
User's Guide



S P E E C H I F Y 3 . 0

Document History

Date	Release Name
November 2003	Sixth Edition, update for Speechify 3.0
August 2003	Sixth Edition, Speechify 3.0
February 2003	Fifth Edition, update 3 for Speechify 2.1.6
January 2003	Fifth Edition, update 2 for Speechify 2.1.5
September 2002	Fifth Edition, update 1 for Speechify 2.1.3
May 2002	Fifth Edition, Speechify 2.1
January 2002	Fourth Edition, update for Speechify 2.0
December 2001	Fourth Edition, Speechify 2.0
May 2001	Third Edition, Speechify 1.2
March 2001	Second Edition, Speechify 1.1
November 2000	First Edition, Speechify 1.0

Notice

Copyright © 2000–2003 by ScanSoft, Inc. All rights reserved.

The information in this document is subject to change without notice.

Use of this document is subject to certain restrictions and limitations set forth in a license agreement entered into between SpeechWorks International, Inc. and purchaser of the Speechify software. Please refer to the SpeechWorks license agreement for license use rights and restrictions.

SpeechWorks is a registered trademark, and OpenSpeech, DialogModules, SMARTRecognizer, Speechify, Speechify Solo, SpeechSecure, SpeechSite, SpeechWorks Here, the SpeechWorks logo and the SpeechWorks Here logo are trademarks or registered trademarks of SpeechWorks International, Inc. in the United States and other countries. All other trademarks are property of their respective owners. Windows NT is a registered trademark of Microsoft Corporation.

Portions of Speechify software are subject to copyrights of GlobeTrotter Software, Inc.

Published by:

ScanSoft, Inc.
Worldwide Headquarters
9 Centennial Drive
Peabody, MA 01960
United States

Table of Contents

Preface	xi
Support services	xiii

Operations I

I. Installing and Configuring Speechify	3
Hardware and software requirements.....	4
Installation on Windows	6
Installing the Speechify client and server.....	6
Installing a voice	8
Uninstalling the Speechify client and server	9
Uninstalling a voice.....	9
Installation on Red Hat Linux.....	10
Installing the Speechify server	10
Installing the Speechify client.....	10
Installing a voice	11
Installed file locations	12
Uninstalling the Speechify server.....	12
Uninstalling the Speechify client.....	13
Uninstalling a voice.....	13
Installation on Solaris	14
Installing the Speechify client.....	14
Installed file locations	15
Uninstalling the Speechify client.....	15
Running the Speechify server from the console.....	16
Starting Speechify	16
Stopping Speechify	16
Running Speechify as a Windows service.....	18
Using the MMC snap-in.....	19
Internet fetching	22
Configuring the web server to return a proper MIME content type	23
Configuring a web server to optimize performance	23
Using configuration files	24

Default configuration files	24
User-defined configuration files	25
Configuring the Speechify SAPI 5 interface	25
2. Speechify Logging	27
Overview of logs	27
Error log	28
Diagnostic log.....	28
Server event log.....	28
Client logging	29
Overview of client logs.....	29
Enabling diagnostic logging	30
Logging to a file or stdout/stderr.....	30
Server logging	32
Default logging behavior.....	32
Controlling logging	32
SAPI 5 logging	33
Error and diagnostic message format	33
Event message format	35
Example event log.....	36
Description of events and tokens	36
3. Performance and Sizing	39
Testing methodology	40
Test scenario and application	40
Performance statistics	41
Latency (time-to-first-audio).....	41
Audio buffer underflow.....	42
Resource consumption	42
Server CPU utilization.....	42
Memory use.....	43
Determining the supported number of channels	43

Programming Guide **45**

4. Programming Guide	47
Features	48
Order of API functions.....	49
Callback function	50
Sample time lines	52
Overview of user dictionaries.....	56
Parameter configuration	56
Character set support	57
Using W3C SSML	57
Using SAPI	58
Implementation guidelines	58
5. Using User Dictionaries	61
Overview of dictionaries	61
Loading and activating dictionaries	62
Loading and activating dictionaries from a configuration file	62
Loading, activating, and deactivating dictionaries using API calls.....	63
Loading and activating dictionaries using embedded tags.....	64
Invalid dictionary entries	64
Dictionary file format	65

Text-to-Speech Processing **69**

6. Embedded Tags	71
Using Speechify tags.....	72
Creating pauses	72
Indicating the end of a sentence	73
Customizing pronunciations	74
Customizing word pronunciations.....	74
Specifying user dictionaries.....	75
Character spellout modes.....	76
Pronouncing numbers and years	77

Language-specific customizations.....	78
Inserting bookmarks	79
Controlling the audio characteristics	80
Volume control.....	81
Speaking rate control	82
7. Standard Text Normalization	83
Numbers	84
Numeric expressions	85
Mixed alphanumeric tokens	85
Abbreviations	86
Ambiguous abbreviations.....	86
Periods after abbreviations.....	87
Measurement abbreviations.....	87
Uppercase acronyms and tokens	88
E-mail addresses	88
URLs	89
File names and paths	90
Punctuation	91
Parentheses	92
Hyphen	92
Slash	93
8. Symbolic Phonetic Representations	95
SPR format and error handling	96
Syllable boundaries	97
Syllable-level information	97
Speech sound symbols	97
9. W3C SSML Compliance	99
Speechify's W3C SSML parsing	99
Element support status	100
Support of the "audio" element.....	102
Support of the "say-as" element.....	106

10. User Dictionaries 109

Overview of dictionaries	110
Main dictionary	111
Abbreviation dictionary	111
Root dictionary	112
Mainext dictionary	113
Choosing a dictionary	114
Main and abbreviation dictionaries vs. root dictionary	114
Main dictionary vs. abbreviation dictionary	115
Dictionary interactions	116

11. Improving Speech Quality 117

Introduction	118
Customizing text analysis	119
Customizing the default text analysis	120
Text manipulation	122
Case study – homograph disambiguation	126
Case study – navigation text	127
Speech concatenation issues	129
Logging generated pronunciations	130

Reference Material 131

12. API Reference 133

Calling convention	134
Server's preferred character set	134
Result codes.....	135
SWIttsCallback()	137
SWIttsClosePort()	143
SWIttsDictionaryActivate()	144
SWIttsDictionariesDeactivate()	146
SWIttsDictionaryFree()	147
SWIttsDictionaryLoad()	148

SWIttsGetParameter()	151
SWIttsInit()	154
SWIttsOpenPortEx()	155
SWIttsResourceAllocate()	157
SWIttsResourceFree()	158
SWIttsSetParameter()	159
SWIttsSpeak()	161
SWIttsSpeakEx()	164
SWIttsStop()	167
SWIttsTerm()	168

13. Configuration Files 169

Configuration file format	170
The <lang> element	171
The <param> element	172
The <value> element	172
The <namedValue> element	173
Modifying a configuration file	174
Configuration parameters	175
Audio parameters	175
Cache parameters	176
Engine parameters	177
Environment variable parameters	178
Internet fetch parameters	179
Log parameters	180
Marks parameters	182
Network parameters	182
Preprocessor parameters	183
Server parameters	183
SSML parameters	185
Text format parameters	185
Voice parameters	186

Appendices and Index 187

Appendix A: SAPI 5 189

Compliance	189
------------------	-----

SAPI voice properties	191
Appendix B: Frequently Asked Questions	193
Question types	194
Changing rate or volume	194
Rewinding and fast-forwarding	195
Java interface	195
E-mail preprocessor and SAPI 5	195
W3C SSML and SAPI 5	196
Error codes 107 and 108	196
Connecting to the server	197
Port types	197
OpenPort performance	198
Index.....	199



Preface

Welcome to ScanSoft

Speechify™ is ScanSoft, Inc.'s state of the art Text-To-Speech (TTS) system. This guide is written for application developers who want to add Speechify's high-quality text-to-speech functionality to their applications and for the system operators who operate those systems.

New and changed information

This update to the Sixth Edition corrects the supported operating systems for this product. There are changebars (bold vertical lines as shown in the left margin of this paragraph) to indicate the locations of those changes.

The previous "Sixth Edition" was significantly reorganized to address the needs of the various readers (operations staff, application developers, integrators, etc.). See "How this guide is organized" on [page xii](#) for an overview. See the *SpeechWorks Release Notes* and the *Speechify Migration Guide* for a list of changes in Speechify 3.0.

How this guide is organized

This document includes the following parts:

“[Operations](#)” describes the installation requirements, the installation packages, and how to run the Speechify server. It also describes how ScanSoft arrives at performance numbers, and describes how to enable and use the error log, diagnostic log, and event log.

“[Programming Guide](#)” contains an overview of features, and high-level information on groups of API functions, including how to use Microsoft's SAPI and W3C Speech Synthesis Markup Language (W3C SSML) with Speechify. It also covers the user dictionaries from a programming perspective.

“[Text-to-Speech Processing](#)” describes several different ways to get the desired speech from the input text: the phonetic spelling users can use to explicitly specify word pronunciations; the tags users can insert into input text to customize the speech output; and the user dictionaries available for customizing the pronunciations of words, abbreviations, acronyms, and other sequences. This part also describes how Speechify processes input text before synthesizing the output speech and provides an overview of factors that contribute to output speech quality along with troubleshooting tips for improving the sound.

“[Reference Material](#)” describes the API functions and result codes, and the configuration file format and parameters.

Finally, this document includes appendices to cover the SAPI 5 interface and frequently asked questions, as well as an index.

Additional information

ScanSoft Speech Training Center

For details about training courses available for Network Speech Solutions topics, see the webpage at <http://support.scansoft.com/developers/training/> or send email to speechtraining@scansoft.com.

Available documentation

- ❑ The *Speechify User's Guide* provides installation, programming, and reference information about the Speechify product.
- ❑ The *SpeechWorks Licensing Handbook* describes the process for getting licenses to run the Speechify software, details about configuring your license server, and related topics.
- ❑ There is a *Speechify Language Supplement* for each supported language. These supplements contain language-specific reference information for application developers.
- ❑ The *Speechify Migration Guide* gives instructions for Speechify 2.x users to migrate their systems and applications to Speechify 3.0.
- ❑ The *Speechify E-mail Pre-processor Developer's Guide* covers the types of input that the SpeechWorks E-mail Pre-processor handles, how it processes the messages, the modes that the application can take advantage of at run time, the layout and use of the e-mail substitution dictionary, and the API functions.
- ❑ Review the release notes distributed with this product for the latest information, restrictions, and known problems.

Support services

To receive technical support from ScanSoft, Inc.:

- ❑ Visit the Knowledge Base or ask a question at: <http://developer.scansoft.com>. This site requires a customer username and password.
- ❑ Visit <http://www.scansoft.com> and get a contact telephone number for the nearest ScanSoft office.
- ❑ For issues related to Network Automated Speech Recognition, call the Boston office in North America at +1 617 428-4444 and ask for "technical support."
- ❑ See [Appendix C](#) for more support details.

See [Chapter 2 "Speechify Logging"](#) for information on how to collect diagnostics that SpeechWorks may use to diagnose your problem.

Embedded third-party software

Apache software

This product contains software developed by the Apache Software Foundation (www.apache.org).

Xerces C++ 2.2.0. Copyright © 1999–2001 The Apache Software Foundation. All Rights Reserved.

The Flite Speech Synthesis System

Language Technologies Institute
Carnegie Mellon University
Copyright © 1999–2003
All Rights Reserved.
<http://cmuflite.org>

Dinkumware C++ Library for Visual C++

Developed by P.J. Plauger
Copyright © 1992–2000 by P.J. Plauger
Dinkumware, Ltd.
398 Main Street
Concord MA 01742

RSA Data Security, Inc. MD5 Message-Digest Algorithm

Copyright © 1991–1992, RSA Data Security, Inc. Created 1991. All Rights Reserved.



Operations

The chapters in this part cover these topics:

[Chapter 1 “Installing and Configuring Speechify”](#) describes the installation requirements and installation packages.

[Chapter 2 “Speechify Logging”](#) describes how to enable and use the error log, diagnostic log, and event log.

[Chapter 3 “Performance and Sizing”](#) summarizes how ScanSoft estimates performance and sizing numbers.



Installing and Configuring Speechify

This chapter describes the installation requirements and the installation packages for the Speechify client and server and Speechify voices on the following platforms:

- ❑ Windows
- ❑ Red Hat Linux
- ❑ Sun Solaris (client only)

In This Chapter

- ❑ “Hardware and software requirements” on [page 4](#)
- ❑ “Installation on Windows” on [page 6](#)
- ❑ “Installation on Red Hat Linux” on [page 10](#)
- ❑ “Installation on Solaris” on [page 14](#)
- ❑ “Running the Speechify server from the console” on [page 16](#)
- ❑ “Running Speechify as a Windows service” on [page 18](#)
- ❑ “Internet fetching” on [page 22](#)
- ❑ “Using configuration files” on [page 24](#)

Hardware and software requirements

Server run-time hardware and software requirements

Speechify requires the following minimum hardware and software to run:

1. 500 MHz Intel Pentium III-based computer.
2. Memory requirements vary from one Speechify voice to another because each uses a different amount to start. Contact ScanSoft Technical Support for specific voice requirements. In general, for a box that supports 72 ports, we recommend 512 MB of RAM.
3. The Speechify server run-time software requires 60 MB of free disk space. Each Speechify voice has disk space requirements beyond that which vary from voice to voice. Contact ScanSoft Technical Support for specific voice requirements.
4. Your system paging file size *must* be at least 300 MB.
5. CD-ROM drive to install the software.
6. Network Interface Card.
7. One of the following operating systems, configured for TCP/IP networking:
 - a. Windows 2000 Service Pack 3 or 4, Windows XP Service Pack 1, or Windows 2003 Server
 - b. Red Hat Linux 7.2 or 7.3
 - c. Solaris 8 for Sparc (client only)
8. Microsoft Windows users must have Internet Explorer 5.0 or later installed on their machine. To download Internet Explorer, go to:

<http://www.microsoft.com/windows/ie/>
9. Adobe Acrobat Reader 5.0 or later for reading the full Speechify documentation set.

Client SDK run-time hardware and software requirements

The following minimum hardware and software requirements for the application development tools within the Speechify SDK are:

1. 300 MHz Intel Pentium III-based computer.
2. 64 MB of RAM.
3. 10 MB of free disk space for the Speechify Software Development Kit (SDK).
4. CD-ROM drive to install the software.
5. Network Interface Card.
6. One of the following operating systems, configured for TCP/IP networking:
 - a. Windows 2000 Service Pack 3 or 4, Windows XP Service Pack 1, or Windows 2003 Server
 - b. Red Hat Linux 7.2 or 7.3
 - c. Solaris 8 for Sparc (client only)
7. Microsoft Windows users must have Internet Explorer 5.0 or later installed on their machine. To download Internet Explorer, go to:

<http://www.microsoft.com/windows/ie/>
8. A C/C++ compiler:
 - a. On Windows systems, ScanSoft tests with Microsoft Visual C++ 6 with Service Pack 3. Sample applications are provided with Visual C++ project files.
 - b. On Unix systems, ScanSoft tests with the GNU GCC compiler that ships with each Red Hat Linux release (GNU GCC 2.96 for Red Hat Linux 7.2 and 7.3).
9. Adobe Acrobat Reader 5.0 or later for reading the full Speechify documentation set.

Installation on Windows

To install Speechify on Windows, first install the Speechify client and server, then install each voice separately.



NOTE

If the system is already running an OSR licensing service, do not install the “Speechify licensing service” component described below, as they will conflict. Instead, add your Speechify licenses into your existing OSR license file and restart the OSR licensing service as explained in the *SpeechWorks Licensing Handbook*. Otherwise on reboot you get a Windows error saying one or more services could not be started and the Speechify Licensing Service does not start.

If by accident you install the licensing component and get these errors, you can disable the Speechify licensing service without uninstalling it. Open Control Panel >> Administrative Tools >> Services. Double-click the Speechify Licensing Service, change the Startup Type from Automatic to Disabled, and click OK.

Installing the Speechify client and server

Internet Explorer 5.0 or later must be installed before installing Speechify.

1. Log into Windows as the Administrator.
2. Open the Speechify WinZip package you downloaded from ScanSoft and extract the contents to a temporary directory.
3. Run setup.exe from the temporary directory
4. Follow the on-screen instructions, reading the Welcome screen and the Software License Agreement.

If you choose a “Typical” installation, the Speechify installer asks no further questions and installs the following components to C:\Program Files\Speechify:

- Speechify server run-time
- Speechify client run-time
- Speechify client SDK
- Speechify licensing service

This typical installation omits the “Speechify SAPI client run-time” that is required for running Speechify as a Microsoft SAPI component.

To override these defaults and choose a different install location or different components, choose “Custom.” A dialog box appears for customizing the install.

5. When prompted, you should read the *SpeechWorks Release Notes* to familiarize yourself with any special programming considerations and known bugs.
6. You may now delete the temporary directory where you extracted the Speechify installation package.
7. You must install a Speechify voice before you can run the server. See “Installing a voice” on [page 8](#).

In addition, the Speechify installation does the following:

- ❑ Installing the client SDK creates an environment variable named SWITTSSDK which points to the path where the SDK is installed.
- ❑ The Speechify installation uses Windows Installer technology and updates the copy on your machine if necessary before proceeding with the install.
- ❑ If you choose to install the SAPI interface component, the Speechify installer only installs the SAPI 5 run-time components. It does not contain the SAPI 5 SDK that you can download from here: <http://www.microsoft.com/speech/speechsdk/sdk5.asp>. You need to download the SDK to have access to SAPI 5 documentation and sample code.
- ❑ The Speechify licensing service is automatically configured to start at reboot.



NOTE

After installing Speechify, you may be asked to reboot the system. If you do so before installing Speechify licenses, you get a Windows error saying one or more services could not be started due to the Speechify Licensing Service not starting. It is safe to ignore this message until those licenses are installed.

Installing a voice

This section describes the installation procedures for Speechify voices. For performance and sizing information for each voice, contact ScanSoft technical support.

1. Log into Windows as the Administrator.
2. If you obtained the voice on a CD:
 - a. Insert the Speechify voice CD into the CD-ROM drive.
 - b. The installer should run automatically; if it does not, open the CD drive in Windows Explorer and run setup.exe.
3. If you downloaded the voice:
 - a. Open the Speechify voice WinZip package you downloaded from ScanSoft and extract the contents to a temporary directory.
 - b. Run setup.exe from the temporary directory.
4. Follow the on-screen instructions, reading the Welcome screen and the Software License Agreement; click Install.
5. Installation may take several seconds or minutes depending on the size of the voice data.
6. If you downloaded the voice, you may now delete the temporary directory where you extracted the Speechify installation package.



NOTE

To use a voice via Speechify's SAPI 5 interface when the Speechify server is on a different machine from the application using SAPI:

- ❑ Run the voice installer on the application machine so that the voice is known to SAPI.
- ❑ Reconfigure the voice to point to proper remote hostname. (The default settings for each voice point the SAPI interface to a Speechify server on the local machine.) You can do this via Start >> Settings >> Control Panel >> Speech >> Settings. See "SAPI 5" on [page 189](#) for details.

Uninstalling the Speechify client and server

To uninstall Speechify:

1. Go to Start >> Settings >> Control Panel >> Add/Remove Programs.
2. Remove each of the Speechify voice installations:
 - a. On the Install/Uninstall tab, choose the first “Speechify 3.0 Voice” item in the list of installed applications and click the Add/Remove button
 - b. Follow the instructions to uninstall the voice.
 - c. Repeat a and b until no “Speechify 3.0 Voice” items remain in the list of installed applications.
3. On the Install/Uninstall tab, choose Speechify 3.0 for Windows in the list of installed applications and click the Add/Remove button.
4. Follow the instructions to uninstall Speechify.

Uninstalling a voice

To uninstall a voice:

1. Go to Start >> Settings >> Control Panel >> Add/Remove Programs.
2. On the Install/Uninstall tab, choose the voice package in the list of installed packages and click Remove.
3. Follow the instructions to uninstall the voice.

Installation on Red Hat Linux

To install Speechify on Linux, first install the Speechify client and server, then install each voice separately.

Installing the Speechify server

1. Log into Linux as root.
2. Extract the Speechify tar.gz package you downloaded from ScanSoft to a temporary directory.
3. Change your working directory to the temporary directory.
4. Copy the server files onto the machine:

```
rpm --install Speechify-Engine-3.0-0.i386.rpm
```
5. You may now delete the temporary directory where you extracted the Speechify installation package.
6. You must install a Speechify voice before you can run the server. See “Installing a voice” on [page 11](#).

Installing the Speechify client

1. Log into Linux as root.
2. Extract the Speechify tar.gz package you downloaded from ScanSoft to a temporary directory.
3. Change your working directory to the temporary directory.
4. Copy the client files onto the machine:

```
rpm --install Speechify-Client-3.0-0.i386.rpm
```


5. You may now delete the temporary directory where you extracted the Speechify installation package.



NOTE

By default, the above packages install to /usr/local/SpeechWorks/Speechify. To relocate the packages, specify the --prefix option to install to another directory, e.g., rpm --install --prefix /home/Speechify Speechify-Engine-3.0-0.i386.rpm.

Installing a voice

1. Log into Linux as root.
2. If you obtained the voice on a CD:
 - a. Insert the Speechify voice CD into the CD-ROM drive.
 - b. Open a shell window and mount the CD-ROM. Then change directory to the directory that the CD-ROM device resides on. (This is usually /mnt/cdrom.)
3. If you downloaded the voice:
 - a. Extract the Speechify voice tar.gz package you downloaded from ScanSoft to a temporary directory.
 - b. Change your working directory to the temporary directory.
4. To install voice packages, the command takes this general form:

```
rpm --install Speechify-Vox-<LanguageCode>-<VoiceName>-  
<SampleRate>-<version>-0.i386.rpm
```

For example, to install the Speechify 3.0 version of the 8 kHz US English voice named Tom, type:

```
rpm --install Speechify-Vox-en-US-tom-8kHz-3.0-0.i386.rpm
```

To install the 16 kHz version of Tom, type:

```
rpm --install Speechify-Vox-en-US-tom-16kHz-3.0-0.i386.rpm
```

5. Installation may take several seconds or minutes depending on the size of the voice data.
6. If you downloaded the voice, you may now delete the temporary directory where you extracted the Speechify installation package.

