=orth*

User Dictionaries

Introduction

User dictionaries allow you to specify special pronunciations for particular words or multi-word strings. They make it possible to customize the output of the Text-To-Speech system. In particular, a user dictionary contains mappings from an orthographic string to either a phonetic transcription, or to an orthographic transcription, e.g. to expand abbreviations.

For Asian languages, which don’t use blanks to separate words, user dictionaries also direct Vocalizer Expressive to segment sentences into words. For these languages the replacement string for an orthographic word can be a phonetic transcription or a Pinyin transcription.

Phonetic transcriptions in a particular language are composed of phonemes represented by L&H+ symbols. More information about these phonemes can be found in the **Language and voice documentation**.

You create a user dictionary in Vocalizer Studio and you save it as a binary format. Then you load the binary user dictionary on Vocalizer Expressive.

Vocalizer Expressive consults the user dictionary for each individual word in the input text, and for multi-word fragments tagged by the control sequence *<ESC>\mw*. First it looks up the string “as is”, then the string with leading and trailing quotes and brackets stripped, then with trailing dots stripped, then the string in lower case. It tries these candidates until the lookup returns a hit, or all are missed.

Text format description

A text user dictionary is a plain Unicode text file (encoded in UTF-16 or UTF-8). It contains one or more data sections, each section being a collection of entries that share the same attributes. A section is defined in particular by:

- a header section specifying attributes such as language and replacement type, and

- a number of user dictionary entries.

For instance

```
[Subheader]
Language = <language_code>
Content = <content_spec>
Representation = <repr_spec>

[Data]
<entries>
```

A user dictionary entry is basically a key-value pair. The key is the word or multi-word string, and the value is its replacement string, i.e. an orthographic or a phonetic transcription, e.g.

```
DLL            "Dynamic Link Library"
```

A text user dictionary has to start with a file header section marked with the string "[Header]". The following fields go in that header:

- Language: mandatory field: the language of the entries.
- Name: optional field: a name for the dictionary.
- Description: optional field: a description of what is in the dictionary.
- Content: optional field: the type of entries stored in the following section (phonetic transcriptions or orthographic transcriptions).
- Representation: optional field: the representation of the entries in the following sections.

For instance

```
[Header]
Language = ENG
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING
```

After the file header section, one or more data sections can be specified. Each data section has a "[SubHeader]" and a "[Data]" part, as illustrated above. If the user dictionary contains only one data section, the "[SubHeader]" section can be omitted (and the entries take the attributes from the file header).

The "[SubHeader]" part can have the same fields as the file header marked by "[Header]". For each attribute, the rule applies that a value specified at some point overrules the values specified earlier. So if a "[SubHeader]" part provides a value for an attribute, this becomes the new value for the rest of the user dictionary until it is assigned a new value or until the end of the file. This also means that the value specified in the "[Header]" section is an initial value; it is not the default value for all data sections.

There are two exceptions to this rule: the "Name" and "Description" fields apply to the entire dictionary, and they take their value from the last specified value.

Although the format supports several subheaders with different values for the fields, a user dictionary only supports a single value for the "Language" field.

The "Name" and "Description" are free format text fields to be defined by the user. You can use them for your own convenience.

The "Language" value is the three letter language code used by Vocalizer Expressive, e.g.: ENU for "American English". See the **Language Codes** table above for a list.

The values for the "Content" and "Representation" go together:

- Phonetic transcriptions are marked by EDCT_CONTENT_BROAD_NARROWS and EDCT_REPR_SZZ_STRING.
- Orthographic and Pinyin replacements are marked by EDCT_CONTENT_ORTHOGRAPHIC and EDCT_REPR_SZ_STRING.

The "[Data]" part contains one entry per line:

- The key and value are separated by white space. Double quotes can be used around key and value in case they contain spaces. In that case, the backslash can be used as an escape character for a literal double quote and a literal backslash.
- The dictionary keys are case sensitive.
- An orthographic replacement value is a regular text string. A phonetic replacement starts with the tag "//" followed by the transcription in L&H+ format. And a Pinyin replacement is a Pinyin transcription enclosed by the tag “\x11/>”, e.g. “\x11/>zheng4 shang4wu3 \x11/>”.
- For phonetic replacements it’s recommended not to surround the transcription with a silence phoneme /#/, as this is likely to break the prosody when it used within a sentence.

It’s important to put one or more space characters between the key and the value, and not to put unintentionally spaces after the value (as this will be considered part of the value).

Text format specification

```
dictionary:
        header data |
        header (subheader data)+
```

Chapter II

```
header:
       [Header]
       attribute+

subheader:
       [SubHeader]
       attribute+

attribute:
       Name = <string> |
       Language= <3 letter code> |
       Description = <string> |
       Content = [
         EDCT_CONTENT_BROAD_NARROWS |
         EDCT_CONTENT_ORTHOGRAPHIC
       ] |
       Representation = [
         EDCT_REPR_SZZ_STRING |
         EDCT_REPR_SZ_STRING
       ]

data:
       [Data]
       (key value)*

key:
       <string> |
       <quoted string>

value:
       phonetic |
       orthographic

phonetic:
       //<string>

orthographic :
       <string> |
       <quoted string>
```

Example text user dictionary

This is how a text user dictionary might look like:

```
[Header]
Language = ENG

[SubHeader]
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING

[Data]
zero // #'zi.R+o&U#
addr // #'@.dR+Es#
adm  // #@d.'2ml.n$.'stR+e&I.S$n#

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_SZ_STRING

[Data]
Info            Information
IT              "Information Technology"
```

DLL	*"Dynamic Link Library"*
A-level	*"advanced level"*
Afr	*africa*
Acc	*account*
TEL	*telephone*
Anon	*anonymous*
AP	*"associated press"*

Loading user dictionaries

User dictionaries can be loaded for run-time use in two different ways. Vocalizer Expressive supports multiple loaded user dictionaries, and the dictionaries use the binary dictionary format. The load order determines the precedence, with more recently loaded user dictionaries having precedence over previously loaded ones.

The **ve_ttsResourceLoad()** API function is the preferred method for loading user dictionaries. This allows the application to control where each user dictionary is stored.

An alternative method relies on the <RESOURCES> section in the pipeline header of a voice. By default, this section defines 2 user dictionaries named `userdct/<lng>` and `userdct/<lng>/<voice>` (`<lng>` = language code, `<voice>` = voice name). These names map on the files `userdct_<lng>.dat` and `userdct_<lng>_<voice>.dat`. Installing one or both of these user dictionary files next to the language data will have Vocalizer Expressive automatically load it whenever the voice is selected. They are loaded prior to any dictionaries loaded with **ve_ttsResourceLoad()**, so have the lowest precedence.

<install_path>\languages\<lng>\speech\components
  userdct_<lng>.dat
  userdct_<lng>_<voice>.dat

For example:

C:\Program Files\Nuance\Vocalizer Expressive\languages\enu\speech\components
  userdct_enu.dat
  userdct_enu_ava.dat

User Rulesets

Introduction

Rulesets allow the user to specify "search-and-replace" rules for certain strings in the input text. Whereas user dictionaries only support search and replace functionality for literal strings that are complete words or tagged multi-word fragments, rulesets support any search pattern that can be expressed using regular expressions (e.g. multiple words, part of a word).

The rulesets are applied before any other text normalization is performed, including user dictionary lookup. Only a prompt template set as described below may be applied before.

The details of how the text normalization can be tuned via user rulesets are described in the next section.

A ruleset is basically a collection of rules defined in a UTF-8 text file; each rule specifies a "search pattern" and the corresponding "replacement spec".

The syntax and semantics of the "search pattern" and the "replacement spec" match those of the regular expression library that is used, being PCRE v5.0 which corresponds with the syntax and semantics of Perl 5. For the Perl 5 regular expression syntax, please refer to the Perl regular expressions main page at http://perldoc.perl.org/perlre.html. For a description of PCRE, a free regular expression library, see http://www.pcre.org/.

More details on the syntax are described in the "Ruleset format" section.

Rulesets can be loaded by the **ve_ttsResourceLoad()** API function.

The rules of a loaded ruleset are applied only when the active language matches the language that is specified in the header section of the ruleset. Moreover, a user ruleset can be global in scope, or can be restricted to a block of text marked with a particular tn value (with the *<ESC>**tn*\\ control sequence)

Several rulesets can be active simultaneously for the same language. Rulesets bound to a tn value do not take precedence over global rulesets; all rulesets are equal in terms of precedence. However the more recently loaded rulesets take precedence over earlier loaded ones. This is a change compared to the behavior under previous Vocalizer 1.x releases. This change makes it necessary to load rulesets in a specific order in order to get the desired results.

Tuning text normalization via rulesets

The Regular Expression Text-To-Text (RETTT) component applies the rules of the rulesets. The user rulesets are applied before any other text normalization is performed, including user dictionary lookup. The only transformations on the TTS input text that can occur before RETTT processing is the optional transcoding of the input text to the UTF-16 encoding used internally and the application of a prompt template set (cf. below).

There are 2 different kinds of rulesets: typed and untyped rulesets. The untyped rulesets are also known as global rulesets. Typed rulesets are bound to a specific tn value.

During RETTT processing rulesets are applied in reverse order of loading. This means that the most recently loaded rulesets are executed first. Be aware that this simple rule is a change compared to Vocalizer Expressive v1.

In general RETTT applies a ruleset starting with the first rule (at the top of the ruleset), then the second rule, and so on, working its way down in the ruleset until it has applied the last rule (at the bottom of the ruleset). In this way, the output of one rule is the input to the next rule, and a later rule gets to change the text that was already transformed by the previous rules.

To apply a rule RETTT first determines the text scope, i.e. the fragment(s) in the input where the rule is to match and rewrite. Then within those fragments it looks for strings that match the rule's search specification, and it replaces all the occurrences. For rules in a global ruleset the text scope is always the complete input text.

For rules in a typed ruleset the text scope are the text fragments within a matching *<ESC>\tn=<type>*. These fragments may change from one typed rule to the next as a rule may insert *<ESC>\tn=<type-2>* as part of it replacement string (<type> and <type-2> may be identical or different).

When RETTT completes applying the typed ruleset, by default it removes the *<ESC>\tn=<type>* from the text scope. This allows typed rulesets to override or augment Vocalizer Expressive built-in text normalization types and to delegate portions of their processing to other typed rulesets or to Vocalizer Expressive built-in text normalization types. This default behavior is overridden by defining an output type in the ruleset, as explained below.

It's important to note that the above mechanism and the below example can only work if the order of loading of the typed rulesets is respected. In this respect consider the following recommendations for the loading order of rules, especially if you have resources available that you would like to reuse in future projects:

1. Load global rules (untyped rules) before or after the typed rulesets. Be aware that last loaded rules get executed first.
2. Load typed rulesets in reverse dependency order (reverse topological sort order)
    a. If typed ruleset-P inserts <esc>\tn=C\ and in that way delegates rewriting to a ruleset-of type C, then load ruleset-C before ruleset-P.
    b. The dependency graph should be a directed acyclic graph, i.e. without cycles because of a descendant ruleset calling back into an ancestor ruleset.

Consider for instance this typed ruleset:

```
[header]
language = EN*
type = flight

[data]
# Rule 1
/<slot1>([^ ]*)/   -->   "<ESC>\\tn=airport\\$1 <ESC>\\tn=flight\\"
# Rule 2
/<slot2>([^ ]*)/   -->   "<ESC>\\tn=airport\\$1"
```

And this input text

```
<ESC>\tn=flight\Flight from <slot1>"BRU" to <slot2>"LON".
```

RETTT applies rule 1 (`/<slot1>([^ ]*)/ --> “<ESC>\\tn=airport\\$1<ESC>\\tn=flight\\“`):

Text scope:
<ESC>\tn=flight\Flight from <slot1>"BRU" to <slot2>"LON".

Matches:
<ESC>\tn=flight\Flight from <slot1>"BRU" to <slot2>"LON".

Rewritten as:
<ESC>\tn=flight\Flight from <ESC>\tn=airport\"BRU“ <ESC>\tn=flight\ to <slot2>"LON".

Note that this rule reinserts <ESC>\tn=flight\ as part of the replacement string to set the text scope for the typed rule 2 (`/<slot2>([^ ]*)/ --> “<ESC>\\tn=airport\\$1“`)

Text scope:
<ESC>\tn=flight\Flight from <ESC>\tn=airport\"BRU“ <ESC>\tn=flight\ to <slot2>"LON".

Matches:
<ESC>\tn=flight\Flight from <ESC>\tn=airport\"BRU“ <ESC>\tn=flight\ to <slot2>"LON".

Rewritten as:
<ESC>\tn=flight\Flight from <ESC>\tn=airport\"BRU“<ESC>\tn=flight\ to <ESC>\tn=airport\"LON".

The resulting text from this ruleset is

```
Flight from <ESC>\tn=airport\"BRU" to
<ESC>\tn=airport\"LON".
```

Note that RETTT has removed *<ESC>\tn=flight* from the 2 text fragments, and that the ruleset prepares the input to be further processed by another ruleset typed "airport".

Chapter II

Ruleset format

In general, a ruleset is a UTF-8 text file that consists of a header section, followed by a data section. The format of a ruleset is described formally below using the following notation:

Symbol	Meaning
{…}	Optional part; the part between { and } can be occur once but is not required to.
(…)*	The part between (and) can be occur more then once.
<…>	The part between < and > specifies a variable string constant.
A|B	OR part, A is specified or B is specified.

A ruleset can be formally described as:

```
ruleset :=
        (<comment-line>|<blank-line>)*
        <header-section>
        <data-section>?
```

Comment lines have the '#' character as the first non-blank character.

A blank line is a line consisting entirely of linear whitespace characters. Using regular expression syntax they can be expressed as:

```
comment-line :=
  ^\s*#.*\n
blank-line :=
  ^\s*\n
```

Header Section

The "header" section contains one or more key definitions (the definition of the "language" key is required, see further); each definition can span one line only.

```
header-section :=
        "[header]"\n
        (<comment-line>|<blank-line>|
        <key-definition>)+
```

Comment lines and blank lines can be inserted everywhere.

Key definitions have the following syntax:

```
key-definition :=
        <key-name> = <key-value>\n
```

Blanks (spaces or tabs) before and after the equal sign are optional.

If the key value contains blanks, it must be enclosed in double quotes. If a double quote is needed as part of the value, it needs to be escaped (\"). The actual syntax of the <key-value> depends on the <key-name>.

The key-name and key-value are case insensitive. This means that you can specify them in upper case, lower case or a mix of upper and lower case and that this will have no effect on the rewritings.

The only currently supported key names are “language”, “type” and “type_out”. This means that <key-definition> can be expressed semantically as:

```
key-definition :=
        <language-definition>|
        <type-definition>|
        <type_out-definition>
```

The <language-definition> is required for each header, the value is the 3-letter Vocalizer Expressive language code, a language group or the wildcard ‘*’ for specifying all languages. The 3-letter language code is also used to specify the language of user dictionaries, see the **Language Codes** table above for a list.

Note that the “*” used in the following syntax specification designates the literal asterisk character “*”, and not a repetition.

```
language-definition :=
        language = (
          <language-code-list>
          <language-group>
          \*
        )\n
```

```
language-code-list := <language-code>(,<language-code>)*
language-code := ENA|ENG|ENU|DUN|FRC|GED|…
language-group := EN\* | DU\* | FR\* | GE\* | …
```

The type-definition is optional and specifies that the ruleset is scoped to text marked for a particular tn value (with the *<ESC>\tn* control sequence). A ruleset without a type-definition is global, and applies to the entire input.

type-definition := type = <type-name>\n

The type-name is any non-white-space character sequence, and scopes the text in the input on which RETTT applies the ruleset. The beginning of the text scope is marked with the *<ESC>\tn=type* control sequence. For example, a user ruleset with a type-“financial:stocks” is applied to input

[..] *<ESC>\tn=financial:stocks\text under tn value financial:stocks <ESC>\tn=normal\ [..]*.

The type_out-definition is optional on a typed ruleset, and specifies the output type of the typed ruleset.

type_out-definition := type_out = <type_out-name>\n

The type_out-name is any non-white-space character sequence, and it defines the *<ESC>\tn=type_out* control sequence that RETTT inserts at the beginning of the text scope that it has applied the ruleset to. For example, a user ruleset with a type “financial:stocks” and an output type “financial_stock_values” will rewrite the sample input above into

[..] <ESC>\tn= financial_stock_values\ rewritten text originally under tn value financial:stocks <ESC>\tn=normal\ [..]

RETTT replaces the original tn control sequence with one defined by type_out. If type_out is not specified in the ruleset, RETTT removes the original tn control sequence. RETTT ignores the type_out definition in case of a global ruleset, i.e. a ruleset without a type definition.

Data Section

The "data" section contains zero or more "rules", a rule can occupy one line only.

data-section :=
    "[data]"\n
    (<comment-line>|<blank-line>|<rule>)*

Comments can also be inserted at the end of a rule and start with a '#' character and span till the end of the line.

A rule has the following syntax:

rule :=
    <search-spec> "-->" <replacement-spec> <comment>? \n

The syntax and semantics of the <search-spec> match the one of the used regular expression library, being PCRE v5.0, and this corresponds with the syntax and semantics of Perl 5. The PCRE v5.0 lib is compiled in with support for Unicode code properties. For Perl 5 regular expression syntax, please refer to the Perl regular expressions man page at http://perldoc.perl.org/perlre.html. For a description of PCRE, a free regular expression library, see http://www.pcre.org/.

For a detailed description, see the "pcrepattern.html" document in the PCRE distribution package.

If markup is being used (in the source and/or replacement pattern), it must be in the native Vocalizer Expressive markup format.

Note that special characters and characters with a special meaning need to be escaped.

Some examples are:

- In the search pattern: non-alphanumerical characters with a special meaning like dot(.), asterisk (*), dollar ($), backslash (\) and so on, need to be preceded with a backslash when used literally in a context where they can have a special meaning (e.g. use * for *). In the replacement spec this applies to characters like dollar ($), backslash (\) and double quote (").
- Control characters like \t (Tab), \n (Newline), \r (Return), etc.
- Character codes: \xhh (hh is the hexadecimal character code, e.g. \x1b for Escape), \ooo (ooo is the octal notation, e.g. \033 for Escape).
- Perl5 also predefines some patterns like “\s” (whitespace) and “\d” (numeric).

For a full description please refer to the Perl5 man pages.

Rule example

/\bDavid\b/ --> "Guru of the month May"
Replaces each occurrence of the string "David" by "Guru of the month May".

Search-spec

In general the format of the search-spec is:

search-spec :=
<delimiter> <regular-expression> <delimiter> <modifier>*

<delimiter> is usually '/', but can be any non-whitespace character except for digits, back-slash ('\') and '#'... This facilitates the specification of a regular expression that contains '/', because it eliminates the need to escape the '/'.

<modifier> := [imsx]

Optional modifiers:

- i (search is case-insensitive);
- m (let '^' and '$' match even at embedded newline characters);
- s (let the '.' pattern match even at embedded newline characters, by default '.' matches any arbitrary character, except for a newline);

- x (allows for regular expression extensions like inserting whitespace and comments in <regular-expression>).

Replacement-spec

The format of the replacement spec is a quoted ("…") string or a non-blank string in case the translation is a single word. It may contain back references of the form $n (n: 1, 2, 3, ...) which refer to the actual match for the n-th capturing subpattern in <search-spec>. E.g. $1 denotes the first submatch. A back reference with a number exceeding the total number of submatches in <search-spec>, is translated into an empty string. A literal dollar sign ($) must be escaped (\$).

Everything following <replacement-spec> and on the same line is considered as comment when starting with '#', else it is just ignored.

Some rule examples

/<NUAN>/ --> "Nuance Communications"
Rewrites "<NUAN>" into "Nuance Communications".

/(Quack)/ --> ($1)
Replaces "Quack" by "(Quack)".

/(Quack)/ --> ($2)
Replaces "Quack" by "()".

/(\s):-\)(\s)/ --> "$1ha ha$2"
Where "\s" matches any whitespace character,
$1 corresponds with the matched leading whitespace character and $2 corresponds with the matched trailing whitespace character. This rule rewrites for instance " :-) " into " ha ha ".

/(\r?\n)-{3,} *Begin included message *-{3,}(\r?\n)/ --> "$1Start of included message:$2"
Rewrites for instance
---- Begin included message ----
into
Start of included message:

/\x{20AC}?(\d+)\.(\d{2})\d*/ --> "$1 euro $2 cents"
Rewrites for instance "€9.751" into "9 euro 75 cents".

Restrictions on rulesets

The following restriction applies to rulesets: markers generated while rulesets are loaded have source position fields that represent the position after the rulesets have been applied.

Chapter II

Effect of rulesets on performance

The loading of rulesets can affect synthesis processing performance, increasing latency (time to first audio) and overall CPU use. Certain regular expression patterns are more efficient than others, so it is important to carefully consider pattern efficiency while writing rulesets, and to test the system with and without the rulesets to ensure the performance is acceptable.

E.g. a character class (e.g. "[aeiou]") is more efficient than the equivalent set of alternatives (e.g. "(a|e|i|o|u)").

See the "pcreperform.html" main page of the PCRE package for more details.

Loading rulesets

The **ve_ttsResourceLoad()** API function is used to load rulesets for run-time use. Any number of rulesets can be loaded at run-time. The load order determines the precedence, with more recently loaded rulesets having precedence over previously loaded rulesets. The run-time will only apply rulesets that match the language of the current synthesis voice.

Similarly to user dictionaries an alternative way to load a ruleset is to add it to the <RESOURCES> section in the pipeline header of a voice, e.g.

```
<RESOURCE content-type=" application/x-vocalizer-rettt+text;loader=broker">
  rules/enu
</RESOURCE>
```

Vocalizer Expressive will request the Data Access external service for the data named `rules/enu` at the time that the voice is selected. The default Data Access external service maps the name `rules/enu` on the file `rules_enu.dat` and expects to find that file in the Vocalizer Expressive installation directory. The rulesets in the pipeline header are loaded prior to any loaded with **ve_ttsResourceLoad()**, so have the lowest precedence.

Chapter II

Prompt Templates

Introduction

Template based prompt matching is a text-to-text transformation mechanism executed before the “User Rulesets” described above.

The intended use cases are applications that have an exact control over the expected input, e.g. navigation prompts. The main purpose is to:

- Provide perfect control over the expected output
- Scale well with increasing number of templates

Definitions

We call a prompt the entire input fed to Vocalizer through an API call like **ve_ttsProcessText2Speech()**.

We call slot any portion of a string delimited by ‘<’ and ‘>’ not containing any of these delimiters.

We distinguish open slots: ‘<>’, and instantiated slots: ‘<X>’, where X may be any string containing neither ‘<’ nor ‘>’.

A template consists of an input string and an output string.

Matching of prompts

A prompt is checked against the ordered set of templates. Only the first matching template is used to transform the prompt. If no template matches the prompt is left unchanged.

A template matches if its input string matches the prompt. The match is basically done literally, i.e. case sensitive, no white-space stripping, no patterns, no wild-cards etc.

The only exceptions to this are slots: An open slot in a template input string matches any slot in the prompt. An instantiated slot in a template input string only matches a literally identical instantiated slot in the prompt.

