## Contents

### About this guide
- xi
### Audience
- xi
### New and changed information
- xi
### Related documentation
- xii
### Where to get help
- xiii
### Typographical conventions
- xiii

#### Chapter 1. A text-to-speech primer
- 1
  - Terminology
    - Grapheme
    - Phoneme
    - Morpheme
    - Orthography
    - Prosody
    - Syntax
  - What happens in TTS conversion
    - Linguistic processing
    - Phonetic processing
    - Acoustic processing

#### Chapter 2. Vocalizer architecture
- 7
  - Vocalizer components
    - Vocalizer API library
    - TTS API support libraries
    - Sample programs
  - API support
  - Product support
  - Input/output behavior of Vocalizer
    - Providing the input text
    - Presentation of the input text
    - Language and voice switching
    - Audio output streaming
  - Modes of operation: In-process and client/server
    - In-process mode
    - Client/server mode
Chapter 3. Installing Vocalizer ............................. 17
Overview of installation procedure ...................... 17
Hardware requirements .................................. 17
Software requirements ................................ 18
Windows requirements ................................ 18
Linux requirements .................................... 18
Optional/additional software ............................ 18
Setting up licenses ..................................... 18
Installing on Windows .................................. 19
Installing the common components on Windows .... 19
Installing the voices on Windows ....................... 19
Configuring the licensing on Windows ............... 20
VXIvalue shared components .......................... 20
Installing on Linux .................................... 20
Installing the common components on Linux ....... 21
Installing the voices on Linux ......................... 22
Configuring the shell environment on Linux ....... 22
Configuring the licensing on Linux .................. 22
Configuring Vocalizer .................................. 23
Testing the installation with sample applications .. 23

Chapter 4. Working with the TTS system .......... 27
Preparing text for TTS ................................. 27
Rewriting the orthography .............................. 27
Control sequences .................................... 28
Native control sequence format ....................... 28
SSML markup ......................................... 29
Defining an alternative escape sequence ............ 29
Control sequence tasks ................................ 29
Tuning TTS output with ActivePrompts ............. 40
Prompt concatenation engine .......................... 41
Loading ActivePrompt databases ..................... 45
Specifying pronunciations with user dictionaries ... 45
Dictionary substitution rules ......................... 46
Text format dictionary example ...................... 47
Dictionary format for Vocalizer ....................... 47
Using the dictionary conversion tool (dictcpl) ... 49
Loading user dictionaries .................................................. 50
User dictionary API calls .................................................. 50
Restrictions on user dictionaries ........................................ 51
Defining replacement rules with user rulesets ....................... 51
Tuning text normalization via user rulesets ......................... 52
User ruleset format ......................................................... 53
Binary user rulesets ....................................................... 56
Restrictions on user rulesets ............................................. 56
Effect of user rulesets on performance ................................ 56
Loading user rulesets ..................................................... 57
Language identifier .......................................................... 57
Custom voices ............................................................... 58
Internet fetch support ..................................................... 58
Web server configuration .................................................. 59
MIME content types ....................................................... 59
Error and diagnostic logs .................................................. 60
Vocalizer log files .......................................................... 61
System logging .............................................................. 63
Application provided logging callbacks ............................... 63
Nuance Watcher ............................................................. 63
Customizing error messages ............................................. 64
Suppressing sensitive data ............................................... 65
Application event logs ..................................................... 65
Event log file format ........................................................ 65
Application provided logging callbacks ............................... 65
Input text log files .......................................................... 65
Event logs—merging distributed logs .................................. 67
Event codes and tokens ................................................... 68
Syntax and size of log records .......................................... 68
Suppressing sensitive data ............................................... 69
Timing accuracy ............................................................. 69
Tokens used for every event .............................................. 69
Standard events and tokens .............................................. 69
NVOCactp—ActivePrompt was used ................................. 70
NVOCapps—application session ....................................... 70
NVOCaudf—first audio .................................................... 70
NVOCaudn—next audio ................................................... 71
NVOCfrmt—file format .................................................... 71
NVOCifnd—internet fetch end .......................................... 71
NVOCifst—internet fetch start ......................................... 72
NVOCinpt—input text ..................................................... 72
NVOClise—license end ..................................................... 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVOCIsr—license refused</td>
<td>73</td>
</tr>
<tr>
<td>NVOCIs—license start</td>
<td>73</td>
</tr>
<tr>
<td>NVOClock—license lock</td>
<td>73</td>
</tr>
<tr>
<td>NVOCsynd—synthesis end</td>
<td>73</td>
</tr>
<tr>
<td>NVOCsyst—synthesis start</td>
<td>74</td>
</tr>
<tr>
<td>NVOCsysw—synthesis switch</td>
<td>74</td>
</tr>
<tr>
<td>NVOCunlo—license unlock</td>
<td>75</td>
</tr>
<tr>
<td>Performance and scalability tips</td>
<td>75</td>
</tr>
<tr>
<td>Operating system restrictions</td>
<td>75</td>
</tr>
<tr>
<td>Optimal audio buffer size</td>
<td>75</td>
</tr>
<tr>
<td>Limiting delays when internet fetching is used</td>
<td>75</td>
</tr>
<tr>
<td>Chapter 5. Configuring Vocalizer</td>
<td>77</td>
</tr>
<tr>
<td>Vocalizer parameters</td>
<td>77</td>
</tr>
<tr>
<td>Hard-coded Vocalizer defaults</td>
<td>77</td>
</tr>
<tr>
<td>Configuration file parameters</td>
<td>77</td>
</tr>
<tr>
<td>Parameters specified while opening an engine instance</td>
<td>77</td>
</tr>
<tr>
<td>Parameters changed on an already open engine instance</td>
<td>78</td>
</tr>
<tr>
<td>Parameters specified via markup</td>
<td>78</td>
</tr>
<tr>
<td>Where to set Vocalizer parameters</td>
<td>79</td>
</tr>
<tr>
<td>Configuration files</td>
<td>80</td>
</tr>
<tr>
<td>Configuration file format</td>
<td>81</td>
</tr>
<tr>
<td>Single value configuration parameters</td>
<td>82</td>
</tr>
<tr>
<td>Multiple value configuration parameters</td>
<td>87</td>
</tr>
<tr>
<td>Chapter 6. Vocalizer API</td>
<td>91</td>
</tr>
<tr>
<td>API call sequence</td>
<td>91</td>
</tr>
<tr>
<td>API compatibility</td>
<td>94</td>
</tr>
<tr>
<td>Defined data types</td>
<td>95</td>
</tr>
<tr>
<td>HTTSINSTANCE</td>
<td>95</td>
</tr>
<tr>
<td>HTTSMAP</td>
<td>95</td>
</tr>
<tr>
<td>HTTSTUNINGDATA</td>
<td>95</td>
</tr>
<tr>
<td>HTTSUSRDICT</td>
<td>95</td>
</tr>
<tr>
<td>HTTSVECTOR</td>
<td>95</td>
</tr>
<tr>
<td>TTSRETVAL</td>
<td>95</td>
</tr>
<tr>
<td>TTS_ERROR_SEVERITY</td>
<td>95</td>
</tr>
<tr>
<td>TTS_EVENT</td>
<td>96</td>
</tr>
<tr>
<td>TTS_FETCHINFO_T</td>
<td>97</td>
</tr>
<tr>
<td>TTS_MARKER</td>
<td>98</td>
</tr>
<tr>
<td>TTS_MARKER_BOOK</td>
<td>98</td>
</tr>
<tr>
<td>TTS_MARKER_PARAGRAPH</td>
<td>99</td>
</tr>
<tr>
<td>TTS_MARKER_PHONEME</td>
<td>99</td>
</tr>
<tr>
<td>TTS_MARKER_POS</td>
<td>99</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TTS_MARKER_SENTENCE</td>
<td>100</td>
</tr>
<tr>
<td>TTS_MARKER_WORD</td>
<td>100</td>
</tr>
<tr>
<td>TTS_OPEN_PARAMS</td>
<td>101</td>
</tr>
<tr>
<td>TTS_PARAM</td>
<td>102</td>
</tr>
<tr>
<td>TTS_PARAM_T</td>
<td>103</td>
</tr>
<tr>
<td>TTS_PARAM_VAL_ARRAY_T</td>
<td>103</td>
</tr>
<tr>
<td>TTS_PARAM_VALUE_T</td>
<td>103</td>
</tr>
<tr>
<td>TTS_SPEAK_DATA</td>
<td>104</td>
</tr>
<tr>
<td>TTS_USRDICT_DATA</td>
<td>105</td>
</tr>
<tr>
<td>TTS_VOICE_INFO</td>
<td>106</td>
</tr>
<tr>
<td>Function descriptions</td>
<td>108</td>
</tr>
<tr>
<td>TtsClose</td>
<td>110</td>
</tr>
<tr>
<td>TtsDisableUsrDictEx</td>
<td>111</td>
</tr>
<tr>
<td>TtsDisableUsrDictsEx</td>
<td>112</td>
</tr>
<tr>
<td>TtsEnableUsrDictEx</td>
<td>113</td>
</tr>
<tr>
<td>TtsGetParams</td>
<td>114</td>
</tr>
<tr>
<td>TtsGetVoiceList</td>
<td>115</td>
</tr>
<tr>
<td>TtsLoadTuningData</td>
<td>116</td>
</tr>
<tr>
<td>TtsLoadUsrDictEx</td>
<td>117</td>
</tr>
<tr>
<td>TtsMapCreate</td>
<td>118</td>
</tr>
<tr>
<td>TtsMapDestroy</td>
<td>119</td>
</tr>
<tr>
<td>TtsMapFreeChar</td>
<td>120</td>
</tr>
<tr>
<td>TtsMapGetBool</td>
<td>121</td>
</tr>
<tr>
<td>TtsMapGetChar</td>
<td>122</td>
</tr>
<tr>
<td>TtsMapGetU32</td>
<td>123</td>
</tr>
<tr>
<td>TtsMapSetBool</td>
<td>124</td>
</tr>
<tr>
<td>TtsMapSetChar</td>
<td>125</td>
</tr>
<tr>
<td>TtsMapSetU32</td>
<td>126</td>
</tr>
<tr>
<td>TtsOpen</td>
<td>127</td>
</tr>
<tr>
<td>TtsProcessEx</td>
<td>128</td>
</tr>
<tr>
<td>TtsSessionEnd</td>
<td>129</td>
</tr>
<tr>
<td>TtsSessionStart</td>
<td>130</td>
</tr>
<tr>
<td>TtsSetParams</td>
<td>131</td>
</tr>
<tr>
<td>TtsStop</td>
<td>133</td>
</tr>
<tr>
<td>TtsSystemInit</td>
<td>134</td>
</tr>
<tr>
<td>TtsSystemTerminate</td>
<td>135</td>
</tr>
<tr>
<td>TtsUnloadTuningData</td>
<td>136</td>
</tr>
<tr>
<td>TtsUnloadUsrDictEx</td>
<td>137</td>
</tr>
<tr>
<td>User callbacks</td>
<td>138</td>
</tr>
<tr>
<td>TTS_SOURCE_CB</td>
<td>139</td>
</tr>
<tr>
<td>TTS_DEST_CB</td>
<td>140</td>
</tr>
<tr>
<td>TTS_EVENT_CB</td>
<td>141</td>
</tr>
</tbody>
</table>
Resource consumption ................................. 177
CPU utilization ....................................... 177
Memory use ............................................ 178
Determining the supported number of channels .......... 178

Appendix B. Vocalizer languages ...................... 179

Appendix C. Copyright and licensing for third party software .................. 181
Copyright and Licensing Information for ACE(TM), TAO(TM), CIAO(TM), and CoSMIC(TM) .......... 181
Apache Group Xerces-C XML Parser .................... 183
Dinkumware C++ Library for Visual C++ ............... 183
RSA Data Security, Inc. MD5 Message-Digest Algorithm .......... 184
IBM International Components for Unicode (ICU) .......... 184
Perl Compatible Regular Expressions (PCRE) ............ 184
About this guide

This guide describes the Nuance® Vocalizer for Network state of the art Text-To-Speech (TTS) system.

Audience

This guide is written for application developers who want to add Vocalizer’s high-quality text-to-speech functionality to their applications and for the system operators who operate those systems.

New and changed information

- Vocalizer adds recorded ActivePrompts, a highly flexible system for concatenated prompt recording playback. This feature allows you to add a set of recordings into Vocalizer, where Vocalizer automatically renders the input text using those recordings when possible, rather than only using text-to-speech or requiring applications to specify each recording file name using SSML <audio>. This includes Nuance supplied ActivePrompt databases for flawless playback in some voices of many common data types (alphanumeric, currency, date, telephone number, cardinal number, and time), support for user-defined recorded ActivePrompt databases for all voices, and support for user-defined custom voices that do all playback using concatenated prompt recordings.

- Vocalizer now supports user-defined SSML say-as types, an extension to the user rulesets feature from previous releases.

- Vocalizer now supports not only text rulesets but also binary rulesets.

- Vocalizer has higher SSML conformance, including <mark> with Unicode string names, <phoneme> with the Unicode IPA alphabet, and optional strict validation against the SSML schema to provide immediate feedback about invalid SSML constructs (enabled by default). SSML <lexicon> is also extended to support any type of Vocalizer tuning data: user dictionaries, user rulesets, and ActivePrompt databases.

- Vocalizer has many usability improvements, including rewritten documentation, native API simplifications (while maintaining backward compatibility), improved error and diagnostic information, Unicode user rulesets, and more robust rules for automatic matching of ActivePrompts.

- Vocalizer shifts to a simplified license enforcement model based on a per-process license pool.

- Vocalizer includes security and robustness improvements, such as a “secure context” parameter for suppressing diagnostic logging of sensitive information like credit card numbers.
Vocalizer can optionally generate event log files that trace Vocalizer requests for capacity planning and application tuning purposes. This includes a sample perl script for generating Vocalizer license use reports.

Vocalizer includes a new set of sample applications that are simpler to use, that are more powerful for testing and experimentation, and that provide better reference code.

There are new API functions (TtsSystemInit and TtsSystemTerminate, TtsGetVoiceList, TtsOpen and TtsClose, TtsSessionStart and TtsSessionEnd, TtsLoadTuningData and TtsUnloadTuningData) and types (TTS_OPEN_PARAMS and TTS_VOICE_INFO). Several functions and types have new names, and there are semantic changes in other API details. See the Release Notes for more information.

Vocalizer adds native support for IETF language codes (such as “en-US”).

Vocalizer no longer supports text format dictionaries, and the binary dictionary format has changed.

ActivePrompt databases are now loaded similar to user rulesets, instead of being automatically loaded. The ActivePrompt database format has also changed.

Vocalizer no longer supports the proprietary client/server solution in favor of the MRCP-based Nuance Speech Server product.

Vocalizer no longer supports the Speechify API in favor of the SAPI 5 and native APIs.

Vocalizer drops the American English proper names Custom G2P module, instead handling that expanded set of proper names by default.

Vocalizer does not support the old email pre-processor, instead handling many email specific constructs by default.

Vocalizer drops the external language identifier API, replacing it with an internal language identifier feature that is more powerful and flexible.

Related documentation

The Nuance Vocalizer for Network documentation set contains:

- **Nuance Vocalizer for Network Developer’s Guide**: Describes the TTS system, including installation, programming, tuning, and monitoring. It also provides reference information for the native API, SSML conformance, SAPI 5 conformance, and tuning data file formats.

- **Nuance License Manager Licensing Guide**: Describes how to obtain, install, and configure licenses.

- **Language Supplement**: Covers the language specific normalization rules and the L&H+ phonetic alphabets used by Vocalizer. One supplement is provided for each language installed.

- **Release Notes**: Describes changes from RealSpeak 4.5 to Nuance Vocalizer for Network 5.0 and information about migrating to Nuance Vocalizer for Network 5.0. Defines known bugs in the release and any other late information that was excluded from the standard documentation.

- **Voice Release Notes**: Describes voice specific changes from RealSpeak 4.5 to Nuance Vocalizer for Network 5.0, voice specific performance metrics, known bugs in the
voice release, and any other late information that was excluded from the Language Supplement.

- SSML schema documents: XML schema documents for use with XML editors and other XML based systems to ensure SSML documents are compatible with Vocalizer. These are the W3C SSML 1.0 schema documents with Vocalizer specific extensions.

## Where to get help

If you have questions or problems, go to the technical support portal at network.nuance.com. This web-based resource center includes technical support guides on specific topics, access to software updates, and technical support news. Access is available to direct Nuance customers and to partners with active technical support contracts. Go to network.nuance.com to create your account or to log on.

For those who want to find out more about speech-related topics, Nuance University offers training courses at several locations worldwide. Nuance University also offers web training and will customize course materials for client sites. For more information about training courses and schedules, please visit www.nuance.com/speech/training, send email to training@nuance.com, or call +1 (781) 565-5000 and say “training.”

To submit comments on the documentation, please send email directly to techdoc@nuance.com. Technical support is not available via this email address. Technical support is only provided through network.nuance.com.

## Typographical conventions

Nuance documentation uses the following conventions:

*italic text* Indicates file and path names, web and email addresses, and terms introduced for the first time. For example:

Edit the *ag.cfg* configuration file.

*Courier New* Indicates a value that you replace. For example:

The format for Start Time is `YYYY-MM-DD HH:MM:[SS]`

The text you actually type could be, for example:

2005-09-10 16:33:16

**Note:** Many Nuance products run on both Windows and Unix platforms. Windows syntax is typically used throughout the documentation. If you are running on a Unix platform, substitute Unix syntax. For example, use `$NUANCE` wherever `%NUANCE%` appears, and use “/” in place of “\” in path names. Differences in usage or functionality on Windows and Unix platforms are noted where relevant. Unless explicitly noted, Unix refers to all Unix platforms supported for this product.
This chapter describes text-to-speech and defines general concepts.

Text-to-speech (TTS) refers to the process of converting normal text from a file or command line into synthesized speech that a user can hear and understand. TTS allows a computer to communicate information to the user in situations where visual communication is inadequate or impossible, and can thus add extra value to a product.

In general, TTS provides a very valuable and flexible alternate to 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 possibilities.

Vocalizer also supports mixing digital audio recordings with TTS for applications where a mixed approach is desired, or doing TTS where the generated audio comes only from concatenating digital audio recordings.

**Terminology**

There are several terms that describe the processing and analysis that occurs during TTS conversion. This section defines these terms, which are used in the rest of the discussion.

**Grapheme**

Text consists of a sequence of *graphemes*.

A grapheme is the most elementary building block in the written form of a language. For example, a grapheme can be an alphabetic character, a grouping of characters that represent a single sound, a digit, punctuation, or a symbol such as the dollar sign ($).

**Phoneme**

A phonetic representation consists of a sequence of *phonemes*.

A phoneme is the most elementary building block in the sound system of a language. In essence, a phoneme constitutes a family of sound variants, which a language treats as being the “same.” Its concept allows establishing patterns of organization in the indefinitely large range of sounds heard in a language. Typically, a specific language contains approximately 50 different phonemes.
Nuance has established its own specifications for the representation of phonemes: the L&H+ phonetic alphabet. It associates each phoneme to a sequence of one or more characters. The phonemes of the supported languages with their associated L&H+ representation are described in the appropriate Vocalizer Language Supplement.

**Morpheme**

A *morpheme* is a linguistic unit of semantic meaning. A morpheme can be a single word, like “map”, or a prefix or suffix like “un-” or “-able”. Morphological analysis breaks down a sentence into its component morphemes and examines how they form words.

**Orthography**

Orthography is the set of rules that define how to write a language. A phonemic orthography is a writing system where each grapheme corresponds to a single phoneme in the language: for example, the L&H+ phonetic alphabet.

**Prosody**

The term *prosody* refers to the rhythm, intonation, and stresses used in a spoken language. All these factors can be important in conveying meaning in spoken language.

For example, the word “record” can be a noun or a verb, depending on which syllable receives the stress. The inflection given to a word can also indicate whether it is a statement or a question. In tonal languages like Mandarin Chinese, using the correct prosody is particularly important.

**Syntax**

The *syntax* of a language consists of the rules that define how words are used to construct a correct sentence. For example, the syntax may define the appropriate subject-verb agreement, or the proper word order in a sentence.

**What happens in TTS conversion**

The goal of TTS conversion is to take the input text, and use it to produce audio that conveys the intended meaning of the text as accurately and naturally as possible. Generally, this conversion takes place in three general stages of processing: linguistic, phonetic, and acoustic.

- First, ordinary text is entered into the system. *Linguistic processing* converts this text into a phonetic representation.
- From this representation, the *phonetic processing* calculates the speech parameters.
- Finally, *acoustic processing* uses these parameters to generate a synthetic speech signal.

Each of these stages involves several steps, which are described in this section.

**Linguistic processing**

The linguistic processing of a TTS system performs several tasks: text normalization, orthographic-to-phonetics conversion (that is, 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 TTS system should be able to read aloud any written text, even if it contains a miscellany of abbreviations, dates, currency indicators, time indicators, addresses, telephone numbers, bank account numbers, and punctuation marks such as quotation marks, parentheses, apostrophes, and so on.

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 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. For example, in Spanish, 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 abbreviation that follows the string.

To handle text normalization, TTS 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 TTS system needs a lot of pronunciation knowledge to perform this task, which includes grapheme-to-phoneme conversion, syllabification, and stress assignment.

Different methods 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 TTS systems use a hybrid strategy that combines word dictionaries, morpheme dictionaries, and pronunciation rules. Although the same strategy can be used for the development of all language versions, each language has its own particularities.

Lexical, morphological, and syntactic analysis

Lexical, morphological, and syntactic analysis is needed to solve pronunciation ambiguities. Lexical analysis converts the input characters into categorized blocks of text, such as sentences, words, and special types like dates and numbers. Morphological analysis breaks down a sentence into its component morphemes (such as root words, prefixes, and suffixes) and examines how they form words. Syntactic analysis determines the grammatical structure of the text.

The English verb record 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 example, 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 TTS synthesis. For example, parsing techniques for TTS, 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, that is, 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:

- Segmental synthesis
- Creation of good prosodic patterns

**Segmental synthesis**

This part of the TTS 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:

- Producing a good intonation contour
- Assigning 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 one or more important or dominant words.

In many 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 (for example, declarative versus interrogative, WH-questions versus yes/no-questions) and to mark important syntactic boundaries (for example, 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 TTS 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:

- Phonetic context
- Stress level
- Position within the word
- Syntactic structure of the sentence
- 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 TTS 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 TTS conversion performs the acoustic processing.

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

Vocalizer uses one of a set of proprietary speech synthesizers to create the speech output.
Chapter 2

Vocalizer architecture

This chapter describes the architecture of Vocalizer. It describes the libraries, the supported APIs and other products, and the input/output behavior. There are a few architecture diagrams for various modes of operation. The last section covers the use of Vocalizer in telephony environments.

Vocalizer components

This section provides a brief overview of the components of the Vocalizer system. All components are installed in the Vocalizer installation directory or one of its subdirectories. Since the installation directory is usually specified via the %VNETWORK\$SDK% environment variable, this variable is used in the text when specifying path names.

Vocalizer API library

The Vocalizer interface to the TTS system is implemented as a shared object or DLL (with the accompanying import library), depending on the platform.

For example, on Windows, the Vocalizer API library is named lhstts.dll (with a corresponding import library lhstts.lib). On Unix, the library is named lhstts.so. The library resides in the install_path\common\speech\components subdirectory of the Vocalizer installation directory. The Vocalizer installer does not automatically add the library to the library load path.

This is the only library the application is required to link in or explicitly load in order to use the TTS functionality.

Note that Vocalizer 5.0 also works with the Microsoft SAPI 5 API, see Microsoft SAPI 5 compliance on page 151.

TTS API support libraries

The TTS API uses several shared support libraries; for example, the Internet fetching library used to retrieve documents on an HTTP server. These libraries are DLLs for Windows, and shared objects for most Unix platforms.

These libraries reside in the install_path\common\speech\components subdirectory of the Vocalizer installation directory.

Sample programs

The SDK includes a number of sample programs. Detailed instructions on how to run them are provided in Testing the installation with sample applications on page 23.
API support

Nuance Vocalizer for Network 5.0 supports the following APIs:

- Nuance Vocalizer for Network native C API
- (Windows only) Microsoft SAPI 5 API
- MRCP version 1 and MRCP version 2 protocol when accessed via the Nuance Speech Server product.

Product support

You can use Vocalizer with the following products from Nuance:

- Nuance® Recognizer
- Nuance® Speech Server
- Nuance® Voice Platform

Input/output behavior of Vocalizer

The following section describes Vocalizer’s input/output behaviors.

Providing the input text

The input text can be provided by the application in three ways: as a text buffer, an input stream, or a document specified via a URI.

- The text buffer method is used by passing the text buffer to the TtsProcessEx function (native API), SAPI 5 Speak API (SAPI 5 API), or MRCP SPEAK method (Nuance Speech Server.) This is the simplest method for the application code and yields the best performance.

- The URI method is used by passing the URI to the TtsProcessEx function (native API) or MRCP SPEAK method (Speech Server). This method is not available for the Microsoft SAPI 5 API. Vocalizer supports documents on an HTTP server and local files. See the illustration for URI input on page 10. This method is particularly useful in configurations with multiple servers: the input texts can be stored on a central Web server. Vocalizer supports caching the retrieved documents and HTTP proxy servers.

- The input stream method is not available for the Microsoft SAPI 5 API or Speech Server. For the native API, it is used by passing NULL text and URI buffers to the TtsProcessEx function. Vocalizer then calls the Source callback function implemented by the application to receive individual blocks of input text on an as-needed basis until the Source callback reports there is no more text. For an illustration of this input mode, see In-process mode on page 9.

Presentation of the input text

The input text can be marked up to control aspects of the generated speech such as voice, pronunciation, volume, rate, and so on.

Vocalizer supports several markup languages:

- The native Vocalizer markup language, which is explained in the language specific User’s Guide for each Vocalizer language.
- W3C SSML v1.0 (XML-based) with some proprietary extensions. See Chapter 8 for more information on SSML support.
- Microsoft SAPI 5 XML tags, only supported when using the Microsoft SAPI 5 API. See Chapter 7 for more information on SAPI 5 support.

Vocalizer supports a wide range of character sets and encodings. The engine handles the transcoding of the input text to the native (or internal) Unicode UTF-16 character set.

**Language and voice switching**

The active language and voice can be specified when a TTS engine instance is opened, between the open and the TTS request, or during the processing of input text (via markup). The voice and language can be switched at any location in the input text; doing so produces a sentence break and may affect prosody.

**Audio output streaming**

The audio output is streamed to the application via the Destination callback. This is a handler implemented by the application which receives the audio chunk by chunk. The application can specify the desired audio format (A-law, µ-law, 16-bit linear).

**Modes of operation: In-process and client/server**

Nuance Vocalizer for Network can operate in client/server mode when used with the Speech Server product, or can be directly accessed as an in-process library (also known as single process or all-in-one mode).

**In-process mode**

For in-process mode the Vocalizer service is fully implemented by libraries (DLLs or shared objects) linked in by the user’s application. All TTS components are then running in the same process, so there is no communication overhead.

The examples below show the system layout of an application using Vocalizer in-process. Only one Vocalizer voice, the American English voice Samantha, has been installed on the machine. The application has created one Vocalizer engine instance via the Vocalizer API.
**Text buffer input**

The figure shows that the application has chosen to provide the text input via a text buffer. In this case, the input text is passed to the Vocalizer engine instance. The audio streams in the opposite direction from the engine instance back to the application.

**URI input**

The figure below shows the architecture used when the application specifies the text input via a URI. In this case, the Vocalizer instance will rely on the Nuance internet fetch component to retrieve the content of the URI.
Chapter 2 Vocalizer architecture

Modes of operation: In-process and client/server

Note that this is a simplified presentation of the internet fetching: in reality the fetch library uses a configurable cache, so the overhead of retrieving previously downloaded documents is minimal.

Client/server mode

In client/server mode the user application submits MRCP requests to the Speech Server.

Nuance Speech Server is an add-on product that provides a network-aware server for ASR and TTS services. The server can reside on any machine on the network, and clients and servers do not have to run on the same operating system. A server instance performs the actual TTS conversion, and sends the generated speech.

Speech Server can create multiple server instances, and each instance runs in a separate thread; so each server can handle multiple requests (one per instance) simultaneously.

A single Speech Server process can handle TTS requests for all the Vocalizer languages and voices that have been installed on the server machine, with no hard limit on the number of languages and voices.

The application tells the server which initial language, accent and/or voice to use for the TTS request. Note that the input text for the TTS request can also contain markup to switch the language and/or voice.

Multiple TTS server processes (or machines) can work with multiple clients (or applications).

Speech Server is fully multi-threaded, and Vocalizer is fully thread-safe.
The figure below shows an example system layout for a client on a Linux machine that is connected with one of two available servers: one running on a Windows machine and one on a Linux machine. The Linux server has two Vocalizer voices available: the American English voice “Samantha,” and a British English voice “Emily.” The figure shows a snapshot of the situation at time x. Before that time, the application has opened an MRCP connection to create a Vocalizer engine instance #1 on the Linux server.

After establishing the MRCP connection, the application sends a request to convert an American English text to speech.

When at a later point in time a German text needs to be processed, the application must open an MRCP connection to the Windows server since the Linux server has not been equipped with a German voice. The application has the option to keep the MRCP connection and corresponding Vocalizer engine instance #1 open for TTS processing of future American or British English texts. In any case, a new Vocalizer engine instance #2
Use of Vocalizer in telephony environments

There are three components to any telephony application using speech technology:

- The main application
- The telephony device
- The speech engines: text-to-speech and/or speech recognition

The main application serves as the brains of the entire system, and is responsible for the overall setup and control of the speech engines and telephony devices. The main application is the master of the system; the telephony device and the speech engines are slaves to the main application.

The telephony device is the point at which voice input or output occurs in the telephony application. In traditional telephony applications, the point of entry is the telephony voice board.

Telephony applications are designed to service many customers at the same time. The concept of a “voice port” is often used in this domain. Each voice port can service one customer at a time. One port is usually associated with one telephone line.

has to be created on the Windows server. The following figure is a snapshot of the processing of the German text.
There are two major system models:

- **All-in-one model:** For this model all the system components run on the same computer. This is the typical configuration in small systems handling a small number of ports. In the all-in-one system, all voice data can be routed between the telephony device and the speech engines through the main application with no network overhead. The application will stream the audio output from the Vocalizer engine to the telephony device. Typically, one Vocalizer engine instance will serve one port.

- **Client/server model:** For this model the application, telephony devices, and speech engines can run on separate machines; the model is enabled by hosting the Nuance Recognizer and/or Vocalizer under the Speech Server. This configuration allows the application to offload the heavy speech engine processing to another computer, allowing the main application to handle more ports on a single computer. Separating the speech engines and/or telephony devices creates a modular system that is more fault-tolerant, flexible, and manageable. In this system, the application can still be the middleman streaming all voice data between the speech engines and the telephony device, or the application can control the speech engines and telephony devices with all the voice data streaming directly between the speech engines and telephony device without application intervention.

### Multiple engine instances

Telephony applications are designed to service many customers at the same time. The concept of a *voice port* is often used in this domain. Each voice port can service one customer at a time.

Typically, a telephony application will direct all TTS requests for one telephone call or dialog session to the same TTS engine instance. The term *call* will be used to refer to a telephone call by a customer or any form of dialog session.

The instance can in principle be created and initialized at the start of the call and destroyed when the call is terminated. But it is usually more efficient to keep the TTS engine instance alive and reuse it for another call; this means one TTS engine instance is assigned to one voice port for a lifetime that is usually much longer than one call.

Note that reusing a TTS engine instance for a new call usually requires the application to restore the engine instance to a well-known state before reusing it for another call. This is needed when the settings of an instance are changed in the course of one call (for example, adjusting the volume, switching the voice or language, loading user dictionaries).

For Speech Server based solutions, the application’s MRCP client library is in complete control of the threading and audio streaming model.

For Microsoft SAPI 5 API based solutions, SAPI 5 supports both synchronous and asynchronous models. See the Microsoft provided SAPI 5 documentation for more information.

For the native API, synthesis is done using a synchronous (blocking) function, so the application must use multiple threads to allow concurrent handling of multiple voice ports. Usually one thread is created for each voice port. The audio is streamed back to the telephone application via the Destination callback. The first argument of the Destination callback, the application data pointer, can be used to direct the audio to the appropriate voice port.
Real-time responsiveness and audio streaming

To support real-time audio streaming, the engine should return audio chunks at a rate that is faster or equal to the play-back rate. The Vocalizer engine attempts to minimize the latency for each TTS request by sending audio back as soon as the buffer provided by the application can be filled completely.

The engine instance achieves this by narrowing the window moving over the data as much as possible without degrading the speech quality. One of the first processing steps is to split off a next sentence. Then the linguistic processing is performed on that sentence. The unit selection process normally operates on one sentence, but for long sentences, it limits its window to one phrase. The final subprocess, the synthesizer, processes one speech unit at a time and narrows its scope to a small chunk of speech samples when nearing the output step.

As soon as that chunk of speech is sent to the application, it can be played back. For the native API, the Destination callback should return as soon as possible, to allow the instance to process the rest of the speech unit, sentence, or text, and fill a next buffer with audio before the audio of the previous buffer has played out. Note that the size of the output chunk is determined by the application: for reasons of efficiency it should not be too small, but to minimize the latency it shouldn’t be too big. A good compromise is a buffer big enough for half a second of audio rounded up to the nearest 1K boundary. (For example, 4096 bytes for 8 kHz sampling rate audio in µ-law or A-law format, or 8192 bytes for 8 kHz sampling rate audio in linear 16-bit PCM format.)

The above explains that the latency for the first audio chunk of a sentence is usually longer than for the following chunks, which is convenient since the effect of an underrun at the start of a sentence is less critical: it results in a longer pause between two sentences.

The latency also depends on the type of input text. The application designer should provision Vocalizer with an adequate safety margin for the possible variance in the latency. For instance, if most TTS requests consist of a text with normal sentences but a few may have extremely long sentences (for example, poorly punctuated e-mails), then allowances should be made for situations where the TTS engine instance will have to deal with long sections of text with no punctuation. Such an occurrence may result in an extended inter-sentence latency, normally audible as a longer pause between two sentences. To reduce the risk for extended inter-sentence latencies, the engine will split up very long sentences at an appropriate location (such as a phrase boundary). It is very rare to have a natural sentence that long. (The length depends on the language and the sentence’s content, but it’s usually around 750 characters.)

Note that when servicing multiple voice ports, instances are not influenced by the badly punctuated input of another instance.

If an audio chunk is only delivered after the previous chunk has already finished playing out, a gap in the speech output will be heard. Such an intra-sentence gap can have a stronger audible effect: for example, it can occur within a word, and often sounds like a click. But such an “underrun” is less likely, and will only start to appear when operating Vocalizer at very high density levels that are at or above the computer’s processing capacity, or when other processes on the system are consuming a lot of CPU or doing a lot of heavy I/O. The effects can be masked by maintaining a queue of audio chunks which allows audio output to accumulate faster than real time to compensate for the rare occasion of a non-real-time response. Usually the audio output device will support the queuing or buffering of audio chunks before they are effectively played out.

In any case, when using the native API, it’s safer for the Destination callback to return a buffer for the next chunk as soon as possible instead of waiting until the playback of the previous chunk has finished. Of course, when the instance runs faster than real-time, the
insertion of some waiting can be appropriate when the size of the queue grows too much (a fixed maximum on the number of buffers would then result in an overrun). Note that because of the throttling mechanism implemented by Vocalizer, the audio chunk delivery rate is limited to two times real-time, thus reducing the risk for overruns. Also note that the first 5 seconds of generated audio are by default not throttled.

**VoiceXML platform integrations**

Nuance Vocalizer for Network is designed to support all of the prompting requirements of VoiceXML platforms, so VoiceXML platforms can simply delegate all prompt playback (including text-to-speech and audio recordings) to Vocalizer. VoiceXML platforms do so by extracting all the VoiceXML fragments relating to prompts into an SSML document, then submitting the SSML to Vocalizer for playback, obtaining a single audio stream from Vocalizer that unifies all the audio from the text-to-speech and audio recordings. This design yields a more robust and efficient platform that is easier to develop and maintain.

For details on how to use Vocalizer in this way, see Integrating Vocalizer into VoiceXML platforms on page 169.

**Using Vocalizer from within multiple processes**

Nuance Vocalizer for Network is designed for use in multi-threaded programs where many simultaneous TTS instances are active at once within a single process, but it can also be used in multi-process environments. However, careful configuration is required when multiple processes access Vocalizer at the same time on the same machine, otherwise the Internet fetch cache may grow larger than expected, or the log files can collide and get corrupted.

For the Internet fetch cache, it is safe to use one cache directory for multi-process environments because Vocalizer automatically partitions the cache into subdirectories based on the current OS process ID. However, all the other Internet fetch cache parameters are implemented on a per-process basis, not on a per-machine basis. For example, if 10 processes use Vocalizer on the same computer at the same time with the same cache_directory and cache_total_size set to 200 MB, then the cache_directory could get as large as 2000 MB (200 MB * 10 processes). The cache_size parameter or server disk space should be adjusted to compensate for this.

For the Vocalizer log files, by default they will collide for multi-process environments. This can be avoided in one of several different ways:

1. Disable the Vocalizer log files, instead relying on Vocalizer’s logging to the operating system’s log, application provided logging callbacks, or Nuance Management Station. See Error and diagnostic logs on page 60 for details.

2. Run each process under a separate user account. By default, all log filenames include the username for the running process, making them unique.

3. Set the Vocalizer temporary files directory $TMPDIR as an environment variable, with a unique value for each process. By default, all the log files are created under $TMPDIR, making them unique. However, this directory must exist before the process starts using Vocalizer.

4. Change the log file paths within the Vocalizer configuration file to include ${PID}, a variable for the current OS process ID, to make the paths unique for each process. Alternatively, you can use an application-defined environment variable that has a unique value for each process, for example, using an application specific environment
variable called SERVER_ID that stores a unique server instance number, and then using ${SERVER_ID} within the Vocalizer configuration file.

5 Create separate user XML configuration files for each process, manually configuring unique log file paths for each. This can be done by copying install_path\config\ttsrshclient.xml to a new file, then removing all the parameters except the ones that need to be overridden; either just TMPDIR, or all of the log file paths. For native C API integrations, specify a different user XML file for each process when calling TtsSystemInit. For Nuance Speech Server integrations, configure each Speech Server instance with a different user XML file within the Speech Server configuration file.
Chapter 3

Installing Vocalizer

The following chapter provides instructions on how to install Vocalizer.