Installed file locations

This table shows where certain files are located after installing various packages:

Directory	Description
/usr/local/SpeechWorks/Speechify/bin/	server binaries, pre-built sample applications, and e-mail preprocessor dictionary
/usr/local/SpeechWorks/Speechify/config/	baseline Speechify configuration files
/usr/local/SpeechWorks/Speechify/doc/	documentation files
/usr/local/SpeechWorks/Speechify/flexlm/	license server binaries, documentation, and license folder
/usr/local/SpeechWorks/Speechify/include/	header files for SWI libraries
/usr/local/SpeechWorks/Speechify/lib/	client library, e-mail pre-processor library, server internal libraries
/usr/local/SpeechWorks/Speechify/samples/	source for sample applications

Uninstalling the Speechify server

1. Log into Linux as root.
2. First, uninstall any voices you have installed. See “Uninstalling a voice” on [page 13](#).
3. Next, uninstall the Speechify server/engine files:

```
rpm --erase Speechify-Engine-3.0-0
```

Uninstalling the Speechify client

1. Log into Linux as root.
2. Next, uninstall the Speechify client files:

```
rpm --erase Speechify-Client-3.0-0
```

Uninstalling a voice

1. Log into Linux as root.
2. To remove a voice package, use the name of the package you installed without the .i386.rpm extension. To uninstall the examples above, use

```
rpm --erase Speechify-Vox-en-US-tom-8kHz-3.0-0
```

and

```
rpm --erase Speechify-Vox-en-US-tom-16kHz-3.0-0
```

Installation on Solaris

By default, the packages below install in /opt/SpeechWorks/Speechify/bin. To relocate the packages, specify the -R option to install to another directory, e.g.:

```
pkgadd -R /home/Speechify ttsClient
```

Also, you may have to explicitly set the path to where the gunzip executable is located. For example:

```
/usr/local/bin/gunzip  
Speechify.Client.3.0.0.sparc.pkg.tar.gz
```



NOTE

There is no supported Speechify server for Solaris, only the Speechify client. Therefore, there are no Speechify voices for Solaris.

Installing the Speechify client

1. Log into Solaris as root.
2. Copy the Speechify tar.gz package you downloaded from ScanSoft to a temporary directory.
3. Change to the directory you just created.
4. Unpack the client software:

```
gunzip Speechify.Client.3.0.0.sparc.pkg.tar.gz  
tar xvf Speechify.Client.3.0.0.sparc.pkg.tar
```

Add the package:

```
pkgadd -d . ttsClient
```

5. You may now delete the .tar package and the temporary directory where you extracted the Speechify installation package.

Installed file locations

This table shows where certain files are located after installing various packages:

Directory	Description
/opt/SpeechWorks/Speechify/bin/	pre-built sample applications, e-mail preprocessor dictionary
/opt/SpeechWorks/Speechify/doc/	documentation files
/opt/SpeechWorks/Speechify/include/	header files for SWI libraries
/opt/SpeechWorks/Speechify/lib/	client library, e-mail pre-processor library
/opt/SpeechWorks/Speechify/samples/	source for sample applications

Uninstalling the Speechify client

1. Log into Solaris as root.
2. Next, uninstall the Speechify client files:

```
pkgrm ttsClient
```

If you did not install Speechify in /opt/SpeechWorks/Speechify/bin, specify the -R option with the actual directory, e.g.:

```
pkgrm -R /home/Speechify ttsClient
```

Running the Speechify server from the console

Starting Speechify

The Speechify server can be started from the command line on all platforms. (It can also run as a service, or daemon, on Windows. See “Running Speechify as a Windows service” on [page 18](#).)

The Speechify binary is located in the /bin directory of the Speechify install directory. On UNIX platforms the binary is named Speechify and on Windows, Speechify.exe.

```
Speechify <options>
```

The following table lists the command-line options and their descriptions. Some switches require additional arguments.

Option	Description
--help	Prints a short summary of these switches and then exits.
--version	Prints a short banner and then exits. The banner displays the Speechify release, the client/server protocol version, and the build date and time.
--config	Specifies a Speechify configuration file to load. There may be more than one config option specified. When multiple config options are specified, values in later configuration files (rightmost config option) override values in prior configuration files. Default: <InstallDir>/config/SWIttsConfig.xml

See “Using configuration files” on [page 24](#) for information about starting Speechify with the provided configuration files.

Stopping Speechify

The Speechify server has the following shutdown modes. These modes are triggered differently depending on the operating system, and on Windows depending on whether the server is being run as a Windows Service or not. Users may “escalate” a

shutdown by sending a second, higher priority shutdown request (for example, doing a "kill -INT" on Linux, then getting impatient while waiting for speak requests to complete and doing a "kill -TERM").

Mode	Description
Graceful	<p>Refuse all new client connection attempts, waiting for speak requests to complete. As each request completes, that client connection is dropped. When all the connections are dropped, the server exits.</p> <ul style="list-style-type: none"> ❑ Windows Service: use the Speechify MMC "Stop Voice" button or right-click on the service and select the "Stop This Voice (graceful)" option, use Control Panel >> Administrative Tools >> Services to stop the voice service, or use a command-line Windows service control tool to stop the service ❑ Windows console: Ctrl-c ❑ Linux console: Ctrl-c or "kill -INT"
Interrupt	<p>Refuse all new client connection attempts, immediately disconnect all existing connections, then the server exits.</p> <ul style="list-style-type: none"> ❑ Windows Service: right-click on the service and select the "Stop This Voice (hard)" option, or use the Windows service control tool to send message 128 to the service ❑ Windows console: Ctrl-Break, shut down the console, log off, shut down the machine ❑ Linux console: "kill -QUIT" or "kill -TERM"
Hard	<p>Operating system level termination of the server, where all connections are abruptly terminated by the operating system. Not recommended unless attempts to perform an interrupt shutdown fail.</p> <ul style="list-style-type: none"> ❑ Windows Service: not possible (Windows restriction) ❑ Windows console: Task Manager "End Process" command ❑ Linux console: "kill -KILL" (same as "kill -9")

When a graceful or interrupt shutdown is initiated, Speechify reports an informational message to the error log. If a graceful shutdown does not complete within 10 seconds, Speechify reports a warning to the error log indicating that the shutdown has been delayed, indicating the number of speak requests that are still in progress. These warnings repeat every 10 seconds until the shutdown completes.

Running Speechify as a Windows service

On Windows systems, each Speechify voice is installed as a Windows service. This allows operational control of individual servers via standard Windows service control tools such as Control Panel >> Administrative Tools >> Services, the **net** command from the command line prompt, or even via WMI scripting through the Win32_Service object. For example, to start the server for the 8 kHz voice named Tom, type the following command from the command prompt:

```
net start SpfyTom8
```

To stop the server using “graceful” stop as described in “Stopping Speechify” on [page 16](#), use this command:

```
net stop SpfyTom8
```

To start the same server through the Control Panel Services application:

- ❑ Select “Speechify Voice – US English, Tom, 8kHz” in the service list
- ❑ Press the start (black triangle) button

To stop the same server through the Control Panel Services application:

- ❑ Select “Speechify Voice – US English, Tom, 8kHz” in the service list
- ❑ Press the stop (black square) button

The service names for other voices follow the same pattern. For example, the service name for the 16 kHz voice named Karen is “Speechify Voice – Australian English, Karen, 16kHz” in the Services list, and SpfyKaren16 as the short name used from the command prompt.

As installed, each voice's service is not set for automatic startup. If you need this (for example, if you want the voice to start automatically when your machine is rebooted), you can set it to Automatic via the Control Panel Services application. Double-click the service name for the voice to bring up its properties, then change Startup Type from Manual to Automatic and click OK.

When a voice server's service starts, it uses the configuration specified by combining the standard “baseline” configuration file (installed during the Speechify core installation) and the voice's own configuration file (installed during the voice's installation) located in:

```
<InstallDir>/config/SWIttsConfig.xml  
<InstallDir>/<language>/<name>/<name><format>.xml
```


For example, the configuration file for the 8 kHz voice named Jill is <InstallDir>/en-US/jill/jill8.xml. The contents and format of these files are described in [Chapter 13](#). Although you can edit these files manually, you can edit most common configuration settings via the MMC snap-in.

Using the MMC snap-in

The Microsoft Management Console (MMC) is a standard Windows framework that contains various administrative tools for managing systems. Although Windows includes many standard administrative tools, the MMC is extensible so that software providers such as ScanSoft can provide “snap-ins” that users can plug into their own consoles. These snap-ins provide a simple interface to tasks that are unique to a specific product.

Speechify comes with an MMC snap-in that lists the installed voices on a machine and lets you start them, stop them, and configure some of their behavior by editing various parameters. Speechify also comes with a default MMC console that is configured to contain the Speechify snap-in. You do not have to use this console; you can add the snap-in to other consoles.

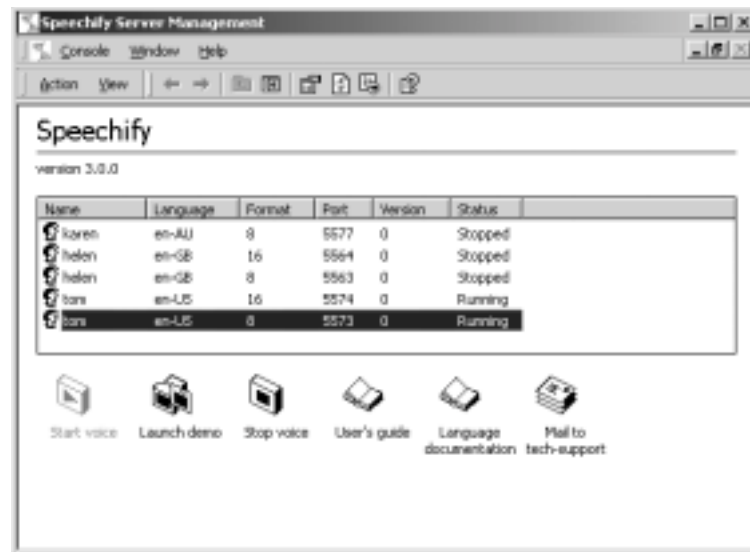
Also note that the default console supplied with Speechify is a console “taskpad.” This is a form of console that wraps the usual snap-in table with additional GUI elements to increase ease of use. Although the instructions below refer to menu items such as “Start This Voice,” note that there are additional buttons provided in the taskpad that map to these items.

Starting the Speechify MMC snap-in

To start the Speechify MMC snap-in, use this path:

Start >> Programs >> Speechify >> Speechify Server Management

The MMC console appears in a Speechify Server Management window. The snap-in displays a list of Speechify voices installed on the machine. For example:



Starting and stopping a voice via the snap-in

To start a voice that is not currently running, right-click on the voice and choose **Start This Voice**. To stop a voice, right-click and choose either **Stop This Voice (graceful)** or **Stop This Voice (hard)**. In each case, a dialog box appears with a progress bar to indicate the status of the request. See “Stopping Speechify” on [page 16](#) for a description of “graceful” and “hard.”

Editing a voice configuration

Use the Properties dialog box to view or edit certain properties of a voice. To do so, either double-click on a voice or right-click and choose **Properties**. A dialog box appears with several tabs (described below). Once you have finished editing any settings, click **Ok** to make them permanent. Click **Cancel** to cancel any changes.



NOTE

Any configuration changes made to a running server do not take effect until the next time the server is started.

The tabs on the Properties dialog box are named:

- ❑ General
- ❑ Network
- ❑ Logging

The General tab contains several read-only fields of information about the voice:

Field name	Description
Name	The name of the voice, e.g., tom or helen.
Language	The voice's language in ISO form, e.g., en-US.
Format	The voice's database format.
Voice version	The voice's version number, e.g., 0, 1, or 2.
Minimum Speechify server version required	The minimum server version required to run this voice. Otherwise the server exits with an error message on startup.
Installation path:	The path to which the voice was installed.

The Network tab contains options related to network connections:

Field name	Description
Sockets port	Specifies the sockets port where the Speechify server should listen for incoming connections. If you are running multiple instances of Speechify on one server machine, this number must be different for each instance.
Maximum connections	Number of Speechify client connections to support (number of ports opened via <code>SWIttsOpenPortEx()</code> against this server instance).

The Logging tab contains options related to logging:

Field name	Description
Diagnostic File	The file name used for diagnostic logs or an empty field to disable logging to a diagnostic file.
Event File	The file name used for event logs or an empty field to disable logging to an event file.
Cache Directory	The path used to cache fetched HTTP items.

These MMC options correspond to the configuration parameters in the following table. See “Using configuration files” on [page 24](#) and “Configuration parameters” on [page 175](#).

Field name	Parameter name
General >> Name	tts.voice.name
General >> Language	tts.voice.language
General >> Format	tts.voice.format
General >> Voice version	<none>
General >> Minimum Speechify server version required	<none>
General >> Installation path:	tts.voice.dir
Network >> Sockets port	tts.server.port
Network >> Maximum connections	tts.server.numPorts
Logging >> Diagnostic File	tts.log.diagnostic.file
Logging >> Event File	tts.log.event.file
Logging >> Cache Directory	tts.cache.dir

Internet fetching

Speechify allows you to use the Internet to fetch content to speak from a URI (with `SWIttsSpeakEx()`), load a dictionary from a URI (with `SWIttsDictionaryLoad()`), and play pre-recorded audio (with the W3C SSML `<audio>` element).

To configure Speechify fetching and caching, use the configuration file cache and Internet fetch parameters. (See “Cache parameters” on [page 176](#) and “Internet fetch parameters” on [page 179](#).)

For information about URI handling, see the description of the `SWIttsDictionaryData` structure under “`SWIttsDictionaryLoad()`” on [page 148](#) and the description of the `SWIttsSpeakData` structure under “`SWIttsSpeakEx()`” on [page 164](#).

Configuring the web server to return a proper MIME content type

Speechify can be used with a web server that can deliver content including documents to speak, audio files to insert via W3C SSML `<audio>` elements, and dictionaries. When a web server delivers the content, it also provides the MIME type for the content and information about the desired caching behavior for the content.

The MIME type being delivered for specific content types must be configured for Speechify to function correctly. These sections list the MIME types that should be returned by the web server for content types supported by Speechify:

- ❑ Dictionaries: “SWIttsDictionaryLoad()” on [page 148](#).
- ❑ Text to be spoken: “SWIttsSpeakEx()” on [page 164](#).
- ❑ W3C SSML `<audio>` insertion fetches: “Supported audio formats” on [page 104](#).

As a workaround for web servers that are not configured correctly to return these MIME types, set the `tts.inet.extensionRules` parameter in your Speechify configuration file. (See “`tts.inet.extensionRules`” on [page 179](#).)

Configuring a web server to optimize performance

This section describes how to configure a web server to optimize performance by establishing a caching policy.

When content from a URI is required (such as from a `SWIttsSpeakEx()` call, `SWIttsDictionaryLoad()` call, or speaking a W3C SSML `<audio>` element), the Internet fetch cache is consulted to determine whether previously fetched data can be used instead of contacting the web server. If the data is present and valid, the cache entry is used. HTTP/1.1 specifies two methods for verifying whether cached entries are valid:

1. First, check the expiration time against the current time.

Each cache entry contains an expiration time that was returned by the web server for the previous fetch (most commonly via a “Cache-control: max-age” or “Expires” header). A cache entry is assumed to be valid if it has not expired (i.e., if its expiration time is later than the current time).

2. Second, if a cache entry is expired, perform an HTTP conditional fetch based on last-modified time and entity tags.

Speechify sends the last-modified time and any entity tags (typically a hash calculated off the data) returned by the web server for the cached fetch. The web server can then use the last-modified time and/or entity tags to determine whether to return updated URI data (cached entry is invalid) or whether to inform Speechify that it may use the cached entry (cached entry is valid).

To minimize HTTP network traffic and thus minimize Speechify CPU use and response times, configure the HTTP server to return appropriate expiration times for fetches. Since no communication occurs between Speechify and the web server if a cache entry is not expired, setting nonzero expiration times can reduce HTTP fetches significantly. Some web servers provide the convenience of defining a default value for these cache controls based on the file extension or document MIME type being fetched.

Using configuration files

Speechify uses XML configuration files to configure the server. These configuration files are loaded from the Speechify server command line and modified by hand or, on Windows, with the Speechify MMC snap-in.

When starting the server from the command line, the typical command specifies two configuration files: first the baseline configuration file for the system-level default parameters, then the voice-specific configuration file for the voice-specific parameters. When running Speechify as a Windows service, the behind-the-scenes service command-line arguments use this form as well.

For more information about configuration files, see [Chapter 13](#). For information about starting Speechify, see “Starting Speechify” on [page 16](#).

Default configuration files

Speechify server and voice installations provide default Speechify configuration files. The *Speechify server* installation includes the system-wide configuration file, `<InstallDir>/config/SWIttsConfig.xml`. This file defines system-level defaults for parameters such as the default format for text and the default Speechify licensing mode.

Each *Speechify voice* installation also includes an overlay configuration file, <VoiceName><SampleRate>.xml in the voice installation directory, e.g., en-US/tom/tom8.xml for the U.S. English Tom 8 kHz voice. This file defines the voice name, language, sampling rate, and a unique server TCP/IP port number for that voice, as well as other parameters that are frequently edited on a per voice basis.

Windows example:

```
Speechify --config "%SWITTSSDK%\config\SWittsConfig.xml"  
--config "%SWITTSSDK%\en-US\tom\tom8.xml"
```

Linux example:

```
Speechify --config ${SWITTSSDK}/config/SWittsConfig.xml  
--config ${SWITTSSDK}/en-US/tom/tom8.xml
```

User-defined configuration files

Advanced users can choose to add additional layers of configuration files, such as an additional network-wide configuration file that overrides the baseline defaults. This network configuration file might be inserted between the ScanSoft supplied baseline and voice specific configuration files.

Configuring the Speechify SAPI 5 interface

The SAPI 5 API was not explicitly designed to allow for client/server speech synthesis systems such as Speechify. By default, the Speechify installer configures Speechify's SAPI 5 interface to look for the server on the same machine as the client (i.e., localhost) but it is possible to configure the interface to look for a server elsewhere on the network, even on a non-Windows machine. To do this configuration, the interface provides a properties dialog box accessible via the Control Panel Speech application. This dialog box also lets you examine the voice's attributes and turn diagnostic logging on or off for technical support purposes. See "SAPI voice properties" on [page 191](#) for details.



Speechify Logging

This chapter describes the Speechify error, diagnostic, and event logs: what information is reported in the logs and how to control the logging behavior.

In This Chapter

- ❑ “Overview of logs” on [page 27](#)
- ❑ “Client logging” on [page 29](#)
- ❑ “Server logging” on [page 32](#)
- ❑ “SAPI 5 logging” on [page 33](#)
- ❑ “Error and diagnostic message format” on [page 33](#)
- ❑ “Event message format” on [page 35](#)

Overview of logs

This section describes Speechify’s comprehensive logging facilities:

Log type	Brief description	Supported by...
“Error log” on page 28	Report system faults (errors) and possible system faults (warnings) to the system operator for diagnosis and repair	Speechify client and server; SAPI 5 interface
“Diagnostic log” on page 28	Trace and diagnose system behavior for developers and ScanSoft support staff	Speechify client and server; SAPI 5 interface
“Server event log” on page 28	Stores information for capacity planning, performance monitoring, monitoring application synthesis usage, and application tuning	Speechify server only

Error log

Speechify uses the error log to report system faults (errors) and possible system faults (warnings) to the system operator for diagnosis and repair. The client and server report errors by error numbers that are mapped to text with an associated severity level. This mapping is maintained in a separate XML document that allows integrators to localize and customize the text without code changes. It also allows integrators and developers to review and re-define severity levels without code changes. Optional key/value pairs are used to convey details about the fault, such as the file name associated with the fault. Speechify installs a US English version of this XML document in <InstallDir>/doc/SpeechifyErrors.en-US.xml.

Diagnostic log

Developers and ScanSoft support staff use the diagnostic log to trace and diagnose system behavior. Diagnostic messages are hard-coded text since they are private messages that are not intended for system operator use. Diagnostic logging has little or no performance impact if disabled but can impact system performance if turned on.

Server event log

The Speechify server can automatically log usage activity to a file with a well-defined format. Set the `tts.log.event.file` configuration parameter to turn on this "event logging." The events that the server logs describe such things as speak requests, acquiring and releasing licenses, client connections opened and closed, automatically generated pronunciations that may require tuning, and cross-references to the error log for error conditions. The event log summarizes usage activity in order to support reporting for such things as capacity planning, performance, application synthesis usage, and application tuning.

Key/value pairs are used to convey details about each event, such as the number of characters in a speak request or the number of bytes of audio sent to the client.

You can process the file with tools such as Perl or grep to view statistics, identify problem areas, or generate reports.



NOTE

Do not confuse the event log with the error log (not all error details are logged to the event log) or with the diagnostic log that ScanSoft technical support may request to resolve support issues.

Client logging

Overview of client logs

The client produces two classes of messages: error and diagnostic. Error messages are always on, and diagnostic messages are optional (disabled by default). Note that these diagnostic and error messages are local messages from the Speechify client: the server does not deliver diagnostic and error messages to the client for reporting by the client library. For example, if the client loses the Speechify server connection, or if an invalid argument (such as a NULL pointer) is passed to an API function, the Speechify client library detects these failures itself and reports them to the application with a client library error message. The Speechify server detects other types of errors, e.g., a W3C SSML document parse failure. Thus, only the server error log contains the *details* of the parse error, and the Speechify client library reports the parse error only via a `SWItts_SSML_PARSE_ERROR` return code from `SWIttsSpeak()`. Although it is important to monitor the Speechify client error and diagnostic logs, it is also important to monitor the Speechify server logs as well.

The Speechify client library indicates error and (if enabled) diagnostic messages to an application by passing `SWItts_cbLogError` and `SWItts_cbDiagnostic` events to the application's `SWIttsCallback()` function. These events allow applications to integrate Speechify client logging into the application log, simplifying monitoring and administration of the final integrated application. For simple interactive applications, the client library can optionally report these messages to a log file or to standard output, enabling applications to potentially ignore these events in the application's `SWIttsCallback()` function.

