Cerence TTS Embedded 19.12

User's Guide And Programmer's Reference

Cerence, Inc.

Dec 12, 2019

CONTENTS

CHAPTER

ONE

GETTING STARTED

Introduction

Thank you for using Cerence TTS Embedded, 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 Cerence's state of the art automotive Text-To-Speech technology.

Installing the SDK

The Cerence TTS Embedded SDK packages can be downloaded from Cerence support website, http://distro.cerence.com/.

First make sure to remove the voices and the engine of any previous version of the Cerence TTS Embedded SDK. Then install the Cerence TTS Embedded 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.

It's important that you download at least one voice package, extract the voice in a directory, and use this as a data location. You learn more about this in section **Flexible deployment configuration**. If you install the engine and voice under a single installation directory, it 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, Cerence TTS Embedded 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 Cerence TTS Embedded. Error code 0x80000012".

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.

Cerence TTS Embedded 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.

Cerence TTS Embedded 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 Cerence.

Functional components

The main functional components of the Cerence TTS Embedded 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.

Features

With the Cerence TTS Embedded 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

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 1 - Getting Started introduces you to Text-To-Speech in general.

Chapter 2 - Working with the Text-To-Speech System consists of a Text-To-Speech primer. It contains definitions of concepts and characteristics of Cerence TTS. It also explains about application development.

Chapter 3 - Text-To-Speech System Reference provides operational instructions for Cerence TTS Embedded. It reviews the functionality of the system, and describes the way in which the user can customize the pronunciation of input texts.

Chapter 4 - 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).

Chapter 5 - SAPI5.1 Compliance provides instructions on how to use SAPI5.

Appendix A: Cerence TTS Languages and Language Codes gives a list of all languages and the associated language codes

Appendix B - Cerence TTS Voices gives a list of supported voices and operating points.

Appendix C - Copyright and licensing of third-party software gives the copyright and licensing information of third-party packages used by Cerence TTS Embedded.

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`.

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.

CHAPTER

TWO

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.

Cerence TTS Embedded 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.

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.

Cerence TTS uses one of a set of proprietary speech synthesizers to create the speech output.

Voice operating points

Cerence TTS 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 Cerence TTS Embedded can synthesize one and the same voice in different ways. We say that Cerence TTS Embedded uses a particular voice operating point to synthesize speech for the voice. The following voice operating points are offered:

- 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 Cerence TTS Embedded SDK voice packages.
- the size of the speechbase (in number of speech units): embedded-pro < embedded-high (where the order represents overall voice quality), and
- the type of encoding of the speech units: embedded-pro ~ embedded-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.

Cerence 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 a special pronunciation for a particular word (e.g. an abbreviation) or a sequence of several worlds (e.g. an artist name). They can contain orthographic as well as phonetic information, and 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 Cerence proprietary 3-letter language code and this code is used as a part of filename of a user dictionary and other purposes. E.g. the language code for ”American English” is `ENU`.

A full list of languages and language codes can be found in *Appendix A: Cerence TTS Languages and Language Codes*

Supplying external services

Cerence TTS Embedded 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 Cerence TTS Embedded allocate and free memory blocks. It is a required service.
- the Critical Sections service:
  allows Cerence TTS Embedded 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 Cerence TTS Embedded with the language and voice-specific data. The Data Streams service is required, the Data Mappings service is optional.
- the User Log service:
  lets Cerence TTS Embedded 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 Clock service:
  allows Cerence TTS Embedded to retrieve timing information in the form of clock readings. The user supplies such clock values by suitably wrapping the native functions of the underlying operating system that retrieve time information. This is an optional service.
- the Output Delivery service:
  lets Cerence TTS Embedded 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 Cerence TTS Embedded.

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 Cerence TTS Embedded with a correct implementation that is fast enough to let Cerence TTS Embedded 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

Cerence TTS Embedded 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.

Cerence TTS Embedded 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 Cerence TTS Embedded component that needs a data component, queries a data access service for it by name.

Reference deployment configuration

In the Cerence TTS Embedded 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 Cerence TTS Embedded 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.

Flexible deployment configuration

The straightforward approach to deploy is to install the Cerence TTS Embedded engine and voice packs under a single root directory, e.g.

```
root_ve
+---common
+---doc
+---inc
+---languages
|   +---common
|   +---enu
|   +---frc
|   \---spm
+---lib
+---sample
\---tools
```

However the reference implementation of the data access services supports several locations to discover voice packs. This allows different ways to deploy as illustrated by these two sample use cases.

You can create a separate location for code and data, and share the voice packs, e.g.

```
root_ve_code
+---win32
|   +---common
|   +---doc
|   +---inc
|   +---languages
|       \---common
|   +---lib
|   +---sample
|   \---tools
\---win64
    +---common
    +---doc
    +---inc
```

(continues on next page)

(continued from previous page)

```
    +---languages
        \---common
    +---lib
    +---sample
    \---tools
```

```
root_ve_data
+---doc
    \---languages
+---enu
+---frc
\---spm
```

You can create different data locations, e.g.

```
root_ve
+---common
+---doc
+---inc
+---languages
|   \---common
+---lib
+---sample
\---tools
```

```
root_ve_data
+---proj-x
|   +---doc
|   \---languages
|       \---enu
\---proj-y
    +---doc
    \---languages
        +---frc
        \---spm
```

If you want to work with several data locations, the client code is to configure the data access service with the list of data locations. This is illustrated in the sample program read_file:

- The test driver accepts one or more command line options (-I dir)+.
- It passes these as `stResources.u16NbrOfDataInstall` of data locations in `stResources.apDataInstall[]` to `vplatform_GetInterfaces()`.

If you work with more than one data location, chances are that you end up with duplicate voice packs over these locations. You'll want to configure the data access service to detect such duplicates and error on it. To that purpose the client code is to set the flag `F_ERROR_CHECK` on `stResources.bFlags`., as illustrated in the sample program read_file.

By default, the reference implementation of the data access service only tracks one occurrence, but it's not defined which one.

Types of data access services

Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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, event and error messages. It is optional, but when supplied, it is added as a log subscriber to the log system of Cerence TTS Embedded, 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 how to 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.

Clock service

The Clock service defines an interface for providing Cerence TTS Embedded with clock information from the operating system, including the real time clock, and, if the operating system supports retrieving such information, the time spent by the current thread in user and/or system/kernel mode.

The Clock service is optional, but if it is not supplied certain features of Cerence TTS Embedded that depend on the knowledge of the clock information might be unavailable, for example the forced early emission of output audio data when the system is approaching a buffer underrun condition.

The reference implementation stores the reference time when the Clock service is initialized, by calling the suitable native function provided by the operating system. A call to the function *pfGetRelativeTime()* similarily retrieves the current time from the operating system, and returns its difference with respect to the stored reference time, expressed in milliseconds.

Multithreading service

The Multithreading service provides an interface for using threads inside the TTS engine. It is optional, and when supplied for some voice operating points the TTS engine can use this service to parallellise processing to reduce the latency.

The Multihreading service provides functions to create, close, start and join a thread and functions the retrieve the ID of the calling thread and to put the calling thread into a sleep state.

When this service is specified, also the Semaphore service should be implemented in order to provide synchronization between threads.

Semaphore service

The Semaphore service defines an interface for providing Cerence TTS Embedded with semaphore functionality for synchronization. It is optional but it should be specified if the Multithreading service is specified.

The Semaphore service provides functions to create, destroy, acquire and release semaphores.

Preparing a text for Text-To-Speech

Introduction

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

Cerence TTS Embedded 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 Cerence TTS Embedded expects the input text to be encoded in platform endian UTF-16. Cerence TTS Embedded does not require a specific Unicode version per language.

Cerence TTS Embedded 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.

Cerence TTS Embedded 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
- Changing the timbre of the speaker
- 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 ActivePrompt database
- Changing the speaking style

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 Cerence TTS Embedded 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 and embedded-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 timbre of the speaker

The control sequence `<ESC>\timbre=<level>\` sets the timbre of the speaker to the specified value, where *<level>* is between 50 and 200, and 100 is the default. The control sequence changes the perceived age or gender of the voice. This is achieved through frequency warping, which corresponds to a change in vocal tract length in modeled speech. Pitch is often changed together with timbre. Increasing timbre results in a younger sounding voice, whereas lowering timbre results in an older sounding voice. A female voice with a low timbre and pitch can be perceived as male, whereas a male voice with a high timbre and pitch starts sounding female. Intermediate settings result in the perception of a genderless voice.

Child voices can be obtained by increasing the timbre of a female voice. Cartoon voices can be obtained by increasing timbre of a male voice. Timbre and pitch settings between 80 and 120 result in relatively high quality modification depending on the voice. For lower and higher values some distortion may be audible.

Example:

```
I can <ESC>\timbre=150\ be like a child  <ESC>\timbre=75\ or like I was old.
```

Note also that the timbre and pitch effects can be combined to obtain even more peculiar ”personas”. For instance using combined timbre and pitch parameters with values between 80 and 90 on a female voice it is possible to realize a sort of genderless persona.

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 Cerence TTS Embedded 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 in L&H+ or NT-SAMPA can be extended as:

```
<ESC>\toi=<lhp \| nts>:"<orth_text>"\<phon_text>
```

This defines *<orth_text>* as the orthographic counterpart of the phonetic fragment *<phon_text>*. Cerence TTS Embedded 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\
```

Note that Cerence TTS Embedded does not support such an orthographic counterpart for Pinyin text or diacritized text.

Note that for certain voices and languages the orthographic counterpart may be realized differently than when presented to Cerence TTS Embedded in isolation.

It is possible to provide Cerence TTS Embedded with the Pinyin text for an orthographic character in Chinese input. Use the control sequence `<ESC>\tagpyt=<pinyin>` to define *<pinyin>* as the Pinyin text for the following Chinese character.

Example:

```
" 基金大\ <ESC>\tagpyt=sha4\厦" is read as "ji1.jin1.da4.sha4".
```

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 4 - 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 Cerence TTS Embedded and the audio that the user supplies through a Recorded ActivePrompt 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. Cerence TTS Embedded only supports inserting WAV format audio files that contain linear 16-bit PCM samples. If the recording's sampling rate does not match the current voice, Cerence TTS Embedded resamples it before inserting it in the speech output.

The control sequence accepts an optional alternate text as

<ESC>\audio="<path>":"<alternate_text>"\

This defines *<alternate_text>* as the fallback text for the audio recording at *<path>*. Cerence TTS Embedded reads *<alternate_text>* in case the digital audio recording is unavailable or incompatible.

Note that Cerence TTS Embedded extracts *<alternate_text>* from the control sequence without the surrounding double quotes. If *<alternate_text>* itself contains the double quote character, it is to be escaped as "\" ".

For example:

```
One two <ESC>\audio="c:\recordings\three.wav":"oops"\ four
```

The control sequence inserts the fallback text *oops* in case the *three.wav* does not meet the specific conditions mentioned above. The impact of changes in rate, volume and pitch on the alternate text are audibly the same as that on normal input text.

Inserting a bookmark

The control sequence <ESC>\mrk=<name>\ marks the position where it appears in the input text with the bookmark string <name>, and has Cerence TTS Embedded 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 4 - Text-To-Speech Function Reference*.

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

Some examples:

```
This bookmark <ESC>\mrk=ref_1\ marks a reference point.
Another <ESC>\mrk=ref-2\ 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.

Changing the speaking style

This control sequence tells Cerence TTS Embedded to have the current voice read in a given speaking style like lively, neutral, conversational, formal, didactic or apologetic, instead of the default style. The style specified as value for the style parameter should be supported by the voice, otherwise there would be no change in the speaking style.

For example:

```
<ESC>\style=lively\This text would be read in lively style, in case
it is supported by the voice.
```

To reset the speaking style to default, use default as the value for the style parameter.

For example:

```
<ESC>\style=default\This text would be read in default style of the
voice.
```

You learn about the style names that are supported by a voice operating point in *Appendix B - Cerence TTS Voices*. In case that the current voice operating point does not support a style `name`, it will silently ignore the style change by `<ESC>\style=name\` and keep the current style.

Entering phonetic input

Cerence TTS Embedded 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. Cerence TTS Embedded supports 2 phonetic alphabets, both of which can conveniently be entered from a keyboard:

- L&H+ is a Cerence specific alphabet. In the **Language and 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 Cerence TTS Embedded 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 Cerence TTS Designer and you save it as a binary format. Then you load the binary user dictionary on Cerence TTS Embedded.