Overview of installation procedure

The installation of Vocalizer is automated. Here is a summary:

1. Verify that the target system meets the Hardware requirements.
2. Verify installation of the prerequisite Software requirements.
3. If necessary, remove the existing Vocalizer or RealSpeak installation. See the Release Notes for more information.
4. Install or configure the license server and download and install licenses. See Setting up licenses.
5. Install the common components and the voices. Configure the licensing. See the appropriate section:
   - Installing on Windows
   - Installing on Linux
6. Set the environment variables to configure the system. See Configuring Vocalizer.
7. Test the installation. See Testing the installation with sample applications.

Hardware requirements

- Intel Pentium III or later (single, dual, and quad-core processors)
- 512 MB RAM
  - Refer to the Language Supplement for each language for more information about memory use.
- 50 MB of free disk space for the core software.
- Sufficient free disk space for at least one voice and language. Requirement depends on the voice and language, but is typically 700–1500 MB.
Software requirements

Windows requirements

Install Nuance Vocalizer for Network on any of the following Windows systems:

- Windows 2003 Server (32-bit version)
- Windows XP Professional with Service Pack 2 or later
- Windows Vista
- Windows 2008 Server (32-bit)
- VMware

Vocalizer uses its own build of the OpenSSL libraries on Windows.

Linux requirements

Install Nuance Vocalizer for Network on any of the following Linux systems (32-bit versions only):

- Red Hat Enterprise Server 4.0 (update 3)
- Red Hat Enterprise Linux 4.0 (AS, ES, or WS)
- Red Hat Enterprise Linux 5.0 (AS, ES, or WS)
- VMware

Vocalizer uses the OS-supplied OpenSSL libraries on Linux. (It gets dynamically loaded using dlopen)

Optional/additional software

Optional. Any web browser (for viewing documentation in HTML format).

Optional. PDF viewing software (for viewing documentation in PDF format).

Optional. An ANSI C compiler (needed for integrating voice processing platforms; also needed for application development via the C API).

Nuance Vocalizer for Network is locale safe: it can run on machines with a non en-US locale. It always uses “C” locale standards for floating point and integer data representation for input and output files used by the TTS code, such as configuration files, log files, and event log files.

Nuance Vocalizer for Network comes with third-party software that is installed via the Vocalizer installer. For more details, see Appendix C.

Setting up licenses

Vocalizer requires a valid license file to perform TTS services. This license file is not supplied with the software, but can be obtained from Nuance.

The document install_path\doc\vnetworkv5\licensing_handbook.pdf under the Vocalizer installation directory describes the licensing in detail. It explains how to obtain and manage licenses and how to install and configure licensing servers. Note that this document refers to the in-process mode of Vocalizer as “all-in-one”.
Vocalizer uses run-time licensing that is based on the number of initialized TTS engine instances, and is co-resident with LAN based licensing servers. Vocalizer uses the FLEXnet third-party software to implement a **floating license model**, so licenses are not required to be dedicated to a specific Vocalizer machine. Instead, one license is needed for each TTS engine instance.

### Installing on Windows

The installation is a simple process that consists of two major steps: common install and voice specific install. Both installers consist of a limited number of input screens which makes it possible for the user to provide the necessary information and to configure the SDK.

The common installer gives the user the opportunity to configure the installation directory. The voice specific installer uses the information retrieved during the installation of the common part to determine the target path of the voice specific installer.

#### Installing the common components on Windows

1. For Windows XP and Vista, ensure all other users are logged out.
2. Open the installer’s InstallShield Wizard. On Vista, ignore the warnings about the installer not being signed and proceed.
3. Click Next to continue. The License Agreement screen appears. Read and accept the terms of the agreement.
4. Click Next to continue. The Custom Setup screen appears. You can also specify the installation path. (Default installation path: C:\Program Files\Nuance\Nuance Vocalizer for Network 5.0\.)
5. Click Next to continue. The Ready to Install screen appears. The setup is now ready to install the selected components.
6. Click Install to continue. The Installing screen appears, displaying the progress of your installation. The installer will now install all the selected components.
7. When the components have been installed, the Finish screen appears. Click Finish to complete the installation.

#### Installing the voices on Windows

After installing the Vocalizer common components, repeat the following steps for every voice that needs to be installed:

1. Open the voice installer’s InstallShield Wizard.
2. Click Next to continue. The Ready to Install screen appears. The setup is now ready to install the selected components.
3. Click Install to continue. The installer installs the selected voice.
4. When the voice has finished installing (and this can take a while), the Finish screen appears. Click Finish to complete the installation.
Configuring the licensing on Windows

You must configure licensing. See Setting up licenses on page 18 for an introduction, and the Nuance License Manager Licensing Guide for more details.

Here is the procedure for installing a licensing server on Windows. The licensing server can also be installed on a Linux machine.

1. Install the licensing server.
2. Determine the hostid of the Windows licensing server.
   
   In order to generate a license, Nuance requires the hostid (Ethernet address for Windows) of the machine that will run the licensing server. The hostid can be obtained using the following command:

   lmutil.exe lmhostid

3. Get your license file and store it on the licensing server.
4. Configure and run the licensing server.
5. Configure Vocalizer to use the appropriate licensing server (the licensing server list), updating the Vocalizer configuration file variables tts_license_ports, tts_license_ports_overdraft_thresh, and license_servers.

   Note: Test the installation with nvncmdline. See Testing the installation with sample applications on page 23 for details.

VXI value shared components

On Windows, the common components installer installs components that are shared with the Nuance Recognizer product under a Common Files\SpeechWorks directory under the standard Program Files directory for the machine, typically C:\Program Files\Common Files\SpeechWorks. This is the standard Windows location for files that are shared (common) between multiple products, and while this location cannot be changed, it only contains 4 MB of data. This data allows applications to pass the same Internet fetch controls (VXIMap and VXIVector objects) into both Vocalizer and the Nuance Recognizer, where that directory includes the required C headers, link library, and runtime DLL. These files should not be removed; they are automatically removed when the last Nuance product that uses those libraries is uninstalled.

Installing on Linux

Vocalizer is distributed in RPM format.

The API installation comes with two RPM files, one for the Vocalizer common components (including the API libraries) and one for the licensing components.

A voice installation comes with one RPM file which is voice specific.

The default location of the install is /usr/local/Nuance/Vocalizer_for_Network_5.0.

You need to be root or have su permissions to install the software. The common components RPM must be installed first. The RPMs can be relocated.

Note: If you do relocate the common components RPM, be sure to relocate any subsequent voice RPM to the same directory. Please see the RPM man pages for additional options.
Installing the common components on Linux

Complete the following steps to install the common components on Linux:

1. Install the OpenSSL libraries if they are not already installed.

   Vocalizer dynamically loads the OpenSSL libraries using the dlopen function using the names `libssl.so` and `libcrypto.so`, which are typically symbolic links within `/usr/lib` to the most recent OpenSSL patch release on the system (typically `/lib/libcrypto.so.version` and `/lib/libssl.so.version`). Vocalizer can work with OpenSSL 0.9.6b or any later 0.9.x release by adjusting those symbolic links, but is currently tested on Linux with openssl-0.9.7a-43.1.

2. Install the common components with the `nvninstall.sh` shell script.

   Several files come with this script. It is crucial that all of these files are present in the directory where you run `nvninstall`:
   - `NuanceVersion-version.i386.rpm` (where `version` is currently 4.0.04.4-10)
   - `nvninstall.sh`
   - `append.sh`
   - `license_agreement.txt`
   - `linux-ck`
   - `nvn-api-version.i386.rpm` (where `version` is 5.0.0-buildid, and `buildid` is a numerical value representing the build)

   The `nvninstall` script takes care of several installation steps:
   - The script checks whether the target operating system is being supported
   - The script checks whether the user has appropriate access rights. (The user should be root.)
   - The script checks whether this is a clean install or an update install.
   - The script and the RPMs installed by it explicitly check the dependencies and report any missing dependency prior to aborting the installation.
   - The script installs the two RPMs shipped and listed above, and allows you to change the default installation path for both.
   - The script displays the license agreement and requires explicit approval.

   If you want to install the common components without the `nvninstall script`, it is crucial to realize that some critical checks may be omitted, and that this may lead to undesired or problematic behavior. In all cases, the `NuanceVersion-version.i386.rpm` must be installed prior to the `nvn-api-version.i386.rpm`. If not, the `nvn-api-version.i386.rpm` fails to install due to the missing dependency.

   Run the `nvninstall` script without arguments.
Installing the voices on Linux

Install the purchased voices:

- Execute the following command for each voice that you need to install:

  ```
rpm -i nvn-full-voice-spec-version-buildid.i386.rpm
  
  version 5.0.0
  full-voice-spec The full specification of the voice to be installed, for example:
  American-English-en-US-Samantha
  buildid A numeric value representing the exact build.
  ```

  The -i option assumes you have installed the `nvn-api-version.i386.rpm` in its default location.

- Alternately, you could install a voice with the following options:

  ```
rpm -Uvh --prefix=/target/path/must/match/nvn-api/install/path
  nvn-full-voice-spec-version-buildid.i386.rpm
  
  The --prefix option allows you to relocate the voice installer to another directory. It’s imperative that this relocation directory is the same as the one specified during the `nvn-api-version.i386.rpm` installation. If not, the installed voice will not work.
  ```

Configuring the shell environment on Linux

Update your environment settings:

1. Set the VNETWORKV5_SDK variable to point to the Vocalizer install directory. The default location of the directory is `/usr/local/Nuance/Vocalizer_for_Network_5.0`

   For example, using the C shell, execute the following command:
   ```
   setenv VNETWORKV5_SDK /usr/local/Nuance/Vocalizer_for_Network_5.0
   ```

2. Update the PATH environment variable to include the following:

   ```
   $VNETWORKV5_SDK/common/speech/components
   
   For example, using the C shell, execute the following command:
   ```
   % setenv PATH $VNETWORKV5_SDK/common/speech/components:$PATH
   ```

3. Update the LD_LIBRARY_PATH environment variable to include the following:

   ```
   /usr/local/Nuance/Vocalizer_for_Network_5.0/common/speech/components
   
   For example, if you are using the C shell, execute the following command:
   ```
   % setenv LD_LIBRARY_PATH
   /usr/local/Nuance/Vocalizer_for_Network_5.0/common/speech/components:$LD_LIBRARY_PATH
   ```

Configuring the licensing on Linux

You must configure licensing. See Setting up licenses on page 18 for an introduction, and the Nuance License Manager Licensing Guide for more details.
Here is the procedure for installing a licensing server on Linux. The licensing server can also be installed on a Windows machine.

1. **Install the licensing server.**

2. **Determine the hostid of the Linux licensing server.**

   In order to generate a license, Nuance requires the hostid of the machine that will run the licensing server. The hostid can be obtained using the following command:

   ```bash
   lmutil lmhostid
   ```

3. **Get your license file and store it on the licensing server.**

4. **Configure and run the licensing server.**

5. **Configure Vocalizer to use the appropriate licensing server (the licensing server list), updating the Vocalizer configuration file variables `tts_license_ports`, `tts_license_ports_overdraft_thresh`, and `license_servers`.

   **Note:** Test the installation with `nvncmdline`. See **Testing the installation with sample applications** on page 23 for details.

### Configuring Vocalizer

Before running Vocalizer, the following environment variables should be set no matter which operating system is used. Some variables are optional. Unix specific environment variables are described in **Installing on Linux** on page 20.

<table>
<thead>
<tr>
<th>Name</th>
<th>When</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VNETWORKV5_SDK</code></td>
<td>always, optional</td>
<td>Vocalizer install directory. Optional only if Vocalizer is installed to the default location, otherwise, this setting is required.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On Windows, the default location is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>C:\Program Files\Nuance\Vocalizer for Network 5.0</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>On Unix, the default location is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>/usr/local/Nuance/Vocalizer_for_Network_5.0</code></td>
</tr>
<tr>
<td><code>PATH</code></td>
<td>always, required</td>
<td>Add to the PATH:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>$VNETWORKV5_SDK\common\speech\components</code></td>
</tr>
</tbody>
</table>

### Testing the installation with sample applications

For demonstration purposes, Vocalizer comes with several applications. The simplest one is the `nvncmdline` program. This can be run to verify the installation.

This program processes one text file or input text string, generating an audio file. By default it processes plain text with optional native Vocalizer markup and produces an 8 kHz µ-law audio file, but it can also be configured to process SSML and produce an 8 kHz A-law audio file or a 8 kHz or 22 kHz linear 16-bit PCM audio file.
To run the sample program, open a command prompt, then run `nvncmdline` from the Vocalizer installation directory as follows:

```
nvncmdline <flags>
```

### Required flags:

- `-l` Language, IETF language code or a Vocalizer language name
- `-n` Voice name
- `-o` Output file name. Can contain directories, both absolute and relative.

### Optional flags:

- `-s` String to speak
- `-f` Text file to speak

```
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a</code></td>
<td>MIME content type describing the output audio format. Default is “audio/basic”</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>MIME content type describing the input text format. Default is “text/plain;charset=iso-8859-1”. For SSML, use “application/synthesis+ssml”.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Saves the output file in WAV format</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Volume [0–100]</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Rate [1–100]</td>
</tr>
<tr>
<td><code>-ew</code></td>
<td>Print word marks</td>
</tr>
<tr>
<td><code>-ep</code></td>
<td>Print phoneme marks</td>
</tr>
<tr>
<td><code>-t</code></td>
<td>Text user rulesets, may be specified multiple times.</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Binary user rulesets, may be specified multiple times.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>User dictionary, may be specified multiple times.</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>ActivePrompt database, may be specified multiple times.</td>
</tr>
<tr>
<td><code>-S</code></td>
<td>Secure context, disable logging sensitive data.</td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Show product version information and exit.</td>
</tr>
</tbody>
</table>
```

For example, the following command process the text “Hello there,” and saves it to `nvncmdline.ulaw` as a µ-law file.

```
%VNETWORKV5_SDK%\nvncmdline -l "en-US" -n Samantha -s "Hello there" -o cmdline.ulaw -a audio/basic
```

The following command speaks the SSML file `input.ssml` and saves it to `nvncmdline.pcm` as an 8 kHz linear PCM file:

```
%VNETWORKV5_SDK%\nvncmdline -l "en-US" -n Samantha -f input.ssml -c application/synthesis+ssml -o cmdline.pcm -a audio/L16;rate=8000
```

To display a help screen, run the program without arguments.

If the program returns with a `TTS_E_LIC_NO_LICENSE` error, it means that Vocalizer could not acquire a license. Check your license configuration.
If no error is returned, everything went fine and the audio file was generated.

The source code of nvncmdline, available in `install_path\api\demos\nvncmdline\src`, is a good starting point for writing an application using the native Vocalizer API. It demonstrates how to use all of the major API functions in the correct sequence, including advanced functionality such as loading user dictionaries and tuning data, changing the rate and volume, and speaking from a URI that is fetched by Vocalizer.
Chapter 4

Working with the TTS system

This chapter covers the following topics:

- Preparing text for TTS
- Tuning TTS output with ActivePrompts
- Specifying pronunciations with user dictionaries
- Defining replacement rules with user rulesets
- Application event logs
- Performance and scalability tips covers some information about sizing and performance.

Preparing text for TTS

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

Vocalizer offers a set of additional mechanisms to intervene in the automatic pronunciation process by means of control sequences specified within the input text or external information files that override the internal system behavior.

In Vocalizer, there are several direct ways to intervene in the pronunciation of text:

- Rewriting the orthography (see Rewriting the orthography on page 27)
- Using control sequences (see Control sequences on page 28)
- Entering phonetic input (see Inserting phonetic input on page 31)
- Tuning with ActivePrompts (see Tuning TTS output with ActivePrompts on page 40)
- Specifying pronunciations in user dictionaries (see Specifying pronunciations with user dictionaries on page 45)
- Using user rulesets (see Defining replacement rules with user rulesets on page 51)

Rewriting the orthography

As the TTS system has limitations, not all messages will come out equally well. Experiment with different ways to phrase the same message (for example, using synonyms or changing word order), and you can often obtain a better result.
You can do this by re-writing static input text, but even dynamically generated text can be rewritten using search and replace patterns via user rulesets. See Defining replacement rules with user rulesets on page 51 for more information.

Control sequences

A control sequence is a piece of text that is not read out, but instead affects how other text is spoken, or that performs a specific task. For example, you can use a control sequence to tell Vocalizer to speak a particular word in your text more loudly than the others, or to insert a bookmark that will appear in your application logs.

By using control sequences, you can acquire full control over the pronunciation of the input text. Control sequences can also be used to insert bookmarks in the text, and perform other tasks that do not directly affect text synthesis.

Vocalizer supports two types of control sequences:

- **Native control sequences:** Vocalizer supports a proprietary syntax for control sequences, which is described in the appropriate HTML format Language Supplement found in the %VNETWORKV5_SDK%\doc\languages directory.
- **SSML markup:** Vocalizer accepts SSML (Speech Synthesis Markup Language) elements in an XML document.

Native control sequence format

All native control sequences follow this general syntax notation:

```
<E> <parameter> = <value> 
```

In this notation:

- `<E>` normally represents the escape character “\x1B” (decimal 27) that generates the ASCII character 27 (Hex 1B). However, it can also be a user-defined escape sequence specified in the Vocalizer configuration file (ttsrshclient.xml).
- `<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

For example, you can insert a half-second pause in your text with the pause parameter:

```
Welcome to our phone system. <E>\pause=500\ How may I help you? 
```

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 affecting the pronunciation of a word should be located outside of that word. If entered inside a word, they will break it into two words.

It is possible to use native control sequences within an SSML document. However, this is difficult because the default `<E>` escape character for native control sequences is forbidden in XML documents. For more on using native control sequences within an XML document, see Defining an alternative escape sequence on page 29.
SSML markup

The SSML markup language includes several elements that have the same effect as native Vocalizer control sequences. For example, you can insert a half-second pause using the SSML <break> element:

Welcome to our phone system. <break time="500ms"/> How may I help you?

See the SSML Specification for a list of SSML elements, and Vocalizer SSML support on page 161 for details on how Vocalizer supports them.

You cannot use SSML outside an XML document.

Using native control sequences within SSML can lead to unexpected behavior, so test them carefully. See Native control sequences on page 163 for more information.

Defining an alternative escape sequence

Under some circumstances you may be unable to use the <ESC> escape sequence (for example, you wish to include a native control sequence in an SSML document). You may also wish to supplement <ESC> with an alternative sequence of your own.

To define an alternative sequence, you must specify it in the <escape_sequence> parameter in the ttsrshclient.xml configuration file. For example, to define three hashmarks (###) as the escape sequence:

<escape_sequence>###</escape_sequence>

You can then use this new sequence instead of <ESC>:

Welcome to our phone system. ###\pause=500\ How may I help you?

This alternative sequence supplements <ESC> rather than replacing it, so you can still use <ESC> for control sequences in non-XML documents. For information on the <escape_sequence> parameter, see Speak parameters on page 83.

Control sequence tasks

The following table summarizes the tasks you can achieve using control sequences, and whether they are supported in SSML as well as the native format:

<table>
<thead>
<tr>
<th>Task</th>
<th>Native</th>
<th>SSML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inserting a digital audio recording</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inserting an ActivePrompt</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Activating implicit matching for an ActivePrompt domain</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inserting phonetic input</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inserting Pinyin input for Chinese languages</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Marking a multi-word string for lookup in the user dictionary</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Inserting a pause</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Guiding text normalization</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inserting a bookmark</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Changing the speaking rate</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Changing the volume</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Inserting a digital audio recording

This control sequence inserts a digital audio recording at a specific location in the text.
For example, this sequence plays the audio recording found at c:\recordings\beep.wav:

Say your name at the beep. `<ESC>\audio="c:\recordings\beep.wav"`\n
The control sequence `<ESC>\audio="<path>"` inserts the recording specified by `<path>`, a URI or local file system path. Vocalizer supports inserting headerless or WAV format audio files that contain µ-law, A-law, or linear 16-bit PCM samples, and the recording’s sampling rate must match the current Vocalizer voice.

The SSML equivalent of this control sequence is the `<audio>` element:

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

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 Tuning TTS output with ActivePrompts on page 40.
This control sequence has no equivalent in SSML.

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. If the value is empty (for example, `<ESC>\domain\`), the most recently activated domain is activated.

For example:

`<ESC>\domain=banking\Is your account number 238773?`

The SSML equivalent of this control sequence consists of using the ssft-domaintype attribute within a `<p>` or `<s>` element:

`<s ssft-domaintype="banking">Is your account number 238773?</s>`
Inserting phonetic input

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

The phonetic input string is composed of symbols of the L&H+ phonetic alphabet, a Nuance specific alphabet that can be conveniently entered from a keyboard. See the Language Supplement for the subset of the L&H+ Phonetic Alphabet relevant for each language, along with examples to help you construct proper phonetic text. However the following general information applies across all languages, with American English examples to help illustrate proper use.

The SSML equivalent of this control sequence is the <phoneme> element:

Would you like a &lt;phoneme alphabet="ipa" ph="təˈmeɪ.toʊ">tomato</phoneme>?

Use the control sequence <ESC><toilet=lhp> to mark the beginning of a piece of phonetic text (switch to L&H+ phonetic input mode), and <ESC><toilet=orth> to mark the end (switch back to orthographic input mode).

In addition to the L&H+ phonetic symbols in the Language Supplement, the following characters should be used to clarify the pronunciation of the phonetic input string:

<table>
<thead>
<tr>
<th>L&amp;H+ symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>Primary word stress</td>
<td><code>&lt;ESC&gt;&lt;toilet=lhp&gt; R+I.'kOR+d &lt;ESC&gt;&lt;toilet=orth&gt; (the verb &quot;record&quot;)</code> versus: <code>&lt;ESC&gt;&lt;toilet=lhp&gt; 'R+E.kOR+d &lt;ESC&gt;&lt;toilet=orth&gt; (the noun &quot;record&quot;)</code></td>
</tr>
<tr>
<td>'2</td>
<td>Secondary word stress</td>
<td><code>&lt;ESC&gt;&lt;toilet=lhp&gt; 2Ek.spl$.'ne&amp;I.S$n &lt;ESC&gt;&lt;toilet=orth&gt; (&quot;explanation&quot;)</code></td>
</tr>
<tr>
<td>&quot;</td>
<td>Sentence accent</td>
<td><code>&lt;ESC&gt;&lt;toilet=lhp&gt;DER+_AR+_tu_@k.sEnts_ ?In_DI_'sEn.t$ns &lt;ESC&gt;&lt;toilet=orth&gt; (&quot;There are TWO ACCENTS in this sentence&quot;)</code></td>
</tr>
<tr>
<td>.</td>
<td>Syllable boundary</td>
<td><code>&lt;ESC&gt;&lt;toilet=lhp&gt; 'sI.$b$I &lt;ESC&gt;&lt;toilet=orth&gt; (&quot;syllable&quot;)</code></td>
</tr>
<tr>
<td>#</td>
<td>Silence (pause)</td>
<td><code>&lt;ESC&gt;&lt;toilet=lhp&gt; ?a&amp;I.&quot;sEd#do&amp;Unt.&quot;du_It &lt;ESC&gt;&lt;toilet=orth&gt; (&quot;I said: don't do it.&quot;)</code></td>
</tr>
</tbody>
</table>
Punctuation marks remain useful within phonetic input to assure correct intonation. Each punctuation mark must be preceded by an asterisk.

<table>
<thead>
<tr>
<th>L&amp;H+ symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>Word delimiter</td>
</tr>
<tr>
<td>*</td>
<td>End of declarative</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>!</td>
<td>End of exclamation</td>
</tr>
<tr>
<td>?</td>
<td>End of question</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
</tbody>
</table>

Example:

<ESC>\toi=lhp"jEs.t$.de&I*,_De&I_'lEft_"?E0.1i*. <ESC>\toi=orth

("Yesterday, they left early.")

Lexical stress and sentence accents can be indicated in phonetic strings by using a single quote (‘) or double quote (”) respectively. Vocalizer automatically converts all lexical stress marks into sentence accents if the phonetic input doesn’t contain any sentence accents.

Note that manually specified lexical stress marks and sentence accents sometimes have no effect in Vocalizer because the synthesis module sometimes needs to override the requested stress or accent.

Example:

<ESC>\toi=lhp\If_D$_'wE.D$R+_Is_fa&In_t$.'mA.R+o&U*,_wi_wIlLiv_fOR+_nu.'jOR+k*.<ESC>\toi=orth

("If the weather is fine tomorrow, we will leave for New York.")

If the phonetic input contains at least one manually added sentence accent, no additional sentence accents are assigned by Vocalizer. Therefore, only those words marked with a double quote (”) will get a sentence accent. As a consequence, input containing only one manual sentence accent produces an almost flat intonation on all the other words.

Example:

<ESC>\toi=lhp\If_D$_wE.D$R+_Is_fa&In_t$."mA.R+o&U*,_wi_wIlLiv_fOR+_nu_jOR+k*.<ESC>\toi=orth

(Only one sentence accent will be realized in, “If the weather is fine tomorrow, we will leave for New York.")

Phonetic input can also be combined with orthographic input. If no sentence accents are found in the input text (indicated by <ESC>\sent_accent in orthographic input, or by " in phonetic input), Vocalizer automatically assigns sentence accents. In the orthographic part of the input, Vocalizer realizes these sentence accents on the basis of part-of-speech and syntactic information. In the phonetic part of the input, all lexical stress marks (if any) are converted into sentence accents. If there are no lexical stress marks, no sentence accent will be realized for the phonetic part of the input. If the user has manually specified one or more sentence accents, no additional sentence accents are realized.
Examples:

If the weather is fine tomorrow, we will leave for <ESC>\toi=lhp\nu."jOR+k <ESC>\toi=orth\.
(No sentence accents are found; Vocalizer automatically assigns sentence accents.)

If the weather is fine tomorrow, we will leave for <ESC>\toi=lhp\nu."jOR+k <ESC>\toi=orth\.
(A sentence accent is specified in the phonetic part of the input text. No additional sentence accents will be realized.)

If the weather is <ESC>\sent_accent\fine tomorrow, we will leave for <ESC>\toi=lhp\nu."jOR+k <ESC>\toi=orth\.
(A sentence accent is specified in the orthographic part of the input text. No additional sentence accents will be realized.)

If the weather is <ESC>\sent_accent\fine tomorrow, we will leave for <ESC>\toi=lhp\nu."jOR+k <ESC>\toi=orth\.
(Two sentence accents were specified; no additional sentence accents will be realized.)

Inserting Pinyin input for Chinese languages

For Chinese languages, the control sequence <ESC>\toi=pyt\ can be used to insert Pinyin input. Use <ESC>\toi=pyt\ at the beginning of the Pinyin input, and <ESC>\toi=orth\ at the end (to restore orthographic input mode).

To ensure correct output, the Latin regions should also be marked using <ESC>\lang=latin\ at the beginning and <ESC>\lang=normal\ at the end.

For example:

<ESC>\toi=pyt\ zhe4li5 miao2 shu4 le5 wo3 gong1 si1 <ESC>\lang=latin\ TTS<ESC>\lang=normal\ xi4 tong3 dui4 tuo1 li2 xu4 lie4 de5 zhi1 chi2.
<ESC>\toi=orth\

This control sequence has no equivalent in SSML.

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

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

For example:

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

This is explained in Specifying pronunciations with user dictionaries on page 45.

This control sequence has no equivalent in SSML.

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 msec.
The SSML equivalent of this control sequence is the `<break>` element:

```xml
<prompt>His name is <break time="300ms"/> Michael.</prompt>
```

### Guiding text normalization

The control sequence `<ESC>\tn=<type>` is used to guide the text normalization processing step. For details on all the supported text normalization types and supported input formats, see the Language Supplement for each language. This control sequence is the equivalent of the SSML `<say-as>` element.

If the text within the control sequence doesn’t match a supported input format, Vocalizer spells the content. While Vocalizer supports a broad range of input formats, application developers should still be careful about the text input format, and should always specify `<ESC>\tn=normal` to specify the end of the text block with that specialized text normalization format.

Some common text normalization (TN) types are listed below. These types can also be used in SSML via `<say-as>`, where the TN type is specified using the interpret-as attribute, as indicated in these examples.

<table>
<thead>
<tr>
<th>TN type</th>
<th>Use</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Address reading</td>
<td><code>&lt;ESC&gt;\tn=address\Apt. 7-12, 28 N. Whitney St., Saint Augustine Beach, FL 32084-6715&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;address&quot;&gt;Apt. 7-12, 28 N. Whitney St., Saint Augustine Beach, FL 32084-6715&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>alphanumeric</td>
<td>Alias of spell:alphanumeric</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>Alias of vxml:boolean</td>
<td></td>
</tr>
<tr>
<td>cardinal</td>
<td>Alias of number</td>
<td></td>
</tr>
<tr>
<td>characters</td>
<td>Alias of spell:alphanumeric</td>
<td></td>
</tr>
<tr>
<td>currency</td>
<td>Currency reading</td>
<td><code>&lt;ESC&gt;\tn=currency\12USD&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;currency&quot;&gt;12USD&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>date</td>
<td>Date reading</td>
<td><code>&lt;ESC&gt;\tn=date\12/3/1995&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;date&quot;&gt;12/3/1995&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>digits</td>
<td>Alias of spell:alphanumeric</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Proper name reading</td>
<td><code>&lt;ESC&gt;\tn=name\Care Telecom Ltd&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;name&quot;&gt;Care Telecom Ltd&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>number</td>
<td>Number reading</td>
<td><code>&lt;ESC&gt;\tn=number\1343455&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;number&quot;&gt;1343455&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>ordinal</td>
<td>Ordinal number reading</td>
<td><code>&lt;ESC&gt;\tn=ordinal\12th&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;ordinal&quot;&gt;12th&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>phone</td>
<td>Telephone number reading</td>
<td><code>&lt;ESC&gt;\tn=vxml:phone\1-800-688-0068&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;phone&quot;&gt;1-800-688-0068&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>raw</td>
<td>Block expansions of abbreviations and acronyms.</td>
<td><code>&lt;ESC&gt;\tn=raw\app.&lt;ESC&gt;\tn=normal\n&lt;say-as interpret-as=&quot;raw&quot;&gt;app.&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>TN type</td>
<td>Use</td>
<td>Examples</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sms</td>
<td>Short message service (SMS) reading</td>
<td><code>&lt;ESC&gt;\tn=sms\CU (-:&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;sms&quot;&gt;CU (-:&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>spell</td>
<td>Alias of spell:strict</td>
<td></td>
</tr>
<tr>
<td>spell:alphanumeric</td>
<td>Spell alphanumeric characters except for white space and punctuation</td>
<td><code>&lt;ESC&gt;\tn=spell:alphanumeric\a34y&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;spell&quot; format=&quot;alphanumeric&quot;&gt; a34y&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>spell:strict</td>
<td>Spell all characters including white space and punctuation</td>
<td><code>&lt;ESC&gt;\tn=spell:strict\ a34y-347&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;spell&quot; format=&quot;strict&quot;&gt;a34y-347&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>state</td>
<td>(Not all languages.) State, city, and province names and abbreviations reading</td>
<td><code>&lt;ESC&gt;\tn=state\ FL&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;state&quot;&gt;FL&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>streetname</td>
<td>(Not all languages.) Street name and abbreviation reading</td>
<td><code>&lt;ESC&gt;\tn=streetname\ Emerson Rd.&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;streetname&quot;&gt;Emerson Rd.&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>streetnumber</td>
<td>(Not all languages.) Street number reading</td>
<td><code>&lt;ESC&gt;\tn=streetnumber\11001-11010&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;streetnumber&quot;&gt;11001-11010&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>telephone</td>
<td>Alias of phone</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>Time of day reading</td>
<td><code>&lt;ESC&gt;\tn=time\10:00&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;time&quot;&gt;10:00&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:boolean</td>
<td>VoiceXML 2.0 defined type for boolean input</td>
<td><code>&lt;ESC&gt;\tn=vxml:boolean\true&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:boolean&quot;&gt;true&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:currency</td>
<td>VoiceXML 2.0 defined type for currencies</td>
<td><code>&lt;ESC&gt;\tn=vxml:currency\EUR15.23&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:currency&quot;&gt;EUR15.23&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:date</td>
<td>VoiceXML 2.0 defined type for dates</td>
<td><code>&lt;ESC&gt;\tn=vxml:date\20100102&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:date&quot;&gt;20100102&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:digits</td>
<td>VoiceXML 2.0 defined type for digit sequences</td>
<td><code>&lt;ESC&gt;\tn=vxml:digits\20051225&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:digits&quot;&gt;20051225&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:number</td>
<td>VoiceXML 2.0 defined type for numbers</td>
<td><code>&lt;ESC&gt;\tn=number\+15243.1235&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:number&quot;&gt;+15243.1235&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:phone</td>
<td>VoiceXML 2.0 defined type for telephone numbers</td>
<td><code>&lt;ESC&gt;\tn=vxml:phone\7815655000&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:phone&quot;&gt;7815655000&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>vxml:time</td>
<td>VoiceXML 2.0 defined type for time strings</td>
<td><code>&lt;ESC&gt;\tn=vxml:time\0100a&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;vxml:time&quot;&gt;0100a&lt;/say-as&gt;</code></td>
</tr>
<tr>
<td>zip</td>
<td>(American English only.) ZIP codes</td>
<td><code>&lt;ESC&gt;\tn=zip\01803&lt;ESC&gt;\tn=normal\ &lt;say-as interpret-as=&quot;zip&quot;&gt;01803&lt;/say-as&gt;</code></td>
</tr>
</tbody>
</table>

**address:** This type provides optimal reading for complete postal addresses. The addressee portion (name portion) should not be included in the address to avoid undesired
expansions of name specific abbreviations. Instead, the name portion should be included in a separate <ESC>\tn=address\ section prior to the <ESC>\tn=address\.

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

Some examples:

\[<ESC>\tn=address\ 244 Perryn Rd Ithaca, NY 14850<ESC>\tn=normal\]

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

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

**raw:** This type provides a more literal reading of the text, such as blocking an undesired abbreviation expansion. <ESC>\tn=raw\ operates on the abbreviations and acronyms as listed in each Language Supplement, but may impact the surrounding text as well. For example, the <ESC>\tn=raw\ in the following text would also block recognition of “12/6” as a date:

\[<ESC>\tn=raw\ Wed. <ESC>\tn=normal\ 12/6\]

**spell:** Vocalizer supports two TN types for spelling text: <ESC>\tn=spell:alphanumeric\ and <ESC>\tn=spell:strict\.

- <ESC>\tn=spell:strict has the following behavior:
  - All characters are spelled, including white space, special characters, and punctuation marks.
  - Characters with diacritics are pronounced as such. (For example, ú is spoken as “u with acute accent.”)
  - “Upper case” is pronounced for upper case letters. (For example, “Abc” is spoken as “Upper case a, b, c.”)

- <ESC>\tn=spell:alphanumeric has the following behavior:
  - All alphabetic and numeric characters are spelled. This excludes white space, special characters, and punctuation marks.
  - Characters with diacritics are pronounced as such. (For example, ú is spoken as “u with acute accent.”)
  - “Upper case” is pronounced for upper case letters. (For example, “Abc” is spoken as “Upper case a, b, c.”)

**vxml:** The vxml prefixed TN types conform to the VoiceXML 2.0 specification: http://www.w3.org/TR/voicexml20/#dmlABuiltins. The vxml input formats are also handled by the non-vxml-prefixed counterparts. For example, <ESC>\tn=time\ covers all the input formats supported by <ESC>\tn=vxml:time\.

**Inserting a bookmark**

The control sequence <ESC>\mrk=\name\ marks the position where it appears in the input text, and has Vocalizer track this position throughout the TTS conversion. A Vocalizer bookmark (\name) can be any text sequence. 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 TTS_EVENT_CB callback function (TTS_EVENT_CB on page 141).
The use of this control sequence does not affect the speech output process.

Some examples:

This bookmark `<ESC>`\mrk=bookmark 1\ does a reference point.  
Another `<ESC>`\mrk=-bookmark 2\ does the same.

The SSML equivalent of this control sequence is the `<mark>` element:

```xml
<prompt>This bookmark `<mark name="bookmark1"/> marks a reference point.  
Another `<mark name="bookmark2"/> does the same.</prompt>
```

### 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.

The SSML equivalent is the rate attribute of the `<prosody>` element:

```xml
<prompt>I can `<prosody rate="+50%">speed up the rate</prosody>  
 `<prosody rate="-25%">or slow it down</prosody>`</prompt>
```

See Rate scale conversion on page 165 for more information.

### 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 typically the default volume.  
For example:

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

The SSML equivalent is the volume attribute of the `<prosody>` element:

```xml
<prompt>`<prosody volume="-50%">I can speak rather quietly,</prosody>  
 `<prosody volume="+50%">but also very loudly.</prosody>`</prompt>
```

See Volume scale conversion on page 164 for more information.

### 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?

This control sequence has no equivalent in SSML, although you can use the `<break>` element to set the length of pauses explicitly.

### 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\`<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 \texttt{\textbackslash readmode=char}, set the end of sentence pause duration using \texttt{\textbackslash wait} instead.

This control sequence has no equivalent in SSML.

**Controlling end-of-sentence detection**

The control sequences \texttt{\textbackslash eos=1} and \texttt{\textbackslash eos=0} control end of sentence detection, with \texttt{\textbackslash eos=1} forcing a sentence break and \texttt{\textbackslash eos=0} suppressing a sentence break. To suppress a sentence break, the \texttt{\textbackslash 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 \texttt{\textbackslash readmode=explicit\_eos} as shown below.

Some examples:

Tom lives in the U.S. \texttt{\textbackslash eos=1} So does John.
180 Park Ave. \texttt{\textbackslash eos=0} Room 24

The SSML equivalent of this control sequence is the \texttt{<s>} (or \texttt{<sentence>}) element to force a sentence break, and a \texttt{<break>} with attribute strength set to “none” to suppress a break:

\begin{verbatim}
<s>
Tom lives in the U.S.\textbackslash e</s>
So does John. 180 Park Ave. \texttt{<break strength="none"/>} Room 24\texttt{/s}
\end{verbatim}

There is no SSML equivalent for the \texttt{\textbackslash readmode=explicit\_eos} sequence.

**Controlling the read mode**

The control sequence \texttt{\textbackslash readmode=mode} can change the reading mode from sentence mode (the default) to various specialized modes:

<table>
<thead>
<tr>
<th>Read mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash readmode=sent}</td>
<td>Sentence mode (the default)</td>
</tr>
<tr>
<td>\texttt{\textbackslash readmode=char}</td>
<td>Character mode (similar to spelling)</td>
</tr>
<tr>
<td>\texttt{\textbackslash readmode=word}</td>
<td>Word-by-word mode</td>
</tr>
<tr>
<td>\texttt{\textbackslash readmode=line}</td>
<td>Line-by-line mode</td>
</tr>
<tr>
<td>\texttt{\textbackslash readmode=explicit_eos}</td>
<td>Explicit end-of-sentence mode (sentence breaks only where indicated by \texttt{\textbackslash eos=1})</td>
</tr>
</tbody>
</table>

Examples:

\begin{verbatim}
<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=word\ Please buy green apples.  
(This sentence will be read word by word.)
<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.)
\end{verbatim}
Chapter 4 Working with the TTS system

Control sequences

39 Nuance Proprietary

This control sequence has no equivalent in SSML.