Transformation of prompts

If a template matches a prompt the latter is replaced by the template’s output string.

No further template is applied to this prompt. It is, however, subject to all down-stream "usual text normalization" including user dictionaries, user rules, Active Prompt matching etc.

A template's output string may contain anything that constitutes valid input to the TTS engine. The output string is fed down-stream as is with the following exceptions:

- '$n' is replaced by the contents of the n-th slot of the prompt. If there are less than n slots in the template's input string, the template set loading/compilation fails.
- If the output is to contain '$' literally it must be escaped by doubling: '$$'. The template set loading/compilation fails if the output string contains an un-escaped '$' not followed by a number.

Prompt template set format

A prompt template set is represented by a UTF-8 text file that consists of a header section, followed by the templates:

```
PTTfile =
  header
  { template }
.

header =
  "FORMAT"        ":" value
  "VERSION"       ":" value
  "TEMPLATEDELIM" ":" value
  "INOUTDELIM"    ":" value
.

value = string .

template =
  templatedelim
  input
  inoutdelim
  output
.

input = string .
output = string .

string = '"' escapedUtf8String '"'
```

Strings may contain '\x[0-9a-fA-F][0-9a-fA-F]'. On parsing these sequences are transformed to the character corresponding to the hexadecimal number. This is a pure convenience to ease the writing of the text file. The behavior is identical[1] as if the characters are used directly.

Strings are "-delimited. If they are to contain '"' or '\', they must be escaped as '\"' resp. '\\'. On parsing, any '\' that is not preceded by '\' is dropped.

Comments can be denoted with character '#' at any place except inside strings. They range to the end of the line.

Example template set

```
# Example template set
FORMAT:        "Text Template 1.0"
VERSION:       "1.0.0"
TEMPLATEDELIM: "-------"
INOUTDELIM:    "==>"
-------
"In <3m>, turn left into <>."  # one instantiated slot, one open slot
==>
"In \x1b\\audio=c:/home/SampleWaves/3.wav\\meters, turn left into $2."
-------
"In <>, turn left into <Main Street>."
==>
"In $1, turn left into
\x1b\\audio=c:/home/SampleWaves/main_street.wav\\."
-------
"In <>, turn left into <>."
==>
"\x1b\\prompt=nav::in\\ $1, \x1b\\prompt=nav::turn_left_into\\ $2."
-------
"You will arrive in dist<> at destination."
==>
"You will arrive in \x1b\\tn=dist\\$1\x1b\\tn=normal\\ at destination."
-------
"You will arrive in duration<> at destination."
==>
"You will arrive in \x1b\\tn=duration\\$1\x1b\\tn=normal\\ at
destination."
-------
"Hello. This is a very long help text."
==>
"okay"
-------
"okay"
==>
"oops $$99"
```

Remark

The above example suggests in the 4th and 5th template a way how to implement "typed slots": the "<>" slot markup is prefixed by a label ("dist" vs. "duration") in some cases. Note that this is simply a convention a template set designer may adopt. The template matching neither implements nor needs to implement any dedicated mechanism for this.

Commented input output examples

Prompt	In <3m>, turn left into <Main Street>.
Output	In <ESC>\audio=c:/home/SampleWaves/3.wav\meters, turn left into Main Street.
Applying template 1. Note that the input also matches templates 2 and 3.	

[1] The only exception is '\x5c' which behaves like an escaped backslash, i.e. '\\' rather than '\'.

Chapter II

Prompt	In <200m>, turn left into <Main Street>.
Output	In 200m, turn left into <ESC>\audio=c:/home/SampleWaves/main_street.wav\.
Applying template 2.	

Prompt	In <200m>, turn left into <Narrow Road>.
Output	<ESC>\prompt=nav::in\ 200m, <ESC>\prompt=nav::turn_left_into\ Narrow Road.
Applying template 3.	

Prompt	Hello. This is a very long help text.
Output	Okay
Do not apply the last template although it matches "okay".	

Prompt	You will arrive in dist<50m> at destination.
Output	You will arrive in <ESC>\tn=dist\50m<ESC>\tn=normal\ at destination.
Typed slot: "dist<>" transformed to "tn=dist"	

Prompt	You will arrive in duration<50m> at destination.
Output	You will arrive in <ESC>\tn=duration\50m<ESC>\tn=normal\ at destination.
Typed slot: "duration<>" transformed to "tn=duration"	

Compilation of a prompt template set

A template set file, usually identified by the extension ".ptt", may be compiled offline into a representation that is more efficient to load at run-time. Syntax and consistency checks are then also anticipated offline. It is therefore strongly recommended to employ compiled template sets, commonly denoted by the extension ".ptb", for any production use.

A template set can be compiled as follows:

```
ptt2ptb.exe --infile navi.ptt --outfile navi.ptb
```

Prompt templates vs. user rulesets

The functionality of prompt templates may be fully emulated by user rulesets, i.e. it is possible to write for each template set a user ruleset that produces exactly the same input-output behavior. However, the scaling of resource requirements and latency with respect to the number of templates is very different: for the PCRE based ruleset approach the latency scales linearly with the number of rules, whereas for the template approach it only scales logarithmically. In practice this means when rulesets start to introduce noticeable latency for some hundreds of rules, it is possible to achieve fast responses with a set of a million templates or more.

The price of this efficiency gain is obviously the strongly reduced expressivity in the matching: open slots are the only "pattern matching".

Also, not allowing cascading application of multiple template sets may seem as a restriction. However, whereas splitting and cascading of rulesets is good practice, allowing the same with template sets would be inefficient: it is far better to merge all templates into one set.

The simplicity of the template approach may be considered as another advantage: it is for the developer/maintainer/user almost trivial to predict the output for any given input. This is obviously not the case for complex rulesets.

Restrictions on prompt templates

As for user rulesets, the following restriction also applies to prompt templates: markers generated while a prompt template set is loaded have source position fields that represent the position after the template set has been applied.

Loading prompt template sets

The **ve_ttsResourceLoad()** API function is used to load a prompt template set for run-time use. Only one template set can be loaded at a time. The application of a template set does not depend on the language that is currently active.

The corresponding mime-types are "application/x-vocalizer-pt+bin" and "application/x-vocalizer-pt+text" for the compiled resp. textual version of the data.

Similarly to user dictionaries and rulesets an alternative way to load a template set is to add it to the <RESOURCES> section in the pipeline header of a voice, e.g.

```
<RESOURCE content-type="application/x-vocalizer-pt+bin;loader=broker">
  navigation-prompt-templates
</RESOURCE>
```

Vocalizer Expressive will request the Data Access external service for the data named `navigation-prompt-templates` at the time that the voice is selected. The default Data Access external service maps the name `navigation-prompt-templates` on the file `navigation-prompt-templates.dat` and expects to find that file in the Vocalizer Expressive installation directory.

Remark

Currently, loading a compiled template set through the header file requires less heap memory at the expense of some additional data accesses at template application time. In the future, we will provide means to achieve an identical behavior with an explicit API call.

ActivePrompts

Introduction

Nuance Vocalizer Expressive supports tuning Text-To-Speech synthesis through Nuance ActivePrompts. Whereas User Dictionaries and User Rulesets tune the input text, ActivePrompts are meant to tune the synthesis for particular text fragments.

ActivePrompts are created with the Nuance Vocalizer Studio product (a graphical TTS tuning environment) and are stored in an ActivePrompt database for run-time use. There are two types of ActivePrompts:

- Recorded ActivePrompts are digital audio recordings that are used as-is to construct the speech output. They are usually stored together with other recordings in a compressed database (much smaller than individual audio files), or could optionally be stored as individual WAV files.
- Tuned ActivePrompts don't store the actual audio, but rather synthesizer instructions that allow the TTS system to synthesize the prompt in the desired way. These synthesizer instructions are much smaller than the audio that will be produced.

Loading ActivePrompt databases

The **ve_ttsResourceLoad()** API function is used to load ActivePrompt databases for run-time use. Any number of ActivePrompt databases can be loaded at run-time. The load order determines the precedence, with more recently loaded ActivePrompt databases having precedence over previously loaded databases. The run-time will only consult ActivePrompt databases that are activated and match the current synthesis voice.

An ActivePrompts database may also be loaded by adding it to the <RESOURCES> section in the pipeline header of a voice, e.g.

```
<RESOURCE content-type=" application/x-vocalizer-activeprompt-db;loader=broker">
  apdb/rp/sdk/allison/full/gildedphrases/base
</RESOURCE>
```

Vocalizer Expressive requests the Data Access external service for the data `apdb/rp/sdk/allison/full/gildedphrases/base` at the time that the voice is selected. The default Data Access external service maps that name on the file `apdb_rp_sdk_allison_full_gildedphrases_base.dat` and looks for it in the Vocalizer Expressive installation directory. The ActivePrompts databases in the pipeline header are loaded prior to any loaded with **ve_ttsResourceLoad()**, and have lower precedence.

A Recorded ActivePrompts database consists of 2 data components:

1. The ActivePrompts symbolic data define the text, and describe the conditions for using the ActivePrompts. The client needs to copy these data into memory and load them on a TTS instance by passing the memory buffer as an argument to **ve_ttsResourceLoad()**.
2. The ActivePrompts audio is stored in an ActivePrompts speechbase, or as individual audio files. By default the TTS instance looks for the ActivePrompts audio in an ActivePrompts speechbase named after the domain of the ActivePrompts database and after the current voice. If the MIME content type contains a uriprefix or urisuffix attribute, or there's no ActivePrompts speechbase available, it will look for audio files.

For a Tuned ActivePrompts database there is only one data component which stores both the symbolic data and the synthesizer instructions. Again, the client needs to copy these data into memory and load them on a TTS instance by passing the memory buffer as an argument to **ve_ttsResourceLoad()**.

At run-time, all ActivePrompts can be used in two different ways:

- either explicitly inserted using the *<ESC>\prompt=<prompt>* control sequence,
- or by implicit matching where ActivePrompts are automatically used whenever the input text matches the ActivePrompt text.

For implicit matching, the ActivePrompts database can be marked to run in either automatic mode or normal mode. In automatic mode, implicit matches are automatically enabled across all the text in all speak requests. In normal mode, the *<ESC>\domain=<domain>* control sequence must be used to enable implicit matches for specific regions within the input text.

An ActivePrompts database can be marked for automatic mode when it is build, and in this case it will always run in automatic mode. In the other case, you can still mark it to run in automatic mode when you load it by calling the function **ve_ttsResourceLoad()**.

Automatic matching can be further restricted on an ActivePrompts database so it is only done within a text marked with a particular tn value (with the *<ESC>\tn* control sequence). This is useful to work for instance with a recorded ActivePrompts database for spelling that is used exclusively for text wrapped in *<ESC>\tn=spell*.

Multi-lingual voices

Introduction

A multi-lingual (ML) voice is capable of reading text that contains fragments in one or more foreign languages. This voice capability is an asset for instance for a navigation system that the driver relies on while traveling abroad. For instance, the navigation system of a German driver in Spain may want the German voice Anna-ML to read out the following notification:

```
Die Ausfahrt Richtung <ESC>\lang=SPE\ Palma de Mallorca
<ESC>\lang=GED\ kommt nach 120 Metern.
```

Note that the location name "Palma de Mallorca" is entered in its regular written form, and as it is known to be a Spanish name it is tagged as such with the control sequence *<ESC>\lang=SPE*. This control sequence defines the language of the text (Spanish), which is in this case different from the native language of the voice (German). Hence it takes a ML voice to read this navigation message out properly.

Levels of competency in multi-linguality

To read the message with the Spanish location name the German voice Anna-ML relies on these multi-lingual capabilities:

- She has knowledge of the pronunciation rules of the foreign language Spanish. She knows for instance that the Spanish letter combination "ll" like in "Mallorca" sounds like the phoneme /j/.
- She is able to articulate foreign sounds that are outside the German sound repository. For instance she can realize the letter "r" in "Mallorca" as a Spanish /r/ in a way that is pretty close to how a native Spanish speaker pronounces it.

Linguistic knowledge

Orthographic fragments

The knowledge of foreign pronunciation rules is an extension to the linguistic processing capabilities of a voice. This knowledge is required to generate an appropriate phonetic transcription for a foreign orthographic text fragment. A ML voice is able to acquire this knowledge by accessing the language data of the foreign language. A ML voice is typically able to load the language data of one or more foreign languages. This set of foreign languages is predefined per ML voice and documented in the voice-specific supplement.

If a voice does not have this linguistic knowledge about a foreign language, it will read a piece of foreign orthographic text according to the pronunciation rules of its native language. For example, in the case where you deactivate the Spanish linguistic knowledge on the German voice Anna-ML and have her read out the same navigation message

```
Die Ausfahrt Richtung <ESC>\lang=SPE\ Palma de Mallorca
<ESC>\lang=GED\ kommt nach 120 Metern.
```

she will read out “Mallorca” according to the German pronunciation rules (though she knows that it’s a Spanish word from *<ESC>\lang=SPE*), and transcribe the “ll” as /l/.

In practice the foreign linguistic knowledge of a ML voice is limited to a predefined set of languages that form the ML set of the voice.

Phonetic fragments

Even without the linguistic knowledge about a foreign language a voice may still be able to accept a piece of phonetic input in a foreign language, e.g.

```
Die Ausfahrt Richtung <ESC>\lang=SPE\ <ESC>\toi=lhp\
'pal.ma_De_ma.'Jor6.ka <ESC>\toi=orth\ <ESC>\lang=GED\
kommt nach 120 Metern..
```

Knowledge of the phonemes of a foreign language is a basic ML skill that a voice may have. This skill is also limited to a predefined set of languages.

Pronunciation

A sound repository that covers foreign languages is an extension to the acoustic processing capabilities of a voice. A ML voice with this capability knows how to exploit the richness of extended voice data, and is able to articulate foreign sounds. If a voice lacks this extension, it can merely rely on its native sound repository to approximate foreign sounds.

We define 3 levels to express how accurately a voice realizes foreign sounds compared to a native voice of the foreign language: *near native*, *accented* or *basic* pronunciation.

- A ML voice with near native pronunciation reads foreign fragments nearly like a native speaker of the foreign language does.
- A ML voice with accented pronunciation has a clear accent reading foreign fragments.
- And a ML voice with basic pronunciation reads foreign fragments merely using the sounds of its native language.

Voices with a near native or accented pronunciation have a voice name with the suffix “-ML” e.g. Anna-ML is a German ML voice with near native pronunciation in French, English, Italian and Spanish. Whether a ML voice is designed for near native or accented pronunciation is determined by the cultural and market expectations.

Note that a ML voice can have near native pronunciation in a certain set of languages (the ML set) and basic pronunciation in additional languages.

Foreign language proficiency

A ML voice may have different skill levels in linguistic knowledge and in pronunciation, and these combinations define four different levels of competency in multi-linguality. We call these levels *superior*, *enhanced*, *standard* and *basic* foreign language proficiency.

- A ML voice is superior in a foreign language if it has the linguistic knowledge to read orthographic fragments in the foreign language, and pronounce them nearly as well as a native speaker of the foreign language. The voice name has a suffix “-ML”, e.g. German Anna-ML is superior in English.
- A ML voice is enhanced in a foreign language if it has the linguistic knowledge to read orthographic fragments in the foreign language, and pronounce them with an accent. The voice name has a suffix “-ML”, e.g. French Audrey-ML is enhanced in English.
- A ML voice has standard proficiency in a foreign language if it has the linguistic knowledge to read orthographic fragments in the foreign language, but can only approximate the foreign phonemes with the sounds of its native language, e.g. French Thomas has standard proficiency in English.
- A ML voice has basic proficiency in a foreign language if it only has the linguistic knowledge to read phonetic fragments in the foreign language, and articulates the foreign phonemes with the sounds of its native language, e.g. German Anna-ML has basic proficiency in Dutch.

You learn about the ML skills of a voice in the **Language and voice documentation**. For instance, German Anna-ML is superior in French, Italian, English, Spanish and basic in Dutch.

Note that a ML voice is designed to read text in its native language with embedded fragments in a foreign language. To read an entire input text in a foreign language it’s always better to select a native voice for that language.

Language identification

Vocalizer Expressive has a language identification component (LID) that it may call to detect the language of a piece of input text.

By default, the LID is not activated, and a ML voice relies on the user to set the language of foreign fragments tagging them with *<ESC>\lang=<lng>* as in the sample navigation message above. The LID is triggered by the control sequence *<ESC>\lang=unknown*, e.g.

```
Ihre größten Erfolge <ESC>\lang=unknown\(Live in Concert)
<ESC>\lang=normal\
```

The control parameter VE_PARAM_LIDSCOPE defines the text scope for LID. This defines what the pieces of the input are on which LID detects the language. By default, the LID works on the text marked with *<ESC>\lang=unknown* and if this consists of several sentences, it determines the language for each sentence. Alternatively it works on the entire input text, and determines the language per sentence.

How to work with a ML voice

You work with a ML voice as with any other voice in that you configure it on a Vocalizer Expressive instance, then feed it input text and receive the audio stream. The special steps for a ML voice are for you to understand its ML capabilities, to ensure that it has the additional linguistic knowledge loaded for the foreign languages that will be present in the input, and to mark the fragments of foreign text in the input.

Learn about the ML capabilities of your ML voice

Consult the **Language and voice documentation** to find the foreign languages that the voice can load, and the type of pronunciation that it can realize. This tells you what foreign input you can input to the voice.

Ensure that foreign linguistic knowledge is activated

The ML voices with the suffix “-ML” in their name, e.g. German Anna-ML, come with extended voice data and offer better than basic pronunciation of foreign languages. For these voices Vocalizer Expressive loads the foreign linguistic knowledge by default, and to work with them you don’t need to take any other action than to select them.

When you select a ML voice without the “-ML” prefix, e.g. German Anna, you explicitly load the linguistic knowledge for one or more of the supported foreign languages by calling the function

ve_ttsSetParamList() with the parameter VE_PARAM_EXTRAESCLANG passing it a string value like “eng,iti,spe”. This string argument is a comma-separated list of language codes, and it defines the foreign languages that may be present in the input. On this call Vocalizer Expressive will load the foreign language data on the voice (at an additional cost of some 400kB heap per foreign language).

If a ML voice is activated on foreign languages, it will also make use of foreign user dictionaries. In particular it may look up words both in the foreign user and in the user dictionaries of the native language of the voice:

- If a foreign language is tagged in the input by *<ESC>\lang=<lng>* Vocalizer Expressive looks up words in the user dictionaries of the language <lng>.
- If the foreign language is detected the LID (e.g. via *<ESC>\lang=unknown*), then Vocalizer Expressive consults the user dictionaries of that detected language and the user dictionaries of the native language.

User dictionaries remain prioritized in the usual way, i.e. the most recently loaded user dictionary is consulted first.

In addition a user dictionary can be loaded on a voice such that it overrides the language of the text for the words covered by its entries. You may want to do this for a word like “featuring” that you always want read out in the English way no matter what context it appears in. You mark a user dictionary as such by adding “;mode=langoverwriting” to the MIME type passed in **ve_ttsResourceLoad()**.

Ensure that domain-specific linguistic knowledge is activated

For the ML voices with suffix “-ML” Vocalizer Expressive loads the linguistic knowledge for the MP3 domain by default. This directly improves their skills to read text marked with *<ESC>\tn=mpthree*.

When you select another ML voice you explicitly load the MP3 linguistic knowledge by calling the function **ve_ttsSetParamList()** with the parameter VE_PARAM_EXTRAESCTN passing it the string value “mpthree”. On this call Vocalizer Expressive will load the MP3 language data on the voice.

Mark pieces of foreign text in the input

You tag pieces of foreign text in the input with the control sequence *<ESC>\lang=<lng>*. If you know the language of the text you set <lng> to the 3-letter language code (case-insensitive) to mark the beginning and you set <lng> to “normal” to mark its end, e.g.

<text-in-native-language-of-the-voice> <ESC>\lang=eng\ <text-in-eng> <ESC>\lang=normal\ <text-in-native-language-of-the-voice>.

If you don't know the language of the text, set <lng> to "unknown" for Vocalizer Expressive to detect it automatically, e.g

<text-in-native-language-of-the-voice> <ESC>\lang=unknown\ <text-in-a-foreign-language> <ESC>\lang=normal\ <text-in-native-language-of-the-voice>

If you enter a piece of foreign phonetic input, first set *<ESC>\lang=<lng>* and then set *<ESC>\toi=<phon>*, e.g.

Follow the direction <ESC>\lang=frf\ <ESC>\toi=lhp\ #buR.'ZE%~# <ESC>\toi=orth\ <ESC>\lang=enu\ for 5 kilometer.

If the voice does not support the foreign language, Vocalizer Expressive logs a warning and it drops the phonetic fragment, or falls back on the orthographic form if this is supplied in the *<ESC>\toi=<phon>:<orth>*.

If you enter a piece of orthographic input in a foreign language not supported by the voice, Vocalizer Expressive logs a warning and the voice ignores the *<ESC>\lang=<lng>* reading the foreign orthographic fragment as a fragment in its native language.

Alternatively to tagging the language of the text explicitly, you can also let LID run sentence by sentence calling the function **ve_ttsSetParamList()** with parameter VE_PARAMETER_LIDSCOPE.

How to work with native and foreign phonetic input

Phonetic text may be embedded in the input, and it is tagged with *<ESC>\toi*. The voice handles such phonetic text taking one of three possible actions:

- The voice reads the phonetic text.
- The voice drops the entire phonetic text.
- The voice reads the orthographic counterpart (if this is given in the input).

The choice between these three actions depends on the (syntactical) correctness of the phonetic text, and the presence of an orthographic counterpart.

The (syntactical) correctness of the phonetic text

Phonetic text should be composed of phonetic symbols of the language of concern. This set of phoneme symbols for a language is given in the **Language and voice documentation**. It is syntactically incorrect if it contains one or more symbols outside of this set.

The language of the phonetic text is either the native language of the voice (default) or a foreign language.

- Phonetic text in the native language should be composed of phonetic symbols of the native language. A voice always supports phonetic text in its native language.
- The language of the phonetic text can be tagged with *<ESC>\lang=<lng>*.as being different from the native language of the voice. In that case the phonetic text should be composed of phonetic symbols of the foreign language. A voice supports foreign phonetic text in each language of the Vocalizer Expressive portfolio (through cross-language mapping).

The presence of an orthographic counterpart

Phonetic text can have an orthographic counterpart in the input. In that case it's a phonetic + orthographic fragment *<ESC>\toi=lhp:"orth_text"\ phon_text <esc>\toi=orth*.

The following sections describe how a voice handles phonetic input under these different conditions.

1 Correct phonetic text

The voice reads the native phonetic transcription.

Example: Native phonetic text

- Input: <esc>\toi=lhp:"Seoul Bahnhof"\ '?ERnst_'?a.b$_'StRa:.s$ <esc>\toi=orth\
- Voice: GED Anna
- In GED the transcription /'?ERnst_'?a.b$_'StRa:.s$/ is valid, so GED Anna reads /'?ERnst_'?a.b$_'StRa:.s$/

Example: Foreign phonetic text

- Input: <esc>\lang=ged\ <esc>\toi=lhp:"Seoul Bahnhof"\ '?ERnst_'?a.b$_'StRa:.s$ <esc>\toi=orth\
- Voice: ITI Alice
- The transcription /'?ERnst_'?a.b$_'StRa:.s$/ is tagged as German by <esc>\lang\, and it is valid in German. ITI Alice supports a German phonetic transcription (through CLM from GED to ITI), hence reads the phonetic transcription.