The `SWItts_cbLogError` and `SWItts_cbDiagnostic` events have a structure associated with them, `SWIttsMessagePacket`, that the client sends with the event. This structure contains detailed information that let the application writer determine how, where, and in what format, logging should happen. For more information on these codes and the structure, see “`SWIttsCallback()`” on [page 137](#).

Speechify client library error and diagnostic messages reported to a log file or to standard output also have a well-defined format as documented below.

Enabling diagnostic logging

Diagnostic logging is intended for diagnosing application or Speechify client library issues, not for general run-time use. Because the client can output a very large amount of diagnostic messages, and doing so can have a significant performance impact, the client does not report diagnostic messages unless you explicitly tell it to do so by setting an environment variable named `SWITTSLOGDIAG`. When you set this variable to *any* value, the logging is turned on. Unless it is set, the client library sends no diagnostic messages to `SWIttsCallback()`, and does not log diagnostic messages to stdout/stderr or a log file. This example is from a Windows command shell:

```
set SWITTSLOGDIAG=1
```

Logging to a file or stdout/stderr

While recommended, it is not necessary to implement code in the application's `SWIttsCallback()` function to make Speechify client error and diagnostic messages visible to the system operator. To guarantee that ScanSoft technical support staff can always obtain a log, default logging abilities are built into the client. To enable these, set one or more of the following environment variables. Note that these variables merely create additional output locations: even if these variables are set, the application callback function is still invoked with the status codes `SWItts_cbLogError` and (if diagnostic logging is enabled) `SWItts_cbDiagnostic`.

Variable	Description
<code>SWITTSLOGTOSTD</code>	If this variable exists (value is unimportant) then the client library logs error and (if enabled) diagnostic messages to stdout and stderr.

Variable	Description
SWITTSLOGTOFILE	File name where error and diagnostic messages are written.
SWITTSMAXLOGSIZE	Maximum size of the log file configured via SWITTSLOGTOFILE in Kb. If this variable is not set, there is no limit on the log file size. When the log reaches the size limit, it rolls over into a file named the same as the log file with a .old suffix, then the logging resumes.

Each client error message line contains the following fields. Each field is separated by the “|” character. Key/value pairs are specified by a sequence of letters and digits for the key name, followed by the “=” character, followed by the value:

- ❑ A timestamp
- ❑ The port number associated with the error (-1 if the port number was unknown at the time of the message generation)
- ❑ The thread instance in which the error occurred (currently the operating system thread ID)
- ❑ The error ID number
- ❑ Zero or more key/value pairs with data specific to that error

```
Sep 22 11:45:33.19|0|7603|121|socketfunc=connect() |
  message=Non Fatal Error : 10061, WSAECONNREFUSED:
  Connection refused
```

In the above example, the error occurred on September 22, at approximately 11:45 am. The port number was 0, the thread ID was 7603, and the error number was 121. Looking at the SpeechifyErrors.en-US.xml file, this error code refers to “Speechify client: Socket operation failed”. The socketfunc key/value pair reports that the operating system connect function failed, and the message key/value pair reports the operating system error text for the failure (connection refused by the server).

Client diagnostic message lines are similar, except that a hardcoded diagnostic message is reported instead of the error ID number and key/value pairs:

- ❑ A timestamp
- ❑ The port number associated with the error (-1 if the port number was unknown at the time of the message generation)
- ❑ The thread instance in which the error occurred (currently the operating system thread ID)
- ❑ The diagnostic message

For example:

```
Sep 22 11:45:33.19|-1|7603|Entered SWIttsInit
```

In the above example, the diagnostic message was reported on September 22, at approximately 11:45 am. The port number was unknown (-1), and the thread ID was 7603. The diagnostic message indicates that `SWIttsInit()` was called by the application.

Server logging

Default logging behavior

By default, the Speechify server logs error messages to a file, to the system log (Windows Event Log or Unix syslog facility), and to stdout (not stderr). By default, the Speechify server does not log diagnostic messages or events.

Controlling logging

The default server logging behavior can be modified through server XML configuration file options. When running the server on Windows, you can also control these options via the Speechify MMC snap-in which edits the XML configuration file for you. The remainder of this section discusses the most important logging options to consider. For details on how to configure each of these options, and for information on other more detailed logging configuration options, see “Log parameters” on [page 180](#). Windows users should also see “Using the MMC snap-in” on [page 19](#) for more details on using the MMC snap-in.

As mentioned above, the server logs errors (and if enabled) diagnostics to stdout, the system log (Windows Event Log or Unix syslog facility), and to a file. If you wish to change these, use `tts.log.diagnostic.toStdout` to enable/disable logging to standard output, use `tts.log.diagnostic.toSystemLog` to enable/disable logging to the system log, and use `tts.log.diagnostic.file` to change the log file (set it to an empty string to disable logging to a file). There are additional configuration parameters for controlling the log file format, rollover size, and other details.

Speechify reports errors by number along with associated severity and error text loaded from a Speechify XML error mapping file. The default Speechify XML error mapping file is `SpeechifyErrors.en-US.xml`. You can optionally change `tts.log.diagnostic.errorMapFile` to redirect the server to a customized version with different text and/or severity levels, including translating the error text to a different language. Speechify supports XML error mapping files using any character set that is supported for Speechify XML dictionaries and W3C SSML documents, including ISO-9959-1 (Latin 1), UTF-8, UTF-16, and Shift-JIS.

To enable event logging, change `tts.log.event.file` to configure an event log file, making sure this file is unique for each server instance (Speechify server voice and format pair) running on the system by including the voice name and format in the file name. The recommended value is to take advantage of Speechify server configuration file variables and specify: `${TMPDIR}/SWIttsDiagnosticLog_${tts.voice.name}_${tts.voice.format}_${USER}.txt`. See “Event message format” on [page 35](#) for more details on event logging.

Like the Speechify client, the server does not report diagnostic messages by default. Diagnostic messages should only be enabled when requested by ScanSoft support for diagnosing server issues. Otherwise the output size would be very large and significantly degrade server performance. To enable detailed diagnostic logging on the server, ScanSoft Technical Support staff will supply entries to add to the `tts.log.diagnostic.tags` section of your Speechify XML configuration file.

SAPI 5 logging

The SAPI 5 interface for Speechify also has comprehensive logging abilities. The SAPI 5 interface logs error messages to the Windows Event Log. It can also log diagnostic content to a file for review by Speechify technical support. To enable this diagnostic logging, see “SAPI 5” on [page 189](#) for instructions.

Error and diagnostic message format

Each server error message line contains the following fields. Each field is separated by the “|” character. Key/value pairs are specified by a sequence of letters and digits for the key name, followed by the “=” character, followed by the value:

- ❑ A timestamp
- ❑ The thread instance in which the error occurred (currently the operating system thread ID)
- ❑ A server logical channel number associated with the error (-1 for global server messages not associated with a single channel)
- ❑ An error severity indication as loaded from the Speechify XML error mapping file (CRITICAL to indicate the server is completely out-of-service, SEVERE to indicate a major server fault that impacts one or more client connections, WARNING to indicate an application fault that impacts a single client operation, and INFO to indicate an important server informational message such as startup or shutdown)
- ❑ The error ID number
- ❑ The error text for that error ID number as loaded from the Speechify XML error mapping file
- ❑ Zero or more key/value pairs with data specific to that error

Examples:

```
Sep 13 14:01:13.52|329||0|WARNING|5066|Speechify server:  
SSML XML parse fatal error|uri=http://myserver/ssml1.xml|  
line=1|column=1|message=Invalid document structure
```

In the above example, the error message was reported on September 13, a little after 2 pm. The thread ID was 329, the logical channel was 0, the severity was WARNING (an application level issue), and the error number was 5066. This error indicates the W3C SSML document fetched from <http://myserver/ssml1.xml> had an error starting at line 1 column 1 where the XML document structure was invalid (most likely did not have a valid XML header).

```
Apr 18 11:56:44.76|340||32|WARNING|6287|Speechify server:  
Invalid SPR entered|SPR=\\![k]|reason=No vowel in syllable  
or word
```

In this example, on logical channel 32 Speechify found an invalid SPR (pronunciation), `\\[k]`, in a dictionary or other input text. The log entry includes both the SPR and the reason why it's incorrect. (The list of possible reasons is listed in "SPR format and error handling" on [page 96](#).)

Server diagnostic message lines are similar, except that a diagnostic tag ID number and a hardcoded diagnostic message is reported instead of the error severity, error ID number, and key/value pairs:

- ❑ A timestamp
- ❑ The thread instance in which the error occurred (currently the operating system thread ID)
- ❑ A server logical channel number associated with the error (-1 for global server messages not associated with a single channel)
- ❑ The diagnostic tag ID
- ❑ The diagnostic message

Event message format

The Speechify event log can be configured as a UTF-8 (Unicode, the default) or ISO-8859-1 (Latin-1) text file that contains one event per line. An event never spans more than one line. Each line is terminated by a new-line control sequence; Unix systems use “\n”, Windows systems use “\r\n”.

Each server event line contain the following fields. Each field is separated by the “|” character. Token/value pairs are specified by a sequence of letters and digits for the token name, followed by the “=” character, followed by the token value:

- ❑ A timestamp with month, day, and time to the hundredth of a second.
- ❑ Two token/value pairs that describe the amount of CPU the current server instance has used for that specific client connection so far:
 - The UCPU token records the User CPU used since the client connection was established.
 - The KCPU token records the total Kernel CPU used since the client connection was established.
- ❑ A server logical channel number associated with the event (-1 for global server messages not associated with a single channel)
- ❑ The event name
- ❑ Zero or more token/value pairs of data specific to that event.



NOTE

Any script or tool written to process Speechify event log files should allow for and ignore any unknown events or unknown fields within an event. Otherwise that script or tool may break with future Speechify releases.

Example event log

This sample shows a simple series of events: the server starts (STRT then INIT), a client connects (OPEN), the client sets two parameters (SETP) and makes one speak request (SPKB then FAUD), the request completes (SPKE), then the client closes the connection (CLOS).

```
Mar 18 10:46:56.47|UCPU=120|KCPU=250|-1|STRT|VERS=3.0.0|
    PROT=20030203|NAME=tom|LANG=en-US|FRMT=8|HOST=andromeda1|
    PORT=5573
Mar 18 10:46:57.15|UCPU=1021|KCPU=530|-1|INIT
Mar 18 10:47:25.27|UCPU=0|KCPU=0|0|OPEN|clientIP=127.0.0.1
Mar 18 10:47:25.32|UCPU=20|KCPU=10|0|SETP|
    NAME=tts.marks.word|VALU=false
Mar 18 10:47:25.37|UCPU=20|KCPU=10|0|SETP|
    NAME=tts.marks.phoneme|VALU=true
Mar 18 10:47:25.53|UCPU=40|KCPU=20|0|SPKB|NBYT=164|NCHR=81
Mar 18 10:47:28.40|UCPU=70|KCPU=40|0|FAUD
Mar 18 10:47:28.59|UCPU=70|KCPU=50|0|SPKE|XMIT=23651
Mar 18 10:47:28.62|UCPU=70|KCPU=50|0|CLOS
```

Description of events and tokens

The following list shows the current set of Speechify event names and tokens.



NOTE

This list may expand in future releases. Any tool written to process Speechify event log files should allow for and ignore any unknown events or unknown tokens within an event.

Description	Token names	Example values	Meaning
STRT – Indicates that Speechify has launched and is beginning the initialization phase.			
	VERS	3.0.0	The version of Speechify
	PROT	20030203	The version of the client/server protocol used.
	LANG	en-US	The language of the server.
	NAME	tom	The name of the voice used.
	FRMT	8	The audio format of the speech database.

Description	Token names	Example values	Meaning
	HOST	myserver	The server host name.
	PORT	5573	The sockets port the server listens on.
INIT – Initialization done (no tokens)			
OPEN – Open connection from client			
	CLIP	10.2.1.244	The IP address of the client making the connection.
CLOS – Close connection to client (no tokens)			
SPKB – Speak begin. Logged when a speak request is received.			
	NBYT	4000	The number of bytes in the synthesis text.
	NCHR	2000	The number of characters in the synthesis text.
FAUD – First audio. Logged when the first audio packet is sent to the client. (no tokens)			
STRQ – Stop request (no tokens)			
STOP – Stop			
	XMIT	5678	The number of bytes of audio sent to the client before stopping.
SPKE – Speak end			
	XMIT	10450	The number of bytes of audio sent for the speak request.
SETP – Set parameter			
	NAME	tts.marks.word	The name of the modified parameter.
	VALU	false	The new value of the parameter.
ERR – Error			
	ERRN	10002	The server error number

Description	Token names	Example values	Meaning
IFET – Internet fetch			
	URL	http://myserver/ssml1.xml	The URL being fetched
	SRC	fetchCache	Data source for the final URL data: <ul style="list-style-type: none"> <input type="checkbox"/> fetchCache: loaded from the local Internet fetch disk cache <input type="checkbox"/> localFile: URL is a local file, accessed from disk <input type="checkbox"/> internetFetch: URL was fetched from a web server
PRON – Word pronunciation generated by text-to-phoneme rules			
	WORD	phoneme	The word whose pronunciation was generated by text-to-phoneme rules
	SPR	\[.1fo.2him]	The word's pronunciation in SPR symbols.
	CNT	1	The number of times the word's pronunciation was generated by the text-to-phoneme rules in the speak request.
LOCK – Attempt to obtain a license			
	VOIC	en-US_Tom	The language and voice name
	RSLT	ok	Result of the lock operation: <ul style="list-style-type: none"> <input type="checkbox"/> ok: success <input type="checkbox"/> fail: failure
UNLO – License released			
	VOIC	en-US_Tom	The language and voice name



Performance and Sizing

This chapter describes the methodology used to characterize the performance and scalability of Speechify. It contains an overview of the method chosen, a description of the tool that ScanSoft uses to run tests, and some details about specific metrics in the test.

Actual test results for any given Speechify release, platform, and voice are provided in a document that can be downloaded from ScanSoft technical support.

Ultimately, the model ScanSoft chose to measure Speechify performance and the criteria for interpreting the test results is arbitrary when compared to each individual user's deployment context. ScanSoft has tried to pick the most representative approach, but the reader should be aware that their own situation might have different requirements.

In This Chapter

- ❑ “Testing methodology” on [page 40](#)
- ❑ “Test scenario and application” on [page 40](#)
- ❑ “Performance statistics” on [page 41](#)
- ❑ “Resource consumption” on [page 42](#)
- ❑ “Determining the supported number of channels” on [page 43](#)

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.

ScanSoft chooses to test Speechify 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 ScanSoft model, if the testing application requests 10 seconds of audio from the server 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, the test application and the Speechify client are run on a machine separate from that hosting the server.

Test scenario and application

The Speechify SDK contains a set of sample applications. One of these samples, named `ttsLoad`, is the same testing application ScanSoft uses to test Speechify performance. Because of the number of variables in client/server configurations, access to this tool allows users to test performance on their specific configuration. For usage details including command-line options, see the file `<InstallDir>\samples\loadtest\README.txt`.

The application submits text to the server 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 the server. A

few of these are described in the sections below. All of the measurements, except server CPU utilization and memory usage, are made on the application side of the Speechify API.

Speechify's performance depends on the text being synthesized, including its content, complexity, individual utterance length, presence of unusual strings, abbreviations, etc. Sample texts are taken from a selection of newspaper articles with each heading and paragraph forming a separate speak request that is sent to the server in a random order, observing the timing model described above. The Speechify sample supplies the input texts that ScanSoft uses for testing. These input texts are included in files named `loadText_en.txt`, `loadText_ja.txt`, etc., for the different languages.

Performance statistics

ScanSoft uses several metrics to evaluate test results. You should understand their meaning to estimate the tasks and loads that Speechify 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 41](#)
- ❑ “Audio buffer underflow” on [page 42](#)

Latency (time-to-first-audio)

Applications with performance concerns start with the most obvious criterion: once the application asks Speechify 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 `SWIttsSpeak()` and the arrival of the first corresponding audio packet in the callback function. Although Speechify 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 the server 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 server-side metrics needed for proper sizing estimation: CPU use and memory use.

Server CPU utilization

Server CPU utilization is the percentage of the total CPU time spent processing user or system calls on behalf of Speechify. As more TTS ports are opened and used for processing, CPU utilization increases approximately linearly. When CPU utilization exceeds 85%, the Speechify server starts to saturate and performance degrades rapidly as further ports are opened.

Memory use

The Speechify server's base memory requirement varies per voice, and is usually on the order of 50–80 MB. Contact ScanSoft technical support for detailed sizing data that describes the base memory usage for each voice. In addition, each engine instance (correlating to each connection created by calling `SWIttsOpenPortEx()`) requires incremental amounts of memory, usually about 1 MB. Therefore, the formula to calculate total memory requirements is: $\text{base} + (\# \text{ engines} * 1\text{MB})$. For 50 instances of a voice that has a base of 60 MB, that results in a memory use of 110 MB.

Determining the supported number of channels

ScanSoft reduces all of these metrics down to one number, claiming that Speechify 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 $\leq 250\text{ms}$
- ❑ Audio buffer underflow rate $\leq 0.04\%$
- ❑ CPU utilization $\leq 85\%$

As mentioned above, these thresholds may not suit your requirements and you may wish to rerun tests with different criteria



Programming Guide

The chapters in this part cover these topics:

[Chapter 4 “Programming Guide”](#) contains an overview of features, and high-level information on groups of API functions.

[Chapter 5 “Using User Dictionaries”](#) describes how to load and activate dictionaries, and explains the dictionary format.



Programming Guide

This chapter contains an overview of features, and high-level information on groups of API functions.

For a detailed explanation of the API functions, see “API Reference” on [page 133](#).

In This Chapter

- ❑ “Features” on [page 48](#)
- ❑ “Order of API functions” on [page 49](#)
- ❑ “Callback function” on [page 50](#)
- ❑ “Sample time lines” on [page 52](#)
- ❑ “Overview of user dictionaries” on [page 56](#)
- ❑ “Parameter configuration” on [page 56](#)
- ❑ “Character set support” on [page 57](#)
- ❑ “Using W3C SSML” on [page 57](#)
- ❑ “Using SAPI” on [page 58](#)
- ❑ “Implementation guidelines” on [page 58](#)

Features

Speechify offers the following features:

- ❑ A client with a C language interface.
- ❑ A flexible threading model so that you can call the API functions from any thread in your process.
- ❑ Support for a variety of character sets.
- ❑ A variety of tags to be embedded in text to customize the speech output.
- ❑ Bookmark, wordmark, and phoneme-mark support, for synchronizing external events with the speech output.
- ❑ Support for file-based dictionaries, to be loaded at initialization or run time.

Order of API functions

This section provides an outline for your Speechify application:

1. First, call the `SWIttsInit()` function to initialize the client library. This is done once per application process no matter how many TTS ports are opened so that the library can allocate resources and initialize data structures.

You must call this function before calling any other; the other functions return an error if it has not been called.

2. Call `SWIttsOpenPortEx()` to open a TTS port on the server. This establishes a connection to the server. `SWIttsOpenPortEx()` registers an application-defined callback function with the Speechify client library. This callback function is invoked to receive all audio, supporting information, and errors generated by the server.

3. Call `SWIttsSpeak()` to start synthesizing the specified text on the server.

After receiving this call, the server begins sending audio packets to the callback function defined above. The `SWIttsSpeak()` call itself returns without waiting for synthesis to complete or even begin.

4. Call `SWIttsClosePort()` after you finish making speak requests, to close down the TTS port. The client tells the server to shutdown this port and then closes the connection to the server.

5. Call `SWIttsTerm()` when your process is ready to shut down. This function tells the client to shut itself down and clean up any resources.

Pseudo-code for a simple Speechify application looks like this:

```
SWIttsInit()
SWIttsOpenPortEx()
while (more TTS requests to make) {
    SWIttsSpeak()
    wait for SWItts_cbEnd to be received by the callback
}
SWIttsClosePort()
wait for SWItts_cbPortClosed to be received by the callback
SWIttsTerm()
```

Call `SWIttsStop()` to stop a TTS request before it has completed. Once this is called, the client sends no further information to your callback except a confirmation that the server has stopped acting on the request.

Callback function

Of the functions mentioned above, all operate synchronously except for `SWIttsSpeak()`, `SWIttsStop()`, and `SWIttsClosePort()`. When one of these three functions is called, it returns immediately before the operation is complete. A return code indicating success only indicates that the message was communicated to the server successfully. In order to capture the output produced and returned by the server, you must provide a function for the client library to call. This function is called a *callback* function and you pass a pointer to it when you call `SWIttsOpenPortEx()`. Your callback function's behavior must conform to the description of the `SWIttsCallback()` function specified in “`SWIttsCallback()`” on [page 137](#).

When the client library receives data from the server, it calls your callback and passes a message and any data relevant to that message. The set of possible messages is defined in the `SWItts.h` header file in the enumeration `SWItts_cbStatus` and listed below along with the type of data that is sent to the callback in each case:

<code>SWItts_cbStatus</code>	Data type
<code>SWItts_cbAudio</code>	<code>SWIttsAudioPacket</code>
<code>SWItts_cbBookmark</code>	<code>SWIttsBookMark</code>
<code>SWItts_cbDiagnostic</code>	<code>SWIttsMessagePacket</code>
<code>SWItts_cbEnd</code>	<code>NULL</code>
<code>SWItts_cbError</code>	<code>NULL</code>
<code>SWItts_cbLogError</code>	<code>SWIttsMessagePacket</code>
<code>SWItts_cbPhonememark</code>	<code>SWIttsPhonemeMark</code>
<code>SWItts_cbPortClosed</code>	<code>NULL</code>
<code>SWItts_cbStart</code>	<code>NULL</code>
<code>SWItts_cbStopped</code>	<code>NULL</code>
<code>SWItts_cbWordmark</code>	<code>SWIttsWordMark</code>

The `SWIttsAudioPacket`, `SWIttsBookMark`, `SWIttsWordMark`, `SWIttsPhonemeMark`, and `SWIttsMessagePacket` structures are defined and further explained in “`SWIttsCallback()`” on [page 137](#).

For a speak request that proceeds normally, message codes occur in the following manner:

- ❑ A `SWItts_cbStart` message arrives indicating that `SWItts_cbAudio` messages are about to arrive.

- ❑ One or more SWItts_cbAudio messages arrive along with their SWIttsAudioPacket structures. The SWIttsAudioPacket contains a pointer to a buffer of audio samples.
- ❑ A SWItts_cbEnd message arrives indicating that all speech has been sent.