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.

The SSML equivalent of this control sequence is the <voice> element:

<prompt><voice name="Samantha">Hello, this is Samantha.</voice>
<prompt><voice name="Tom">Hello, this is Tom.</prompt>

To use this control sequence, you must have more than one voice installed.

Labeling text for language identification

The control sequence <ESC>\lang=unknown\ labels all the text from that position up to a <ESC>\lang=normal\ or the end of the input as being from an unknown language. When the language identifier scope is configured to user-defined using the Vocalizer configuration file or API (user-defined is the default), this triggers Vocalizer to use its built-in language identifier to determine the language. For other language identifier scopes, this control sequence is simply ignored.

Vocalizer does language identification on a sentence-by-sentence basis within that region, where for each sentence it will use statistical models to determine the language, then switch the synthesis voice to a voice for that language if necessary. The synthesis voice will be restored to the original voice at the next <ESC>\lang=normal\ or the end of the synthesis request.

Note: Vocalizer does not support specifying an explicit language name instead of “unknown”.

Language identification is only supported for a limited set of languages. For more information on the language identifier in general, and the supported languages, see Language identifier on page 57.

Example:

Le titre de la chanson est : <ESC>\lang=unknown\In Between
<ESC>\lang=normal\n
The SSML equivalent of this control sequence is the xml:lang attribute, which is available for several SSML and VoiceXML elements, including <p>, <prompt>, and <s>:

<prompt>Le titre de la chanson est:</prompt>
<prompt xml:lang="unknown">In Between</prompt>

Indicating a paragraph break

The control sequence <ESC>\para\ indicates a paragraph break, and also implies a sentence break. The only difference between this and <ESC>\eos=1\ (end of sentence) is that this triggers the delivery of a paragraph mark event.
Example:

Introduction to Vocalizer. <ESC>\para\ Vocalizer is a state-of-the-art text to speech system.

The SSML equivalent of this control sequence is the <p> (or <paragraph>) element:

<p>Introduction to Vocalizer.</p>
<p>Vocalizer is a state-of-the-art text to speech system.</p>

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

This control sequence has no equivalent in SSML.

Tuning TTS output with ActivePrompts

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

- **Recorded ActivePrompts** are digital audio recordings that are indexed by an ActivePrompt database. The recordings are stored as individual audio files on a web server or file system. This indexing enables context-sensitive expansions of static or dynamic input text to a sequence of pre-recorded audio recordings, making Vocalizer a powerful prompt concatenation engine for recording-only or mixed TTS and recording applications.

- **Tuned ActivePrompts** are an ActivePrompt database that stores synthesizer instructions so input text fragments are spoken in a particular way. These instructions are created by an application developer using Nuance Vocalizer Studio to adjust tuning parameters and listen to various versions of a prompt, then freezing the prompt. These synthesizer instructions are much smaller than the audio that will be produced.

At runtime, all ActivePrompts can be used in three different ways:

- Explicit insertion using the Nuance <prompt> extension to SSML or the native <ESC>\prompt=prompt\ control sequence.

- Implicit matching where ActivePrompts are automatically used whenever the input text matches the ActivePrompt text. For implicit matching, there are two sub-modes:
  - Automatic mode, where implicit matches are automatically enabled across all the text in all speak requests.
  - Normal mode, where the Nuance ssft-domaintype extension to SSML or the native <ESC>\domain=domain\ control sequence is used to enable implicit matches for specific regions within the input text.

For recorded ActivePrompt databases, automatic matching can be further restricted so it is only done within a text normalization block (<ESC>\tn\ control sequence or SSML <say-as> element) for a specific type. For example, a recorded ActivePrompt database for
Prompt concatenation engine

The Vocalizer prompt concatenation engine feature leverages recorded ActivePrompts to support near flawless playback of static and dynamic input text by concatenating recordings rather than using full TTS. This includes support for recordings only or mixed TTS and recordings, and support for creating custom voices for recording only playback.

Many speech applications are built by manually specifying carrier prompt recordings using SSML <audio>, then using an application library to expand dynamic content like alphanumeric sequences, dates, times, cardinal numbers, and telephone numbers to sequences of SSML <audio> elements. However, Vocalizer’s prompt concatenation engine gives better sounding results with the following advantages:

- Application developers don’t need to purchase, create, or maintain libraries for expanding dynamic content like alphanumeric sequences, dates, times, cardinal numbers, and telephone numbers. Instead, the application can just specify plain input text for Vocalizer to expand, then create an ActivePrompt database that defines the necessary recordings.

- ActivePrompts support context sensitive rules, including prompts that start and/or end on a sentence boundary, on a phrase boundary, on a sentence or phrase boundary, with a specific punctuation symbol, or are phrase internal. For playing back dynamic content, even recording just three variations of each prompt (phrase initial, phrase final, and phrase internal) gives a huge quality boost, producing very natural sounding output.

- Some Vocalizer voices include predefined ActivePrompt databases and recordings for a variety of dynamic types, along with recording scripts that allow easily re-recording those in a different voice. These optionally support phrase initial, phrase final, and phrase internal recording variations for very high quality output as described above. See the Release Notes for each voice to see where this feature is offered, and for the details.

- For static prompts, application developers can choose between specifying plain input text (avoids tediously specifying recording file names), SSML <audio> (recording file names), SSML <prompt> (ActivePrompt names), or using a mixed approach.

- Providing plain input text for all the static and dynamic prompts makes it easy to create rapid application prototypes and to follow rapid application development (RAD) models such as Agile or Extreme Programming, because it uses Vocalizer text-to-speech for all the prompts at the beginning of the project, then adding ActivePrompt databases and recordings later on as required, and independent of the application code.

- Vocalizer produces a single audio stream for all the content rather than relying on rapid fetching and concatenation of individual recording files by another system component such as a telephony platform. This ensures the recordings are contiguous, rather than having the extra gaps that some telephony platforms introduce, which lead to slow playback.

- This solution is extensible to the wide variety of languages and dynamic data types supported by Vocalizer, rather than requiring special linguistic knowledge and major code updates for each new language or data type.
Recorded ActivePrompt database creation

The first step for using Vocalizer for prompt concatenation is to define the set of recordings, then enter them into Nuance Vocalizer Studio to create an ActivePrompt database. Each prompt needs the following:

- Logical prompt name, which the run-time engine transforms to a recording file name by appending a recording file suffix (such as .wav) and then using it as a URI relative to the ActivePrompt database. For example, if the ActivePrompt database
  http://myserver/apdb_rp_tom_alphanum.dat contains a prompt named alphanum/f.alpha0
  and the database specifies a file suffix of .wav, the recording file must be
  http://myserver/alphanum/f.alpha0.wav.

- Input text matched by the prompt. Vocalizer does its matching using a normalized form of the input text (converts it to lowercase, normalizes spaces, expands abbreviations, and so on) so there is flexibility for differences between the run-time and prompt text. It is best to think of this as word-by-word matching after expanding dynamic types like dates, times, and numbers to their word sequence (such as 110 to “one hundred ten”).

- Boundary conditions, one for each side of the input text. This can be one of: sentence boundary, phrase boundary, sentence or phrase boundary, a specific punctuation symbol, phrase internal, or a wildcard (anything).

Some Vocalizer voices include predefined ActivePrompt databases and recordings for a variety of dynamic types, along with recording scripts. When possible, it is best to rely on those ActivePrompt databases (optionally re-recording them with the application voice talent) rather than re-creating those databases from scratch.

For other languages or voices, carefully consider the set of recordings required to speak the application’s static and dynamic content. For the static content, consider each carrier phrase, which are typically listed in user interface documents and straightforward to define. Dynamic content is a bit more challenging: it requires knowing the language specific output word sequences, then determining what variations to record for better sounding output.

For example, a basic recording set for digits playback could be one recording for each of the numbers 0 through 9, using wildcard boundary conditions. While that would produce understandable output, it would not sound natural. Much better output could be obtained by recording three variations of each number 0 through 9: one for phrase initial contexts (left boundary condition of sentence or phrase), one for phrase medial contexts (wildcard boundary conditions), and one for phrase final contexts (right boundary condition of sentence or phrase). Even better output could be obtained by recording digit pairs in those three contexts, so that a digit sequence like “0 2 3 7” after a carrier phrase would be played with one phrase internal recording for “0 2”, then one phrase final recording for “3 7”. Of course, this involves cost versus benefit decisions and may require experimentation to determine the lowest cost solution with a target quality level. The Nuance predefined ActivePrompt databases use both of these techniques and come with recording scripts, so even for new languages and types they provide a good reference point for making these decisions.

For flexibility, it is best to create a separate ActivePrompt database for each dynamic type so applications can selectively enable them, such as one for alphanumeric sequences and another for dates, otherwise they may conflict with each other. They can use the same prompt names and recordings, which is desirable to improve run-time Internet fetch cache performance.

For additional flexibility, Vocalizer’s ActivePrompt run-time engine supports fallback if a recording is missing. This fallback builds a sophisticated ActivePrompt database that has
features like multiple prompt variations, but later only recording a more basic prompt set. Vocalizer is of course also a text-to-speech engine, so if it fails to find any match it will automatically fall back to using text-to-speech output (except for recording only custom voices as described below).

The list of ActivePrompts used at run-time is available within the Vocalizer event log, a log that reports information for application tuning and capacity planning purposes. This is often helpful during ActivePrompt development and testing.

Recording creation

Once the ActivePrompt databases are defined, the next step is obtaining a set of recordings. When using a Nuance predefined ActivePrompt database, this can mean using the Nuance recordings as-is or re-recording them. Custom ActivePrompt databases will always need to be recorded.

As described above, Vocalizer supports fallback if a recording is missing, allowing a smaller set of base recordings to be made for a specific deployment rather than always having to record the full set of recordings specified by the ActivePrompt database.

Vocalizer supports inserting headerless or WAV format audio files that contain μ-law, A-law, or linear 16-bit PCM samples, and the recording’s sampling rate must match the current Vocalizer voice.

Recording only custom voices

Vocalizer makes it easy to define custom voices for recording only playback. This is done by simply choosing or creating a set of ActivePrompt databases, choosing or creating a set of recordings, choosing a custom voice name, and running a Vocalizer tool to add the new voice. These voices are referred to as “CPR-only” (concatenated prompt recording only) voices, and can be used with alternative Vocalizer CPR-only licenses instead of requiring full text-to-speech licenses.

The new voice must be based on an existing installed Vocalizer voice, such as defining a custom Maureen voice based on the Nuance Tom voice. The custom voice re-uses the Vocalizer data of the template voice for input text processing.

Custom recording-only voices do not support fallback to text-to-speech. Instead, if any portion of the input text cannot be satisfied by the current recordings, Vocalizer logs an error and the current speak request fails. The logged error specifies the portions of the input text that cannot be satisfied using the normalized input text: it is quite readable but may not exactly match the original input text, and it is romanized for Asian languages. If text-to-speech fallback is required, you should use a Nuance provided voice and just provide ActivePrompt databases and recordings for the application voice talent.

To define a custom voice for recording only playback, use the `makecprvoice` tool from the Vocalizer installation, specifying the template TTS voice, the target voice attributes, and the ActivePrompt databases that define the recordings to use:

```
makecprvoice flags
```

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--language</td>
<td>Vocalizer language name, such as “American English”</td>
</tr>
<tr>
<td>--voice</td>
<td>Custom voice name, such as maureen</td>
</tr>
<tr>
<td>--gender</td>
<td>Custom voice gender, male or female</td>
</tr>
<tr>
<td>--age</td>
<td>(Optional) Custom voice age category, adult or child</td>
</tr>
</tbody>
</table>
For example, the following command defines an American English 8 kHz female voice named Maureen, based on the American English 8 kHz voice named Tom:

```
makecprvoice --language "American English" --voice maureen --gender female --age adult --frequency-hz 8000 --template-voice tom
--activeprompt-db http://myserver/apdb_rp_maureen_alphanum.dat
--activeprompt-db http://myserver/apdb_rp_maureen_carrier.dat
```

**Application development**

Applications use ActivePrompts by loading them into the system (see the next section) and then referencing the ActivePrompts. Those references can be explicit references using the ActivePrompt names (SSML `<prompt>` or the native `<ESC>`\`prompt\` control sequence), or they can be implicit references where the Vocalizer engine automatically searches the ActivePrompt database for each synthesis request, substituting ActivePrompts whenever the normalized input text matches an ActivePrompt’s normalized input text and boundary constraints. Implicit ActivePrompt references can be further controlled by configuring each ActivePrompt database for either fully automatic mode (ActivePrompt database is always consulted) or normal mode (ActivePrompt database is only consulted when explicitly enabled by SSML `ssft-domaintype` or the native `<ESC>`\`domain\` control sequence).

For dynamic content like alphanumeric sequences, dates, times, cardinal numbers, and telephone numbers, it is best to use implicit ActivePrompt references using the normal mode. This mode selectively enables the proper database for each dynamic data type, avoiding conflicts. The input text then becomes a SSML `ssft-domaintype` or native `<ESC>`\`domain\` control sequence to enable the desired types, then the carrier phrase and text to speak. For example, the following SSML example speaks a pre-recorded carrier phrase (`part_code_intro.wav`) with the dynamic portion wrapped within a `<say-as>` element that explicitly specifies the `spell:alphanumeric` type:

```
<s ssft-domaintype="spell:alphanumeric">
<audio src="part_code_intro.wav">The part code is</audio>
<say-as interpret-as="spell:alphanumeric">8jihpey3wy</say-as>
</s>
```

**Note:** Some XML based application development environments block the use of Nuance SSML extensions like `ssft-domaintype`. For those environments, set the `escape_sequence` parameter in the Vocalizer configuration file so you can use a sequence like “\!” instead of the `<ESC>` character (the `<ESC>` character is not allowed in XML documents), then use the native `<ESC>`\`domain\` control sequence such as “\!\domain=spell:alphanumeric\”.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--frequency-hz</code></td>
<td>Custom voice frequency in hertz, such as 8000</td>
</tr>
<tr>
<td><code>--template-voice</code></td>
<td>Template voice name, such as tom. This voice must be an installed Nuance voice, and must match the custom voice’s language and frequency, but does not have to match the gender or age.</td>
</tr>
<tr>
<td><code>--activeprompt-db</code></td>
<td>ActivePrompt database URI or file path. This can be specified multiple times to define a custom voice that relies on multiple ActivePrompt databases.</td>
</tr>
<tr>
<td><code>--user-ruleset</code></td>
<td>User ruleset URI or file path. This can be specified zero or more times to define a custom voice that relies on user rulesets to do text normalization. This is usually just used for rulesets that are specific to the ActivePrompt databases for the new voice, for example, a user ruleset that defines an <code>&lt;ESC&gt;</code>`tn=flightnumber` text normalization type with a matching ActivePrompt database. It is better to load general purpose user rulesets (those that apply to both full TTS and CPR-only voices) using the Vocalizer XML configuration file, so that they apply all the installed voices.</td>
</tr>
</tbody>
</table>
This XML can of course be generated in many alternative ways. For example, in a VoiceXML environment, the VoiceXML expr attribute can be used to specify an ECMAliScript variable that contains the dynamic portion, and an ActivePrompt can be used for the carrier phrase:

```xml
<s ssft-domaintype="spell:alphanumeric">
The part code is
<say-as interpret-as="spell:alphanumeric"><value expr="partcode"/>
</say-as></s>
```

For all dynamic content, it is important to make sure the input format is compatible with Vocalizer. For common data types that are supported by Vocalizer, this is best done by wrapping the dynamic portion within the matching Vocalizer SSML <say-as> or the native <ESC>\tn\ control sequence, checking the Vocalizer Language Supplement to ensure the input format is compatible.

### Loading ActivePrompt databases

When you load custom voices for recording-only playback—voices defined using `makecprvoice`—Vocalizer automatically loads the ActivePrompt databases that were specified when running makecprvoice. (They get added into the Vocalizer file that defines the voice.)

For all other voices and ActivePrompt databases, use the TtsLoadTuningData API function, the SSML `<lexicon>` element, or the `<default_activeprompt_dbs>` XML configuration file parameter to load ActivePrompt databases for runtime use. You can load any number of ActivePrompt databases at runtime. The load order determines the precedence, with more recently loaded ActivePrompt databases having precedence over previously loaded databases. At runtime, Vocalizer only consults ActivePrompt databases that match the current synthesis voice.

For recorded ActivePrompt databases, the recordings are found relative to the URI or file path used to load the ActivePrompt database. For example, if the ActivePrompt database `http://myserver/apdb_rp_tom_alphanum.dat` contains a prompt named `alphanum/f.alpha0` and the database specifies a file suffix of `.wav`, the recording file must be `http://myserver/alphanum/f.alpha0.wav`.

### Specifying pronunciations with user dictionaries

User dictionaries allow you to specify special pronunciations for particular words or strings of characters (for example, abbreviations) and can contain orthographic information (plain text substitution, such as expanding “NaturallySpeaking” to “naturally speaking”) as well as phonetic information. They make it possible to customize the output of the TTS system.

Dictionaries work by substituting a string specified by the user (the destination string or replacement string) for each occurrence of a word in the original input text that matches the source string of a dictionary entry. A user dictionary is a binary file that contains these mappings. You can create a user dictionary using the Nuance Vocalizer Studio graphical user interface, or by using the conversion tool (`dictcpl`) to convert from an input text format dictionary to a binary dictionary.

Vocalizer consults the user dictionary for each individual word in the input text, including multi-word fragments, to check whether to replace the input text with a destination string from the dictionary.
Dictionary substitution rules

Dictionaries work by substituting a string specified by the user (the destination string or replacement string) for each occurrence of a word in the original input text that matches the source string of a dictionary entry. Source strings can contain white space characters only if the control sequence <ESC>\mw\ is used to tag the text fragment in the input text as a multi-word sequence.

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

When no case-sensitive match is found, Vocalizer consults the user dictionary for the all-lower case version of the input text key.

- If the source string contains at least one capital letter, the substitution is case-sensitive and only occurs for an exact match. For example, when the dictionary contains an entry for “DLL”, only the text input key “DLL” will match for that entry.

- If the source string does not contain capital letters, the substitution is case-insensitive. For example, when the dictionary contains an entry for the source string “dll”, text input keys such as “dll”, “Dll”, and “dLL” will all match it. However, if there is a separate source string with capital letters that also matches the input, this capitalized string will take precedence: for example, “DLL” will also match the source string “dll”, but only if there is no separate “DLL” source string in the dictionary.

Note that it is possible for two separate dictionary entries to have source strings that differ only in casing.

The destination string of a dictionary entry can be orthographic or phonetic text. Phonetic strings must be presented using the L&H+ phonetic alphabet. For more information about phonemes, see your Language Supplement.

Additional dictionary substitution rules include:

- When the same source string occurs more than once in the same subheader, the last occurrence determines the destination string.

- When the same source string occurs in different subheaders with different content type (one phonetic and one orthographic), the occurrence in the first subheader determines the corresponding destination string.

- Only complete words can be matched; if the source string in the dictionary is only a substring of a word in the input text, it will not be substituted.

User dictionaries use Vocalizer language codes. See Vocalizer languages on page 179 for a list.
Text format dictionary example

Following is an example of what the input file for a 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$.\'stR+eI.S$n#

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_SZ_STRING

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

Dictionary format for Vocalizer

Textual dictionaries must only be encoded in UTF-8. Note that they may not contain the 3-byte UTF-8 preamble, also known as the UTF-8 BOM or signature.

The general format of textual dictionaries consists of one [Header] label and its properties, and several [SubHeader]-[Data] label couples with their properties and data. Each [SubHeader] describes the expected data properties (such as orthographic or phonetic text) while [Data] describes the actual source string that needs to be replaced with a destination string.

The simplest dictionary consists of one [Header] label and one [Data] label; but while it’s syntactically correct, such a dictionary doesn’t specify any actions.
Here is an example of the format:

[Header]
Language = language_code

[SubHeader]
Content=content_type
Representation=representation_type
Language = language_code

[Data]
source_string separator destination_string

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>language_code</td>
<td>The three-letter code used to identify the language; for example, ENU for American English. The language code is mandatory; it must be specified either in the header, or in each sub-header. Only one language may be used in each dictionary. For a list of available language codes, see Appendix B.</td>
</tr>
<tr>
<td>content_type</td>
<td>The type of content that will be checked against the dictionary. There are two options: EDCT_CONTENT_ORTHOGRAPHIC for orthographic strings, EDCT_CONTENT_BROAD_NARROWS for phonetic strings. The content type determines the representation type. You must specify the content type in each sub-header in the dictionary.</td>
</tr>
<tr>
<td>representation_type</td>
<td>The representation type used for the output: EDCT_REPR_SZ_STRING if the content type is EDCT_CONTENT_ORTHOGRAPHIC, EDCT_REPR_SZZ_STRING if the content type is EDCT_CONTENT_BROAD_NARROWS. You must specify the representation type for each sub-header in the dictionary.</td>
</tr>
<tr>
<td>source_string</td>
<td>The source string that is to be replaced. Multi-word source strings must be enclosed in double quotes (&quot;), and are used at run time only when the matching text is enclosed within a &lt;ESC&gt;\mw\ control sequence.</td>
</tr>
<tr>
<td>separator</td>
<td>The separator between the source string and the destination string. This separator must be a tab character.</td>
</tr>
<tr>
<td>destination_string</td>
<td>One or more words to be used to replace the source string. If this destination string consists of phonetic symbols, it must be preceded by two forward slashes (/!). Multi-word orthographic destination strings may be enclosed in double quotes (&quot;).</td>
</tr>
</tbody>
</table>

Each dictionary may include several sub-header sections; each sub-header may include several data sections; and each data section may include several different
source-destination string pairings. Each source-destination string pair must appear on a separate line within the data section.

Here is an example of a short dictionary:

```
[Header]
Language = ENU

[SubHeader]
Content = EDCT_CONTENT_ORTHOGRAPHIC
Representation = EDCT_REPR_SZ_STRING

[Data]
DLL Dynamic Link Library
Hello Welcome to the demonstration of the American English Text-to-Speech system.
info Information

[SubHeader]
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING

[Data]
addr // '@.dR+Es
```

**Possible errors**

If you experience an error, it may be one of the following:

- Text dictionaries must only be encoded in UTF-8. Note that all characters in the 7-bit US-ASCII range (hex 20 to 7f) are encoded the same way in UTF-8, US-ASCII, Windows-1252, ISO-8859-1, and other formats. So dictionaries which only use character codes in the ASCII range can be encoded in (for example) Windows-1252.

  If a non US-ASCII character is present (for example, ä) and the encoding used is (for example) Windows-1252, then when the dictionary is compiled an error will be returned. Similarly, when the dictionary file is opened in Nuance Vocalizer Studio (see below), a fatal error will be displayed.

- When the content type is EDCT_CONTENT_ORTHOGRAPHIC, the destination strings for this subheader must consist only of orthographic characters. A phonetic string will be interpreted as an orthographic string, and no error is returned.

- When the content type is EDCT_CONTENT_BROAD_NARROWS, the destination strings expected for this subheader must consist only of phonetic characters; an error is returned for any destination string that isn’t preceded by two forward slashes (//).

- When unknown symbols are used in phonetic content, they are ignored.

- Only one language can be specified. If more than one language is specified, no error is returned but the dictionary is ignored.

- The specified language has to be installed. If the language is not installed, no error is returned but the dictionary is ignored.

**Using the dictionary conversion tool (dictcpl)**

The conversion tool (`dictcpl`) can convert a text format dictionary to a binary dictionary. The tool is a console program; you can open a DOS prompt to run the program:

```
dictcpl -o binary-dictionary text-dictionary
```
Options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o</td>
<td>Binary dictionary (output)</td>
</tr>
</tbody>
</table>

For example:

dictcpl -o userdct_enu.bdc enu.txt

Loading user dictionaries

Use the TtsLoadUsrDictEx API function, SSML `<lexicon>` element, or the `<default_dictionaries>` XML configuration file parameter to load user dictionaries for runtime use. Any number of user dictionaries can be loaded at runtime. The load order determines the precedence, with more recently loaded dictionaries having precedence over previously loaded dictionaries.

The runtime will only consult user dictionaries whose language matches the current synthesis language.

User dictionary API calls

From a developer’s point of view, a distinction has to be made between the term *dictionary* and dictionary *instance*. A dictionary is the actual file and its content, while a dictionary instance is the loaded version of a dictionary into memory. A handle to a dictionary instance points to a loaded version. A dictionary instance is always linked to one particular TTS engine instance, but one TTS engine instance can be linked to multiple dictionary instances. In the remaining text, the content should make clear whether dictionaries or dictionary instances are being discussed.

First, a dictionary instance has to be loaded by calling TtsLoadUsrDictEx. This implicitly also enables a dictionary instance for use, with default priority. All loaded dictionaries thus have the same initial priority. In this case, the order of loading determines the priority.

To change priority, call TtsEnableUsrDictEx. The dictionary instance has to be disabled first with TtsDisableUsrDictEx. TtsDisableUsrDictEx can also be used to simply disable (exclude) the dictionary for a lookup. Disabling is not the same as unloading; disabling means that the dictionary instance remains in memory and waits for being enabled again, while unloading a dictionary instance means that the dictionary instance is actually removed from memory.
A typical sequence of dictionary API calls may look like the following:

```c
HTTSPROPERTY hInstance;
TtsOpen (&hInstance, &OpenParams, &instanceData)
...
TTS_USRDICT_DATA dictData1;
HTTSPROPERTY hDctEg1;
memset(&dictData1,0,sizeof(TTS_USRDICT_DATA));
dictData1.uri = "c:\us_english1.dct";

TTS_USRDICT_DATA dictData2;
HTTSPROPERTY hDctEg2;
memset(&dictData2,0,sizeof(TTS_USRDICT_DATA));
dictData2.uri = "c:\us_english2.dct";

TtsLoadUsrDictEx(hInstance,&dictData1,&hDctEg1));
// dictionary 1 is loaded and enabled with
// default priority
TtsProcessEx(hInstance,pSpeakData);
// dictionary 1 is used for lookup
TtsLoadUsrDictEx(hInstance,&dictData1,&hDctEg1);
TtsProcessEx(hInstance,pSpeakData);
// dictionary 2 is looked up, if no entry is
// found dictionary 1 is used for lookup
TtsDisableUsrDictEx(hInstance,hDctEg1);
TtsEnableUsrDictEx(hInstance,hDctEg1,0xF));
// priority change of dictionary 1
TtsProcessEx(hInstance,pSpeakData);
// dictionary 1 is looked up, if no entry is
// found dictionary 2 is used for lookup
TtsUnloadUsrDictEx(hInstance,hDctEg1);

TtsProcessEx(hInstance,pSpeakData);
// dictionary 2 is used for lookup
TtsUnloadUsrDictEx(hInstance,hDctEg2);
TtsProcessEx(hInstance,pSpeakData);
// no dictionaries used
```

**Restrictions on user dictionaries**

You cannot call dictionary functions on a TTS engine instance that is in the state of processing.

**Defining replacement rules with user rulesets**

User 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, user rulesets support any search pattern that can be expressed using regular expressions (for example, multiple words, part of a word).
The user rulesets are applied after SSML parsing (conversion from SSML markup to proprietary control sequence), but before any other text normalization is performed, including user dictionary lookup.

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

A user ruleset is basically a collection of rules; 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/.

User rulesets can be global in scope, or can define a customized text normalization type where the transformations are scoped to input text fragments. These text fragments are labeled as a particular data type by SSML <say-as> or the native <ESC>\tn\ control sequence. Customized text normalization types can augment or override the Vocalizer built-in text normalization types. The remainder of this document refers to customized text normalization type rulesets as “typed rulesets.”

More details on the syntax are described in User ruleset format on page 53.

The TtsLoadTuningData API function, SSML <lexicon> element, or the <default_rulesets> XML configuration file parameter is used to load user rulesets for runtime use. Any number of user rulesets can be loaded at runtime. The load order determines the precedence, with more recently loaded user rulesets having precedence over previously loaded user rulesets.

The rules of a loaded user ruleset are applied only when the active language matches the language that is specified in the header section of the user ruleset.

**Tuning text normalization via user rulesets**

The Regular Expression Text-To-Text (RETTT) engine instance applies the rules of the user rulesets. It is a subcomponent of a text-to-text engine instance.

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 user ruleset processing is the optional transcoding of the input text to the UTF-16 encoding used internally, and SSML parsing (expansion of SSML markup to native control sequences). If the TTS input is provided via the input text callback mechanism, it is first collected entirely, before the rules are applied.

During user ruleset processing, typed rulesets are applied first, then global rulesets. For typed rulesets, the first rule of the most recently loaded active ruleset is applied first to any text fragments within a matching SSML <say-as> or <ESC>\tn\ type. Then, regardless of the effect of this rule, the second rule is applied to that fragment; and so on. The rewriting stops when the last rule of the first loaded ruleset has been applied. In fact, it’s possible that a later rule changes an input string that was already transformed by a previous rule. For global rulesets the processing is the same, but the rules are applied to complete input text rather than certain fragments.

For typed rulesets, the SSML <say-as> or <ESC>\tn\ type gets stripped by the engine as part of processing the typed ruleset, whether or not the typed ruleset actually makes any changes to the contained text fragment. This allows typed rulesets to override Vocalizer built-in text normalization types. However, typed rulesets are free to add a <ESC>\tn\
wrapper around the text as part of their processing (they must use the native control sequence because SSML processing is done before ruleset processing), allowing typed rulesets to delegate portions of their processing to Vocalizer built-in text normalization types, or to augment the Vocalizer built-in text normalization types by adding back the same `\n\n` wrapper.

**User ruleset format**

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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{…}</code></td>
<td>Optional part; the part between <code>{</code> and <code>}</code> can occur once but is not required.</td>
</tr>
<tr>
<td><code>(…)*</code></td>
<td>The part between <code>(</code> and <code>)</code> can occur more than once.</td>
</tr>
<tr>
<td><code>&lt;…&gt;</code></td>
<td>The part between <code>&lt;</code> and <code>&gt;</code> specifies a variable string constant.</td>
</tr>
<tr>
<td>`A</td>
<td>B`</td>
</tr>
</tbody>
</table>

Comment lines use the “#” character as the first non-blank character.

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

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

Here is an example of a working user ruleset. The elements are discussed in more detail in the following sections.

```
[header]
language = ENU

[data]
/NUAN/ --> "Nuance"
/\x{20ac}(\d+).\d*/ --> "$1 euro \$2 cents"
```

**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]"
  (<comment-line>|<blank-line>)|
  <key-definition>+ |
```

Comment lines and blank lines can be inserted anywhere.

Key definitions have the following syntax:

```
key-definition :=
  <key-name> = <key-value><comment>? 
```
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.

A comment can follow the key-value; it lasts until the end of the line.

```
comment :=       #.*$
```

The only currently supported key names are language, charset, and type. This means that key-definition can be expressed semantically as:

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

The language-definition is required for each header. The value is a 3-letter Vocalizer language code, a language group (the first 2 letters of a 3-letter Vocalizer language code followed by a *), or a wildcard * for specifying all languages. 3-letter Vocalizer language codes are also used to specify the language of user dictionaries; see Vocalizer languages on page 179 for a list. Language groups match all the 3-letter Vocalizer language codes with the same first 2 letters. (The "\*" below designates locations where the literal asterisk character "*" is meant instead of a repetition.)

```
language-definition :=
language = (<language-code-list>|<language-group>|\*)<comment>?

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

The charset-definition is optional and specifies the character set used for the encoding of the rules. Currently the character set must be UTF-8.

```
charset-definition :=
   charset = <charset id> <comment>?

charset id :=
   "utf-8"
```

The type-definition is optional and specifies that the ruleset is scoped to a particular text normalization type as accessed via SSML <say-as> or the native <ESC>\tn\ control sequence, otherwise it is global.

```
type-definition :=
type = <type-name><comment>?
```

The type-name is any non-white-space character sequence, and corresponds to the value of the native <ESC>\tn\ control sequence, or the SSML <say-as> interpret-as, format, and detail attributes appended together with a colon delimiter and with trailing colons removed. For example, a user ruleset with a type-name of financial:stocks can be accessed using any of these:

```
<ESC>\tn=financial:stocks
<say-as interpret-as="financial" format="stocks">
<say-as interpret-as="financial:stocks">
```

Data section

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

```
data-section :=
   "[data]"
   ("comment-line"|"blank-line"|"rule")*
```
Comments can also be inserted at the end of a rule and start with a ‘#’ character and span until the end of the line.

A rule has the following syntax:

\[
\text{rule} := \text{<search-spec>} \text{--->} \text{<replacement-spec>} \text{<comment}? \n}
\]

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

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

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

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

Some examples are:

- In the search pattern: non-alphanumeric 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 (for example, 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), and so on.
- Character codes: \x\hh (\hh is the hexadecimal character code, for example, \x1b for Escape), \ooo (ooo is the octal notation, for example, \033 for Escape).
- Perl5 also predefines some patterns like \s (white space) and \d (numeric).

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

**Rule example**

```
/David/ --> "Guru of the month May"
```

Replaces each occurrence of the string “David” with “Guru of the month May”.

**Search-spec**

In general the format of the search spec is:

\[
\text{Search-spec} := \text{delimiter} \text{<regular-expression>} \text{delimiter} \text{<modifier>]*}
\]

The delimiter is usually “/”, but can be any non-whitespace character except digits, backslash (“\”), and “#”. This facilitates the specification of a regular expression that contains “/”, because it eliminates the need to escape the “/”.

\[
\text{<modifier>} := [imsx]
\]

Optional modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Search is case-insensitive.</td>
</tr>
<tr>
<td>m</td>
<td>Let “^” and “$” match even at embedded newline characters.</td>
</tr>
</tbody>
</table>
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 nth capturing subpattern in <search-spec>; for example, $1 denotes the first submatch. A back reference with a number exceeding the total number of submatches in <search-spec>, is translated into an empty string. A literal dollar sign ($) 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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Original text</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>/&lt;NUAN&gt;/ --&gt; &quot;Nuance Communications&quot;</td>
<td>&lt;NUAN&gt;</td>
<td>Nuance Communications</td>
</tr>
<tr>
<td>/(Quack)/ --&gt; ($1)</td>
<td>Quack</td>
<td>(Quack)</td>
</tr>
<tr>
<td>/(Quack)/ --&gt; ($2)</td>
<td>Quack</td>
<td>()</td>
</tr>
<tr>
<td>/\s:-\s\s/ --&gt; &quot;$1ha ha$2&quot;</td>
<td>Where \s matches any white space character, $1 corresponds with the matched leading white space character and $2 corresponds with the matched trailing white space character. This rule rewrites for instance &quot; :-) &quot; into &quot; ha ha &quot;.</td>
<td></td>
</tr>
<tr>
<td>/(r?\n)-{3,} *Begin included message *-{3,}(r?\n)/ --&gt; &quot;$1Start of included message:$2&quot;</td>
<td>---- Begin included message ---- (For example)</td>
<td>Start of included message:</td>
</tr>
<tr>
<td>/\x80 ?(d+).(d(2))d+/ --&gt; &quot;$1 euro $2 cents&quot;</td>
<td>€9.751 (For example)</td>
<td>9 euro 75 cents</td>
</tr>
</tbody>
</table>

### Binary user rulesets

Vocalizer also has the ability to load pre-compiled user rulesets, called binary user rulesets. While Nuance-supplied Vocalizer add-ons may include binary user rulesets, the text format is more compact, and Vocalizer is very efficient at compiling rulesets, so the binary user ruleset compiler is not available for customer use.

### Restrictions on user rulesets

The following restriction applies to user rulesets: markers generated while user rulesets are loaded have source position fields that represent the positions after the user ruleset transformations (not the original text positions).

### Effect of user rulesets on performance

The loading of user rulesets can effect 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 user rulesets, and to test the system with and without the user rulesets to ensure the performance is acceptable.

For example, a character class (such as “[aeiou]”) is more efficient than the equivalent set of alternatives (such as “(a|e|i|o|u)”).

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

**Loading user rulesets**

Use the TtsLoadTuningData API function, SSML `<lexicon>` element, or the `<default_rulesets>` XML configuration file parameter to load user rulesets for runtime use. Any number of user rulesets can be loaded at runtime. The load order determines the precedence, with more recently loaded user rulesets having precedence over previously loaded databases.

The runtime will only apply user rulesets that match the language of the current synthesis voice.

**Language identifier**

Vocalizer includes a built-in language identifier that can be used to automatically switch the synthesis language for blocks of text where the language is unknown by the application. This language identifier can be set to three different scopes via the native API or Vocalizer configuration file settings:

- User-defined, where `<ESC>`\lang=unknown\ control sequences are used to indicate text blocks where the language is unknown.
- Message, where the entire input text is considered to have an unknown language.
- None, where language identification is disabled.

The language identifier uses a statistical model and supports the following languages:

- Basque
- Danish
- Dutch
- English
- French
- German
- Italian
- Norwegian
- Portuguese
- Spanish
- Swedish

For each text region where the language is unknown, Vocalizer does language identification on a sentence-by-sentence basis. For each sentence it first identifies the language, then if necessary it switches the synthesis voice to a voice for that target language. The synthesis voice is restored to the original synthesis voice at the end of the synthesis operation, or if it encounters an `<ESC>`\lang=normal\ control sequence when operating with a user-defined scope.

Vocalizer supports further controlling language identification through language identifier mode and language list parameters that can be set via the native API or Vocalizer configuration file settings.

The mode controls how low-confidence results are handled:
- **rejection** (the default): the language is left unchanged from the previous language.
- **forced-choice**: forces a language switch to the highest confidence language in all cases.

The language list is a list of language codes. The language identifier automatically restricts the matched languages to the ones where voices are installed, this further limits the permissible languages for language identification, and also sets the precedence order for matching languages when they have an equal confidence (first listed has a higher precedence). By default, Vocalizer uses all the languages currently supported by the language identifier.

### Custom voices

Vocalizer offers support for custom voices. Nuance develops a custom voice at the request of a specific customer, often using voice talent contracted by that customer. As part of this process the custom voice font will be given a name that will uniquely identify it for the customer. Each custom voice will go with a specific language (for example, American English).

Vocalizer allows the selection of a voice in three different ways:

- When the engine instance is initialized, such as by setting the szVoice field in the TTS_OPEN_PARAMS structure for the native API. For example:
  ```c
  TTS_OPEN_PARAMS.szVoice = "Elizabeth";
  ```
- Setting the parameter on an existing engine instance, such as by using the TtsSetParams function for the native API. However, this cannot be used to change the voice while synthesis is in progress.
- Using markup, which can switch the voice mid-synthesis: via the SSML `<voice>` element and voice attribute, the Microsoft SAPI 5 `<voice>` tag or the native `<ESC>\voice\` tag.