2 One or more invalid phoneme symbols

2.1 No orthographic counterpart

The voice drops the entire foreign phonetic text.

2.2 Orthographic counterpart

This case corresponds to a phonetic + orthographic fragment *<ESC>\toi=lhp:"orth_text"\ phon_text <esc>\toi=orth*.

The voice reads the orthographic counterpart (orth_text) of the phonetic transcription (phon_text).

Example:

- Input: <esc>\lang=ged\ <esc>\toi=lhp:"Seoul Bahnhof"\ 'sO.ul.ljOk <esc>\toi=orth\
- Voice: ITI Alice
- The language of the transcription /'sO.ul.ljOk/ is tagged as German by <esc>\lang\, but the L&H+ symbol /u/ is invalid for German, So, ITI Alice falls back to reading the orthographic counterpart "Seoul Bahnhof".

Traversing through the input

Introduction

The support for traversing through the input text is feature that allows you navigating forwards and backwards in the input text. This proves useful to let the listener navigate through a longer input text like a news item, an elaborate description of a place of interest, an e-mail message or even an e-book chapter. It avoids the need to have Vocalizer Expressive synthesize the audio for the entire input text. Instead it lets you direct Vocalizer Expressive to a particular point in the input text and start reading from there.

Basics of traversing

There are 2 steps that you need to take. First you let Vocalizer Expressive analyze the input text and you collect a list of possible jump points. And then you tell Vocalizer Expressive to synthesize starting at the jump point of your choice.

Text analysis

In the text analysis phase you call the API function **ve_ttsAnalyzeText()**. This makes Vocalizer Expressive generate text analysis (TA) info about the input text. The TA info basically describes the locations in the input text where you can later navigate to. These locations are either sentence boundaries or bookmark locations (defined by the control sequence *<ESC>\mrk=<nr>*).

The granularity of these jump points is restricted to a sentence, which is the basic prosodic unit. In that way Vocalizer Expressive can start at a given jump point and read the text at that point in the same way as if it had arrived there reading the input text from the very beginning. Note that you can break this behavior by inserting a bookmark in the middle of a sentence and using that as a jump point. In that case Vocalizer Expressive will read the tail of the sentence as a sentence on its own, and this probably sounds unnaturally. So we recommend putting navigation bookmarks on sentence boundaries.

Vocalizer Expressive delivers the TA info through the same Output Delivery service that it uses for the audio. The client has to provide the memory to store the TA info, it must keep track of the TA info to let Vocalizer Expressive jump in the second phase and it is responsible for freeing the memory afterwards.

Jump and synthesize

In this phase you typically start off calling **ve_ttsProcessText2Speech()** to have Vocalizer Expressive synthesize from the beginning of the input text. As it synthesizes, it transfers the audio and markers. Based on the text element markers you can track progress in the TA info.

When you receive from the user the instruction to jump forward or backward, you abort the current synthesis request. Then you determine the point to jump to based upon the current position, and you call the API function **ve_ttsProcessText2SpeechStartingAt()** to have Vocalizer Expressive synthesize from that particular location.

Traversing and control sequences

As Vocalizer Expressive navigates to a jump point and starts reading from there, it skips the text from the beginning up to the jump point. That skipped text may contain control sequences, and these impact the state of Vocalizer Expressive at the jump point. For instance consider that you have Vocalizer Expressive jump to the second sentence in this input:

```
Normal and <ESC>\vol=90\ louder. Still louder here.
```

At that location the volume level is still at 90, set to that value by *<ESC>\vol=90* in the first sentence.

Vocalizer Expressive is able to track a limited number of control sequences up to a jump point: *<ESC>\vol=<level>*, *<ESC>\rate=<level>*, *<ESC>\pitch=<level>* and *<ESC>\wait=<value>*. It embeds in the TA info handles to the state per jump point, and it preserves this state info for the last call to **ve_ttsAnalyzeText()**.

Chapter II

Traversing and user rulesets

The RETTT component, which puts user rules into effect, suffers from a known limitation: it is unable to tell how the rewritten text maps onto the original input text. As a consequence, when you load user rulesets on Vocalizer Expressive, it delivers markers and TA info that reference into the rewritten text instead of into the input text.

That is why Vocalizer Expressive also delivers the rewritten text next to the TA info as part of the text analysis results. This allows you selecting a jump point in the rewritten text, and passing the rewritten text into **ve_ttsProcessText2SpeechStartingAt()**. This function makes sure not to exercise the user rules a second time.

Vocalizer Expressive 2.0

Chapter III

Text-To-Speech System Reference

User's Guide and
Programmer's Reference
Revision 1

Chapter III

Text-To-Speech System Reference

Introduction

This chapter gives general information on how to install Nuance Vocalizer Expressive and use the API in an application.

Multiple-language and multiple-voice support

Introduction

Nuance Vocalizer Expressive contains two types of components:

- Language-independent code components (e.g. the Vocalizer Expressive API component and the text preprocessor component)
- Language and voice-dependent data components (language and voice dependent data files)

The user can switch at run-time from one language to another or from one voice to another by calling the appropriate API function.

Installation requirements

In its default deployment configuration Vocalizer Expressive consists of three types of files:

- Run-time components (DLLs, SOs) for the code components
- Binary files for the data components
- System information files (broker header files), which describe the data components

This implies that the target architecture must have the following properties:

- It needs a way to access Vocalizer Expressive data components, either from a file system or from in-memory.
- It supports run-time components.

In the reference deployment configuration the components are installed into several directories:

- A common directory for the code components

- Zero to many language directories for the data components

Application development

Unicode support

This version of Vocalizer Expressive has a char only interface. Strings like language and voice names, which are passed between the client and the Vocalizer Expressive API functions, are of type char

The character encoding of the input text must be platform-endian UTF-16 (the default) or UTF-8 (optionally configured via the **ve_ttsSetParamList()** API call).

Error tracing

The API functions return an error code from a limited set of distinctive error codes. In general, an API function returns the code NUAN_OK if it completes successfully. If it was unable to execute the request, it returns an error code.

The error codes are primarily designed for developers. The codes give useful information as to the condition that triggered the error. When the client gets an error code at run-time (as opposed to at development-time) it can only take action on a few typical error codes. This is explained in detail in **Chapter IV: Text-To-Speech Function Reference**.

When an API call returns an error code, the TTS instance, upon which the client called the API function, is left in a valid state, and thus ready to accept a next request. So the client does not need to reinitialize upon an error.

Double-call functions

The Vocalizer Expressive API functions that query for information about the system, are double-call functions. This means that the client typically makes two consecutive calls:

- With the first call the client retrieves the size of the result set, for which it must allocate space.
- Then with a second call the client provides the allocated space and gets the actual result set copied in it.

For more information on double-call functions, please refer to **Chapter IV: Text-To-Speech Function Reference**.

Chapter III

Basic call sequence

The sequence below gives a schematic overview of the calls that are essential to have a simple speak request work correctly.

- **ve_ttsInitialize()** creates and initializes the Vocalizer Expressive class.
- **ve_ttsOpen()** creates a Vocalizer Expressive object on the class.
- **ve_ttsSetParamList()** sets the language and voice, and other control parameters such as volume, pitch and rate level on the object.
- **ve_ttsSetOutDevice()** associates with the object an output device that it will transfer the synthesized speech signal to.
- **ve_ttsProcessText2Speech()** has the object synthesize a speech signal for the given input text.
- **ve_ttsClose()** closes the object.
- **ve_ttsUnInitialize()** closes the class.

For more details on the overall implementation or on the use of the individual TTS API calls, refer to **Chapter IV: Text-To-Speech Function Reference.**

Multiple instances

Vocalizer Expressive supports multiple open instances (Vocalizer object as created by **ve_ttsOpen()**) at a time.

Multi-threading

Vocalizer Expressive is thread-safe to run several simultaneous speak requests, each in its own thread. But the client is required to attach a Critical Sections service to Vocalizer Expressive.

Asynchronous API

The Vocalizer Expressive API has only one asynchronous function:

- **ve_ttsStop()**

This function returns immediately before completing the real task, and Vocalizer Expressive later generates an event to notify the completion. The client should only take the next action after it receives that message.

Chapter III

Broker header files

Introduction

Vocalizer Expressive expects the client to inform it about the supported product configurations. This is in particular the set of languages, voices and voice operating points that are available from the deployed application.

In the reference deployment configuration of the Vocalizer Expressive software each product configuration is described by a separate *pipeline broker header file* (shorthand: pipeline header). This is an XML text file with extension `.hdr` describing the processing pipeline for a particular language, voice and voice operating point.

When the client creates the TTS class (through the function **ve_ttsInitialize()**), it must pass a string argument that concatenates all the pipeline headers that Vocalizer Expressive must take into account.

A broker header file always starts with the following XML tags:

```
<?xml version="1.0"?>
<NUANCE>
<VERSION>NUAN_1.0</VERSION>
<HEADER>
```

And it ends with the closing tags:

```
</HEADER>
</NUANCE>
```

The order of the elements within the HEADER element is of no importance. These elements usually describe properties of the concerned component.

A required element is BROKERSTRING. This element defines the name of the pipeline.

For example:

```
<BROKERSTRING>
  pipeline/American English/samantha/22/embedded-
compact/text/pcm
</BROKERSTRING>
```

This is the name of the processing pipeline for the Embedded Compact voice operating point of Samantha.

The pipeline header

This file defines the *processing pipeline*, a particular sequence of components that execute the Text-To-Speech conversion. Each distinct pipeline has its own pipeline broker header file, which specifies the sequence as well as the configuration of parameters like the voice. This is an example for the Embedded High voice operating point (embedded-high) of Ava:

```
<?xml version="1.0"?>
<NUANCE>
<VERSION>NUAN_1.0</VERSION>
<HEADER>
  <BROKERSTRING>pipeline/American English/ava/22/embedded-
high/text/pcm</BROKERSTRING>
  <PRIORITY>4315</PRIORITY>
  <PARAMETERS>
    <language>American English</language>
    <langcode>ENU</langcode>
    <langid>10</langid>
    <langversion>5.2.4.15083</langversion>
    <nativetypeofchar>utf-16</nativetypeofchar>
    <voice>Ava</voice>
    <voiceversion>5.2.4.13289</voiceversion>
    <gender>Female</gender>
    <age>Adult</age>
    <fecfg>cfg4</fecfg>
    <voiceml>no</voiceml>
    <mlset>enu,frc,spm</mlset>

<extclccfg>*+mpthree=clc/enu/mpthreevadml,frc+*=clc/frc/cfg3,s
pm+*=clc/spm/cfg3</extclccfg>
    <datapackagename>enu/ava/embedded-high</datapackagename>
    <fedataprefix></fedataprefix>
    <feextcfgdataprefix></feextcfgdataprefix>
    <fedatapackaging>clc</fedatapackaging>
    <fevoice>Ava</fevoice>
    <frequencyhz>22050</frequencyhz>
    <voicemodel>full_155mrf22</voicemodel>
    <voiceoperatingpoint>embedded-high</voiceoperatingpoint>
    <reduction>full</reduction>
    <coder>155mrf22</coder>
    <bitrate>270</bitrate>
    <overheadframes>06</overheadframes>

<audiooutputmimetype>audio/L16;rate=22050</audiooutputmimetype
>
    <audiooutputsamplebits>16</audiooutputsamplebits>
    <typeofsynthesis>psola</typeofsynthesis>
    <nlucompatvc6be>yes</nlucompatvc6be>
    <compatstrongbnd>no</compatstrongbnd>
  </PARAMETERS>
  <OBJECTS>
    <AUDIOFETCHER>audiofetch</AUDIOFETCHER>
    <DCTEG>dcteg</DCTEG>
    <DOMAINMNGR>domain_mngr</DOMAINMNGR>
    <FE_DCTLKP>fe/fe_dctlkp</FE_DCTLKP>
    <FE_DEPES>fe/fe_depes</FE_DEPES>
    <INET>inetspi</INET>
    <PHONMAP>phonmap</PHONMAP>
  </OBJECTS>
  <COMPONENTS>
    <COMPONENT>xcoder</COMPONENT>
    <COMPONENT>ttt/re</COMPONENT>
    <COMPONENT>pp/text_parser</COMPONENT>
    <COMPONENT>pp/sent_parser</COMPONENT>
    <COMPONENT>pp/word_parser</COMPONENT>
    <COMPONENT>fe/fe_lid</COMPONENT>
    <COMPONENT>fe/fe_voice_switch</COMPONENT>
    <COMPONENT>fe/fe_udwl</COMPONENT>
    <COMPONENT>fe/fe_clcml</COMPONENT>
    <COMPONENT>fe/fe_promptoriorth</COMPONENT>
    <COMPONENT>fe/fe_sprop</COMPONENT>
```

```
    <COMPONENT>fe/fe_unixlit</COMPONENT>
    <COMPONENT>fe/fe_initlingdb</COMPONENT>
    <COMPONENT>fe/fe_promptorth</COMPONENT>
    <COMPONENT>fe/tokentn</COMPONENT>
    <COMPONENT>fe/fe_abbrtn</COMPONENT>
    <COMPONENT>fe/fe_puncsptn</COMPONENT>
    <COMPONENT>fe/fe_oneword</COMPONENT>
    <COMPONENT>fe/fe_pos</COMPONENT>
    <COMPONENT>fe/fe_hmogrph</COMPONENT>
    <COMPONENT>fe/fe_phrasing</COMPONENT>
    <COMPONENT>fe/fe_normout</COMPONENT>
    <COMPONENT>fe/fe_prmfx</COMPONENT>
    <COMPONENT>fe/fe_prompt</COMPONENT>
    <COMPONENT>fe/fe_global</COMPONENT>
    <COMPONENT>fe/be_adapt</COMPONENT>
    <COMPONENT>be/featextract</COMPONENT>
    <COMPONENT>uselect/bet1</COMPONENT>
    <COMPONENT>synth/bet1</COMPONENT>
    <COMPONENT>audioinserter</COMPONENT>
    <COMPONENT>phonmap/mrk</COMPONENT>
    <COMPONENT>xcoder/mrksync</COMPONENT>
  </COMPONENTS>
  <RESOURCES>
    <!-- The next entries specify default broker strings for
the user dictionary resources, once language specific and once
voice specific -->
    <RESOURCE content-type="application/edct-bin-
dictionary;loader=broker">
      userdct/enu
    </RESOURCE>
    <RESOURCE content-type="application/edct-bin-
dictionary;loader=broker">
      userdct/enu/ava
    </RESOURCE>
    <!-- The next entries specify default broker strings
tuning resources if available -->
    <RESOURCE content-type="application/x-vocalizer-
activeprompt-db;loader=broker;mode=automatic">
      apdb/rp/sdk/ava/full/gildedphrases/base
    </RESOURCE>
  </RESOURCES>
</HEADER>
</NUANCE>
```

The PRIORITY element is useful when more than one voice is installed for a language. If only the language is specified in the argument list of the **ve_ttsSetParamList()** function, then the voice with the highest priority is loaded. The highest priority is 65535 and the lowest is 0. If this element is not defined, or the voices have the same priority, then it is undetermined which voice will be loaded by default.

The PARAMETERS element specifies internal Vocalizer Expressive parameters that are set when the voice is loaded. Most of these parameters should not be changed, or should only be changed through the **ve_ttsSetParamList()** API call.

The COMPONENTS and OBJECTS elements define the code components that are used to execute the pipeline. The proper selection and ordering of these components is very important. These sections should not be changed except when requested by Nuance.

The RESOURCES element defines data components that are used to execute the pipeline. Unless requested by Nuance, the Nuance supplied default entries should be left as-is, with entries appended as needed.

- RESOURCE elements with a content-type of `application/edct-bin-dictionary` are used to support user dictionary loading. These elements should not be changed except when requested by Nuance. Example:

  ```
  <RESOURCE content-type="application/edct-bin-
  dictionary;loader=broker">userdct/enu</RESOURCE>
  ```

  ```
  <RESOURCE content-type="application/edct-bin-
  dictionary;loader=broker">userdct/enu/ava</RESOURCE>
  ```

  This offers the possibility to load a language and a voice specific user dictionary. Be aware of the implicit priority: first language-specific, then voice-specific. The last user dictionary (voice-specific) has priority over the previous one (language-specific).

An integration engineer may want to change the pipeline broker header files for these reasons:

- Prioritize the different voices for the same language via the PRIORITY element in the pipeline broker header files. See also **The pipeline broker header file** section.
- Tune certain TTS parameters via the PARAMETERS subelements in the pipeline header file for each voice. See also **The pipeline broker header file** section.

The logging header

This is an optional file that defines the settings for the different log subscribers that are built into Vocalizer Expressive:

```
<?xml version="1.0"?>
<NUANCE>
<VERSION>NUAN_1.0</VERSION>
<HEADER>
  <BROKERSTRING>logging</BROKERSTRING>
  <LOG.DIAGNOSTIC>true</LOG.DIAGNOSTIC>
  <LOG.DIAGNOSTIC.TOSTDOUT>false</LOG.DIAGNOSTIC.TOSTDOUT>
  <LOG.DIAGNOSTIC.DIR></LOG.DIAGNOSTIC.DIR>
  <LOG.DIAGNOSTIC.FILEMAXSIZEMB>50</LOG.DIAGNOSTIC.FILEMAXSIZEMB>
  <LOG.DIAGNOSTIC.LEVEL>0</LOG.DIAGNOSTIC.LEVEL>
  <LOG.DIAGNOSTIC.FILEMIMETYPE>
    text/plain;charset=iso-8859-1</LOG.DIAGNOSTIC.FILEMIMETYPE>
</HEADER>
</NUANCE>
```

For instance, to enable the diagnostic logger in the reference deployment configuration of the Vocalizer Expressive software you create the directory ./common/speech/ve and copy the file ve_logging.hdr from the ./sample directory there

Vocalizer Expressive 1.4

Chapter IV

Text-To-Speech Function Reference

User's Guide and
Programmer's Reference
Revision 1

Chapter IV

This Function Reference offers an exhaustive description of the Application Programming Interface (API) for Nuance Vocalizer Expressive. This API is defined in ve_ttsapi.h within the Nuance Vocalizer Expressive package.

The section **Function Directory** is an alphabetical reference of all the API functions. Each separate entry gives an operative description, syntax, parameters, return values, optional comments and a list of related functions.

The section **Structures and Type Definitions** is a detailed list of the type and structure definitions

The section **Return and Error Codes** is a detailed list of error code definitions.

The section **Notification Messages** is a detailed list of notification messages delivered during synthesis.

After reading these sections you should be able to use Nuance Vocalizer Expressive in applications.

For more information on the use of the Text-To-Speech (TTS) system, refer to **Chapter III: Text-To-Speech System Reference** of this manual.

Chapter IV

Text-To-Speech Function Reference

Function directory

This section describes the functions of the Text-To-Speech (TTS) API and the interface functions to the external services in alphabetical order.

For each function the following information is supplied:

- Description
- Syntax
- Parameters
- Return values
- Notification messages
- Comments on implementation issues
- See also reference to related functions

For each parameter, an attribute indicates the direction in which the data is passed:

- [in]: the argument is passed by the application (read-only access).
- [out]: the argument is passed to the application (write-only access).
- [in, out]: the argument is passed by the application at the entry of the function and to the application at the exit of the function (read-write access).

External services

Vocalizer Expressive relies on a number of services that the user needs to implement, and therefore are called external services. These external services are abstractions of platform resources, and they allow the user to select an implementation that best suits the target application and platform.

An external service basically is a collection of callback functions. The TTS class and/or its instances call these functions for particular requests, and thus pass control to the user-defined implementation.

On each function call they pass a handle of the external service. This makes the template interface of an external service look like this:

```
typedef void* VE_EXTERNAL_HINSTANCE

struct VE_EXTERNAL_INTERFACE_S {
  NUAN_ERROR pfRequest_1(
    VE_EXTERNAL_HINSTANCE hExt,
    …
  );
  …
}

typedef struct VE_EXTERNAL_INTERFACE_S
  VE_EXTERNAL_INTERFACE;
```

The different external services are the Heap service, the Critical Sections service, the data access services Data Streams and Data Mappings, the User Log service and the Output Delivery service. Their interfaces are documented in a separate section after the API functions.

Important remarks

Error handling

All Vocalizer Expressive API functions return a code of type NUAN_ERROR. In general, an API function returns the code NUAN_OK if it completes successfully. If not, it returns an error code.

The section **Return and Error Codes** contains the list of error codes with a general indication of the condition that may trigger it. This list is exhaustive in that it covers all possible error codes from all API functions. This does not mean however that each API function may return each of the error codes. The next section **API functions** gives for each API function the typical errors and possible user action under Return values.

When a TTS instance returns an error code indicating that it was unable to execute the request, it remains in a valid state, and thus is ready to accept a next request. So the client does not need to reinitialize upon an error.

There is one exception for the code NUAN_E_OUTOFMEMORY. The TTS instance returns this error code when it failed to allocate memory, and this blocks it further operation. This has to be considered as a fatal error, and the client has to reinitialize.

Asynchronous functions

The following API function is asynchronous:

- **ve_ttsStop()**

This function returns before the actual synthesis task is stopped and completes. Vocalizer Expressive may generate further synthesis results, wait for **ve_ttsProcessText2Speech()** to return before proceeding with actions for new synthesis operations.

Double-call functions

The Vocalizer Expressive API functions for querying system information are double-call functions. That means the application typically makes two consecutive calls to receive the desired information. This protocol makes the application responsible for providing memory for the information, with the API function only responsible for copying in the requested information.

- With the first call the application gets the size of the information so it knows the amount of elements it has to allocate.
- After it has allocated the memory, the application calls the function a second time to fill the allocated space with the required data.
- As soon as the application is done with the buffer, it can free the memory again.

This is a sample prototype of a double-call function, where *pVoiceList* is the application provided buffer and *pusNbrOfElements* is the buffer length in elements (**not** bytes):

NUAN_ERROR
ve_ttsGetVoiceList (
const VE_HSPEECH *hTtsCl,*
const char * *szLanguage,*
VE_VOICEINFO **pVoiceList* ,
unsigned short * *pusNbrOfElements*
)

- If the buffer (in this case *pVoiceList*) is NULL, on output the element size argument (in this case **pusNbrOfElements)* is filled with the required size in elements (in this case VE_VOICEINFO structures), and the function returns NUAN_OK
- If the buffer (in this case *pVoiceList*) is non-NULL, on input the element size argument (in this case * *pusNbrOfElements*) contains the allocated size for the buffer in elements (in this case VE_VOICEINFO structures). If the element size is sufficient for the request, the function fills the buffer with the requested data, then sets the element size argument to the actual number of elements copied (which could be smaller than the provided buffer size). If the element size is too small, the function fills the buffer with the amount of the requested data that fits, and then sets the element size

argument to the required size in elements for the full request data.