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 Cerence TTS Embedded, e.g.: ENU for "American English". See *Appendix A: Cerence TTS Languages and Language Codes* 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)+
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
```

(continues on next page)

(continued from previous page)

```
    ]
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.'2mI.n$.'stR+e&I.S$n#
"P!nk & Indigo Girls" //'pInKk_@nd_'In.dI.go&U_'gEOR+lz
"R.O.O.T.S. (Deluxe Version)" //'R+uts_d$.'l^ks_'vEOR+.S$n

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_SZ_STRING

[Data]
IT          "Information Technology"
DLL         "Dynamic Link Library"
A-level     "advanced level"
Afr         africa
Acc         account
TEL         telephone
Anon        anonymous
AP          "associated press"
```

Lookup in user dictionaries

Longest match

Cerence TTS Embedded consults the user dictionary for multi-word fragments and for individual words in the input text. These are the basic rules for the lookup:

- It looks up fragments from left to right in the input text.
- It takes the longer match of a multi-word string before a smaller match and before the match of an individual word.

- It looks up in user dictionaries going from the most recently loaded user dictionary to the firstly loaded one.

Example: Lookup left to right

If you load a user dictionary with these entries:

```
"Buena Vista" "(Buena+Vista)"
"Vista Social Club" "(Vista+Social+Club)"
```

And feed this input:

```
Let's meet in the Buena Vista Social Club at 10 PM.
```

Cerence TTS picks up these matches:

```
Let's meet in the (Buena+Vista) Social Club at 10 PM.
```

Example: Longest match

If you load a user dictionary with these entries:

```
"Buena Vista" "(Buena+Vista)"
"Buena Vista Social Club" "(Buena+Vista+Social+Club)"
```

And feed this input:

```
Let's meet in the Buena Vista Social Club at 10 PM.
```

Cerence TTS picks up these matches:

```
Let's meet in the (Buena+Vista+Social+Club) at 10 PM.
```

Tagged multi-word fragment

A multi-word fragment may be tagged by the control sequence `<ESC>\mw\` . Cerence TTS Embedded takes the match of a tagged multi-word string before the match of an untagged multi-word string. In this way `<ESC>\mw\` may be used to overrule the longest-match approach, and have a smaller entry looked up:

- If a tagged multi-word string is not matching with an entry, then the ESC>mwis ignored and the regular longest-match takes place on the input.
- If a tagged multi-word string is matching with an entry, the opening ESC>mwmarks the end for the longest-match in the left context.

Example: Tagged multi-word string is not matched

If you load a user dictionary with these entries:

```
"Buena Vista" "(Buena+Vista)"
"Buena Vista Social Club" "(Buena+Vista+Social+Club)"
```

And feed this input:

```
Let's meet in the Buena Vista <ESC>\mw\ Social Club <ESC>\mw\ at 10 PM.
```

Cerence TTS picks up these matches:

```
Let's meet in the (Buena+Vista+Social+Club) at 10 PM.
```

Example: Tagged multi-word string is matched

If you load a user dictionary with these entries:

```
"Buena Vista" "(Buena+Vista)"
"Social Club" "(Social+Club)"
"Buena Vista Social Club" "(Buena+Vista+Social+Club)"
```

And feed this input:

```
Let's meet in the Buena Vista <ESC>\mw\ Social Club <ESC>\mw\ at 10 PM.
```

Cerence TTS picks up these matches:

```
Let's meet in the (Buena+Vista) (Social+Club) at 10 PM.
```

Fragments looked up

Cerence TTS Embedded can match a user dictionary entry against different forms of writing. The particular form may change according to its context where the entry text appears in the input. For instance, at the beginning of a sentence the entry text is proper cased in a lot of languages, and at the end of a sentence there are usually one or more punctuation marks appended.

That's why Cerence TTS Embedded extracts a number of forms from the fragment in the input and looks that form up in the user dictionary. It tries these candidates until the lookup returns a hit, or all are missed:

- First Cerence TTS Embedded looks up the fragment “as is”
- Then the fragment with leading and trailing quotes and brackets stripped
- Then with trailing dots stripped
- Then the fragment converted to lower case..

For a multi-word text fragment it matches a blank character between words with a set of blank characters. But it requires the fragment and a matching entry have the same number of blank characters between their words.

Loading user dictionaries

User dictionaries can be loaded for run-time use in two different ways. Cerence TTS Embedded 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 Cerence TTS Embedded 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\Cerence\Cerence TTS Embedded\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 in **Prompt templates** 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. **Prompt templates**).

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.

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 Cerence TTS Embedded built-in text normalization types and to delegate portions of their processing to other typed rulesets or to Cerence TTS Embedded 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=flightas 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".

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 Cerence TTS Embedded 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 *Appendix A: Cerence TTS Languages and Language Codes* 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 Cerence TTS Embedded 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, backslash (‘\’) 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

```
/<CRNC>/ --> "Cerence, Inc."
```

Rewrites “<CRNC>” into “Cerence, Inc.” .

```
/(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.

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>
```

Cerence TTS Embedded 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 Cerence TTS Embedded installation directory. The rulesets in the pipeline header are loaded prior to any loaded with `ve_ttsResourceLoad()`, so have the lowest precedence.

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 Cerence TTS Embedded through an API call like `ve_ttsProcessText2Speech()`.

A *template* describes a transformation of a prompt. It consists of an input string and an output string.

We call *slot* a variable part of the input string or of the prompt. Its contents may be referenced in the output string.

In the template description slots are marked by:

```
<slot type="TYPE">...</slot>.
```

In the prompt they are delimited by escape sequences:

```
<ESC>\slotbegin=TYPE\...<ESC>\slotend\.
```

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[1].

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 empty (aka open) slot in a template input string matches a slot in the prompt with any contents. A non-empty (aka instantiated) slot in a template input string only matches a slot in the prompt with literally identical contents.
Slots also need to match with respect to types: The default value for the type of a template slot is "*" . It matches any type of a prompt slot. Any other type value (including the empty string) must match the type of the prompt slot exactly. An unspecified prompt slot type is equivalent to the empty string.

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:

- '$n' is replaced by the contents of the n-th slot of the prompt. If there are less than n slots in the template's input string, the template set loading/compilation fails.
- '${n:newtype}' is replaced by the contents of the n-th slot of the prompt but with all occurrences of `<ESC>\tn=TYPE\` replaced by `<ESC>\tn=newtype\` where TYPE is the type of the slot. If there are less than n slots in the template's input string, the template set loading/compilation fails.

[1] Note that slot markup of not matching prompts will be removed automatically downstream.

- 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 valid slot number.

Prompt template set format

A prompt template set is represented by an XML file conforming to the following DTD:

```
<!DOCTYPE pts [
<!ELEMENT pts       (template*)>
<!ATTLIST pts
          format    CDATA #FIXED "Text Template 2.2"
          esc       CDATA #REQUIRED
          version   CDATA #REQUIRED>
<!ELEMENT template  (input,output)>
<!ELEMENT input     (#PCDATA|slot)*>
<!ELEMENT slot      (#PCDATA)>
<!ATTLIST slot
          type      CDATA #IMPLIED>
<!ELEMENT output    (#PCDATA)>
]>
```

The value of <pts esc=" ⋯" > defines a string to be interpreted as `<ESC>`. All occurrences of this string in the non-markup parts of the template descriptions are internally replaced by `<ESC>`. The only purpose of this is to provide a means to refer in the templates to `<ESC>` which otherwise is an illegal character in XML.

The version specified by <pts version=" ⋯" > is an arbitrary string that may serve to label the template set. It is preserved in the compiled template set (cf. *below*). Note that this is meant to refer to the contents of the template set; it does not refer to the version of the data format.

Example template set

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE template [
    <!ENTITY esc "\!" >
]>
<pts format="Text Template 2.2"
    esc="&esc;"
    version="initial">
    <template>
        <input>In <slot>3m</slot>, turn left into <slot/>.</input>
        <output>In &esc;\audio=c:/home/SampleWaves/3.wav\meters, turn left into $2.</output>
    </template>
    <template>
        <input>In <slot/>, turn left into <slot>Main Street</slot>.</input>
        <output>In $1, turn left into &esc;\audio=c:/home/SampleWaves/main_street.wav\.</
↪output>
    </template>
    <template>
        <input>In <slot/>, turn left into <slot/>.</input>
        <output>&esc;\prompt=nav::in\ $1, &esc;\prompt=nav::turn_left_into\ $2.</output>
    </template>
    <template>
        <input>You will arrive in <slot type="dist"/> at destination.</input>
        <output>You will arrive in ${1:distance} at destination.</output>
    </template>
    <template>
        <input>You will arrive in <slot/> at destination.</input>
```

(continues on next page)

(continued from previous page)

```
        <output>You will arrive in approximately $1 at destination.</output>
    </template>
    <template>
        <input>Hello. This is a very long help text.</input>
        <output>okay</output>
    </template>
    <template>
        <input>okay</input>
        <output>oops $$99</output>
    </template>
</pts>
```

Remarks

The entity "&esc;" has been defined in the example as a simple commodity. It is not part of the description format. Defining its value to match <pts esc=" …" > is a convenient way to make the template descriptions independent of the actual string used to substitute <ESC>.

Commented input output examples

Prompt	`In <ESC>\slotbegin\3m<ESC>\slotend\, turn left into <ESC>\slotbegin\Main Street<ESC>\slotend\.`
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.	

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

Prompt	`In <ESC>\slotbegin\200m<ESC>\slotend\, turn left into <ESC>\slotbegin\Narrow Road<ESC>\slotend\.`
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 iteratively the last template although it matches "okay".	

Prompt	`You will arrive in <ESC>\slotbegin=dist\<ESC>\tn=dist\ 50m <ESC>\tn=normal\<ESC>\slotend\ at destination.`
Output	`You will arrive in <ESC>\tn=distance\ 50m <ESC>\tn=normal\ at destination.`
Applying "type instantiated" template 4, note the mapping "dist" => "distance"	

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>
```

Cerence TTS Embedded 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 Cerence TTS Embedded installation directory.

Note: 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

Cerence TTS Embedded supports tuning Text-To-Speech synthesis through 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 Cerence TTS Designer product (a graphical TTS tuning environment) and are stored in an ActivePrompt database for run-time use. There are two types of ActivePrompts:

- A Recorded ActivePrompt brings the digital audio recording that Cerence TTS uses as-is to construct the speech output for a text fragment. The audio recordings of a set of Recorded ActivePrompts are usually stored in a compressed database (much smaller than individual audio files), or could optionally be stored as individual WAV files.
- A Tuned ActivePrompt doesn't store the actual audio for a text fragment, but rather defines the speech units that Cerence TTS uses to synthesize it in the desired way. The sequence of speech unit IDs is much smaller than the audio that will be produced.

  A regular Tuned ActivePrompt only defines the speech units, and these speech units are readily available in the voice data. A Tuned ActivePrompt with vocal add-on not only defines the speech units for a fragment, but also contains the voice data for those speech units as an extension to the regular voice data. In that way, a Tuned ActivePrompt database with vocal add-on is a self-contained tuning resource that can be used on different versions of a voice.

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 ActivePrompt 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>
```

Cerence TTS Embedded requests the Data Access external service for the data identified by `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 Cerence TTS Embedded installation directory. The ActivePrompt databases in the pipeline header are loaded prior to any loaded with `ve_ttsResourceLoad()`, and have lower precedence.

A Recorded ActivePrompt 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 ActivePrompt 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 plain Tuned ActivePrompt database there is only one data component which stores both the symbolic data and the list of speech units. A Tuned ActivePrompt database with vocal add-on also has that data component, and in addition 3 voice data components which extend the regular voice data. As for a Recorded ActivePrompt database the client needs to copy the symbolic 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 ActivePrompt 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. Such a region starts directly after `<ESC>\domain=<domain>\` and ends whenever `<ESC>\domain\` or `<ESC>\voice=<name>\` appears in the input. If you want to enable implicit matching of an ActivePrompt database on a further region, you have to mark its beginning again with control sequence `<ESC>\domain=<domain>\`.

An ActivePrompt 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 ActivePrompt 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 ActivePrompt 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

Cerence TTS Embedded 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.

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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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>\` Cerence TTS Embedded 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 Cerence TTS Embedded 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()`.

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 Cerence TTS Embedded 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, Cerence TTS Embedded 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, Cerence TTS Embedded 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 Cerence TTS Embedded 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.

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.

One or more invalid phoneme symbols

No orthographic counterpart

The voice drops the entire foreign phonetic text.

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).

The orthographic counterpart may not always be realized as if entered as plain orthographic input. The realization of the orthographic counterpart depends on a number of aspects that may impact that realization. For multilingual voices, Arabic voices and Asian voices there can be deviations. Under all circumstances the orthographic counterpart is only an approach to realize the best possible output for invalid phonetic input.