For example, to write the synthesized speech to a file, the pseudo-code for your callback might look like this:

```
if message == SWItts_cbStart
    open file
else if message == SWItts_cbAudio
    write buffer to file
else if message == SWItts_cbEnd
    close file
```

You must copy audio from the SWIttsAudioPacket structure to your own buffers before the callback returns. Once the callback returns, the client library may free the SWIttsAudioPacket structure.

Sometimes a speak request does not proceed normally, such as when you call SWIttsStop() to stop a speak request before it completes. (You might want to do this to support barge-in.) In this case, no more audio messages are sent to your callback, and instead of receiving the SWItts_cbEnd message, your callback receives the SWItts_cbStopped message, indicating that the speak request was stopped before proper completion.

There are other ways that messages can proceed if you use the bookmark, wordmark, and phoneme-mark features. These are shown in the sample time lines below.

Sample time lines

Given a high-level knowledge of the order of API functions required and the order in which the client calls your callback, here are some sample time lines to illustrate the complete order of messages.

Simple interaction time line

[Figure 4-1](#) shows a simple interaction where a speak request proceeds from start to finish with no other messages such as errors or stop requests.

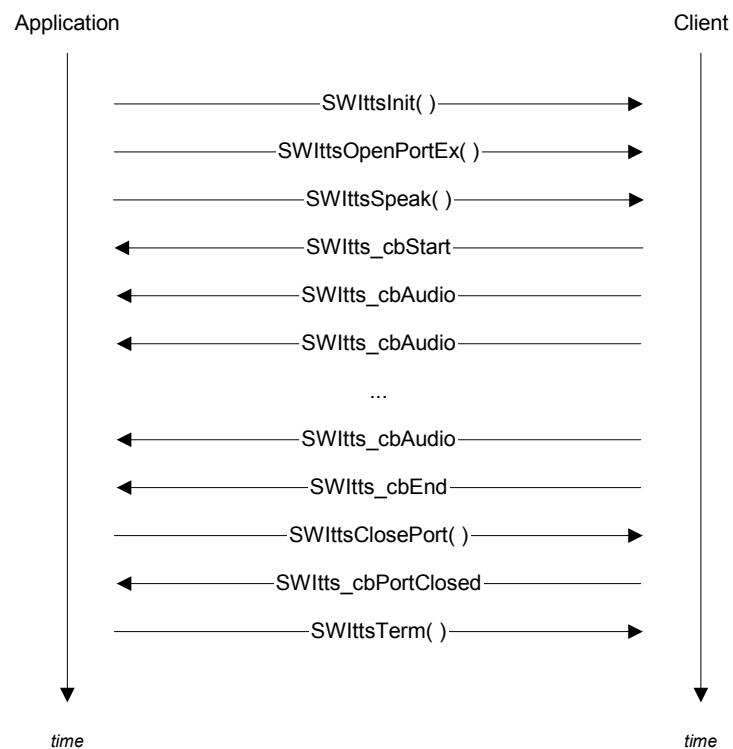


Figure 4-1 Speak request from start to finish

Time line with bookmarks

Figure 4-2 illustrates the application/client communication when bookmarks have been embedded in the input text and therefore bookmark messages are being returned with audio messages.

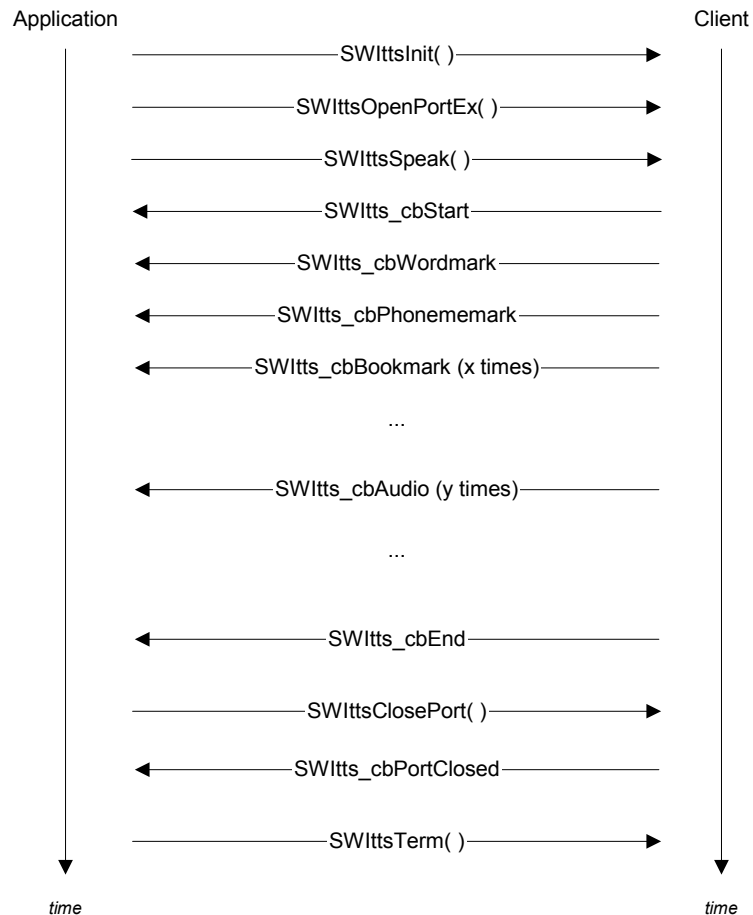


Figure 4-2 Application/client communication with bookmarks

For example, if your input text is “This is \bmbbookmark1 a test.” you may expect the callback would receive audio messages containing samples for the words “This is,” then a bookmark message, then audio messages for the words “a test.” Instead, Speechify only guarantees that a bookmark message is sent before any audio messages that contain the bookmark’s corresponding audio sample (indicated by the bookmark’s timestamp). That is, a bookmark never arrives “late” but it may and usually does arrive “early.”

Speechify server calculates and sends bookmark locations every time it sees the beginning of a “phrase.” A phrase can be loosely understood as some period of speech that is followed by a pause. Phrases are often delimited by the punctuation marks such as periods, commas, colons, semicolons, exclamation points, question marks, and parentheses, but there is not an exact correspondence between punctuation and phrase boundaries.

For simple input texts you can figure out when bookmarks are sent to your callback. For example, the text “This is a \bmbookmark1 test,” is a single phrase, so the bookmark message arrives before any of the audio messages. The time line shown in [Figure 4-2](#) illustrates that scenario. If your text is “Hello, this is a \bmbookmark1 test.” then your callback receives the audio messages for the phrase “Hello,” a bookmark message, then the audio messages for the phrase “this is a test.”



NOTE

Any mark messages you use (e.g., bookmark, wordmark, phoneme mark) are always sent to the callback function before the corresponding audio messages.

For more complex input text, it may be difficult to predict when your bookmark messages will be returned, since there is not an exact correlation between punctuation and phrase boundaries. Speechify's standard text processing inserts phrase boundaries at locations where there is no punctuation, in order to enhance understandability. Conversely, some punctuation marks do not correspond to a phrase boundary.

Time line with SWIttsStop()

Figure 4-3 illustrates the communication between the application and the client when the `SWIttsStop()` function is called. Note that the `SWItts_cbEnd` message is not generated when this function is called. Instead you receive the `SWItts_cbStopped` message. You always receive either `SWItts_cbEnd` or `SWItts_cbStopped`, but never both.

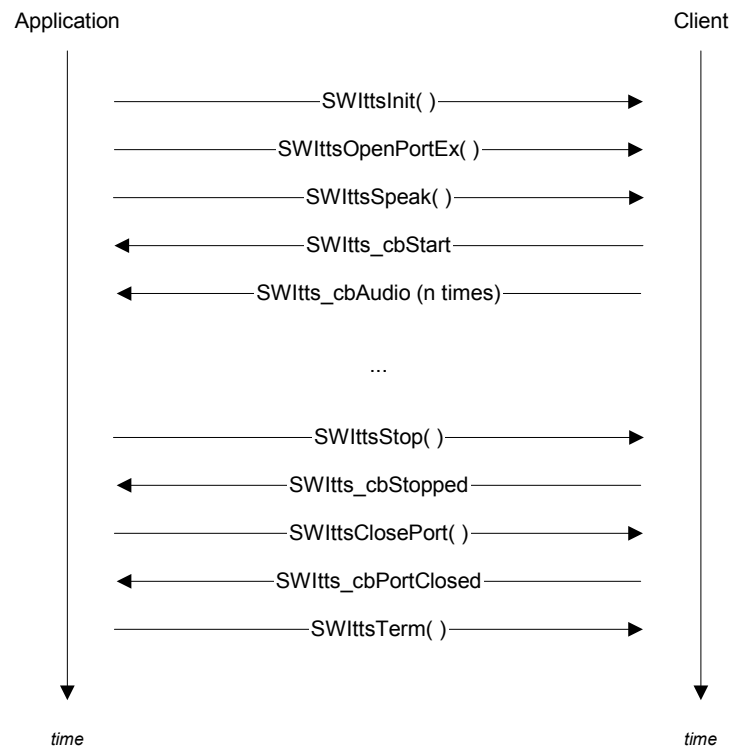


Figure 4-3 `SWIttsStop()` function

Overview of user dictionaries

Speechify supports user dictionaries for customizing the pronunciations of words, proper names, abbreviations, acronyms, and other sequences. These dictionaries are in XML format and are loaded from a URI or from memory. Multiple dictionaries can be loaded and activated for prioritized lookup. You can add words to Speechify's vocabulary by creating and loading XML dictionaries. See [Chapter 9](#) for details.

[Chapter 5](#) describes how Speechify user dictionaries are loaded, activated, and formatted. For detailed discussion of how to use Speechify's user dictionary facilities to customize pronunciations, see [Chapter 9](#). For more information on the dictionary functions, see [Chapter 12](#).

Parameter configuration

The Speechify server ships with a default, system-wide configuration file that defines basics such as the Internet fetch proxy server and the default audio format. The server can be configured via one or more configuration files, so that you can create voice-specific configuration files to override the default configuration file and set the voice name, sampling rate, and other parameters for your application.

[Chapter 13](#) describes all of the Speechify server configuration parameters. These are configured in an XML configuration file. Some can also be changed at run time for individual port resources via `SWIttsSetParameter()`, while others can be read via `SWIttsGetParameter()` for informational purposes but cannot be changed.

Both `SWIttsSetParameter()` and `SWIttsGetParameter()` are synchronous calls that cannot be made while synthesis is taking place.

For the list of parameters that can be set via the API, see “`SWIttsSetParameter()`” on [page 159](#). For the list of all parameters, see [Chapter 13](#).

Character set support

Speechify supports strings in multiple character sets as input to `SWIttsSpeak()`, `SWIttsSpeakEx()`, and `SWIttsDictionaryLoad()`. For a list of supported character sets, see “`SWIttsSpeak()`” on [page 161](#).

Using W3C SSML

W3C Speech Synthesis Markup Language (W3C SSML) provides a rich, XML-based standardized markup language for assisting in the generation of synthetic speech. It provides standardized XML elements for controlling pronunciation, volume, rate, and other output characteristics across synthesis-capable platforms. Speechify implements the December 2, 2002 Last Call Working Draft of W3C SSML. You can obtain that draft specification from <http://www.w3.org/TR/2002/WD-speech-synthesis-20021202>.

The following types of users should consider SSML:

- ❑ VoiceXML platform providers: W3C VoiceXML uses W3C SSML markup for all prompting. VoiceXML platform providers can thus easily extract all VoiceXML prompt instructions into a W3C SSML document and delegate that W3C SSML document to Speechify for playback. Speechify can even handle VoiceXML `<audio>` elements, seamlessly inserting them into the synthesis stream.
- ❑ Users who are speaking static prompts with embedded variables that are dates, times, ordinal numbers, or another type that matches a Speechify supported W3C SSML `<say-as>` element type: `<say-as>` provides a way to explicitly label the input data format to ensure proper interpretation by Speechify, such as `<say-as interpret-as="date" format="mdy">` for a date in MM/DD/YYYY or MMDDYYYY format. This can be very useful for ensuring proper speech output when inserting variable data from a recognizer or database into static prompts. This capability is not available when using native Speechify front-end tags.
- ❑ Any user who strongly prefers public standards based technologies

To use W3C SSML with Speechify, create your text-to-speech input as an XML document that conforms to the W3C SSML specification, then direct Speechify to speak it using the `application/synthesis+ssml` MIME content type. Speechify then

parses and speaks this XML document. For more information, see “W3C SSML Compliance” on [page 99](#), “SWIttsSpeak()” on [page 161](#), and “SWIttsSpeakEx()” on [page 164](#).

Using SAPI

SAPI is the Microsoft Speech API, a standard set of API functions for applications and speech engine providers to use that promotes engine and application independence. Speechify includes a SAPI 5 interface so that applications written to use SAPI 5 engines can easily incorporate Speechify. This guide does not include general programming information regarding SAPI. Users should refer to Microsoft's Speech SDK for API documentation, sample applications, and programming guidelines. As of this writing, the SDK is available at <http://www.microsoft.com/speech>.

Speechify supports most of the features in the SAPI 5 specification. For a detailed list of compliance with the SAPI 5 standard, see “SAPI 5” on [page 189](#).

Implementation guidelines

To support multiple simultaneous TTS ports, you can employ a multi-threaded process model that opens multiple ports in one process or a multi-process model that opens one port per process.

The current implementation of the client spawns a thread for each open port. This thread is created in the `SWIttsOpenPortEx()` function and destroyed in the `SWIttsClosePort()` function, no matter which model you choose for your application code. The spawned threads receive the asynchronous network events for that port, and then call your callback function. You must be efficient in your callback code; blocking or waiting for an event may cause networking errors.

The threads created by `SWIttsOpenPortEx()` have their stack sizes set to reasonable default values, which typically do not need to be changed. However, you can override the default by setting the `SWITTSTHREADSTACKSIZE` environment variable to a desired value (in bytes).

Opening and closing a TTS port for every speak request is costly. The act of opening and closing a port requires some processing and resource management overhead on the server, as does creating or tearing down the networking sockets on both the client and server. Thus, we recommend that you open a port for the duration of a “session” (defined by you). This session could be for the length of a phone call or indefinitely.



Using User Dictionaries

Speechify supports user dictionaries for customizing the pronunciations of words, proper names, abbreviations, acronyms, and other sequences. For example, if you want the sequence *SWI* to be pronounced *Speech Works* every time it occurs, you can create a dictionary entry specifying this pronunciation.

This chapter describes how Speechify user dictionaries are loaded, activated, and formatted. For detailed discussion of how to use Speechify's user dictionary facilities to customize pronunciations, see [Chapter 9](#).

In This Chapter

- ❑ “Overview of dictionaries” on [page 61](#)
- ❑ “Loading and activating dictionaries” on [page 62](#)
- ❑ “Dictionary file format” on [page 65](#)

Overview of dictionaries

A dictionary entry consists of a *key* and a *translation value*. When the key matches a string in the input text, the translation value replaces it, and the translation is then pronounced according to Speechify's internal pronunciation rules.

For most languages, Speechify provides three types of user dictionaries:

- ❑ “[Main dictionary](#)”. The main dictionary is an all-purpose exception dictionary that can be used to replace a single token in an input text with almost any other valid input sequence.

- ❑ “[Abbreviation dictionary](#)”. The abbreviation dictionary is used to expand abbreviations, and provides for special treatment of period-delimited tokens.
- ❑ “[Root dictionary](#)”. The root dictionary is used to specify pronunciations, orthographically or via SPRs, of words or morphological word roots.

In Japanese, Speechify uses a single user dictionary type (“Mainext”), described in the *Speechify Language Supplement* for ja-JP.

Loading and activating dictionaries

When you *load* a dictionary, the contents of the dictionary are read into memory, but the dictionary is not used unless it is *activated*. When a dictionary is activated, it is assigned a *priority*. If you have activated more than one dictionary of a given type, entries in higher priority dictionaries take precedence over entries in lower priority dictionaries of the same type.

This section describes the three ways to load and activate dictionaries:

- ❑ “Loading and activating dictionaries from a configuration file” on [page 62](#)
- ❑ “Loading, activating, and deactivating dictionaries using API calls” on [page 63](#)
- ❑ “Loading and activating dictionaries using embedded tags” on [page 64](#)

Loading and activating dictionaries from a configuration file

Within configuration files, dictionaries are specified as URIs. When `SWIttsOpenPortEx()` is called, each dictionary specified by a URI in the `tts.engine.dictionaries` parameter in a configuration file is loaded and activated.

You can specify the dictionary priority using the `name` attribute. If you do not specify a priority, a default priority is assigned. See “`tts.engine.dictionaries`” on [page 177](#) and “`SWIttsDictionaryActivate()`” on [page 144](#).

Since dictionaries are typically language-specific, the `tts.engine.dictionaries` parameter normally appears within a non-default language element, or in a voice configuration file (if you want the dictionary to be activated only for a specific voice). For example:

```
<lang name="en-US">

  <!-- *** ENGINE PARAMETERS *** -->

  <!-- Dictionary -->
  <param name="tts.engine.dictionaries">
    <namedValue name="1"> http://myserver/mydict1.xml
    </namedValue>
    <namedValue name="2"> file:/users/mydict2.xml
    </namedValue>
  </param>
</lang>
```

You can load more than one dictionary in the `namedValue` list. All dictionaries in a `namedValue` list that reside in a `<lang>` block that matches the server's are loaded and activated. Loading and activating a dictionary whose language doesn't match that of the server is essentially equivalent to loading and activating a dictionary with no entries.

If loading a dictionary fails because the URI cannot be found, a warning is logged to the error log, and the server continues to initialize. If loading a dictionary fails for any other reason, an error is logged and `SWIttsOpenPortEx()` returns an error.

A dictionary activated in the configuration file cannot be deactivated while Speechify is running.

Loading, activating, and deactivating dictionaries using API calls

Once the server has initialized, you can load dictionaries with `SWIttsDictionaryLoad()`. Each dictionary is specified as either a URI or an in-memory buffer. After a dictionary has been loaded, it can be activated using `SWIttsDictionaryActivate()`. `SWIttsDictionaryActivate()` requires that you explicitly set the priority of each dictionary relative to other active dictionaries of the same type.

`SWIttsDictionariesDeactivate()` deactivates all dictionaries that have been activated by a call to `SWIttsDictionaryActivate()`. It does not deactivate dictionaries activated through a configuration file specification.

Loading and activating dictionaries using embedded tags

You can load and activate a dictionary using an embedded tag in your input text. In this case, the dictionary is activated for the entire speak request, and is deactivated and unloaded after the speak request. The tag specifies the location of the dictionary as a URI, and optionally specifies a priority. If no priority is specified, a default priority is assigned. See “Specifying user dictionaries” on [page 75](#) for details and examples.

Invalid dictionary entries

Valid dictionary keys and translations are described in [Chapter 9](#) and in the user dictionary chapters of the language supplements. If a dictionary entry contains an invalid key or translation, an error is logged during loading, and the loading aborts with an error.

Here is an example of an entry in the error log:

```
Jul 24 12:04:54.96|2256||0|WARNING|6313|Speechify server:
Dictionary key is invalid|type=main|key=cat.|
reason=Final' 'punctuation' 'not' 'allowed
```

This log entry includes both the invalid dictionary entry's key or translation value and the reason that it is invalid. The list of reasons includes:

- ❑ Final punctuation not allowed
- ❑ Invalid nonalphabetic character
- ❑ No uppercase allowed
- ❑ Only single word allowed
- ❑ Out-of-range character
- ❑ SPR Error: Invalid bracketing
- ❑ SPR Error: Invalid phone
- ❑ SPR Error: Invalid stress placement
- ❑ SPR Error: Invalid stress value
- ❑ SPR Error: Word or syllable with no vowel
- ❑ Word contains no vowel

Dictionary file format

A dictionary is XML (either a file or an array of bytes) that lists keys and translations. Here is a sample user dictionary file, followed by a description of the elements and attributes:

```
<?xml version="1.0" ?>
<!DOCTYPE lexicon PUBLIC "-//SpeechWorks//DTD LEXICON 1.0//
  EN" "lexicon.dtd">
<lexicon xml:lang="en-US" type="main" alphabet="text">
  <entry key="Win2K">
    <definition value="windows two thousand"/>
  </entry>
  <entry key="TTS">
    <definition value="text to speech"/>
  </entry>
</lexicon>
```

The sample dictionary consists of the following parts:

❑ XML declaration:

```
<?xml version="1.0" ?>
```

❑ Document type declaration:

```
<!DOCTYPE lexicon PUBLIC "-//SpeechWorks//DTD LEXICON 1.0/
/EN" "lexicon.dtd">
```

❑ Body (a <lexicon> element):

```
<lexicon xml:lang="en-US" type="main" alphabet="text">
  <entry key="Win2K">
    <definition value="windows two thousand"/>
  </entry>
  <entry key="TTS">
    <definition value="text to speech"/>
  </entry>
</lexicon>
```

The <lexicon> element

The <lexicon> element is the root element of the document, and is the container element for <entry> elements, which in turn contain <definition> elements defining translations.

The <lexicon> element has one required attribute, `xml:lang`, and two optional attributes, `type` and `alphabet`. The `xml:lang` and `alphabet` attributes can be overridden by attributes of <entry> and <definition> elements, while the `type` attribute applies only to the <lexicon> element.

The `xml:lang` attribute

The `xml:lang` attribute specifies the language for which the dictionary can be used. Because the language attribute sets the default language for all vocabulary items in that <lexicon>, a typical application creates separate files for each target language. However, an application can mix languages in a dictionary, which is particularly useful when supporting different dialects of a single language. For example, an application may have a <lexicon> element whose language is “en” and have individual <entry> or <definition> elements specifying languages “en-GB” and “en-US.” The result is a single lexicon that supports both en-GB and en-US.

The `type` attribute

The `type` attribute specifies the dictionary type. In all languages other than Japanese, the valid types are `main`, `abbreviation`, and `root`. If the `type` attribute is not specified, it defaults to `main`. In Japanese, there is only one valid type, `mainext`, which is also the default type.