Chapter IV

API functions

ve_ttsAnalyzeText

Description

The function **ve_ttsAnalyzeText()** scans an input text and generates text analysis (TA) info and the text rewritten by loaded user ruleset. The TA info describes the locations in the input text that the client may navigate to later calling the function **ve_ttsProcessText2SpeechStartingAt()**. It gives the type of such a location (begin of a sentence, or bookmark), its position and the detected language at that position.

The function **ve_ttsAnalyzeText()** uses the Output Delivery callback function to transfer the TA info and the rewritten input text. Both data streams are to be collected and passed to **ve_ttsProcessText2SpeechStartingAt()** when Vocalizer Expressive is to jump to a given position.

Syntax

NUAN_ERROR
ve_ttsAnalyzeText(
  VE_HINSTANCE *hTtsInst*,
  const VE_INTEXT * *pInText*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pInText	[in] Structure describing the input text.

Return Values

The return value NUAN_OK indicates that the function was successful in scanning the input text.

The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

Notification Messages

VE_MSG_TAIBEGIN	Begin of text analysis.
VE_MSG_TAIEND	End of text analysis.
VE_MSG_TAIBUFREQ	Request for output

	buffers.
VE_MSG_TAIBUFDONE	Ready with an output buffer.

This function delivers these notification messages in a similar way that the function **ve_ttsProcessText2Speech()** delivers notification messages.

Comments

The structure VE_ OUTTAINFO is used to transfer the generated TA info and rewritten text to the client. To this purpose it calls the Output Delivery service to deliver the output flows block by block in the same way that **ve_ProcessText2Speech()** does this to transfer audio and markers.

The client is to cast to VE_ OUTTAINFO the member (void * pParam) of the VE_CALLBACKMSG structure on receiving a block of data by the Output Delivery callback.

For a description of the callback messages, see the **Notification messages** subsection in the **Data types, structures and type definitions** section.

The structure VE_ OUTTAINFO is defined as:

```
typedef struct {
  size_t      cntRewrittenTextLen;
  short     * pRewrittenTextBuf;
  size_t      cntTaInfoListLen;
  void      * pTaInfoList;
} VE_OUTTAINFO;
```

When the application gets the message VE_MSG_TAIBUFREQ, it has to allocate memory for the output data buffer *pRewrittenTextBuf* and fill *uRewrittenTextLen* with the size (in bytes) of this buffer. Also, the application has to allocate memory for the array of jump points (of type VE_TA_NODE) *pTaInfoList* and fill in *uTaInfoListLen* the size (in bytes) of the allocated buffer.

When the application gets the message VE_MSG_TAIBUFDONE, *uRewrittenTextLen* contains the size (in bytes) of the data copied in the output data buffer. *uTaInfoListLen* contains the size (number of markers) copied in the marker array.

See also ve_ttsProcessText2SpeechStartingAt()

Chapter IV

ve_ttsClose

Description

The function **ve_ttsClose()** closes a TTS instance and frees all resources allocated for the instance. If the user calls this function during synthesis an error code will be returned.

Syntax

NUAN_ERROR
ve_ttsClose(
  VE_HINSTANCE *hTtsInst*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_WRONG_STATE: the TTS instance is still busy executing another API call. Wait until that is done.

See also ve_ttsOpen()

ve_ttsGetAdditionalProductInfo

The function **ve_ttsGetAdditionalProductInfo()** returns the date and a possible identifier string for the product build in a VE_ADDITIONAL_PRODUCTINFO structure.

Syntax

NUAN_ERROR
ve_ ttsGetAdditionalProductInfo (
VE_ADDITIONAL_PRODUCTINFO * *pProductInfo*
)

Parameters

pProductInfo	[out] A pointer to the structure that will be filled with the product information.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

Chapter IV

ve_ ttsGetClmInfo

Description

The function **ve_ttsGetClmInfo ()** returns info on the supported cross-language mappings (CLM). Vocalizer relies on these data to read phonetic text in a foreign language.

Syntax

NUAN_ERROR
ttsGetClmInfo(
```
 const VE_HSPEECH     hSpeech,
 const char          * szLanguage,
 VE_CLMINFO          * pClmInfo
)
```

Parameters

hSpeech	[in] Handle to the TTS class of concern.
szLanguage	[in] A zero terminated string indicating the target language for which we want to know the CLM info.
pClmInfo	[out] Address of a variable of type VE_CLMINFO. On output, the function fills the variable with the necessary CLM info.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_BUFFERTOOSMALL: There was not enough space in an array argument for the function to return all information.

NUAN_E_NOTFOUND: No CLM info was available for the provided language.

Chapter IV

ve_ttsGetLanguageList

Description

The function **ve_ttsGetLanguageList()** returns the list of the installed languages.

Syntax

NUAN_ERROR
ve_ttsGetLanguageList(
  const VE_HSPEECH *hTtsCl*,
  VE_LANGUAGE * *pLanguages*,
  NUAN_U16 * *pusNbrOfElements*
)

Parameters

hTtsCl	[in] Handle to the TTS class of concern.
pLanguages	[out] A pointer to an array that will be filled with the available languages. It is the application's responsibility to allocate this array.
pusNbrOfElements	[in/out] As input, the variable that this argument points at contains the number of elements in the array *pLanguages*. If *pLanguages* is non-NULL and the specified number is lower than the total number of languages available, the function returns the error code NUAN_E_BUFFERTOOSMALL. On output, the function fills the variable to which *pusNbrOfElements* is pointing with the total number of languages available.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_BUFFERTOOSMALL: There was not enough space in an array argument for the function to return all information. Use the double-call mechanism to first learn the required size, then to retrieve the information.

Comments

If no language is available, this function also returns NUAN_OK, but **pusNbrOfElements* is set to 0.

The application may use double-call mechanism to get the list of installed languages. First call this function with *pLanguages* = NULL to have this function set **pusNbrOfElements* to the number of installed languages, then call this function again with an application allocated output buffer to obtain information on all the languages. For more information on this mechanism, see the **Double-call Functions** subsection within the **Important Remarks** section of this chapter.

Chapter IV

ve_ttsGetLipSyncInfo

Description

The function **ve_ttsGetLipSyncInfo()** retrieves the visible mouth positions for the specified phoneme ID.

The phoneme ID is part of the marker information returned by the callback function VE_CBOUTNOTIFY. See the **Data Types and Structures** section for more information on the VE_LIPSYNC structure, the VE_MARKINFO structure that contains the phoneme ID, and the VE_OUTDATA structure passed to the VE_CBOUTNOTIFY callback that contains the VE_MARKINFO structure.

Syntax

NUAN_ERROR
ve_ttsGetLipSyncInfo(
  VE_HINSTANCE *hTtsInst* ,
  NUAN_U16 *usPhoneme* ,
  VE_LIPSYNC **pTtsLipSync*
)

Parameter Values

hTtsInst	[in] Handle to the TTS instance of concern.
usPhoneme	[in] Phoneme ID of the L&H+ phoneme of concern.
pTtsLipSync	[out] A pointer to the structure that will be filled with the lip synchronization information.

Return values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFRANGE: A phoneme ID is invalid for the current language and voice.

Comments

For more information about the L&H+ phonetic symbol table, please refer to the section **Entering Phonetic Input in Chapter II: <Language> Text-To-Speech System of the User's Guide for <Language>**.

Chapter IV

ve_ttsGetNtsInfo

Description

The function **ve_ttsGetNtsInfo()** returns info on the supported NT-SAMPA mapping for a given language. Vocalizer relies on these data to read phonetic text in NT-SAMPA.

Syntax

NUAN_ERROR
ve_ttsGetNtsInfo(
  const VE_HSPEECH hSpeech,
  const char * szLanguage,
  VE_NTSINFO * pNtsInfo
)

Parameters

hSpeech	[in] Handle to the TTS class of concern.
szLanguage	[in] A zero terminated string indicating the language.
pNtsInfo	[out] Address of a variable of type VE_NTSINFO. On output, the function fills the variable with the available info.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_BUFFERTOOSMALL: There was not enough space in an array argument for the function to return all information.

NUAN_E_NOTFOUND: No NTS info was available for the language.

ve_ttsGetParamList

Description

The function **ve_ttsGetParamList()** returns the value of different control parameters of a TTS instance. This includes the language name, voice name, voice operating point, frequency and many others, see the VE_PARAMID type description for the full list.

Syntax

NUAN_ERROR
ve_ttsGetParamList(
 VE_HINSTANCE *hTtsInst* ,
 VE_PARAM **pTtsParam* ,
 NUAN_U16 *usNbrOfParam*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pTtsParam	[in/out] Points to the buffers into which the requested parameters will be copied. It is the responsibility of the application to allocate memory for this buffer and to set the *eID* member of each element to indicate the parameter to retrieve into that element.
usNbrOfParam	[in] Specifies the number of parameters specified in *pTtsParam*.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_INVALIDARG: A parameter ID is invalid.

Comments

This function retrieves the parameters that are currently set on a specified TTS instance. In order to use this function, the user must first allocate an array of VE_PARAM structures for *pTtsParam*. Then for each member of that array, the user must set the parameter ID in the *eID* member. This function will then fill the *uValue* member of each element based on its parameter ID.

See also ve_ttsSetParamList()

Chapter IV

ve_ttsGetProductVersion

Description

The function **ve_ttsGetProductVersion()** returns the product version number of the TTS engine in a VE_PRODUCT_VERSION structure.

Syntax

NUAN_ERROR
ve_ttsGetProductVersion (
  VE_PRODUCT_VERSION * *pTtsProductVersion*
)

Parameters

pTtsProductVersion	[out] A pointer to the structure that will be filled with the product version information.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

Chapter IV

ve_ttsGetSpeechDBList

Description

The function **ve_ttsGetSpeechDBList()** returns detailed information on all available speech databases for a specific voice and language in an array of VE_SPEECHDBINFO structures.

Syntax

NUAN_ERROR
ve_ttsGetSpeechDBList(
 const VE_HSPEECH *hTtsCl*,
 const char **szLanguage*,
 const char **szVoice*,
 VE_SPEECHDBINFO **pSpeechDBList* ,
 NUAN_U16 * *pusNbrOfElements*
)

Parameters

hTtsCl	[in] Handle to the TTS class of concern.
szLanguage	[in] Language to query for speech databases.
szVoice	[in] Voice name to query for speech databases.
pSpeechDBList	[out] A pointer to an array that will be filled with the available speech databases. It is the application's responsibility to allocate this array.
pusNbrOfElements	[in/out] As input, the variable that this argument points at contains the number of elements in the array *pSpeechDBList*. If *pSpeechDBList* is non-NULL and the specified number is lower than the total number of speech databases available, the function returns the error code NUAN_E_BUFFERTOOSMALL. On output, the function fills the variable to which *pusNbrOfElements* is pointing with the total number of speech databases available.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_BUFFERTOOSMALL: There was not enough space in an array argument for the function to return all information. Use the double-call pattern to first learn the required size, then to retrieve the information.

Comments

The application may use double-call mechanism to get the list of installed speech databases. First call this function with *pSpeechDBList* = NULL to have this function set **pusNbrOfElements* to the number of installed speech databases that match the query, then call this function again with an application allocated output buffer to obtain information on the speech databases. For more information on this mechanism, see the **Double-call Functions** subsection within the **Important Remarks** section of this chapter.

Chapter IV

ve_ttsGetVoiceList

Description

The function **ve_ttsGetVoiceList()** returns information on available voices for a specified language in an array of VE_VOICEINFO structures.

Syntax

NUAN_ERROR
ve_ttsGetVoiceList(
 const VE_HSPEECH *hTtsCl*,
 const char **szLanguage* ,
 VE_VOICEINFO **pVoiceList* ,
 unsigned short * *pusNbrOfElements*
)

Parameters

hTtsCl	[in] Handle to the TTS class of concern.
szLanguage	[in] Language name to query for voices
pVoiceList	[out] A pointer to an array that will be filled with the available voices. It is the application's responsibility to allocate this array.
pusNbrOfElements	[in/out] As input, the variable that this argument points at contains the number of elements in the array *pVoiceList.* If *pVoiceList* is non-NULL and the specified number is lower than the total number of voices available, the function returns the error code NUAN_E_BUFFERTOOSMALL. On output, the function fills the variable to which *pusNbrOfElements* is pointing with the total number of voices available.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_BUFFERTOOSMALL: There was not enough space in an array argument for the function to return all information. Use the double-call pattern to first learn the required size, then to retrieve the information.

Comments

The application may use double-call mechanism to get the list of installed voices. First call this function with *pVoiceList* = NULL to have this function set **pusNbrOfElements* to the number of installed voices that match the query, then call this function again with an application allocated output buffer to obtain information on the voices. For more information on this mechanism, see the **Double-call Functions** subsection within the **Important Remarks** section of this chapter.

Chapter IV

ve_ttsInitialize

Description

The function **ve_ttsInitialize()** creates a TTS class and associates it with a set of resources.

Syntax

NUAN_ERROR
ve_ttsInitialize(
  const VE_INSTALL * *pResources*,
  VE_HSPEECH * *phTtsCl*
)

Parameters

pResources	[in] Description of available resources, including the external service interface pointers.
phTtsCl	[out] Pointer to the handle of the new TTS class.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_NULLPOINTER: A pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_INVALIDPARAM: The resource structure has inappropriate contents.

NUAN_E_VERSION: The version of the resource structure is inappropriate. Make sure to provide external services.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

Comments

The VE_INSTALL structure contains the external services that the client supplies to the TTS class, as well as a list of installed product configurations (available languages, voices and voice operating points). For more details refer to the description of VE_INSTALL in the section **Type definitions**.

Calls to all other API functions must be preceded by a call to **ve_ttsInitialize()**.

As a rule, handles to objects that have been opened after a call to **ve_ttsInitialize()** should be closed before calling **ve_ttsUnInitialize()**.

See also ve_ttsUnInitialize(), VE_INSTALL

Chapter IV

ve_ttsOpen

Description

The function **ve_ttsOpen()** creates a TTS instance based on a specified TTS class. Vocalizer Expressive supports one or more TTS instances per class.

Syntax

NUAN_ERROR
ve_ttsOpen(
 const VE_HSPEECH *hTtsCl*,
 void * *hHeap*,
 void * *hLog*,
 VE_HINSTANCE **phTtsInst*)

Parameters

hTtsCl	[in] Handle to the TTS class of concern.
hHeap	[in] Heap handle to associate with the TTS instance.
hLog	[in] User log handle to associate with the TTS instance.
phTtsInst	[out] A pointer to the handle to the new TTS instance.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

Comments

This function does not load a voice. The application must call **ve_ttsSetParamList()** select a language and/or voice prior to synthesis.

See also ve_ttsClose()

Chapter IV

ve_ttsPause

Description

The function **ve_ttsPause()** passes to the TTS instance a request to pause the playback of the synthesized audio. As it is the callback function that controls the PCM output stream, this function will do little more than send a VE_MSG_PAUSE message to the output callback device. It is the implementation of the output callback that determines if pause/resume functionality is supported or not and implements the pause and resume.

Syntax

NUAN_ERROR
ve_ttsPause(
  VE_HINSTANCE *hTtsInst*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_WRONG_STATE: the TTS instance is not executing a synthesis request.

Notification Messages

VE_MSG_PAUSE	Pause the output stream.

See also ve_ttsResume()

Chapter IV

ve_ttsProcessText2Speech

Description

The function **ve_ttsProcessText2Speech()** reads out an input text. In particular, the TTS instance generates a series of audio and marker buffers until the complete speech signal for the input text is synthesized. Generated audio and marker buffers are transferred to the application by using the callback function.

To have Vocalizer Expressive stop generating audio data quickly, you can have the callback function block further transfer by setting the pointers to the audio buffer to NULL and returning the error code NUAN_E_TTS_USERSTOP. An alternative is to request Vocalizer Expressive to stop as soon as it can, by calling the function **ve_ttsStop()**.

Syntax

NUAN_ERROR
ve_ttsProcessText2Speech(
  VE_HINSTANCE *hTtsInst*,
  const VE_INTEXT * *pInText*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pInText	[in] Structure describing the input text. The expected character encoding is platform-endian UTF-16 (the default) or UTF-8 (optionally configured via the **ve_ttsSetParamList()** API call). It can be either regular text or SMS where phonetic input can be included via an ESC sequence *(<esc>/+)*.

Return Values

The return value NUAN_OK indicates that the function was successful in running the input text through the TTS processing steps. This does not mean however that every character in the input text has been read out, and that each piece of the input text has a counterpart in the delivered audio samples.

The **ve_ttsProcessText2Speech()** function returns NUAN_OK in these particular cases:

- **Vocalizer Expressive has not delivered any speech at all:**

  This happens in case the input text itself is empty, in case the entire input is in a foreign language or in case the input text is encoded in a Windows code page.

- **Vocalizer Expressive has delivered speech for some parts of the input text:**

  This means that Vocalizer Expressive may have dropped for instance a foreign character or word from the input as the current voice doesn't know how to pronounce it.

The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_NOTFOUND: the TTS instance failed to get access to some data. This can be

- A data component that it requested from the Data Access service,
- A prompt that is referenced in an *<ESC>\prompt* in the input text, and that is not available from the any of the loaded ActivePrompt databases.

NUAN_E_COULDNOTOPENFILE: the TTS instance failed to access the audio file referenced in an *<ESC>\audio* in the input text.

NUAN_E_TTS_USERSTOP: the TTS instance aborted the speech synthesis on request of the client, either by a call to **ve_ttsStop()** or on an error of the Output Delivery service.

Notification Messages

VE_MSG_BEGINPROCESS	Begin generating PCM data.
VE_MSG_ENDPROCESS	End of generating PCM data.
VE_MSG_OUTBUFREQ	Request for output buffers message.
VE_MSG_OUTBUFDONE	Ready with a full PCM data buffer.
VE_MSG_STOP	A request to stop synthesis was received. (The stop is not complete until VE_MSG_ENDPROCESS is received.)
VE_MSG_PAUSE	Synthesis was paused.
VE_MSG_RESUME	Synthesis was resumed after being paused.

During synthesis, the message VE_MSG_BEGINPROCESS is delivered first. Then the message VE_MSG_OUTBUFREQ and the message VE_MSG_OUTBUFDONE are delivered until there is no more speech output. At the end of processing, the TTS system sends the message VE_MSG_ENDPROCESS to the application.

The application can receive other messages (VE_MSG_STOP, VE_MSG_PAUSE or VE_MSG_RESUME) between VE_MSG_BEGINPROCESS and VE_MSG_ENDPROCESS if the application calls one of the functions **ve_ttsStop()**, **ve_ttsPause()** or **ve_ttsResume()** respectively.

Comments

The structure VE_OUTDATA is used to transfer the generated PCM data and markers to the application. The message VE_MSG_OUTBUFREQ is to request that the application allocates memory for the output data and VE_MSG_OUTBUFDONE to send the output data to the application.

For a description of the callback messages, see the **Notification messages** subsection in the **Data types, structures and type definitions** section.

The structure VE_OUTDATA is defined as:

```
typedef struct {
  VE_AUDIOFORMAT eAudioFormat;
  size_t         cntPcmBufLen;
  void         * pOutPcmBuf;
  size_t         cntMrkListLen;
  VE_MARKINFO *pMrkList;
} VE_OUTDATA;
```

When the application gets the message VE_MSG_OUTBUFREQ, it has to allocate memory for the output data buffer *pOutPcmBuf* and fill *ulPcmBufLen* with the size (in bytes) of this buffer. Also, the application has to allocate memory for the marker array *pMrkList* and fill in *ulMrkListLen* the size (in bytes) of the allocated buffer.

When the application gets the message VE_MSG_OUTBUFDONE, *ulPcmBufLen* contains the size (in bytes) of the data copied in the output PCM buffer. *ulMrkListLen* contains the size (number of markers) copied in the marker array.

The marker array *pMrkList* contains the information on the phoneme IDs so the lip synchronization information can optionally be looked up with **ve_ttsGetLipSyncInfo()**.

Vocalizer Expressive keeps the contents of the audio and the marker buffers synchronized for the callback delivery. Specifically, it ensures that the markers for position X and the audio for position X are always delivered in a single call, delivering only partially filled marker or sample buffers if one fills up before the other.

In the special case where there are more markers for position X than can fit in the marker buffer, Vocalizer Expressive follows the principle of always delivering markers at or before the matching audio. First, it delivers all the audio and markers prior to position X.

Then if the count of markers for position X is greater than the marker buffer, it'll do calls with an empty sample buffer to deliver the markers in excess of what fits in the marker buffer, repeating that until the number of remaining markers for position X fit in the marker buffer. Then it'll deliver the audio for position X and remaining markers for position X in a single call.

See also ve_ttsStop()

Chapter IV

ve_ttsProcessText2SpeechStartingAt

Description

The function **ve_ttsProcessText2SpeechStartingAt()** synthesizes an input text like **ve_ttsProcessText2Speech()** does, but it starts at a given location in the input text. The locations in the input text that the client may navigate to, are described by the TA info generated by the function **ve_ttsAnalyzeText()**. The text passed to this function is also generated by the function **ve_ttsAnalyzeText()**: it is the input text rewritten by user rulesets and encoded in UTF-8. The function **ve_ttsProcessText2SpeechStartingAt()** makes sure not to apply the user ruleset again to this text.

Syntax

NUAN_ERROR
ve_ttsProcessText2SpeechStartingAt(
  VE_HINSTANCE *hTtsInst*,
  const VE_INTEXT * *pInText,*
  const VE_TA_NODE * *pTaInfo*,
  const size_t cnt*TaIndex*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pInText	[in] Structure describing the input text rewritten by loaded user ruleset and generated by **ve_ttsAnalyzeText()**.
pTaInfo	[in] List of jump points generated by **ve_ttsAnalyzeText().**
cnt*TaIndex*	[in] Index of jump point that defines the location to start reading from.

Return Values

The return value NUAN_OK indicates that the function was successful in running the input text through the TTS processing steps.

The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_NOTFOUND: the TTS instance failed to get access to some data. This can be

- A data component that it requested from the Data Access service,
- A prompt that is referenced in an *<ESC>\prompt* in the input text, and that is not available from the any of the loaded ActivePrompt databases.

NUAN_E_COULDNOTOPENFILE: the TTS instance failed to access the audio file referenced in an *<ESC>\audio* in the input text.

NUAN_E_TTS_USERSTOP: the TTS instance aborted the speech synthesis on request of the client, either by a call to **ve_ttsStop()** or on an error of the Output Delivery service.

Notification Messages

Same as **ve_ttsProcessText2Speech()**.

Comments

This function is used in combination with **ve_ttsAnalyzeText()** to traverse through the input text. It takes as arguments the input text rewritten by loaded user rulesets, and the TA info, both of which were generated before by **ve_ttsAnalyzeText()**.

It makes sure to not to exercise the loaded user rulesets again on the rewritten input text.

This function delivers markers that are positioned in the rewritten input text with respect to the jump point that marks the start location. In particular the field **ulSrcPos** of a marker defines its offset in the rewritten input text with respect to the jump point pTaInfo[uTaIndex] (which is considered at offset 0). You turn that into an offset with respect to the beginning of the rewritten input text by adding the **positionInText** field of the jump point.