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 Cerence TTS Embedded synthesize the audio for the entire input text. Instead it lets you direct Cerence TTS Embedded 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 Cerence TTS Embedded analyze the input text and you collect a list of possible jump points. And then you tell Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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.

Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded synthesize from that particular location.

Traversing and control sequences

As Cerence TTS Embedded 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 Cerence TTS Embedded at the jump point. For instance consider that you have Cerence TTS Embedded 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.

Cerence TTS Embedded 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().

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 Cerence TTS Embedded, it delivers markers and TA info that reference into the rewritten text instead of into the input text.

That is why Cerence TTS Embedded 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.

SSML support

Speech Synthesizer Markup Language (SSML) is a markup language specification for voice browsers established by the World Wide Web Consortium (W3C). SSML provides a rich, XML-based markup language that assists synthetic speech generation in web and other applications. The essential role of the markup language is to standardize control of pronunciation, volume, rate, and other aspects of speech.

Cerence TTS Embedded offers a built-in preprocessor that supports most of the W3C specification http://www.w3.org/TR/2004/REC-speech-synthesis-20040907 "Speech Synthesis Markup Language Specification Version 1.0 – W3C Recommendation 7 September 2004".

Moreover Cerence TTS Embedded extends SSML with a few Cerence-specific elements/attributes, see below for details.

Support for the SSML 1.0 Recommendation September 2004

All elements/attributes in the September 2004 Recommendation are supported, regardless of their rating (MUST, REQUIRED, SHALL, SHOULD, RECOMMENDED, MAY, OPTIONAL), except for the following:

- The <voice> element is handled as follows:
  - The variant attribute is not supported.
  - The age attribute is supported. But to make this attribute useful, a set of voices with varying age over the same language and gender needs to be installed. This may require the use of custom voices.
- The <prosody> element is handled as follows:
  - Duration, range, and contour values are ignored.
- The <meta> element: The http-equiv attribute is not supported.
- The <say-as> element: The SSML specification does not standardize the list of <say-as> attribute values. See the section on the <say-as> element for information on Cerence TTS Embedded's support of these attributes.

XML document encodings

Cerence TTS Embedded can manage most part of XML document encodings, as far as the XML document contains an appropriate header declaring the document encoding such as:

```
<?xml version="1.0" encoding="utf-8"?>
```

The encoding attribute, if omitted, has "utf-8" as its default value.

Supported encodings:

"utf-8", "utf-16" (little endian), "iso-8859-1", "iso-8859-2", "iso-8859-5", "iso-8859-6", "iso-8859-7", "iso-8859-9", "iso-8859-15", "windows-central-european", "windows-1250", "windows-1250", "windows-latin-2", "windows-cyrillic", "windows-1251", "windows-western", "windows-1252", "windows-latin-1", "windows-greek", "windows-1253", "windows-turkish", "windows-1254", "windows-arabic", "windows-1256", "utf-16-be" (big endian), "western-european-dos", "cp-858"

As for any other XML document, the following characters: < (less than), > (greater than), & (ampersand), " (double quotation), and ' (apostrophe) are special characters and therefore cannot be used "as is" in the attribute's values nor in the body of a SSML document. Instead they require to be escaped as <, &, etc. according to the following table:

Name	Character	Unicode code point (decimal)	Standard	Description
quot	”	U+0022 (34)	XML 1.0	double quotation mark
amp	&	U+0026 (38)	XML 1.0	ampersand
apos	’	U+0027 (39)	XML 1.0	apostrophe (apostrophe-quote)
lt	<	U+003C (60)	XML 1.0	less-than sign
gt	>	U+003E (62)	XML 1.0	greater-than sign

Alternatively, it is possible to escape them as any other character using their unicode value, either in decimal or hexadecimal.

For instance the ampersand character can be escaped also as `&` (hex) or `&` (dec). For a more general description of XML and HTML entities see:

List of XML and HTML character entity references on Wikipedia

SSML validation

Cerence TTS Embedded does not use a fully validating parser for XML/SSML. It requires that the input text is well-formed XML. It does not provide validation of the SSML tags semantics via DTDs or schemas.

Should input text be not well-formed, an error would occur: depending on the degree of syntactic incorrectness, text may be read as if it were plain text, read only partially, not read at all.

When validation errors occur, Cerence TTS Embedded logs errors or warnings to the normal error log destinations (a file, a system log, and/or an application callback depending on the configuration settings), then returns a TTS_E_SSML_PARSE_ERROR error from the `ve_ttsProcessText2Speech` call. When this occurs, review the error logs for details about the SSML document errors.

The <speak> element

The <speak> element is the root element of every SSML document.

- xml:lang is a required attribute specifying the language of the root document.
- xml:base is an optional attribute specifying the Base URI of the root document.
- ssft-domaintype is an optional attribute specifying the domain to be used for the entire ssml document.
  This attribute is a Cerence extension of the SSML standard.
  The attribute value is the ActivePrompt domain name.
  The ssft-domaintype is equivalent to the native `<ESC>\domain=domain_name\` escape sequence.
- version attribute is a required attribute that indicates the version of the specification to be used for the document and must have the value “1.0”.

The xml:lang attribute value must be one of the language IETF codes. Documents of a language not supported by Cerence TTS Embedded are read using current voice's mother tongue. The syntax of a language string must be one of the following:

- a 2 character ISO639-1 language code (e.g. “fr”)
- a 2 character ISO639-1 language code followed by a 2 character ISO3166-1 country code, separated by a dash (e.g. “en-US”)

The <p>, <s> and <lang> elements

A <p> element represents a paragraph. An <s> element represents a sentence. xml:lang is an optional attribute on the <p> and <s> elements (for a description of xml:lang see <speak> element). The use of <p> and <s> elements is optional. Where text occurs without an enclosing <p> or <s> element Cerence TTS Embedded attempts to determine the structure using language-specific knowledge of the format of plain text. Although not strictly compliant with SSML 1.0, the element <lang> will be as well accepted: its syntax and its attributes are the same as <p> and <s>: the effect is a simple language change with no sentence nor paragraph boundary added. The ssft-domaintype is an optional attribute specifying the domain to be used for the sentence or the paragraph element. This attribute is a Cerence extension of the SSML standard. The attribute value is the ActivePrompt domain name. The ssft-domaintype is equivalent to the native `<ESC>\domain=domain_name\` escape sequence.

The <break> element

The <break> element is an empty element that controls the pausing or other prosodic boundaries between words. The supported attributes of this element are:

- strength: is an optional attribute having one of the following values: none, x-weak, weak, medium (default value), strong, or x-strong.
- time: is an optional attribute indicating the duration of a pause to be inserted in the output in seconds or milliseconds. E.g. 250ms, 3s.

The <phoneme> element

The <phoneme> element provides a phonemic/phonetic pronunciation for the contained text. The phoneme element may be empty or contain human-readable text that can be used for non-spoken rendering of the document.

- The ph attribute is a required attribute that specifies the phoneme string.
- The alphabet attribute is an optional attribute that specifies the phonemic/phonetic alphabet and can assume one the following values (that is the phonetic alphabets supported by Cerence TTS Embedded): nt-sampa (default) (aliases x-sampa, nts, x-navteq-sampa), lhp (alias: x-l&h+), pinyin (for Chinese only), diacritized (for Arabic only). Note that ipa is currently unsupported. Should the phonetic alphabet be unknown, the orthographic alternative text will be pronounced instead.

The <prosody> element

The <prosody> element permits control of the pitch, speaking rate, timbre of the speaker and volume of the speech output. The attributes, all optional, are:

- pitch: the baseline pitch for the contained text. Legal values are: a relative change (e.g. +20%) or x-low, low, medium, high, x-high, or default. Labels x-low through x-high represent a sequence of monotonically non-decreasing pitch levels. Absolute values in hz are not supported.
- rate: a change in the speaking rate for the contained text. Legal values are: a relative change or number (e.g. -10%, 0.5, etc.) or x-slow, slow, medium, fast, x-fast, or default. Labels x-slow through x-fast represent a sequence of monotonically non-decreasing speaking rates. When a number is used to specify a relative change it acts as a multiplier of the default rate. For example, a value of 1 means no change in speaking rate, a value of 2 means a speaking rate twice the default rate, and a value of 0.5 means a speaking rate of half the default rate. The default rate for a voice depends on the language and dialect and on the personality of the voice.
- volume: the volume for the contained text in the range 0.0 to 100.0 (higher values are louder and specifying a value of zero is equivalent to specifying silent). Legal values are: number, a relative change or silent, x-soft, soft, medium, loud, x-loud, or default. The volume scale is linear amplitude. The default is 80.0. Labels silent through x-loud represent a sequence of monotonically non-decreasing volume levels.
- timbre: the speaker timbre modulation for the contained text. This modifier turns the voice into a younger or older version of itself. Values must be in the interval [50,200] where values greater

than 100% make the voice sound younger; reversely, values lesser than 100% make the voice sound older. The default value is 100%, the inherent timbre of the voice.

The following attributes are currently not supported: contour, range, duration

Volume scale conversion

The SSML default value for the prosody attribute “volume” is 100 and the range is 0-100.

The SSML value is mapped to the native Cerence TTS Embedded value as if was set via the native `<ESC>\vol=x\` sequence (whose default value is 80, though). The mapping is one-to-one.

Some special “symbolic” values are admitted by SSML specification and they are mapped as follows:

Symbolic value	SSML volume value
silent	0
x-soft	26
soft	52
medium	80
loud	90
x-loud	100

Rate scale conversion

The SSML prosody attribute “rate” represents a percent increment (either positive or negative) from the default speaking rate. The final value is mapped to the appropriate native Cerence TTS Embedded `<ESC>\rate=value\` escape sequence (whose range is 50-400)

Some special “symbolic” values are admitted:

Symbolic value	Native Cerence TTS
x-slow	50
slow	75
medium	100
default	100
fast	150
x-fast	200

Pitch scale conversion

The SSML prosody attribute “pitch” represents a percent increment (either positive or negative) from the default pitch. The final value is mapped to the appropriate native Cerence TTS `<ESC>\pitch=value\` escape sequence (whose range is 50-200)

Some special “symbolic” values are admitted:

Symbolic value	Percentage increment
x-high	+60%
high	+35%
medium	0%
default	0%
low	-15%
x-low	-30%

Timbre scale conversion

The SSML prosody attribute "timbre" represents a percent increment (either positive or negative) from the default timbre of the speaker. The final value is mapped to the appropriate native Cerence TTS *<ESC>timbre=value* escape sequence (whose range is 50-200).

Some special "symbolic" values are admitted:

Symbolic value	**Percentage increment**
x-young	+35%
young	+20%
medium	0%
default	0%
old	-20%
x-old	-35%

The <emphasis> element

The <emphasis> element requests that the contained text be spoken with emphasis. The optional level attribute indicates the strength of emphasis to be applied. Defined values are strong, moderate, none and reduced. The default level is moderate. The values none, moderate, and strong are monotonically non-decreasing in strength.

The <style> element

The <style> element sets an alternative speaking style (e.g. lively, neutral, conversational, formal, didactic or apologetic) instead of the normal one (default). The only attribute supported is "name" (i.e. the name of the requested style as a string). Note that it is not guaranteed that a style is supported by all voices.

You learn about the style names that are supported by a voice operating point in *Appendix B - Cerence TTS Voices*. In case that the current voice operating point does not support a style `name`, it will silently ignore the style change by `<ESC>\style=name\` and keep the current style.

The <say-as> element

The <say-as> element allows the author to indicate information on the type of text construct contained within the element and to help specify the level of detail for rendering the contained text. The <say-as> element can only contain text to be rendered. The supported attributes are interpret-as (required) and format (optional, valid only when interpret-as=" date"). Possible values for interpret-as are described in the following table:

Value	Aliases	Description
normal		Default text normalization
spell		Spell out the input text that follows
address		Expand text as an address, including street names and numbers, zip codes, state names, etc.
zip		Expand text as a zip code
sms		Expand text as a sms message, reading web addresses, smileys, email addresses, etc.
distance		Expand text as a distance measurement
time		Expand text as a clock reading (hour, minutes, am, pm), a duration or a time range
currency		Expand text as a decimal currency including currency abbreviations
date		Expand text as a date. The date format may be optionally specified via format attribute, to supersede the language defaults, e.g. "dmy" or "mdy".
digits	code	Expand numbers or codes reading them digit by digit
number	cardinal	Expand cardinal/ comma formatted numbers up to 15 digits
ordinal		Expand number as ordinal
decimal	real, rational	Same as number but including comma/dot normalization.
phone	telephone	Expand text as a telephone number including country codes, prefixes, tel. word indicators, etc.