The `alphabet` attribute

The `alphabet` attribute specifies the alphabet of the dictionary translations. Valid values are:

- ❑ `text`: the translation is in plain text. This is the only valid value for the abbreviation dictionary, and can also be used in main and root dictionaries.
- ❑ `SWItts`: for translation values consisting of an SPR tag.
- ❑ `OSR`: If the `alphabet` attribute of a vocabulary item is not specified, it defaults to `OSR`. Speechify does not load any elements with an `OSR` alphabet.

Note that `alphabet` is not required for DTD validation, but since Speechify ignores items with the `OSR` alphabet value, you should make sure each of your Speechify entries has `alphabet` set to `SWItts` or `text`.

The <entry> element

Each <lexicon> element contains one or more <entry> elements.

Each <entry> element must specify a key attribute value. When the key matches a string in the input text, the translation specified in the <definition> element is substituted for it.

The <entry> element may explicitly specify values for the xml:lang and alphabet attributes; if it does not, it inherits the values from the parent <lexicon> element.

The <definition> element

Each <definition> element specifies the translation value of the key.

The <definition> element may explicitly specify values for the xml:lang and alphabet attributes; if it does not, it inherits the values from the enclosing <entry> or <lexicon> elements.

If the alphabet value of a definition is SWItts, you must use the phoneme set of the language being spoken.

If an <entry> element contains multiple <definition> elements, Speechify uses the one whose xml:lang value best matches the server. For example, if the language of the server is en-US, a <definition> element with an xml:lang attribute of "en-US" is the best match; an xml:lang attribute of "en" is the next best match. No other xml:lang values match. If there are two <definition> elements with the same xml:lang attribute value, the first one is used.

Inheriting and overriding attributes

The xml:lang and alphabet attributes can modify the <lexicon>, <entry>, or <definition> elements. If a child element does not explicitly stipulate an attribute value, it inherits the value of the parent element. Otherwise, the attribute value of the child element overrides the attribute value of the parent element.

This behavior can be exploited to create shared dictionaries containing entries or definitions that apply only to specific dialects of a language. For example, the following dictionary can be used by either en-US or en-GB, but the translations of the key are different in the two dialects.

```
<?xml version="1.0" ?>
<!DOCTYPE lexicon PUBLIC "-//SpeechWorks//DTD
    LEXICON 1.0//EN" "lexicon.dtd">
<lexicon xml:lang="en" type="abbreviation" alphabet="text">
  <entry key="p">
    <definition value="page" xml:lang="en-US"/>
    <definition value="pence" xml:lang="en-GB"/>
  </entry>
</lexicon>
```

When you override the default with this technique, remember that if the alphabet is SWItts, you must define the translation value with phonemes from the target language. For example:

```
<?xml version="1.0" ?>
<!DOCTYPE lexicon PUBLIC "-//SpeechWorks//DTD
    LEXICON 1.0//EN" "lexicon.dtd">
<lexicon xml:lang="en" type="root" alphabet="SWItts">
  <entry key="thorough">
    <definition value="1TR2o" xml:lang="en-US"/>
    <definition value="1THrx" xml:lang="en-GB"/>
  </entry>
</lexicon>
```

You can also make use of attribute inheritance to create dictionary entries that use different alphabets. For example, a root dictionary might specify some translations using the text alphabet, and others using the SWItts alphabet. The attribute value can be specified either for the <entry> element or for the <definition> element. In the following example, the <lexicon> alphabet value specifies a default which can be overridden by individual <entry> or <definition> elements.

```
<?xml version="1.0" ?>
<!DOCTYPE lexicon PUBLIC "-//SpeechWorks//DTD
    LEXICON 1.0//EN" "lexicon.dtd">
<lexicon xml:lang="en-US" type="root" alphabet="SWItts">
  <entry key="tomato">
    <definition value="tx1maFo"/>
  </entry>
  <entry key="potato" alphabet="text">
    <definition value="potahto"/>
  </entry>
  <entry key="pajamas">
    <definition value="pajahmas" alphabet="text"/>
  </entry>
</lexicon>
```



Text-to-Speech Processing

The chapters in this part cover these topics:

[Chapter 6 “Standard Text Normalization”](#) describes how Speechify processes input text before synthesizing the output speech.

[Chapter 2 “Embedded Tags”](#) describes tags that users can insert into input text to customize the speech output in a variety of ways.

[Chapter 7 “Symbolic Phonetic Representations”](#) describes the phonetic spelling the user can enter to explicitly specify word pronunciations.

[Chapter 8 “W3C SSML Compliance”](#) describes Speechify’s compliance with the W3C SSML specification.

[Chapter 9 “User Dictionaries”](#) describes the user dictionaries available for customizing the pronunciations of words, abbreviations, acronyms, and other sequences.

[Chapter 10 “Improving Speech Quality”](#) provides an overview of factors that contribute to output speech quality along with troubleshooting tips for improving the sound.



Embedded Tags

Embedded tags are special codes that can be inserted into input text to customize Speechify's behavior in a variety of ways. You can use embedded tags to:

- ❑ Create pauses at specified points in the speech output.
- ❑ Indicate the end of a sentence.
- ❑ Customize word pronunciations using phonetic spelling.
- ❑ Load and activate user dictionaries.
- ❑ Spell out sequences of characters by name.
- ❑ Specify the interpretation of certain numeric expressions.
- ❑ Insert bookmarks in the text so your application receives notification when a specific point in the synthesis job has been reached.
- ❑ Control the volume and speaking rate of the speech output.

In This Chapter

- ❑ “Using Speechify tags” on [page 72](#)
- ❑ “Creating pauses” on [page 72](#)
- ❑ “Indicating the end of a sentence” on [page 73](#)
- ❑ “Customizing pronunciations” on [page 74](#)
- ❑ “Inserting bookmarks” on [page 79](#)
- ❑ “Controlling the audio characteristics” on [page 80](#)

Using Speechify tags

A Speechify tag begins with the sequence \! followed immediately by a string of alphanumeric characters. For example:

\!p300	Synthesize a pause of 300 ms.
\!tsc	Pronounce all characters individually by name.

Separate a tag from any preceding input by at least one unit of white space. A tag cannot be followed immediately by an alphanumeric character, though most tags may be followed immediately by punctuation, parentheses, and similar symbols. (The bookmark tag is an exception, since it must end in white space; see “Inserting bookmarks” on [page 79](#).)

Any sequence of non-white-space characters beginning with the prefix \! that is not a recognizable Speechify tag is ignored in the speech output.

A Speechify tag setting is valid for the duration of the speak request. For example, unless spellout mode (see “Character spellout modes” on [page 76](#)) is explicitly reset during the speak request with another tag, its setting returns to default when the speak request is finished (either by completing normally or when stopped in the middle).

Creating pauses

Use a pause tag to create a pause of a particular duration at a specified point in the speech output.

Pause tag	Description
\!pN	Create a pause N milliseconds long. Valid range: 1–32767

To create a pause longer than 32,767 ms, use a series of consecutive pause tags.

The behavior produced by the pause tag varies depending on its location in the text. When a pause tag is placed immediately before a punctuation mark, the standard pause duration triggered by the punctuation is replaced by the pause duration specified in the tag. In other locations, the tag creates an additional pause.

For example, sentence (a) has a default 150 ms pause at the comma. Sentences (b) and (c) replace the default pause with a longer and shorter pause, respectively, while sentence (d) inserts an additional pause of 300 ms, resulting in a total pause duration of 450 ms. In sentence (e) a 25 ms pause is inserted in a location where no pause would otherwise occur.

Input	Pronunciation
(a) Tom is a good swimmer, because he took lessons as a child.	Tom is a good swimmer (150 ms pause), because he took lessons as a child.
(b) Tom is a good swimmer \p300, because he took lessons as a child.	Tom is a good swimmer (300 ms pause), because he took lessons as a child.
(c) Tom is a good swimmer \p100, because he took lessons as a child.	Tom is a good swimmer (100 ms pause), because he took lessons as a child.
(d) Tom is a good swimmer, \p300 because he took lessons as a child.	Tom is a good swimmer (150 ms pause), (300 ms pause) because he took lessons as a child.
(e) Tom is a good swimmer, because he took lessons \p25 as a child.	Tom is a good swimmer (150 ms pause), because he took lessons (25 ms pause) as a child.

Indicating the end of a sentence

Use an end-of-sentence tag to trigger an end of sentence at the preceding word, whether or not that word is followed by punctuation.

End of sentence tag	Description
---------------------	-------------

\leos	Treat the preceding word as the end of sentence.
-------	--

This tag is useful for forcing a sentence end in unpunctuated text, such as lists, or after a period-final abbreviation when the context does not unambiguously determine a sentence end. Here are some examples. (Periods in the Pronunciation column indicate sentence ends.)

Input	Pronunciation
apples pears bananas	apples pears bananas
apples \leos pears \leos bananas \leos	apples. pears. bananas.

Input	Pronunciation
The Dow industrials were down 15.50 at 4,585.90 at 1:59 p.m. NYSE decliners led advancers...	the dow industrials were down fifteen point five zero at four thousand five hundred eighty-five point nine zero at one fifty-nine pea emm new york stock exchange decliners led advancers...
The Dow industrials were down 15.50 at 4,585.90 at 1:59 p.m. \leos NYSE decliners led advancers...	the dow industrials were down fifteen point five zero at four thousand five hundred eighty-five at one fifty-nine pea emm. new york stock exchange decliners led advancers...

Customizing pronunciations

In certain cases, you may want to specify a pronunciation that differs from the one generated by Speechify's internal text analysis rules. The tags described in this section are used to modify Speechify's default text processing behavior in a variety of ways:

- ❑ "Customizing word pronunciations" on [page 74](#)
- ❑ "Specifying user dictionaries" on [page 75](#)
- ❑ "Language-specific customizations" on [page 78](#)
- ❑ "Character spellout modes" on [page 76](#)
- ❑ "Pronouncing numbers and years" on [page 77](#)

Customizing word pronunciations

You can use a Symbolic Phonetic Representation (SPR) to specify the exact pronunciation of a word using Speechify's special phonetic symbols. The SPR tag takes the following form:

SPR tag	Description
\[SPR]	Pronounce the word phonetically as specified inside the square brackets.



NOTE

Unlike other tags, the `\![SPR]` tag does not modify the pronunciation of *following* text. Instead, it is used *in place of* the word for which it specifies a pronunciation.

“Symbolic Phonetic Representations” on [page 95](#) provides detailed information on formatting SPRs.

Specifying user dictionaries

Use the dictionary tag to apply dictionaries for the current speak request. These dictionaries are deactivated and unloaded after the speak request. Like the W3C SSML `<lexicon>` element, this embedded tag must occur before all other markup and non-white-space text contained within the request. It applies to the entire speak request, but not subsequent speak requests. There is no limit on the number of these tags (and dictionaries) you use within a single request.

Dictionary tag	Description
<code>\ldi<priority>[<uri>]</code>	<p><code><priority></code> is an optional integer greater than 0. See “SWIttsDictionaryActivate()” on page 144 for details about the dictionary priority. See “tts.engine.dictionaryDefaultPriorityBase” on page 177 for details about the default priority.</p> <p><code><uri></code> is the URI to the dictionary. See “SWIttsDictionaryLoad()” on page 148 for more information about supported URI formats.</p>

Examples:

```
\!di[http://myserver/dict1.xml] Hello there  
  
\!di128[http://myserver/dict1.xml] Hello there  
  
\!di128[c:\Program Files\SpeechWorks\Speechify\MyDict1.xml]  
\!di129[c:\Program Files\SpeechWorks\Speechify\MyDict2.xml]  
This is my text  
  
\!di[c:\Program Files\SpeechWorks\Speechify\MyDict1.xml]  
\!di[c:\Program Files\SpeechWorks\Speechify\MyDict2.xml]  
This is my text
```

For more information about dictionaries, see [Chapter 9](#). For more information about dictionary priorities, see “SWIttsDictionaryActivate()” on [page 144](#).

Character spellout modes

The following tags are used to trigger character-by-character spellout of subsequent text.

Spellout mode tag	Description
\!ts0	Default mode.
\!tsc	All-character spellout mode: pronounce all characters individually by name.
\!tsa	Alphanumeric spellout mode: pronounce only alphanumeric characters by name.
\!tsr	Radio spellout mode: like alphanumeric mode, but alphabetic characters are spelled out according to the International Radio Alphabet. Supported in English only.

For example:

Input	Pronunciation
My account number is \!tsa 487-B12.	My account number is four eight seven bee one two.
My account number is \!tsc 487-B12.	My account number is four eight seven dash bee one two.
The last name is spelled \!tsr Dvorak.	The last name is spelled delta victor oscar romeo alpha kilo.

Spellout modes remain in effect throughout the speak request, unless they are turned off by the tag `\!ts0`, which restores the engine to its default processing mode. For example:

Input	Pronunciation
The composer's name is spelled <code>\!tsa</code> Franz Liszt <code>\!ts0</code> and pronounced "Franz Liszt."	The composer's name is spelled eff ar ay en zee ell aye ess zee tee and pronounced Franz Liszt.

There are many words which are spelled out as a result of Speechify's default text processing behavior. In such cases, the use of a spellout mode tag may have no additional effect. For example:

Input	Pronunciation
He works for either the CIA or the FBI.	He works for either the cee aye ay or the eff bee aye.
He works for either the <code>\!tsa</code> CIA <code>\!ts0</code> or the <code>\!tsa</code> FBI <code>\!ts0</code> .	He works for either the cee aye ay or the eff bee aye.

In alphanumeric and radio spellout modes, punctuation is interpreted exactly as it is in the default (non-spellout) mode; i.e., in most contexts it triggers a phrase break. In all-character spellout mode, punctuation is spelled out like any other character, rather than being interpreted as punctuation. Speech output continues without pause until the mode is turned off. For example:

Input	Pronunciation
<code>\!tsa</code> 3, 2, 1 <code>\!ts0</code> blastoff.	three, two, one, blastoff
<code>\!tsc</code> 3, 2, 1 <code>\!ts0</code> , blastoff.	three comma two comma one, blastoff

Pronouncing numbers and years

In some languages (for example, English and German), a four digit numeric sequence with no internal commas or trailing decimal digits, like *1984*, can be interpreted either as a year (*nineteen eighty four*) or as a quantity (*one thousand nine hundred eighty four*). Speechify applies the year interpretation by default, as in:

Input	Pronunciation
He was born in May 1945.	He was born in May nineteen forty five.

To override or restore the default year interpretation, use the following tags:

Year mode tag	Description
\ny0	Quantity interpretation.
\ny1	Year interpretation (default).

For example:

Input	Pronunciation
In May \ny0 1945 people emigrated.	In May one thousand nine hundred forty five people emigrated.

Each tag remains in effect throughout the speak request, unless the interpretation is toggled by the use of the other tag. For example:

Input	Pronunciation
\ny0 1945 \ny1 people emigrated in 1945.	One thousand nine hundred forty five people emigrated in nineteen forty five.



NOTE

These tags only operate in English and German, where there is a difference in pronunciation between the year and quantity interpretations.

Language-specific customizations

Some tags are supported only in specific languages, in some cases because the tags solve problems specific to those languages. For example:

- ❑ Mexican Spanish (es-MX) has tags for pronouncing the dollar sign (\$) as “pesos” or “dollars” depending on the needs of the application.
- ❑ All varieties of English (en-US, en-GB, and en-AU) support a tag for specifying the order of elements in a date.

- ❑ All varieties of English (en-US, en-GB, and en-AU) support a tag for specifying the locale of the document's origin.
- ❑ United States English (en-US), French French (fr-FR), and German (de-DE) have a tag for pronouncing postal addresses.

Such tags are described per language in the corresponding *Speechify Language Supplement*.

Inserting bookmarks

A bookmark tag inserted into the text triggers notification when that point in the synthesis job is reached. Bookmarks are useful for timing other events with the speech output. For example, you may want to insert additional audio between specific points in the speech data, or log that the user heard a certain portion of the text before you stop the speech. Bookmark tags have the following format:

Bookmark tag	Description
<code>\bmstr</code>	Insert a user bookmark with identifier str , which is any string ending in white space and containing no internal white space.

When you insert a bookmark, the client sends a data structure to your callback function. This data structure contains a bookmark identifier and a timestamp indicating where in the audio data the bookmark occurs. (See “SWIttsCallback()” on [page 137](#).) The timestamps in the bookmark structures ensure that the bookmarks are resolved to the correct point in the audio data. For example:

Input	Output
The clarinet \bmCL came in just before the first violins. \bmVL1	The clarinet came in just before the first violins.

Bookmark \bmCL triggers a bookmark data structure containing the identifier CL and the timestamp of the sample just after *clarinet*. Bookmark \bmVL1 triggers a data structure containing the identifier VL1, and the timestamp at the end of the audio data for this sentence.

See the sample “Time line with bookmarks” on [page 53](#) and its description for more information about bookmark notifications.



NOTE

The bookmark identifier is a string of any non-white-space characters, and it must be delimited by final white space. It is returned as a 0-terminated wide character string. (See “SWIttsCallback()” on [page 137](#).)

Controlling the audio characteristics

Each voice has a default volume and speaking rate, known as the baseline (default) values. An application can adjust these values for each open port individually. You can also use tags to adjust these settings within the text. Thus, there are three possible settings for rate and volume, in order of precedence:

1. Embedded tags – set relative to port-specific or baseline default, described below. Separate embedded tags control volume and speaking rate.
2. Port-specific settings – use `SWIttsSetParameter()` to set `tts.audio.volume` and `tts.audio.rate` as percentages of the baseline values, or use the XML configuration file to change the initial volume and rate port setting for new ports. These port-specific values are used if there are no overriding embedded tags.
3. Baseline – the default values for the voice when you install it; you cannot change these values, only override them.

For example, you can set a port-specific value for volume if the baseline value is too loud for your application. Set `tts.audio.volume` to a value less than 100 (percent) to make the volume quieter than the default. To adjust the volume within the text, use the embedded tags.

There are embedded tags to adjust volume and rate relative to the port-specific values and relative to the baseline values. For example, if you want a particular word or phrase to be spoken more quickly than the rest of the text, you can use the embedded tag to change the rate relative to the port-specific rate. If you want a particular word or phrase spoken at the baseline volume, no matter what volume the rest of the text is, use an embedded tag to set the volume to 100 percent of the baseline volume.

Setting a parameter relative to the port-specific value is useful if the port-specific value is unknown at the time that the source text is composed. For example, setting the volume of a word or phrase to 50 percent of the port-specific value guarantees

that the corresponding audio is half as loud as the surrounding text. Conversely, setting the volume to 50 percent of the baseline default has the same effect only if the port-specific value is set to 100; if the port-specific value is already 50 percent of baseline, there is no difference for the tagged piece of text.

Volume control

The server produces audio output at the default volume, which is the maximum amplitude that may be obtained for all utterances without causing distortion (clipping). This setting minimizes quantization noise in the digital waveform that is delivered to the application by using all of the available numerical range. The application can override this on any given port by using the `SWIttsSetParameter()` function to set the port-specific volume (`tts.audio.volume`) to a lower value.

The port-specific volume applies to all text spoken on that port, i.e., in that conversation. To adjust the volume for a particular word or phrase, use these tags to vary volume:

Volume control tags	Description
<code>\vpN</code>	Set volume to the given percentage of the port-specific value. (<i>N</i> is an integer greater or equal to 0.)
<code>\vpr</code>	Reset volume to the port-specific value.
<code>\vdN</code>	Set volume to the given percentage of the baseline default value. (<i>N</i> is an integer greater or equal to 0.)
<code>\vdr</code>	Reset volume to the server default value.

Note that the tags allow the effective volume to be set above 100, for example, `\vd120`. The server attempts to scale its audio output accordingly, and may produce clipped and distorted output.

These examples assume port-specific volume has been set to 50 with `SWIttsSetParameter()` prior to each `SWIttsSpeak()` call:

Input	Result
The flight departs at <code>\vp150</code> seven-fifty <code>\vpr</code> in the evening.	[volume = 50] The flight departs at [volume = 75] seven-fifty [volume = 50] in the evening.
Severe <code>\vd60</code> storm warning <code>\vd50</code> has been issued for <code>\vd80</code> Suffolk and Nassau <code>\vpr</code> counties.	[volume = 50] Severe [volume = 60] storm warning [volume = 50] has been issued for [volume = 80] Suffolk and Nassau [volume = 50] counties.

Speaking rate control

The baseline speaking rate of the synthesized speech is chosen to result in the highest audio quality. Departures from this rate are generally detrimental to audio quality and should be employed with care. Extreme variations may result in unintelligible or unnatural-sounding speech. To improve the match between the desired and obtained rates, include at least a few words in the scope of a rate change tag.

Server performance generally decreases when rate change is requested, because increased computation is necessary to implement rate variation. The exact performance impact depends on the given text and the magnitude of the requested rate change.

Note that changes in speaking rate do not translate precisely into duration changes. So an utterance spoken at half the default rate is not exactly twice as long as the same utterance spoken at the default rate, although it is close. Also, rate changes affect the duration of any pauses in the output – including pauses that are specified explicitly with a `\!p` tag.

The following tags control rate changes within an utterance:

Speaking rate control tags	Description
<code>\lrpN</code>	Set rate to the given percentage of the port-specific value. (N is an integer greater than 0.)
<code>\lrpr</code>	Reset rate to the port-specific value.
<code>\lrdN</code>	Set rate to the given percentage of the server default value. (N is an integer from 33 to 300.)
<code>\lrdr</code>	Reset rate to the server default value.

The rate cannot be set to less than one-third or more than 3 times the server default. Attempting to do so results in a value at the corresponding minimum or maximum.

These examples assume port-specific rate has been set to 200 with `SWIttsSetParameter()` prior to each `SWIttsSpeak()` call:

Input	Result
The flight departs at <code>\lrp50</code> seven-fifty <code>\lrpr</code> in the evening.	[rate = twice default] The flight departs at [rate = default] seven-fifty [rate = twice default] in the evening.
<code>\lrd150</code> The share price of Acme Corporation stock is <code>\lrd120</code> \$7.00	[rate = twice default] [rate = 1.5 times default] The share price of Acme Corporation stock is [rate = 1.2 times default] seven dollars.



Standard Text Normalization

Speechify's *text normalization* interprets the input text and converts it into a sequence of fully spelled-out words. This process is primarily responsible for an intelligent reading of text, and includes: converting digits, expanding abbreviations and acronyms, handling punctuation, and interpreting special characters such as currency symbols.