### Internet fetch support

Vocalizer supports fetching many types of data off web servers or local file systems:

- Input text when using the native API or Speech Server
- Digital audio recordings
- User dictionaries
- User rulesets
- ActivePrompt databases (including digital audio recordings referenced by these databases)

For these fetches, Vocalizer supports the HTTP/1.1 protocol with the following features:

- http:// access for unencrypted requests
- https:// access using the popular OpenSSL open source toolkit (http://www.openssl.org), supporting the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols. Vocalizer uses the OS supplied OpenSSL libraries on Linux (it gets dynamically loaded using dlopen), but uses its own build of the OpenSSL libraries on Windows.
- file:// access for unencrypted local file system access
● Configurable disk cache for http:// and https:// access with HTTP/1.1 compliant caching policies

● Configurable cookie support for http:// and https:// access

● Configurable base URLs for http:// and https:// access

● Configurable timeouts for http:// and https:// access

● Proxy server support for http:// and https:// access

● Configurable file extension to MIME content type mapping rules for file:// access

For information on the configuration options for the configurable features, see Configuring Vocalizer on page 77.

**Note:** Many older HTTP/1.1 proxy servers rely on a HTTP/1.1 extension called the CONNECT method for handling https:// connections, but Vocalizer does not support this due to it having major security flaws and not being an official standard. This CONNECT method is not part of the HTTP/1.1 specification, it was described in an IETF Internet Draft written in 1998 by Ari Luotonens that expired in 1999. The security flaws are described in detail by US-CERT VU#150227 (http://www.kb.cert.org/vuls/id/150227). For proper https:// proxy support, choose a proxy server that acts as a https:// endpoint, decrypting the data and re-encrypting it for delivery to the origin web server.

### Web server configuration

Web servers are responsible for returning a MIME content type for each fetched document, however by default most web servers do not support at least some of the file extensions that are commonly used with Vocalizer. In these cases the web server returns either application/octet-stream (HTTP/1.1 compliant method) or text/plain (not HTTP/1.1 compliant). This leads Vocalizer to incorrectly handle the data, such as incorrectly speaking SSML markup as plain text, failing to do an audio insertion, or failing to load tuning data. To avoid this, all web servers should be configured to return the proper MIME content types as described in the next section.

Web servers also control the caching policy for all fetched data via HTTP/1.1 response headers. By default many web servers do not specify the cache policy for at least some of the file types that are commonly used with Vocalizer. This means Vocalizer must re-fetch the data every time it is used, which can significantly impact the performance of both Vocalizer and the web server. It is thus important to ensure the web server specifies the proper HTTP/1.1 Cache-Control headers to allow Vocalizer to cache the data, such as “Cache-Control: max-age=1440” to allow Vocalizer to cache fetched data for 24 hours (1440 seconds). The exact cache duration should be carefully chosen based on the application’s requirements, particularly for frequently fetched data like digital audio recording insertions: long enough to make web server fetches infrequent, but short enough so that updated data is used by Vocalizer within a reasonable time frame. Vocalizer’s Internet fetch cache does not persist across process restarts, so if a long cache duration is specified and an emergency update is required, stop and re-start the process that Vocalizer runs within.

For more information on optimizing Internet fetch performance, see Limiting delays when internet fetching is used on page 74.

### MIME content types

Vocalizer relies on the following MIME content types to handle fetched data properly. For http:// and https:// access, the web server needs to be configured to return these MIME content types. For local file system access, Vocalizer uses the inet_extension_rules
configuration parameter to map file extensions to MIME content types, by default using the recommended file extensions shown below. Applications are free to use different file extensions (Vocalizer never relies on hardcoded file extensions), but then the web server and/or inet_extension_rules should be configured with the proper MIME content type mapping.

<table>
<thead>
<tr>
<th>MIME content type</th>
<th>Vocalizer data type</th>
<th>Recommended file extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/edct-bin-dictionary</td>
<td>Binary format user dictionary</td>
<td>.bdc or .dcb</td>
</tr>
<tr>
<td>application/synthesis+ssml</td>
<td>SSML input text</td>
<td>.ssml</td>
</tr>
<tr>
<td>application/x-vocalizer-activeprompt-db</td>
<td>ActivePrompt database</td>
<td>(application defined)</td>
</tr>
<tr>
<td>application/x-vocalizer-activeprompt-db;mode=automatic</td>
<td>ActivePrompt database, overriding it to work in automatic insertion mode</td>
<td>(application defined)</td>
</tr>
<tr>
<td>application/x-vocalizer-rettt+text</td>
<td>Text format user ruleset</td>
<td>(application defined)</td>
</tr>
<tr>
<td>application/x-vocalizer-rettt+bin</td>
<td>Binary format user ruleset</td>
<td>(application defined)</td>
</tr>
<tr>
<td>audio/basic</td>
<td>Headerless 8kHz µ-law audio recording</td>
<td>.ulaw or .mulaw</td>
</tr>
<tr>
<td>audio/L16;rate=8000</td>
<td>Headerless 8kHz 16-bit linear PCM audio recording</td>
<td>.L16</td>
</tr>
<tr>
<td>audio/L16;rate=22050</td>
<td>Headerless 22kHz 16-bit linear PCM audio recording</td>
<td>(application defined)</td>
</tr>
<tr>
<td>audio/x-alaw-basic</td>
<td>Headerless 8kHz A-law audio recording</td>
<td>.alaw</td>
</tr>
<tr>
<td>audio/x-wav</td>
<td>Audio recording with a RIFF WAV header containing 8kHz 16-bit linear PCM samples, 8kHz A-law samples, 8kHz µ-law audio samples, or 22kHz 16-bit linear PCM audio samples</td>
<td>.wav</td>
</tr>
<tr>
<td>text/plain</td>
<td>Plain input text using the default Vocalizer character encoding for the current language</td>
<td>.txt</td>
</tr>
<tr>
<td>text/plain; charset=euc-jp</td>
<td>Plain input text using the Japanese EUC encoding</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=iso-8859-1</td>
<td>Plain input text using the ISO-8859-1 (Latin 1) encoding</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=shift-jis</td>
<td>Plain input text using the Japanese Shift-JIS encoding</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=utf-16</td>
<td>Plain input text using the Unicode UTF-16 encoding (the best recommended encoding)</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=utf-8</td>
<td>Plain input text using the Unicode UTF-8 encoding (another recommended encoding)</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=windows-1252</td>
<td>Plain input text using the Windows-1252 encoding</td>
<td>(application defined)</td>
</tr>
<tr>
<td>text/plain; charset=charset</td>
<td>Plain input text with another character encoding as specified by charset. See TTS_SPEAK_DATA on page 104 for more information on using alternate character encodings.</td>
<td>(application defined)</td>
</tr>
</tbody>
</table>

**Error and diagnostic logs**

At runtime, Vocalizer reports error log messages for problems encountered by the TTS engine. While all Vocalizer APIs return error codes that give a high-level indication of the type of failure, it is important to monitor the error logs. The error logs provide detailed...
information about the errors returned by the API so they can be quickly resolved (such as the specific URI, line, and column containing a SSML parse error). You should also check the error logs for warnings about application problems that don’t trigger a fatal error but may lead to unexpected behavior and should be corrected (such as an SSML say-as type that is not supported).

Vocalizer can optionally be configured to report diagnostic information as well. If you file a support incident with Nuance Technical Support, they may ask you to enable diagnostic logging so Nuance can analyze and resolve your issue.

These error and diagnostic logs are written to the following destinations:

- Vocalizer error and diagnostic log files
- Operating system’s log: Windows Application Log or Unix syslog
- (Native API only) Application provided logging callbacks, TTS_LOG_ERROR_CB and TTS_LOG_DIAGNOSTIC_CB
- (Speech Server only) Nuance Watcher and the Nuance Management Station

Vocalizer log files

Vocalizer log files are enabled by default, but can be disabled via the log_file_enabled parameter in the Vocalizer configuration file. By default they are written to...

The log file is written in an XML format. It starts with the usual XML declaration, includes a comment to indicate this is a Vocalizer log file, then provides an internal DTD that describes the file format. The actual log messages appear within an <xml_log> root element, with <error>, <entry>, and <event> elements (one per line) that report the log messages.

- <error> reports an error message, warning message, or informational message.
- <entry> reports a diagnostic message. These are off by default and only present when diagnostic logging is enabled (log_level is set to 2 or more in the Vocalizer configuration file). Diagnostic logging can significantly affect performance and should only be enabled for diagnosing problems as requested by Nuance Technical Support.
- <event> reports an application event message as documented in Application event logs on page 65. These are off by default and only present when diagnostic logging is enabled (log_level is set to 5 or more in the Vocalizer configuration file). The separate application event log file should be used to obtain event logs, this redundant information is only included for Nuance support purposes.

Here is a log file example:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE xml_log[<!ELEMENT xml_log (entry|error|event)*><!ELEMENT entry (t, mi, mn, mv, thr, id, sid, l, (tm | (bl, bs, bm)))><!ELEMENT error (t, mi, mn, mv, thr, id, sid, s, eid, tm, mp*)><!ELEMENT event (t, thr, id, sid, evt, tm, mp*)><!ELEMENT t (#PCDATA)><!ELEMENT mi (#PCDATA)><!ELEMENT mn (#PCDATA)><!ELEMENT mv (#PCDATA)><!ELEMENT thr (#PCDATA)><!ELEMENT id (#PCDATA)]>``
Note: The root <xml_log> element only gets closed (</xml_log>) when the log file is closed after a rollover, or if Vocalizer is shut down normally. If you wish to parse the log files using an XML parser, make sure the parser can tolerate an unclosed <xml_log>, or make sure you look for the </xml_log> and append it in-memory when necessary before calling the XML parser.

The log entries are composed of XML elements within the <error>, <entry>, and <event> elements. The names are kept short to minimize log file disk space use.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Timestamp</td>
</tr>
<tr>
<td>mi</td>
<td>Vocalizer module ID number</td>
</tr>
<tr>
<td>mn</td>
<td>Vocalizer module name</td>
</tr>
<tr>
<td>mv</td>
<td>Vocalizer module version number</td>
</tr>
<tr>
<td>thr</td>
<td>Operating system thread ID</td>
</tr>
<tr>
<td>id</td>
<td>Vocalizer instance ID number (hexadecimal)</td>
</tr>
<tr>
<td>sid</td>
<td>Session identifier as set by TtsSessionStart for the native API, otherwise empty</td>
</tr>
<tr>
<td>s</td>
<td>&lt;error&gt; elements only: the error severity</td>
</tr>
<tr>
<td></td>
<td>▪ CRITICAL (critical error that affects all engine instances)</td>
</tr>
<tr>
<td></td>
<td>▪ SEVERE (severe error that affects one engine instance)</td>
</tr>
<tr>
<td></td>
<td>▪ WARNING (non-fatal application error)</td>
</tr>
<tr>
<td></td>
<td>▪ INFO (informational notice like the list of installed voices at startup)</td>
</tr>
<tr>
<td>eid</td>
<td>&lt;error&gt; elements only: the error ID number, corresponding to the &quot;num&quot; field in install_path\doc\VocalizerLogStrings.enu.xml</td>
</tr>
<tr>
<td>l</td>
<td>&lt;entry&gt; elements only: log_level value where that message starts appearing</td>
</tr>
<tr>
<td>evt</td>
<td>&lt;event&gt; elements only: the event ID number</td>
</tr>
</tbody>
</table>
System logging

Vocalizer logs errors to the operating system’s log by default, but that can be disabled via the log_to_system_log parameter in the Vocalizer configuration file. This makes it easier to monitor Vocalizer as part of an overall system.

On Windows, the system log is the Windows Application Log, which can be viewed using Windows Event Viewer. On Unix, Vocalizer logs to the system log using the syslog() function call, with the Unix syslogd daemon choosing the log destination(s).

Diagnostic and event messages are never logged to the system log.

Application provided logging callbacks

For the native API, when the log_cb_enabled configuration parameter is true (the default setting), the following entries are logged:

- Vocalizer logs errors to the application provided logging callback TTS_LOG_ERROR_CB when that callback is not NULL.
- Vocalizer logs diagnostic messages to the application provided logging callback TTS_LOG_DIAGNOSTIC_CB when that callback is not NULL and the log_level parameter in the Vocalizer log file is set to 2 or greater (by default it is 0).
- Vocalizer logs event notifications to the application provided logging callback TTS_LOG_EVENT_CB when that callback is not NULL. (For more information on event notifications see Application event logs on page 65.)

Applications can use this feature to integrate Vocalizer log messages within an application defined logging system. See TTS_LOG_ERROR_CB on page 142, TTS_LOG_DIAGNOSTIC_CB on page 146, and TTS_LOG_EVENT_CB on page 144 for details.

Nuance Watcher provides an equivalent to this for Speech Server deployments. There is no equivalent to this in the Microsoft SAPI 5 API.

Nuance Watcher

When Vocalizer is used via Speech Server, Vocalizer errors are reported to Nuance Watcher, a tool for monitoring C/C++ processes that is included with Speech Server. You can use the Nuance Watcher in one of two ways:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tm</td>
<td>Text message: the error description (corresponding to the &lt;error&gt; content in install_path\doc\VocalizerLogStrings.enu.xml), diagnostic message, or event name</td>
</tr>
<tr>
<td>mp</td>
<td>&lt;error&gt; and &lt;events&gt; elements only: message pair that further describes the event or error, where the “key” attribute gives the name of a variable and the content of the element gives the value of the variable. For example, a key of “URI” with a value of “<a href="http://myserver/input.ssml%E2%80%9D">http://myserver/input.ssml”</a> could be used to indicate a specific URI that triggered the error.</td>
</tr>
<tr>
<td>bl</td>
<td>&lt;entry&gt; elements only: label for a logged block of binary data, rarely used</td>
</tr>
<tr>
<td>bs</td>
<td>&lt;entry&gt; elements only: size of a logged block of binary data, rarely used</td>
</tr>
<tr>
<td>bm</td>
<td>&lt;entry&gt; elements only: base 64 encoded logged block of binary data, rarely used</td>
</tr>
</tbody>
</table>
As part of the Nuance Management Station. The Management Station is a tool for configuring, implementing, and managing all aspects of a Nuance voice network. The Management Station makes the functions of the Nuance Watcher available transparently.

As a standalone process. Working with the Nuance Watcher as a standalone process requires you to issue commands directly to the watcher service using telnet, HTTP, or SNMP.

See the Nuance Management Station Administrator’s Guide for information about using Management Station. See the Watcher User’s Guide for information about using Nuance Watcher in a standalone configuration.

Customizing error messages

For error logging, the Vocalizer engine reports all errors using an error ID that is mapped to an error message and severity level using an XML file. This file is configured using the log_file_errormap parameter in the Vocalizer configuration file, by default install_path\doc\VocalizerLogStrings.enu.xml within the Vocalizer installation. Customers can use this to customize error reporting:

- Suppress error or warning messages that are expected for a specific deployment
- Customize error messages, such as translating them to another language or adjusting them to be more specific to a deployment
- Customize error severity levels

To do so, you can either edit the default file, or you can make a copy, edit it, and then change log_file_errormap. That file is an XML file with <error> elements, one for each error Vocalizer can log. Each <error> indicates the following:

- The “num” attribute indicates the error ID number as logged by the Vocalizer engine. These should not be changed.
- The “severity” attribute indicates the error severity:
  - 1 for CRITICAL errors
  - 2 for SEVERE errors
  - 3 for WARNING messages
  - 4 for INFO (informational) messages
  - 5 to completely disable that error (will not be reported)
- The content of the <error> element gives the error message. This can be changed to any descriptive text you like, including a Unicode string if you wish to translate the message to another language. However, note that the default file uses the ISO-8859-1 character encoding in its XML declaration: if you wish to use Unicode, change that to utf-8 or utf-16, then use a text editor to save the file in the matching encoding instead of ISO-8859-1.

Here is an example of an error entry:

<error num="53009" severity="3">Concatenated prompt recording voice cannot find enough recorded ActivePrompts to fully speak the input text</error>
Suppressing sensitive data

You can use the TTS_SECURE_CONTEXT_PARAM parameter in TtsSetParams on page 131 to suppress logging sensitive data to the error and diagnostic logs.

You can also set “secure_context” via an SSML <meta> element to affect the current speak request only. For example:

```xml
<meta name="secure_context" content="1"/>
```

Application event logs

At runtime, Vocalizer writes event logs to report TTS engine statistics for application tuning and capacity planning. Vocalizer controls the event log files, but you control other aspects with XML configuration file parameters. When using Vocalizer via the Speech Server, all Vocalizer events also appear in the Speech Server call logs. When using the native API, applications can also register a callback to receive event notifications, allowing the application to write events to an application specific location or use them for online monitoring and reporting.

These logs are similar in format, content, and purpose to Nuance Recognizer 9 call logs, but the files are similar to Vocalizer error and diagnostic logs, that is, at most there are two event log files, rather than a hierarchical directory tree of separate event logs.

For more information about call logs, see the Nuance Recognizer Reference.

Event log file format

Event logs are encoded as UTF-8. For storage and transport purposes (for example, storing in CVS and transporting via ftp), you should treat these files as binary.

Application provided logging callbacks

For the native API, Vocalizer logs event notifications to the application provided logging callback TTS_LOG_EVENT_CB when that callback is not NULL and the log_cb_enabled configuration parameter is true. (This is enabled even if the event_log_file_enabled configuration parameter is false, that parameter only controls the Vocalizer written event log files.) Applications can use this feature to integrate Vocalizer event notifications within an application defined logging system. See TTS_LOG_EVENT_CB on page 144 for details.

The Speech Server provides an equivalent to this where this information is written to the Speech Server call log, then is uploaded to the Management Station. There is no equivalent to this in the Microsoft SAPI 5 API.

Input text log files

Vocalizer supports logging the full input text for speak requests, important for analyzing the text spoken by Vocalizer and reviewing it for TTS tuning purposes. It does so by logging the input text to a separate XML format input text log file, then logging a NVOCinpt event to the event log to correlate that information with a speak request within the main event log. (See NVOCinpt—input text on page 71.) This is similar to Nuance Recognizer waveform logging. Like Nuance Recognizer waveform logging, Vocalizer configuration parameters can be used to limit the number of simultaneous speak requests where the input text is being logged or to completely disable input text...
capture. When the secure_context parameter is set, input text capture is automatically
disabled for the corresponding speak requests.

To configure input text logging, set the event_log_input_text_max_capture configuration
parameter to allow disabling input capture completely, or throttling input text capture to
a limited number of simultaneous speak requests to limit the performance impact and the
logged data volume. Set the event_log_input_text_file_base_name configuration
parameter to specify the XML file name used for logging input texts. Set the
event_log_input_text_file_max_size configuration parameter to specify the maximum
size for the input text log file. (See Diagnostic and error logging parameters on page 86 for
details of these configuration parameters.)

The input text log file is a UTF-8 encoded XML file, where each input text is written using
an <entry> element with a unique “id” attribute that is generated using the current date,
timestamp to the millisecond, and session ID, similar to how Nuance Recognizer
waveform capture file names are generated. By generating the “id” attribute in this
manner, the ID will be unique for at least that single system, and if the session IDs
specified by the application are unique then the ID will be unique across the entire
deployment. The content of the <entry> element is the input text.

The logged input text is the original plain text or SSML input, without any modifications
except for transcoding to UTF-8 for logging purposes. For Microsoft SAPI 5 based
applications, Microsoft SAPI 5 XML markup is parsed by SAPI 5 before it is passed to
Vocalizer, so the captured input text will be plain text with Vocalizer control sequences,
rather than the original XML markup.

Note: For SSML with a encoding specified in the XML declaration, that encoding is not
updated to indicate the input log file's encoding of UTF-8. Before playing back that SSML
for analysis, make sure you update the encoding attribute to specify encoding="utf-8" (or
simply remove the encoding attribute).

The NVOCinpt events that cross-reference these entries report the MIME content type for
the input text (MIME token), a reference to the input text (TXID token, empty if input text
capture logging is disabled in the configuration file or for a secure context), and the text
input size in bytes (TXSZ token). See NVOCinpt—input text on page 71 for more
information.

Example event log entries:
TIME=20090318155132858|CHAN=cmdline:user1|EVNT=NVOCinpt|MIME=application/synthesis+ssml|TXID=NUAN-20090318155132858-cmdline:user1-txt|TXSZ=196|UCPU=46|SCPU=0
TIME=20090318164254632|CHAN=cmdline:user1|EVNT=NVOCinpt|MIME=text/plain;charset=utf-16|TXID=NUAN-20090318164254632-cmdline:user1-txt|TXSZ=10|UCPU=0|SCPU=0
Example input text log file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Nuance Vocalizer Input Text Log File -->
<!DOCTYPE input_text[<!ELEMENT input_text (entry)*><!ELEMENT entry (#PCDATA)><!ATTLIST entry id ID #REQUIRED]>]
<input_text>
  <entry id="NUAN-20090318155124436-cmdline:user1-txt">
    &lt;?xml version=&quot;1.0&quot;?&gt;
    &lt;!DOCTYPE speech [&lt;ELEMENT speech (#PCDATA)&gt;&lt;!ELEMENT speech (
    xmlns="http://www.w3.org/2001/10/synthesis"&gt;
    &lt;say xml:lang=&quot;en-US&quot; version=&quot;1.0&quot; xmlns=http://www.w3.org/2001/10/synthesis&quot;&gt;
      Your next payment is due on &lt;say-as interpret-as=&quot;date&quot; 20090115&lt;/say-as&gt;&lt;/speech&gt;
    </entry>
  <entry id="NUAN-20090318164254632-cmdline:user1-txt">hello</entry>
</input_text>
```

**Event logs—merging distributed logs**

A platform integration might have several different event log streams depending which products are being used. For example, each of the following components can write an event log (or call log):

- VoiceXML browser
- Nuance Dialog Modules
- Speech recognition service (for example, Nuance Recognizer)
- Audio output service (text-to-speech) engine

To merge the logs, you can set “SWI.appsessionid” and “SWI.appstepid” parameters via SSML `<meta>`; setting both of these parameters generates a NVOCapps event. Nuance packaged applications use these identifiers to merge application logs, Nuance Recognizer logs, and Vocalizer logs to enable analysis, tuning, and reporting. The parameters are typically set several times during a session to provide information about logical steps within the application. For example:

```xml
<meta name="SWI.appsessionid" content="431cc972eaa41c1a22e99ac59f5e4fa4"/>
<meta name="SWI.appstepid" content="3"/>
```
Event codes and tokens

Below is a sample log record. All records contain token/value pairs separated by vertical bars “|”. Tokens are always uppercase:

```plaintext
TIME=20081118110031601|CHAN=|EVNT=NVOCfmt|ENCDF=|UCPU=0|SCPU=0
TIME=20081118110031757|CHAN=|EVNT=NVOClock|LUSED=0|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0
TIME=20081118110031851|CHAN=02031558|EVNT=NVOCliiss|LUSED=1|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifst|URI=http://myserver/cpr_recs/en.us/samanth a.8kHz/apdb_rp_samantha_cpr_samantha.dat|PROP=|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifnd|URI=http://myserver/cpr_recs/en.us/samanth a.8kHz/apdb_rp_samantha_cpr_samantha.dat|FRST=SUCCESS|MIME=application/octet-stream|SIZE=1720 |DSRC=http|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCsyst|LANG=American English|VOIC=Samantha|VMDL=full_encryptf8|FREQ=8000|PVER=5.2.0|LVER=5.2.0.8278|VVER=5.2.0.7151 |UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifst|URI=alphanum/f.alpha1.wav|PROP/inet.urlBase=http://myserver/cpr_recs/en.us/samantha.8kHz/apdb_rp_samantha_cpr_samantha.dat|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifnd|URI=http://myserver/cpr_recs/en.us/samanth a.8kHz/alphanum/f.alpha1.wav|FRST=SUCCESS|MIME=audio/x-wav|SIZE=17658|DSRC=http|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCactp|APID=cpr_samantha::alphanum/f.alpha1|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudf|SAMP=4096|FREQ=8|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudn|SAMP=4096|FREQ=8|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudn|SAMP=2128|FREQ=8|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudn|SAMP=2583|FREQ=8|UCPU=0|SCPU=0
TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudn|SAMP=1667|FREQ=8|UCPU=0|SCPU=0
TIME=20081118110031882|CHAN=02031558|EVNT=NVOCsyst|LANG=American English|VOIC=Samantha|VMDL=full_encryptf8|FREQ=8000|PVER=5.2.0|LVER=5.2.0.8278|VVER=5.2.0.7151 |UCPU=0|SCPU=0
```

Every log record begins with the TIME token, which has a value with the format: YYYYMMDDhhmmssmmm.

Another token that appears in every record is EVNT, which indicates the event that has occurred. Event codes logged by Vocalizer are always preceded with the uppercase NVOC prefix and then spelled with lowercase letters. For example, NVOCsyst is the code for “Synthesis Start.” (The codes are described on Standard events and tokens on page 68.) User-defined events should not start with NVOC to avoid interfering with Vocalizer events.

Syntax and size of log records

Each line (record) of the event log file describes a single event. The lines are terminated by newline control sequences; the maximum size of a line is 10 kilobytes.

Within each record the format is:

- Event codes and tokens are separated by the “|” character.
- If there is a “|” character in data, it is quoted by inserting an additional “|” character to form the sequence “||”.
- Tokens are separated from their values by the “=” character, and a single token can have more than one “=” character.

Nuance Vocalizer for Network 5.0

Nuance Proprietary

Developer’s Guide
Suppressing sensitive data

You can use the TTS_SECURE_CONTEXT_PARAM parameter in TtsSetParams on page 131 to suppress logging sensitive data to the event log. All affected events report a “SECURE=1” token and substitute the string "_SUPPRESSED" where sensitive data would otherwise appear.

You can also set “secure_context” via an SSML <meta> element to affect the current speak request only. For example:

<meta name="secure_context" content="1"/>

Timing accuracy

All time-related codes log times in millisecond units (unless specified otherwise), and are accurate to within 0.01 second.

Tokens used for every event

The first entries in each log record are TIME, CHAN, and EVNT; the last entries are UCPU and SCPU.

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>System time when the event occurred, in the following format (accurate to within 1/100th of a second): YYYYMMDDhhmmssmmm</td>
</tr>
<tr>
<td>CHAN</td>
<td>A unique session identification name provided in calls to the TtsSessionStart function.</td>
</tr>
<tr>
<td>EVNT</td>
<td>Prefix used for event codes. Limited to 8 characters; longer names are truncated. All Vocalizer event codes are prefixed with &quot;NVOC&quot;.</td>
</tr>
<tr>
<td>UCPU</td>
<td>The current running value of “user” CPU time consumed from the start of synthesis. This value is reported in milliseconds, accurate to within 1/100th of a second.</td>
</tr>
<tr>
<td>SCPU</td>
<td>The current running value of “system” CPU time consumed from the start of synthesis. This value is reported in milliseconds, accurate to within 1/100th of a second.</td>
</tr>
</tbody>
</table>

Standard events and tokens

The following list shows groups of standard Vocalizer event codes. Following this list are descriptions of each event code and its tokens. Production sites might encounter additional codes and tokens not described here due to events inserted by the application.

NVOCapps—application session
NVOCaudf—first audio
NVOCaudn—next audio
NVOClise—license end
NVOCliss—license refused
NVOCliss—license start
NVOClock—license lock
NVOCunlo—license unlock
NVOCactp—ActivePrompt was used

This event is written when an ActivePrompt is used during synthesis. These events are written in the order in which the ActivePrompts are matched (highest to lowest precedence across the input text), not the order in which they appear in the output audio stream:

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>APID</td>
<td>String</td>
<td>ActivePrompt ID of the form domain::prompt.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCactp|APID=cpr_samantha::alphanum/f.alpha1|UCPU=0|SCPU=0

NVOCapps—application session

This event is written when the application provides a session and step identifier using SSML <meta>. These identifiers can be set by Nuance packaged applications; they are used to merge application logs, Nuance Recognizer logs, and Vocalizer logs and therefore enable analysis, tuning, and reporting. The event has the following tokens:

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESN</td>
<td>String</td>
<td>Identifier that uniquely identifies the session. This is also known as the app-session ID.</td>
</tr>
<tr>
<td>STEP</td>
<td>String</td>
<td>A counter of synthesis events; it generally increments once for each synthesis.</td>
</tr>
</tbody>
</table>

TIME=20081118115242132|CHAN=02031558|EVNT=NVOCapps|SESN=431cc972ea41c1a22e99ac59f5e4fa4|STEP=3|UCPU=46|SCPU=0

NVOCaudf—first audio

This event indicates the first audio packet for a synthesis operation. For streaming synthesis operations, the time duration from submitting a synthesis request (NVOCsyst) to the time delivering the first audio packet (NVOCaudf) is a major factor in the end-user perceived latency, a critical factor to measure to ensure good response times.
The event has the following tokens:

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMP</td>
<td>Integer</td>
<td>Number of samples in the audio packet.</td>
</tr>
<tr>
<td>FREQ</td>
<td>Integer</td>
<td>Sampling frequency in kHz.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCaudf|SAMP=4096|FREQ=8|UCPU=0|SCPU=0

**NVOCaudn—next audio**

This event indicates an audio packet for a synthesis operation after the first audio packet. See **NVOCaudf—first audio** on page 69 for the list of tokens.

TIME=20081118110031601|CHAN=|EVNT=NVOCaudn|SAMP=4096|FREQ=8|UCPU=0|SCPU=0

**NVOCfrmt—file format**

This event indicates the file format for Vocalizer-written event log files; it is never delivered to the application event logging callback.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCD</td>
<td>String</td>
<td>File encoding, currently always UTF-8.</td>
</tr>
</tbody>
</table>

TIME=20081118110031601|CHAN=|EVNT=NVOCfrmt|ENCD=UTF-8|UCPU=0|SCPU=0

**NVOCifnd—internet fetch end**

This event indicates an internet fetch has ended:

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>String</td>
<td>URI that was fetched. If the fetch was successful, this will be an absolute URI.</td>
</tr>
<tr>
<td>FRST</td>
<td>String</td>
<td>Fetch result: SUCCESS if the fetch succeeded; otherwise a brief error description.</td>
</tr>
<tr>
<td>MIME</td>
<td>String</td>
<td>MIME content type. Empty string if the fetch failed.</td>
</tr>
<tr>
<td>SIZE</td>
<td>Integer</td>
<td>Size in bytes. 0 if the fetch failed.</td>
</tr>
<tr>
<td>DSRC</td>
<td>String</td>
<td>Data source:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ empty string if the fetch failed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ “file” if it was a local file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ “http” if it was fetched from a web server</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ “cache” if it was loaded from a cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ “validated” if it was loaded from a cache, but only after validating the cached copy with the web server.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifnd|URI=http://myserver/cp r_recs/en.us/samantha.8kHz/alpha num/f.alpha1.wav|FRST=SUCCESS|MIME=audio/x-wav|SIZE=17658|DSRC=http|UCPU=0|SCPU=0
**NVOCifst—internet fetch start**

This event indicates an internet fetch has started.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>String</td>
<td>URI being fetched; may be a relative URI.</td>
</tr>
<tr>
<td>PROP</td>
<td>String</td>
<td>Semicolon-separated list of internet fetch properties, where each property is of the form <code>property=value</code>.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCifst|URI=alphanum/f.alpha1.wav|PROP=inet.urlBase=http://myserver/cpr_recs/en.us/samantha.8kHz/apdb_r p_samantha_cpr_samantha.dat|UCPU=0|SCPU=0

**NVOCinpt—input text**

This event provides the input text for the speak request. NVOCinpt is logged at the end of each speak request due to the possibility of SSML markup specifying a secure context. If an error occurred or the synthesis request was stopped, NVOCinpt reports the input obtained prior to the error or interruption.

**Note:** In the Speech Server environment, the Speech Server logs a similar event called RSdata.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIME</td>
<td>String</td>
<td>MIME content type for the input text</td>
</tr>
<tr>
<td>TXID</td>
<td>String</td>
<td>Text ID within the separate input text logging XML file (&lt;entry&gt; id attribute), as configured using the event_log_input_text_file_base_name configuration file parameter. If input text logging is disabled in the configuration file, this token has an empty value.</td>
</tr>
<tr>
<td>TXSZ</td>
<td>Integer</td>
<td>Input size in bytes, the raw input prior to any processing. The entry in the input text logging XML file might have a different size because that file contains the input text after transcoding to UTF-8. If an error occurred or the synthesis request was stopped, this reports the input obtained prior to the error or stopping.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCinpt|MIME=application/synthesis+ssml|TXID=NUAN-20081118110031866-02031558-txt|TXSZ=196|UCPU=46|SCPU=0
NVOClise—license end

NVOCliss and NVOClise events indicate the licenses used at the beginning and end of a synthesis operation. NVOClise is triggered at the end of synthesis, and the tokens describe the count of licenses in use prior to freeing the license.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFEAT</td>
<td>String</td>
<td>License features. A comma-separated list showing which features are associated with the license. This starts with either “tts” for a full TTS license or “cpr” for a CPR-only license, with the remaining tokens for this event reflecting the licenses for just that type. Then it may be followed by one or more modifier features, such as “unthrottled” for a license with audio throttling disabled.</td>
</tr>
<tr>
<td>LMAX</td>
<td>Integer</td>
<td>License maximum. The maximum number of available licenses, licenses actually checked out and available for use by Vocalizer.</td>
</tr>
<tr>
<td>LUSED</td>
<td>Integer</td>
<td>Licenses used, the current number of synthesis operations.</td>
</tr>
<tr>
<td>OMAX</td>
<td>Integer</td>
<td>Overdraft maximum. The number of available license ports, not including overdraft ports.</td>
</tr>
</tbody>
</table>

TIME=2008111811031882|CHAN=02031558|EVNT=NVOClise|LUSED=1|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0

NVOClisr—license refused

The NVOClisr event indicates a synthesis operation failed because there was no license available; the tokens describe the count of licenses currently in use. See NVOClise—license end on page 72 for the list of tokens.

TIME=2008111811031882|CHAN=02031558|EVNT=NVOClisr|LUSED=100|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0

NVOCliss—license start

The NVOCliss and NVOClise events indicate the licenses used at the beginning and end of a synthesis operation. NVOCliss is triggered at the start of a synthesis, and the tokens describe the count of licenses in use after incrementing for the new license. See NVOClise—license end on page 72 for descriptions of the tokens.

TIME=2008111811031881|CHAN=02031558|EVNT=NVOCliss|LUSED=1|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0

NVOClock—license lock

NVOClock logs the time when a license is checked out (locked) during Vocalizer initialization and shows which license features are used. See NVOClise—license end on page 72 for the list of tokens.

TIME=2008111811031757|CHAN=|EVNT=NVOClock|LUSED=0|LMAX=100|OMAX=90|LFEAT=tts|UCPU=0|SCPU=0

NVOCsynd—synthesis end

This event is logged at the end of synthesis. If the secure context parameter was set for the synthesis, the event includes a SECURE=1 token even though this event doesn’t contain any sensitive information that needs to be suppressed. This is done so an application can
detect if a secure context is in place, suppressing its own logging of sensitive speak request information.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPT</td>
<td>Integer</td>
<td>Input size in bytes; the raw input prior to any processing. If an error occurred or the synthesis request was stopped, this token reports the input obtained prior to the error or stopping.</td>
</tr>
<tr>
<td>DURS</td>
<td>Integer</td>
<td>Output duration in ms. If an error occurred or the synthesis request was stopped, this token reports the audio returned prior to the error or stopping.</td>
</tr>
<tr>
<td>RSTT</td>
<td>String</td>
<td>Result status:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “error” if an error occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “ok” if synthesis succeeded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “stop” if the application stopped the synthesis operation</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCsyst|INPT=7|DURS=2333|RSTT=ok|UCPU=0|SCPU=0

**NVOCsyst—synthesis start**

This event is logged at the beginning of synthesis. The language and voice information is for the primary synthesis language and voice. (NVOCsyst does not report languages and voices used due to mid-synthesis voice switches; see **NVOCsysw—synthesis switch** on page 73.)

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANG</td>
<td>String</td>
<td>Language name.</td>
</tr>
<tr>
<td>VOIC</td>
<td>String</td>
<td>Voice name.</td>
</tr>
<tr>
<td>VMDL</td>
<td>String</td>
<td>Voice model.</td>
</tr>
<tr>
<td>FREQ</td>
<td>Integer</td>
<td>Sampling frequency in kHz.</td>
</tr>
<tr>
<td>PVER</td>
<td>String</td>
<td>Vocalizer product version number.</td>
</tr>
<tr>
<td>LVER</td>
<td>String</td>
<td>Vocalizer language version number.</td>
</tr>
<tr>
<td>VVER</td>
<td>String</td>
<td>Vocalizer voice version number.</td>
</tr>
</tbody>
</table>

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCsyst|LANG=American English|VOIC=Samantha|VMDL=full_encryptf8|FREQ=8000|PVER=5.2.0|LVER=5.2.0.8278|VVER=5.2.0.7151|UCPU=0|SCPU=0

**NVOCsysw—synthesis switch**

This event is logged when a mid-synthesis voice switch occurs, where the event tokens report the new voice. See **NVOCsyst—synthesis start** on page 73 for the list of tokens.

TIME=20081118110031866|CHAN=02031558|EVNT=NVOCsysw|INPT=7|DURS=2333|RSTT=ok|UCPU=0|SCPU=0
Performance and scalability tips

Operating system restrictions

Each instance of Vocalizer requires a number of file handles which are used for accessing, among others, the voice database. Some operating systems, such as Microsoft Windows, have a default limit of file handles per process. If you have a very large number of Vocalizer instances, or if the application uses many file handles of its own, the application can run out of file handles. In order to avoid this problem, Nuance recommends that you set the amount of file handles to an appropriate value. For Microsoft Windows, you can set this value by having your application issue the _setmaxstdio C-runtime call. See your compiler or operating system manual for more information.

For Unix, the number of file handles can be increased by means of the limit/ulimit commands. For more information about these commands, refer to the man pages or to the compiler manual.

Optimal audio buffer size

The audio buffer size is an important factor for minimizing latency (time to first audio) and avoiding underruns, where larger buffer sizes are more efficient for CPU use but increase latency and the risk of underruns. A good starting point is a buffer that is big enough for half a second of audio rounded up to the nearest multiple of 1024 bytes (1K):

- 4096 bytes for an 8 kHz sampling rate voice for µ-law or A-law audio output
- 8192 bytes for an 8 kHz sampling rate voice for linear 16-bit PCM audio output
- 22528 bytes for a 22kHz sampling rate voice

For the native API, this buffer is provided by the application and is specified via the return value of the TTS_DEST_CB typed Destination callback function.

For Speech Server based applications, the audio buffer size used for synthesis within Speech Server is controlled by a NSSserver.cfg configuration parameter (by default 8192 bytes), but the RTP packet size is generally smaller and is controlled by the negotiated RTP stream parameters.

For SAPI 5 based applications, this is determined by SAPI 5 and cannot be changed.

Limiting delays when internet fetching is used

When content such as input texts, user dictionaries, user rulesets, and ActivePrompt databases are located on a Web server, this can result in delays when the content is fetched for the first time. Since the internet fetch library uses a (configurable) cache, the download time will be minimal if the cache has been configured well (big enough, reasonable cache entry expiration time), the web server is configured to support caching all the data (specifies HTTP/1.1 caching parameters like maxage), and the cache has been warmed up.
To warm up the cache, the application can perform a number of dummy speak requests. For input texts, the content will already be cached before the first audio packet is delivered. So during the warmup, the application can stop the synthesis request after the first audio packet to speed up the warmup.

Audio content specified via the SSML <audio> tag is always fetched on message (normally a sentence) boundaries, but not necessarily before the first audio packet is delivered. User dictionaries, user rulesets, and ActivePrompt databases can be loaded and unloaded to obtain a copy in the cache without consuming RAM. If RAM usage is not a problem, load them as soon as possible and leave them loaded.
Chapter 5

Configuring Vocalizer

This chapter describes the different ways to configure Vocalizer. It describes:

- Configuration files
- Multiple value configuration parameters
- Vocalizer parameters

Vocalizer parameters

Vocalizer has many configurable parameters, some of which can only be configured on a global (per-process) basis, and others which can be changed for individual instances. There are five possible origins, listed here from the lowest to the highest precedence:

- Hard-coded Vocalizer defaults
- Configuration file parameters
- Parameters specified while opening an engine instance
- Parameters changed on an already open engine instance
- Parameters specified via markup (native escapes, SSML, or SAPI 5 XML markup)

Hard-coded Vocalizer defaults

Some Vocalizer parameters have hard-coded defaults. For example, if no user rulesets are specified in the configuration files or via the API, Vocalizer does not load any. As another example, Vocalizer defaults to a sentence-by-sentence read mode unless otherwise instructed via markup.

Configuration file parameters

You can configure parameters that are global in scope (shared over all engine instances) with an XML configuration file, by default install_path\config\ttsrshclient.xml (for the native API or Speech Server mode) or install_path\config\ttssapi.xml (for the SAPI5 API) within the Vocalizer installation directory. XML configuration files also allow you to specify defaults for some configuration parameters that can be changed via the API and/or markup, such as the default rate and volume. The use of configuration files is explained in more detail under Configuration files on page 80.

Parameters specified while opening an engine instance

Depending on the API used to access Vocalizer, some Vocalizer parameters are specified when opening a TTS engine instance rather than applying globally. In the native API this is done by setting values in the TTS_OPEN_PARAMS structure prior to calling the
TtsOpen API function, such as the initial language, voice, frequency, and audio format for the instance. For Speech Server some parameters like the audio format can be specified when establishing the MRCP connection (SIP INVITE for MRCPv2). For the Microsoft SAPI 5 API this is done by selecting a voice name with the desired attributes, optionally using SAPI 5 API calls to search the installed voices for those attributes.

For example, using the native API:

```c
TTS_OPEN_PARAMS params;
params.szLanguage = "French";
params.szVoice = "Sophie";
params.nFrequency = TTS_FREQ_8KHZ;
params.nOutputType = TTS_LINEAR_16BIT;
```

Most speak parameters can be set using the TtsSetParams function. In most cases parameters cannot be updated when the instance is busy executing the TtsProcessEx function (which performs the text-to-speech conversion process). Only the volume and the rate can be updated while speaking.

### Parameters changed on an already open engine instance

Depending on the API used to access Vocalizer, some Vocalizer parameters can be changed on an already open TTS engine instance, and affect that instance only. In most cases these parameters are a superset of what can be specified while opening the engine instance. In the native API this is done by the TtsSetParams API function, which allows changing most of the parameters specified for the TtsOpen API function and many other parameters. For Speech Server this can be done by the SET-PARAMS method, which can specify many parameters. For the Microsoft SAPI 5 API this is done by using the SetVoice method to change voices, or by set methods such as SetRate and SetVolume.

Most parameters cannot be set during synthesis. The only exceptions are the rate and volume parameters for the native API and Microsoft SAPI 5 API, which can be set during synthesis. However these changes only apply to audio buffers delivered after those APIs were called, which may be a significant lag from an end-user perspective as Vocalizer normally delivers audio faster than real time. This means there may be a significant amount of audio in the audio device playback queue with the old rate and volume setting, the end user either needs to listen to all that audio prior to hearing the new rate and volume, or the application has to do special work to account for that buffering (do rate and volume changes on the audio device instead of using Vocalizer API calls, stop and restart synthesis, or limit the amount of queued audio buffers).

### Parameters specified via markup

Many parameters can also be updated by inserting markup in the input text. These settings only apply to the current synthesis request: they are restored to the previous values for the next synthesis request. Vocalizer supports two markup languages: the native one (the default) and W3C SSML (with some proprietary extensions). When the Microsoft SAPI 5 API is used, Microsoft SAPI 5 markup is additionally supported. As a reminder:

- The native markup language is described in [Control sequence tasks](#) on page 29, with language specific details in each [Language Supplement](#).
- SSML markup is covered in [Vocalizer SSML support](#) on page 161.
- Microsoft SAPI 5 markup is covered in [Microsoft SAPI 5 compliance](#) on page 151.
Where to set Vocalizer parameters

The table in this section provides an overview of parameters and how they can be set.

As an example of how to interpret the table, here is a description for how to set the voice parameter using the native API:

- The voice can be set via the TtsOpen API function, by setting the appropriate values in the TTS_OPEN_PARAMS structure.
- Using the TtsSetParams function, if the instance is not busy executing the TtsProcessEx function in another thread.
- Using markup: the SSML &lt;voice&gt; element and voice attribute, or the native &lt;ESC&gt;\voice\ tag.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration files</th>
<th>Native API: TtsOpen</th>
<th>Native API: TtsSetParams</th>
<th>Native API: TtsProcessEx</th>
<th>Native API: other APIs</th>
<th>MRCP methods</th>
<th>SAPI 5 APIs</th>
<th>Native markup</th>
<th>SSML markup</th>
<th>SAPI 5 markup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation directory</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporary files directory</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User ID for log file names</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Licensing parameters</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSML validation mode</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging parameters for diagnostic logs</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging parameters for application event logs</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch cache parameters</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch proxy server parameters</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch extension mapping rules</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch user agent string</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch accept cookies</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch base URI</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch timeout</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch cache maxage</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet fetch cache maxstale</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loaded user dictionaries</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loaded user rulesets</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loaded ActivePrompt databases</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speaking rate</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Configuration files

Vocalizer can be configured using XML configuration files, by default `install_path\config\ttsrshclient.xml` (for native API based applications) and `install_path\config\ttssapi.xml` (for SAPI API based applications) within the installation directory.

For the native API, an additional user defined configuration file can be specified via `TtsSystemInit`; the settings in this additional configuration file override the baseline settings specified in `install_path\config\ttsrshclient.xml`. (The user defined file can specify only the parameters you wish to override, rather than having to specify every parameter.)
Configuration file format

Within each configuration file, each configuration parameter is specified by one or more XML elements, with the value of the parameter contained within that element. Here is a sample configuration file, followed by a description of the elements and attributes:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xml-stylesheet type="text/xsl" href="ttsrshclient.xsl"/>
<ttsrshclient version="5.0.0"
xmlns="http://www.nuance.com/nvn50/ttsrshclient">
  <tts_license_ports>100</tts_license_ports>
  <tts_license_ports_overdraft_thresh>90</tts_license_ports_overdraft_thresh>
  <cpr_license_ports>100</cpr_license_ports>
  <cpr_license_ports_overdraft_thresh>90</cpr_license_ports_overdraft_thresh>
  <license_servers>
    <server>27000@localhost</server>
  </license_servers>
</ttsrshclient>
```

The sample configuration file consists of several parts, as described below:

- **XML declaration**
- **Root element**
- **Main body**

Each configuration file value can optionally reference one or more variables, specified with the variable name inside “${…}”, for example, ${VNETWORKV5_SDK} for the Vocalizer installation directory. These variables can be any environment variable, or any one of the special variables discussed in Environment variable overrides on page 82.

**XML declaration**

The file begins with a standard XML declaration:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Style sheet declaration for viewing the file in a Web browser (optional):

```xml
<xml-stylesheet type="text/xsl" href="ttsrshclient.xsl"/>
```

**Root element**

The header also includes the root element of the document (specified in the document type declaration), that is, the container element for parameter elements. For Vocalizer Native API configuration files this must be `<ttsrshclient>`, while for Vocalizer SAPI 5 configuration files this must be `<ttssapi>`.

```xml
<ttsrshclient version="5.0.0"
xmlns="http://www.nuance.com/nvn50/ttsrshclient">
```
Main body

The main body of the file contains one or more elements which are parameters, for example:

```xml
<tts_license_ports>100</tts_license_ports>
<tts_license_ports_overdraft_thresh>90
</tts_license_ports_overdraft_thresh>
<cpr_license_ports>100</cpr_license_ports>
<cpr_license_ports_overdraft_thresh>90
</cpr_license_ports_overdraft_thresh>
<license_servers>
  <server>27000@localhost</server>
</license_servers>
```

A list of eligible parameters appears in the next section.

Single value configuration parameters

The following parameter names are specified as an element name. Not all parameters need to be set; some are optional, with Vocalizer automatically detecting the appropriate value for that installation.

Environment variable overrides

Each of these special variables can be overridden: Vocalizer first looks for an override within the XML configuration file, then an environment variable with the same name, then uses the defaults here.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;PID&gt;</td>
<td>Current process ID.</td>
<td>The OS-defined process ID for the process running Vocalizer. This is very useful for setting up Vocalizer to be used within multiple processes on the same computer at the same time without log files conflicting: update the configured log file paths to include <code>${PID}</code> within the value so the log files are unique for each process.</td>
</tr>
<tr>
<td>&lt;TMPDIR&gt;</td>
<td>Temporary files directory; default location for the Vocalizer log files and Internet fetch cache.</td>
<td>On Windows: the OS-defined temporary files directory (typically <code>C:\Documents and Settings\${USER}\Local Settings\Temp</code>). On Linux: <code>/tmp</code>.</td>
</tr>
<tr>
<td>&lt;USER&gt;</td>
<td>Current user’s login name.</td>
<td>OS-defined username for the process running Vocalizer</td>
</tr>
<tr>
<td>&lt;VNETWORKV5_SDK&gt;</td>
<td>Installation directory.</td>
<td>On Windows: The path of the loaded Vocalizer DLLs. On Linux: The default product installation directory for the Vocalizer RPM packages.</td>
</tr>
</tbody>
</table>
### Licensing parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cpr_license_dynamic_ports&gt;</td>
<td>Maximum number of CPR-only licenses that can be dynamically acquired after exhausting all CPR-only licenses acquired at startup. That is, the number of CPR-only licenses that may be cached in addition to the number of pre-allocated licenses defined by &lt;cpr_license_ports&gt;. Unless you are sharing licenses among many TTS hosts, we recommend using &lt;cpr_license_ports&gt;.</td>
<td>0</td>
</tr>
<tr>
<td>&lt;cpr_license_ports_overdraft_thresh&gt;</td>
<td>Threshold at which warnings will be logged to indicate that the system is near the CPR license limit.</td>
<td>0</td>
</tr>
<tr>
<td>&lt;cpr_license_ports&gt;</td>
<td>Number of concatenated prompt recording (CPR) only licenses to acquire at startup.</td>
<td>0</td>
</tr>
<tr>
<td>&lt;dynamic_license_expiration_time&gt;</td>
<td>Number of seconds to keep unused dynamically acquired licenses before automatically relinquishing them.</td>
<td>300</td>
</tr>
<tr>
<td>&lt;license_servers&gt;</td>
<td>See license_servers on page 88.</td>
<td>27000@localhost</td>
</tr>
<tr>
<td>&lt;tts_license_dynamic_ports&gt;</td>
<td>Maximum number of full TTS licenses that can be dynamically acquired after exhausting all full TTS licenses acquired at startup. That is, the number of full TTS licenses that may be cached in addition to the number of pre-allocated licenses defined by &lt;tts_license_ports&gt;. Unless you are sharing licenses among many TTS hosts, we recommend using &lt;tts_license_ports&gt;.</td>
<td>0</td>
</tr>
<tr>
<td>&lt;tts_license_ports_overdraft_thresh&gt;</td>
<td>Threshold at which warnings will be logged to indicate that the system is near the full TTS license limit.</td>
<td>90</td>
</tr>
<tr>
<td>&lt;tts_license_ports&gt;</td>
<td>Number of full TTS licenses to acquire at startup.</td>
<td>100</td>
</tr>
</tbody>
</table>

### Speak parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;default_rate&gt;</td>
<td>Default speaking rate on the Vocalizer rate scale of 1–100. This value is overridden if the rate is set via the native API, MRCP protocol, or markup, and has no effect for the Microsoft SAPI API where SAPI always specifies its own default rate.</td>
<td>50</td>
</tr>
<tr>
<td>&lt;default_volume&gt;</td>
<td>Default volume level on the Vocalizer volume scale of 0–100. This value is overridden if the volume is set via the native API, MRCP protocol, or markup, and has no effect for the Microsoft SAPI API where SAPI always specifies its own default volume.</td>
<td>80</td>
</tr>
</tbody>
</table>
### <escape_sequence>

If this parameter is non-empty, it overrides the default Vocalizer escape sequence `\x1b` with that string. This is useful to allow Vocalizer native control sequences in environments like SSML documents where the `<ESC>` character (`\x1b`) is not permitted.

The string provided here must be considered a regular expression and should therefore adhere to the user ruleset regular expression syntax (PCRE syntax). This means that special characters should be escaped with a `\`.

For example:
- `<escape_sequence>\!\!\</escape_sequence>`
  The sequence `!!` will be the escape sequence.
- `<escape_sequence>\!\</escape_sequence>`
  The sequence `!` will be the escape sequence.

### ssml_validation

SSML validation mode.

- strict (default) validates the input against the SSML 1.0 Recommendation with Nuance extensions, logging error messages and returning an API error if validation fails.
- warn performs the same validation, but only logs errors rather than failing out of the speak operation.
- none skips validation.

"strict" is the most robust setting because it ensures the application and Vocalizer engine properly cooperate, rather then forcing Vocalizer to try and handle bad input that may lead to strange TTS output.

---

### Language identifier parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;language_identifier_languages&gt;</td>
<td>See Language identifier parameters on page 84.</td>
<td></td>
</tr>
<tr>
<td>&lt;language_identifier_mode&gt;</td>
<td>Language identifier mode. Set to:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• rejection (the default) to keep the current language as-is if there is low-confidence.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• forced-choice to always switch to the language with the highest confidence.</td>
<td></td>
</tr>
<tr>
<td>&lt;language_identifier_scope&gt;</td>
<td>Language identifier scope. Set to:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• user-defined (the default) to use the language identifier only for blocks labeled with the <code>&lt;ESC&gt;\lang=unknown\</code> control sequence.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• message to use the language identifier automatically on each input message (typically a sentence).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• none to disable the language identifier.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;default_activeprompt_dbs&gt;</td>
<td>See default_activeprompts_dbs on page 88.</td>
<td>None.</td>
</tr>
</tbody>
</table>
### Internet fetch cache parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cache_directory&gt;</code></td>
<td>Directory name for the disk cache. If relative, the file path is relative to the containing configuration file. If Vocalizer is used within multiple processes on the same computer at the same time, it is safe to use one cache directory for all of them because Vocalizer automatically partitions the cache into subdirectories based on the OS process ID. However, all the other Internet fetch cache parameters are implemented on a per-process basis, not a per-machine basis. For example, if 10 processes use Vocalizer on the same computer at the same time with the same cache_directory and cache_total_size set to 200 MB, then the cache_directory could get as large as 2000 MB (200 MB * 10 processes).</td>
<td></td>
</tr>
<tr>
<td><code>&lt;cache_entry_exp_time&gt;</code></td>
<td>Time when an unused disk cache entry gets purged, in seconds.</td>
<td>3600</td>
</tr>
<tr>
<td><code>&lt;cache_entry_max_size&gt;</code></td>
<td>Maximum size of a single disk cache entry in MB.</td>
<td>20</td>
</tr>
<tr>
<td><code>&lt;cache_low_watermark&gt;</code></td>
<td>When the maximum cache size is reached, Vocalizer clears expired entries until it gets to this low-watermark threshold, in MB.</td>
<td>180</td>
</tr>
<tr>
<td><code>&lt;cache_timeout_open&gt;</code></td>
<td>Timeout for opening a cache entry in milliseconds. If this expires, Vocalizer logs a warning and then proceeds with re-fetching the URI. This should be rare, only happening in cases where the web server requires frequent revalidations of the cached copy, there is high contention for the entry across instances, and the download stalls.</td>
<td>5000</td>
</tr>
<tr>
<td><code>&lt;cache_total_size&gt;</code></td>
<td>Maximum size of the disk cache in MB.</td>
<td>200</td>
</tr>
<tr>
<td><code>&lt;cache_unlock_entries_at_startup&gt;</code></td>
<td>Reserved for future use; leave unchanged.</td>
<td>True</td>
</tr>
</tbody>
</table>

### Internet fetch parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;inet_accept_cookies&gt;</code></td>
<td>Whether to accept HTTP cookies (True or False).</td>
<td>True</td>
</tr>
<tr>
<td><code>&lt;inet_extension_rules&gt;</code></td>
<td>See <code>inet_extension_rules</code> on page 87.</td>
<td>8080</td>
</tr>
<tr>
<td><code>&lt;inet_proxy_server_port&gt;</code></td>
<td>Port of the http proxy server to use, for example 8080, ignored unless <code>inet_proxy_server</code> is non-empty.</td>
<td>No proxy used (empty value).</td>
</tr>
<tr>
<td><code>&lt;inet_proxy_server&gt;</code></td>
<td>IP address of an http proxy server to use, for example, 127.0.0.1.</td>
<td>No proxy used (empty value).</td>
</tr>
</tbody>
</table>
### Diagnostic and error logging parameters

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;inet_timeout_download&gt;</code></td>
<td>Default timeout for downloading a URI (from open through the final byte), in ms.</td>
<td>30000</td>
</tr>
<tr>
<td><code>&lt;inet_timeout_io&gt;</code></td>
<td>Default timeout for reading or writing a block of data over a web server connection, in ms.</td>
<td>30000</td>
</tr>
<tr>
<td><code>&lt;inet_timeout_open&gt;</code></td>
<td>Default timeout for opening a connection to a web server, in ms.</td>
<td>30000</td>
</tr>
<tr>
<td><code>&lt;inet_user_agent&gt;</code></td>
<td>User agent name in HTTP/1.1 headers.</td>
<td>Vocalizer for Network/5.0</td>
</tr>
<tr>
<td><code>&lt;event_log_file_base_name&gt;</code></td>
<td>Event log file base name. This will have 1.txt and 2.txt appended for the initial and rollover log filenames. If relative, the path is relative to the containing configuration file.</td>
<td>$(TMPDIR)/ttsrshclient_event_log_$<em>$(USER)$</em></td>
</tr>
<tr>
<td><code>&lt;event_log_file_enabled&gt;</code></td>
<td>Whether to log events to a file.</td>
<td>False</td>
</tr>
<tr>
<td><code>&lt;event_log_file_max_size&gt;</code></td>
<td>Event log file maximum size in MB.</td>
<td>50</td>
</tr>
<tr>
<td><code>&lt;event_log_input_text_file_base_name&gt;</code></td>
<td>Input text logging file base name. This will have 1.xml and 2.xml appended for the initial and rollover log filenames. If this value is relative, the path is relative to the containing configuration file.</td>
<td>%TMPDIR%	tsrshclient_event_log_input_text_%USER%_</td>
</tr>
<tr>
<td><code>&lt;event_log_input_text_file_max_size&gt;</code></td>
<td>Input text logging file maximum size, in MB.</td>
<td>50</td>
</tr>
<tr>
<td><code>&lt;event_log_input_text_max_capture&gt;</code></td>
<td>When event logging is enabled, the maximum number of speak requests where the input text is captured and logged at once: -1 (default) for unlimited 0 for disabling input text capture a positive number for a limit (The Vocalizer secure context parameter can be used to disable input text capture for specific TTS instances or speak requests.)</td>
<td>-1</td>
</tr>
<tr>
<td><code>&lt;log_cb_enabled&gt;</code></td>
<td>Whether to log errors via the callback mechanism.</td>
<td>True</td>
</tr>
<tr>
<td><code>&lt;log_file_base_name&gt;</code></td>
<td>Error and diagnostic log file base name. This will have 1.xml and 2.xml appended for the initial and rollover log filenames. If relative, the path is relative to the containing configuration file. If empty, messages will go to standard output.</td>
<td>%TMPDIR%	tsrshclient_log_%USER%_</td>
</tr>
<tr>
<td><code>&lt;log_file_enabled&gt;</code></td>
<td>Whether to log errors and diagnostics (True or False).</td>
<td>True</td>
</tr>
</tbody>
</table>
Multiple value configuration parameters

The following parameters can have multiple values, and are specified as a combination of nested elements. Not all parameters need to be set; some are optional, with Vocalizer automatically detecting the appropriate value for that installation.

**inet_extension_rules**

Rules for mapping file name extensions to MIME content types, specified as a sequence of `<extension>` elements where the name attribute is the extension and the value is the MIME content type.

For example, these are the Vocalizer defaults:

```
<inet_extension_rules>
  <extension name=".alaw"> audio/x-alaw-basic </extension>
  <extension name=".ulaw"> audio/basic </extension>
  <extension name=".mulaw"> audio/basic </extension>
  <extension name=".wav"> audio/x-wav </extension>
  <extension name=".L16"> audio/l16;rate=8000 </extension>
  <extension name=".txt"> text/plain </extension>
  <extension name=".xml"> text/xml </extension>
  <extension name=".ssml"> application/ssml+xml </extension>
  <extension name=".bdc"> application/edct-bin-dictionary </extension>
  <extension name=".dct"> application/edct-text-dictionary </extension>
  <extension name=".tdc"> application/edct-text-dictionary </extension>
</inet_extension_rules>
```

**default_dictionaries**

List of default dictionaries to load, where each matching dictionary is loaded when each port is opened. This element is optional; by default, no user dictionaries are loaded. A user dictionary is specified via a `<dictionary>` element. The priority attribute is optional. The value is the dictionary path or URI. User dictionaries are language-specific as indicated in the user dictionary file header, at runtime Vocalizer only does lookups within user dictionaries that match the current synthesis language.
For example: the following instructions would load *american_english.bdc*:

```
<default_dictionaries>
  <dictionary>
    http://myserver/american_english.bdc
  </dictionary>
</default_dictionaries>
```

**default_activeprompts_dbs**

List of default ActivePrompt databases to load, where each matching database is loaded when each port is opened. This element is optional; by default no ActivePrompt databases are loaded. A default ActivePrompt database is specified via an `<activeprompt_db>` element. The value is the ActivePrompt database path or URI.

ActivePrompt databases are voice-specific as indicated in the database header. At runtime Vocalizer only applies ActivePrompt databases that match the current synthesis voice.

For example, the following would load *apdb_rp_samantha_time.dat*.

```
<default_activeprompt_dbs>
  <activeprompt_db>http://myserver/apdb_rp_samantha_time.dat
  </activeprompt_db>
</default_activeprompt_dbs>
```

**default_rulesets**

List of default user rulesets to load, where each matching user ruleset is loaded when each port is opened. This element is optional; by default no user rulesets are loaded. A default user ruleset is specified via a `<ruleset>` element. The value is the user ruleset path or URI.

User rulesets are language-specific as indicated in the user ruleset file header, at runtime Vocalizer only applies user rulesets that match the current synthesis language.

For example, the following would load *american_english.txt*.

```
<default_rulesets>
  <ruleset>
    http://myserver/american_english.txt
  </ruleset>
</default_rulesets>
```

**license_servers**

Vocalizer licensing server TCP/IP addresses, where at least one licensing server must be defined, and multiple values are used to configure redundant licensing servers for fail-over support. See *Nuance License Manager Licensing Guide* for detail. For example, this is the Vocalizer default:

```
<license_servers>
  <server>27000@localhost</server>
</license_servers>
```

**language_identifier_languages**

Language identifier languages, a list of `<language>` elements containing 3-letter language codes. The language identifier automatically restricts the matched languages to the ones where voices are installed; this further limits the permissible languages for language identification, and also sets the precedence order for matching languages when they have an equal confidence (first listed has a higher precedence).
Here is the default:

```xml
<language_identifier_languages>
  <language_code>enu</language_code>
  <language_code>spm</language_code>
  <language_code>frf</language_code>
  <language_code>bae</language_code>
  <language_code>dun</language_code>
  <language_code>dub</language_code>
  <language_code>eng</language_code>
  <language_code>ena</language_code>
  <language_code>eni</language_code>
  <language_code>frf</language_code>
  <language_code>ged</language_code>
  <language_code>iti</language_code>
  <language_code>non</language_code>
  <language_code>ptp</language_code>
  <language_code>ptb</language_code>
  <language_code>spe</language_code>
  <language_code>swh</language_code>
</language_identifier_languages>
```
The following chapter describes the Vocalizer native API.

**API call sequence**

The following call sequence shows how to use the Native API. See the sample program `nvncmdline`, described in Testing the installation with sample applications on page 23, for extra details. The code is available in the installation under `install_path\api\demos\nvncmdline`.

The Application component refers to the source code that uses the API.

1. The Application calls the `TtsSystemInit` function to initialize the Vocalizer library. `TtsSystemInit` is called once per process; if it is called more than once per process, it can lead to memory leaks or crashes.

2. The Application calls the `TtsOpen` function to create a TTS engine instance. An application that serves multiple end users (such as an IVR platform) should do this multiple times, once for each simultaneous session it will support. At that time a number of general parameters can be specified such as the default language and voice, the destination callback used to stream audio to the application, and the desired audio format.

   The source callback function pointer can be specified to use input streaming for the input text. But this is optional since the TTS input can also be specified at the time a TTS action is requested via the `TtsProcessEx` function, which is preferred. This approach is easier to use, is more efficient, and enables the specification of the input via a URI, a filename or a text buffer.

3. When using user dictionaries, the application must:
   
   a. Define a `TTS_USRDICT_DATA` structure for each dictionary instance; this structure describes where to find the dictionary data and how to use it. This approach enables references to URI addresses. Fetch properties can be defined in case of remote access to the dictionary data.
   
   b. Optionally call `TtsMapCreate` to create a map data structure to control fetch properties such as the base URI or to override the default download timeout specified in the Vocalizer configuration file. The application calls `TtsMapSet*` functions (`TtsMapSetBool`, `TtsMapSetChar`, and so on) to define these properties in the created map data structure.
   
   c. Call the `TtsLoadUsrDictEx` function for each dictionary; this function returns a handle to a dictionary instance. Note that this must be repeated for each TTS
instance opened via TtsOpen; user dictionaries are loaded separately for each TTS instance.

4. When using user rulesets and/or ActivePrompt databases, the application defines a TTS_FETCHINFO_T structure for each user ruleset or ActivePrompt database; this structure describes where to find the data and how to use it. Like user dictionaries, this approach enables references to URI addresses. Fetch properties can be defined in case of remote access to the data using the same API calls described for user dictionaries above. The application then calls the TtsLoadTuningData function for each user ruleset and ActivePrompt database; this function returns a handle to the loaded tuning data instance. Note that this must be repeated for each TTS instance opened via TtsOpen, tuning data is loaded separately for each TTS instance.

5. Call TtsSetParams to adjust additional speak parameters if desired, such as the speaking rate and volume.

6. If the input callback method is not used, the Application sets up a TTS_SPEAK_DATA structure describing the input text; this structure supports specifying a URI, a filename or a memory block. If fetch properties need to be specified, the Application calls TtsMapCreate to create a fetch property map. The application then uses the TtsMapSet* functions to add properties to the map one by one.

If the method of streaming the input via the TTS source callback is used, the TTS_SPEAK_DATA structure members specifying the input, uri, and data, must be set to NULL.

7. The Application calls the TtsProcessEx function to convert the input text to audio of the type defined in TTS_OPEN_PARAMS (specified in step 1). The input can be specified via the TTS_SPEAK_DATA structure (URI or text buffer) or the source callback method. The TtsProcessEx function executes the TTS action synchronously; it only returns when all the speech samples have been generated.

8. The audio is streamed back to the application via the TTS_DEST_CB Destination callback.

9. If required, the Application can perform several TtsProcessEx calls, with different TTS_SPEAK_DATA and/or different TTS_USRDICT_DATA instances. When using dictionaries, TtsEnableUsrDictEx, TtsDisableUsrDictEx and TtsDisableUsrDictsEx can be used to enable dictionaries, change the priorities in which dictionaries are called, and disable dictionaries. The speak parameters can be updated using the TtsSetParams function. Note that a limited number of parameters can be updated while TtsProcessEx is busy (just the speaking rate and volume). When the input text contains markup controlling the speech generation, the parameters are updated for the course of the current TtsProcessEx execution, but at the end of the speak request, the parameters reset to the original values used at the start of the TtsProcessEx call.

10. When using dictionaries, the Application can optionally unload each dictionary by calling the TtsUnloadUsrDictEx function for each loaded dictionary. If this isn’t done, they are automatically unloaded when TtsClose is called to close the TTS instance.

11. When using user rulesets and/or ActivePrompt databases, the Application can optionally unload each user ruleset and ActivePrompt database by calling the TtsUnloadTuningData function for each one loaded. If this isn’t done, they are automatically unloaded when TtsClose is called to close the TTS instance.

12. When maps of fetch properties are created for either TTS_SPEAK_DATA, TTS_USRDICT_DATA, or TTS_FETCHINFO_T, the Application must call the TtsMapDestroy function for each map that has been created, otherwise there will be a memory leak.
13 The Application calls the TtsClose function to cleanup the TTS engine instance.

14 The Application calls the TtsSystemTerminate function when it is completely done with Vocalizer and ready to shut down (exit the process). The application should not call this if it wishes to do future Vocalizer operations; it is not safe to call TtsSystemInit and TtsSystemTerminate more than once within the same process.
API compatibility

Vocalizer supports API compatibility with RealSpeak Telecom Host 4.x and some earlier releases. However, this documentation does not describe older API calls from those products that are harder to use and/or less functional than the current API calls. New applications should restrict themselves to the current documented API calls, and Nuance encourages shifting older applications over to the current API calls as well.

To ensure your application only relies on the current documented API calls, set the following C pre-processor definitions in your code or on the compiler command line. This also helps avoid conflicts with Microsoft Windows API headers, as some of the backward compatible types conflict with those headers.

- TTSSO_NO_BACKWARD_COMPATIBLE_TYPES disables the outdated data types
- TTSSO_NO_BACKWARD_COMPATIBLE_FUNCS disables the outdated API functions
Defined data types

This section describes the defined data types that are required to interact with the API. Unless otherwise indicated, each of these types are declared in the header file `lh_ttso.h`.

**HTTSINSTANCE**

HTTSINSTANCE is the type representing the handle to an open TTS engine instance. It is returned from a successful call to the `TtsOpen` function.

**HTTSMAP**

HTTSMAP is the type representing the handle to an open TTS map. A TTS map contains the fetch properties for speech data, a dictionary instance, or a tuning data instance. It is returned from a successful call to the `TtsMapCreate` function.

**HTTSTUNINGDATA**

HTTSTUNINGDATA is the type representing the handle to a tuning data (user ruleset or ActivePrompt database) instance. It is returned from a successful call to the `TtsLoadTuningData` function.

**HTTSUSRDICT**

HTTSUSRDICT is the type representing the handle to a dictionary instance. It is returned from a successful call to the `TtsLoadUsrDictEx` function.

**HTTSVECTOR**

HTTSVECTOR is the type representing the handle to an open TTS vector. It is a member of the `TTS_SPEAK_DATA` and `TTS_USRDICT_DATA` structures, it is used for storing Internet fetch cookie jars, and it is updated when the API is called.

**TTSRETVAL**

TTSRETVAL is the type representing a TTS error. This type is in the header `ttso_types.h`, but the TTSRETVAL values are defined in `lh_err.h`.

**TTS_ERROR_SEVERITY**

TTS_ERROR_SEVERITY defines all the error severity levels that can be delivered to the `TTS_LOG_ERROR_CB` callback function.

```c
typedef enum TTS_ERROR_SEVERITY {
    TTS_SEVERITY_UNKNOWN = 0, /* Error severity is unknown */
    TTS_SEVERITY_CRITICAL, /* All instances out of service */
    TTS_SEVERITY_SEVERE, /* Service affecting failure */
    TTS_SEVERITY_WARNING, /* Application or non-service affecting failure */
    TTS_SEVERITY_INFO /* Informational message */
} TTS_ERROR_SEVERITY;
```
**TTS_EVENT**

TTS_EVENT defines all event types that can be delivered to the TTS_EVENT_CB callback function.

```c
typedef enum TTS_EVENT {
    TTS_EVENT_SENTENCEMARK,
    TTS_EVENT_BOOKMARK,
    TTS_EVENT_WORDMARK,
    TTS_EVENT_PHONEMEMARK,
    TTS_EVENT_PARAGRAPHMARK,
} TTS_EVENT;
```

Events normally mark the beginning of a particular kind of data (sentence, word, and so on) in the audio output. But an event is also issued when the audio output reaches the position of a bookmark inserted in the input text.

<table>
<thead>
<tr>
<th>TTS_EVENT_SENTENCEMARK</th>
<th>Marks the beginning of a sentence.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Use the type TTS_MARKER_SENTENCE to store the marker’s properties.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TTS_EVENT_BOOKMARK</th>
<th>Marks the position of a user bookmark; bookmarks can be inserted in the input text via the SSML <code>&lt;mark&gt;</code> element or the Vocalizer <code>&lt;ESC&gt;\mrk=x\</code> tag.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Use the type TTS_MARKER_BOOK to store the marker’s properties.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TTS_EVENT_WORDMARK</th>
<th>Marks the beginning of a word.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Note that word marks indicate words as identified by Vocalizer’s lexical analyzer, and early phase of TTS processing that is done before text normalization. This is sufficient for most input texts and most applications, but in some cases, the word markers may not match what a human would consider a single word. For example, there may be word marks for isolated punctuation, the word mark may include trailing punctuation, and for special text normalization types a single word mark may span a region that is actually spoken as multiple words (for example, a date written in <code>yyyy-mm-dd</code> form might only trigger one word mark).</td>
</tr>
<tr>
<td></td>
<td>Use the type TTS_MARKER_WORD to store the marker’s properties.</td>
</tr>
</tbody>
</table>
TTS_FETCHINFO_T is used to provide all the information to load (fetch) a user ruleset or ActivePrompt database.

typedef struct {
    const LH_CHAR * szUri;
    const LH_CHAR * szContentType;
    HTTSMAP hFetchProperties;
} TTS_FETCHINFO_T;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>szUri</td>
<td>String (zero-terminated) specifying the location of the user ruleset or ActivePrompt database. This can be an http address (http://) or a file name (regular or with file://).</td>
</tr>
</tbody>
</table>
| szContentType | MIME content type of the tuning data:
|               | - For a text user ruleset: "application/x-vocalizer-rettt+text" (TTS_MIME_RULESET_TEXT in lh_ttsso.h) |
|               | - For an ActivePrompt database: "application/x-vocalizer-activerprompt-db" (TTS_MIMEACTIVEPROMPT_DB in lh_ttsso.h). Append ";mode=automatic" (TTS_MIMEACTIVEPROMPT_DB_AUTOMATIC in lh_ttsso.h) to override the default ActivePrompt matching mode to automatic. |
|               | - For a binary user ruleset: "application/x-vocalizer-rettt+bin" (TTS_MIME_RULESET_BIN in lh_ttsso.h) |
TTS_MARKER

TTS_MARKER provides bit masks for all marker types. By bitwise or’ing the types of interest, an integer is created that can be used to specify the TTS_MARKER_MODE_PARAM parameter via the TtsSetParams functions. Only the corresponding event types will be issued by the event callback function. However, TTS_EVENT_SENTENCEMARK events are always generated.

typedef enum TTS_MARKER {
    TTS_MRK_SENTENCE = 0x0001,
    TTS_MRK_WORD = 0x0002,
    TTS_MRK_PHONEME = 0x0004,
    TTS_MRK_BOOK = 0x0008,
    TTS_MRK_PARAGRAPH = 0x0200
} TTS_MARKER;

TTS_MARKER_BOOK

TTS_MARKER_BOOK describes the parameters for a bookmark marker. This structure is passed to TTS_EVENT_CB callback when the event is TTS_EVENT_BOOKMARK.

typedef struct TTS_MARKER_BOOK {
    const LH_CHAR * szID;
    TTS_MARKER_POS mrkPos;
    const wchar_t * wszID;
} TTS_MARKER_BOOK;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hFetchProperties</td>
<td>Optional; specify NULL if not used. Used to set the properties of the fetch (note that some properties like for instance URL_BASE are also used for file fetching). The properties are stored in a map. The following functions maintain this map: TtsMapCreate, TtsMapDestroy, TtsMapSetChar, and TtsMapGetChar. See the lh_inettypes.h header file for available properties. For example, use SPIINET_URL_BASE to support relative URIs and filenames and SPIINET_TIMEOUT_DOWNLOAD to override the default fetch timeout configured in the Vocalizer configuration file.</td>
</tr>
<tr>
<td>szID</td>
<td>The bookmark string as a NULL terminated char string (ISO-8859-1 string). Both szID and wszID indicate the bookmark ID string. szID is only accurate for ISO-8859-1 characters. (Unicode characters within the original bookmark ID are changed to a question mark (“?”) character.)</td>
</tr>
<tr>
<td>mrkPos</td>
<td>Struct that describes the data values for one marker. Consists of nInputPos, nInputLen, nOutputPos, and nOutputLen.</td>
</tr>
<tr>
<td>wszID</td>
<td>The bookmark string as a NULL-terminated wchar_t string (Unicode string). Both szID and wszID indicate the bookmark ID string. wszID is recommended because, as type wchar_t, it supports Unicode characters; it is accurate for all possible bookmark IDs within the input text.</td>
</tr>
</tbody>
</table>
**TTS_MARKER_PARAGRAPH**

TTS_MARKER_PARAGRAPH describes the parameters for a paragraph marker. This structure is passed to TTS_EVENT_CB callback when the event is TTS_EVENT_PARAGRAPHMARK.

```c
typedef struct TTS_MARKER_PARAGRAPH {
    TTS_MARKER_POS mrkPos;
} TTS_MARKER_PARAGRAPH;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrkPos</td>
<td>Struct that describes the data values for one marker. Consists of nInputPos, nInputLen, nOutputPos, and nOutputLen.</td>
</tr>
</tbody>
</table>

**TTS_MARKER_PHONEME**

TTS_MARKER_PHONEME describes the parameters for a phoneme marker. This structure is passed to TTS_EVENT_CB callback when the event is TTS_EVENT_PHONEMEMARK.