Note: SSML 1.0 only specifies the <say-as> element, its attributes, and their purpose. It does not enumerate the possible values for the attributes. That is why current implementation of <say-as> is limited to the native text normalization modes supported by Cerence TTS Embedded.

The <say-as format="slot"> in the context of Prompt Templates

The attribute format="slot" of the <say-as> element enables the markup of slots destined for use with Prompt Templates. The usual mapping to native text normalizer modes is done as described above[2], however the entire slot is wrapped in the native slot markup:

```
<ESC>\slotbegin=TYPE\...<ESC>\slotend\
```

The <voice> element

The <voice>element is a production element that requests a change in speaking voice. The supported attributes are the following (ordered by their weight to obtain the voice to match best):

- name: the optional voice name
- xml:lang : optional language specification attribute.
- gender: optional attribute indicating the preferred gender of the voice to speak the contained text. Enumerated values are: male, female, neutral.
- age: optional attribute indicating the preferred age in years (since birth) of the voice to speak the contained text. Acceptable values are: an integer, adult, child.

The attribute variant is currently not supported. Current active voice is preferred whenever its weight is the same of other voices.

[2] With the exception that "format" is obviously not available anymore for "date".

The <audio> element

The <audio> element supports the insertion of recorded audio files. The required attribute is src, which is the URI of a document with an appropriate MIME type. URIs may be absolute or relative to the base:uri specified in <speak> element. Audio files may be local ("file://", or absolute paths) or remote ("http://"). The only audio format supported is ".WAV", containing linear 16 bit PCM samples. If the recording's sampling rate does not match the current voice, Cerence TTS Embedded resamples it before inserting it in the speech output.

When SSML <audio> is used to insert an audio recording, and the fetch fails or the audio format is not supported, Cerence TTS Embedded proceeds with using the <audio> element's content as fallback as described in the SSML 1.0 Recommendation. Cerence TTS Embedded only supports insertion of text input as fallback, while nested elements are not accepted as a part of the fallback (as opposed to the SSML 1.0 Recommendation).

The <sub> element

The <sub> element is employed to indicate that the text in the "alias" attribute value replaces the contained text for pronunciation. This allows a document to contain both a spoken and written form. The required alias attribute specifies the string to be spoken instead of the enclosed string. The processor should apply text normalization to the alias value. The sub element can only contain text (no elements).

The <prompt> element

A <prompt id="xxx"> element is used to insert an ActivePrompt at a specific location in the text. SSML <prompt id="xxx"> maps directly to a Cerence TTS `<ESC>\prompt=xxx\` tag.

Native control sequences

Cerence TTS supports native control sequences within SSML documents. The default `<ESC>` character used to initiate a native control sequence, `\x1B`, although not allowed in XML documents as per the W3C XML 1.0 specification, is admitted by Cerence TTS Embedded XML.

However, beware that mixing native control sequences with SSML markup can lead to unexpected behavior, because Cerence TTS Embedded internally handles SSML by expanding it to native control sequences as well. The Cerence TTS Embedded SSML parser is "blind" to any native control sequences within the SSML document, so if a native control sequence is used that conflicts with the SSML parser expansions, unexpected behavior may occur. For example, if `<ESC>\voice\` is used to switch the voice, then SSML <voice> is used to change the voice again, at the end of the SSML <voice> element the voice is restored to the previous language as determined by the SSML parser state, not the voice selected by the native `<ESC>\voice\` control sequence. For this reason, Cerence recommends the following:

- Within SSML documents, use SSML markup whenever possible instead of native control sequences. (That is, only use native control sequences when there is no SSML equivalent.)
- When using native control sequences, try and switch the entire document to using native control sequences if there are any possibilities for conflicts (that is, if the native control sequence interacts with the SSML markup).
- Carefully test for unexpected interactions.

The <break> element

A `<break strength="xxx">` element is implemented as a pause of a certain duration, so it maps directly to a Cerence TTS `<ESC>\pause=x\` tag. The only exception is the SSML `<break strength="none">` element, which maps to a Cerence TTS `<ESC>\eos=0\` tag.

The table below specifies the mapping and the corresponding native markup.

Symbolic value	Duration in ms of the pause	Native markup value
x-weak	20	`<ESC>\pause=20\`
weak	100	`<ESC>\pause=100\`
Medium	500	`<ESC>\pause=500\`
strong	1000	`<ESC>\pause=1000\`
x-strong	1500	`<ESC>\pause=1500\`
none	0	`<ESC>\eos=0\`

When using both the time and the strength attributes, the time attribute gets precedence.

CHAPTER

THREE

TEXT-TO-SPEECH SYSTEM REFERENCE

Introduction

This chapter gives general information on how to install Cerence TTS Embedded and use the API in an application.

Multiple-language and multiple-voice support

Introduction

Cerence TTS Embedded contains two types of components:

- Language-independent code components (e.g. the Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded has a char only interface. Strings like language and voice names, which are passed between the client and the Cerence TTS Embedded 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 4 - 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 Cerence TTS Embedded 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 4 - Text-To-Speech Function Reference*.

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 TTS class.
- `ve_ttsOpen()` creates a TTS object on the class.
- `ve_ttsSetParamList()` sets the language and voice, and other control parameters such as volume, pitch, rate and timbre 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 4 - Text-To-Speech Function Reference*.

Multiple instances

Cerence TTS Embedded supports multiple open instances (TTS objects as created by `ve_ttsOpen()`) at a time.

Multi-threading

Cerence TTS Embedded 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 Cerence TTS Embedded.

Asynchronous API

The Cerence TTS Embedded API has only one asynchronous function:

- `ve_ttsStop()`

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

Broker header files

Introduction

Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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/ava/22/embedded-high/1.0.2/text/pcm
</BROKERSTRING>
```

This is the name of the processing pipeline for the embedded-high voice operating point of Ava.

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
<?xml version="1.0"?>
<NUANCE>
<VERSION>NUAN_1.0</VERSION>
<HEADER>
  <BROKERSTRING>pipeline/American English/ava/22/embedded-high/1.0.2/text/pcm</BROKERSTRING>
  <PRIORITY>5335</PRIORITY>
  <PARAMETERS>
    <language>American English</language>
    <langcode>ENU</langcode>
    <langietf>en-US</langietf>
    <langversion>8.0.7</langversion>
    <voice>Ava</voice>
    <voiceversion>1.0.0</voiceversion>
    <vopversion>1.0.2</vopversion>
    <gender>Female</gender>
    <age>Adult</age>
    <fecfg>cfg4</fecfg>
    <voiceml>no</voiceml>
    <mlset>enu,frc,spm</mlset>
    <extclccfg>frc+*=clc/frc/cfg3,spm+*=clc/spm/cfg3</extclccfg>
    <datapackagename>enu/ava/embedded-high/1-0-2</datapackagename>
    <fedataprefix></fedataprefix>
    <feextcfgdataprefix></feextcfgdataprefix>
    <fedatapackaging>clc</fedatapackaging>
    <fevoice>Ava</fevoice>
    <frequencyhz>22050</frequencyhz>
    <voicemodel>vop1_sch1f22</voicemodel>
    <voiceoperatingpoint>embedded-high</voiceoperatingpoint>
    <reduction>vop1</reduction>
    <coder>sch1f22</coder>
    <bitrate></bitrate>
    <overheadframes></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/tbm</COMPONENT>
    <COMPONENT>ttt/re</COMPONENT>
    <COMPONENT>pp/text_parser</COMPONENT>
```

(continues on next page)

(continued from previous page)

```
    <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>
  </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 Cerence TTS Embedded 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 Cerence.

The RESOURCES element defines data components that are used to execute the pipeline. Unless requested by Cerence, the 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

Cerence. 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 Cerence TTS Embedded:

```
<?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 Cerence TTS Embedded software you create the directory `./common/speech/ve` and copy the file `ve_logging.hdr` from the `./sample` directory there

CHAPTER

FOUR

TEXT-TO-SPEECH FUNCTION REFERENCE

This Function Reference offers an exhaustive description of the Application Programming Interface (API) for Cerence TTS Embedded. This API is defined in ve_ttsapi.h within the Cerence TTS Embedded 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 *Data types, 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 Cerence TTS Embedded in applications.

For more information on the use of the Text-To-Speech (TTS) system, refer to *Chapter 3 - Text-To-Speech System Reference* of this manual.

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

Cerence TTS Embedded 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, the Clock service and the Output Delivery service. Their interfaces are documented in a separate section after the API functions.

Important remarks

Error handling

All Cerence TTS Embedded 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. Cerence TTS Embedded may generate further synthesis results, wait for `ve_ttsProcessText2Speech()` to return before proceeding with actions for new synthesis operations.

Double-call functions

The Cerence TTS Embedded 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.

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 Cerence TTS Embedded is to jump to a given position.

Syntax

```
NUAN_ERROR ve_ttsAnalyzeText(
  VE_HINSTANCE hTtsInst,
```

(continues on next page)

(continued from previous page)

```
  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()*.

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.

ve_ttsGetClmInfo

Description

The function `ve_ttsGetClmInfo ()` returns info on the supported cross-language mappings (CLM). Cerence TTS 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.

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.

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, Structures and Type Definitions* 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 2*, as well as the **Language and voice documentation**.

ve_ttsGetNtsInfo

Description

The function `ve_ttsGetNtsInfo()` returns info on the supported NT-SAMPA mapping for a given language. Cerence TTS 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()*.

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.

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.

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.

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*.

ve_ttsOpen

Description

The function `ve_ttsOpen()` creates a TTS instance based on a specified TTS class. Cerence TTS Embedded 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()*.

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()*.

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 Cerence TTS Embedded 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 Cerence TTS Embedded 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:

- **Cerence TTS Embedded 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.

- **Cerence TTS Embedded has delivered speech for some parts of the input text:**

  This means that Cerence TTS Embedded 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().

Cerence TTS Embedded 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, Cerence TTS Embedded 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()*.

ve_ttsProcessText2SpeechStartingAt

Description

The function ve_ttsProcessText2SpeechStartingAt() synthesizes an input text similar to what 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 cntTaIndex
)
```

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()\ .
cntTaIndex	[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()*.

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 cntInDataLength,
    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 Cerence TTS Embedded 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 Cerence TTS Embedded will not apply the data during synthesis.

Cerence TTS Embedded 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.

Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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 Cerence TTS Embedded 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_ttsResourceUnload(
    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()*.

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()*.

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.

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 in another thread, i.e. executing the function `ve_ttsProcessText2Speech()`. 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 : "embedded-high"

Cerence TTS 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. Cerence TTS 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, Cerence TTS selects the preferred voice in the language and the higher-quality voice operating point.
- If you only specify the voice, Cerence TTS looks for it over all available languages, and selects the higher-quality voice operating point.
- If you specify the language and the voice, Cerence TTS 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 : "embedded-high"

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

- VE_PARAM_LANGUAGE : "American English"
- VE_PARAM_VOICE : "Paulina"
- VE_PARAM_VOICE_OPERATING_POINT : "embedded-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()*.

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()*.

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 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_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`

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

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

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.

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.

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.

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.

Multithreading service

The following functions define an interface for multithreading. 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.

pfOpen

Description

The function *pfOpen() is called when a TTS class or an instance needs to instantiate an handler for a new thread.

Syntax

```
NUAN_ERROR (*pfOpen)(
    void *hClass,
    void *hHeap,
    void **hThread
)
```

Parameters

hClass	[in] not used in this context.
hHeap	[in] pointer to the heap currently used
hThread	[out] handler of the thread of concern

Return Values

If the function completes successfully, it returns the code NUAN_OK. If not, it returns an error code.

pfClose

Description

The function *pfClose() is called when a TTS class or an instance no longer needs a thread instance previously created with *pfOpen(). It frees the memory associated with the handler of a thread.

Syntax

```
NUAN_ERROR (*pfClose)(
    void *hThread
)
```

Parameters

hThread	[in] handler of the thread of concern

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfStart

Description

The function `*pfStart()` sets the function containing the thread body and launches its execution in the new thread handled by hThread.

Syntax

```
NUAN_ERROR (*pfStart)(
    void *hThread,
    VPLATFORM_THREAD_STARTFUNC *hStartFunction,
    void *hArgs,
    size_t hStackSize
)
```

Parameters

hThread	[in] handler of the thread of concern.
hStartFunction	[in] pointer to the main function of the thread
hArgs	[in] handler of the thread of concern
hStackSize	[in] size in bytes of the thread stack. If 0 the stack size will be set to the given platform's default.