See also ve_ttsAnalyzeText(), ve_ttsProcessText2Speech()

Chapter IV

ve_ttsResourceLoad

Description

The function **ve_ttsResourceLoad()** loads tuning data for use during synthesis, including user dictionaries, user rulesets, and ActivePrompt databases.

Syntax
NUAN_ERROR
ve_ttsResourceLoad(
 VE_HINSTANCE *hTtsInst*,
 const char * *szMimeContentType*,
 size_t cnt*InDataLength*,
 const void * *pInData*,
 VE_HRESOURCE * *phResource*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
szMimeContentType	[in] MIME content type of the data to load (details under Comments)
cntInDataLength	[in] Length of the data to load in bytes.
pInData	[in] Data to load. To conserve memory Vocalizer Expressive does not deep copy this data, so it must remain valid until the data is unloaded or the instance is closed. The data buffer needs to be aligned on a 4-byte boundary.
phResource	[out] A pointer to the handle to the newly loaded TTS tuning resource.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_NULLPOINTER: a pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_INVALIDARG: a size argument is 0.

NUAN_E_WRONG_STATE: the TTS instance is still busy executing another API call. Wait until that is done.

NUAN_E_NOTFOUND: the TTS instance has no pipeline component that can work with the type of tuning data. Check that the MIME type is a string listed below.

Comments

If the loaded resource does not match the current synthesis language and voice, the load will still succeed, but Vocalizer Expressive will not apply the data during synthesis.

Vocalizer Expressive supports loading multiple user rulesets, multiple user dictionaries, and multiple ActivePrompt databases.

For each category of tuning data, more recently loaded data has precedence over previously loaded data. But all user rulesets have precedence over all user dictionaries, and all user dictionaries have precedence over all ActivePrompt databases.

Vocalizer Expressive accepts the following values for *szMimeContentType*:

- *application/edct-bin-dictionary* for a User Dictionary in binary format.

  Append the parameter/value pair *";mode= langoverwriting"* if you want the user dictionary to override the language of the text for the words covered by its entries. This is explained in the section on multi-lingual voices.

- *application/x-vocalizer-rettt+text* for a User Ruleset, which is a text file encoded in UTF-8.

- *application/x-vocalizer-pt+bin* and *application/x-vocalizer-pt+text* for the binary resp. textual version of a prompt template set.

- *application/x-vocalizer-activeprompt-db* for an ActivePrompt database.

  By default Vocalizer Expressive only enables implicit matching as it loads an ActivePrompt database if the ActivePrompt database was marked to run in automatic mode at build time. Otherwise it only activates it when it finds the appropriate *<ESC>\domain=domain_name* in the input text. To enable implicit matching at load time append the following parameter/value pair to the MIME content type:

  - "*;mode=automatic*"

  For finding recorded audio referenced by ActivePrompt databases, by default Vocalizer Expressive tries to access a recorded prompt speechbase named "apdb_cs/voice_name/domain_name/freq_tag" e.g. "apdb_cs/xander/expressive/f22".

  To have Vocalizer Expressive access the recordings as individual files instead, append these additional parameter/value pairs to the MIME content type:

  - "*;uriprefix=<path>*" to specify a prefix to use when constructing the pathname of a prompt recording,
  - "*;urisuffix=<path>*" to specify a suffix to use when constructing the pathname of a prompt recording,

  For example:

application/x-vocalizer-activeprompt-db;uriprefix=/audio/;urisuffix=.wav.

The final recording pathname will be *<uriprefix><ActivePrompt ID><urisuffix>*, with *<uriprefix>* defaulting to an empty string, and *<urisuffix>* using a default value as selected when building the ActivePrompt database (typically ".wav").

See also ve_ttsResourceUnload()

ve_ttsResourceUnload

Description

The function **ve_ttsResourceUnload()** unloads tuning resources that were previously loaded with **ve_ttsResourceLoad()**, such as user dictionaries, user rulesets, and ActivePrompt databases.

Syntax
NUAN_ERROR
ve_ttsResourceLoad(
 VE_HINSTANCE *hTtsInst*,
 VE_HRESOURCE hResource
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
hResource	[in] Handle to the loaded TTS tuning resource to unload.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below

NUAN_E_INVALIDHANDLE: A handle is invalid:

- The TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.
- The tuning resource handle is not valid. Make sure that it is a handle created by **ve_ttsResourceLoad()**.

NUAN_E_WRONG_STATE: the TTS instance is still busy executing another API call. Wait until that is done.

Comments

See also ve_ttsResourceLoad()

Chapter IV

ve_ttsResume

Description

The function **ve_ttsResume()** passes to the TTS instance a request to resume playback of the synthesized audio. As it is the callback function that controls the PCM output stream, this function will do little more than send a VE_MSG_RESUME message to the output callback device. It is the implementation of the output callback that determines if pause/resume functionality is supported or not and implements the pause and resume.

Syntax

NUAN_ERROR
ve_ttsResume(
VE_HINSTANCE *hTtsInst*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_WRONG_STATE: the TTS instance is not executing a synthesis request.

Notification Messages

VE_MSG_RESUME	Resume the output stream.

See also ve_ttsPause()

Chapter IV

ve_ttsSetOutDevice

Description

The function **ve_ttsSetOutDevice()** associates an output device handle with the TTS instance. The TTS system will call the output callback function to transfer the output audio data and optional markers.

Syntax

NUAN_ERROR
ve_ttsSetOutDevice(
 VE_HINSTANCE *hTtsInst* ,
 VE_OUTDEVINFO **pOutDevInfo*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pOutDevInfo	[in] Output device information structure

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_NULLPOINTER: a pointer argument is NULL. Provide a valid location where the function can find information.

NUAN_E_INVALIDPARAM: The output device structure has inappropriate contents.

NUAN_E_WRONG_STATE: the TTS instance is still busy executing another API call. Wait until that is done.

Chapter IV

ve_ttsSetParamList

Description

The function **ve_ttsSetParamList()** sets the value of control parameters such as the language, voice name, volume level, speech rate, etc. See the description of the VE_PARAMID enumeration for details.

You call this function first to configure a new TTS instance with a voice. Only then it accepts changes for other control parameters such as speech rate and text mode. Later you can call this function again to switch the TTS instance to another voice.

Syntax

NUAN_ERROR
ve_ttsSetParamList(
 VE_HINSTANCE *hTtsInst*,
 VE_PARAM **pTtsParam* ,
 NUAN_U16 *usNbrOfParam*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pTtsParam	[in] Points to an array of parameters the application wants to set.
usNbrOfParam	[in] Number of parameters in *pTtsParam*.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_NULLPOINTER: a pointer argument is NULL. Provide a valid location where the function can return information.

NUAN_E_WRONG_STATE: the TTS instance is not capable to accept a change of one or more control parameters. This may happen in case that

- The TTS instance is busy reading out text, i.e. executing the function **ve_ttsProcessText2Speech()** in another thread. In this case you should wait until the TTS instance is done with the speak request.
- The TTS instance has just been created, and has not yet constructed the processing pipeline. You should first set the language and/or the voice.

- The TTS instance has no instantiated processing pipeline components as it is configured for init mode `VE_INITMODE_LOAD_OPEN_ALL_EACH_TIME.` . You should switch to the default init mode `VE_INITMODE_LOAD_ONCE_OPEN_ALL` to let the TTS instance accept parameter changes.

NUAN_E_NOTIMPLEMENTED: the current configuration of the TTS instance does not support a particular control parameter.

NUAN_E_MODULENOTFOUND, NUAN_E_COULDNOTOPENFILE or NUAN_E_NOK: the TTS instance failed to get access to a data component that it requested from the Data Access service.

NUAN_E_FILEREADERROR: the TTS instance found unexpected contents in a data component or in a pipeline .hdr file.

NUAN_E_OUTOFMEMORY: The function failed to acquire a block of memory from the Heap service.

NUAN_E_INVALIDARG: A parameter ID or value is invalid. This may happen when you pass a wide-char string argument, or the sizeof(int) is different.

Comments

This function sets all the specified parameters. The parameter IDs and values in the array of VE_PARAM structures must be filled in with the IDs and values of the parameters you want to set.

You call this function first to configure a new TTS instance (created by **ve_ttsOpen**) with a voice. You can do that with a call passing a fully-specified voice, i.e. in the parameter argument list you provide a value for the language, the voice and the voice operating point, e.g.:

- VE_PARAM_LANGUAGE : "Mexican Spanish"
- VE_PARAM_VOICE : "Paulina"
- VE_PARAM_VOICE_OPERATING_POINT : "premium-high"

Vocalizer also accepts a partially-specified voice. In this case you give only the language, or only the voice name, or only the language and voice. Vocalizer will select the best value for the unspecified parameters, and configure the TTS instance with a voice that gives the best speech output:

- If you only specify the language, Vocalizer selects the preferred voice in the language and the higher-quality voice operating point.
- If you only specify the voice, Vocalizer looks for it over all available languages, and selects the higher-quality voice operating point.

- If you specify the language and the voice, Vocalizer selects the higher-quality voice operating point.

On return of the function the TTS instance will have a value for each of the parameters language, voice and operating point. You can retrieve their value through **ve_ttsGetParamList()**.

Configuring the TTS instance with a voice allows it instantiating the processing pipeline components, and loading the language and voice data components. With the processing pipeline components in place the TTS instance can validate requested changes for other control parameters such as speech rate and text mode, and put them into effect if the voice is capable.

You can also call this function later to switch the TTS instance to a different voice. Similarly to the initial configuration you can pass a fully-specified voice in the argument list. In contrast, you can't enter a partially-specified voice simply by leaving out a parameter like language from the argument list: this will keep the current value of the language, and this is likely not to combine well with the new value of supplied parameters. For instance, if your TTS instance is configured for

- VE_PARAM_LANGUAGE : "American English"
- VE_PARAM_VOICE : "Ava"
- VE_PARAM_VOICE_OPERATING_POINT : "premium-high"

and you only pass parameter VE_PARAM_VOICE : "Paulina", Vocalizer will look for a voice that matches

- VE_PARAM_LANGUAGE : "American English"
- VE_PARAM_VOICE : "Paulina"
- VE_PARAM_VOICE_OPERATING_POINT :
- "premium-high"

and find none.

Instead you exploit the support for a partially-specified voice by supplying an empty string value for the parameters that you don't specify. In the previous example you switch from Ava to Paulina providing

- VE_PARAM_LANGUAGE : ""
- VE_PARAM_VOICE : "Paulina"
- VE_PARAM_VOICE_OPERATING_POINT : ""

With a call to this function you may want to change the parameter VE_PARAM_INITMODE to instruct the TTS instance about the time to create/remove instances of the pipeline components and load/unload data components. The TTS instance can defer this moment until it actually needs the data components during the call to **ve_ttsProcessText2Speech()**. For more details about the parameter VE_PARAM_INITMODE refer to the VE_PARAMID type description.

See also ve_ttsGetParamList()

Chapter IV

ve_ttsStop

Description

The function **ve_ttsStop()** aborts the current speak request.

Syntax

NUAN_ERROR
ve_ttsStop(
 VE_HINSTANCE *hTtsInst*
)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.

Return values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS instance handle is not valid. Make sure that it is a handle created by **ve_ttsOpen()**.

NUAN_E_WRONG_STATE: the TTS instance is not executing a synthesis request.

Notification Messages

VE_MSG_STOP	Stop speaking a text or generating PCM data.

Comments

When the function **ve_ttsProcessText2Speech()** is called, the TTS instance sends the message VE_MSG_BEGINPROCESS to the application. From this moment on the function **ve_ttsStop()** can be called.

The **ve_ttsStop()** function is asynchronous. It can be called either from a separate thread or from within the callback function. The stop is completed only when the stopped **ve_ttsProcessText2Speech()** function sends the VE_MSG_ENDPROCESS to the application and returns.

See also ve_ttsProcessText2Speech()

Chapter IV

ve_ttsUnInitialize

Description

The function **ve_ttsUnInitialize()** removes a TTS class and frees all allocated resources.

Syntax

NUAN_ERROR
ve_ttsUnInitialize(
 const VE_HSPEECH *hTtsCl*
)

Parameters

hTtsCl	[in] Handle to the TTS class of concern.

Return Value

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code. The typical error codes are described below.

NUAN_E_INVALIDHANDLE: the TTS class handle is not valid. Make sure that it is a handle created by **ve_ttsInitialize()**.

NUAN_E_WRONG_STATE: there are still TTS instances created from the TTS class. Close those TTS instances first with **ve_ttsClose()**.

Comments

In an application, each call to **ve_ttsInitialize()** should be balanced with a call to **ve_ttsUnInitialize()**. If not, there will be a memory leak.

Before you call **ve_ttsUnInitialize()**, you should make sure the handle to all open TTS instances are closed. If you fail to do so, this function will return an error.

If this function fails, the application behaves as if no call to **ve_ttsUnInitialize()** has taken place.

See also ve_ttsInitialize()

Heap service

The following functions define a memory allocation interface. The function prototypes are modeled after the C standard memory allocation functions.

pfCalloc

Description

The function ***pfCalloc()** is called when a TTS class or an instance needs a new memory block which is initialized with zeroes.

Syntax

void *
(*pfCalloc)(
 void * *hHeap,*
 size_t *cElements,*
 size_t *cElementBytes*
)

Parameters

hHeap	[in] Heap handle of concern.
cElements	[in] Number of elements to allocate.
cElementBytes	[in] Size (in bytes) of an element.

Return Values

If the function completes successfully, it returns the address at which the memory is allocated. If no memory is available it returns NULL.

Comments

This function must return an address which starts a memory block of at least *cElements*cElementBytes* bytes. The memory must be aligned such that any C-type can be stored at the returned address. The returned memory must be filled with zeroes. *hHeap* identifies the heap that is used.

The following semantics apply (size = *cElements*cElementBytes*):

size	What happens	Return value
> 0	Allocation succeeds	Valid address
> 0	Allocation fails	NULL
0	No operation	NULL

Chapter IV

pfFree

Description

The function ***pfFree()** is called when a TTS class or an instance wants to free a memory block.

Syntax

void
(*pfFree)(
  void * *hHeap*,
  void * *pData*
)

Parameters

hHeap	[in] Heap handle of concern.
pData	[in] Start of the memory block to free.

Return Values

Void.

Comments

hHeap identifies the heap that is used.

The following semantics apply:

pData	What happens
Valid address	pData freed
NULL	No operation

Chapter IV

pfMalloc

Description

The function ***pfMalloc()** is called when a TTS class or an instance needs a new memory block.

Syntax

void *
(* pfMalloc)(
  void * *hHeap,*
  size_t *cBytes*
)

Parameters

hHeap	[in] Heap handle of concern
cBytes	[in] Number of bytes to allocate

Return Values

If the function completes successfully, it returns the address at which the memory is allocated. If no memory is available it returns NULL..

Comments

This function should return an address which starts a memory block of at least *cBytes* bytes. The memory should be aligned such that any C-type can be stored at the returned address. *hHeap* identifies the heap that is used.

The following semantics apply:

cBytes	**What happens**	**Return value**
> 0	Allocation succeeds	Valid address
> 0	Allocation fails	NULL
0	No operation	NULL

pfRealloc

Description

The function ***pfRealloc()** is called when a TTS class or an instance wants to grow or shrink a memory block.

Syntax

void *
pfRealloc(
 void * *hHeap,*
 void * *pData,*
 size_t *cBytes*
)

Parameters

hHeap	[in] Heap handle of concern.
pData	[in] Start of the memory block to grow or to shrink.
cBytes	[in] Number of bytes to allocate

Return Values

If the function completes successfully, it returns the address at which the memory is allocated. If no memory is available it returns NULL..

Comments

This function should return an address which starts a memory block of at least *cBytes* bytes. The memory should be aligned such that any C-type can be stored at the returned address. Data starting at the address *pData* should be copied into the new memory block. The data may be truncated if *cBytes* is less than the length of the memory block starting at address *pData*. *hHeap* identifies the heap that is used.

The following semantics apply:

pData	cBytes	What happens	Return value	Side effect
valid address	> 0	Allocation succeeds	valid address	pData freed
valid address	> 0	Allocation fails	NULL	pData not freed
valid address	0	No operation	NULL	pData not freed
NULL	> 0	Allocation succeeds	valid address	None
NULL	> 0	Allocation fails	NULL	None
NULL	0	No operation	NULL	None

Chapter IV

Critical Sections service

The following functions define a critical sections (mutex) interface. The function prototypes are similar to many OS specific critical section libraries.

pfClose

Description

The function ***pfClose()*** is called when a TTS class or an instance wants to release a critical section created with **pfOpen()**. The TTS class or instance no longer needs the critical section to control thread-safe execution of code.

Syntax

NUAN_ERROR
(*pfClose)(
  void * *hCritSec*
)

Parameters

hCritSec	[in] Handle of the critical section of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

pfEnter

Description

The function ***pfEnter()** is called when a TTS class or an instance needs the exclusive right to execute a piece of code, and thus wants to wait for a critical section to grant the ownership.

Syntax

NUAN_ERROR
(*pfEnter)(
  void * *hCritSec*
)

Parameters

hCritSec	[in] Handle of the critical section of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

This function must support recursive calls. This means that a critical section may get locked by a thread, and then on the same call stack locked again by the same thread. In other words, a successful call on a critical section must not block a second call in the same thread.

Chapter IV

pfLeave

Description

The function ***pfLeave()** is called when a TTS class or an instance wants to release ownership of a critical section previously acquired with **pfEnter()**.

Syntax

NUAN_ERROR
(*pfLeave)(
  void * *hCritSec*
)

Parameters

hCritSec	[in] Handle of the critical section of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

pfOpen

Description

The function ***pfOpen()** is called when a TTS class or an instance needs a critical section to control thread-safe execution of code.

Syntax

NUAN_ERROR
(*pfOpen)(
 void * *hCCritSec,*
 void * *hHeap*,
 void ** *phCritSec*
)

Parameters

hCCritSec	[in] Class handle of the Critical Sections service
hHeap	[in] Heap handle to associate with the critical section
phCritSec	[out] Location for the created critical section

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

Data Streams service

The following functions define an I/O stream interface. The function prototypes are modeled after the C standard file I/O functions. Error handling is taken into account in that these functions may fail and set an error indicator on the stream.

pfClose

Description

The function ***pfClose()** is called when a TTS class or an instance is done reading from or writing to a data stream.

Syntax

NUAN_ERROR
(*pfClose)(
  void * *hStream*
)

Parameters

hStream	[in] Handle of the data stream of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

pfError

Description

The function ***pfError()** is called when a TTS class or an instance wants to check the error indicator of the data stream.

Syntax

NUAN_ERROR
(*pfError)(
 void * *hStream*
)

Parameters

hStream	[in] Handle of the data stream of concern.

Return Values

If the data stream is in an error state, it returns an error code, otherwise the function returns the code NUAN_OK.

pfGetSize

Description

The function ***pfGetSize()** is called when a TTS class or an instance wants to learn the size of the data available from a data stream.

Syntax

size_t
(*pfGetSize)(
 void * *hStream*
)

Parameters

hStream	[in] Handle of the data stream of concern.

Return Values

The total size (in bytes) of the data available from a data stream.

Chapter IV

pfOpen

Description

The function ***pfOpen()** is called when a TTS class or an instance wants to create a data stream to read from or write to.

Syntax

NUAN_ERROR
(*pfOpen)(
 void * *hCData,*
 void * *hHeap*,
 const char * *szDataId*,
 const char * *szMode*,
 void ** *phStream*
)

Parameters

hCData	[in] Class handle of the Data Streams service
hHeap	[in] Heap handle to associate with the data stream
szDataId	[in] Unique name of data
szMode	[in] Code that defines the read/write mode
phStream	[out] Location for the created data stream

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

The data stream is identified by a logical name *szDataId*, not a file path, as Vocalizer Expressive does not make assumptions as to where the data physically resides.

A TTS class or instance may call this function several times on the same data, thus with identical *szDataId* values.

This function must always support *szMode* "rb" (binary read). This function should support mode "w" (text write) to support Vocalizer Expressive product builds with extra logging support.

Chapter IV

pfRead

Description

The function ***pfRead()** is called when a TTS class or an instance wants a block of data copied from a data stream into a memory buffer.

Syntax

size_t
(*pfRead)(
  void * *pBuffer*,
  size_t *cElementBytes*,
  size_t *cElements*,
  void * *hStream*
)

Parameters

pBuffer	[in] Start of the memory buffer
cElementBytes	[in] Size (in bytes) of an element
cElements	[in] Number of elements to read
hStream	[in] Handle of the data stream of concern

Return Values

The number of elements actually copied into *pBuffer*. If this number is less than *cElements*, then Vocalizer Expressive assumes the end of the data stream is reached.

Comments

This function should copy *cElement* elements of size *cElementBytes* (in bytes) from the data stream *hStream* into *pBuffer*. The function expects that *pBuffer* is sufficiently big.

pfSeek

Description

The function ***pfSeek()** is called when a TTS class or an instance wants to change the position for the next I/O operation on a data stream.

Syntax

NUAN_ERROR
(*pfSeek)(
 void * *hStream,*
 size_t c*Offset*,
 VE_STREAM_ORIGIN *eOrigin*,
 VE_STREAM_DIRECTION *eDirection*
)

Parameters

hStream	[in] Handle of the data stream of concern.
cOffset	[in] Number of bytes to jump
eOrigin	[in] Indicates the origin to jump from
eDirection	[in] Indicates the direction to jump

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

The prototype of this function deviates from the ANSI C specification, which only uses an origin and an offset (type int). However, that implicitly limits the file size that can be handled to 2GB (INT_MAX), because most C library implementations just convert the origin plus offset to an absolute value (of type "int"), then change the position to that absolute value.

This prototype allows to work with files up to 4GB, which is possible for Vocalizer Expressive speechbases, and can be implemented using functions like Microsoft Visual Studio fseeki64() or Linux fseeko().

Chapter IV

pfWrite

Description

The function ***pfWrite()** is called when a TTS class or an instance wants a block of data copied from a memory buffer to a data stream. It is only called in Vocalizer Expressive builds that include extra logging support.

Syntax

size_t
(*pfWrite)(
  const void * *pBuffer*,
  size_t *cElementBytes*,
  size_t *cElements*,
  void * *hStream*
)

Parameters

pBuffer	[in] Start of the memory buffer to write
cElementBytes	[in] Size (in bytes) of an element
cElements	[in] Number of elements to write
hStream	[in] Handle of the data stream of concern.

Return Values

The number of elements actually copied into *hStream*. If this number is less than *cElements*, then Vocalizer Expressive assumes the file write failed.

Comments

This function is to copy *cElement* elements of size *cElementBytes* (in bytes) from *pBuffer* to the data stream *hStream*.

Chapter IV

Data Mappings service

The following functions define a data mapping interface for read-only access to data. This data access model is stricter than the data stream model as the caller does not own the data, hence must not touch them.

This interface allows the client to optimize resource usage and performance for platforms like WinCE that have native OS support for memory mapped file access, or for applications and platforms where data should reside in ROM (minimize RAM use at the cost of performance) or be loaded into RAM in their entirety at startup (maximize performance).

pfClose

Description

The function ***pfClose()** is called when a TTS class or an instance is done reading from a data mapping.