While this chapter describes Speechify's default text processing, [Chapter 10](#) emphasizes the role of application developers in improving the quality of the synthesized output. For a complete understanding, you should read both chapters.

In This Chapter

- ❑ “Numbers” on [page 84](#)
- ❑ “Numeric expressions” on [page 85](#)
- ❑ “Mixed alphanumeric tokens” on [page 85](#)
- ❑ “Abbreviations” on [page 86](#)
- ❑ “Uppercase acronyms and tokens” on [page 88](#)
- ❑ “E-mail addresses” on [page 88](#)
- ❑ “URLs” on [page 89](#)
- ❑ “File names and paths” on [page 90](#)
- ❑ “Punctuation” on [page 91](#)
- ❑ “Parentheses” on [page 92](#)
- ❑ “Hyphen” on [page 92](#)
- ❑ “Slash” on [page 93](#)

Relevant languages

This chapter covers the major features of Speechify's *default* text normalization, which applies unless overridden by a main or abbreviation dictionary entry (see “User Dictionaries” on [page 109](#)) or an embedded tag (see “Embedded Tags” on [page 71](#)).

The discussion in this chapter applies to de-DE, en-AU, en-GB, en-US, es-MX, fr-CA, fr-FR, and pt-BR, but not to ja-JP. Examples in this guide are illustrated with en-US. See the appropriate language supplements for language-specific behaviors and examples.

Neutral expansions

In general, the default normalization expands expressions neutrally unless there is a disambiguating context. For example, the expression “1/2” can be interpreted in a variety of ways: “one half,” “one over two,” “January second,” “February first,” and so on. In special cases it is possible to disambiguate the expression; for example, in “3 1/2” it is a fraction and should be read “one half.”

When a disambiguating context cannot be identified, Speechify supplies a neutral reading, such as “one slash two,” on the theory that it is preferable to provide a reading that is intelligible in all circumstances instead of a specialized reading which is wrong for the context (e.g., reading “one half” when “January first” was intended).

Level of detail provided for normalization

In this chapter and in the language supplements, processing is described in general terms, to help application developers construct main and abbreviation dictionaries, custom preprocessors, and marked-up text. It is beyond the scope of this chapter to provide an exhaustive description of the entire set of sophisticated, context-sensitive text normalization rules used in Speechify, and you may therefore observe deviations from the generalizations provided here in specific instances.

Numbers

Speechify handles cardinal and ordinal numbers in standard formats, floating point digits, negative numbers, and fractions where these are unambiguous. See the appropriate language supplement for language-specific details on formats and interpretation.

Numeric expressions

Speechify handles dates, times, and monetary expressions in a variety of currencies and formats. Individual languages also handle phone numbers and social security numbers. See the appropriate language supplement for language-specific details on formats and interpretation.

Mixed alphanumeric tokens

Mixed alphanumeric sequences are either spelled, divided into words, or in special cases read as whole words. For example:

Example	Expansion
55th	fifty-fifth
1930s	nineteen thirties
3RD	third
VOS34	vee oh ess thirty-four
32XC	thirty-two eks cee
J2EE	jay two ee ee
Group12	group twelve
24hour	twenty-four hour
12min	twelve minutes

See the appropriate language supplement for any language-specific exceptions.

Abbreviations

Ambiguous abbreviations

There are two types of ambiguity in abbreviations:

- ❑ Some abbreviations have more than one expansion. For example, “St.” can mean “street” or “saint.” These abbreviations are homographs (words that have different pronunciations for different meanings). For more information on homographs, see [page 10-126](#).

Speechify employs a set of context-sensitive rules to disambiguate this type of ambiguous abbreviation:

Input	Expansion
Fortress Mt.	fortress mountain
Mt. Hood	mount hood
I live at 12 Houndhill Ct., Waterford, Ct. 06385	i live at twelve houndhill court, waterford connecticut oh six three eight five
Dr. Smith lives at 24 Rodeo Dr., Waco, Texas 02189	doctor smith lives at twenty-four rodeo drive, waco, texas, oh two one eight nine.

- ❑ Some abbreviations have the same form as non-abbreviated words. For example, “in” is either the abbreviation for “inches,” or the preposition “in.”

By default, Speechify assumes the text is a whole word and only interprets it an abbreviation in specific disambiguating contexts. For instance, when the abbreviation ends in a period, and the following word begins with a lowercase letter, the text is interpreted as an abbreviation. For example:

Input	Expansion
There are 5 in. of snow on the ground.	there are five inches of snow on the ground.
Put 5 in. Then take 3 out.	put five in. then take three out.

Periods after abbreviations

Abbreviations do not necessarily end in a period. For example:

Input	Expansion
We met Mr Smith yesterday.	we met mister smith yesterday.

If the abbreviation does end in a final period, Speechify expands the abbreviation and then determines whether the period also indicates a sentence end. For example:

Input	Expansion
There are 5 ft. of snow on the ground.	there are five feet of snow on the ground.
It snowed 5 ft. The next day it all melted.	it snowed five feet. the next day it all melted.

Measurement abbreviations

Measurement abbreviations are expanded to singular or plural depending on the value of a preceding number or other disambiguating context. In the absence of a disambiguating context, they default to a plural expansion. For example:

Input	Expansion
There are 5 in. of snow on the ground.	there are five inches of snow on the ground.
There is 1 in. of snow on the ground.	there is one inch of snow on the ground.
There are 1.1 in. of snow on the ground.	there are one point one inches of snow on the ground.
How many cm. are in a km?	how many centimeters are in a kilometer?
cm.	centimeters

Uppercase acronyms and tokens

Uppercase acronyms and other tokens written in uppercase are either read out letter by letter, read as a whole word, or given an idiosyncratic interpretation. Examples:

Examples	Pronunciation
TTSCMUCIA	tee tee esscee emm youcee eye ay
AAA	triple ay
UNICEF	unicef

A possessive suffix can be attached to an acronym without interfering with the ordinary acronym processing. For example:

Input	Expansion
CIA's	cee eye ay's
UNICEF's	unicef's

E-mail addresses

An e-mail address is divided into two portions, a username and a domain name, separated by an at sign (@). A phrase break is inserted following the username. Symbols in the e-mail address are read by name. The following table illustrates with en-US expansions.

Symbol	Expansion
@	at
.	dot
-	dash
_	underscore

Username

The username is spelled out character by character, unless word boundaries are indicated unambiguously. Sequences of two and three letters are always spelled out. Digit sequences are read digit by digit. Examples (commas are used to indicate phrasing):

Username	Expansion
bruce_smith	bruce underscore smith,
john.jones	john dot jones,
star-chaser	star dash chaser,
steve5050	steve five zero five zero,
red	ar ee dee,
rfrmac	ar eff ar emm ay cee,

Domain name

Two- and three-letter domain and country extensions are either read as words or spelled out, following standard convention. The host name is read as a single word, unless word boundaries are indicated unambiguously. English examples:

Host name	Expansion
access1.net	access one dot net
hawaii.rr.com	hawaii dot ar ar dot com
cornell.edu	cornell dot ee dee you
amazon.co.uk	amazon dot cee oh dot you kay

URLs

A token beginning with “www.” or “http://” or “ftp://” is interpreted as a URL. A phrase break is inserted following “http://” or “ftp://,” and the “://” piece is not pronounced.

Symbols in a URL are expanded as follows in US English:

Symbol	Expansion
/	URL-final: not expanded otherwise: slash
.	dot
-	dash
_	underscore

Each slash-delimited segment of the URL is expanded as follows: Two- and three-letter domain and country extensions are either read as words or spelled out, following standard conventions. Each remaining segment is read as a single word, unless word boundaries are indicated unambiguously. Examples:

URL	Expansion
http://www.lobin.freeseve.co.uk	aitch tee tee pee, double you double you double you dot lobin dot freeseve dot cee oh dot you kay
sarhs-price/page001.html	sarhs dash price slash page zero zero one dot aitch tee em ell
www.serbia-info.com/news/	double you double you double you dot serbia dash info dot com slash news

File names and paths

Forward and backward slashes are not expanded at the end of a path. In other contexts, they are expanded by name. Symbols in paths are expanded by name. The following table illustrates the en-US expansions.

Symbol	Expansion
/	URL-final: not expanded otherwise: slash
\	URL-final: not expanded otherwise: backslash
.	dot
-	dash
_	underscore

Each slash-delimited segment of a path is read as a single word, unless word boundaries are unambiguously indicated. Common file name extensions are read as a word or spelled out, following standard conventions.

Path	Expansion
C:\docs\my_book\chapter12.doc	cee colon backslash docs backslash my underscore book backslash chapter twelve dot doc
/product/release/speechify-2-1-5/ release-notes.txt	slash product slash release slash speechify dash two dash one dash five slash release dash notes dot tee eks tee

Punctuation

Punctuation generally triggers a phrase break, except in a limited set of special cases that are determined on a language-specific basis. Some English examples of commas that don't produce phrase breaks:

Input	Expansion
He lives in Boston, Massachusetts.	he lives in boston massachusetts.
That is a very, very old building.	that is a very very old building.
John won't come, either.	john won't come either.
SpeechWorks International, Inc.	speechworks international incorporated.
Drive D: is full.	drive dee is full.

Parentheses

Parentheses generally trigger a phrase break, except in a very limited set of special cases which vary from language to language. See the appropriate language supplement for relevant examples. Some English examples (commas used to indicate phrasing).

Input	Expansion
Tom (my son) and Susan (my daughter)	Tom, my son, and Susan, my daughter
book(s)	books
getText()	get text

Hyphen

The hyphen is read neutrally as “dash” (or the equivalent) unless it can be disambiguated with a high degree of confidence. See the appropriate language supplement for language-specific examples.

Input	Expansion
mid-90s	mid nineties
A-1	ay one
32-bit	thirty-two bit
-7	minus seven
April 3-4	April third to fourth
pp. 35-40	pages thirty-five to forty
1974-1975	nineteen seventy-four to nineteen seventy-five
2-3 inches	two to three inches
3-2	three dash two (since the desired expansion could be “three to two,” “three minus two,” “three two,” etc.)

Slash

A slash is read as “slash” (or the equivalent) unless one of the following is true:

- a. The following word is a unit of measure, when the slash is read as per (or the equivalent),
- b. When the entire token is a familiar expression.

See the appropriate language supplement for language-specific examples.

Input	Expansion
Seattle/Boston	seattle slash boston
cm/sec	centimeters per second
he/she	he or she



Symbolic Phonetic Representations

A Symbolic Phonetic Representation (SPR) is the phonetic spelling used by Speechify to represent the pronunciation of a single word. An SPR represents:

- ❑ The sounds of the word
- ❑ How these sounds are divided into syllables
- ❑ Which syllables receive stress, tone, or accent

You can use SPRs as input to Speechify in order to specify pronunciations that are different from those generated by the system automatically. SPRs are used in two different ways:

- ❑ Enter an SPR directly into text in place of the ordinary spelling of a word. In the following example, an SPR replaces the word *root* in order to trigger a pronunciation that rhymes with *foot* rather than *boot*:

```
We need to get to the \![rUt] of the problem.
```

- ❑ To make the pronunciation change more permanent, enter an SPR as the translation value of either a main or root dictionary entry. The specified pronunciation is then generated whenever that word is encountered in any text. (See “User Dictionaries” on [page 109](#).)

In This Chapter

- ❑ “SPR format and error handling” on [page 96](#)
- ❑ “Syllable boundaries” on [page 97](#)
- ❑ “Syllable-level information” on [page 97](#)
- ❑ “Speech sound symbols” on [page 97](#)

SPR format and error handling

An SPR consists of a sequence of allowable SPR symbols for a given language, enclosed in square brackets [] and prefixed with the embedded tag indicator \! (See “Embedded Tags” on [page 71](#).)

These examples are valid SPRs in US English:

```
though \![.1Do]
shocking \![.1Sa.0kIG]
```

The periods signal the beginning of a new syllable, the digits 1 and 0 indicate the stress levels of the syllables, and the letters D, o, S, a, k, I, and G represent specific US English speech sounds. Each of these elements of the SPR is discussed below.

An SPR entry which does not conform to the requirements detailed in this chapter is considered invalid and ignored. Speechify logs an error to the error log, and continues processing as normal.

Here is an example of an entry in the error log. If you enter an invalid SPR, \![k], the following would be logged to the error log:

```
May 07 11:08:23.31|1592||0|WARNING|6287|Speechify server:
Warning, invalid SPR entered|SPR=\![k]|reason=No vowel in
syllable or word
```

This log entry includes both the SPR and the reason why it's incorrect. The list of reasons includes:

- ☐ invalid bracketing
- ☐ no vowel in syllable or word
- ☐ invalid stress value
- ☐ invalid stress placement
- ☐ invalid phone symbol
- ☐ unmatched single quote

Syllable boundaries

You can use periods to delimit syllables in an SPR, although they are not required. In de-DE, these periods determine the location of the syllable boundaries of the word. In other languages the periods are only used to enhance the readability of the SPR, and do not affect the way the word is syllabified in the speech output. Speechify's internal syllabification rules apply as usual to divide the word into syllables. See the appropriate language supplement for details.

Syllable-level information

Use digits to indicate the stress or (in Japanese) the pitch accent of the syllable. In languages other than Japanese, a polysyllabic word with no stress marked is assigned a default stress. To ensure the correct pronunciation of the word, you should mark stress explicitly in the SPR. Marking stress and pitch accent in an SPR varies by language and is described in the appropriate *Speechify Language Supplement*.

Speech sound symbols

Each language uses its own inventory of SPR symbols for representing its speech sounds. See the appropriate *Speechify Language Supplement* for a table of SPR symbols and examples of words in which each sound occurs. These tables show valid symbols for vowels, consonants, syllable stresses, and syllable boundaries.



NOTE

Letters are case-sensitive, so \[e] and \[E] represent two distinct sounds. Multi-character symbols must be contained in single quotes; for example, French *peu* is represented \[p'eu']. SPRs containing sound symbols that do not belong to the inventory of the current language are invalid, and generate an error in the error log as described above, with the invalid symbol ignored.

Some speech sounds have limited distributional patterns in specific languages. For example, in English, the sound [G] of *sing* \[.1sIG] does not occur at the beginning of a word. Other US English sounds that have a particularly narrow distribution are the flap [F], and the syllabic nasal [N]. Entering a sound symbol in a context where it does not normally occur may result in unnatural-sounding speech.

Speechify applies a sophisticated set of linguistic rules to its input to reflect the processes by which sounds change in specific contexts in natural language. For example, in US English, the sound [t] of *write* \[.1rYt] is pronounced as a flap [F] in *writer* \[.1rY.0FR]. SPR input undergoes these modifications just as ordinary text input does. In this example, whether you enter \[.1rY.0tR] or \[.1rY.0FR], the speech output is the same.



W3C SSML Compliance

Speechify can accept input in the form of W3C Speech Synthesis Markup Language (SSML). Speechify complies with the Last Call Working Draft specification (2 December 2002) which can be found at <http://www.w3.org/TR/speech-synthesis>. That document describes all the markup specifications that make up the W3C SSML.

In This Chapter

- ❑ “Speechify’s W3C SSML parsing” on [page 99](#)
- ❑ “Element support status” on [page 100](#)

Speechify’s W3C SSML parsing

Speechify includes appropriate processing for almost all of the elements and attributes defined by the specification. Speechify has two modes of W3C SSML parsing: *strict* and *non-strict*, as configured by `tts.ssml.doStrictValidation` in the Speechify XML configuration file.

- ❑ In strict mode, any documents that fail validation against the W3C SSML 2 December 2002 XML schema document (installed in `doc/synthesis.xsd`) are rejected, with `SWItts_SSML_PARSE_ERROR` returned for the `speak` request.
- ❑ In non-strict mode, documents that are not valid XML documents are rejected, with `SWItts_SSML_PARSE_ERROR` returned for the `speak` request. However, Speechify does not attempt to validate the document against the W3C SSML schema document: instead, it processes the document on a best-effort basis in order to allow Speechify to support most documents conforming to older W3C

SSML working drafts. In this mode, for tags whose content fails to parse or is ambiguously formatted, the server attempts to synthesize the raw text surrounded by the tag, while also reporting a warning to the server error log. In practice, this often results in correct output.

In addition, for both strict and non-strict mode there are cases where the Speechify server may encounter other issues during parsing, such as unsupported say-as interpret-as/format combinations, or text within a say-as tag that cannot be recognized as the specified data type. In those cases, Speechify speaks the information on a best-effort basis (usually using normal Speechify front-end processing rules) and logs a warning to the Speechify error log..

Element support status

The table below outlines the extent to which W3C SSML elements are currently supported by Speechify. It also notes the limitations on element attributes and content that Speechify currently requires of supported elements.

Elements	Support	Attributes and content notes/limitations
Audio	Yes	Speechify supports VoiceXML extended <audio> attributes “fetchtimeout,” “fetchhint,” “maxage,” and “maxstale.” Audio is queued for the output stream only when the entire fetch successfully completes, so it is not possible for fetch errors to result in partially played audio. See Support of the “audio” element below for details.
Break	Yes	<p>If the time attribute is not used, the default break duration is 0.7 seconds. Speechify assigns a break duration for each attribute:</p> <ul style="list-style-type: none">❑ x-small: 0.17 sec❑ small: 0.35 sec❑ medium: 0.7 sec❑ large: 1 sec❑ x-large: 1.35 sec <p>Speechify supports break durations between .001 and 32.767 sec. To create a break longer than 32.767 sec, use a series of consecutive <break> elements.</p>
Desc	Yes	Supported, but Speechify does not provide access to the description text.

Elements	Support	Attributes and content notes/limitations
Lexicon	Yes	<p>Dictionaries must be in the SpeechWorks XML dictionary format.</p> <p>ScanSoft supports an extra attribute, priority, which permits specifying the dictionary's priority. If the priority attribute is not set, the priority is determined by the <code>tts.engine.dictionaryDefaultPriorityBase</code> value in the Speechify configuration file.</p>
Mark	Yes	Unlike native embedded tags, Unicode characters are allowed.
Metadata	Yes	Supported, but Speechify does not provide access to the metadata.
Paragraph, p	Yes	
Phoneme	Partial	<p>The phonetic representation (i.e., the "ph" attribute) must be specified in the Speechify SPR format. If the "alphabet" attribute is not specified, Speechify SPR format is assumed. Otherwise if the alphabet value is not "SWItts," a warning is logged, with the element content spoken instead of the phoneme string. See "Symbolic Phonetic Representations" on page 95.</p>
Prosody	Partial	<p>The attributes "rate" and "volume" are supported; "pitch," "contour," "range," and "duration" are not. (The unsupported attributes are ignored with a warning logged to the Speechify error log.) Note that the values given for these attributes are interpreted relative to the corresponding port-specific values in effect at the time as set by <code>SWIttsSetParameter()</code>. (See "Controlling the audio characteristics" on page 80 for an explanation of port-specific values.)</p> <p>Named rate values are treated as follows:</p> <ul style="list-style-type: none"> ❑ x-fast: 200% of the normal rate ❑ fast: 150% of the normal rate ❑ medium: use the normal rate ❑ slow: 66% of the normal rate ❑ x-slow: 33% of the normal rate ❑ default: use the normal rate <p>Named volume values are treated as follows:</p> <ul style="list-style-type: none"> ❑ silent: not audible ❑ x-soft: 16% of the normal volume ❑ soft: 33% of the normal volume ❑ medium: 66% of the normal volume ❑ loud: use the normal volume ❑ x-loud: 150% of the normal volume ❑ default: use the normal volume
Say-as	Yes	<p>While W3C SSML defines the "interpret-as," "format," and "detail" attributes, their possible values are not standardized. Speechify supports a wide variety of values. See Support of the "say-as" element below for details.</p>
Sentence, s	Yes	

Elements	Support	Attributes and content notes/limitations
Speak	Yes	The "xml:lang" attribute is only available for languages supported by Speechify. If an unsupported language is specified, the server logs an error and attempts to process subsequent tags using US English rules.
Sub	Yes	
Voice	Partial	The "variant" attribute is ignored. Switching voices and languages is not supported, so if the "xml:lang," "gender," "name," and/or "age" attribute do not match the current Speechify server voice, Speechify logs a warning to the error log with the contained text spoken in the current Speechify server voice.

Support of the "audio" element

W3C SSML <audio> supports the insertion of pre-recorded audio into the synthesis stream. The src attribute is required, and indicates the audio to insert, which may be a local file on the Speechify server host (file:// access) or fetched from a web server (http:// access). If the audio to insert is not found, and the <audio> element has a body, the body of the <audio> element will be spoken as fallback.

For example:

```
<?xml version="1.0"?>
<speech version="1.0" xmlns="http://www.w3.org/2001/10/
  synthesis">
  Test audio with fallback

  <audio src="http://myhost/1.ulaw"/>
  <audio src="http://myhost/2.ulaw">
    2.ulaw fetch failed
  </audio>
</speech>
```

VoiceXML <audio> attributes

W3C SSML 1.0 only specifies a single attribute for <audio>, the src attribute (required). The VoiceXML specification and W3C VoiceXML 2.0 working draft specify additional attributes as summarized in the table below. The table also discusses unsupported attributes.

Attribute	Directly supported by Speechify	Description
caching (VoiceXML 1.0 only)	No	Either safe to force a query to fetch the most recent copy of the content (identical to maxage=0), or fast to use the cached copy of the content if it has not expired (default behavior as if caching were not specified).
expr (VoiceXML 2.0 WD only)	No	Dynamically determine the URI to fetch by evaluating this ECMAScript expression.
fetchhint	Yes	Defines when the interpreter context should retrieve content from the server. prefetch indicates a file may be downloaded when the page is loaded, whereas safe indicates a file that should only be downloaded when actually needed.
fetchtimeout	Yes	The interval to wait for the content to be returned before throwing an error.badfetch event.
maxage (VoiceXML 2.0 WD only)	Yes	Indicates that the document is willing to use content whose age is no greater than the specified time in seconds. The document is not willing to use stale content, unless maxstale is also provided.
maxstale (VoiceXML 2.0 WD only)	Yes	Indicates that the document is willing to use content that has exceeded its expiration time. If maxstale is assigned a value, then the document is willing to accept content that has exceeded its expiration time by no more than the specified number of seconds.