Types

`VPLATFORM_THREAD_STARTFUNC` is a typedef:

```
(void*)(*VPLATFORM_THREAD_STARTFUNC)(void*)
```

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfJoin

Description

The function `*pfJoin()` suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

Syntax

```
NUAN_ERROR (*pfJoin)(
    void *hThread,
    void *hStatus
)
```

Parameters

hThread	[in] handler of the thread of concern
hStatus	[out] the hThread exit status

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfSleepMs

Description

The function `*pfSleepMs()` puts the calling thread in a sleep state.

Syntax

```
NUAN_ERROR (*pfSleepMs)(
    void *hThread,
    unsigned short *pMs,
)
```

Parameters

hThread	[in] handler of the thread of concern
pMs	[in] duration in ms of the sleep time

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfCallingId

Description

The function `*pfCallingId()` returns the unique ID of the calling thread.

Syntax

```
NUAN_ERROR (*pfCallingId)(
    void *hThread,
    unsigned int *hThreadId,
)
```

Parameters

hThread	[in] handler of the thread of concern
hThreadId	[out] the unique ID of the calling thread.

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

Semaphores Service

The following functions define an interface for semaphores. 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.

pfOpen

Description

The function `*pfOpen()` is called when a TTS class or an instance needs to instantiate an handler for a new semaphore.

Syntax

```
NUAN_ERROR (*pfOpen)(
    void *hClass,
    void *hHeap,
    int *pInit,
    int *pMax,
    void **hSemaphore
)
```

Parameters

hClass	[in] not used in this context.
hHeap	[in] pointer to the heap currently used
pInit	[in] the initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to pMax.
pMax	[in] the maximum count for the semaphore object. This value must be greater than zero.
hSemaphore	[out] handler of the semaphore of concern

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfClose

Description

The function `*pfClose()` is called when a TTS class or an instance no longer needs a semaphore instance previously created with `*pfOpen()`. It frees the memory associated with the handler of a semaphore.

Syntax

```
NUAN_ERROR (*pfClose)(
    void *hSemaphore
)
```

Parameters

hSemaphore	[in] handler of the semaphore of concern

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfAcquire

Description

The function `*pfAcquire()` decrements the semaphore handled by `hSemaphore`. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has value zero, then the call blocks until either it become possible to perform the decrement (someone else calls the `*pfRelease()`).

Syntax

```
NUAN_ERROR (*pfAcquire)(
    void *hSemaphore
)
```

Parameters

hSemaphore	[in] handler of the semaphore of concern

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

pfRelease

Description

The function `*pfRelease()` increments (unlocks) the semaphore handled by `hSemaphore`. If the semaphore's value consequently becomes greater than zero, then another thread blocked in a `*pfAcquire()` call will be woken up and proceed to lock the semaphore.

Syntax

```
NUAN_ERROR (*pfRelease)(
    void *hSemaphore
)
```

Parameters

hSemaphore	[in] handler of the semaphore of concern

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

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.

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.

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 Cerence TTS Embedded 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 Cerence TTS Embedded product builds with extra logging support.

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 Cerence TTS Embedded 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 cOffset,
    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 Cerence TTS Embedded speechbases, and can be implemented using functions like Microsoft Visual Studio `fseeki64()` or Linux `fseeko()`.

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 Cerence TTS Embedded 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 Cerence TTS Embedded 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`.

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.

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 Cerence TTS Embedded 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.

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 cOffset,
    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.

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 Cerence TTS Embedded 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.

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.

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.).

pfOutEvent

Description

The function `*pfOutEvent()` is called when a TTS class or an instance wants to report an event by ID.

Syntax

```
void
(*pfOutEvent)(
    void *hLog,
    NUAN_U32 u32EventId,
    size_t cKeyValues,
    const char **aszKeys,
    const char **asValues
)
```

Parameters

hLog	[in] Log handle of concern.
u32EventId	[in] Event number
cKeyValues	[in] Number of key-value pairs
aszKeys	[in] List of keys
asValues	[in] List of values

Return Values

None.

Comments

The argument `u32EventID` identifies any possible event that can be reported by Cerence TTS Embedded.

Each event comes with its own set of key/value pairs containing supplemental information to clarify the event.

The possible events are:

3	Indicates the beginning of a TTS request
4	Indicates the end of a TTS request
8	Indicates the moment the TTS system provides the first audio
9	Indicates all other audio events
19	Indicates that a user dictionary lookup happened
20	Indicates that a ruleset lookup happened
27	Forced early emission indicates high load and may perform sub-optimal, producing lower quality audio. This only applies to back-end technologies that support the forced early emission feature.

Clock Service

The following function defines a clock interface for retrieving clock values from the underlying operating system.

pfGetRelativeTime

Description

The function `*pfGetRelativeTime()` is called when a TTS class or an instance needs to know the number of milliseconds elapsed since a reference time.

Syntax

```
NUAN_ERROR (*pfGetRelativeTime)(
    void *hClock,
    NUAN_CLOCK_DATA *pClockData
)
```

Parameters

hClock	[in] Clock handle of concern
pClockData	[out] Pointer to the structure where the real, user, and system time values should be written

Return Values

If the function completes successfully, it returns the code `NUAN_OK`. If not, it returns an error code.

Comments

The clock service implementation is expected to store the clock values in the provided `pClockData` structure. If the underlying operating system does not allow the retrieval of the time spent by the current thread in user and/or system/kernel mode, the implementation must store zero in the `utime` and/or `stime` members of the structure `pClockData`.

Note that the values stored in `pClockData` must be expressed as an unsigned integer number of milliseconds elapsed since the reference time. Implementations are free to choose a reference time other than the service initialization time, and to internally use a different time representation, but they must eventually convert the internal represention to milliseconds when outputting values to `pClockData`.

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`	If an error occurred; this causes TTS to stop processing or `<any other value>`

Comments

The output device should check the `pcbMessage` structure to know the meaning of the notified message.

Data Types, Structures and Type Definitions

Below are data structures and data types that are specific for Cerence TTS Embedded. 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;
typedef NUAN_U32       NUAN_TIME;
```

Type definitions

NUAN_CLOCK_DATA

```
typedef struct {
    NUAN_TIME       rtime,
    NUAN_TIME       utime,
    NUAN_TIME       stime,
} NUAN_CLOCK_DATA;
```

Description

The structure `NUAN_CLOCK_DATA` defines a point in time as a triple: real time (i.e., wall time), user time (i.e., amount of time spent by the current thread in user mode), and system time (i.e., amount of time spent by the current thread in system/kernel mode). The value of each member is the number of milliseconds elapsed since a reference time set by the *Clock Service*, for example during its initialization. If the underlying operating system does not provide user and/or system times, the corresponding values are set to zero.

A pointer to a `NUAN_CLOCK_DATA` structure is passed to the function `*pfGetRelativeTime()` of the Clock service to store the current relative time obtained by the service.

Members

rtime	Real time elapsed since the time reference, in milliseconds
utime	Time spent by the current thread in user mode since the time reference, in milliseconds
stime	Time spent by the current thread in system/kernel mode since the time reference, in milliseconds

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 Cerence TTS Embedded.

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_CLOCK_INTERFACE

Description

Set of functions that define the interface of the *Clock Service*. The prototypes of these functions are specified in the section *Clock Service* of the *Function Directory*.

Members

pfGetRelativeTime	[in] Pointer to a function to retrieve the real, user, and system time values.

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_THREAD_INTERFACE

Description

Set of functions that define the interface of the *Multithreading Service*. The prototypes of these functions are specified in the section *Multithreading Service* of the *Function Directory*.

Members

pfOpen	[in] Pointer to a function to create a thread
pfClose	[in] Pointer to a function to destroy a thread instance created with `pfOpen()`
pfStart	[in] Pointer to a function to set the thread function and launch it in a separate thread created with `pfOpen()`.
pfJoin	[in] Pointer to a function that suspends the calling thread until the joined one returns.
pfSleepMs	[in] Pointer to a function that puts in a sleep state the calling thread for a given amount of time.
pfCallingID	[in] Pointer to a function that returns the unique ID of the calling thread.

VE_SEMAPHORE_INTERFACE

Description

Set of functions that define the interface of the *Semaphores Service*. The prototypes of these functions are specified in the section *Semaphores Service* of the *Function Directory*.

Members

pfOpen	[in] Pointer to a function to create a semaphore
pfClose	[in] Pointer to a function to destroy a semaphore instance created with `pfOpen()`
pfAcquire	[in] Pointer to a function that decrements the semaphore handled by `hSemaphore`.
pfRelease	[in] Pointer to a function that increments (unlocks) the semaphore handled by `hSemaphore`.

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 Cerence TTS Embedded 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

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

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()`.
pIClock	Interface of the Clock service. The TTS class and its instances will call these functions to retrieve the real, user, and system time values. This interface is optional and may be `NULL` if no clock service is supplied.
hClock	Clock handle of the Clock service. The TTS class and its instances will pass this handle as an argument to the *pIClock->pfGetRelativeTime()* function call.
pIThread	Interface of the Multithreading service. The TTS class and its instances will call these functions to use the threading primitives of the given platform. The interface is optional and may be `NULL` if no thread service is supplied.
pISemaphore	Interface of the Semaphore service. The TTS class and its instances will call these functions to get access to the synchronization primitives based on semaphores provided by the given platform. The interface is optional and may be `NULL` if no thread service is supplied.
hThdClass	Class handle of the Multithreading and Semaphore services. The TTS class and its instances pass this class handle as an argument on `pIThread->pfOpen()` and `pISemaphore->pfOpen()` calls to create threads or semaphores. This may be `NULL` if no thread service is supplied or if the thread service does not require class data.

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

See *Appendix A: Cerence TTS Languages and Language Codes*.

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.
pfOutEvent	[in] Pointer to a function to report an event by ID.

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 {
    VE_MARKTYPE       eMrkType;
    size_t            cntSrcPos;
    size_t            cntSrcTextLen;
    size_t            cntDestPos;
    size_t            cntDestLen;
    NUAN_U16          usValue;
    NUAN_U32          ulValue;
    char              *szValue;
} VE_MARKINFO;
```

Description

Definition of the marker information structure.

Members

eMrkType	Type of marker
cntSrcPos	Marker position (as byte offset) in the input text
cntSrcTextLen	Length (in bytes) of the piece of text covered by the marker
cntDestPos	Marker position (as an offset in samples) in the output PCM data
cntDestLen	Length (in samples) of the audio fragment covered by the marker
usValue	Used for phoneme markers: L&H+ phoneme symbol ID
ulValue	Parameter value
szValue	String value parameter. Used for a.o. the prompt identification string (`VE_MRK_PROMPT`, as provided in Cerence TTS Designer, or a bookmark marker (`VE_MRK_BOOKMARK`)

Comments

The parameter **`usValue`** 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:

eMrkType	*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_PROMPT       = 0x0010,
};
```

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. The name of the marker is returned as a char string in the szValue field of the marker. This string is owned by the TTS instance, and only valid during the call to the Output Delivery service.
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 szValue 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;
```

(continues on next page)

(continued from previous page)

```
    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
cntPcmBufLen	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

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.

Cerence TTS Embedded 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_PARAM_VOP_VERSION             = 21,
    VE_PARAM_DISABLE_FINAL_SILENCE   = 22,
    VE_PARAM_NOCLMLANGUAGES          = 23,
    VE_PARAM_TIMBRE                  = 24
} VE_PARAMID;
```

Description

Identifier of different parameters.

Members

`VE_PARAM_LANGUAGE`	Language name as found in the broker header file. Parameter value field: `szStringValue`
`VE_PARAM_VOICE`	Voice name as found in the broker header file. Parameter value field: `szStringValue`
`VE_PARAM_VOICE_OPERATING_POINT`	Name of the voice operating point as found in the broker header file. The supported names are the following: • "embedded-high" • "embedded-pro" • "embedded-compact" Parameter value field: `szStringValue`
`VE_PARAM_FREQUENCY`	Sampling frequency, see the definition of `VE_FREQUENCY`. Parameter value field: `usValue`

Continued on next page

Table 1 – continued from previous page

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>\`. 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_TYPE_OF_CHAR	Character encoding for the synthesis input text passed into `ve_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_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_TIMBRE	Timbre level, a scale factor (in %) on the timbre of the current speaker. The range is [50..200]; For value higher than 100 [100..200] the voice sounds younger; reversely, values lesser than 100 [50..100] make the voice sound older. 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`

Continued on next page

Table 1 – continued from previous page

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_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_LIDSCOPE	Defines the parts of the input text for which Cerence TTS Embedded 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 a fragment tagged by <ESC>\lang=unknown. The language of the text within the fragment is determined sentence by sentence (as set by the current read mode), or for the entire fragment in case this is smaller than a sentence. This is the default value. Parameter value field: usValue
VE_PARAM_LIDVOICESWITCH	Defines whether Cerence TTS Embedded 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

Continued on next page

Table 1 – continued from previous page

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
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 the 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_VOP_VERSION	Version number of the voice operating point as found in the broker header file. Parameter value field: szStringValue
VE_PARAM_DISABLE_FINAL_SILENCE	Enable/disable the synthesis of the wait factor silence after the final sentence. Set to 1 to disable. Default value: **0**. Parameter value field: usValue

Continued on next page

Table 1 – continued from previous page

VE_PARAM_NOCLMLANGUAGES	This is the list of foreign languages that a VOP can read (using foreign sounds). The value is a string with a comma-separate list of 3-letter language codes e.g "enu,frc,spm". The value can only be queried by `ve_ttsGetParamList()`.It can not be modified by `ve_ttsSetParamList()`. 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 Cerence TTS Embedded 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 "rn" .