```c
typedef struct TTS_MARKER_PHONEME {
    const LH_CHAR * szName;
    TTS_MARKER_POS mrkPos;
} TTS_MARKER_PHONEME;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>szName</td>
<td>A NULL-terminated L&amp;H+ phoneme string.</td>
</tr>
<tr>
<td>mrkPos</td>
<td>Struct that describes the data values for one marker. Consists of nInputPos, nInputLen, nOutputPos, and nOutputLen.</td>
</tr>
</tbody>
</table>

**TTS_MARKER_POS**

TTS_MARKER_POS describes the common properties of a marker. This structure is part of a marker structure that describes a particular kind of marker (TTS_MARKER_BOOK, TTS_MARKER_PHONEME...).

```c
typedef struct TTS_MARKER_POS {
    LH_U32 nInputPos;
    LH_U32 nInputLen;
    LH_U32 nOutputPos;
    LH_U32 nOutputLen;
} TTS_MARKER_POS;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nInputPos</td>
<td>Starting position for the marker within the input text in bytes, counted from the beginning of the input text. However, when SSML input is used or user rulesets are active, these positions will refer to the text positions after SSML processing (after expansion to proprietary markup) and after user ruleset transformations, not the original input text positions.</td>
</tr>
<tr>
<td>nInputLen</td>
<td>Length of the marker within the input text in bytes.</td>
</tr>
<tr>
<td>nOutputPos</td>
<td>Starting position for the marker within the output audio stream in samples, counted from the beginning of the audio stream.</td>
</tr>
<tr>
<td>nOutputLen</td>
<td>Length of the marker within the output audio stream in samples.</td>
</tr>
</tbody>
</table>
Not every marker type supports the four attributes. Here’s an overview of which TTS_EVENT event type supports what kind of data:

<table>
<thead>
<tr>
<th>Event type</th>
<th>nInputPos</th>
<th>nInput Len</th>
<th>nOutput Pos</th>
<th>nOutput Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOKMARK</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SENTENCEMARK</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>WORDMARK</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PARAGRAPHMARK</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PHONEMEMARK</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**TTS_MARKER_SENTENCE**

TTS_MARKER_SENTENCE describes the parameters for a sentence marker. This structure is passed to TTS_EVENT_CB callback when the event is TTS_EVENT_SENTENCEMARK.

```c
typedef struct TTS_MARKER_SENTENCE {
    TTS_MARKER_POS mrkPos;
} TTS_MARKER_SENTENCE;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrkPos</td>
<td>Struct that describes the data values for one marker. Consists of nInputPos, nInputLen, nOutputPos, and nOutputLen.</td>
</tr>
</tbody>
</table>

**TTS_MARKER_WORD**

TTS_MARKER_WORD describes the parameters for a word marker. This structure is passed to TTS_EVENT_CB callback when the event is TTS_EVENT_WORDMARK.

```c
typedef struct TTS_MARKER_WORD {
    TTS_MARKER_POS mrkPos;
} TTS_MARKER_WORD;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrkPos</td>
<td>Struct that describes the data values for one marker. Consists of nInputPos, nInputLen, nOutputPos, and nOutputLen.</td>
</tr>
</tbody>
</table>
TTS_OPEN_PARAMS

A TTS_OPEN_PARAMS typed structure is used to specify the (initial) parameters for a given TTS engine instance when calling TtsOpen.

typedef struct TTS_OPEN_PARAMS {
   /* Format version of this struct */
   TTS_VERSION fmtVersion;

   /* Voice parameters */
   LH_CHAR * szLanguage;
   LH_CHAR * szVoice;
   LH_U16 nFrequency;
   LH_U16 nOutputType;

   /* Synthesis callbacks */
   TTS_SOURCE_CB * TtsSourceCb;
   TTS_DEST_CB * TtsDestCb;
   TTS_EVENT_CB * TtsEventCb;

   /* Logging callbacks, any or all of these may be NULL */
   TTS_LOG_ERROR_CB * TtsLogErrorCb;
   TTS_LOG_EVENT_CB * TtsLogEventCb;
   TTS_LOG_DIAGNOSTIC_CB * TtsDiagnosticsCb;
} TTS_OPEN_PARAMS;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmtVersion</td>
<td>Structure version to allow forward compatibility with future releases, use TTS_CURRENT_VERSION.</td>
</tr>
<tr>
<td>szLanguage</td>
<td>Language string, either a Vocalizer Language name (for example, “American English”) or an IETF language code (“en-US”). See Vocalizer languages on page 179 for a full list.</td>
</tr>
<tr>
<td>szVoice</td>
<td>Voice name string. (For example, “Samantha”. )</td>
</tr>
<tr>
<td>nFrequency</td>
<td>Voice sampling rate: TTS_FREQ_8KHZ, TTS_FREQ_11KHZ (currently not supported), or TTS_FREQ_22KHZ.</td>
</tr>
<tr>
<td>nOutputType</td>
<td>Audio output format:</td>
</tr>
<tr>
<td></td>
<td>- TTS_LINEAR_16BIT for linear 16-bit PCM samples</td>
</tr>
<tr>
<td></td>
<td>- TTS_MULAW_8BIT for 8-bit µ-law samples (8KHz voices only)</td>
</tr>
<tr>
<td></td>
<td>- TTS_ALAW_8BIT for 8-bit A-law samples (8KHz voices only)</td>
</tr>
<tr>
<td>TtsSourceCb</td>
<td>(Optional, may be NULL.) Application defined callback for supplying the input text when the TTS_SPEAK_DATA structure that is passed to TtsProcessEx has NULL uri and data fields.</td>
</tr>
<tr>
<td>TtsDestCb</td>
<td>Application defined callback for supplying the audio output buffer and receiving audio output.</td>
</tr>
<tr>
<td>TtsEventCb</td>
<td>(Optional, may be NULL.) Application defined callback for TTS marker notifications, including bookmarks, word marks, phoneme marks, sentence marks, and paragraph marks.</td>
</tr>
</tbody>
</table>
TTS_PARAM

TTS_PARAM indicates the parameter type when calling TtsSetParams and TtsGetParams. See TtsSetParams on page 131 for a description of each parameter.

typedef enum TTS_PARAM {
    TTS_LANGUAGE_PARAM = 0,
    TTS_VOICE_PARAM = 1,
    TTS_VOLUME_LARGESCALE_PARAM = 6,
    TTS_RATE_LARGESCALE_PARAM = 7,
    TTS_OUTPUT_TYPE_PARAM = 9,
    TTS_MARKER_MODE_PARAM = 10,
    TTS_FREQUENCY_PARAM = 16,
    TTS_SECURE_CONTEXT_PARAM = 17,
    TTS_PRODUCT_VERSION_PARAM = 18,
    TTS_LID_SCOPE_PARAM = 19,
    TTS_LID_MODE_PARAM = 20,
    TTS_LID_LANGUAGES_PARAM = 21,
    TTS_TOTAL_PARAMS /* indicates the total number of parameters */
} TTS_PARAM;
**TTS_PARAM_T**

TTS_PARAM_T describes one parameter name, parameter value couple. See TtsSetParams on page 131 for a list of possible values for the nParam field for each parameter.

```c
typedef struct TTS_PARAM_S {
    TTS_PARAM nParam;
    TTS_PARAM_VALUE_T paramValue;
} TTS_PARAM_T;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nParam</td>
<td>The parameter ID</td>
</tr>
<tr>
<td>paramValue</td>
<td>A union that allows you to store the parameter value as one of a number of types.</td>
</tr>
</tbody>
</table>

**TTS_PARAM_VAL_ARRAY_T**

TTS_PARAM_VAL_ARRAY_T structure is used as a storage place for a NULL terminated string.

```c
typedef struct TTS_PARAM_VAL_ARRAY_S {
    LH_U16 nValue;
    LH_VOID * pValue
} TTS_PARAM_VAL_ARRAY_T;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nValue</td>
<td>A 16-bit unsigned value that indicates the length of the string in pValue.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ For TtsSetParams calls, set this to the length of the string excluding the NULL terminator.</td>
</tr>
<tr>
<td></td>
<td>▪ For TtsGetParams calls, set this to the length of the buffer in pValue including the byte for the NULL terminator.</td>
</tr>
<tr>
<td>pValue</td>
<td>For TtsSetParams calls, a pointer to the parameter value string.</td>
</tr>
<tr>
<td></td>
<td>For TtsGetParams calls, a pointer to a string buffer to be filled with the parameter value.</td>
</tr>
</tbody>
</table>

**TTS_PARAM_VALUE_T**

TTS_PARAM_VALUE_T union type is used as a storage place for the value of a parameter that is specified via a TTS_PARAM typed structure.

```c
typedef union TTS_PARAM_VAL_U {  
    LH_U32 nNo;
    TTS_PARAM_VAL_ARRAY_T array;
    VOID * pObj;
    HTTSMAP hMap
} TTS_PARAM_VALUE_T;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nNo</td>
<td>A 32-bit unsigned integer for parameters with an integral value.</td>
</tr>
<tr>
<td>array</td>
<td>A TTS_PARAM_VAL_ARRAY_T structure for parameters with a string value.</td>
</tr>
<tr>
<td>pObj</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>hMap</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
**TTS_SPEAK_DATA**

**TTS_SPEAK_DATA** is used when calling TtsProcessEx. It is used to describe the location of the input data for a text to speech action and its properties. Note that it is still possible to use the source callback method; in this case the uri and data structure members should be set to NULL.

```c
typedef struct {
    LH_CHAR* uri;
    LH_VOID* data;
    LH_U32 lengthBytes;
    LH_CHAR* contentType;
    HTTSMAP fetchProperties;
    HTTSVECTOR fetchCookieJar;
} TTS_SPEAK_DATA;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>String specifying the location of the input data. This can be an http address (http://) or a file name (regular or with file://). Set the uri member to NULL to indicate that the input data is provided via the data member or the source callback.</td>
</tr>
<tr>
<td>data</td>
<td>Pointer to buffer containing the input text. This structure member will be used only when uri is NULL. Set both uri and data to NULL to use the source callback function.</td>
</tr>
<tr>
<td>lengthBytes</td>
<td>The length of the data buffer in bytes. Set this to 0 if the data field is set to NULL.</td>
</tr>
</tbody>
</table>
| contentType   | String that specifies the MIME content type of the data. The string is case-sensitive.  
  - When specifying data using the data member, this string must be specified.  
  - When specifying data using the uri member, this is optional: a NULL indicates the MIME content type should be automatically detected from the URI fetch, while a non-NULL value overrides the MIME content type from the URI fetch. For http:// access the web server returns the MIME content type, while for file:// access the inet_extension_rules XML configuration parameter is consulted to map the file’s extension to a MIME content type.  
  - When specifying data using the source callback method, this is optional but strongly recommended: NULL results in presuming plain text in a language-specific character set, which is only supported for backward compatibility with older releases.  
  - Supported values are:  
    - "application/synthesis+ssml" (preferred) or "text/xml" for a W3C SSML document  
    - "text/plain;charset=charset" for a plain text document, where charset is replaced by a character set name. Recommended character sets are Unicode UTF-16 (what Vocalizer uses internally) and UTF-8, but Vocalizer also supports a broad variety of character sets—see the description below.  
| fetchProperties | Used to set the properties of the fetch. The properties are stored in a map. The following functions manipulate this map: TtsMapCreate, TtsMapDestroy, TtsMapSetChar, and TtsMapGetChar. See the lh_inettypes.h header file for available properties. |
| fetchCookieJar | Reserved for future use; pass NULL. |
Vocalizer does its internal processing using Unicode UTF-16. When the input text is in a different character set, Vocalizer transcodes it to UTF-16 at the start of its processing. The following table lists some common supported character sets.

<table>
<thead>
<tr>
<th>Character set</th>
<th>Languages</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>All languages</td>
<td></td>
</tr>
<tr>
<td>UTF-16</td>
<td>All languages</td>
<td>This is the recommended character set, as Vocalizer uses UTF-16 for its internal processing. (If UTF-16 is not convenient, UTF-8 is the next best choice.) If the byte order mark is missing, big-endian is assumed.</td>
</tr>
<tr>
<td>ISO-8859-1</td>
<td>Western languages</td>
<td></td>
</tr>
<tr>
<td>windows-1252</td>
<td>Western languages</td>
<td></td>
</tr>
<tr>
<td>EUC-jp</td>
<td>Japanese</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(synonym: EUC)</td>
</tr>
<tr>
<td>Shift-JIS</td>
<td>Japanese</td>
<td></td>
</tr>
</tbody>
</table>

A third party component called IBM ICU is used to transcode the input character set to the native UTF-16 character set. It supports a very broad range of character sets. For more information, see Copyright and licensing for third party software on page 181.

For more information about the character sets for the contentType parameter, see the following web sites:

- [www.iana.org/assignments/character-sets](http://www.iana.org/assignments/character-sets)

**TTS_USRDICT_DATA**

TTS_USRDICT_DATA is used when calling TtsLoadUsrDictEx. It is used to describe the properties of a dictionary instance.

```c
typedef struct {
    LH_U32 version;
    LH_CHAR * uri;
    LH_VOID * data;
    LH_U32 lengthBytes;
    LH_CHAR * contentType;
    HTTSMAP fetchProperties;
    HTTSVECTOR fetchCookieJar;
} TTS_USRDICT_DATA;
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>Structure version to allow forward compatibility with future releases. Use TTS_CURRENT_VERSION.</td>
</tr>
<tr>
<td>uri</td>
<td>String specifying the location of the dictionary. This can be an http address (http:///) or a file name (regular or with file:///). Set the uri member to NULL to indicate that the input data is read from the data member.</td>
</tr>
<tr>
<td>data</td>
<td>Pointer to a buffer containing the user dictionary data. This structure member will be used when uri is NULL.</td>
</tr>
</tbody>
</table>
The TTS_VOICE_INFO structure is used by TtsGetVoiceList to return information about an installed TTS engine voice.

typedef struct TTS_VOICE_INFO {
    LH_CHAR szVersion[TTS_MAX_STRING_LENGTH];
    LH_CHAR szLanguage[TTS_MAX_STRING_LENGTH];
    LH_CHAR szLanguageIETF[TTS_MAX_STRING_LENGTH];
    LH_CHAR szLanguageTLW[4];
    LH_CHAR szVoice[TTS_MAX_STRING_LENGTH];
    LH_CHAR szAge[TTS_MAX_STRING_LENGTH];
    LH_CHAR szGender[TTS_MAX_STRING_LENGTH];
    LH_CHAR szVoiceModel[TTS_MAX_STRING_LENGTH];
    LH_U16 nFrequency;
} TTS_VOICE_INFO;

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lengthBytes</td>
<td>The length of the data buffer in bytes. Specify 0 when the data member is NULL.</td>
</tr>
<tr>
<td>contentType</td>
<td>String that specifies the MIME content type of the user dictionary.</td>
</tr>
<tr>
<td></td>
<td>- When specifying data using the data member, this string must be specified.</td>
</tr>
<tr>
<td></td>
<td>- When specifying data using the uri member, this is optional: a NULL indicates the MIME content type should be automatically detected from the URI fetch, while a non-NULL value overrides the MIME content type from the URI fetch. For http:// access the web server returns the MIME content type, while for file:// access the inet_extension_rules XML configuration parameter is consulted to map the file's extension to a MIME content type.</td>
</tr>
<tr>
<td></td>
<td>Supported values are:</td>
</tr>
<tr>
<td></td>
<td>- &quot;application/edct-bin-dictionary&quot; (TTS_MIME_USRDICT_BINARY) for a Vocalizer binary format user dictionary.</td>
</tr>
<tr>
<td>fetchProperties</td>
<td>Used to set the properties of the fetch. The properties are stored in a map.</td>
</tr>
<tr>
<td></td>
<td>The following functions manipulate this map:</td>
</tr>
<tr>
<td></td>
<td>- TtsMapCreate</td>
</tr>
<tr>
<td></td>
<td>- TtsMapDestroy</td>
</tr>
<tr>
<td></td>
<td>- TtsMapSetChar</td>
</tr>
<tr>
<td></td>
<td>- TtsMapGetChar</td>
</tr>
<tr>
<td></td>
<td>See the lh_inettypes.h header file for available properties.</td>
</tr>
<tr>
<td>fetchCookieJar</td>
<td>Reserved for future use; pass NULL.</td>
</tr>
</tbody>
</table>

**TTS_VOICE_INFO**

The TTS_VOICE_INFO structure is used by TtsGetVoiceList to return information about an installed TTS engine voice.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szVersion</td>
<td>Voice version number, such as 5.2.0.7151</td>
</tr>
<tr>
<td>szLanguage</td>
<td>Language name, such as American English</td>
</tr>
<tr>
<td>szLanguageIETF</td>
<td>IETF language code, such as en-us</td>
</tr>
<tr>
<td>szLanguageTLW</td>
<td>Three-letter language code as used in user dictionaries and user rulesets, such as ENU</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>szVoice</td>
<td>Voice name, such as Samantha</td>
</tr>
<tr>
<td>szAge</td>
<td>Voice age, such as Adult</td>
</tr>
<tr>
<td>szGender</td>
<td>Voice gender: Male, Female, or Neutral</td>
</tr>
<tr>
<td>szVoiceModel</td>
<td>Vocalizer internal name for the synthesizer technology</td>
</tr>
<tr>
<td>nFrequency</td>
<td>Voice sampling rate:</td>
</tr>
<tr>
<td></td>
<td>▪ TTS_FREQ_8KHZ</td>
</tr>
<tr>
<td></td>
<td>▪ TTS_FREQ_11KHZ (currently not supported)</td>
</tr>
<tr>
<td></td>
<td>▪ TTS_FREQ_22KHZ</td>
</tr>
</tbody>
</table>
## Function descriptions

The Vocalizer API functions are described in this section.

The following table shows an alphabetical list of the Vocalizer API functions.

<table>
<thead>
<tr>
<th>Use this function...</th>
<th>To perform this task...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TtsClose</td>
<td>Close an engine instance and free all its resources.</td>
</tr>
<tr>
<td>TtsDisableUsrDictEx</td>
<td>Disable a dictionary instance for a channel.</td>
</tr>
<tr>
<td>TtsDisableUsrDictsEx</td>
<td>Disable all dictionary instances for a channel.</td>
</tr>
<tr>
<td>TtsEnableUsrDictEx</td>
<td>Enable a dictionary instance for a channel.</td>
</tr>
<tr>
<td>TtsGetParams</td>
<td>Get the value of one or more parameters.</td>
</tr>
<tr>
<td>TtsGetVoiceList</td>
<td>Get the list of available voices and their properties</td>
</tr>
<tr>
<td>TtsLoadTuningData</td>
<td>Load an ActivePrompt database or user ruleset for use by a TTS engine instance.</td>
</tr>
<tr>
<td>TtsLoadUsrDictEx</td>
<td>Load a user dictionary for use by the engine, returning a new dictionary instance.</td>
</tr>
<tr>
<td>TtsMapCreate</td>
<td>Create a key/value map for specifying internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapDestroy</td>
<td>Destroy a key/value map that stored internet fetch properties</td>
</tr>
<tr>
<td>TtsMapFreeChar</td>
<td>Free a character string associated with a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapGetBool</td>
<td>Get a boolean value from a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapGetChar</td>
<td>Get a character string value from a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapGetU32</td>
<td>Get an unsigned integer value from a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapSetBool</td>
<td>Set a boolean value in a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapSetChar</td>
<td>Set a character string value in a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsMapSetU32</td>
<td>Set an unsigned integer value in a key/value map that stores internet fetch properties.</td>
</tr>
<tr>
<td>TtsOpen</td>
<td>Open an instance of the TTS engine.</td>
</tr>
<tr>
<td>TtsProcessEx</td>
<td>Convert input data (text) into output data (audio).</td>
</tr>
<tr>
<td>TtsSessionEnd</td>
<td>Disassociate a TTS engine instance with a session ID string.</td>
</tr>
<tr>
<td>TtsSessionStart</td>
<td>Associate a TTS engine instance with a session ID string.</td>
</tr>
<tr>
<td>TtsSetParams</td>
<td>Set the value of one or more parameters.</td>
</tr>
<tr>
<td>TtsStop</td>
<td>Stops the Text-To-Speech process.</td>
</tr>
<tr>
<td>TtsSystemInit</td>
<td>One-time process level initialization of the Vocalizer library.</td>
</tr>
<tr>
<td>TtsSystemTerminate</td>
<td>One-time process level shutdown of the Vocalizer library.</td>
</tr>
<tr>
<td>TtsUnloadTuningData</td>
<td>Unload an ActivePrompt database or user ruleset.</td>
</tr>
<tr>
<td>TtsUnloadUsrDictEx</td>
<td>Unload a dictionary instance from memory; frees memory resources.</td>
</tr>
</tbody>
</table>
The functions can be organized in these groups:

Initialize and shutdown Vocalizer:

- **TtsSystemInit** on page 134
- **TtsSystemTerminate** on page 135

Manage the TTS process:

- **TtsClose** on page 110
- **TtsGetVoiceList** on page 115
- **TtsOpen** on page 127
- **TtsProcessEx** on page 128
- **TtsSessionEnd** on page 129
- **TtsSessionStart** on page 130
- **TtsStop** on page 133

Manage TTS maps for specifying internet fetch properties:

- **TtsMapCreate** on page 118
- **TtsMapDestroy** on page 119
- **TtsMapFreeChar** on page 120
- **TtsMapGetBool** on page 121
- **TtsMapGetChar** on page 122
- **TtsMapGetU32** on page 123
- **TtsMapSetBool** on page 124
- **TtsMapSetChar** on page 125
- **TtsMapSetU32** on page 126

Manage user dictionaries and tuning data:

- **TtsDisableUsrDictEx** on page 111
- **TtsDisableUsrDictsEx** on page 112
- **TtsEnableUsrDictEx** on page 113
- **TtsLoadTuningData** on page 116
- **TtsLoadUsrDictEx** on page 117
- **TtsUnloadTuningData** on page 136
- **TtsUnloadUsrDictEx** on page 137

Manage parameters:

- **TtsGetParams** on page 114
- **TtsSetParams** on page 131
TtsClose

TTSRETCVAL TtsClose(
   HTTSINSTANCE hTtsInst);

Close a TTS engine and free all its associated resources. This automatically unloads any associated user dictionaries, user rulesets, and ActivePrompt databases.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance to close.</td>
</tr>
</tbody>
</table>

**Error codes**: This function can return license related errors. See Error codes on page 147.
TtsDisableUsrDictEx

TTSRETVAL TtsDisableUsrDictEx ( 
    HTTSINSTANCE hTtsInst, 
    HTTSUSR_DICT hUsrDict)

Disables a user dictionary instance on a TTS engine instance. The dictionary instance must first have been enabled for use by the instance using TtsEnableUsrDictEx or TtsLoadUsrDictEx. Note that disabling the dictionary does not unload it from memory. To unload a dictionary, use the TtsUnloadUsrDictEx function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>hUsrDict</td>
<td>[in] Handle to a loaded dictionary instance.</td>
</tr>
</tbody>
</table>

Always use this function before calling TtsEnableUsrDictEx. Refer to the API call sequence on page 91 for more information.
**TtsDisableUsrDictsEx**

TTSRETVAL TtsDisableUsrDictsEx(HTTSINSTANCE hTtsInst)

Disables all user dictionary instances on a TTS engine instance. The dictionary instances must first have been enabled for use by the instance using TtsEnableUsrDictsEx or TtsLoadUsrDictsEx. Note that disabling the dictionaries does not unload them from memory. To unload a dictionary, use the TtsUnloadUsrDictsEx function. To disable dictionaries one by one use TtsDisableUsrDictsEx.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
</tbody>
</table>

Refer to the API call sequence on page 91 for more information.
TtsEnableUsrDictEx

TTSRETVAL TtsEnableUsrDictEx ( 
  HTTSINSTANCE hTtsInst, 
  HTTSUSRDICT hUsrDict, 
  LH_U32 u32Priority)

Enables a user dictionary instance and/or changes its priority on a TTS engine instance. Once a dictionary instance has been loaded by TtsLoadUsrDictEx on a TTS engine instance, the default priority has been attached; the dictionary is enabled with the default (lowest) priority. If two dictionaries have the default priority, the order in which the dictionaries are loaded is important. The last loaded dictionary has the ‘highest’ priority. This means that when a token has to be processed, a lookup will take place using the last loaded dictionary first.

To change the default priority, the dictionary has to be disabled by calling TtsDisableUsrDictEx and enabled again by calling TtsEnableUsrDictEx.

TtsEnableUsrDictEx can also be called to enable a dictionary again that has been disabled by a previous call of TtsDisableUsrDictEx.

Once a dictionary has been loaded using the TtsLoadUsrDictEx function, it can be enabled for use by only one TTS engine instance at a time; if two instances want to use the same dictionary then the dictionary must be loaded separately for each instance. Each dictionary has a unique priority; no two dictionaries can have the same priority at the same time except the default priority. The highest possible priority value is 0; the higher the priority, the lower the value of the priority parameter should be.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>hUsrDict</td>
<td>[in] Handle to a loaded dictionary instance.</td>
</tr>
<tr>
<td>u32Priority</td>
<td>[in] Sets the priority for the dictionary instance.</td>
</tr>
</tbody>
</table>

Always call TtsDisableUsrDictEx or TtsDisableUsrDictsEx before calling TtsEnableUsrDictEx.

Refer to the API call sequence on page 91 for more information.
TtsGetParams

Retrieves the values of one or more parameters. See TtsSetParams on page 131 for a list of supported parameters and possible values.

TTSRETVAL TtsGetParams(
    HTTSINSTANCE hTtsInst,
    TTS_PARAM_T* pParams,
    LH_U16 cParams)

The TtsGetParams and TtsSetParams functions operate independently of the escape sequences that can also be used to set the volume and rate. TtsGetParams values do not reflect parameter changes that result from escape sequences embedded in the input text.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>pParams</td>
<td>[in, out] Specifies which parameters to retrieve. Each element in this array specifies an individual parameter to retrieve:</td>
</tr>
<tr>
<td></td>
<td>▪ Set pParams[index].nParam to the parameter ID (TTS_PARAM value).</td>
</tr>
<tr>
<td></td>
<td>▪ For each string parameter, set pParams[index].paramValue.array.pValue to a string buffer large enough to hold the value</td>
</tr>
<tr>
<td></td>
<td>▪ For each string parameter, set pParams[index].paramValue.array.nValue to the size of the string buffer, including the byte for the NULL terminator.</td>
</tr>
<tr>
<td>cParams</td>
<td>[in] Number of parameters to get.</td>
</tr>
</tbody>
</table>
TtsGetVoiceList

Retrieves information about the installed Vocalizer voices.

TTSRETRVAL TtsGetVoiceList (  
    TTS_VOICE_INFO * pVoiceList,  
    LH_U16 * pu16VoiceList);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pVoiceList</td>
<td>[out] Pointer to an array of TTS_VOICE_INFO structures that will be filled with information about the available voices. This may be set to NULL to query the number of installed voices.</td>
</tr>
<tr>
<td>pu16VoiceList</td>
<td>[in, out] On input, this must indicate the number of TTS_VOICE_INFO structures allocated for pVoiceList. On output, this is set to the actual number of TTS_VOICE_INFO structures filled in pVoiceList, or if pVoiceList is NULL, this is set to the total number of installed voices.</td>
</tr>
</tbody>
</table>
TtsLoadTuningData

Load a user ruleset or ActivePrompt database for use by a TTS instance.

TTSRETVAL TtsLoadTuningData (  
   HTTSINSTANCE hTtsInst,  
   const TTS_FETCHINFO_T * pTuningData,  
   HTTSTUNINGDATA * phTuningData)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>pTuningData</td>
<td>[in] Const pointer to a tuning data properties description, which provides the URI to the tuning data, a MIME content type that describes the type of data to load, and optional internet fetch properties.</td>
</tr>
<tr>
<td>phTuningData</td>
<td>[out] Handle to the tuning data instance.</td>
</tr>
</tbody>
</table>
TtsLoadUsrDictEx

Load a user dictionary instance into memory.

TTSRETVAL TtsLoadUsrDictEx (  
  HTTSENSTANCE hTtsInst,  
  const TTS_USRDICT_DATA* pUsrDictData,  
  HTTSUSRDICT* phUsrDict)

The dictionary is implicitly enabled with the default priority. (See TtsEnableUsrDictEx on page 113 for making dictionaries explicitly enabled with a chosen priority.)

Each dictionary instance is initialized with the default (lowest) priority. All dictionary instances must have a different priority (except the default priority, which can be used by several dictionaries). The priority can be set by TtsEnableUsrDictEx. If two dictionaries have the default priority, the order in which the dictionaries are loaded is important. The last loaded dictionary has the ‘highest’ priority. This means that when a token has to be processed, a lookup will take place using the last loaded dictionary first.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>pUsrDictData</td>
<td>[in] Const pointer to a dictionary properties description.</td>
</tr>
<tr>
<td>phUsrDict</td>
<td>[out] Handle to a dictionary instance.</td>
</tr>
</tbody>
</table>

The order in which dictionaries are looked up can be changed by using TtsEnableUsrDictEx and setting the priority to a different value.

Refer to the API call sequence on page 91 for more information.
TtsMapCreate

Create an empty map that can be used to store the internet fetch properties for TTS_SPEAK_DATA, TTS_USRDICT_DATA, or TTS_FETCHINFO_T structures

TTSRETVVAL TtsMapCreate(HTTSMAP* phTtsMap)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phTtsMap</td>
<td>[out] Pointer to a handle to a TTS Map instance.</td>
</tr>
</tbody>
</table>
TtsMapDestroy

Destroy a map.

TTSRETV AL TtsMapDestroy(HTTSMAP phTtsMap)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
</tbody>
</table>
TtsMapFreeChar

Free the character string returned by an earlier call to TtsMapGetChar.

TTSRETVAl TtsMapFreeChar (  
    HTTSMAP hTtsMap,  
    const LH_CHAR* pszValue)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>pszValue</td>
<td>[in] The character string to free as earlier returned by TtsMapGetChar.</td>
</tr>
</tbody>
</table>
TtsMapGetBool

Gets a named property of type boolean (LH_S32) from a map.

TTSRETVAL TtsMapGetBool(
    HTTSMAP hTtsMap,
    const LH_CHAR* szKey,
    LH_S32* pbValue)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
<tr>
<td>pbValue</td>
<td>[out] The value of the property.</td>
</tr>
</tbody>
</table>
TtsMapGetChar

Gets a named property of type string (LH_CHAR *) from a map. On success this function allocates the character string, and the application is responsible for freeing it by calling TtsMapFreeChar.

```c
TTSRETVAl TtsMapGetChar(
    HTTSMAP hTtsMap,
    const LH_CHAR* szKey,
    const LH_CHAR** pszValue)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
<tr>
<td>pszValue</td>
<td>[out] The value of the property.</td>
</tr>
</tbody>
</table>
TtsMapGetU32

Gets a named property of type unsigned 32-bit integer (LH_U32) from a map.

TTSRETVAL TtsMapGetU32 (  
   HTTSMAP hTtsMap,  
   const LH_CHAR* szKey,  
   LH_U32* pu32Value)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
<tr>
<td>pu32Value</td>
<td>[out] The value of the property.</td>
</tr>
</tbody>
</table>
TtsMapSetBool

Set a named property of type boolean (LH_S32) in a map. The list of available properties can be found in the header file \texttt{lh\_inettypes.h}.

\begin{verbatim}
TTSRETVAL TtsMapSetBool ( 
    HTTSMAP hTtsMap, 
    const LH_CHAR* szKey, 
    LH_S32 bValue);
\end{verbatim}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
<tr>
<td>bValue</td>
<td>[in] The value of the property.</td>
</tr>
</tbody>
</table>
TtsMapSetChar

Set a named property of type string (LH_CHAR *) in a map. The string is deep copied into the map. The list of available properties can be found in the header file lh_inettypes.h.

TTSRETVAL TtsMapSetChar (  
    HTTSMAP hTtsMap,  
    const LH_CHAR* szKey,  
    const LH_CHAR* szValue)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
</tbody>
</table>
TtsMapSetU32

Set a named property of type unsigned 32-bit integer (LH_U32) in a map. The list of available properties can be found in the header file lh_inettypes.h.

TTSRETVAL TtsMapSetU32 (  
  HTTSMAP hTtsMap,  
  const LH_CHAR* szKey,  
  LH_U32 u32Value)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsMap</td>
<td>[in] Handle to a TTS Map instance.</td>
</tr>
<tr>
<td>szKey</td>
<td>[in] The name of the property.</td>
</tr>
<tr>
<td>u32Value</td>
<td>[in] The value of the property.</td>
</tr>
</tbody>
</table>
**TtsOpen**

Open a new TTS engine instance. The engine is created according to the information specified in the pOpenParams parameter.

```c
TTSRETVAL TtsOpen(
    HTTSINSTANCE * phTtsInst,
    TTS_OPEN_PARAMS * pOpenParams,
    LH_VOID * pAppData);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phTtsInst</td>
<td>[out] This pointer receives the handle for the newly created TTS engine</td>
</tr>
<tr>
<td></td>
<td>instance. This handle is passed to other TTS functions to identify the</td>
</tr>
<tr>
<td></td>
<td>instance.</td>
</tr>
<tr>
<td>pOpenParams</td>
<td>[in] This is a pointer to a structure whose members are used to control</td>
</tr>
<tr>
<td></td>
<td>aspects and behaviors of the created engine. Members of this structure must</td>
</tr>
<tr>
<td></td>
<td>be filled in; see TTS_OPEN_PARAMS on page 101 for more information.</td>
</tr>
<tr>
<td>pAppData</td>
<td>[in] Application specific data that is passed back to the application each</td>
</tr>
<tr>
<td></td>
<td>time one of the callbacks is invoked.</td>
</tr>
</tbody>
</table>

**Error codes:** This function can return license-related errors. See Error codes on page 147.
TtsProcessEx

Convert input text data into speech of the previously specified output format. The input data properties can be described via the pSpeakData argument. To use the source callback, set the structure members specifying the input (uri and data) to NULL.

TTSRETVALS TtsProcessEx (  
    HTTSINSTANCE hTtsInst,  
    const TTS_SPEAK_DATA* pSpeakData)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance</td>
</tr>
<tr>
<td>pSpeakData</td>
<td>[in] Pointer to TTS_SPEAK_DATA typed structure, which describes where the input data for text to speech can be found and its properties. Refer to the TTS_SPEAK_DATA structure type description for more details.</td>
</tr>
</tbody>
</table>

Error codes: This function can return licensing related errors. See Error codes on page 147.

**Note:** The TtsProcessEx function determines the TTS input method as follows:

1. Check the uri member in the TTS_SPEAK_DATA structure. If it’s non-NULL, use the specified URI.
2. If uri is NULL, check the data member. If it’s non-NULL, use the specified buffer.
3. If both uri and data are NULL, use the source callback function of type TTS_SOURCE_CB.

Refer to the description of the TTS_SPEAK_DATA structure for more details about the supported character sets for the input text data.
TtsSessionEnd

Disassociates a TTS engine instance with the application defined session identifier string previously set via TtsSessionStart.

\[
\text{TTSRETVAL TtsSessionEnd (}
\text{HTTSINSTANCE hTtsInst);}\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance</td>
</tr>
</tbody>
</table>
TtsSessionStart

Associates a TTS engine instance with an application defined session identifier string. That string is reported in all Vocalizer error, diagnostic, and event messages for the instance going forward. This is typically called once at the start of a sequence of requests for a single end-user, then TtsSessionEnd is called if the instance will then be re-used for another end-user.

TTSRETVAL TtsSessionStart ( 
    HTTSINSTANCE hTtsInst, 
    const LH_CHAR * szSessionID)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance</td>
</tr>
<tr>
<td>szSessionID</td>
<td>[in] Application-defined session identifier string</td>
</tr>
</tbody>
</table>
### TtsSetParams

Sets one or more TTS engine instance parameters to a specified value.

```c
TTSRETVLF TtsSetParams(
    HTTSINSTANCE hTtsInst,
    const TTS_PARAM_T * pParams,
    LH_U16 cParams)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
</tbody>
</table>
| pParams  | [in] List of parameters to set. Each element in this array specifies an individual parameter to set:  
  - Set pParams[index].nParam to the parameter ID (TTS_PARAM value).  
  - For each string parameter, set pParams[index].paramValue.array.pValue to a NULL-terminated string buffer containing the value.  
  - For each string parameter, set pParams[index].paramValue.array.nValue to the size of the value string, excluding the NULL terminator. |
| cParams  | [in] Number of parameters to set. |

Only TTS_VOLUME_PARAM and TTS_RATE_PARAM may be set while synthesis is in progress. For those two parameters, the change takes effect quickly, but because Vocalizer typically delivers audio faster than real time, there may be a significant number of audio buffers waiting for playback at the audio device; there may be a lag before the end user hears the effects. Applications can avoid this lag by doing rate and volume changes with the audio device instead of with Vocalizer, or by stopping and restarting synthesis, or by minimizing the number of audio buffers queued for the audio output device.

The following table lists the currently supported parameters and some of their corresponding default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Valid values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTS_FREQUENCY_PARAM</td>
<td>▪️ TTS_FREQ_8KHZ</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>▪️ TS_FREQ_11KHZ (currently not supported)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪️ TTS_FREQ_22KHZ.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stored in pParams[index].paramValue.nNo.</td>
<td></td>
</tr>
<tr>
<td>TTS_LANGUAGE_PARAM</td>
<td>ASCII character string value stored in the pParams[index].paramValue.array field; language name. If the instance’s current voice doesn’t match this language, TTS_VOICE_PARAM must also be specified to set a new voice.</td>
<td>N/A</td>
</tr>
<tr>
<td>TTS_LID_LANGUAGES_PARAM</td>
<td>An empty string or a comma-separated list of 3-letter language codes, stored in the pParams[index].paramValue.array field. The language identifier automatically restricts the matched languages to the ones where voices are installed; this further limits the permissible languages for language identification, and also sets the precedence order for matching languages when they have an equal confidence (first listed has a higher precedence). An empty string specifies all the languages currently supported by the language identifier (a subset of all Vocalizer languages).</td>
<td>N/A</td>
</tr>
<tr>
<td>Parameter</td>
<td>Valid values</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>TTS_LID_MODE_PARAM</td>
<td>Language identifier mode stored in pParams[index].paramValue.array field.</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>• rejection: if there is low confidence, the current language remains as-is.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• forced-choice: the language always switches to the language with the highest confidence.</td>
<td></td>
</tr>
<tr>
<td>TTS_LID_SCOPE_PARAM</td>
<td>Language identifier scope stored in pParams[index].paramValue.array field.</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>• user-defined: the language identifier is only used for blocks labeled with the &lt;ESC&gt;⟨unknown\control sequence.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• message: the language identifier is automatically used on each input message (typically a sentence).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• none: disables the language identifier</td>
<td></td>
</tr>
<tr>
<td>TTS_MARKER_MODE_PARAM</td>
<td>A bitmask of one or more of the following:</td>
<td>None.</td>
</tr>
<tr>
<td></td>
<td>• TTS_MRK_SENTENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TTS_MRK_WORD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TTS_MRK_PHONEME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TTS_MRK_BOOK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TTS_MRK_PARAGRAPH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stored in pParams[index].paramValue.nNo.</td>
<td></td>
</tr>
<tr>
<td>TTS_OUTPUT_TYPE_PARAM</td>
<td>• TTS_LINEAR_16BIT</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>• TTS_MULAW_8BIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TTS_ALAW_8BIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stored in pParams[index].paramValue.nNo.</td>
<td></td>
</tr>
<tr>
<td>TTS_PRODUCT_VERSION_PARAM</td>
<td>Read-only. Returns a product identifier string in the pParams[index].paramValue.array field.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Current product version string.</td>
<td></td>
</tr>
<tr>
<td>TTS_RATE_LARGESCALE_PARAM</td>
<td>1 to 100 (inclusive). Stored in pParams[index].paramValue.nNo.</td>
<td>50</td>
</tr>
<tr>
<td>TTS_SECURE_CONTEXT_PARAM</td>
<td>1 (true) or 0 (false). Stored in pParams[index].paramValue.nNo.</td>
<td>0</td>
</tr>
<tr>
<td>TTS_VOICE_PARAM</td>
<td>ASCII character string value stored in pParams[index].paramValue.array field; voice name. If the new voice is for a different language than the instance's current language, TTS_LANGUAGE_PARAM must also be specified.</td>
<td>N/A</td>
</tr>
<tr>
<td>TTS_VOLUME_LARGESCALE_PARAM</td>
<td>0 to 100 (inclusive). Stored in pParams[index].paramValue.nNo.</td>
<td>80</td>
</tr>
</tbody>
</table>
**TtsStop**

Stops the TTS conversion process initiated by a call to TtsProcessEx.