Syntax

NUAN_ERROR
(*pfClose)(
  void * *hMapping*
)

Parameters

hMapping	[in] Handle of the data mapping of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

pfFreeze

Description

The function ***pfFreeze()** is called when a TTS class or an instance wants to freeze the current mapped data block, and will not remap it on the data mapping. This means that it will not call **pfMap()** again, but simply read from the current mapped data block, then **pfUnmap()** it.

Syntax

NUAN_ERROR
(*pfFreeze)(
 void * *hMapping*
)

Parameters

hMapping	[in] Handle of data mapping of concern.

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

This function allows data mapping implementations to optimize for common cases where Vocalizer Expressive maps a data chunk during **ve_ttsOpen()** with the intent of never remapping it. For example, a data mapping implementation based on ANSI C file I/O may use this to close file handles that won't be required anymore.

Chapter IV

pfMap

Description

The function ***pfMap()** is called when a TTS class or an instance wants a data mapping to provide a read-only window on a data block.

Syntax

NUAN_ERROR
(*pfMap)(
  void * *hMapping*,
  size_t c*Offset*,
  size_t * *pcBytes*,
  const void ** *ppData*
)

Parameters

hMapping	[in] Handle of the data mapping of concern.
cOffset	[in] Start of the mapped data block as an offset (in bytes) to the very beginning of the data
pcBytes	[in out] Size (in bytes) of the mapped data block. If this is 0, this requests mapping the entire file. This must be updated to indicate the actually mapped size on output.
ppData	[out] Location for the pointer to mapped data block

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

If *cOffset* is 2-byte or 4-byte aligned relative to the start of the data mapping (*cOffset % 2 == 0* or *cOffset % 4 == 0)*, the returned memory should maintain that alignment such that a 2 byte C-type (when *cOffset* is 2-byte aligned*)* or 4 byte C-type (when *cOffset* is 4-byte aligned) can be accessed at the returned address.

If pcBytes is 0, this requests mapping all the data. If *pcBytes* is larger than the available amount of data, but the available amount of data is still larger then 0, that amount of data should be mapped with *pcBytes* updated to indicate the amount of data actually mapped.

Chapter IV

pfOpen

Description

The function ***pfOpen()** is called when a TTS class or an instance wants to create a data mapping to read data from.

Syntax

NUAN_ERROR
(*pfOpen)(
  void * *hCData,*
  void * *hHeap*,
  const char * *szDataId*,
  void ** *phMapping*
)

Parameters

hCData	[in] Class handle of the data access service
hHeap	[in] Heap handle to associate with the created data mapping
szDataId	[in] Unique name of the data to map
phMapping	[out] Location for the created data mapping

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Comments

The data stream is identified by a logical name *szDataId*, not a file path, as Vocalizer Expressive does not make assumptions as to where the data physically resides.

A TTS class or instance may call this function several times on the same data, thus with an identical *szDataId* value.

pfUnmap

Description

The function ***pfUnmap()** is called when a TTS class or an instance is done reading the currently mapped data block acquired with **pfMap()**.

Syntax

NUAN_ERROR
(*pfUnmap)(
  void * *hMapping*,
  const void * *pData*
)

Parameters

hMapping	[in] Handle of the data mapping of concern.
pData	[in] Start of the mapped data block to unmap

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

Chapter IV

User Log service

The following functions define a logging interface for reporting diagnostic and error messages.

pfDiagnostic

Description

The function ***pfDiagnostic()** is called when a TTS class or an instance wants to report a diagnostic message. This function is only called when diagnostic logging is configured (by default it is disabled).

Syntax

void
(*pfDiagnostic)(
  void * *hLog,*
  NUAN_S32 *s32Level*,
  const char * *szMessage*
)

Parameters

hLog	[in] Log handle of concern
s32Level	[in] Log level
szMessage	[in] Diagnostic log message

Return Values

None.

Chapter IV

pfError

Description

The function ***pfError()** is called when a TTS class or an instance wants to report an error by ID.

Syntax

void
(*pfError)(
void * *hLog,*
NUAN_U32 *u32ErrorId,*
size_t *cKeyValues,*
const char ** *aszKeys,*
const char ** *asValues*
)

Parameters

hLog	[in] Log handle of concern.
u32ErrorId	[in] Error number
cKeyValues	[in] Number of key-value pairs
aszKeys	[in] List of keys
asValues	[in] List of values

Return Values

None.

Comments

The argument *u32ErrorID* identifies an entry in VocalizerLogStrings.enu.xml, and the key/value pairs contain supplemental information to clarify the error (such as the name of a data object that couldn't be opened, etc.).

Chapter IV

Output Delivery service

The following functions define an output delivery interface for delivering the output audio and markers for TTS operations.

VE_CBOUTNOTIFY

Description

This is the prototype of the call-back function that the TTS instance uses for sending notification messages to its output device. The body of this function is located in the application. The TTS instance calls this function when it requires an output PCM buffer , and when it has a PCM buffer to transfer. Please see the messages VE_MSG_OUTBUFREQ and VE_MSG_OUTBUFDONE, and the structure VE_OUTDATA for more information on handling the speech output generation. For an example see the sample program.

Syntax

NUAN_ERROR
VE_CBOUTNOTIFY(
VE_HINSTANCE *hTtsInst*,
void * *pUserData*,
VE_CALLBACKMSG **pcbMessage*)

Parameters

hTtsInst	[in] Handle to the TTS instance of concern.
pUserData	[in] Handle to an output device instance. This output device handle can be used by the application to handle a user specific device as output. It can also be used to deliver user data with the output callback.
pcbMessage	[in] Message structure sent to the output device connected to the calling TTS instance.

Return Values

The function should return one of the following return codes:

NUAN_OK	On successful completion
NUAN_E_TTS_USERSTOP	If the current synthesis should be stopped
NUAN_E_SYSTEM_ERROR, or <any other value>	If an error occurred; this causes TTS to stop processing

Comments

The output device should check the *pcbMessage* structure to know the meaning of the notified message.

Chapter IV

Data types, structures and type definitions

Below are data structures and data types that are specific for Vocalizer Expressive. As this is a Unicode environment, the following basic data types are defined.

```
typedef NUAN_U16             VE_VERSION;
typedef unsigned char        NUAN_U8;
typedef signed   char        NUAN_S8;
typedef unsigned short       NUAN_U16;
typedef signed   short       NUAN_S16;
typedef unsigned long        NUAN_U32;
typedef signed   long        NUAN_S32;
```

The structures returning information about available languages, voices and speechbases contain an extra member identifying the language ID. This language ID can be used with the VE_PARAM_LANGUAGE_NR parameter or as alternative for the language string in the **ve_ttsGetXxxList( )** query functions.

Chapter IV

Type definitions

VE_ADDITIONAL_PRODUCTINFO

```
typedef struct {
    NUAN_U16          buildYear
    NUAN_U8           buildMonth
    NUAN_U8           buildDay
    char              buildInfoStr[256]
}   VE_ADDITIONAL_PRODUCTINFO;
```

Description

The structure VE_ADDITIONAL_PRODUCTINFO defines the build date and a possible custom identifier of the build. This information is returned by **ve_ttsGetAdditionalProductInfo()**.

Members

buildYear	Year of the build
buildMonth	Month of the build
buildDay	Day of the build
buildInfoStr	String identifier of the build.

See also ve_ttsGetAdditionalProductInfo()

VE_AUDIOFORMAT

```
typedef enum {
     VE_16LINEAR,
     VE_MU_LAW,
     VE_A_LAW
}         VE_AUDIOFORMAT;
```

Description

Audio output formats, but only VE_16LINEAR is supported by Vocalizer Expressive.

Members

VE_16LINEAR	linear PCM, signed 16-bit per sample, platform-endian, mono
VE_MU_LAW	μ-law PCM format: not currently supported

VE_A_LAW	A-law PCM format: not currently supported

See also ve_ttsProcessText2Speech()

VE_CALLBACKMSG

```
typedef struct {
    VE_MSG                         eMessage;
    NUAN_S32                       lValue;
    void                           *pParam;
}   VE_CALLBACKMSG;
```

Description

The structure VE_CALLBACKMSG is returned in all notification messages on the Output Delivery service. It contains the complete description of the transferred message.

For a description of the different callback messages, see the **Notification messages** section.

Members

eMessage	The meaning of the sent message
lValue	Message data argument, dependent on the notified message
pParam	Message data pointer, dependent on the notified message. On a message of type VE_MSG_OUTBUFDONE it points to a VE_OUTDATA structure with a block of audio and markers. On a message of type VE_MSG_TAIBUFDONE it points to a VE_OUTTAINFO struct with a block of text analysis info and rewritten text.

VE_CLMINFO

```
typedef struct {
    char   szFileVersion[VE_MAX_STRING_LENGTH];
    char   szSrcVersion[VE_MAX_VERSIONSTRING_LENGTH];
    char   szDstVersion[VE_MAX_VERSIONSTRING_LENGTH];
}        VE_CLMINFO;
```

Description

The structure VE_CLMINFO is used to transfer CLM information to the user.

Members

szFileVersion	Version info of the CLM data file.
szSrcVersion	Version info of the source language.

szDstVersion	Version info of the target language.

VE_CRITSEC_INTERFACE

Description

Set of functions that define the interface of the Critical Sections service. The prototypes of these functions are specified in the section **Critical Sections service** of the **Function directory**.

Members

pfOpen	[in] Pointer to a function to create a critical section
pfClose	[in] Pointer to a function to release a critical section created with *pfOpen()*
pfEnter	[in] Pointer to a function to acquire ownership of a critical section.
pfLeave	[in] Pointer to a function to release ownership of a critical section

VE_DATA_MAPPING_INTERFACE

Description

Set of functions that define the interface of the optional Data Mappings service. The prototypes of these functions are specified in the section **Data Mappings service** of the **Function directory**.

Members

pfOpen	[in] Pointer to a function to create a data mapping.
pfClose	[in] Pointer to a function to release a data mapping created with *pfOpen()*
pfMap	[in] Pointer to a function to get read-only access to a data block from a data mapping
pfUnmap	[in] Pointer to a function to release the access to a mapped data block acquired with *pfMap()*
pfFreeze	[in] Pointer to a function to freeze the currenly mapped data block on a data mapping. This function pointer is optional, and may be NULL.

VE_DATA_STREAM_INTERFACE

Description

Set of functions that define the interface of the Data Streams service. The prototypes of these functions are specified in the section **Data Streams service** of the **Function directory**.

Members

pfOpen	[in] Pointer to a function to open a data stream.
pfClose	[in] Pointer to a function to close a data stream created with *pfOpen()*
pfRead	[in] Pointer to a function to read a block of data from a data stream
pfSeek	[in] Pointer to a function to change the position for the next I/O operation on a data stream
pfGetSize	[in] Pointer to a function to learn the total size of the data available from a data stream.
pfError	[in] Pointer to a function to check the error indicator of a data stream.
pfWrite	[in] Pointer to a function to write a block of data to a data stream. This function pointer is optional and may be NULL. It is only called when the Vocalizer Expressive build includes extra logging.

VE_FREQUENCY

```
enum VE_FREQUENCY {
    VE_FREQ_8KHZ =                      8,
    VE_FREQ_11KHZ =                     11,
    VE_FREQ_16KHZ =                     16,
    VE_FREQ_22KHZ =                     22
}
```

Description

Enumeration of possible frequencies.

Members

VE_FREQ_8KHZ	For 8 kHz PCM output: not supported
VE_FREQ_11KHZ	For 11 kHz PCM output: not supported
VE_FREQ_16KHZ	For 16 kHz PCM output: not supported
VE_FREQ_22KHZ	For 22 kHz PCM output

VE_HEAP_INTERFACE

Description

Set of functions that define the interface of the Heap service. The prototypes of these functions are specified in the section **Heap service** of the **Function directory**.

Members

pfMalloc	[in] Pointer to a function to allocate a block of memory.
pfCalloc	[in] Pointer to a function to allocate a block of memory initialized with zeroes.
pfRealloc	[in] Pointer to a function to reallocate a block of memory
pfFree	[in] Pointer to a function to free an allocated block of memory

VE_INITMODE

```
typedef enum {
  VE_INITMODE_LOAD_ONCE_OPEN_ALL       = 0xC,
  VE_INITMODE_LOAD_OPEN_ALL_EACH_TIME  = 0x3,
} VE_INITMODE;
```

Description

Enumerating all possible TTS initialization modes.

Members

VE_INITMODE_LOAD_ONCE_OPEN_ALL

Load and open all modules once (modules remain loaded until **ve_ttsClose**() is called)

VE_INITMODE_LOAD_OPEN_ALL_EACH_TIME

Load and open all modules for each speak request.

See also VE_PARAMID

VE_INSTALL

```
typedef struct {
          VE_VERSION                       fmtVersion;
    const char                           * pszBrokerInfo;
    const VE_HEAP_INTERFACE              * pIHeap;
          void                           * hHeap;
    const VE_CRITSEC_INTERFACE           * pICritSec;
          void                           * hCCritSec;
    const VE_DATA_STREAM_INTERFACE       * pIDataStream;
    const VE_DATA_MAPPING_INTERFACE      * pIDataMapping;
          void                           * hCData;
    const VE_LOG_INTERFACE               * pILog;
          void                           * hLog;
} VE_INSTALL;
```

Description

Structure containing the supplied external services and describing installed configurations.

Members

fmtVersion	Version of this structure. Set it to VE_CURRENT_VERSION.
pszBrokerInfo	Broker information., a null terminated string that concatenates the pipeline headers of the installed configurations.
pIHeap	Interface of the Heap service. The TTS class and its instances will call these heap functions for memory management.
hHeap	Heap handle of the Heap service. The TTS class will pass this heap handle as argument to *pIHeap* function calls; individual TTS instances get their own heap handle passed through **ve_ttsOpen**().
pICritSec	Interface of the Critical Sections service. This interface is optional and may be NULL if thread-safe operation is not required. The TTS class and its instances will call these functions to create critical sections, and to use them to work in a thread-safe mode.
hCCritSec	Class handle of the Critical Sections service. The TTS class and its instances will pass this class handle as argument on *pICritSec->pfOpen()* function calls to create critical sections.
pIDataStream	Interface of the Data Streams service. The TTS class and its instances call these functions to request access to data by name.
pIDataMapping	Interface of the Data Mappings service. This interface is optional and may be NULL. If available, it takes precedence over pIDataStream, meaning that the TTS class and its instances call these functions instead of the data stream functions to request read-only access to data by name.
hCData	Class handle of the data access services. The TTS class and its instances pass this class handle as an argument on *pIDataStream->pfOpen()* and *pIDataMapping->pfOpen()* calls to create data streams or data mappings.
pILog	Interface of the User Log service. The TTS class and its instances will call these functions for reporting error and diagnostic messages.

	This interface is optional and may be NULL if a user log is not required.
hLog	Log handle of the User Log service.
	The TTS class will pass this handle as argument to *pILog* function calls; individual TTS instances get their own log handle passed through **ve_ttsOpen**()..

VE_INTEXT

```
typedef struct {
    VE_TEXTFORMAT             eTextFormat;
    size_t                    cntTextLength;
    void                      *szInText;
}        VE_INTEXT;
```

Description

The structure VE_INTEXT is used to transfer the input text from the input device to the calling TTS instance via a callback function implemented by the application (the input device).

Members

eTextFormat	Format of the input text found in the buffer *szInText*.
cntTextLength	Length of the input text found in the buffer *szInText*, in bytes.
szInText	Pointer to the input text to be processed

VE_LANGUAGE

```
typedef struct {
    char         szLanguage[VE_MAX_STRING_LENGTH];
    char         szLanguageTLW[4];
    char         szVersion[VE_MAX_STRING_LENGTH];
}        VE_ LANGUAGE;
```

Description

Information on a language.

Members

szLanguage	Name of language
szLanguageTLW	3-letter language code (e.g. ENU)
szVersion	Version string

See also ve_ttsGetLanguageList()

3-letter language codes

Language name	language code
Arabic	ARW
American English	ENU
Australian English	ENA
Belgian Dutch	DUB
Brazilian Portuguese	PTB
American English	ENU
Australian English	ENA
Brazilian Portuguese	PTB
British English	ENG
Canadian French	FRC
Chinese Mandarin	MNC
Czech	CZC
Danish	DAD
Dutch	DUN
Finnish	FIF
French	FRF
German	GED
Greek	GRG
Hindi	HII
Hong Kong Cantonese	CAH
Hungarian	HUH
Indian English	ENI
Indonesian	IDI
Italian	ITI
Japanese	JPJ
Korean	KOK
Mexican Spanish	SPM
Norwegian	NON
Polish	PLP
Portuguese	PTP
Romanian	ROR
Russian	RUR
Spanish	SPE
Swedish	SWS
Taiwanese Mandarin	MNT
Thai	THT
Turkish	TRT

Symbian Language IDs

Symbolic Name	enum	value
	ELanguageTest	0
UK English	ELanguageEnglish	1
French	ELanguageFrench	2

Chapter IV

Symbolic Name	enum	value
German	ELanguageGerman	3
Spanish	ELanguageSpanish	4
Italian	ELanguageItalian	5
Swedish	ELanguageSwedish	6
Danish	ELanguageDanish	7
Norwegian	ELanguageNorwegian	8
Finnish	ELanguageFinnish	9
American	ELanguageAmerican	10
Swiss French	ELanguageSwissFrench	11
Swiss German	ELanguageSwissGerman	12
Portuguese	ELanguagePortuguese	13
Turkish	ELanguageTurkish	14
Icelandic	ELanguageIcelandic	15
Russian	ELanguageRussian	16
Hungarian	ELanguageHungarian	17
Dutch	ELanguageDutch	18
Belgian Flemish	ELanguageBelgianFlemish	19
Australian English	ELanguageAustralian	20
Belgian French	ELanguageBelgianFrench	21
Austrian German	ELanguageAustrian	22
New Zealand English	ELanguageNewZealand	23
International French	ELanguageInternationalFrench	24
Czech	ELanguageCzech	25
Slovak	ELanguageSlovak	26
Polish	ELanguagePolish	27
Slovenian	ELanguageSlovenian	28
Taiwanese Chinese	ELanguageTaiwanChinese	29
Hong Kong Chinese	ELanguageHongKongChinese	30
Peoples Republic of Chinas Chinese	ELanguagePrcChinese	31
Japanese	ELanguageJapanese	32
Thai	ELanguageThai	33
Afrikaans	ELanguageAfrikaans	34
Albanian	ELanguageAlbanian	35
Amharic	ELanguageAmharic	36
Arabic	ELanguageArabic	37
Armenian	ELanguageArmenian	38
Tagalog	ELanguageTagalog	39
Belarussian	ELanguageBelarussian	40
Bengali	ELanguageBengali	41
Bulgarian	ELanguageBulgarian	42
Burmese	ELanguageBurmese	43

Symbolic Name	enum	value
Catalan	ELanguageCatalan	44
Croation	ELanguageCroatian	45
Canadian English	ELanguageCanadianEnglish	46
International English	ELanguageInternationalEnglish	47
South African English	ELanguageSouthAfricanEnglish	48
Estonian	ELanguageEstonian	49
Farsi	ELanguageFarsi	50
Canadian French	ELanguageCanadianFrench	51
Gaelic	ELanguageScotsGaelic	52
Georgian	ELanguageGeorgian	53
Greek	ELanguageGreek	54
Cyprus Greek	ELanguageCyprusGreek	55
Gujarati	ELanguageGujarati	56
Hebrew	ELanguageHebrew	57
Hindi	ELanguageHindi	58
Indonesian	ELanguageIndonesian	59
Irish	ELanguageIrish	60
Swiss Italian	ELanguageSwissItalian	61
Kannada	ELanguageKannada	62
Kazakh	ELanguageKazakh	63
Kmer	ELanguageKhmer	64
Korean	ELanguageKorean	65
Lao	ELanguageLao	66
Latvian	ELanguageLatvian	67
Lithuanian	ELanguageLithuanian	68
Macedonian	ELanguageMacedonian	69
Malay	ELanguageMalay	70
Malayalam	ELanguageMalayalam	71
Marathi	ELanguageMarathi	72
Moldovian	ELanguageMoldavian	73
Mongolian	ELanguageMongolian	74
Norwegian Nynorsk	ELanguageNorwegianNynorsk	75
Brazilian Portuguese	ELanguageBrazilianPortuguese	76
Punjabi	ELanguagePunjabi	77
Romanian	ELanguageRomanian	78
Serbian	ELanguageSerbian	79
Sinhalese	ELanguageSinhalese	80
Somali	ELanguageSomali	81
International Spanish	ELanguageInternationalSpanish	82
American Spanish	ELanguageLatinAmericanSpanish	83

Symbolic Name	enum	value
Swahili	ELanguageSwahili	84
Finland Swedish	ELanguageFinlandSwedish	85
reserved for future use	ELanguageReserved1	86
Tamil	ELanguageTamil	87
Telugu	ELanguageTelugu	88
Tibetan	ELanguageTibetan	89
Tigrinya	ELanguageTigrinya	90
Cyprus Turkish	ELanguageCyprusTurkish	91
Turkmen	ELanguageTurkmen	92
Ukrainian	ELanguageUkrainian	93
Urdu	ELanguageUrdu	94
reserved for future use	ELanguageReserved2	95
Vietnamese	ELanguageVietnamese	96
Welsh	ELanguageWelsh	97
Zulu	ELanguageZulu	98
@deprecated 6.2	ELanguageOther	99
@deprecated 6.2	ELanguageNone	0xFFFF

No Symbian Language IDs are available for ENI and ENA

VE_LIPSYNC

```
typedef          struct {
    NUAN_S16            sJawOpen;
    NUAN_S16            sTeethUpVisible;
    NUAN_S16            sTeethLoVisible;
    NUAN_S16            sMouthHeight;
    NUAN_S16            sMouthWidth;
    NUAN_S16            sMouthUpturn;
    NUAN_S16            sTonguePos;
    NUAN_S16            sLipTension;
    char                szLHPhoneme[VE_MAX_PHONEMELEN];
}       VE_LIPSYNC;
```

Description

Lip synchronization structure.

Members

sJawOpen — Opening angle of the jaw on a 0 to 255 linear scale, where 0 = fully closed, and 255 = completely open.

sTeethUpVisible — Indicates if upper teeth are visible on a 0 to 255 linear scale, where 0 = upper teeth are completely hidden, 128 = only the teeth are visible, and 255 = upper teeth and gums are completely exposed.

sTeethLoVisible	Indicates if lower teeth are visible on a 0 to 255 linear scale, where 0 = lower teeth are completely hidden, 128 = only the teeth are visible, and 255 = lower teeth and gums are completely exposed.
sMouthHeight	Mouth height on a 0 to 255 linear scale, where 0 = minimum height (mouth and lips are closed) and 255 = maximum possible height for the mouth.
sMouthWidth	Mouth or lips width on a 0 to 255 linear scale, where 0 = minimum width (mouth and lips are puckered) and 255 = maximum possible width for the mouth.
sMouthUpturn	Indicates how much the mouth is turned up at the corners on a 0 to 255 linear scale, where 0 = mouth corners turning down, 128 = neutral, and 255 = mouth is fully upturned.
sTonguePos	Indicates the tongue position relative to the upper teeth on a 0 to 255 linear scale, where 0 = tongue is completely relaxed, and 255 = tongue is against the upper teeth.
sLipTension	Lip tension on a 0 to 255 linear scale, where 0 = lips are completely relaxed, and 255 = lips are very tense.
szLHPhoneme	Matching L&H+ phonetic symbol.