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;
```

Description

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 (using control sequence `<ESC>\mrk=<name>\ `).

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 Cerence TTS Embedded 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	SSML formatted text

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()*.

Return and Error 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
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

Continued on next page

Table 2 – continued from previous page

Value	Meaning
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 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

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.

CHAPTER

FIVE

SAPI5.1 COMPLIANCE

API Support

This section lists which Microsoft Text-To-Speech API v5.1 functions (Text-to-Speech engine Interface) are supported by Cerence TTS Embedded. 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

Text-to-Speech Interface

With the exception of IsUISupported and DisplayUI, the Microsoft SAPI5 layer supports all functions of the Cerence TTS Embedded 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

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

Cerence TTS Embedded 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 Cerence TTS Embedded.

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 Cerence TTS Embedded voices is 90 instead of 100.

ISpVoice::GetVolume

The default volume of Cerence TTS Embedded 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 Cerence TTS Embedded.

ISpVoice::DisplayUI

This member function is not supported by Cerence TTS Embedded.

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 Cerence TTS Embedded'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 Cerence TTS Embedded.

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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".

Emph

Description

This XML tag emphasizes the next sentence to be spoken.

Syntax

```
<Emph> Input text </Emph>
```

Comments

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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

Cerence TTS Embedded 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" .

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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".

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

Cerence TTS Embedded 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>
```

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

Cerence TTS Embedded supports this control tag.

The default volume of Cerence TTS Embedded 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" .

APPENDIX

A

CERENCE TTS LANGUAGES AND LANGUAGE CODES

The table below lists the Cerence TTS languages and the associated languages codes supported by Cerence TTS Embedded.

Language name	Language code
Arabic	ARW
Arabic Gulf	ARG
American English	ENU
Argentinian Spanish	SPA
Australian English	ENA
Basque	BAE
Belgian Dutch	DUB
Bengali	BEI
Bhojpuri	BHI
Brazilian Portuguese	PTB
British English	ENG
Canadian French	FRC
Catalan	CAE
Chilean Spanish	SPL
Chinese Mandarin	MNC
Colombian Spanish	SPC
Croatian	HRH
Czech	CZC
Danish	DAD
Dongbei	DOC
Dutch	DUN
Farsi	FAI
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
Kannada	KAI
Korean	KOK
Malay	MSM

Continued on next page

Table 1 – continued from previous page

Language name	Language code
Marathi	MAI
Mexican Spanish	SPM
Norwegian	NON
Polish	PLP
Portuguese	PTP
Romanian	ROR
Russian	RUR
Sichuanese	SIC
Shaanxi	SXC
Shanghainese	SHC
Slovak	SKS
Scottish English	ENS
South African English	ENZ
Spanish	SPE
Swedish	SWS
Tamil	TAI
Taiwanese Mandarin	MNT
Telugu	TEI
Thai	THT
Turkish	TRT
Ukrainian	UKU
Valencian	VAE
Vietnamese	VIV

APPENDIX

B

CERENCE TTS VOICES

The table below lists the Cerence TTS voices supported by Cerence TTS Embedded:

- LNG: language code
- Voice: voice name
- Gender: "Female" | "Male"
- Operating point: "embedded-compact" | "embedded-pro" | "embedded-high" | "embedded-premium"

For each operating point, the table lists the available foreign language extensions (language codes under ML) and the supported styles. The table also specifies if a gilded speech add-on is available for a given voice.

Note that you can only use a voice operating point if it is installed.