```c
TTSRETV AL TtsStop(
    HTTSINSTANCE hTtsInst)
```

Since the process functions are synchronous (blocking), the TtsStop function must be called from a different thread than the one that called the TtsProcessEx function, or from within the output callback function. The TtsStop function always succeeds unless the engine instance is not speaking.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance</td>
</tr>
</tbody>
</table>
TtsSystemInit

One-time initialization of the Vocalizer library. It must be called only once for each process (calling it more than once per process could cause memory leaks or crashes), and must be called prior to any other Vocalizer API function.

TTSRETVAL TtsSystemInit(
    const LH_CHAR * szUserConfigFile);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szUserConfigFile</td>
<td>[in] (Optional. May be NULL.) User XML configuration file that can override the standard Vocalizer XML configuration file settings.</td>
</tr>
</tbody>
</table>
TtsSystemTerminate

One-time shutdown of the Vocalizer library. It must be called only once for each process (calling it more than once per process could cause memory leaks or crashes), it must be called after closing all Vocalizer TTS engine instances, and no other Vocalizer API calls may be made for the remainder of the process lifetime after this API function is called.

TTSRETVAL TtsSystemTerminate(
    void);
TtsUnloadTuningData

Unload a user ruleset or ActivePrompt database that was previously loaded using TtsLoadTuningData, so that the TTS instance will no longer use that data.

TTSRETVAL TtsUnloadTuningData (  
  HTTSINSTANCE hTtsInst,  
  HTTSTUNINGDATA hTuningData  
);  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>hTuningData</td>
<td>[in] Handle to the tuning data instance to unload.</td>
</tr>
</tbody>
</table>
TtsUnloadUsrDictEx

Unloads a user dictionary instance, freeing the resources associated with it.

TTSRETRVAL TtsUnloadUsrDictEx(
    HTTSINSTANCE hTtsInst,
    HTTSUSRDIRCT hUsrDict);

A user dictionary instance can be unloaded when it is enabled; a user dictionary is either enabled implicitly by TtsLoadUsrDictEx or explicitly by TtsEnableUsrDictEx.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hTtsInst</td>
<td>[in] Handle to a TTS engine instance.</td>
</tr>
<tr>
<td>hUsrDict</td>
<td>[in] Handle to a dictionary instance.</td>
</tr>
</tbody>
</table>
User callbacks

This section describes the callbacks that the user (application) needs to implement and register when using the Vocalizer native API. Callbacks are registered by passing their pointers in the TTS_OPEN_PARAMS structure when TtsOpen is called.
TTS_SOURCE_CB

define TTSRETVAL (*TTS_SOURCE_CB)(
    LH_VOID* pAppData,
    LH_VOID* pInputBuffer,
    LH_U32 cInputBufferAlloc,
    LH_U32* pcInputBuffer);

This callback is only invoked when the input streaming mode of the TTS engine is enabled.

It is used by an engine instance to request a block of input text from the application. The function is called multiple times, allowing an unlimited amount of data to be delivered.

Each time the application puts data into pInputBuffer, the function should return TTS_SUCCESS.

When there is no more input data for the current TTS action the function should return TTS_ENDOFDATA. Then, the TTS engine knows the previous input corresponded with the last input block for a Speak action and the callback will no longer be called until the TtsProcessEx function returns. Any data in the buffer when TTS_ENDOFDATA is returned is ignored.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>pInputBuffer</td>
<td>[out] Pointer to a data buffer that is to be filled with the input text. This buffer is provided by the callback function, no need to allocate memory for it.</td>
</tr>
<tr>
<td>cInputBufferAlloc</td>
<td>Size in bytes of the buffer pointed to by pInputBuffer. This is the maximum amount of data that can be placed in pInputBuffer.</td>
</tr>
<tr>
<td>pcInputBuffer</td>
<td>[out] Number of bytes that were actually placed in the buffer.</td>
</tr>
</tbody>
</table>

Return values:

- TTS_SUCCESS
- TTS_ENDOFDATA

Note: This callback should not be registered unless both the uri and data member of the TTS_SPEAK_DATA structure can be NULL. This approach makes it possible to combine the TtsSource source callback function with the TtsProcessEx function.

When using the source callback method, the input text must be encoded with the character set indicated by the TTS_SPEAK_DATA structure passed to TtsProcessEx. See TTS_SPEAK_DATA on page 104.
TTS_DEST_CB

typedef LH VOID* (*TTS_DEST_CB)(
    LH VOID* pAppData,
    LH_U16 nOutputType,
    LH VOID* pAudio,
    LH_U32 cAudioBytes,
    LH_U32* pcAudioBufferAlloc);

This callback is invoked when the TTS engine instance needs to deliver output data to the application.

The main input parameters of the callback are the address of an application-provided buffer containing output data and the size in bytes of the data. The application provides the buffer for the engine to fill using the return value of the function, which is a pointer to the next buffer to be filled. The size of this application buffer is set in the *pcAudioBufferAlloc parameter before the function returns. The first call to TTS_DEST_CB passes in NULL for the output buffer and 0 for the data size, indicating that the TTS engine instance has not yet been given a buffer to fill. This also occurs each time the engine has finished processing a message unit.

In the simplest case, the application allocates a single buffer and returns its address every time, but the application might have a queue of buffers to prevent unnecessary copying of data.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>nOutputType</td>
<td>Data type that is being delivered. Currently only TTS_OUTPUTTYPE_PCM is supported.</td>
</tr>
<tr>
<td>pAudio</td>
<td>Pointer to the output data buffer.</td>
</tr>
<tr>
<td>cAudioBytes</td>
<td>Size of the buffer in bytes. For optimal performance, the buffer size should be set to a buffer sufficient for 0.5 seconds of audio at the current sampling rate, rounded up to the nearest multiple of 1024 bytes. See Optimal audio buffer size on page 75.</td>
</tr>
<tr>
<td>pcAudioBufferAlloc</td>
<td>[out] Size in bytes of the “new” data buffer passed back via the return value.</td>
</tr>
</tbody>
</table>

Return value(s):
Pointer to the next output buffer
TTS_EVENT_CB

typedef TTSRETVAL (*TTS_EVENT_CB)(
    LH_VOID* pAppData,
    LH_VOID* pEvent,
    LH_U16 cEventBytes,
    LH_U16 eEvent);

This callback is used to return markers to the application. Each marker represents a single event. There is one call to TTS_EVENT_CB for each separate marker. A marker is thrown before the call to TTS_DEST_CB that delivers the first audio sample aligned with it. In other words, the user receives the marker information in advance of the corresponding speech.

The different event types are explained under the TTS_EVENT data type topic.

You must specify which marker types to receive by calling TtsSetParams for the TTS_MARKER_MODE_PARAM parameter. See the description of the TTS_MARKER data structure for a description of the supported marker types.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>pEvent</td>
<td>Pointer to a buffer containing an event type specific structure providing the event related information. The type of the structure for each event type is listed below. The event type is the first item in each pair, the corresponding structure is the second item.</td>
</tr>
<tr>
<td>cEventBytes</td>
<td>Size of the buffer in bytes.</td>
</tr>
<tr>
<td>eEvent</td>
<td>Type of event that occurred. Can be one of the following:</td>
</tr>
</tbody>
</table>

- TTS_EVENT_SENTENCEMARK/TTS_MARKER_SENTENCE
- TTS_EVENT_BOOKMARK/TTS_MARKER_BOOK
- TTS_EVENT_WORDMARK/TTS_MARKER_WORD
- TTS_EVENT_PHONEMEMARK/TTS_MARKER_PHONEME
- TTS_EVENT_PARAGRAPHMARK/TTS_MARKER_PARAGRAPH
TTS_LOG_ERROR_CB

typedef LH VOID (*TTS_LOG_ERROR_CB)(
    LH VOID* pAppData,
    LH U32 u32ErrorID,
    const wchar_t* szErrorIDText,
    TTS_ERROR_SEVERITY eErrorSeverity,
    const VXIVector* pKeys,
    const VXIVector* pValues);

This callback is invoked when the TTS engine encounters errors and this callback is non-NULL with the log_cb_enabled XML configuration parameter set to true. The function provides detailed error information, allowing applications to report that information to system operators via application logs or other notification systems. However, the application should not presume that the error is fatal; instead, it should wait and examine the return code of the API function (TTSRETCVAL value).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>u32ErrorID</td>
<td>[out] Unique error ID number. This error ID corresponds to the “num” attribute within install_path\doc\VocalizerLogStrings.enu.xml.</td>
</tr>
<tr>
<td>szErrorIDText</td>
<td>[out] The associated error string for that error ID, corresponding to the content of the error with a matching “num” attribute within install_path\doc\VocalizerLogStrings.enu.xml.</td>
</tr>
<tr>
<td>eErrorSeverity</td>
<td>[out] The severity level for that error ID, corresponding to the “severity” attribute of the error with a matching “num” attribute within install_path\doc\VocalizerLogStrings.enu.xml.</td>
</tr>
<tr>
<td></td>
<td>- 1: critical error</td>
</tr>
<tr>
<td></td>
<td>- 2: severe error</td>
</tr>
<tr>
<td></td>
<td>- 3: warning</td>
</tr>
<tr>
<td></td>
<td>- 4: informational message</td>
</tr>
<tr>
<td></td>
<td>- 5: disabled error (callback will not be called for these)</td>
</tr>
<tr>
<td>pKeys</td>
<td>[out] Vector containing the keys of the error string.</td>
</tr>
<tr>
<td>pValues</td>
<td>[out] Vector containing the values of the error string's keys</td>
</tr>
</tbody>
</table>

Return values: None

Note: The VXIVector parameters represent the keys and values as they make up the arguments of the complete error string. The number of elements in both vectors should at all times be the same and their values should at all times be strings (VALUE_STRING).

A typical usage of these VXIVectors is demonstrated in the nvmcmdline sample application, which is shipped with the product.
Sample code:

```c
static LH_VOID TtsLogErrorCb(LH_VOID* lpAppData,
    LH_U32 iErrorID,
    const wchar_t* szErrorIDString,
    TTS_ERROR_SEVERITY eErrorSeverity,
    const VXIVector* lpKeys,
    const VXIVector* lpValues)
{
    std::wstring strMessage;
    const wchar_t *szSEVERITY[] = {
        L"UNKNOWN", /* Unknown error severity */
        L"CRITICAL", /* All instances out of service */
        L"SEVERE", /* Service affecting failure */
        L"WARNING", /* Application or non-service affecting failure */
        L"INFO", /* Informational message */
    };

    strMessage = szErrorIDString;

    for (size_t i = 0;
        (i < VXIVectorLength(lpKeys)) &&
        (i < VXIVectorLength(lpValues)); i++)
    {
        strMessage += L", ";
        strMessage += VXIStringCStr((const VXIString *)
            VXIVectorGetElement(lpKeys, i));
        strMessage += L"=";
        strMessage += VXIStringCStr((const VXIString *)
            VXIVectorGetElement(lpValues, i));
    }

    wprintf(L"%s %u: %s\n", szSEVERITY[eErrorSeverity], iErrorID,
        strMessage.c_str());
}
```
TTS_LOG_EVENT_CB

typedef LH_VOID (*TTS_LOG_EVENT_CB)(
    LH_VOID* pAppData,
    const wchar_t* szEvent,
    const VXIVector* pKeys,
    const VXIVector* pValues);

This callback is invoked by the TTS engine to report events for application tuning and capacity planning purposes when this callback is non-NULL and the log_cb_enabled XML configuration parameter is true. The function provides detailed event information, allowing applications to generate application trace logs or capture and report statistical information such as license use.

For a list of the events and tokens, see Application event logs on page 65.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>szEvent</td>
<td>[out] Event name.</td>
</tr>
<tr>
<td>pKeys</td>
<td>[out] Vector containing the event token key strings.</td>
</tr>
<tr>
<td>pValues</td>
<td>[out] Vector containing the event token value strings.</td>
</tr>
</tbody>
</table>

Return values: None

Note: The VXIVector parameters represent the keys and values as they make up the arguments of the complete event string. The number of elements in both vectors should at all times be the same and their values should at all times be strings (VALUE_STRING).

A typical usage of these VXIVectors is demonstrated in the nvncmdline sample application, which is shipped with the product.
Sample code:

```c
static LH_VOID TtsLogEventCb(LH_VOID* lpAppData,
    const wchar_t* szEvent,
    const VXIVector* lpKeys,
    const VXIVector* lpValues)
{
    std::wstring strMessage;
    for (size_t i = 0;
        (i < VXIVectorLength(lpKeys)) &&
        (i < VXIVectorLength(lpValues)); i++)
    {
        if (i > 0) strMessage += L", ";
        strMessage += VXIStringCStr((const VXIString *)
            VXIVectorGetElement(lpKeys, i));
        strMessage += L"=\n";
        strMessage += VXIStringCStr((const VXIString *)
            VXIVectorGetElement(lpValues, i));
    }

    // This demonstrates logging the event to the console
    wprintf(L"EVENT %s: %s\n", szEvent, strMessage.c_str());

    // This demonstrates forwarding the event to the Nuance
    // Recognizer for inclusion in its event logs, giving a
    // merged Recognizer and Vocalizer event log for application
    // tuning and capacity planning.
    //
    SWIrecLogEvent(rec, szEvent, strMessage.c_str());
}
```
TTS_LOG_DIAGNOSTIC_CB

typedef LH_VOID (*TTS_LOG_DIAGNOSTIC_CB)(
    LH_VOID* pAppData,
    LH_S32 s32Level,
    const LH_CHAR* szMessage);

This callback is invoked when this callback is non-NULL, the log_cb_enabled XML configuration parameter is true, and the log_level XML configuration parameter is set to enable diagnostic logging. By default, log_level is set to disable diagnostic logging, so this callback is not normally called. Diagnostic logging (and thus this callback) should normally be disabled on production systems, and only turned on when necessary for debugging system problems, because this logging significantly impacts system performance.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAppData</td>
<td>Application data pointer that was passed into TtsOpen.</td>
</tr>
<tr>
<td>s32Level</td>
<td>[out] Diagnostic message level, matching the log_level value where this message starts appearing.</td>
</tr>
<tr>
<td>szMessage</td>
<td>[out] Diagnostic message string.</td>
</tr>
</tbody>
</table>

Return values: None

Sample code:

```c
static LH_VOID TtsLogDiagnosticsCb(LH_VOID* lpAppData,
    LH_S32 iLevel, const LH_CHAR* szMessage)
{
    printf("%d: %s\n", iLevel, szMessage);
}
```
Error codes

This is a list of all the possible error codes returned by Vocalizer API functions:

- TTS_SUCCESS
- TTS_ERROR
- TTS_E_ALREADY_DEFINED
- TTS_E_BAD_FREQ
- TTS_E_BAD_HANDLE
- TTS_E_BAD_LANG
- TTS_E_BAD_OUTPUT
- TTS_E_BAD_TYPE
- TTS_E_BAD_VOICE
- TTS_E_BADCOMMAND
- TTS_E_BUF_TOO_SMALL
- TTS_E_CONVERSION_FAILED
- TTS_E_DICTIONARY_ALREADY_UNLOADING
- TTS_E_END_OF_INPUT
- TTS_E_ENGINE_ALREADY_INITIALIZED
- TTS_EENGINE_ALREADY_UNINITIALIZED
- TTS_EENGINE_NOT_FOUND
- TTS_EENGINE_OVERLOAD
- TTS_E_FEAT_EXTRACT
- TTS_E_INET_EXCEED_MAXSIZE
- TTS_E_INET_FATAL
- TTS_E_INET_FETCH_ERROR
- TTS_E_INET_FETCH_TIMEOUT
- TTS_E_INET_INPUTOUTPUT
- TTS_E_INET_INVALID_PROP_NAME
- TTS_E_INET_INVALID_PROP_VALUE
- TTS_E_INET_NON_FATAL
- TTS_E_INET_NOT_ENTRY_CREATED
- TTS_E_INET_NOT_ENTRY_LOCKED
- TTS_E_INET_NOT_MODIFIED
- TTS_E_INET_PLATFORM
- TTS_E_INET_UNMAPPED
- TTS_E_INET_UNSUPPORTED
- TTS_E_INET_WOULD_BLOCK
- TTS_E_INSTANCE_BUSY
- TTS_E_INTERNAL_ERROR
- TTS_E_INVALID_POINTER
- TTS_E_INVALIDINST
- TTS_E_INVALIDPARM
- TTS_E_KEY_EXISTS
- TTS_E_LIC_LICENSE_ALLOCATED
- TTS_E_LIC_LICENSE_FREED
- TTS_E_LIC_NO_LICENSE
- TTS_E_LIC_SYSTEM_ERROR
- TTS_E_LIC_UNSUPPORTED
- TTS_E_MAXCHANNELS
- TTS_E_MISSING_FUNC
- TTS_E_MISSING_SL
- TTS_E_MODULE_NOT_FOUND
- TTS_E_NETWORK_CONNECTIONCLOSED
- TTS_E_NETWORK_CONNECTIONREFUSED
- TTS_E_NETWORK_FUNCTION_ERROR
- TTS_E_NETWORK_INTERNAL_ERROR
- TTS_E_NETWORK_OPENPORTFAILED
- TTS_E_NETWORK_PROBLEM
- TTS_E_NETWORK_RETRANSMIT
- TTS_E_NETWORK_SENDFAILED
- TTS_E_NETWORK_TIMEOUT
- TTS_E_NO_INPUT_TEXT
- TTS_E_NO_KEY
- TTS_E_NO_MATCH_FOUND
- TTS_E_NO_MORE_MEMBERS
- TTS_E_NOT_COMPATIBLE
- TTS_E_NOT_FOUND
- TTS_E_NOTINITIALIZED
- TTS_E_NULL_POINTER
- TTS_E_NULL_STRING
- TTS_E_OUT_OF_RANGE
- TTS_E_OUTOFMEMORY
- TTS_E_PARAMERROR
- TTS_E_QUEUE_EMPTY
- TTS_E_QUEUE_FULL
- TTS_E_SSML_PARSE_ERROR
- TTS_E_SYSTEMERROR
- TTS_E_TRANS_EMAIL
- TTS_E_UDCT_ACTIONNOTALLOWED
- TTS_E_UDCT_ALWAYENABLED
- TTS_E_UDCT_ALWAYINENG
- TTS_E_UDCT_BUSY
- TTS_E_UDCT_COULDNOTOPENFILE
- TTS_E_UDCT_DATAFAILURE
- TTS_E_UDCT_DUPLSOURCEWORD
- TTS_E_UDCT_FILEIO
- TTS_E_UDCT_FILEREADERROR
- TTS_E_UDCT_FILEWRITEERROR
- TTS_E_UDCT_FULLENG
- TTS_E_UDCT_INVALIDENGHNDL
- TTS_E_UDCT_INVALIDENTRYDATA
- TTS_E_UDCT_INVALIDFILE
- TTS_E_UDCT_INVALIDHNDL
- TTS_E_UDCT_INVALIDIDOPER
- TTS_E_UDCT_LANGUAGECONFLICT
- TTS_E_UDCT_MAXDESTSPACE
- TTS_E_UDCT_MAXENG
- TTS_E_UDCT_MAXENTRIES
- TTS_E_UDCT_MAXSOURCESPACE
- TTS_E_UDCT_MEMALLOC
- TTS_E_UDCT_NOENTRY
- TTS_E_UDCT_NOT_LOCAL
- TTS_E_UDCT_NOTLOADED
- TTS_E_UDCT_OHERUSER
- TTS_E_UDCT_PRIORITYINUSE
- TTS_E_UDCT_READONLY
- TTS_E_UDCT_STILLINUSE
- TTS_E_UDCT_WRONGTXTDCTFORMAT
- TTS_E_UNKNOWN
- TTS_E_WRONG_STATE
- TTS_W_ENDOFINPUT
- TTS_W_UDCT_ALREADYLOADED
- TTS_W_WARNING
Chapter 7

Microsoft SAPI 5 compliance

This section lists which Microsoft SAPI 5 Text-To-Speech API functions are supported by Vocalizer. For more details on each of these functions, see the Microsoft documentation for the Microsoft Speech SDK 5.1. For more information on Vocalizer support for Microsoft SAPI 5 phoneme sets, see the Vocalizer user guide for each Vocalizer language.

SAPI API support overview

This provides an overview of supported Microsoft SAPI 5 API methods.

SAPI 5 text-to-speech engine interface

Vocalizer implements the Microsoft SAPI 5 Text-To-Speech Engine interface. Applications do not directly call this interface; they instead call into Microsoft provided SAPI 5 components that are implemented on top of this interface. The following table lists the SAPI 5 Text-To-Speech Engine interfaces and methods, describing which of these are supported by Vocalizer.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Function name</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISpTTSEngine</td>
<td>Speak</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetOutputFormat</td>
<td>Supported</td>
</tr>
<tr>
<td>ISpTTSEngineSite</td>
<td>ISpEventSink</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetActions</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetRate</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetVolume</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetSkipInfo</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>CompleteSkip</td>
<td>Supported</td>
</tr>
</tbody>
</table>

SAPI 5 text-to-speech application interface

Applications directly call this interface, which is implemented by Microsoft provided SAPI 5 components. Those Microsoft components then call into Vocalizer SAPI 5 Text-To-Speech Engine implementation. The following table lists the SAPI 5
Text-To-Speech Engine interfaces and methods, describing which of these are supported when the Vocalizer engine is used.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Function name</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>IspVoice</td>
<td>SetOutput</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetOutputObjectToken</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetOutputStream</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Pause</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Resume</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SetVoice</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetVoice</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Speak</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SpeakStream</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetStatus</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Skip</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td>SetPriority</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetPriority</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SetAlertBoundary</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetAlertBoundary</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SetRate</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetRate</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SetVolume</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetVolume</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>WaitUntilDone</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SetSyncSpeakTimeout</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>GetSyncSpeakTimeout</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>SpeakCompleteEvent</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>IsUISupported</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td>DisplayUI</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

SAPI API support details

ISpVoice is the only public Microsoft SAPI 5 interface for application access to 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 also be
changed in real time using SAPI API methods such as 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.

This section provides a list of Microsoft SAPI 5 Text-To-Speech interface (ISpVoice) member functions with notes on Vocalizer specific extensions or limitations. For a description of each member function, see the Microsoft Speech SDK v5.1 Reference chapter on Text-To-Speech interfaces (ISpVoice).

**ISpVoice::ISpEventSource** No Vocalizer specific remarks.

**ISpVoice::SetOutput** Vocalizer only supports 8 kHz and 22 kHz voices. If the application chooses other frequencies, then the Microsoft SAPI 5 layer will use conversion software installed in the PC, which might cause speech quality degradation and performance degradation. If other frequencies are required, speak to your Nuance sales representative to see if a different Vocalizer product variant would be a better match for your application.

**ISpVoice::GetOutputObjectToken** See ISpVoice::SetOutput.

**ISpVoice::GetOutputStream** No Vocalizer specific remarks.

**ISpVoice::Pause** No Vocalizer specific remarks.

**ISpVoice::Resume** No Vocalizer specific remarks.

**ISpVoice::SetVoice** No Vocalizer specific remarks.

**ISpVoice::GetVoice** No Vocalizer specific remarks.

**ISpVoice::Speak** No Vocalizer specific remarks.

**ISpVoice::SpeakStream** No Vocalizer specific remarks.

**ISpVoice::GetStatus** No Vocalizer specific remarks.

**ISpVoice::Skip** Not supported by Vocalizer.

**ISpVoice::SetPriority** No Vocalizer specific remarks.

**ISpVoice::GetPriority** No Vocalizer specific remarks.

**ISpVoice::SetAlertBoundary** No Vocalizer specific remarks.

**ISpVoice::GetAlertBoundary** No Vocalizer specific remarks.

**ISpVoice::SetRate** No Vocalizer specific remarks.

**ISpVoice::GetRate** No Vocalizer specific remarks.

**ISpVoice::SetVolume** SAPI enforces its own default volume. This means that the ttssapi.xml configuration file does not support specifying the default rate and volume, and that the Vocalizer default volume when used via SAPI is 100 (the maximum volume) instead of 80.

**ISpVoice::GetVolume** See ISpVoice::SetVolume note.

**ISpVoice::WaitUntilDone** No Vocalizer specific remarks.

**ISpVoice::SetSyncSpeakTimeout** No Vocalizer specific remarks.

**ISpVoice::GetSyncSpeakTimeout** No Vocalizer specific remarks.
ISpVoice::SetSyncSpeakTimeout No Vocalizer specific remarks.

ISpVoice::SpeakCompleteEvent No Vocalizer specific remarks.

ISpVoice::IsUISupported This member function is not supported by Vocalizer as Vocalizer does not provide a custom SAPI 5 Control Panel User Interface.

ISpVoice::DisplayUI This member function is not supported by Vocalizer as Vocalizer does not provide a custom SAPI 5 Control Panel User Interface.

SAPI 5 XML tag support

In this section you will find an alphabetical list of the Microsoft SAPI 5 Text-To-Speech XML tags. These 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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Gives a description of the XML tag.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Displays the syntax of the XML tag.</td>
</tr>
<tr>
<td>Comments</td>
<td>Gives remarks that are specific to Vocalizer's support of the XML tag.</td>
</tr>
<tr>
<td>Example</td>
<td>Shows how to use the XML tag.</td>
</tr>
</tbody>
</table>

Note: Incorrectly specified SAPI XML control tags are ignored and treated as white space. Also note that SAPI XML control tags that are not described in the following table are not supported.

For more information on the use and syntax of SAPI 5 XML tags, see the Text-To-Speech Interface chapter of the Microsoft Speech SDK V5.1 Reference manual.

<table>
<thead>
<tr>
<th>Control Tag</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Bookmark&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Context&gt;</td>
<td>Partially supported; see the Language Supplement</td>
</tr>
<tr>
<td>&lt;Emph&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Lang&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Partofsp&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Pitch&gt;</td>
<td>Not Supported</td>
</tr>
<tr>
<td>&lt;Pron&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Rate&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Silence&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Spell&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Voice&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>&lt;Volume&gt;</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Bookmark

This XML tag indicates a bookmark in the text.

Syntax:

<bookmark mark="string"/>
Example:
This sentence contains a <bookmark mark="bookmark_one"/> bookmark.

Context

This XML tag sets the context for the text that follows, determining how specific strings should be spoken. This is equivalent to the <ESC>\tn\ native control sequence. See the appropriate Language Supplement for the supported <ESC>\tn\ types.

Syntax:

例:
例：

Emph

This XML tag emphasizes a sentence to be spoken.

Syntax:

例：
例：

Lang

This XML tag indicates a language change in the text. This tag is handled by the Microsoft SAPI 5 Layer.

Syntax:

例：
例：

Partofsp

This XML tag indicates the part-of-speech of the next word. This tag is effective only when the word is in a SAPI 5 lexicon and has the same part-of-speech setting as in the lexicon.

Syntax:

Comments:

例：
例：

The following part-of-speech types are supported:

- <Partofsp Part="noun"/>
- <Partofsp Part="verb"/>
- <Partofsp Part="modifier"/>
- <Partofsp Part="function"/>
Pitch

This XML tag is used to control the pitch of a voice.

Syntax:

\[
\text{<Pitch Absmiddle=string> Input Text </Pitch>}
\]

Comments:

Vocalizer does not support this tag.

Example:

\[
\text{<pitch absmiddle="5">This is a test.</pitch>}
\]

Pron

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 SAPI 5 phonemes (not native Vocalizer L&H+ phonemes).

Syntax:

\[
\text{<pron sym=phonetic string>}
\]

or

\[
\text{<pron sym=phonetic string>Input text</pron>}
\]

Comments:

The supported Microsoft SAPI 5 phoneme symbols can be found in the Vocalizer user guide for each language. If no phoneme table is available for a specific language, then this tag is not supported for that language.

Example:

\[
\text{<pron sym="h eh l ow \& w er l l d"> hello world </pron>}
\]

Rate

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:

\[
\text{<rate absspeed=number>Input text</rate>}
\]
or

\(<rate\ speed=number>Input\ text</rate>\)

Examples:

\(<rate\ absspeed="5">This\ is\ a\ sentence.</rate>\)
\(<rate\ speed="5">This\ is\ a\ faster\ sentence.\ </rate>\)
\(<rate\ speed="-5">This\ is\ a\ slower\ sentence.\ </rate>\)

Silence

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:

\(<silence\ msec=number>Input\ text</silence>\)

Example:

\(<silence\ msec="500"/>\ This\ is\ a\ sentence.\)

Spell

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:

\(<spell>Input\ text</spell>\)

Example:

\(<spell>UN</spell>\)

Voice

The Voice tag is completely implemented by the Microsoft provided SAPI 5 components, with no control by Vocalizer other than publishing the Vocalizer voice attributes.

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 for the EnumerateTokens and SpFindBestToken methods in the SAPI 5 ISpObjectTokenCategory interface. The selected voice follows exactly the same rules as the latter of these two functions. That is, it selects a voice where all the required attributes are present, and where 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 Reference manual.

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>

Examples:

<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>

Volume

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>

Example:

<volume level="50">This is a sentence.</volume>

Using the Microsoft SAPI 5 lexicon

Microsoft SAPI 5 provides lexicons so that users and applications can specify pronunciation and part of speech information for particular words. As such, SAPI compliant text-to-speech engines like Vocalizer need to support these lexicons to guarantee uniformity of pronunciation and part of speech information.

There are two types of lexicons in SAPI: user lexicons and application lexicons.

- **User lexicons**: Each user who logs in to a computer has a user lexicon. Initially, this lexicon is empty; words can be added either programmatically, or by using an engine’s add/remove words UI component. (For example, the sample application Dictation Pad provides an Add/Remove Words dialog.)

- **Application lexicons**: Applications can create and ship their own lexicons of specialized words. These lexicons are fixed and cannot be edited.

Detailed information on how to use the Microsoft SAPI 5 lexicons can be found in the manual *Microsoft Speech SDK V5.1*, chapter “ISpLexicon Interface”.

Using Vocalizer user dictionaries

The Vocalizer SAPI 5 engine can be configured to automatically load Vocalizer proprietary user dictionaries that tune Vocalizer pronunciations, as well as Vocalizer user rulesets and ActivePrompt databases. This mechanism provides a better alternative to the Microsoft defined SAPI 5 lexicon solution, as Vocalizer user dictionaries can specify native Vocalizer phoneme strings for more accurate pronunciations, and Vocalizer user
dictionaries are stored in files which are easier to manage than registry-based Microsoft SAPI 5 lexicons.

To configure default Vocalizer user dictionaries, modify the `<default_dictionaries>` element within the Vocalizer SAPI 5 configuration file (install_path\config\tssapi.xml within the Vocalizer installation directory). See Configuration files on page 80 for more information.

Using SSML via SAPI

Vocalizer supports W3C SSML markup for text submitted via the Microsoft SAPI API. However, the application must use special methods to do so, and the available methods depend on the installed version of the Microsoft SAPI components (sapi.dll). Nuance does not redistribute these components; they are a standard part of Windows. Contact Microsoft for more information.

Microsoft SAPI 5.2 and earlier do not have native support for SSML markup. By default, they try to interpret the SSML as SAPI 5 XML tags, strip out the markup, and speak the resulting text. This behavior results in incorrect text-to-speech output. To work around this, specify the SPF_IS_NOT_XML flag when calling the ISpVoice::Speak method. While this flag may seem counter-intuitive, this blocks the Microsoft SAPI XML parser, allowing the SSML markup to be passed to Vocalizer as-is. As long as the SSML markup is encoded as little-endian UTF-16 and has a valid SSML header, Vocalizer’s SAPI integration auto-detects the SSML markup and parses it during Vocalizer’s internal SSML parser, with SSML conformance as described in Vocalizer SSML support on page 161.

Microsoft SAPI 5.3 and later have native support for SSML markup, where the Microsoft SAPI library parses the SSML markup itself, then passes parsed SSML fragments to Vocalizer for synthesis. (This behavior is similar to how all versions of SAPI handle SAPI 5 XML tags.) However, this conversion loses some SSML information, and SAPI 5.3 currently has bugs that block proper handling for some SSML constructs, so SSML conformance is lower than Vocalizer’s built-in SSML parser. Thus, the SPF_IS_NOT_XML method described for SAPI 5.2 above is recommended; this method continues to work with Microsoft SAPI 5.3.
Speech Synthesizer Markup Language (SSML) is part of a set of markup language specifications for voice browsers, as established by the World Wide Web Consortium (W3C). SSML was designed to provide 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 provide a standard way to control aspects of speech such as pronunciation, volume, and rate.

The Nuance Vocalizer for Network SDK provides a built-in preprocessor that supports most of the SSML 1.0 September 2004 Recommendation (REC). Moreover Vocalizer extends SSML with a few Nuance-specific elements/attributes. Vocalizer includes a set of SSML schema documents that are extended to recognize the Vocalizer SSML extensions. These are available as install_path/doc/synthesis.xsd and install_path/doc/synthesis-core.xsd.

Please refer to the appropriate Language Supplement for information on language-specific support.

Links

For background on SSML, refer to these W3C specification Web pages:


SSML compliance

Vocalizer is designed to meet W3C specification requirements, subject to the qualifications and exceptions described in this section.

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 where they cannot be implemented due to the nature of the Vocalizer engine. In such cases the markup element is detected and parsed, but ignored. This applies to the following elements and or properties:

- The <emphasis> element: the “none” level is not supported. Using this element does not necessarily lead to audible differences, as the system may elect to ignore these targets in order to produce optimal natural speech output.
- 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. At present, this would require the use of custom voices.

The `<prosody>` element is handled as follows:

- Duration, pitch, pitch-range, and contour values are ignored.

The `<break>` element: if you set `<break strength="none">`, this will only have an audible effect when the TTS engine would have inserted a sentence break without the `<break>` element.

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 Say-as support on page 166 for information on Vocalizer’s support of these attributes.

XML document encodings

Vocalizer uses the Apache Xerces-C XML parser with the IBM ICU library for supporting a wide variety of character encodings. The UTF-16 character set is recommended for optimal performance, as Vocalizer and Xerces-C use UTF-16 for all their internal processing. The next best choice is UTF-8. For more information on Xerces-C and ICU, see Copyright and licensing for third party software on page 181.

For more information about supported character sets, see the following web sites:

- http://www.iana.org/assignments/character-sets

Language codes

SSML requires specifying the document language using xml:lang, an IETF language code. For a list of IETF language codes supported by Vocalizer, see Vocalizer languages on page 179.

SSML validation

By default, Vocalizer does strict validation of input SSML documents against compiled-in versions of the SSML schema documents with Vocalizer specific enhancements (install_path\doc\synthesis.xsd and install_path\doc\synthesis-core.xsd within the Vocalizer installation). This is the most robust setting because it ensures the application and Vocalizer engine properly cooperate, rather than forcing Vocalizer to try and handle bad SSML input that may lead to strange TTS output. However, if required, the validation mode can be adjusted using the ssml_validation parameter in the Vocalizer configuration file.

When validation errors occur, Vocalizer 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 TtsProcessEx API call. When this occurs, review the error logs for details about the SSML document errors.
Audio fetch error handling

When SSML <audio> is used to insert an audio recording, and the fetch fails or the audio format is not supported, Vocalizer proceeds with using the <audio> element’s content as fallback as described in the SSML 1.0 Recommendation. However, the default Vocalizer handling of certain error cases is different then what the SSML 1.0 Recommendation states.

If the <audio> fallback exists and can be spoken, the speak request succeeds, but Vocalizer logs a single warning message to indicate the audio URI that could not be spoken and note that its fallback was used. This error message (error ID 1422) is specific to the case of a failed <audio> insertion that has successful fallback, so if that message is undesirable, it can be disabled as described in Customizing error messages on page 64.

If the <audio> fallback does not exist or cannot be spoken (such as another <audio> element without fallback that could not be spoken), Vocalizer logs a series of errors that indicate the failure details, including the audio URI. Then if the ssml_validation parameter in the Vocalizer configuration parameter is set to “strict” (the default), the speak request fails with a TTS_E_SSML_PARSE_ERROR error code, with no audio delivered to the application. If the ssml_validation parameter is set to any other value, the speak request proceeds, merely omitting the <audio> element.

The ssml_validation “strict” setting gives the most robust behavior, making sure the listener doesn’t hear confusing or misleading audio for the case of a failed <audio> element with no or failed fallback. However, Vocalizer’s behavior in “strict” validation mode is stricter then the SSML 1.0 Recommendation: that specification says the TTS engine should just notify the application of the error but proceed with the speak request (Vocalizer’s behavior in “warn” and “none” validation mode). However, Vocalizer is usually used in environments where there is no mechanism for supporting non-fatal error notifications (for example VoiceXML, MRCP, and SAPI 5 all lack this), so the application cannot apply an appropriate fallback strategy of its own such as transferring a call to a human unless the speak request completely fails. Thus Vocalizer’s default behavior of “strict” makes the overall solution more robust.

If this is undesirable, change ssml_validation to “warn,” or simply ensure all SSML requests contain text-to-speech fallback for all <audio> elements, even if it is simply a very small pause by using a <break strength="x-weak"> element.

Native control sequences

Vocalizer supports using native control sequences within SSML documents. However, the default <ESC> character used to initiate a native control sequence, “\x1B”, is not allowed in XML documents as per the W3C XML 1.0 specification. The escape_sequence Vocalizer configuration file parameter can be used to specify an alternate sequence that permits using native control sequences within SSML documents.

However, beware that mixing native control sequences with SSML markup can lead to unexpected behavior, because Vocalizer internally handles SSML by expanding it to native control sequences as well. The Vocalizer 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, Nuance 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.

**Volume scale conversion**

The realization of volume numerical values is SSML compliant. The default value is 100, and the scale is amplitude linear. Note that although the SSML specifies that the range is 0 to 100, we internally support a more extended range (0 to 200). The values above 100 can only be reached via relative changes or the symbolic values “loud” and “x-loud”.

The table below describes the mapping between the SSML volume scale and the Nuance native volume scale (where the volume value is an integer in the range 0 to 100 which can be set via the native \esc\vol=x\ markup).

<table>
<thead>
<tr>
<th>SSML volume value</th>
<th>Amplitude amplification factor</th>
<th>Loudness in dB</th>
<th>Nuance volume value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
<td>∞ dB</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0.10</td>
<td>-20.0 dB</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>0.20</td>
<td>-14.0 dB</td>
<td>33</td>
</tr>
<tr>
<td>30</td>
<td>0.30</td>
<td>-10.5 dB</td>
<td>45</td>
</tr>
<tr>
<td>40</td>
<td>0.40</td>
<td>-8.0 dB</td>
<td>53</td>
</tr>
<tr>
<td>50</td>
<td>0.50</td>
<td>-6.0 dB</td>
<td>60</td>
</tr>
<tr>
<td>60</td>
<td>0.60</td>
<td>-4.4 dB</td>
<td>65</td>
</tr>
<tr>
<td>70</td>
<td>0.70</td>
<td>-3.1 dB</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>0.80</td>
<td>-1.9 dB</td>
<td>74</td>
</tr>
<tr>
<td>90</td>
<td>0.90</td>
<td>-0.9 dB</td>
<td>77</td>
</tr>
<tr>
<td>100</td>
<td>1.00</td>
<td>0.0 dB</td>
<td>80</td>
</tr>
<tr>
<td>(141)</td>
<td>1.41</td>
<td>+3.0 dB</td>
<td>90</td>
</tr>
<tr>
<td>(200)</td>
<td>2.00</td>
<td>+6.0 dB</td>
<td>100</td>
</tr>
</tbody>
</table>

The formula for converting the SSML value Vssml to the amplification factor A is very simple:

\[
A = \frac{V_{ssml}}{100}
\]

The formula for converting a non-zero amplification factor A to the corresponding Vocalizer volume value Xvocalizer is:

\[
X_{vocalizer} = \text{Round}((20 \times \log10(A) / 0.30) + 80)
\]

The formula for converting a non-zero Vocalizer value Xvocalizer to the dB value Y is:

\[
Y \text{ (dB)} = (X_{vocalizer} - 80) \times 0.3 \text{ dB}
\]
The SSML symbolic values are mapped as follows:

<table>
<thead>
<tr>
<th>SSML volume value</th>
<th>Symbolic value</th>
<th>Amplitude amplification factor</th>
<th>Loudness in dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>silent</td>
<td>0.00</td>
<td>∞ dB</td>
</tr>
<tr>
<td>18</td>
<td>x-soft</td>
<td>0.18</td>
<td>-15.0 dB</td>
</tr>
<tr>
<td>50</td>
<td>soft</td>
<td>0.50</td>
<td>-6.0 dB</td>
</tr>
<tr>
<td>100</td>
<td>medium</td>
<td>1.00</td>
<td>0.0 dB</td>
</tr>
<tr>
<td>(141)</td>
<td>loud</td>
<td>1.41</td>
<td>+3.0 dB</td>
</tr>
<tr>
<td>(200)</td>
<td>x-loud</td>
<td>2.00</td>
<td>+6.0 dB</td>
</tr>
</tbody>
</table>

**Rate scale conversion**

We fully support SSML rate markup. The following tables/rules can be used to map SSML markup to equivalent Vocalizer native markup.

The default value is 1.00.

<table>
<thead>
<tr>
<th>SSML “number” value</th>
<th>Symbolic value</th>
<th>SSML percentage w.r.t. voice default</th>
<th>Vocalizer native rate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>x-slow</td>
<td>-50%</td>
<td>50</td>
</tr>
<tr>
<td>0.70</td>
<td>slow</td>
<td>-30%</td>
<td>70</td>
</tr>
<tr>
<td>1.00</td>
<td>medium</td>
<td>+0%</td>
<td>100</td>
</tr>
<tr>
<td>1.60</td>
<td>fast</td>
<td>+60%</td>
<td>160</td>
</tr>
<tr>
<td>2.50</td>
<td>x-fast</td>
<td>+150%</td>
<td>250</td>
</tr>
</tbody>
</table>

SSML descriptive and number values change the rate against the voice default. All other rate changes are relative against the (XML) parent element.

Here are the formulas used to convert an SSML `<prosody rate="Xssml">` value into a Vocalizer `<ESC><rate=Yvocalizer\>` value. Do note that rate changes are relative to the parent element; you must keep track of all changes to your SSML value before converting. For example, to change from 50 to 25 is -50%; but to restore the original value from 25 to 50 is +100%.

For increasing the rate (Xssml > 1.0)

\[
Y_{vocalizer} = \text{Round}(100 \times X_{ssml})
\]

When decreasing the rate (Xssml < 1.0)

\[
Y_{vocalizer} = \text{Round}(50 - (1 - X_{ssml}) \times 100)
\]

**Break implementation**

A `<break strength="xxx">` element is implemented as a pause of a certain duration, so it maps directly to a Vocalizer `<ESC><pause=x\>` tag. The only exception is the SSML `<break strength="none">` element, which maps to a Vocalizer `<ESC><eos=0\>` tag.
The table below specifies the mapping and the corresponding native markup.

<table>
<thead>
<tr>
<th>Symbolic value</th>
<th>Duration in ms of the pause</th>
<th>Native markup value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-weak</td>
<td>100</td>
<td>&lt;ESC&gt;\pause=100\</td>
</tr>
<tr>
<td>weak</td>
<td>200</td>
<td>&lt;ESC&gt;\pause=200\</td>
</tr>
<tr>
<td>medium</td>
<td>400</td>
<td>&lt;ESC&gt;\pause=400\</td>
</tr>
<tr>
<td>strong</td>
<td>700</td>
<td>&lt;ESC&gt;\pause=700\</td>
</tr>
<tr>
<td>x-strong</td>
<td>1200</td>
<td>&lt;ESC&gt;\pause=1200\</td>
</tr>
<tr>
<td>none</td>
<td>0</td>
<td>&lt;ESC&gt;\eos=0\</td>
</tr>
</tbody>
</table>

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

**Say-as support**

While the W3C SSML 1.0 Recommendation specifies the say-as element and its semantics with the interpret-as, format, and detail attributes, it does not define any specific say-as types. Vocalizer implements a significant number of built-in SSML say-as types, and the user rulesets feature can override those types or add entirely new SSML say-as types.

Vocalizer does this by mapping each say-as element to a native <ESC>\tn\ control sequence by building a string that consists of the interpret-as, format, and detail attributes with colon separators, using an empty string for unspecified attributes, and then stripping trailing colons off the end of the final string. Vocalizer then uses the following precedence for handling the say-as type:

- Look for a user ruleset with a matching “type” attribute (which may include wildcards).
- Look for a matching Vocalizer front-end <ESC>\tn\ type.
- Trim off the “detail” attribute, then look for a matching Vocalizer front-end <ESC>\tn\ type. If found, it logs a warning, but then proceeds with that revised type.
- Trim off the “format” and “detail” attributes, then look for a matching Vocalizer front-end <ESC>\tn\ type. If found, it logs a warning, but then proceeds with that revised type.
- Log a warning, then handle the text as if it was plain text (as if the <say-as> wrapper was discarded).

For details on the supported SSML say-as and <ESC>\tn\ types, see the appropriate Language Supplement for an exact list of the types supported for your language and for language-specific implementation details that apply for each type.

**The lexicon element**

Nuance supports loading user dictionaries, user rulesets, and ActivePrompt databases through the SSML <lexicon> element. The value for the uri attribute must be a valid URI to a user dictionary, user ruleset, or ActivePrompt database.

The “type” attribute is optional. If specified this overrides the MIME content type returned by the web server (for http:// access) or extension mapping rules (for local file access). Valid values are:

- application/edct-bin-dictionary for a Vocalizer binary format user dictionary
- application/x-vocalizer-rettt+text for a text user ruleset
- application/x-vocalizer-rettt+bin for a binary user ruleset
- application/x-vocalizer-activeprompt-db for an ActivePrompt database, optionally with "mode=automatic" appended to override its default matching mode to automatic

When using a web server, remember to add entries to your web server’s MIME content type mapping table to associate the file extensions with the correct MIME content types. If you do not, the lexicon load fails unless the “type” attribute is specified. (Other approaches for setting the MIME content type may also be possible, depending on the web server used.) For local file access, the inet_extension_rules Vocalizer configuration file parameter is used to map file extensions to the MIME content type.

All <lexicon> elements are parsed, and tuning data loaded before starting text to speech conversion. This tuning data is unloaded when the last sample buffer is generated, or when the TTS process is stopped, so <lexicon> elements only affect the current synthesis request.

As always, check the SSML specification for additional information.
Nuance SSML extensions

The Vocalizer SSML extensions are:

- The `<audio>` element supports four extra attributes to control internet fetching as described for the W3C VoiceXML 2.0 specification's version of the `<audio>` element. See http://www.w3.org/TR/voicexml20 for details. (Vocalizer does not support the VoiceXML 2.0 `<audio>` expr attribute, however.)
  - `fetchtimeout`: time to attempt to open and read the audio document. The value must be an unsigned integer with a mandatory suffix as required by the VoiceXML 2.0 specification, “s” for seconds, “ms” for milliseconds.
  - `maxage`: value for the HTTP 1.1 cache-control max-age directive. This specifies the application is willing to accept a cached copy of the audio document no older than this value. A value of 0 may be used to force re-validating the cached copy with the origin server. In most cases, this attribute should not be present, thus allowing the origin server to control cache expiration. The value must be an unsigned integer to specify the number of seconds; as required by the VoiceXML 2.0 specification, it must not have a suffix. (That is, “s" and “ms" are not allowed.)
  - `maxstale`: value for the HTTP 1.1 cache-Control max-stale directive. This specifies the client is willing to accept a cached copy that is expired by up to this value past the expiration time specified by the origin server. In most cases, this property should be set to 0 or not present, thus respecting the cache expiration time specified by the origin server. The value must be an unsigned integer to specify the number of seconds, as required by the VoiceXML 2.0 specification it must not have a suffix. (That is, “s" and “ms" are not allowed.)
  - `fetchhint`: “prefetch” to allow prefetching the audio content, “safe” (the default) to follow HTTP/1.1 caching semantics. Vocalizer allows this attribute, but currently does not behave differently for “prefetch” mode.

- The `<phoneme>` element supports specifying L&H+ phoneme strings when the alphabet attribute is set to “x-l&h+”, and phoneme strings in the IPA alphabet when the alphabet attribute is set to “ipa”. Note that the ampersand is a reserved XML character, so in an SSML document the L&H+ alphabet needs to be specified with alphabet="x\\&amp;l\&h+". Phoneme strings in the IPA alphabet should also use the necessary escape characters, as they cannot be expressed otherwise. See the appropriate Language Supplement for a list of escape codes.

- An optional ssft-domaintype attribute on `<speak>`, `<s>`, and `<p>` can be used for activating an ActivePrompt domain, equivalent to the `<ESC>\domain\` native control sequence. The attribute value is the ActivePrompt domain name.

- A new `<prompt>` element that supports specifying ActivePrompt IDs, equivalent to the `<ESC>\prompt\` native control sequence. The “id” attribute is required, and specifies the ActivePrompt in `<domain>:<prompt>` format. The content of the element specifies fallback text that is only spoken if the ActivePrompt cannot be found (similar to SSML `<audio>`).
Chapter 9

Integrating Vocalizer into VoiceXML platforms

Nuance Vocalizer for Network is designed to support all the prompting requirements of VoiceXML platforms, so VoiceXML platforms can simply delegate all prompt playback (including text-to-speech and audio recordings) to Vocalizer. VoiceXML platforms do so by extracting all the VoiceXML fragments relating to prompts into a SSML document, then submitting the SSML to Vocalizer for playback, obtaining a single audio stream from Vocalizer that unifies all the audio from the text-to-speech and audio recordings. This yields a more robust and efficient platform that is easier to develop and maintain.

While this is fairly straightforward, there are some subtle issues to consider in order to ensure that VoiceXML application developers have full access to all Vocalizer capabilities, particularly around which VoiceXML elements and attributes to pass-through to the SSML document, and which elements to omit. This chapter clarifies this integration effort.

VoiceXML elements to pass through to SSML

The first step in constructing a SSML document is to identify the VoiceXML elements that should be extracted into an SSML document for handling by Vocalizer, as summarized in this table. Except for the <audio> element which is described in a separate section below, all of these should be passed through as-is to the SSML document. While there are a small number of SSML 1.0 features that Vocalizer doesn’t support (rarely used features like pitch contours), it is safe to pass through all valid SSML 1.0 content as-is because Vocalizer gracefully uses fallback for those features it does not support.

<table>
<thead>
<tr>
<th>VoiceXML element</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio</td>
<td>Specifies audio files to be played</td>
<td>See the audio element section below for important implementation details. The VoiceXML “expr” attribute must be evaluated by the VoiceXML interpreter, converting it to the static “src” attribute.</td>
</tr>
<tr>
<td>break</td>
<td>Specifies a pause in the speech output</td>
<td></td>
</tr>
<tr>
<td>desc</td>
<td>Provides a description of a non-speech audio source in &lt;audio&gt;</td>
<td>Only for use in visual interfaces, so while this can be passed through, it is currently ignored by Vocalizer.</td>
</tr>
<tr>
<td>emphasis</td>
<td>Specifies text to speak with emphasis</td>
<td></td>
</tr>
<tr>
<td>VoiceXML element</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lexicon</td>
<td>Specifies a pronunciation lexicon</td>
<td>Very important to pass through so applications can tune text-to-speech via Vocalizer user dictionaries, user rule sets, and ActivePrompt databases (three different types of lexicons).</td>
</tr>
<tr>
<td>mark</td>
<td>Bookmark, used by VoiceXML 2.1 to indicate the barge-in location.</td>
<td>The VoiceXML 2.1 &quot;nameexpr&quot; attribute must be evaluated by the VoiceXML interpreter, converting it to the static &quot;name&quot; attribute.</td>
</tr>
<tr>
<td>meta</td>
<td>Specifies meta and “http-equiv” properties</td>
<td>Important to pass through so applications can block logging of sensitive information like credit card numbers by using name=&quot;secure_context&quot;, or to specify Internet fetch controls for &lt;audio&gt; and &lt;lexicon&gt; elements (currently ignored by Vocalizer but will be implemented in a future release). It is best to pass through all &lt;meta&gt; elements as-is because they have very little overhead, Vocalizer gracefully ignores irrelevant meta content, and future Vocalizer releases are likely to add more &lt;meta&gt; properties.</td>
</tr>
<tr>
<td>metadata</td>
<td>Specifies XML metadata content</td>
<td>Only for application developer reference purposes, so while this can be passed through, it is currently ignored by Vocalizer.</td>
</tr>
<tr>
<td>p</td>
<td>Identifies a paragraph</td>
<td></td>
</tr>
<tr>
<td>phoneme</td>
<td>Specifies a phonetic pronunciation</td>
<td>Vocalizer supports IPA and the L&amp;H+ phoneme alphabets. It is safe to do blind pass-through, Vocalizer uses the fallback for unsupported alphabets.</td>
</tr>
<tr>
<td>prosody</td>
<td>Specifies prosodic information</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Identifies a sentence</td>
<td></td>
</tr>
<tr>
<td>say-as</td>
<td>Specifies the type of text</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>Specifies replacement spoken text for the contained text</td>
<td></td>
</tr>
<tr>
<td>voice</td>
<td>Specifies voice characteristics</td>
<td></td>
</tr>
</tbody>
</table>

**Handling audio elements**

Some VoiceXML platforms handle `<audio>` element playback themselves, but it is better to delegate all prompts (including all `<audio>` elements) to Vocalizer:

- Vocalizer makes the VoiceXML platform easier to design and implement, handling all the complex logic for `<audio>` fetches and fallback on failure, sequencing `<audio>` fetches with text-to-speech fragments, handling audio file headers and sample conversions, and delivering all of it to the VoiceXML platform as a single real-time audio stream.
Vocalizer does sophisticated HTTP/1.1 compliant fetching and caching of http://, https://, and file:// URLs, minimizing latency for the caller and the load on the web servers that host <audio> content. In tests of deployed applications, this fetching and caching has proven to be more efficient than the <audio> fetching of several major VoiceXML platforms.

Vocalizer supplies numerous controls for tuning this fetching and caching, and the Vocalizer event logs provide detailed fetching and caching information to make it easy to measure and tune the system and troubleshoot problems. See Internet fetch support on page 58 for details on HTTP/1.1 fetching and caching support, and Application event logs on page 65 for logging details.

Vocalizer supports the VoiceXML fetchtimeout, fetchhint, maxage, and maxstale attributes, despite those not being part of the SSML 1.0 specification. The only VoiceXML attribute that needs to be handled externally is using the VoiceXML interpreter context to convert the “expr” attribute to the static “src” attribute.

Vocalizer handles all the audio formats required by the VoiceXML specification: headerless or WAV format audio files that contain 8 kHz μ-law or A-law samples. Vocalizer also supports headerless or WAV format audio files that contain linear 16-bit PCM samples. (However Vocalizer does not currently support the Sun audio “au” header that is mentioned by the VoiceXML specification, but is not required or even recommended.)

Vocalizer does intelligent text-to-speech processing that considers the full context, including <audio> insertions. Providing the full context including audio insertions allows for optimal audio quality (including blending the audio and text-to-speech segments) and makes it easier for VoiceXML application developers to tune Vocalizer’s speech output.

Handling Nuance extensions to SSML

Nuance SSML extensions on page 168 documents extensions to SSML that are handled by Vocalizer. VoiceXML platforms should allow for these extensions within VoiceXML documents, passing them through to the SSML document that will be spoken by Vocalizer. In summary, those that should be implemented are:

- The four extra <audio> element attributes that are specified by VoiceXML 2.0, as described in the previous section.
- The “x-l&h+” alphabet for <phoneme>.
- An optional ssft-domaintype attribute for <speak>, <s>, and <p>.

Note: Vocalizer also supports a new SSML <prompt> element for explicit ActivePrompt insertions. However this conflicts with VoiceXML <prompt>, and may safely be left out of VoiceXML integrations (ActivePrompt insertions remain usable via proprietary markup or via automatic matching).

Handling old VoiceXML specifications

VoiceXML platforms may implement any VoiceXML specification, but Vocalizer only supports the SSML 1.0 Recommendation (September 2004) as used by the VoiceXML 2.0 Recommendation (March 2004) and VoiceXML 2.1 Recommendation (June 2007). By default Vocalizer returns a SSML parse error for any elements or attributes that don’t comply with the SSML 1.0 Recommendation. Platforms that wish to implement older
VoiceXML specifications such as VoiceXML 2.0 working drafts or VoiceXML 1.0 may need to convert some older elements and attributes to the SSML 1.0 Recommendation syntax. This is typically straightforward since during the evolution of SSML 1.0 there were many feature additions and syntax changes, but few major semantic changes.

**Generating an SSML document**

The final step in delegating prompts to Vocalizer is to generate the SSML document. This requires choosing a character encoding for the document, constructing a SSML document header, and passing through the relevant VoiceXML elements as described in the previous sections. It is important to make these SSML documents as big as possible, rather than generating lots of small SSML documents for each individual VoiceXML element: this significantly reduces processing overhead, supplies contextual information that is important for optimal audio quality, and makes it much easier for VoiceXML application developers to tune the speech output.

For choosing the character encoding, the best choice is a Unicode encoding, such as UTF-8 or UTF-16. Most VoiceXML platforms use a XML parser library, and most XML parsers return the text as UTF-8 or UTF-16 already. This makes it so the VoiceXML extraction can be language independent, handling any world language without VoiceXML platform code changes. Vocalizer does all of its internal processing using UTF-16, so that is the most CPU efficient choice. However UTF-8 is also a good choice, particularly for MRCP based integrations (such as when using Nuance Speech Server) where the headers are all ASCII (a subset of UTF-8) and thus the use of UTF-8 makes the MRCP messages more readable, simplifying monitoring and troubleshooting.

For the SSML document header, it is important to comply with the SSML 1.0 specification: a “speak” root element with the following attributes:

<table>
<thead>
<tr>
<th><code>&lt;speak&gt;</code> attribute</th>
<th>Required/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>required</td>
<td>Must be “1.0”.</td>
</tr>
<tr>
<td>xml:lang</td>
<td>required</td>
<td>Language for the document. Pass through the xml:lang property for the current VoiceXML interpreter scope as inherited from the closest enclosing VoiceXML <code>&lt;prompt&gt;</code>, or from the <code>&lt;vxml&gt;</code> root element if there is no <code>&lt;prompt&gt;</code> or no xml:lang on the <code>&lt;prompt&gt;</code> elements.</td>
</tr>
<tr>
<td>xmlns</td>
<td>required</td>
<td>Must specify the XML namespace for SSML 1.0, “<a href="http://www.w3.org/2001/10/synthesis%E2%80%9D">http://www.w3.org/2001/10/synthesis”</a>, otherwise Vocalizer’s XML parser does not recognize it as a valid SSML 1.0 document and Vocalizer returns an SSML parse error.</td>
</tr>
<tr>
<td>xml:base</td>
<td>optional but highly recommended</td>
<td>Specify an absolute URI for the VoiceXML document that is being extracted into the SSML document. This allows Vocalizer to properly resolve relative URIs, and is also very helpful for diagnostic purposes.</td>
</tr>
</tbody>
</table>
The SSML 1.0 specification shows the following on the first `<speak>` element example:
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance". This just makes it so "xsi" can be used to refer to the XML namespace for XML 1.0 Schema later on, used for xsi:schemaLocation.

<table>
<thead>
<tr>
<th><code>&lt;speak&gt;</code> attribute</th>
<th>Required/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlns:xsi</td>
<td>optional</td>
<td>The SSML 1.0 specification shows the following on the first <code>&lt;speak&gt;</code> element example: xmlns:xsi=&quot;<a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a>&quot;. This just makes it so &quot;xsi&quot; can be used to refer to the XML namespace for XML 1.0 Schema later on, used for xsi:schemaLocation.</td>
</tr>
<tr>
<td>xsi:schemaLocation</td>
<td>optional</td>
<td>Specifies where the SSML 1.0 schema document can be found, ignored by Vocalizer since it already has the SSML 1.0 schema document compiled in.</td>
</tr>
</tbody>
</table>
Appendix A

Performance and sizing

This chapter describes the methodology used to characterize the performance and scalability of Vocalizer. It contains an overview of the method chosen, a description of the tool that Nuance uses to run tests, and details about specific metrics in the test.

Actual test results for any given Vocalizer release, platform, and voice are provided in the Release Notes for the voice.

Ultimately, the model Nuance chose to measure Vocalizer performance and the criteria for interpreting the test results is arbitrary when compared to each individual user’s deployment context. Nuance has tried to pick the most representative approach, but the reader should be aware that their own situation might have different requirements.

This appendix covers:

- Testing methodology on page 176
- Test scenario and application on page 176
- Performance statistics on page 176
- Resource consumption on page 177
- Determining the supported number of channels on page 178
Testing methodology

Because of the wide variety of situations where TTS is used, it is impractical to model and
test all of those scenarios. There is a spectrum of possible ways. At one end, a model could
simulate an application that requires TTS output a small percentage of the time—say 15%.
Results from this model are only useful when sizing a similar application. At the other
end, another model could have all engine instances synthesizing all of the time, even if
they are faster than real-time. Results from this method are also dubious because that
level of aggressive use is atypical.

Nuance chooses to test Vocalizer with a method that lies in the middle. The model
simulates an application that has TTS output playing on all output channels 100% of the
time. This is different from having all engines synthesizing 100% of the time. For example,
in the Nuance model, if the testing application requests 10 seconds of audio from
Vocalizer and receives all of the audio data within one second, it waits another 9 seconds
before sending the next request. This model is probably also atypical—few to no
applications require that much TTS—but the results from it offer a more accurate way to
size a deployment than the other two ends of the spectrum.

These tests assume no specific tuning of any operating system to improve application
performance. For all measurements, Vocalizer runs in-process within the test application,
and uses the native Vocalizer API.

Test scenario and application

The Vocalizer SDK contains a set of sample applications. One of these samples, named
nvnload, is the same testing application Nuance uses to test Vocalizer performance.
Because there are so many different ways of building an application and configuring
Vocalizer, access to this tool allows users to test performance on their specific
configuration. For usage details including command line options, see the file
install_path/api/demos/nvnload/README.txt.

The application submits text to Vocalizer over a variable and configurable number of
engine instances. Command-line options specify the initial and final number of engines
as well as the increment. For example, a test run may begin with 40 engines and progress
to 70 in increments of five. At each step, a configurable number of speak requests, for
example 100, is submitted per engine. The test application’s output is a table of result data
with each row showing the results from each incremental run. The row shows various
statistics describing the responsiveness and speed of Vocalizer. A few of these are
described in the sections below. All of the measurements, except CPU utilization and
memory usage, are made on the application side of the Vocalizer API.

Vocalizer’s performance depends on the text being synthesized, including its content,
complexity, individual utterance length, presence of unusual strings, abbreviations, and
so on. Sample texts are taken from a selection of newspaper articles with each heading
and paragraph forming a separate speak request that is sent to Vocalizer in a random
order, observing the timing model described above. The Vocalizer sample supplies the
input texts that Nuance uses for testing. These input texts are included in files named
loadTest_American_English.txt, loadTest_French.txt, and so on, for the different languages
stored in the test_data subdirectory within the Vocalizer installation.
Performance statistics

Nuance uses several metrics to evaluate test results. You should understand their meaning to estimate the tasks and loads that Vocalizer can handle in a given configuration. The output table from the testing application has several columns of results from each test. This description focuses on the most important two:

- **Latency (time-to-first-audio)** on page 177
- **Audio buffer underflow** on page 177

**Latency (time-to-first-audio)**

Applications with performance concerns start with the most obvious criterion: once the application asks Vocalizer to synthesize text, how long will it be until the application receives the first audio data? This time span is called latency, defined as the time between the application’s call to TtsProcessEx and the arrival of the first corresponding audio packet in the callback function. Although Vocalizer uses streaming audio to minimize latency, the TTS engine must process whole phrases at a time to obtain optimal rhythm and intonation. This causes a processing delay but the size of this delay is highly dependent on the characteristics of the requested text, specifically the length and, to a lesser degree, the complexity of the first phrase. For example, an input that begins with a one-word phrase such as “Hello” should have shorter latency than an input that begins with a 30 word sentence.

**Audio buffer underflow**

Once the first audio data has been delivered and audio output (to a caller or PC user) can begin, the application is concerned with whether subsequent audio data will arrive fast enough to avoid having “gaps” in the audio. When these gaps occur, this is called an underflow. More tightly defined, **audio buffer underflow** refers to the audio dropout that occurs when the application consumes (plays out) all of the audio it received from Vocalizer and is idle while waiting for the next audio data packet.

The audio buffer underflow rate is the percentage of underflows that occur over time. For example, assume that each audio packet passed to the callback function contains 512 ms of audio data. An underflow rate of 1% therefore translates to a potential gap every 100 audio buffers, or 51.2 seconds of audio. A rate of 0.01% equals a gap on average once every 90 minutes.

**Resource consumption**

The testing application does not test two important metrics needed for proper sizing estimation: CPU use and memory use. These have to be monitored using external tools such as Windows Performance Monitor or Unix ps.

**CPU utilization**

CPU utilization is the percentage of the total CPU time spent processing user or system calls on behalf of Vocalizer. As more TTS ports are opened and used for processing, CPU utilization increases approximately linearly. When CPU utilization exceeds 85%, performance degrades rapidly as further ports are opened.
Memory use

Vocalizer’s base memory requirement varies per voice, and is usually around 10–20 MB. See the Release Notes for each voice for detailed sizing data that describes the base memory usage for each voice. In addition, each engine instance (as created by calling TtsOpen) requires incremental amounts of memory, usually about 500 KB. Therefore, the formula to calculate total memory requirements is: base + (# engines * 500 KB). For 50 instances of a voice that has a base of 15 MB, that results in a memory use of 40 MB.

Determining the supported number of channels

Nuance reduces all of these metrics down to one number, claiming that Vocalizer supports “X” channels on a given configuration. This is the maximum number of simultaneous connections that can be made by our test application while keeping the following performance limits:

- Average latency <= 250ms
- Audio buffer underflow rate <= 0.04%
- CPU utilization <= 85%

As mentioned above, these thresholds may not suit your requirements and you may wish to rerun tests with different criteria.
The following table shows a list of all the Vocalizer language names and codes, sorted according to the language name. It also specifies the IETF language code (which is used in SSML and can optionally be used for specifying voices) and the 3-letter Vocalizer language code (which is used to specify the language in the header of user dictionaries and user rulesets).

**Note:** This table lists the currently reserved Vocalizer language names and codes, it does not imply these languages are actually available for Vocalizer. Go to the technical support portal at network.nuance.com for a list of the currently available Vocalizer languages and voices, and contact Nuance sales for information about any language that you require but is not available at network.nuance.com.

<table>
<thead>
<tr>
<th>Language name</th>
<th>ISO language code</th>
<th>Vocalizer language code</th>
</tr>
</thead>
<tbody>
<tr>
<td>American English</td>
<td>en-US</td>
<td>ENU</td>
</tr>
<tr>
<td>Australian English</td>
<td>en-AU</td>
<td>ENA</td>
</tr>
<tr>
<td>Basque</td>
<td>eu-ES</td>
<td>BAE</td>
</tr>
<tr>
<td>Belgian Dutch</td>
<td>nl-BE</td>
<td>DUB</td>
</tr>
<tr>
<td>Belgian French</td>
<td>fr-BE</td>
<td>FRB</td>
</tr>
<tr>
<td>Brazilian Portuguese</td>
<td>pt-BR</td>
<td>PTB</td>
</tr>
<tr>
<td>British English</td>
<td>en-GB</td>
<td>ENG</td>
</tr>
<tr>
<td>Canadian French</td>
<td>fr-CA</td>
<td>FRC</td>
</tr>
<tr>
<td>Danish</td>
<td>da-DK</td>
<td>DAD</td>
</tr>
<tr>
<td>Dutch</td>
<td>nl-NL</td>
<td>DUN</td>
</tr>
<tr>
<td>French</td>
<td>fr-FR</td>
<td>FRF</td>
</tr>
<tr>
<td>German</td>
<td>de-DE</td>
<td>GED</td>
</tr>
<tr>
<td>Hong Kong Cantonese</td>
<td>zh-HK</td>
<td>CAH</td>
</tr>
<tr>
<td>Indian English</td>
<td>en-IN</td>
<td>ENI</td>
</tr>
<tr>
<td>Italian</td>
<td>it-IT</td>
<td>ITI</td>
</tr>
<tr>
<td>Japanese</td>
<td>ja-JP</td>
<td>JPJ</td>
</tr>
<tr>
<td>Korean</td>
<td>ko-KR</td>
<td>KOK</td>
</tr>
<tr>
<td>Mandarin Chinese</td>
<td>zh-CN</td>
<td>MNC</td>
</tr>
<tr>
<td>Language name</td>
<td>ISO language code</td>
<td>Vocalizer language code</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Mexican Spanish</td>
<td>es-MX (alias: es-US)</td>
<td>SPM</td>
</tr>
<tr>
<td>Norwegian</td>
<td>no-NO</td>
<td>NON</td>
</tr>
<tr>
<td>Polish</td>
<td>pl-PL</td>
<td>PLP</td>
</tr>
<tr>
<td>Portuguese</td>
<td>pt-PT</td>
<td>PTP</td>
</tr>
<tr>
<td>Russian</td>
<td>ru-RU</td>
<td>RUR</td>
</tr>
<tr>
<td>Spanish</td>
<td>es-ES</td>
<td>SPE</td>
</tr>
<tr>
<td>Swedish</td>
<td>sv-SE</td>
<td>SWS</td>
</tr>
<tr>
<td>Swiss French</td>
<td>fr-CH</td>
<td>FNC</td>
</tr>
<tr>
<td>Swiss German</td>
<td>de-CH</td>
<td>GEC</td>
</tr>
<tr>
<td>Swiss Italian</td>
<td>it-CH</td>
<td>ITC</td>
</tr>
</tbody>
</table>
Appendix C

Copyright and licensing for third party software

The Nuance Vocalizer for Network SDK utilizes certain open source software packages. Copyright and licensing information for these packages are included in this section.

Copyright and Licensing Information for ACE(TM), TAO(TM), CIAO(TM), and CoSMIC(TM)

[1]ACE™, [2]TAO™, [3]CIAO™, and [4]CoSMIC™ (henceforth referred to as "DOC software") are copyrighted by [5]Douglas C. Schmidt and his [6]research group at [7]Washington University, [8]University of California, Irvine, and [9]Vanderbilt University, Copyright (c) 1993-2009, all rights reserved. Since DOC software is open-source, freely available software, you are free to use, modify, copy, and distribute—perpetually and irrevocably—the DOC software source code and object code produced from the source, as well as copy and distribute modified versions of this software. You must, however, include this copyright statement along with any code built using DOC software that you release. No copyright statement needs to be provided if you just ship binary executables of your software products.

You can use DOC software in commercial and/or binary software releases and are under no obligation to redistribute any of your source code that is built using DOC software. Note, however, that you may not do anything to the DOC software code, such as copyrighting it yourself or claiming authorship of the DOC software code, that will prevent DOC software from being distributed freely using an open-source development model. You needn’t inform anyone that you’re using DOC software in your software, though we encourage you to let [10] us know so we can promote your project in the [11] DOC software success stories.

The ACE, TAO, CIAO, and CoSMIC web sites are maintained by the [12] DOC Group at the [13] Institute for Software Integrated Systems (ISIS) and the [14] Center for Distributed Object Computing of Washington University, St. Louis for the development of open-source software as part of the open-source software community. Submissions are provided by the submitter ‘’as is’’ with no warranties whatsoever, including any warranty of merchantability, noninfringement of third party intellectual property, or fitness for any particular purpose. In no event shall the submitter be liable for any direct, indirect, special, exemplary, punitive, or consequential damages, including without limitation, lost profits, even if advised of the possibility of such damages. Likewise, DOC software is provided as is with no warranties of any kind, including the warranties of design, merchantability, and fitness for a particular purpose, noninfringement, or arising from a course of dealing, usage or trade practice. Washington University, UC Irvine, Vanderbilt University, their employees, and students shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by DOC software or any part thereof. Moreover, in no event will Washington University, UC Irvine, or Vanderbilt...
University, their employees, or students be liable for any lost revenue or profits or other special, indirect and consequential damages.

DOC software is provided with no support and without any obligation on the part of Washington University, UC Irvine, Vanderbilt University, their employees, or students to assist in its use, correction, modification, or enhancement. A number of companies around the world provide commercial support for DOC software, however.

DOC software is Y2K-compliant, as long as the underlying OS platform is Y2K-compliant. Likewise, DOC software is compliant with the new US daylight savings rule passed by Congress as “The Energy Policy Act of 2005,” which established new daylight savings times (DST) rules for the United States that expand DST as of March 2007. Since DOC software obtains time/date and calendaring information from operating systems users will not be affected by the new DST rules as long as they upgrade their operating systems accordingly.

The names ACE™, TAO™, CIAO™, CoSMIC™, Washington University, UC Irvine, and Vanderbilt University, may not be used to endorse or promote products or services derived from this source without express written permission from Washington University, UC Irvine, or Vanderbilt University. This license grants no permission to call products or services derived from this source ACE™, TAO™, CIAO™, or CoSMIC™, nor does it grant permission for the name Washington University, UC Irvine, or Vanderbilt University to appear in their names.

If you have any suggestions, additions, comments, or questions, please let me know.

Douglas C. Schmidt

References
1 http://www.cs.wustl.edu/~schmidt/ACE.html
2 http://www.cs.wustl.edu/~schmidt/TAO.html
3 http://www.cs.wustl.edu/~schmidt/CIAO.html
4 http://www.dre.vanderbilt.edu/cosmic/
5 http://www.dre.vanderbilt.edu/~schmidt/
6 http://www.cs.wustl.edu/~schmidt/ACE-members.html
7 http://www.wustl.edu/
8 http://www.uci.edu/
9 http://www.vanderbilt.edu/
10 doc_group@cs.wustl.edu
11 http://www.cs.wustl.edu/~schmidt/ACE-users.html
12 http://www.dre.vanderbilt.edu/
13 http://www.isis.vanderbilt.edu/
15 http://www.cs.wustl.edu/~schmidt/commercial-support.html
16 d.schmidt@vanderbilt.edu
Apache Group Xerces-C XML Parser

The Apache Software License, Version 1.1 Copyright © 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. 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.

3. The end user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Xerces" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED 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 APACHE SOFTWARE FOUNDATION OR ITS 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.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation and was originally based on software copyright © 1999, International Business Machines, Inc., http://www.ibm.com. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Dinkumware C++ Library for Visual C++

Developed by P.J. Plauger

Copyright © 1992–2000 by P.J. Plauger

Dinkumware, Ltd.
RSA Data Security, Inc. MD5 Message-Digest Algorithm


IBM International Components for Unicode (ICU)

http://oss.software.ibm.com/icu/

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1995–2003 International Business Machines Corporation and others

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

Perl Compatible Regular Expressions (PCRE)

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.
Release 5 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

Written by: Philip Hazel <ph10@cam.ac.uk>


Copyright © 1997–2004 University of Cambridge. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the University of Cambridge nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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.