See also ve_ttsGetLipSyncInfo()

VE_LOG_INTERFACE

Description

Set of functions that define the interface of the User Log service. The prototypes of these functions are specified in the section **User Log service** of the **Function directory**.

Members

pfError	[in] Pointer to a function to report an error by ID.
pfDiagnostic	[in] Pointer to a function to report a diagnostic message.

VE_MARKERMODE

```
typedef enum {
   VE_MRK_OFF = 0,
```

```
        VE_MRK_ON   = 1
    }        VE_MARKERMODE;
```

Description

Control generation of markers, such as phoneme markers.

Members

`VE_MRK_OFF`

Turn off marker generation

`VA_MRK_OFF`

Turn on marker generation

VE_MARKINFO

```
    typedef struct {
         NUAN_U32                       ulMrkInfo;
         VE_MARKTYPE                    eMrkType;
         size_t                         cntSrcPos;
         size_t                         cntSrcTextLen;
         size_t                         cntDestPos;
         size_t                         cntDestLen;
         NUAN_U16                       usPhoneme;
         NUAN_U32                       ulMrkId;
         NUAN_U32                       ulParam;
         char                           *szPromptID;
    }        VE_MARKINFO;
```

Description

Definition of the marker information structure.

Members

ulMrkInfo	Marker specific info (reserved for future)
eMrkType	Type of marker
`cnt`*SrcPos*	Marker position (as byte offset) in the input text
`cnt`*SrcTextLen*	Length (in bytes) of the piece of text covered by the marker
`cnt`*DestPos*	Marker position (as an offset in samples) in the output PCM data
`cnt`*DestLen*	Length (in samples) of the audio fragment covered by the marker
usPhoneme	Used for phoneme markers: L&H+ phoneme symbol ID
ulMrkId	Used for bookmark markers: Marker ID; which corresponds to the unsigned integer specified in the bookmark control sequence.
ulParam	Parameter value

szPromptID	Prompt identification string, as provided in Vocalizer Studio.

Comments

The parameter *usPhoneme* is an index of the L&H+ phoneme table. To get the L&H+ phoneme string, the user must call the function **ve_ttsGetLipSyncInfo()**.

For more info about L&H+ phonetic symbols, refer to the section Entering Phonetic Input in the **User's Guide for <Language>.**

This table shows which fields are supported for a marker type (an unsupported field keeps value 0): V=supported, X=not supported:

<u>**eMrkType**</u>	*ulSrcPos*	*ulSrcTextLen*	*ulDestPos*	*ulDestLen*
VE_MRK_TEXTUNIT	V	V	V	X
VE_MRK_WORD	V	V	V	X
VE_MRK_PHONEME	X	X	V	X
VE_MRK_BOOKMARK	V	V	V	X
VE_MRK_PROMPT	V	X	V	X

VE_MARKTYPE

```
enum VE_MARKTYPE {
     VE_MRK_TEXTUNIT            = 0x0001,
     VE_MRK_WORD                = 0x0002,
     VE_MRK_PHONEME             = 0x0004,
     VE_MRK_BOOKMARK            = 0x0008,
     VE_MRK_SILENCE             = 0x0010,
     VE_MRK_PROMPT              = 0x0400
};
```

Description

Definition of marker types.

Members

VE_MRK_TEXTUNIT	Text unit marker; it marks the start of a text unit (e.g. a sentence for sentence-by-sentence read mode)
VE_MRK_WORD	Word marker; which identifies a word in the input text and its spoken version as an audio fragment in the output PCM stream.
VE_MRK_PHONEME	Phoneme marker: it identifies a

	phoneme and its appearance in the output PCM stream.
VE_MRK_BOOKMARK	Bookmark marker : it marks the occurrence of a bookmark control sequence <*ESC*>*mrk*=<*name*>\\ in the input text.
VE_MRK_PROMPT	Prompt marker: it identifies an ActivePrompt used for synthesis at a given position in the input text. The prompt ID is returned as a char string in the szPromptID field of the marker. This string is owned by the TTS instance, and only valid during the call to the Output Delivery service.

VE_MSG

```
enum VE_MSG {
    VE_MSG_BEGINPROCESS   = 0x00000001,
    VE_MSG_ENDPROCESS     = 0x00000002,
    VE_MSG_PROCESS        = 0x00000004,
    VE_MSG_OUTBUFREQ      = 0x00000008,
    VE_MSG_OUTBUFDONE     = 0x00000010,
    VE_MSG_STOP           = 0x00000020,
    VE_MSG_PAUSE          = 0x00000040,
    VE_MSG_RESUME         = 0x00000080,
    VE_MSG_TAIBEGIN       = 0x00000100,
    VE_MSG_TAIEND         = 0x00000200,
    VE_MSG_TAIBUFREQ      = 0x00000400,
    VE_MSG_TAIBUFDONE     = 0x00000800
}
```

Description

Enumeration of messages notified to the application. For the description of different callback messages, see the section on **Notification messages**.

Members

VE_MSG_BEGINPROCESS

This message is issued when the TTS system starts to generate speech output.

VE_MSG_ENDPROCESS

This message is issued when the TTS system finishes generating speech output and there is no more text input.

VE_MSG_OUTBUFREQ

This message is issued when the TTS system requires data buffers in order to generate PCM data and markers.

VE_MSG_OUTBUFDONE

This message is issued when the TTS system finishes generating a PCM data buffer and/or a marker buffer, and makes this available in a VE_OUTDATA structure.

VE_MSG_STOP

This message is issued when the TTS system receives a request to stop synthesis. (The stop is not complete until VE_MSG_ENDPROCESS is received.)

VE_MSG_PAUSE

This message is issued when the TTS system is paused by the function **ve_ttsPause()**

VE_MSG_RESUME

This message is issued when the TTS system is resumed by the function **ve_ttsResume()**

VE_MSG_PROCESS

Supported in cooperative speak mode only. This message is issued whenever control is returned to the calling application in between receiving VE_MSG_BEGINPROCESS and VE_MSG_ENDPROCESS.

VE_MSG_TAIBEG

This message is issued when the TTS system starts to scan the input text and generate text analysis (TA) info.

VE_MSG_TAIEND

This message is issued when the TTS system finishes scanning the input text.

VE_MSG_TAIBUFREQ

This message is issued when the TTS system requires data buffers for the TA info and the input text rewritten by loaded user rulesets.

VE_MSG_TAIBUFDONE

This message is issued when the TTS system has a block of TA info and rewritten text available in a VE_OUTTAINFO structure.

VE_NTSINFO

```
typedef struct {
    char          szVersion[VE_MAX_STRING_LENGTH];
}       VE_NTSINFO;
```

Description

The structure VE_NTSINFO is used to transfer NT-SAMPA information to the user.

Members

szVersion	NT-SAMPA version info. Tells the L&H+ version and the NT- SAMPA version.

VE_OUTDATA

```
typedef struct {
     VE_AUDIOFORMAT                    eAudioFormat;
     size_t                            cntPcmBufLen;
     Void                              *pOutPcmBuf;
     size_t                            cntMrkListLen
     VE_MARKINFO                       *pMrkList;
}        VE_OUTDATA
```

Description

The structure VE_OUTDATA is used to transfer the generated audio buffer and markers to the application via a callback function.

It is also used to provide empty buffers from the application to the TTS engine.

Members

eAudioFormat	Output data format
`cnt`*PcmBufLen*	Length of PCM data buffer (*pOutPcmBuf*) in bytes.
pOutPcmBuf	Pointer to the PCM data buffer.
cntMrkListLen	Size of the marker information buffer (*pMrkList*) in bytes.
pMrkList	Pointer to an array of marker information structures.

VE_OUTDEVINFO

```
typedef struct {
     void *                            pUserData;
     VE_CBOUTNOTIFY                    *pfOutNotify;
}        VE_OUTDEVINFO;
```

Description

The structure VE_OUTDEVINFO describes the output device.

Members

pUserData	Handle to the output device.
pfOutNotify	Pointer to the output callback function

Chapter IV

VE_OUTTAINFO

```
typedef struct {
     size_t         cntRewrittenTextLen
     char           *pRewrittenTextBuf
     size_t         cntTaInfoListLen
     void           *pTaInfoList
}    VE_OUTTAINFO;
```

Description

The structure VE_OUTTAINFO is used to transfer the output of text analysis to the application via the Output Delivery service. The output data are text analysis info and text rewritten by loaded user rulesets.

Vocalizer Expressive also calls this external service to request the application for data buffers in this structure.

Members

cntRewrittenTextLen	Size (in bytes) of the rewritten text block *pRewrittenTextBuf*.
pRewrittenTextBuf	Pointer to the block of rewritten text (encoded in UTF-16)
cntTaInfoListLen	Number of text analysis info records in *pTaInfoList*.
pTaInfoList	Pointer to the list of text analysis info records.

VE_PARAM

```
typedef struct {
     VE_PARAMID                    eID;
     VE_PARAM_VALUE                uValue;
}        VE_PARAM;
```

Description

Definition of the control parameter value.

Members

eID	Specifies the identifier of the parameter
uValue	Parameter value

VE_PARAM_VALUE

```
typedef union {
   NUAN_U16   usValue;
   char       szStringValue[VE_MAX_STRING_LENGTH];
}        VE_PARAM_VALUE;
```

Description

Definition of different parameter values.

Members

usValue	Used to set and get all parameters except the voice, language, pre-processor mode, and voice operating point
szStringValue	String used to set and get string parameters: voice, language, pre-processor mode, and voice operating point.

VE_PARAMID

```
typedef enum {
     VE_PARAM_LANGUAGE                = 1,
     VE_PARAM_VOICE                   = 2,
     VE_PARAM_VOICE_OPERATING_POINT   = 3,
     VE_PARAM_FREQUENCY               = 4,
     VE_PARAM_EXTRAESCLANG            = 5,
     VE_PARAM_EXTRAESCTN              = 6,
     VE_PARAM_TYPE_OF_CHAR            = 7,
     VE_PARAM_VOLUME                  = 8,
     VE_PARAM_SPEECHRATE              = 9,
     VE_PARAM_PITCH                   = 10,
     VE_PARAM_WAITFACTOR              = 11,
     VE_PARAM_READMODE                = 12,
     VE_PARAM_TEXTMODE                = 13,
     VE_PARAM_MAX_INPUT_LENGTH        = 14,
     VE_PARAM_LIDSCOPE                = 15,
     VE_PARAM_LIDVOICESWITCH          = 16,
     VE_PARAM_LIDMODE                 = 17,
     VE_PARAM_LIDLANGUAGES            = 18,
     VE_PARAM_MARKER_MODE             = 19,
     VE_PARAM_INITMODE                = 20
}        VE_PARAMID;
```

Description

Identifier of different parameters.

Members

VE_PARAM_FREQUENCY

Sampling frequency, see the definition of VE_FREQUENCY.

Parameter value field: usValue.

VE_PARAM_VOLUME

Volume level on a 0 to 100 scale. For each 10 points on the scale the volume changes by 3 dB.

Default value: 80

Parameter value field: usValue.

VE_PARAM_SPEECHRATE

Speech rate level, which is a scale factor (in %) on the default speech rate of the current voice. The valid range is [50..400], with 50 having the voice speak 2x slower, and 400 having the voice speak 4x faster.

Default value: 100 (%)

Parameter value field: usValue.

VE_PARAM_PITCH

Pitch level, a scale factor (in %) on the inherent pitch of the current voice. The range is [50..200]; with value 50 the voice speaks one octave lower (pitch :2), with value 200 the voice speaks one octave higher (pitch x2).

Default value: 100 (%)

Parameter value field: usValue

VE_PARAM_WAITFACTOR

Wait period inserted between two text units (e.g. sentences), on a scale from 0 to 9. Each unit is equivalent to 200ms of silence.

Default value: 1

Parameter value field: usValue.

VE_PARAM_READMODE

Read mode, see the definition of VE_READMODE.

Default value: VE_READMODE_SENT, which has the product read sentence by sentence.

Parameter value field: usValue.

VE_PARAM_LANGUAGE

Language name as found in the broker header files. See also VE_PARAM_LANGUAGE_NR.

Parameter value field: szStringValue.

VE_PARAM_VOICE

Voice name string.

Parameter value field: szStringValue.

VE_PARAM_TYPE_OF_CHAR

Character encoding for the synthesis input text passed into **vmobile_ttsProcessText2Speech()**, VE_TYPE_OF_CHAR_UTF16 for platform-endian UTF-16, or VE_TYPE_OF_CHAR_UTF8 for UTF-8.

Default value: VE_TYPE_OF_CHAR_UTF16

Parameter value field: usValue.

VE_PARAM_MARKER_MODE

Enable/disable marker generation. Can only have the values enumerated in the VE_MARKERMODE type definition.

Default value: VE_MRK_OFF, which disables marker generation.

Parameter value field: usValue.

VE_PARAM_INITMODE

VE_INITMODE_LOAD_ONCE_OPEN_ALL : all components are loaded and the objects are opened by **ve_ttsSetParamList()**. Unloading is done by **ve_ttsClose()**.

VE_INITMODE_LOAD_OPEN_ALL_EACH_TIME : No components are loaded by **ve_ttsSetParamList()**. All components are loaded and the objects are opened before each speak request. The components are unloaded and the objects are closed after each speak request.

Default value: VE_INITMODE_LOAD_ONCE_OPEN_ALL

Parameter value field: usValue

VE_PARAM_TEXTMODE

Text processing mode for the synthesis input text passed into **ve_ttsProcessText2Speech()**. This supports the values, enumerated in the VE_TEXTMODE type definition. When setting the value to VE_TEXTMODE_SMS, additional text processing will be enabled for better processing of SMS input.

Default value: VE_TEXTMODE_STANDARD

Parameter value field: usValue

VE_PARAM_MAX_INPUT_LENGTH

Maximum length (in characters) for a single sentence. Values between 25 and 2500 are supported. Text fragments that are larger than this size and don't contain regular end-of-sentence punctuation, will be cut at or below this size and spoken as 2 or more separate sentences.

Default value: 250

Parameter value field: usValue.

VE_PARAM_VOICE_OPERATING_POINT

Name of the voice operating point. The supported names are the following:

- "premium-high" for the Premium High voice operating point
- "embedded-high" for the Embedded High voice operating point
- "embedded-pro" for the Embedded Pro voice operating point
- "embedded-compact" for the Embedded Compact voice operating point

Parameter value field: szStringValue.

VE_PARAM_LIDSCOPE

Defines the parts of the input text for which Vocalizer Expressive will identify the language of the text (and not expect that it's identical to the native language of the current voice). The supported values are enumerated in VE_LIDSCOPE:

- VE_LIDSCOPE_NONE: Language identification (LID) is deactivated for the entire input text.
- VE_LIDSCOPE_USERDEFINED: LID is activated for fragments tagged by *<ESC>\lang=unknown*. This is the default value.
- VE_LIDSCOPE_MESSAGE: LID is activated for the entire input text, and the language of the text is determined text element by text element (as set by the current read mode).

Parameter value field: usValue.

VE_PARAM_LIDVOICESWITCH

Defines whether Vocalizer Expressive is to switch the voice when it detects foreign input, i.e. the language identification is activated and identifies the language of the text as different from the native language of the voice. Supported values are enumerated in VE_LIDVOICESWITCH:

- VE_LIDVOICESWITCH_OFF: Keep the current voice to read foreign input. A multi-lingual voice may read the foreign input according to the rules of the foreign language. A mono-lingual voice will read it according to its native rules.
- VE_LIDVOICESWITCH_ON: Switch to a voice that has the foreign language as its native language.

Default value: VE_LIDVOICESWITCH_OFF

Parameter value field: usValue.

Chapter IV

VE_PARAM_EXTRAESCLANG

Defines the foreign languages that may appear in the input text. The value is a comma-separated list of 3-letter language codes, e.g. “eng,frf,spe,iti”.

If the current voice supports one or more of these languages as foreign languages, it will load the foreign language data of concern. To learn about the foreign languages supported by a voice refer to the language and voice documentation. The function **ve_ttsSetParamList()** merely sets the value, and doesn't indicate which languages in the list are supported by the current voice.

Parameter value field: szStringValue.

VE_PARAM_EXTRAESCTN

Defines the additional tn types that may appear in the input text. The value is a comma-separated list of types as used in *<ESC>\tn=<type>*, but Vocalizer Expressive currently only supports the value “mpthree” on multi-lingual voices.

If the current voice supports one or more of these additional types, it will load the language data of concern. To learn the additional tn types supported by a voice refer to the voice-specific documentation supplement. The function **ve_ttsSetParamList()** merely sets the value, and doesn't indicate which types in the list are supported by the current voice..

Parameter value field: szStringValue.

VE_PARAM_LIDMODE

Configures the operating mode of the language identification (LID). The supported values are enumerated in VE_LIDMODE:

- VE_LIDMODE_MEMORY_BIASED: LID takes the detected language of preceding sentence into account to determine the language of the current sentence (“yesterday's weather” principle). This mode is recommended for input like e-mails and news paragraphs that are preferably read in a single language.
- VE_ LIDMODE_FORCED_CHOICE: LID only considers the current sentence to determine its language. This mode is recommended for single entries from a domain like music or navigation.

Default value: VE_LIDMODE_MEMORY_BIASED

Parameter value field: szStringValue.

VE_PARAM_LIDLANGUAGES

Restricts the language identification (LID) to a subset of the supported foreign languages. The value is a comma-separated list of 3-letter language codes, e.g. “eng,frf,spe,iti”.
Be aware that LID is by design limited to detecting the following language families only:
rur,dux,enx,frx,spx,ged,iti,sws,non,dad,ptx,bae,trt,plp,czc
The language families that have an ‘x’ at the end, cover any language within that family. E.g. enx can be enu, eng, enz, etc.

Note that LID results are limited to the available languages. It will not detect a language that is not installed.

Be aware that VE_PARAM_LIDLANGUAGES must be a subset of the above supported LID language list.

Parameter value field: szStringValue.

See also VE_PARAM

VE_PRODUCT_VERSION

```
typedef struct {
    NUAN_U8                     major;
    NUAN_U8                     minor;
    NUAN_U8                     maint;
}        VE_PRODUCT_VERSION;
```

Description

The structure VE_PRODUCT_VERSION is filled in by the function **ve_GetProductVersion()**. On a successful return it contains the major, minor and maintenance numbers of the Vocalizer Expressive product. The major and minor version number define the feature set of the product, the maintenance version number refers patches of fixes.

Members

major	Major revision number.
minor	Minor revision number.
maint	Maintenance revision number

VE_READMODE

```
typedef enum {
    VE_READMODE_SENT =        1,
    VE_READMODE_CHAR =        2,
    VE_READMODE_WORD =        3,
    VE_READMODE_LINE =        4
}        VE_READMODE;
```

Description

This enumeration describes different read modes for the TTS system. This read mode determines the way in which the system will split the input text into text units. Each text unit will then be separately processed and pronounced by the TTS system.

Members

VE_READMODE_SENT

Sentence-by-sentence (default read mode)

VE_READMODE_CHAR

Character-by-character (spelling)

VE_READMODE_WORD

Word-by-word mode.

VE_READMODE_LINE

Line-by-line. A line is terminated by "\n" or "\r\n".

VE_SPEECHDBINFO

```
typedef struct {
   char   szVersion[VE_MAX_STRING_LENGTH];
   char   szLanguage[VE_MAX_STRING_LENGTH];
   char   szVoiceName[VE_MAX_STRING_LENGTH];
   Char   szVoiceOperatingPoint[VE_MAX_STRING_LENGTH];
   NUAN_U16  u16Freq;
}      VE_SPEECHDBINFO;
```

Description1

Information on speech databases.

Members

szLanguage	The language name
szVoiceName	The voice name
szVersion	The voice speech database version
szVoiceOperatingPoint	The voice operating point
u16Freq	The frequency

See also ve_ttsGetSpeechDBList()

VE_STREAM_DIRECTION

```
typedef enum {
   VE_STREAM_BACKWARD,
   VE_STREAM_FORWARD
} VE_STREAM_DIRECTION;
```

Description

This enumeration describes the direction for **pfSeek()** on a data stream.

Members

VE_STREAM_BACKWARD	Move towards the beginning of the data stream.
VE_STREAM_FORWARD	Move towards the end of the data stream.

See also VE_DATA_STREAM_INTERFACE

VE_STREAM_ORIGIN

```
typedef enum {
   VE_STREAM_SEEK_SET,
   VE_STREAM_SEEK_CUR,
   VE_STREAM_SEEK_END
} VE_STREAM_ORIGIN;
```

Description

This enumeration describes the origin for **pfSeek()** on a data stream.

Members

VE_STREAM_SEEK_SET	The origin is the beginning of the data stream.
VE_STREAM_SEEK_CUR	The origin is the current position within the data stream.
VE_STREAM_SEEK_END	The origin is the end of the data stream.

See also VE_DATA_STREAM_INTERFACE

VE_TAITYPE

```
typedef enum {
       VE_TAI_TEXTUNIT              = 0x0001,
       VE_TAI_BOOKMARK              = 0x0002
}    VE_TAITYPE;
```

Description

Identifier of different types of jump points generated by text analysis.

Members

VE_TAI_TEXTUNIT

Jump point defined by a sentence boundary.

VE_TAI_BOOKMARK

Jump point defined by a bookmark (control sequence *<ESC>\mrk=<nr>*.

VE_TA_NODE

```
typedef struct {
     VE_TAITYPE        jmpPointType
     size_t            positionInText
     void              *stateInfo
     char              languageIdent[16]
}    VE_TA_NODE;
```

Description

The structure VE_TA_NODE defines a jump point for text analysis and traversal. It is generated as a result of the text analysis phase.

Members

jmpPointType	Type of the jump point.
positionInText	Offset (in bytes) of the jump point in the text rewritten by the loaded user rulesets.
stateInfo	Pointer to the state at the jump point as it is affected by previous control sequences.
languageIdent	Language identification string. This is a 3-letter language code with "_lid" appended, e.g. "eng_lid" in case that Vocalizer Expressive has detected the language at the jump point, or the plain 3-letter code in case that the language of the text at the jump point is defined by the user (through *<ESC>\lang=<s>*).

VE_TEXTFORMAT

```
typedef enum {
```

```
    VE_NORM_TEXT = 0,
    VE_SSML_TEXT = 1
} VE_TEXTFORMAT;
```

Description

This enumeration describes the supported text format.

Members

VE_NORM_TEXT	Normal text
VE_SSML_TEXT	Available on demand via dedicated build.

VE_TEXTMODE

```
typedef enum {
    VE_TEXTMODE_STANDARD =        1,
    VE_TEXTMODE_SMS =             2,
} VE_TEXTMODE;
```

Description

This enumeration describes the text processing mode for the synthesis input text passed into **ve_ttsProcessText2Speech()**,.