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
ARG	Mariam	Female	embedded-compact		Neutral	
ARG	Mariam	Female	embedded-high		Neutral Lively Forceful	
ARG	Mariam	Female	embedded-pro		Neutral	
ARW	Laila	Female	embedded-compact		Neutral	On Demand
ARW	Laila	Female	embedded-high		Neutral	On Demand
ARW	Laila	Female	embedded-pro		Neutral	On Demand
ARW	Maged-ml	Male	embedded-high	ENG	Neutral Lively Forceful	
ARW	Maged-ml	Male	embedded-pro	ENG	Neutral	
ARW	Maged	Male	embedded-compact		Neutral	
ARW	Maged	Male	embedded-high		Neutral	
ARW	Maged	Male	embedded-pro		Neutral	
ARW	Tarik	Male	embedded-compact		Neutral	On Demand
ARW	Tarik	Male	embedded-high		Neutral	On Demand
ARW	Tarik	Male	embedded-pro		Neutral	On Demand
BAE	Miren	Female	embedded-compact		Neutral	On Demand
BAE	Miren	Female	embedded-high		Neutral	On Demand
BAE	Miren	Female	embedded-pro		Neutral	On Demand
BEI	Paya	Female	embedded-compact		Neutral	
BEI	Paya	Female	embedded-high		Neutral	
BEI	Paya	Female	embedded-pro		Neutral	
BGB	Daria	Female	embedded-compact		Neutral	
BGB	Daria	Female	embedded-high		Neutral Lively Forceful	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
BGB	Daria	Female	embedded-pro		Neutral	
BHI	Jaya	Female	embedded-compact		Neutral	
BHI	Jaya	Female	embedded-high		Neutral	
BHI	Jaya	Female	embedded-pro		Neutral	
CAE	Jordi	Male	embedded-compact		Neutral	On Demand
CAE	Jordi	Male	embedded-high		Neutral	On Demand
CAE	Jordi	Male	embedded-pro		Neutral	On Demand
CAE	Montserrat	Female	embedded-compact		Neutral	On Demand
CAE	Montserrat	Female	embedded-high		Neutral	On Demand
CAE	Montserrat	Female	embedded-pro		Neutral	On Demand
CAH	Aasing-ml	Male	embedded-pro	ENG	Neutral	
CAH	Aasing	Male	embedded-compact		Neutral	
CAH	Sin-ji	Female	embedded-pro		Neutral	
CAH	Sinji-ml	Female	embedded-pro	ENG	Neutral	
CAH	Sinji	Female	embedded-compact		Neutral	
CZC	Iveta	Female	embedded-compact		Neutral	
CZC	Iveta	Female	embedded-high		Neutral	
CZC	Iveta	Female	embedded-pro		Neutral	
CZC	Zuzana-ml	Female	embedded-high	ENG GED	Neutral Lively Forceful	
CZC	Zuzana-ml	Female	embedded-pro	ENG GED	Neutral	
CZC	Zuzana	Female	embedded-compact		Neutral	
CZC	Zuzana	Female	embedded-high		Neutral	
CZC	Zuzana	Female	embedded-pro		Neutral	
DAD	Magnus	Male	embedded-compact		Neutral	On Demand
DAD	Magnus	Male	embedded-high		Neutral Lively Forceful	On Demand
DAD	Magnus	Male	embedded-pro		Neutral	On Demand
DAD	Sara	Female	embedded-compact		Neutral	
DAD	Sara	Female	embedded-high		Neutral Lively Forceful	
DAD	Sara	Female	embedded-pro		Neutral	
DOC	Dongmei-ml	Female	embedded-pro	ENU	Neutral	
DUB	Ellen	Female	embedded-compact		Neutral	
DUB	Ellen	Female	embedded-high		Neutral	
DUB	Ellen	Female	embedded-pro		Neutral	
DUN	Claire-ml	Female	embedded-high	ENG FRF GED ITI SPE	Neutral Lively Forceful	
DUN	Claire-ml	Female	embedded-pro	ENG FRF GED ITI SPE	Neutral	
DUN	Claire	Female	embedded-compact		Neutral	
DUN	Claire	Female	embedded-high		Neutral	
DUN	Claire	Female	embedded-pro		Neutral	
DUN	Xander	Male	embedded-compact		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
DUN	Xander	Male	embedded-high		Neutral Lively Forceful	
DUN	Xander	Male	embedded-pro		Neutral	
ENA	Karen	Female	embedded-compact		Neutral	
ENA	Karen	Female	embedded-high		Neutral Lively Forceful	
ENA	Karen	Female	embedded-pro		Neutral	
ENA	Lee	Male	embedded-compact		Neutral	
ENA	Lee	Male	embedded-high		Neutral Lively Forceful	
ENA	Lee	Male	embedded-pro		Neutral	
ENE	Moira	Female	embedded-compact		Neutral	
ENE	Moira	Female	embedded-high		Neutral	
ENE	Moira	Female	embedded-pro		Neutral	
ENG	Daniel	Male	embedded-compact		Neutral	
ENG	Daniel	Male	embedded-high		Neutral	
ENG	Daniel	Male	embedded-pro		Neutral	
ENG	Kate	Female	embedded-compact		Neutral	On Demand
ENG	Kate	Female	embedded-high		Neutral	On Demand
ENG	Kate	Female	embedded-pro		Neutral	On Demand
ENG	Malcolm	Male	embedded-compact		Neutral	
ENG	Malcolm	Male	embedded-high		Neutral Lively Forceful	
ENG	Malcolm	Male	embedded-pro		Neutral	
ENG	Oliver	Male	embedded-compact		Neutral	
ENG	Oliver	Male	embedded-high		Neutral	
ENG	Oliver	Male	embedded-pro		Neutral	
ENG	Serena	Female	embedded-compact		Neutral	
ENG	Serena	Female	embedded-high		Neutral Lively Forceful	
ENG	Serena	Female	embedded-premium		Neutral Lively Forceful	
ENG	Serena	Female	embedded-pro		Neutral	
ENG	Stephanie	Female	embedded-compact		Neutral	
ENG	Stephanie	Female	embedded-high		Neutral	
ENG	Stephanie	Female	embedded-pro		Neutral	
ENI	Rishi-ml	Male	embedded-high	HII	Neutral Lively Forceful	
ENI	Rishi-ml	Male	embedded-pro	HII	Neutral	
ENI	Rishi	Male	embedded-compact		Neutral	
ENI	Rishi	Male	embedded-high		Neutral	
ENI	Rishi	Male	embedded-pro		Neutral	
ENI	Sangeeta	Female	embedded-compact		Neutral	
ENI	Sangeeta	Female	embedded-high		Neutral Lively Forceful	
ENI	Sangeeta	Female	embedded-pro		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
ENI	Veena	Female	embedded-compact		Neutral	
ENI	Veena	Female	embedded-high		Neutral	
ENI	Veena	Female	embedded-pro		Neutral	
ENS	Fiona	Female	embedded-compact		Neutral	
ENS	Fiona	Female	embedded-high		Neutral	
ENS	Fiona	Female	embedded-pro		Neutral	
ENU	Allison	Female	embedded-compact		Neutral	On Demand
ENU	Allison	Female	embedded-high		Neutral	On Demand
ENU	Allison	Female	embedded-pro		Neutral	On Demand
ENU	Ava-ml	Female	embedded-high	FRC SPM	Neutral Lively Forceful	On Demand
ENU	Ava-ml	Female	embedded-premium	FRC SPM	Neutral Lively Forceful	On Demand
ENU	Ava-ml	Female	embedded-pro	FRC SPM	Neutral	On Demand
ENU	Ava	Female	embedded-compact		Neutral	
ENU	Ava	Female	embedded-high		Neutral	
ENU	Ava	Female	embedded-pro		Neutral	
ENU	Evan	Male	embedded-compact		Neutral	
ENU	Evan	Male	embedded-high		Neutral	
ENU	Evan	Male	embedded-pro		Neutral	
ENU	Joelle	Female	embedded-high		Neutral	
ENU	Nathan	Male	embedded-compact		Neutral	
ENU	Nathan	Male	embedded-high		Neutral	
ENU	Nathan	Male	embedded-pro		Neutral	
ENU	Noelle	Female	embedded-high		Neutral	
ENU	Samantha	Female	embedded-compact		Neutral	
ENU	Samantha	Female	embedded-high		Neutral	
ENU	Samantha	Female	embedded-pro		Neutral	
ENU	Susan	Female	embedded-compact		Neutral	On Demand
ENU	Susan	Female	embedded-high		Neutral	On Demand
ENU	Susan	Female	embedded-pro		Neutral	On Demand
ENU	Tom	Male	embedded-compact		Neutral	
ENU	Tom	Male	embedded-high		Neutral Lively Forceful	
ENU	Tom	Male	embedded-pro		Neutral	
ENU	Zoe	Female	embedded-compact		Neutral	
ENU	Zoe	Female	embedded-high		Neutral	
ENU	Zoe	Female	embedded-pro		Neutral	
ENU	Zoe-ml	Female	embedded-high	FRC SPM	Neutral Lively Forceful	
ENU	Zoe-ml	Female	embedded-premium	FRC SPM	Neutral Lively Forceful	
ENU	Zoe-ml	Female	embedded-pro	FRC SPM	Neutral	
ENZ	Tessa	Female	embedded-compact		Neutral	
ENZ	Tessa	Female	embedded-high		Neutral	
ENZ	Tessa	Female	embedded-pro		Neutral	
FAI	Dariush	Male	embedded-compact		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
FAI	Dariush	Male	embedded-high		Neutral Lively Forceful	
FAI	Dariush	Male	embedded-pro		Neutral	
FIF	Onni	Male	embedded-compact		Neutral	On Demand
FIF	Onni	Male	embedded-high		Neutral Lively Forceful	On Demand
FIF	Onni	Male	embedded-pro		Neutral	On Demand
FIF	Satu	Female	embedded-compact		Neutral	
FIF	Satu	Female	embedded-high		Neutral Lively Forceful	
FIF	Satu	Female	embedded-pro		Neutral	
FRC	Amelie-ml	Female	embedded-high	ENU	Neutral Lively Forceful	
FRC	Amelie-ml	Female	embedded-pro	ENU	Neutral	
FRC	Amelie	Female	embedded-compact		Neutral	
FRC	Amelie	Female	embedded-high		Neutral	
FRC	Amelie	Female	embedded-pro		Neutral	
FRF	Audrey-ml	Female	embedded-high	ENG GED ITI SPE	Neutral Lively Forceful	
FRF	Audrey-ml	Female	embedded-premium	ENG GED ITI SPE	Neutral Lively Forceful	
FRF	Audrey-ml	Female	embedded-pro	ENG GED ITI SPE	Neutral	
FRF	Audrey	Female	embedded-compact		Neutral	
FRF	Audrey	Female	embedded-high		Neutral	
FRF	Audrey	Female	embedded-pro		Neutral	
FRF	Aurelie	Female	embedded-compact		Neutral	
FRF	Aurelie	Female	embedded-high		Neutral	
FRF	Aurelie	Female	embedded-pro		Neutral	
FRC	Chantal	Female	embedded-compact		Neutral	
FRC	Chantal	Female	embedded-high		Neutral	
FRC	Chantal	Female	embedded-pro		Neutral	
FRC	Nicolas	Male	embedded-compact		Neutral	
FRC	Nicolas	Male	embedded-high		Neutral Lively Forceful	
FRC	Nicolas	Male	embedded-pro		Neutral	
FRF	Thomas	Male	embedded-compact		Neutral	
FRF	Thomas	Male	embedded-high		Neutral Lively Forceful	
FRF	Thomas	Male	embedded-pro		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
GED	Anna-ml	Female	embedded-high	ENG FRF ITI SPE	Neutral	
GED	Anna-ml	Female	embedded-pro	ENG FRF ITI SPE	Neutral	
GED	Anna	Female	embedded-compact		Neutral	
GED	Anna	Female	embedded-high		Neutral	
GED	Anna	Female	embedded-pro		Neutral	
GED	Markus	Male	embedded-compact		Neutral	
GED	Markus	Male	embedded-high		Neutral	
GED	Markus	Male	embedded-pro		Neutral	
GED	Petra-ml	Female	embedded-high	ENG FRF ITI SPE	Neutral	Available
GED	Petra-ml	Female	embedded-premium	ENG FRF ITI SPE	Neutral Lively Forceful	Available
GED	Petra-ml	Female	embedded-pro	ENG FRF ITI SPE	Neutral	Available
GED	Petra	Female	embedded-compact		Neutral	
GED	Petra	Female	embedded-high		Neutral	
GED	Petra	Female	embedded-pro		Neutral	
GED	Viktor	Male	embedded-compact		Neutral	
GED	Viktor	Male	embedded-high		Neutral Lively Forceful	
GED	Viktor	Male	embedded-pro		Neutral	
GED	Yannick	Male	embedded-compact		Neutral	
GED	Yannick	Male	embedded-high		Neutral	
GED	Yannick	Male	embedded-pro		Neutral	
GLE	Carmela	Female	embedded-compact		Neutral	On Demand
GLE	Carmela	Female	embedded-high		Neutral	On Demand
GLE	Carmela	Female	embedded-pro		Neutral	On Demand
GRG	Melina	Female	embedded-compact		Neutral	
GRG	Melina	Female	embedded-high		Neutral Lively Forceful	
GRG	Melina	Female	embedded-pro		Neutral	
GRG	Nikos	Male	embedded-compact		Neutral	On Demand
GRG	Nikos	Male	embedded-high		Neutral Lively Forceful	On Demand
GRG	Nikos	Male	embedded-pro		Neutral	On Demand
HEI	Carmit	Female	embedded-compact		Neutral	
HEI	Carmit	Female	embedded-high		Neutral Lively Forceful	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
HEI	Carmit	Female	embedded-pro		Neutral	
HII	Kiyara-ml	Female	embedded-high		Neutral Lively Forceful	
HII	Kiyara-ml	Female	embedded-pro		Neutral Lively Forceful	
HII	Lekha	Female	embedded-compact		Neutral	
HII	Lekha	Female	embedded-high		Neutral Lively Forceful	
HII	Lekha	Female	embedded-pro		Neutral	
HII	Neel-ml	Female	embedded-high	ENI	Neutral Lively Forceful	
HII	Neel-ml	Female	embedded-pro	ENI	Neutral	
HII	Neel	Male	embedded-compact		Neutral	
HII	Neel	Male	embedded-high		Neutral	
HII	Neel	Male	embedded-pro		Neutral	
HRH	Lana	Female	embedded-compact		Neutral	
HRH	Lana	Female	embedded-high		Neutral Lively Forceful	
HRH	Lana	Female	embedded-pro		Neutral	
HUH	Mariska	Female	embedded-compact		Neutral	
HUH	Mariska	Female	embedded-high		Neutral Lively Forceful	
HUH	Mariska	Female	embedded-pro		Neutral	
IDI	Damayanti	Female	embedded-compact		Neutral	
IDI	Damayanti	Female	embedded-high		Neutral Lively Forceful	
IDI	Damayanti	Female	embedded-pro		Neutral	
ITI	Alice-ml	Female	embedded-high	ENG FRF GED SPE	Neutral Lively Forceful	
ITI	Alice-ml	Female	embedded-pro	ENG FRF GED SPE	Neutral	
ITI	Alice	Female	embedded-compact		Neutral	
ITI	Alice	Female	embedded-high		Neutral	
ITI	Alice	Female	embedded-pro		Neutral	
ITI	Federica-ml	Female	embedded-high	ENG FRF GED SPE	Neutral Lively Forceful	On Demand
ITI	Federica-ml	Female	embedded-pro	ENG FRF GED SPE	Neutral	On Demand
ITI	Federica	Female	embedded-compact		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
ITI	Federica	Female	embedded-high		Neutral	
ITI	Federica	Female	embedded-pro		Neutral	
ITI	Luca	Male	embedded-compact		Neutral	On Demand
ITI	Luca	Male	embedded-high		Neutral Lively Forceful	On Demand
ITI	Luca	Male	embedded-pro		Neutral	On Demand
ITI	Paola	Female	embedded-compact		Neutral	On Demand
ITI	Paola	Female	embedded-high		Neutral	On Demand
ITI	Paola	Female	embedded-pro		Neutral	On Demand
JPJ	Ayane	Female	embedded-compact		Neutral	
JPJ	Ayane	Female	embedded-high		Neutral	
JPJ	Ayane	Female	embedded-pro		Neutral	
JPJ	Daisuke	Male	embedded-compact		Neutral	
JPJ	Daisuke	Male	embedded-high		Neutral	
JPJ	Daisuke	Male	embedded-pro		Neutral	
JPJ	Ichiro	Male	embedded-compact		Neutral	
JPJ	Ichiro	Male	embedded-high		Neutral	
JPJ	Ichiro	Male	embedded-pro		Neutral	
JPJ	Koharu	Female	embedded-compact		Neutral	
JPJ	Koharu	Female	embedded-high		Neutral	
JPJ	Koharu	Female	embedded-pro		Neutral	
JPJ	Mizuki	Female	embedded-compact		Neutral	
JPJ	Mizuki	Female	embedded-high		Neutral	
JPJ	Mizuki	Female	embedded-pro		Neutral	
JPJ	Sakura	Female	embedded-compact		Neutral	
JPJ	Sakura	Female	embedded-high		Neutral	
JPJ	Sakura	Female	embedded-pro		Neutral	
KAI	Alpana	Female	embedded-compact		Neutral	
KAI	Alpana	Female	embedded-high		Neutral	
KAI	Alpana	Female	embedded-pro		Neutral	
KOK	Minsu	Male	embedded-compact		Neutral	
KOK	Minsu	Male	embedded-high		Neutral Lively Forceful	
KOK	Minsu	Male	embedded-pro		Neutral	
KOK	Nuri-ml	Female	embedded-high	ENU	Neutral	
KOK	Nuri-ml	Female	embedded-pro	ENU	Neutral	
KOK	Sora	Female	embedded-compact		Neutral	
KOK	Sora	Female	embedded-high		Neutral	
KOK	Sora	Female	embedded-pro		Neutral	
KOK	Yuna-ml	Female	embedded-high	ENU	Neutral	
KOK	Yuna-ml	Female	embedded-pro	ENU	Neutral	
KOK	Yuna	Female	embedded-compact		Neutral	
KOK	Yuna	Female	embedded-high		Neutral	
KOK	Yuna	Female	embedded-pro		Neutral	
MAI	Ananya	Female	embedded-compact		Neutral	
MAI	Ananya	Female	embedded-high		Neutral	
MAI	Ananya	Female	embedded-pro		Neutral	
MNC	Bin-bin	Male	embedded-compact		Neutral	
MNC	Binbin-ml	Male	embedded-high	ENU	Neutral	
MNC	Binbin-ml	Male	embedded-pro	ENU	Neutral	
MNC	Bobo-ml	Male	embedded-high	ENU	Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
MNC	Bobo-ml	Male	embedded-pro	ENU	Neutral	
MNC	Lanlan-ml	Female	embedded-high	ENU	Neutral	
MNC	Lanlan-ml	Female	embedded-pro	ENU	Neutral	
MNC	Li-li	Female	embedded-compact		Neutral	
MNC	Lili-ml	Female	embedded-high	ENU	Neutral	
MNC	Lili-ml	Female	embedded-premium	ENU	Neutral	
MNC	Lili-ml	Female	embedded-pro	ENU	Neutral	
MNC	Lisheng-ml	Female	embedded-high	ENU	Neutral	
MNC	Lisheng-ml	Female	embedded-pro	ENU	Neutral	
MNC	Shanshan-ml	Female	embedded-high	ENU	Neutral	
MNC	Shanshan-ml	Female	embedded-pro	ENU	Neutral	
MNC	Shasha-ml	Female	embedded-high	ENU	Neutral	
MNC	Shasha-ml	Female	embedded-pro	ENU	Neutral	
MNC	Taotao-ml	Male	embedded-high	ENU	Neutral	
MNC	Taotao-ml	Male	embedded-pro	ENU	Neutral	
MNC	Tian-tian	Female	embedded-compact		Neutral	
MNC	Tiantian-ml	Female	embedded-high	ENU	Neutral	
MNC	Tiantian-ml	Female	embedded-pro	ENU	Neutral	
MNC	Tingting-ml	Female	embedded-high	ENU	Neutral	
MNC	Tingting-ml	Female	embedded-pro	ENU	Neutral	
MNT	Meijia-ml	Female	embedded-pro	ENU	Neutral	
MNT	Meijia	Female	embedded-compact		Neutral	
MSM	Amira	Female	embedded-compact		Neutral	
MSM	Amira	Female	embedded-high		Neutral Lively Forceful	
MSM	Amira	Female	embedded-pro		Neutral	
NON	Henrik	Male	embedded-compact		Neutral	On Demand
NON	Henrik	Male	embedded-high		Neutral Lively Forceful	On Demand
NON	Henrik	Male	embedded-pro		Neutral	On Demand
NON	Nora	Female	embedded-compact		Neutral	
NON	Nora	Female	embedded-high		Neutral Lively Forceful	
NON	Nora	Female	embedded-pro		Neutral	
PLP	Ewa	Female	embedded-compact		Neutral	
PLP	Ewa	Female	embedded-high		Neutral Lively Forceful	
PLP	Ewa	Female	embedded-pro		Neutral	
PLP	Krzysztof	Male	embedded-compact		Neutral	On Demand
PLP	Krzysztof	Male	embedded-high		Neutral Lively Forceful	On Demand
PLP	Krzysztof	Male	embedded-pro		Neutral	On Demand
PLP	Zosia	Female	embedded-compact		Neutral	On Demand
PLP	Zosia	Female	embedded-high		Neutral	On Demand
PLP	Zosia	Female	embedded-pro		Neutral	On Demand
PTB	Felipe	Male	embedded-compact		Neutral	On Demand
PTB	Felipe	Male	embedded-high		Neutral Lively Forceful	On Demand