Speechify does not support the caching attribute because the W3C VoiceXML 2.0 working committee explicitly removed that from the VoiceXML 2.0 working draft, replacing that with the more useful maxage attribute. To delegate prompting to Speechify, you need to do the following when extracting VoiceXML prompting into a W3C SSML 1.0 document for Speechify:

1. Replace caching="safe" with maxage="0"
2. Omit caching="fast"
3. Specify xml:base for the <speak> element in the W3C SSML document to indicate the base URL (current base URL for VoiceXML processing) for resolving any embedded lexicon and audio elements.

Speechify does not support the `expr` attribute because that involves looking up variables within the VoiceXML variable scope chain. To delegate prompting to Speechify, you need to resolve `expr` to a simple string when extracting VoiceXML prompting into a W3C SSML 1.0 document for Speechify.

Supported audio URIs

Speechify supports `<audio>` URIs as follows. Much of the more sophisticated options (POST, cookies, relative URLs, specifying fetch timeouts) rely on the `SWIttsSpeakEx()` function, which permits passing a read-only `VXIMap` containing fetch properties as well as a read-write `VXIMap` that stores the cookie jar.

This includes the following URI forms:

- ❑ absolute and relative `http://` URLs
- ❑ absolute and relative `file://` URLs
- ❑ local UNIX style absolute and relative file paths (those using forward slashes with no drive letters as in `/temp/myfile.vxml`)
- ❑ (Win32 only) Local Win32 style absolute and relative file paths (those with drive letters and/or back-slashes as in `c:\temp\myfile.vxml`)
- ❑ (Win32 only) Win32 UNC style absolute file paths (those with two leading backward slashes as in `\\netapp1\user\myfile.vxml`)

Supported audio formats

Speechify supports the formats listed below, however, any specific Speechify server instance supports only audio insertions with a sampling rate that matches that of the current Speechify server voice.

For example, when using the 8kHz Tom voice, only 8kHz based audio insertions are supported (`audio/basic`, `audio/x-alaw-basic`, `audio/L16;rate=8000`, and `audio/wav` and `audio/x-wav` containing 8kHz samples). When using the 16kHz Tom voice, only 16kHz based audio insertions are supported (`audio/L16;rate=16000`, and `audio/wav`

and audio/x-wav containing 16kHz samples). If the rates do not match, the Speechify server will report an error to the error log, then proceed to normal SSML <audio> element fallback as if the audio was not found.

Audio format (for 8 kHz Speechify server instances)	Media type
Raw (headerless) 8kHz 16-bit mono big endian (network byte) order linear 16 [PCM] single channel.	audio/L16;rate=8000
Raw (headerless) 8kHz 8 bit mono A-law [PCM] single channel. (G.711)	audio/x-alaw-basic
Raw (headerless) 8kHz 8-bit mono mu-law [PCM] single channel. (G.711)	audio/basic
WAV (RIFF header) 8kHz 16-bit mono linear 16 [PCM] single channel. (WAV always encodes linear 16 samples as little endian.)	audio/wav or audio/x-wav
WAV (RIFF header) 8kHz 8-bit mono A-law [PCM] single channel.	audio/wav or audio/x-wav
WAV (RIFF header) 8kHz 8-bit mono mu-law [PCM] single channel.	audio/wav or audio/x-wav

Audio format (for 16 kHz Speechify server instances)	Media type
Raw (headerless) 16kHz 16-bit mono big endian (network byte) order linear 16 [PCM] single channel.	audio/L16;rate=16000
WAV (RIFF header) 16kHz 16-bit mono linear 16 [PCM] single channel. (WAV always encodes linear 16 samples as little endian.)	audio/wav or audio/x-wav

User-defined audio formats

Many voice processing platforms define custom audio formats, frequently simply audio/basic or audio/x-alaw-basic with a small proprietary header. You can support these formats, by “extending” Speechify. To do this, follow these steps:

1. Build a web server servlet (CGI script, JSP, or ASP) that takes as input (an http:// URI query argument) an audio file and/or URI in the desired audio format, and returns audio in the closest matching Speechify supported audio insertion format. (For example, extract the audio/basic samples from a Sun audio format file, returning simple audio/basic samples with the audio/basic MIME content type.) Have the servlet return HTTP/1.1 caching controls that permit the Speechify

server to cache the returned audio in order to reduce the load on the audio conversion servlet and to reduce Speechify server <audio> resolution latencies.

2. Write the original VoiceXML document or W3C SSML 1.0 document so that it refers to audio via an http:// URI that passes through that audio format conversion servlet, rather than directly referencing the URI or file containing the unsupported audio format

Handling missing URIs

If an <audio> URI is not found, Speechify behaves according to the W3C SSML 1.0 working draft:

1. If the <audio> element has content (fallback), attempt to use the fallback.
2. If there is no fallback, report a warning to the Speechify server log, then proceed without any audio insertion.

Support of the “say-as” element

The “say-as” element identifies the type of text contained therein. This can facilitate the correct pronunciation of the element’s content. Speechify supports all the W3C SSML attributes: “interpret-as,” “format,” and “detail.” However, W3C SSML does not define the possible values for these attributes, nor the precise semantics. The table below lists the Speechify defined values and provides some guidance to their usage:

say-as “interpret-as” attribute	say-as “format” attribute	Notes and cautions
acronym		Letters and numbers are pronounced individually. Use detail="strict" to speak pronunciation (e.g., speaking a comma as “comma”). Same as “letters.”
address		
cardinal		Supported if relevant in the target language.
currency		Supports currency as commonly specified in the country corresponding to the target language. For example, dollars and cents for US English, Euros for Parisian French, and pesos and centavos for Mexican Spanish.

say-as "interpret-as" attribute	say-as "format" attribute	Notes and cautions
date	dmy, mdy, ymd, ym, my, md, y, m, d	If the year is written using only two digits, numbers less than 50 are assumed to occur in the 2000s, greater than 50 in the 1900s.
digits		Supported if relevant in the target language.
duration	hms, hm, ms, h, m, s	For example: "duration:hms" is read out as "<h> hour(s), <m> minute(s), and <s> seconds" (assuming xml:lang was specified as "en-US").
letters		Letters and numbers are pronounced individually. Use detail="strict" to speak pronunciation (e.g., speaking a comma as "comma"). Same as "acronym."
measure		A variety of units (e.g., km, hr, dB, lb, MHz) is supported; the units may appear immediately next to a number (e.g., 1cm) or be separated by a space (e.g., 15 ms); for some units the singular/plural distinction may not always be made correctly.
name		
net	email, uri	Note that e-mail addresses are difficult to pronounce the way a user expects.
number	cardinal, ordinal, digits, telephone	Only formats relevant in the target language are supported; Roman numerals are not supported. All the format values are supported as interpret-as values as well, behaving the same for either syntax.
ordinal		Supported if relevant in the target language.
telephone		Supports telephone numbers as commonly specified in the country corresponding to the target language. For "en-US" this includes: (800) 123-4567, 617-123-4567, and 212 123-4567. Use detail="punctuation" to speak punctuation (e.g., speaking a dash as "dash"). (The resulting output is the same as using letters with detail="strict".)
time	hms, hm, h	The hour should be less than 24, minutes and seconds less than 60; AM/PM is read out only if explicitly specified.
words		This biases the interpretation of the text towards speaking it as a word instead of speaking it as an acronym. However, the text may still be spoken as letters in some cases for particularly difficult to pronounce character sequences.



User Dictionaries

Speechify supports user dictionaries for customizing the pronunciations of words, proper names, abbreviations, acronyms, and other sequences. For example, if you want the sequence *SWI* to be pronounced *Speech Works* every time it occurs, you can create a dictionary entry specifying this pronunciation.

This chapter describes in detail the types of user dictionaries supported in Speechify, the make-up of dictionary entries within each dictionary type, and how to best make use of Speechify's dictionary facilities to customize pronunciations. For details on loading, activating, and formatting dictionary files, see "Using User Dictionaries" on [page 61](#).



NOTE

The user dictionaries are used to specify pronunciations that you would like to persist across one or more speak requests. For customized pronunciations which apply only to specific instances of a text string, use Symbolic Phonetic Representations and the other embedded tags described in "Symbolic Phonetic Representations" on [page 95](#) and "Embedded Tags" on [page 71](#).

In This Chapter

- ❑ "Overview of dictionaries" on [page 110](#)
- ❑ "Main dictionary" on [page 111](#)
- ❑ "Abbreviation dictionary" on [page 111](#)
- ❑ "Root dictionary" on [page 112](#)
- ❑ "Mainext dictionary" on [page 113](#)
- ❑ "Choosing a dictionary" on [page 114](#)

Overview of dictionaries

A dictionary entry consists of a *key* and a *translation value*. When the key matches a string in the input text, the translation value replaces it, and the translation is then pronounced according to Speechify's internal pronunciation rules.

In general, user dictionaries are used to overcome areas where Speechify's pronunciation rules do not produce correct pronunciations. This can be the case with proper names, names of corporations, foreign words, etc.

For most languages, Speechify provides three types of user dictionaries:

- ❑ “**Main dictionary**”. The main dictionary is an all-purpose exception dictionary that can be used to replace a single token in an input text with almost any other valid input sequence.
- ❑ “**Abbreviation dictionary**”. The abbreviation dictionary is used to expand abbreviations, and provides for special treatment of period-delimited tokens.
- ❑ “**Root dictionary**”. The root dictionary is used to specify pronunciations, orthographically or via SPRs, of words or morphological word roots.

In Japanese, Speechify uses a single user dictionary (“mainext”), described in the *Speechify Language Supplement* for ja-JP.

Each dictionary is characterized by the kinds of keys and translation values it accepts. These are described in detail in the subsections below.

Dictionary lookups ignore surrounding parentheses, quotation marks, and the like. For example, if *WHO* is a key in a main dictionary entry, it matches input strings such as *(WHO)* and “*WHO*”.

Main dictionary

The main dictionary is an all-purpose exception dictionary which can replace a word in input text with almost any type of input string. It is more permissive than the other dictionaries in the make-up of its keys and translations:

- ❑ a valid key may contain any characters other than white space
- ❑ a valid translation (e.g., the double-quoted value of <definition>) consists of any valid input string

You can use the main dictionary for:

- ❑ Strings that translate into more than one word
- ❑ Translations which include Symbolic Phonetic Representations or other embedded tags
- ❑ URLs and e-mail addresses with idiosyncratic pronunciations
- ❑ Keys containing digits or other non-alphabetic symbols
- ❑ Acronyms with special pronunciations

Additional notes on main dictionary entries

The main dictionary is case-sensitive. For example, lowercase *who* does not match a main dictionary key *WHO*.

Main dictionary translations may include Speechify embedded tags but not SAPI or W3C SSML tags.

For formatting requirements and examples of valid main dictionary keys and translations, see the appropriate *Speechify Language Supplement*.

Abbreviation dictionary

The abbreviation dictionary is designed to handle word abbreviations which translate to one or more words in ordinary spelling.

Like the main dictionary, the abbreviation dictionary is case-sensitive. For example, if you enter the key *Mar* with translation *march*, lowercase *mar* is still pronounced as *mar*.

For formatting requirements and examples of valid abbreviation dictionary keys and translations, see the appropriate *Speechify Language Supplement*.

Interpretation of trailing periods in the abbreviation dictionary

When you enter a key in the abbreviation dictionary, it is not necessary to include the “trailing” period that is often the final character of an abbreviation (as in the final period of *etc.*). If the key does not contain the trailing period, it matches input text both with and without the period. However, if you want an abbreviation to be pronounced as specified in the translation *only* when it is followed by a period in the text, then you must enter the trailing period in the key. The following table summarizes the use of trailing periods in entry keys:

Key	Matches
inv	inv. inv
sid.	sid.

An abbreviation dictionary entry invokes different assumptions about how to interpret the trailing period in the text than does a main dictionary entry. Since the final period cannot be part of a main dictionary entry key, it is always interpreted as end-of-sentence punctuation. A period following an abbreviation dictionary entry, on the other hand, is ambiguous. It is only interpreted as end-of-sentence punctuation if other appropriate conditions obtain (e.g., if it is followed by at least two spaces and an uppercase letter). For example, input (a) is interpreted as one sentence, while input (b) is interpreted as two sentences.

Input

- (a) It rained 2 cm. on Monday.
- (b) On Sunday it rained 2 cm. On Monday, it was sunny.

Root dictionary

The root dictionary is used for ordinary words, like nouns (including proper names), verbs, or adjectives. Unlike the main and abbreviation dictionaries, lookups are not case-sensitive. While you must enter a root into the dictionary in lowercase letters, the entry matches an input string regardless of case. For example, the pronunciation

specified in the translation still applies when the word begins with an uppercase letter (as the first word in a sentence, for example) or when it is spelled in all uppercase letters for emphasis.

The translation value of a root dictionary entry may be either a Symbolic Phonetic Representation (see “Symbolic Phonetic Representations” on [page 95](#)) or a word in ordinary spelling. For example, if you want the word *route* to be pronounced with the vowel of *out* rather than the vowel of *boot*, you could enter the translation as the SPR `\![rWt]`, or in letters as *rout*, which also produces the desired vowel.

Beware of keys without vowels

Do not specify keys that consist entirely of consonants in the root dictionary; use main dictionary instead.

A root dictionary key cannot consist entirely of consonants, since Speechify spells out sequences of all consonants before root dictionary lookup occurs. An entry like the following can be placed in the main dictionary instead:

```
ng \! [.1EG]
```

Check language supplements for variations

The use of the root dictionary varies among languages. For formatting requirements and examples of valid root dictionary keys and translations, see the appropriate *Speechify Language Supplement*.

Mainext dictionary

The mainext dictionary is the only user dictionary supported in Japanese. See the language supplement for details.

Choosing a dictionary

This section covers guidelines for determining which dictionary to use for particular entries.

Speechify includes two kinds of text processing functions:

- ❑ determining how the input text should be interpreted
- ❑ specifying how words are pronounced

The first kind of processing is discussed in greater detail in [Chapter 6](#), and includes expanding abbreviations and acronyms, reading digit sequences, and interpreting non-alphanumeric characters such as punctuation and currency symbols. For example, specifying that the abbreviation “Fri.” should be read as “Friday” is different from specifying whether the word “Friday” should be pronounced so that the last vowel is the same as in the word “day” or the same as in the word “bee.” This distinction is reflected in the dictionaries, as discussed in the following sections.

Main and abbreviation dictionaries vs. root dictionary

Both the main and abbreviation dictionaries are used to specify how input text should be interpreted, while the root dictionary is used to specify how individual words should be pronounced. Word pronunciations can be specified by either an SPR (phonetic representation) or another word (real or invented) that has the desired pronunciation.

Because words are, with a few exceptions, pronounced the same whether they are in uppercase, lowercase, or mixed case letters, root dictionary lookup is not case-sensitive. An entry that you add with a lowercase key applies to words in text when they are capitalized or uppercase. For example, if you specify that “friday” should be pronounced “frydi,” then “Friday” and “FRIDAY” are also pronounced “frydi.”

On the other hand, main and abbreviation dictionary lookups are case-sensitive, since the reading of particular character strings can be affected by letter case. For example, the abbreviation “ft.” is read as “foot,” while “Ft.” is read as “Fort.” “ERA” might be read as “Equal Rights Amendment,” but “era” should still be read as “era.”

Main dictionary vs. abbreviation dictionary

The main dictionary can be thought of as a string replacement dictionary: with the few exceptions outlined in the Main dictionary section above, it is used to replace any input string with any other input string. Use it when either the key or the translation contains non-alphabetic characters. You can use it to specify the reading of e-mail addresses, alphanumeric sequences such as “Win2K,” digit sequences, and the like, or even to encode entire phrases or sentences in a single token.

The abbreviation dictionary is used to expand what we normally think of as abbreviations: shortened forms of a word that may end in a period to signal that they are abbreviations, and which expand into full words. Abbreviations are entered in a separate dictionary because the final period is handled differently in abbreviations than in main dictionary entries (see “Interpretation of trailing periods in the abbreviation dictionary” on [page 112](#)).

The following table shows some sample entries, along with the dictionary they should be entered in.

Key	Translation	Dictionary
Win2K	Windows 2000 or Windows two thousand	main
4x4	4 by 4 or four by four	main
50%	half	main
Phrase217	How may I help you?	main
FAA	Federal Aviation Administration	main
f.a.a.	free of all average	abbreviation
sunday	sundi or \[1sHindi]	root
quay	key	root

Dictionary interactions

The order of lookups is main, abbreviation, root. Since the translation from one dictionary is looked up in subsequent dictionaries, dictionary entries can feed into one another. For example, given the dictionary entries in the following table:

Key	Translation	Dictionary
h/w	hrs. per wk.	main
hrs.	hours	abbreviation
wk.	week	abbreviation
hours	\[1arz]	root

The input and output are as follows:

Input	Pronunciation
h/w	\[1arz] \[1pR] \[1wik]

If the same key occurs in more than one dictionary, the order of lookups determines which entry is used. For example, given the following entries:

Key	Translation	Dictionary
win	Microsoft Windows	main
win	windows	abbreviation
win	\[wIn]	root

The input and output are as follows:

Input	Pronunciation
win	microsoft windows

Remember, however, that the main and abbreviation dictionary lookups are case-sensitive, while the root dictionary lookups are case-insensitive. Given the entries shown above, only the root dictionary entry will apply to “Win”:

Input	Pronunciation
Win	\[wIn]



Improving Speech Quality

This chapter describes techniques for application developers to improve the quality of Speechify's output. Topics:

- ❑ The introduction discusses the complexity of the text-to-speech task and explains why your applications might need customization to generate optimal output.
- ❑ Most of the chapter discusses how to improve speech output by customizing Speechify's analysis of input text.
- ❑ A final section briefly discusses how to overcome problems in the sound of the speech output.

ScanSoft offers consulting services to help you tune Speechify's output quality and the performance of individual applications.

In This Chapter

- ❑ "Introduction" on [page 118](#)
- ❑ "Customizing text analysis" on [page 119](#)
- ❑ "Text manipulation" on [page 122](#)
- ❑ "Case study – homograph disambiguation" on [page 126](#)
- ❑ "Case study – navigation text" on [page 127](#)
- ❑ "Speech concatenation issues" on [page 129](#)
- ❑ "Logging generated pronunciations" on [page 130](#)

Introduction

When people read a written text aloud, we speak accurately, fluently, and naturally; and we perform the task so automatically that we tend to overlook the extraordinary complexity of factors and of the knowledge that underlies our proficiency. This proficiency encompasses linguistic knowledge, such as rules of spelling, grammar, pronunciation, and word meanings, as well as knowledge of the real world and the pragmatic situations in which the text is used.

Consider the sophisticated linguistic and real-world knowledge involved in interpreting the commas in the following sentences. Commas are typically associated with *phrase breaks* – pauses accompanied by prosodic changes in pitch and duration – as in the following sentences:

- a. I want to spend our vacation in Maine, but Tom prefers to visit Minnesota.
- b. I visited Maine, Wisconsin, and Minnesota.

Sometimes, we insert phrase breaks at locations unmarked by punctuation such as after the word "Wisconsin" in this alternate form of sentence (b):

- c. I visited Maine, Wisconsin and Minnesota.

Conversely, some commas should not result in a phrase break. In sentence (d) below, a comma is inserted between the city and state as a matter of convention, but the punctuation does not signify any prosodic changes.

- d. ScanSoft is headquartered in Peabody, Massachusetts.

Now consider sentences (e) and (f):

- e. I visited Boston, Massachusetts and Washington.
- f. I visited Chicago, Massachusetts and Washington.

Did the writer visit two places or three? That is, should the commas after *Boston* and *Chicago* be ignored, as in (d), or interpreted as phrase breaks, as in (b)? There is nothing about the structure of sentences (e) and (f) to answer these questions. Most likely you will automatically read sentence (e) as a list of two places, with no phrase break following *Boston*, and sentence (f) as a list of three places, with a phrase break following *Chicago* and *Massachusetts*. Your choice is not due to anything about the text itself, but to your knowledge of the real world: you know that Boston is a city in Massachusetts and Chicago is not. The following sentence is challenging even to human readers:

g. I visited Albany, New York and Washington.

Above, did the writer visit two places or three? Given that Albany is the capital city of New York, it is not clear whether this person visited *Albany, NY* or *New York, NY*. Not even real-world knowledge can answer this question; you must have particular knowledge about the writer's experience in order to read this sentence with the appropriate phrase breaks.

Commas and phrase breaks are just one of countless examples that highlight how linguistic complexity and sophistication combine with real-world knowledge enable humans to read a text aloud accurately and naturally. Although text-to-speech systems can incorporate much of the linguistic knowledge and some of the real-world knowledge that people take for granted, it is unrealistic to expect any text-to-speech system to always perform its task perfectly.

Fortunately, there are many techniques available to application developers for improving the quality of Speechify's output. In the following sections, we discuss specific measures that developers can use to assist Speechify in analyzing the text to provide as accurate and natural a rendering as possible.

Customizing text analysis

Speechify performs text analysis on the application's input text. This includes the following components:

- ❑ Text normalization
- ❑ Phrasing
- ❑ Word pronunciations
- ❑ Word emphasis

Speechify's default text analysis

Speechify provides various techniques for application developers to customize the text analysis to suit the needs of each application. These techniques are reviewed in the following sections.

As a prerequisite to using the techniques most effectively, you must understand Speechify's default text analysis, which will apply in the absence of your customizations. By understanding the defaults, you can focus efforts on:

- ❑ Text elements that are not handled automatically
- ❑ Text elements that need to be handled in a non-default way

In addition to the information below, Speechify's default text normalization is described in detail in [Chapter 6](#) and in the corresponding chapters of each language supplement.

Customizing the default text analysis

Strategies for customizing Speechify's text analysis fall into two broad categories: those that require preprocessing of the input text by your application, and those that do not.

Strategies with no preprocessing

User dictionaries – You can populate Speechify's main and abbreviation dictionaries with expansions of abbreviations and other idiosyncratic tokens, and you can use root dictionaries to tune word pronunciations. See [Chapter 9](#) and in the corresponding chapters of each language supplement.

SpeechWorks e-mail preprocessor – ScanSoft offers an add-on processor that improves the output speech when reading e-mail messages aloud. For example, the processor applies knowledge of transport protocols, header syntax, and typical user-entered symbols and syntax. For details, see the *E-mail Pre-processor Developer's Guide*.