Members

VE_TEXTMODE_STANDARD	Regular input text
VE_TEXTMODE_SMS	SMS input text

VE_TYPE_OF_CHAR

```
typedef enum {
    VE_TYPE_OF_CHAR_UTF16 =        1,
    VE_TYPE_OF_CHAR_UTF8 =         2,
} VE_TYPE_OF_CHAR;
```

Description

This enumeration describes the character encoding for the synthesis input text passed into **ve_ttsProcessText2Speech()**,

Members

VE_TYPE_OF_CHAR_UTF16	16-bit platform-endian Unicode UTF-16
VE_TYPE_OF_CHAR_UTF8	8-bit Unicode UTF-8

VE_VOICEINFO

```
typedef struct {
    char          szVersion[VE_MAX_STRING_LENGTH];
    char          szLanguage[VE_MAX_STRING_LENGTH];
    char          szVoiceName[VE_MAX_STRING_LENGTH];
    char          szVoiceAge[VE_MAX_STRING_LENGTH];
    char          szVoiceType[VE_MAX_STRING_LENGTH];
}         VE_VOICEINFO;
```

Description

Information on a voice.

Members

szVersion	The voice version
szLanguage	The language name
szVoiceName	The voice name
szVoiceAge	Age of the speaker
szVoiceType	Voice type of the speaker (male, female, or neutral)

See also ve_ttsGetVoiceList()

Chapter IV

Return codes

Warnings

Value	Meaning
NUAN_W_ALREADYPRESENT	Object already present
NUAN_W_CHARSKIPPED	Characters skipped during a conversion
NUAN_W_ENDOFINPUT	No more input to process
NUAN_W_EOF	End of File reached
NUAN_W_FALSE	False success
NUAN_W_NOINPUTTEXT	No input text
NUAN_W_NON_DOCUMENTED_WARNING	Not documented warning

General return and error codes

Value	Meaning
NUAN_OK	Successful case
NUAN_E_ALREADYDEFINED	Object already defined
NUAN_E_ALREADYINITIALIZED	The API is already initialized
NUAN_E_BUFFERTOOSMALL	Buffer is too small
NUAN_E_BUSY	Instance is busy
NUAN_E_CONVERSIONFAILED	A string conversion has failed
NUAN_E_COULDNOTOPENFILE	Could not open file
NUAN_E_DATA_IN_USE	A data buffer in use
NUAN_E_DICT_CORRUPTBUFFER	The provided buffer is corrupt
NUAN_E_DICT_UNKNOWNSTREAMFORMAT	Unknown format specified
NUAN_E_DICT_WRONGTXTDCTFORMAT	Illegal text dictionary format
NUAN_E_EMPTY_LHSTRING	Received empty L&H+ string
NUAN_E_ENDOFINPUT	No more input to process
NUAN_E_ENGINENOTFOUND	Engine could not be found
NUAN_E_FEATEXTRACT	The feature extraction failed
NUAN_E_FILECLOSE	Error in closing a file
NUAN_E_FILENOTLOADED	Trying to use a file that was not loaded
NUAN_E_FILEREADERROR	Error while reading file
NUAN_E_FILESEEK	Seeking error in a file
NUAN_E_FILEWRITEERROR	Error while writing file
NUAN_E_FOLDERREADERROR	Error while reading folder
NUAN_E_HANDLEWASOPEN	Already open handle passed to an Open function
NUAN_E_INTFNOTFOUND	Interface could not be found
NUAN_E_INVALID_DATA	Invalid data handle

Value	Meaning
NUAN_E_INVALID_DATATYPE	A buffer with an invalid data type was supplied
NUAN_E_INVALID_FLAG_COMBINATION	An invalid flag combination was used as on of the parameters
NUAN_E_INVALIDARG	Argument is not valid
NUAN_E_INVALIDCHAR	Invalid character was used
NUAN_E_INVALIDHANDLE	Handle is not valid
NUAN_E_INVALIDPARAM	Invalid parameter value
NUAN_E_INVALIDPOINTER	An invalid pointer passed
NUAN_E_LANGUAGENOTFOUND	Language could not be found
NUAN_E_MALLOC	Memory allocation failed
NUAN_E_MAPPING	Error mapping a read-only window on a data object
NUAN_E_MAXCHANNELS	Maximum number of instances
NUAN_E_MODULENOTFOUND	A module could not be found
NUAN_E_NOK	General failure
NUAN_E_NON_DOCUMENTED_ERROR	Non documented error
NUAN_E_NOTCOMPATIBLE	Incompatible objects
NUAN_E_NOTFOUND	The object was not found
NUAN_E_NOTIMPLEMENTED	This feature is not supported
NUAN_E_NOTINITIALIZED	API is not properly initialized
NUAN_E_NULL_HANDLE	A NULL handle was passed to a function
NUAN_E_NULL_POINTER	An unexpected NULL pointer was found during processing
NUAN_E_NULLPOINTER	Null pointer as an argument
NUAN_E_OUTOFMEMORY	Not enough memory
NUAN_E_OUTOFRANGE	A value is out of range
NUAN_E_OUTOFRESOURCE	Out of resources
NUAN_E_READONLY	Object is read-only
NUAN_E_SYSTEM_ERROR	An error occurs in the system
NUAN_E_TTS_AUDIOOUTOPEN	Could not open the audio output
NUAN_E_TTS_AUDIOOUTWRITE	Could not write to the audio output
NUAN_E_TTS_DPSLINK	Internal link error
NUAN_E_TTS_DPSOVERFLOW	Internal overflow error: input text contains garbage or is too long
NUAN_E_TTS_ILLFORMEDINPUTDOC	The input document is not well formed
NUAN_E_TTS_INSTBUSY	Specified instance is busy
NUAN_E_TTS_INVALIDINPUTDOC	The input document is not valid
NUAN_E_TTS_MISSING_OUTDEVICE	Missing output device, i.e. no callback function
NUAN_E_TTS_NOINPUTTEXT	No input text has been found
NUAN_E_TTS_NOLANGUAGE	No language has been selected
NUAN_E_TTS_NOMORETEXT	No more text to send to the

Value	Meaning
	TTS system
NUAN_E_TTS_PPNOTFOUND	Specified preprocessor could not be found
NUAN_E_TTS_USERSTOP	Text processing stopped at user request
NUAN_E_TTS_VOICENOTFOUND	Specified voice could not be found
NUAN_E_UNRELEASEDMODULES	Some modules were not released yet
NUAN_E_VERSION	Struct has unsupported version number
NUAN_E_WRONG_BUFFER_SIZE	Specified buffer size incorrect
NUAN_E_WRONG_STATE	Inappropriate command

Chapter IV

Notification messages

This section describes all messages sent to the application by the TTS system.

VE_MSG_BEGINPROCESS

This notification message is sent to the application (output device) when the TTS system starts generating PCM data.

Parameter	Description
eMessage	VE_MSG_BEGINPROCESS
uParam	Reserved for future use
pParam	Reserved for future use.

VE_MSG_ENDPROCESS

This notification message is sent to the application (output device) when the TTS system ends generating PCM data and there is no more text input to process.

Parameter	Description
eMessage	VE_MSG_ENDPROCESS
uParam	Reserved for future use
pParam	Reserved for future use

VE_MSG_OUTBUFDONE

This notification message is sent to the application (output device) when the TTS system generates a PCM data and/or marker buffer. This message is only issued when **ve_ttsProcessText2Speech()** has been called. See the structure VE_OUTDATA for more details.

Parameter	Description
eMessage	VE_MSG_OUTBUFDONE
uParam	Flag to indicate the beginning of a text unit
pParam	Pointer to the structure VE_OUTDATA

uParam (1Value)	
0x0001	New text unit
0x0002	Middle of a text unit
0xFFFF	End of generating PCM data

The application uses the pointer to VE_OUTDATA in order to get the audio and the marker buffers.

VE_MSG_OUTBUFREQ

This notification message is sent to the application (output device) when the TTS system requires data buffers in order to generate a PCM data buffer and/or marker buffer. This message is only issued when **ve_ttsProcessText2Speech()** has been called.

See the structure VE_OUTDATA for more details.

Parameter	Description
eMessage	VE_MSG_OUTBUFREQ
uParam	Reserved for future use
pParam	Pointer to the structure VE_OUTDATA

The application has to allocate the memory for the output buffers.

VE_MSG_PAUSE

This notification message is sent to the application when the TTS system received a request to pause **ve_ttsPause()**. It is up to the application to actually halt the PCM output stream until a VE_MSG_RESUME is received.

Parameter	Description
eMessage	VE_MSG_PAUSE
uParam	Reserved for future use
pParam	Reserved for future use

VE_MSG_RESUME

This notification message is sent to the application when the TTS system received a request to resume synthesis from **ve_ttsResume()**. It is up to the application to again enable the PCM output stream.

Parameter	Description
eMessage	VE_MSG_RESUME
uParam	Reserved for future use
pParam	Reserved for future use

See also VE_MSG_PAUSE

VE_MSG_STOP

This notification message is sent to the application when the TTS system receives a request to stop processing (**ve_ttsStop()** has been called).

Parameter	**Description**
eMessage	VE_MSG_STOP
uParam	Reserved for future use
pParam	Reserved for future use

This notification message can be followed by additional notification messages such as VE_MSG_OUTBUFDONE. The TTS engine has only stopped completely when VE_MSG_ENDPROCESS is sent and **ve_ttsProcessText2Speech( )** returns.

Vocalizer Expressive 1.4

Chapter V

SAPI 5.1 Compliance

User's Guide and
Programmer's Reference
Revision 1

Chapter V

SAPI5 Compliance

API Support

This section lists which Microsoft Text-To-Speech API v5.1 functions (Text-to-Speech engine Interface) are supported by Vocalizer Expressive. For more details on each of these functions, see the chapters on Text-to-speech Engine Interface in the Microsoft Speech SDK v5.1 Reference.

Text-to-Speech engine Interface

Interface	Function Name	Availability
ISpTTSEngine	Speak	Supported
	GetOutputFormat	Supported
ISpTTSEngineSite	ISpEventSink	Supported
	GetActions	Supported
	Write	Supported
	GetRate	Supported
	GetVolume	Supported
	GetSkipInfo	Supported
	CompleteSkip	Not Supported

Chapter V

Text-to-speech Interface

With the exception of IsUISupported and DisplayUI, the Microsoft SAPI5 layer supports all functions of the Nuance Vocalizer Expressive interface.

Interface	Function Name	Availability
IspVoice	SetOutput	Supported
	GetOutputObjectToken	Supported
	GetOutputStream	Supported
	Pause	Supported
	Resume	Supported
	SetVoice	Supported
	GetVoice	Supported
	Speak	Supported
	SpeakStream	Supported
	GetStatus	Supported
	Skip	Not Supported
	SetPriority	Supported
	GetPriority	Supported
	SetAlertBoundary	Supported
	GetAlertBoundary	Supported
	SetRate	Supported
	GetRate	Supported
	SetVolume	Supported
	GetVolume	Supported
	WaitUntilDone	Supported
	SetSyncSpeakTimeout	Supported
	GetSyncSpeakTimeout	Supported
	SpeakCompleteEvent	Supported
	IsUISupported	Not Supported
	DisplayUI	Not Supported

Chapter V

SAPI5 Interface

In this section you find an alphabetical list of member functions of the SAPI5 text-to-speech interface (IspVoice).
For a description of each member function, see the chapter on Text-to-speech Interfaces (ISpVoice), in the Microsoft Speech SDK v5.1 Reference.

ISpVoice Interface

This interface is the only interface for the application to access the Text-To-Speech engine. The ISpVoice interface enables an application to perform text synthesis operations. Applications can speak text strings and text files, or play audio files through this interface. All of these can be done synchronously or asynchronously.
Applications can choose a specific TTS voice using ISpVoice::SetVoice. The state of the voice (for example, rate, pitch, and volume), can be modified using SAPI XML tags that are embedded into the spoken text. Some attributes, like rate and volume, can be changed in real time using ISpVoice::SetRate and ISpVoice::SetVolume. Voices can be set to different priorities using ISpVoice:: SetPriority.
ISpVoice inherits from the ISpEventSource interface. An ISpVoice object forwards events back to the application when the corresponding audio data has been rendered to the output device.

ISpVoice::ISpEventSource

No engine specific remarks.

ISpVoice::SetOutput

Vocalizer Expressive supports only 22 kHz in this product. If the application chooses other frequencies, then the Microsoft SAPI5 layer will use conversion software installed in the PC, which might cause speech quality degradation.

ISpVoice::GetOutputObjectToken

See ISpVoice::SetOutput.

ISpVoice::GetOutputStream

No engine specific remarks.

ISpVoice::Pause

No engine specific remarks.

ISpVoice::Resume

No engine specific remarks.

ISpVoice::SetVoice

No engine specific remarks.

ISpVoice::GetVoice

No engine specific remarks.

ISpVoice::Speak

No engine specific remarks.

ISpVoice::SpeakStream

No engine specific remarks.

ISpVoice::GetStatus

No engine specific remarks.

ISpVoice::Skip

This member function is not supported by Vocalizer Expressive.

ISpVoice::SetPriority

No engine specific remarks.

ISpVoice::GetPriority

No engine specific remarks.

ISpVoice::SetAlertBoundary

No engine specific remarks.

ISpVoice::GetAlertBoundary

No engine specific remarks.

ISpVoice::SetRate

No engine specific remarks.

ISpVoice::GetRate

No engine specific remarks.

ISpVoice::SetVolume

The default volume of Vocalizer Expressive voices is 90 instead of 100.

ISpVoice::GetVolume

The default volume of Vocalizer Expressive voices is 90 instead of 100.

ISpVoice::WaitUntilDone

No engine specific remarks.

ISpVoice::SetSyncSpeakTimeout

No engine specific remarks.

ISpVoice::GetSyncSpeakTimeout

No engine specific remarks.

ISpVoice::SpeakCompleteEvent

No engine specific remarks.

ISpVoice::IsUISupported

This member function is not supported by Vocalizer Expressive.

ISpVoice::DisplayUI

This member function is not supported by Vocalizer Expressive.

Chapter V

SAPI5 XML Tags

In this section you find an alphabetical list of the text-to-speech XML tags that are supported by Microsoft SAPI5. XML tags can be embedded in the input text to change the text-to-speech output. For each XML tag, you will find the following information:

Description	Gives a description of the XML tag
Syntax	Displays the syntax of the XML tag
Comments	Gives remarks that are specific to Vocalizer Expressive's support of the XML tag
Example	Shows how to use the XML tag

Please see the "Microsoft Speech SDK, V5.1" reference, chapter "Text-to-Speech Interface", for more details on the use and syntax of XML tags, as well as on each XML tag separately.

NOTE

1. Only correctly specified XML tags are converted to internally embedded commands. Incorrectly specified control tags are treated as white spaces.

This is an overview of the text-to-speech control tags and their support in Vocalizer Expressive.

Control tag	Availability
<Bookmark>	Supported
<Context>	Partially supported
<Emph>	Not Supported
<Lang>	Supported
<Partofsp>	Not Supported
<Pitch>	Not supported
<Pron>	Supported
<Rate>	Supported
<Silence>	Supported
<Spell>	Supported
<Voice>	Supported
<Volume>	Supported

Bookmark

Description

This XML tag indicates a bookmark in the text.

Syntax

<bookmark mark=string/>

Comments

Vocalizer Expressive supports this control tag.

Example

This sentence contains a
<bookmark mark="bookmark_one"/> bookmark.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Chapter V

Context

Description

This XML tag sets the context for the text that follows, determining how specific strings should be spoken.

Syntax

<Context ID=string> Input Text </Context>

Comments

Vocalizer Expressive only partially supports this control tag.

- The following context types are not supported:

  \context ID="date_mdy"\
  \context ID="date_dmy"\
  \context ID="date_ymd"\
  \context ID="date_ym"\
  \context ID="date_my"\
  \context ID="date_dm"\
  \context ID="date_md"\
  \context ID="date_year"\
  \context ID="time_timeofday"\
  \context ID="time_hms"\
  \context ID="time_hm"\
  \context ID="time_ms"\
  \context ID="number_decimal"\
  \context ID="currency"\

- Some languages do not support this XML tag. See the release note for language specific limitations.

Example

Today is <context ID="date_mdy">12/22/99</Context>.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Chapter V

Emph

Description

This XML tag emphasizes the next sentence to be spoken.

Syntax

<Emph> Input text </Emph>

Comments

Vocalizer Expressive does not support this control tag.

Example

<emph>John and Peter are coming tomorrow</emph>.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Chapter V

Lang

Description

This XML tag indicates a language change in the text. This tag is handled by the Microsoft SAPI5 Layer.

Syntax

<Lang langid=string> Input text </Lang>

Comments

Vocalizer Expressive supports this control tag.

Example

<lang langid="409"> A U.S. English voice should speak this sentence. </lang>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-speech Interface” and “XML TTS Tutorial”.

Chapter V

Partofsp

Description

This XML tag indicates the part-of-speech of the next word. This tag is effective only when the word is in the Lexicon and has the same part-of-speech setting as in the Lexicon.

Syntax

<Partofsp Part=string> word </Partofsp>

Comments

Vocalizer Expressive does not support this control tag.

Example

<Partofsp Part="noun"> A </Partofsp> is the first letter of the alphabet.

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-speech Interface” and “XML TTS Tutorial”.

Pitch

Description

This XML tag is used to control the pitch of a voice.

Syntax

<Pitch Absmiddle=string> Input Text </Pitch>

Comments

Vocalizer Expressive does not support this tag.

Example

<pitch absmiddle="5">This is a test.</pitch>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Chapter V

Pron

Description

The Pron tag inserts a specified pronunciation. The voice will process the sequence of phonemes exactly as they are specified. This tag can be empty, or it can have content. If it does have content, it will be interpreted as providing the pronunciation for the enclosed text. That is, the enclosed text will not be processed as it normally would be.

The Pron tag has one attribute, Sym, whose value is a string of white space separated phonemes.

Syntax

<pron sym=phonetic string> or

<pron sym=phonetic string>Input text</pron>

Comments

Vocalizer Expressive supports this control tag.

Example

<pron sym="h eh 1 l ow & w er 1 l d"> hello world </pron>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-speech Interface” and “XML TTS Tutorial”.

Chapter V

Rate

Description

The Rate tag controls the rate of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.
The Rate tag has two attributes, Speed and AbsSpeed, one of which must be present. The value of both of these attributes should be an integer between negative ten and ten. Values outside this range may be truncated by the engine (but are not truncated by SAPI). The AbsSpeed attribute controls the absolute rate of the voice, so a value of ten always corresponds to a value of ten, a value of five always corresponds to a value of five.

Syntax

<rate absspeed=number>Input text</rate>

or

<rate speed=number>Input text</rate>

Comments

Vocalizer Expressive supports this control tag.

Example

<rate absspeed=”5”>This is a sentence.</rate>

or

<rate speed=”5”>This is a faster sentence. </rate>

<rate speed=”-5”>This is a slower sentence. </rate>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-Speech Interface” and “XML TTS Tutorial”.

Chapter V

Silence

Description

The Silence tag inserts a specified number of milliseconds of silence into the output audio stream. This tag must be empty, and must have one attribute, Msec.

Syntax

<silence msec=number>Input text

Comments

Vocalizer Expressive supports this control tag.

Example

<silence msec="500">This is a sentence.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Chapter V

Spell

Description

The Spell tag forces the voice to spell out all text, rather than using its default word and sentence breaking rules, normalization rules, and so forth. All characters should be expanded to corresponding words (including punctuation, numbers, and so forth). The Spell tag cannot be empty.

Syntax

<spell>Input text</spell>

Comments

Vocalizer Expressive supports this control tag.

Example

<spell>UN</spell>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-speech Interface” and “XML TTS Tutorial”.

Chapter V

Voice

Description

The Voice tag selects a voice based on its attributes, Age, Gender, Language, Name, Vendor, and VendorPreferred. The tag can be empty, in which case it changes the voice for all subsequent text, or it can have content, in which case it only changes the voice for that content.

The Voice tag has two attributes: Required and Optional. These correspond exactly to the required and optional attributes parameters: ISpObjectTokenCategory EnumerateTokens and SpFindBestToken. The selected voice follows exactly the same rules as the latter of these two functions. That is, all the required attributes are present, and more optional attributes are present than with the other installed voices (if several voices have equal numbers of optional attributes one is selected at random).

For more details, see Object Tokens and Registry Settings in the “Microsoft Speech API V5.1”.

In addition, the attributes of the current voice are always added as optional attributes when the Voice tag is used. This means that a voice that is more similar to the current voice will be selected over one that is less similar.
If no voice is found that matches all of the required attributes, no voice change will occur.

Syntax

<voice required=type of info.=info.>Input text</voice>

or

<voice optional=type of info.=info.>Input text</voice>

Comments

Vocalizer Expressive supports this control tag.

Example

<voice required="Gender=Female;Age!=Child">
A female non-child should speak this sentence, if one exists.
</voice> <voice required="Age=Teen">
A teen should speak this sentence - if a female, non-child teen is present, she will be selected over a male teen, for example. </voice>

Chapter V

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-to-speech Interface" and "XML TTS Tutorial".

Volume

Description

The Volume tag controls the volume of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.
The Volume tag has one required attribute: Level. The value of this attribute should be an integer between zero and one hundred. Values outside this range will be truncated.

Syntax

<volume level=number>Input text</volume>

Comments

Vocalizer Expressive supports this control tag.

The default volume of Vocalizer Expressive voices is 90 instead of 100.

Example

<volume level=”50”>This is a sentence .</volume>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-to-speech Interface” and “XML TTS Tutorial”.

Vocalizer Expressive 1.4

Appendix 1

Copyright and licensing of third-party software

User's Guide and
Programmer's Reference
Revision 1

Appendix I

Copyright and licensing of third-party software

The Nuance Vocalizer Expressive software relies on particular open source software packages. The copyright and licensing information for these packages are included in this section.

Kazlib

Kaz Free Software License

Copyright (C) 1999 Kaz Kylheku <kaz@ashi.footprints.net>

Free Software License:

All rights are reserved by the author, with the following exceptions:

Permission is granted to freely reproduce and distribute this software, possibly in exchange for a fee, provided that this copyright notice appears intact. Permission is also granted to adapt this software to produce derivative works, as long as the modified versions carry this copyright notice and additional notices stating that the work has been modified.

This source code may be translated into executable form and incorporated into proprietary software; there is no requirement for such software to contain a copyright notice related to this source.

libfixmath

The MIT License (MIT)

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Appendix I

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PCRE

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Release 5 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

Written by: Philip Hazel <ph10@cam.ac.uk>

University of Cambridge Computing Service,
Cambridge, England. Phone: +44 1223 334714.

Copyright (c) 1997-2004 University of Cambridge
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Cambridge nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Appendix I

RSA MD5

wapiti

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

zlib

zlib/libpng License

zlib.h -- interface of the 'zlib' general purpose compression library version 1.2.6, January 29th, 2012

Copyright (C) 1995-2012 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
Mark Adler

OPUS audio codec

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.