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
PTB	Felipe	Male	embedded-pro		Neutral	On Demand
PTB	Fernanda	Female	embedded-compact		Neutral	
PTB	Fernanda	Female	embedded-high		Neutral	
PTB	Fernanda	Female	embedded-pro		Neutral	
PTB	Luciana	Female	embedded-compact		Neutral	
PTB	Luciana	Female	embedded-high		Neutral Lively Forceful	
PTB	Luciana	Female	embedded-premium		Neutral Lively Forceful	
PTB	Luciana	Female	embedded-pro		Neutral	
PTP	Catarina	Female	embedded-compact		Neutral	
PTP	Catarina	Female	embedded-high		Neutral	
PTP	Catarina	Female	embedded-pro		Neutral	
PTP	Joana	Female	embedded-compact		Neutral	
PTP	Joana	Female	embedded-high		Neutral Lively Forceful	
PTP	Joana	Female	embedded-pro		Neutral	
PTP	Joaquim	Male	embedded-compact		Neutral	On Demand
PTP	Joaquim	Male	embedded-high		Neutral Lively Forceful	On Demand
PTP	Joaquim	Male	embedded-pro		Neutral	On Demand
ROR	Ioana	Female	embedded-compact		Neutral	On Demand
ROR	Ioana	Female	embedded-high		Neutral Lively Forceful	On Demand
ROR	Ioana	Female	embedded-pro		Neutral	On Demand
RUR	Katya-ml	Female	embedded-high	ENG	Neutral Lively Forceful	
RUR	Katya-ml	Female	embedded-pro	ENG	Neutral	
RUR	Katya	Female	embedded-compact		Neutral	
RUR	Katya	Female	embedded-high		Neutral	
RUR	Katya	Female	embedded-pro		Neutral	
RUR	Milena	Female	embedded-compact		Neutral	
RUR	Milena	Female	embedded-high		Neutral	
RUR	Milena	Female	embedded-pro		Neutral	
RUR	Yuri	Male	embedded-compact		Neutral	
RUR	Yuri	Male	embedded-high		Neutral Lively Forceful	
RUR	Yuri	Male	embedded-pro		Neutral	
SHC	Lulu-ml	Female	embedded-high	ENU	Neutral	
SIC	Fangfang-ml	Female	embedded-pro	ENU	Neutral	
SIC	Fangfang	Female	embedded-compact		Neutral	
SIC	Feifei-ml	Female	embedded-pro	ENU	Neutral	
SKS	Laura	Female	embedded-compact		Neutral	
SKS	Laura	Female	embedded-high		Neutral Lively Forceful	
SKS	Laura	Female	embedded-pro		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
SPA	Diego	Male	embedded-compact		Neutral	On Demand
SPA	Diego	Male	embedded-high		Neutral	On Demand
SPA	Diego	Male	embedded-pro		Neutral	On Demand
SPA	Isabela	Female	embedded-compact		Neutral	
SPA	Isabela	Female	embedded-high		Neutral	
SPA	Isabela	Female	embedded-pro		Neutral	
SPC	Carlos	Male	embedded-compact		Neutral	On Demand
SPC	Carlos	Male	embedded-high		Neutral	On Demand
SPC	Carlos	Male	embedded-pro		Neutral	On Demand
SPC	Soledad	Female	embedded-compact		Neutral	On Demand
SPC	Soledad	Female	embedded-high		Neutral	On Demand
SPC	Soledad	Female	embedded-pro		Neutral	On Demand
SPC	Ximena	Female	embedded-compact		Neutral	
SPC	Ximena	Female	embedded-high		Neutral	
SPC	Ximena	Female	embedded-pro		Neutral	
SPE	Jorge	Male	embedded-compact		Neutral	On Demand
SPE	Jorge	Male	embedded-high		Neutral Lively Forceful	On Demand
SPE	Jorge	Male	embedded-pro		Neutral	On Demand
SPE	Marisol-ml	Female	embedded-high	ENG FRF GED ITI PTP	Neutral Lively Forceful	
SPE	Marisol-ml	Female	embedded-pro	ENG FRF GED ITI PTP	Neutral	
SPE	Marisol	Female	embedded-compact		Neutral	
SPE	Marisol	Female	embedded-high		Neutral	
SPE	Marisol	Female	embedded-pro		Neutral	
SPE	Monica-ml	Female	embedded-high	ENG FRF GED ITI	Neutral Lively Forceful	
SPE	Monica-ml	Female	embedded-pro	ENG FRF GED ITI	Neutral	
SPE	Monica	Female	embedded-compact		Neutral	
SPE	Monica	Female	embedded-high		Neutral	
SPE	Monica	Female	embedded-pro		Neutral	
SPL	Francisca	Female	embedded-compact		Neutral	
SPL	Francisca	Female	embedded-high		Neutral	
SPL	Francisca	Female	embedded-pro		Neutral	
SPM	Angelica	Female	embedded-compact		Neutral	
SPM	Angelica	Female	embedded-high		Neutral	
SPM	Angelica	Female	embedded-pro		Neutral	
SPM	Juan	Male	embedded-compact		Neutral	
SPM	Juan	Male	embedded-high		Neutral Lively Forceful	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
SPM	Juan	Male	embedded-pro		Neutral	
SPM	Paulina-ml	Female	embedded-high	ENU FRC	Neutral Lively Forceful	
SPM	Paulina-ml	Female	embedded-premium	ENU FRC	Neutral Lively Forceful	
SPM	Paulina-ml	Female	embedded-pro	ENU FRC	Neutral	
SPM	Paulina	Female	embedded-compact		Neutral	
SPM	Paulina	Female	embedded-high		Neutral	
SPM	Paulina	Female	embedded-pro		Neutral	
SWS	Alva	Female	embedded-compact		Neutral	
SWS	Alva	Female	embedded-high		Neutral Lively Forceful	
SWS	Alva	Female	embedded-pro		Neutral	
SWS	Klara	Female	embedded-compact		Neutral	
SWS	Klara	Female	embedded-high		Neutral	
SWS	Klara	Female	embedded-pro		Neutral	
SWS	Oskar	Male	embedded-compact		Neutral	
SWS	Oskar	Male	embedded-high		Neutral Lively Forceful	
SWS	Oskar	Male	embedded-pro		Neutral	
SXC	Haohao-ml	Male	embedded-pro	ENU	Neutral	
TAI	Vani	Female	embedded-compact		Neutral	
TAI	Vani	Female	embedded-high		Neutral Lively Forceful	
TAI	Vani	Female	embedded-pro		Neutral	
TEI	Geeta	Female	embedded-compact		Neutral	
TEI	Geeta	Female	embedded-high		Neutral Lively Forceful	
TEI	Geeta	Female	embedded-pro		Neutral	
THT	Kanya	Female	embedded-compact		Neutral	
THT	Kanya	Female	embedded-high		Neutral Lively Forceful	
THT	Kanya	Female	embedded-pro		Neutral	
THT	Narisa	Female	embedded-compact		Neutral	
THT	Narisa	Female	embedded-high		Neutral Lively Forceful	
THT	Narisa	Female	embedded-pro		Neutral	
TRT	Cem-ml	Male	embedded-high	ENG	Neutral Lively Forceful	
TRT	Cem-ml	Male	embedded-pro	ENG	Neutral	
TRT	Cem	Male	embedded-compact		Neutral	
TRT	Cem	Male	embedded-high		Neutral	
TRT	Cem	Male	embedded-pro		Neutral	
TRT	Yelda	Female	embedded-compact		Neutral	

Continued on next page

Table 1 – continued from previous page

LNG	Voice	Gender	Operating point	ML	Styles	Gilded speech
TRT	Yelda	Female	embedded-high		Neutral Lively Forceful	
TRT	Yelda	Female	embedded-pro		Neutral	
UKU	Lesya	Female	embedded-compact		Neutral	
UKU	Lesya	Female	embedded-high		Neutral Lively Forceful	
UKU	Lesya	Female	embedded-pro		Neutral	
VAE	Empar	Female	embedded-compact		Neutral	
VAE	Empar	Female	embedded-high		Neutral	
VAE	Empar	Female	embedded-pro		Neutral	
VIV	Linh	Female	embedded-compact		Neutral	
VIV	Linh	Female	embedded-high		Neutral Lively Forceful	
VIV	Linh	Female	embedded-pro		Neutral	

APPENDIX

C

COPYRIGHT AND LICENSING OF THIRD-PARTY SOFTWARE

The Cerence TTS Embedded software relies on particular open source software packages. The copyright and licensing information for these packages are included in this section.

Alladin MD5 v1.6

CURL v7.51.0

(continues on next page)

(continued from previous page)

Expat v2.2.6

fft4g

hts_engine API v1.06

```
/* ----------------------------------------------------------------- */
/*           The HMM-Based Speech Synthesis Engine "hts_engine API"  */
```

(continues on next page)

(continued from previous page)

Intel/DVI ADPCM coder/decoder v1.2

(continues on next page)

(continued from previous page)

Kazlib v1.20

libfixmath r37

Lua v5.3.1

MeCab v0.996

NAIST Japanese dictionary v0.6.3b

OpenBLAS v0.2.20

(continues on next page)

(continued from previous page)

```
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.
```

OPUS audio codec v1.1-beta

```
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.
```

PCRE v5.0

```
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.
```

(continues on next page)

(continued from previous page)

```
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.

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.
```

RSA MD5

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All rights
reserved.

License to copy and use this software is granted provided that it is
identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm"
in all material mentioning or referencing this software or this
function.

License is also granted to make and use derivative works provided that
such works are identified as "derived from the RSA Data Security, Inc.
MD5 Message-Digest Algorithm" in all material mentioning or referencing
the derived work.

RSA Data Security, Inc. makes no representations concerning either the
merchantability of this software or the suitability of this software for
any particular purpose. It is provided "as is" without express or
implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
```

Speech signal processing toolkit (SPTK) v3.5

```
/* ----------------------------------------------------------------- */
/*           The Speech Signal Processing Toolkit (SPTK)             */
```

(continues on next page)

(continued from previous page)

```
/*          developed by SPTK Working Group                          */
/*          http://sp-tk.sourceforge.net/                            */
/* ----------------------------------------------------------------- */
/*                                                                   */
/*  Copyright (c) 1984-2007  Tokyo Institute of Technology           */
/*                           Interdisciplinary Graduate School of    */
/*                           Science and Engineering                 */
/*                                                                   */
/*                1996-2011  Nagoya Institute of Technology          */
/*                           Department of Computer Science          */
/*                                                                   */
/* 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 SPTK working group nor the names of its */
/*   contributors may be used to endorse or promote products derived */
/*   from this software without specific prior written permission.   */
/*                                                                   */
/* 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.                                       */
/* ----------------------------------------------------------------- */
```

strtod v8.4.12

```
This software is copyrighted by the Regents of the University of
California, Sun Microsystems, Inc., Scriptics Corporation, and other
parties.

The following terms apply to all files associated with the software
unless explicitly disclaimed in individual files. The authors hereby
grant permission to use, copy, modify, distribute, and license this
software and its documentation for any purpose, provided that existing
copyright notices are retained in all copies and that this notice is
included verbatim in any distributions. No written agreement, license,
or royalty fee is required for any of the authorized uses. Modifications
to this software may be copyrighted by their authors and need not follow
the licensing terms described here, provided that the new terms are
clearly indicated on the first page of each file where they apply.
```

(continues on next page)

(continued from previous page)

wapiti v1.4.0

zlib v1.2.6