Strategies that require preprocessing

Application developers are encouraged to preprocess their input text before sending it to Speechify. Adding embedded tags and reformatting text are two examples of custom preprocessing, and additional opportunities are described below.

Available techniques

- ❑ **Text manipulation** – You can improve Speechify's output by manipulating punctuation, formatting, and other features of the input text. For details, see "Text manipulation" on [page 122](#).
- ❑ **SPRs** – You can customize word pronunciations by inserting Symbolic Phonetic Representations directly in the input text. This technique is particularly useful for homographs (see "Case study – homograph disambiguation" on [page 126](#)),

which require contextual disambiguation and thus cannot be handled with a user dictionary.

- ❑ Embedded tags – You can insert embedded tags to customize Speechify's text analysis. Example uses for these tags, which constitute a mark-up language, include the following:
 - Specify a spell-out mode
 - Determine the expansion of four-digit sequences as either years or quantities
 - Turn on a special mode for processing of postal addresses
 - Insert pauses

For details, see [Chapter 2](#) and the corresponding chapters of each language supplement.

- ❑ W3C SSML tags – Speechify supports input text marked with tags defined by the W3C Speech Synthesis Markup Language. W3C SSML tags serve the same general purpose as Speechify's embedded tags, though the specific kinds of control can differ. For details, see [Chapter 8](#).

Testing preprocessors

When developing a custom preprocessor, it is essential to test it thoroughly, as there may be unexpected interactions between it and Speechify's default text analysis. (See "Unanticipated results of text manipulation" on [page 124](#).)

Tests should include the following:

- ❑ When writing the preprocessor, the developer should create a suite of sample test sentences for each text element being manipulated. For example, if the processor performs substitutions on target words or phrases, the test sentences should include the targets in various grammatical positions and contexts.
- ❑ The suite should be submitted to the Speechify engine in its raw form (without using the preprocessor) as well as in its processed form, and the outputs should be compared.
- ❑ The suite should be integrated into an ongoing quality assurance program that collects and tests sample sentences from a range of applications. This broad testing assures that preprocessors have the ability to correctly handle future (and perhaps un-anticipated) text content.
- ❑ During application usability tests, developers should include scenarios that focus on areas addressed by any preprocessors. For example, if the application reads tabular information and there is a processor to improve the intelligibility of the data, then a test subject might be asked to listen to the data and perform some task.

- During partial deployments of applications, the project team should solicit quality evaluations from end-users.

Text manipulation

You can manipulate various aspects of the text to affect speech output including: spelling, punctuation, formatting. The section on homographs ([page 10-126](#)) discusses other ways (for example, re-wording the text) to preprocess the input text so that word pronunciations are more accurate.

General guidelines

Before writing a preprocessing script to manipulate text, you should test un-manipulated examples to verify whether preprocessing is needed. Speechify's default text analysis handles many kinds of text correctly.

When processing text, be cautious when manipulating individual words as opposed to complete phrases. Otherwise, the script might disguise information that the default text analysis would ordinarily use. For examples, see "Unanticipated results of text manipulation" on [page 124](#).

Spelling

Correct spelling may seem like an obvious requirement, but it must be emphasized that Speechify does not correct the spelling of your input text; it reads precisely what the text contains.

Historically, ScanSoft has noticed that a surprisingly large number of output errors are the result of incorrect orthography in the input text. In the sentence, "The weahter is expected to be cloudy this morning," a human will likely make an automatic correction to "weather," but Speechify will read the word as something like "wet her" or "wheat her."

Punctuation and phrasing

Appropriate phrasing is essential to the naturalness and intelligibility of the speech output. Human speakers naturally group words into meaningful phrases to provide cues to listeners about the organization of the utterance. Phrasing can even affect the meaning of a sentence, sometimes dramatically. Below, sentence (b) reveals far more disturbing information than sentence (a), although they differ only in phrasing:

- a. Thirty percent of teenagers who are regular smokers report decreased levels of physical activity.
- b. Thirty percent of teenagers, who are regular smokers, report decreased levels of physical activity.

Speechify employs a sophisticated set of rules to add phrase breaks intelligently to unpunctuated text and to suppress certain breaks in punctuated text. However, due to the complexities hinted at earlier in this chapter, Speechify cannot always predict phrases where a human reader would.

You can ensure more natural phrasing from Speechify by doing the following:

- ❑ Punctuating the input text where you want phrase breaks
- ❑ Omitting punctuation from locations where phrase breaks are inappropriate

You can also insert pauses into the speech output using Speechify's pause tag (see "Creating pauses" on [page 72](#)) or the W3C SSML or SAPI pause tags.

Formatting

Speechify is not sensitive to certain kinds of text formatting understood by human readers:

- ❑ Uppercase letters – Speechify does not apply emphasis to words spelled in uppercase letters.
- ❑ Tabs, spaces, and line endings – Speechify does not consider tabs, extra spaces, or line endings when analyzing text. (One exception: line endings are significant

in postal address mode; see the language supplements for details.) Consider this grocery shopping list:

```
Two dozen eggs
4 quarts of Milk
Orange Juice
Bread (Whole Grain)
```

Above, the list items will be spoken as one long phrase because there is no punctuation to separate them. This is not the intent of the author, but Speechify does not know that the text is a list, and Speechify would create new problems if it automatically paused at the end of every line. The best solution is for the application to insert punctuation at the end of each list item.

Visual formatting

Speechify does not consider bold text, italics, fonts, underscores, colors, or bulleted lists when normalizing text. If your text depends on visual formatting elements for its interpretation, you should rewrite it using techniques that are meaningful to Speechify, such as punctuation.

Unanticipated results of text manipulation

A custom preprocessor can interact with Speechify's default text analysis in unexpected ways. For this reason, the testing described on [page 10-121](#) is important for ensuring quality speech output.

The examples below illustrate the importance of preprocessing entire phrases or expressions and not just small pieces of text.

Given the original input text:

```
234 Pine St. W.
```

In address mode, Speechify handles the text without error. But in non-address mode, Speechify speaks “two hundred thirty four pine street double you.” In this case, the developer might (reasonably) preprocess the text by expanding “W.” to “West.” But this change has the unanticipated effect of confusing Speechify’s street/saint disambiguation:

```
Preprocessed input: "234 Pine St. West"  
Incorrect Speechify output: "234 Pine Saint West"
```

Aside from using address mode to get the correct results, the developer could preprocess both abbreviations:

```
New preprocessed input: "234 Pine Street West."
```

However, as the next example shows, there is no guarantee that processing larger phrases will solve all problems. The challenge of writing a preprocessor is to avoid disguising information that Speechify would otherwise use for its default text analysis. Consider the following telephone number:

```
Original text: 607-277-5439
```

By default, Speechify inserts the correct pauses:

```
Default normalization: 6 0 7, 2 7 7, 5 4 3 9
```

But of course, Speechify will not recognize the telephone number if a preprocessor changes the entire string:

```
Processed input: 6 0 7 2 7 7 5 4 3 9
```

The next example shows the importance of testing whether the input text actually needs to be manipulated with a preprocessor.

Given the original text:

```
12.30 am
```

An application developer might be concerned that Speechify will speak the word “am” instead of speaking “ay emm.” (In fact, Speechify speaks the text correctly.) If a script expands “am” to “ay emm” the processed input to Speechify would be:

```
12.30 ay emm
```

With the new input, Speechify does not recognize the text as a clock time. (When a clock time has a period delimiter, Speechify only recognizes it if there is a trailing “am” or “pm.”) As a result, Speechify incorrectly speaks:

```
twelve point three zero ay emm
```

Alternatively, the developer might be concerned that Speechify will not recognize the time correctly. In this case, the script processed input might be:

```
twelve thirty am
```

With this input, Speechify speaks the word “am” as a verb, rather than “ay emm.”

Aside from the preferred solution (doing no preprocessing), the only manipulation that works correctly is to change the entire phrase:

```
twelve thirty ay emm
```

Case study – homograph disambiguation

A *homograph* is a word that has different pronunciations for different meanings. For example, the word *moped* is pronounced one way when it means “a lightweight motorized bicycle”, and another way when it means “was dejected and apathetic.” Homographs can be full words, as in the *moped* example, or abbreviations, such as *Ct.*, which can be expanded to either *Court* or *Connecticut* and *Dr.*, which can be *Doctor* or *Drive*.

Homographs present a major challenge to a text-to-speech system's text analysis capabilities. Very often, context provides clues to the disambiguation of homographs. But in other cases homographs cannot be reliably differentiated without semantic or contextual information that is not currently available to state-of-the-art text-to-speech systems.

Consider a sentence like *I read the newspaper every day*. The sentence does not contain any information to determine whether the verb *read* is past tense (spoken as *red*) or present tense (spoken as *reed*). Human readers would differentiate the two based on the context – either by the surrounding sentences or by known features of the situation – but in difficult cases even humans may make mistakes or repeat a phrase to correct themselves.

The differentiating context may be present within the sentence, in which case a text-to-speech system might correctly predict the desired meaning. For example, the following sentence resolves the ambiguity in the previous example: *I listen to the news and read the newspaper every day*.

Contextual disambiguation can be extremely challenging, however, as in the case of the word *lead* in the following examples:

- a. The report revealed that we are exposing many students to lead in the school building.
- b. The report revealed that we are expecting many students to lead in the school election.

Thus, while Speechify uses a sophisticated set of linguistically-based, context-sensitive rules to determine the correct pronunciation in any given context, it is not possible to avoid mistakes altogether. If your application can preprocess the incoming text before sending it to Speechify, you can use one of the following techniques to elicit the correct pronunciation:

Reword the sentence – Speechify may be more successful with a slightly altered form of the sentence. This is mainly a matter of trial-and-error on the part of the application developer. Because Speechify's homograph disambiguation rules are specific to individual words, and in many cases quite complicated, a change of wording in one part of a sentence can resolve ambiguity in a homograph elsewhere.

Use an SPR – You can use an SPR to specify a pronunciation directly. For example, you could replace the word *lead* with SPRs as follows:

- a. The report revealed that we are exposing many students to \![Ed] in the school building.
- b. The report revealed that we are expecting many students to \![id] in the school election.

Expand ambiguous abbreviations – You can provide the fully expanded form of ambiguous abbreviations, such as *Court* or *Connecticut* rather than *Ct.*

Case study – navigation text

Consider the following text:

- 2: Turn LEFT onto MT VERNON ST. 0.26 miles
- 3: Turn RIGHT onto MASSACHUSETTS AVE/MASS AVE. 0.76 miles

A human might read the text as:

Two. Turn left onto Mount Vernon Street.
Zero point two six miles.

Three. Turn right onto Massachusetts Avenue.
Zero point seven six miles.

Speechify reads the text as:

Two. Turn left onto M T Vernon street.
Zero point two six miles three.
Turn right onto Massachusetts ave slash mass ave.
Zero point seven six miles

Problems caused by uppercase letters in the exaggeration:

- ❑ “M T” should be “Mount”
- ❑ “Ave” was pronounced “ah vey” as in “Ave Maria.” It should be “Avenue.”

Below, the example is changed by filtering the example to indiscriminately normalize the casing:

2: Turn Left onto Mt Vernon St. 0.26 miles
3: Turn Right onto Massachusetts Ave/Mass Ave. 0.76 miles

When the new text is entered, Speechify correctly expands all of the abbreviations:

- ❑ Mt becomes Mount
- ❑ Ave becomes Avenue

Problems caused by context errors.

- ❑ There is no pause between lines; line “three” merges with the distance “Zero point two six miles three.”
- ❑ “Massachusetts Ave” and “Mass Ave” (and the “slash”) are spoken.
- ❑ If we know the address is in Boston, and we know the conventions of the Boston area, we would expand “Ave” to “Avenue” if it is preceded by “Massachusetts” but not if preceded by “Mass.” Thus, we would speak either “Massachusetts Avenue” or “Mass Ave.”

Correcting the context errors requires real-world knowledge. The more known about the application's data, the more that can be done:

- ❑ Given a list of directions, terminate each line with a period to force a pause.

- ❑ Given a slash character that is used as a logical OR symbol, remove the second name. (Alternatively, we could add a pause followed by the parenthetical “also known as...” and another pause.)
- ❑ Given a list of geographically specific terms, we can load our dictionary so that “Mass Ave” is never expanded by the Speechify normalizer.

The final input text:

```
2: Turn left onto Mt. Vernon St. 0.26 miles.  
3: Turn right onto Massachusetts Ave., also known as Mass  
   Ave. 0.76 miles.
```

Speechify reads the text as:

```
Two. Turn left onto Mount Vernon Street.  
Zero point two six miles.
```

```
Three. Turn right onto Massachusetts Avenue,  
also known as Mass Ave. Zero point seven six miles.
```

Speech concatenation issues

Speechify uses a speech synthesis technology known as *concatenative synthesis*. Concatenative synthesis involves recording a large database of human speech and segmenting this database into smaller units of speech. To synthesize output speech, Speechify selects units from this database and joins them together to create new utterances.

When selecting units of speech, Speechify uses a complex algorithm that considers a wide variety of contextual information in an attempt to arrive at an optimal sequence. For example, the sound *k* is acoustically quite different in the words *keep* and *coop*. The difference is due to the acoustic properties of the following vowel (ee versus oo). Because the unit selection process is so complex, one cause of degraded speech quality is the selection of non-optimal units. If Speechify incorrectly selected the *k* from *keep* to synthesize the word *coop* the speech output would not sound natural.

Application developers cannot control the unit selection process, but because the selected units are highly context-dependent, you can improve the output by rewording the input (in addition to using embedded tags, dictionaries, and W3C SSML). In the re-worded text, many of the speech sounds will be in a different

context, which causes different units to be selected. This is an iterative, trial-and-error process. For example, if a word sounds bad, you might try changing the word itself or modifying adjacent words to see what produces the best result.

Logging generated pronunciations

If a word does not appear in a system dictionary or active user root dictionary, Speechify uses text-to-phoneme rules to generate its pronunciation. Because these rule-generated pronunciations may be less accurate than dictionary pronunciations (particularly in the case of highly idiosyncratic vocabulary, such as proper names and foreign words), you may wish to have words with rule-generated pronunciations logged. This can serve as a first step in identifying words that are not pronounced correctly, and which you may wish to add to a root dictionary.

You should bear in mind that the usefulness of logging generated pronunciations as a means of identifying words that may be mispronounced varies considerably from language to language. For example, in Spanish the relationship between letters and phonemes is extremely regular, so that most words are pronounced correctly even though their pronunciations are rule-generated (foreign words and names are exceptions in this regard). In English, with its highly complex and idiosyncratic relationship between letters and phonemes, the likelihood of error in a rule-generated pronunciation is higher, despite the sophistication of the text-to-phoneme rules.

You can toggle this logging with the `tts.log.event.generatedPronunciations` parameter in the Speechify XML configuration file (if event logging is turned on).

After each speak request, Speechify adds an event (PRON) to the event log for each word in the input text whose pronunciation was generated by text-to-phoneme rules. The event contains the word, the pronunciation (in SPR symbols), and the number of times the word occurred in the text. If a speak request contains more than one occurrence of the word, only one PRON event is logged. See “Description of events and tokens” on [page 36](#) for more information.

Note that the pronunciation appears in standard SPR tag format, representing the canonical pronunciation of the word as spoken in isolation (without phonological influences from adjacent words).



Reference Material

The chapters in this part cover these topics:

[Chapter 12 “API Reference”](#) describes the API functions.

[Chapter 13 “Configuration Files”](#) describes the format of the configuration files and the available parameters.



API Reference

This chapter describes the API functions in alphabetical order. All API function prototypes, types, error codes, and constants are located in the header file SWItts.h.

In This Chapter

- ❑ “Calling convention” on [page 134](#)
- ❑ “Server’s preferred character set” on [page 134](#)
- ❑ “Result codes” on [page 135](#)
- ❑ “SWIttsCallback()” on [page 137](#)
- ❑ “SWIttsClosePort()” on [page 143](#)
- ❑ “SWIttsDictionaryActivate()” on [page 144](#)
- ❑ “SWIttsDictionariesDeactivate()” on [page 146](#)
- ❑ “SWIttsDictionaryFree()” on [page 147](#)
- ❑ “SWIttsDictionaryLoad()” on [page 148](#)
- ❑ “SWIttsGetParameter()” on [page 151](#)
- ❑ “SWIttsInit()” on [page 154](#)
- ❑ “SWIttsOpenPortEx()” on [page 155](#)
- ❑ “SWIttsResourceAllocate()” on [page 157](#)
- ❑ “SWIttsResourceFree()” on [page 158](#)
- ❑ “SWIttsSetParameter()” on [page 159](#)
- ❑ “SWIttsSpeak()” on [page 161](#)
- ❑ “SWIttsSpeakEx()” on [page 164](#)
- ❑ “SWIttsStop()” on [page 167](#)
- ❑ “SWIttsTerm()” on [page 168](#)

Calling convention

The calling convention is dependent on the operating system, and is defined in the SWItts.h header file.

On Windows, all Speechify API functions use the stdcall (or Pascal) calling convention. The header files contain the appropriate compiler directives to ensure correct compilation. When writing callback functions, be sure to use the correct calling convention.

Under Windows:

```
#define SWIAPI __stdcall
```

Under UNIX:

```
#define SWIAPI
```

Server's preferred character set

The server's preferred character set is ISO-8859-1 (also known as Latin-1). All strings passed to the server by calls to SWIttsSpeak(), SWIttsSpeakEx(), and SWIttsDictionaryLoad() are converted to ISO-8859-1 before they are processed internally. Consequently, in Speechify 3.0, text entered into these functions must be representable in ISO-8859-1 even if it is encoded in another character set supported by Speechify. (Japanese is an exception, because it does not support ISO-8859-1. The preferred character set for the Japanese server is Shift-JIS.)

Result codes

The following result codes are defined in the enum `SWIttsResult` in `SWItts.h`.

Code	Description
<code>SWItts_ALREADY_EXECUTING_API</code>	This API function cannot be executed because another API function is in progress on this port on another thread.
<code>SWItts_ALREADY_INITIALIZED</code>	<code>SWIttsInit()</code> was called when the Speechify library was already initialized.
<code>SWItts_CONNECT_ERROR</code>	The client could not connect to the server.
<code>SWItts_DICTIONARY_ACTIVE</code>	The dictionary is active; it cannot be activated again or cannot be freed until deactivated.
<code>SWItts_DICTIONARY_LOADED</code>	Dictionary is already loaded.
<code>SWItts_DICTIONARY_NOT_LOADED</code>	Dictionary has not been loaded before attempting to activate it.
<code>SWItts_DICTIONARY_PARSE_ERROR</code>	Any error during dictionary parsing.
<code>SWItts_DICTIONARY_PRIORITY_ALREADY_EXISTS</code>	No duplicate priorities are allowed in dictionaries of the same type and language.
<code>SWItts_ERROR_PORT_ALREADY_STOPPING</code>	<code>SWIttsStop()</code> was called when the port was already in the process of stopping.
<code>SWItts_ERROR_STOP_NOT_SPEAKING</code>	<code>SWIttsStop()</code> was called when the port was not speaking.
<code>SWItts_FATAL_EXCEPTION</code>	(Windows only.) A crash occurred within the client library. A crash log named <code>ALTmon.dmp</code> might have been generated in the Speechify SDK directory. Please send it to Speechify technical support. This is an unrecoverable error and you should close the application.
<code>SWItts_HOST_NOT_FOUND</code>	Could not resolve the host name or IP address.
<code>SWItts_INVALID_MEDIATYPE</code>	Unsupported MIME content type for the dictionary format or speak text format.
<code>SWItts_INVALID_PARAMETER</code>	One of the parameters passed to the function was invalid.
<code>SWItts_INVALID_PORT</code>	The port handle passed is not a valid port handle.
<code>SWItts_INVALID_PRIORITY</code>	Dictionary priority value is out of range.
<code>SWItts_LICENSE_ALLOCATED</code>	A license has already been allocated for this port.
<code>SWItts_LICENSE_FREED</code>	A license has already been freed for this port.
<code>SWItts_MUST_BE_IDLE</code>	This API function can only be called if the TTS port is idle.
<code>SWItts_NO_LICENSE</code>	There are no purchased licenses available.
<code>SWItts_NO_MEMORY</code>	An attempt to allocate memory failed.
<code>SWItts_NO_MUTEX</code>	An attempt to create a new mutex failed.
<code>SWItts_NO_THREAD</code>	An attempt to create a new thread failed.
<code>SWItts_NOT_EXECUTING_API</code>	An internal error. Notify ScanSoft technical support if you see this result code.

Code	Description
SWItts_PORT_ALREADY_SHUT_DOWN	The port is already closed. You cannot invoke SWIttsClosePort() on a port that has been closed.
SWItts_PORT_ALREADY_SHUTTING_DOWN	SWIttsClosePort() was called when the port was already being closed.
SWItts_PORT_SHUTTING_DOWN	A command could not be executed because the port is shutting down.
SWItts_PROTOCOL_ERROR	An error in the client/server communication protocol occurred.
SWItts_READ_ONLY	SWIttsSetParameter() was called with a read-only parameter.
SWItts_SERVER_ERROR	An error occurred on the server.
SWItts_SOCKET_ERROR	A sockets error occurred.
SWItts_SSML_PARSE_ERROR	Could not parse SSML text.
SWItts_SUCCESS	The API function completed successfully.
SWItts_UNINITIALIZED	The TTS client is not initialized.
SWItts_UNKNOWN_CHARSET	Character set is invalid or unsupported.
SWItts_UNSUPPORTED	Feature is not supported.
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_URI_NOT_FOUND or SWItts_URI_TIMEOUT.
SWItts_URI_NOT_FOUND	URI was not found: the file does not exist or the web server does not have a matching URI.
SWItts_URI_TIMEOUT	Timeout during web server URI access.
SWItts_WINSOCK_FAILED	WinSock initialization failed. (Windows only.)

This is the full set of codes that the API functions return. No functions return all the codes. SWItts_SUCCESS and SWItts_FATAL_EXCEPTION are the only codes that are common to all functions. All functions except SWIttsInit() return SWItts_UNINITIALIZED if SWIttsInit() was not the first function called.

SWIttsCallback()

Mode: Synchronous. **IMPORTANT:** You must not block/wait in this function.

User-supplied handler for data returned by the synthesis server.

```
typedef SWIttsResult (SWIAPI SWIttsCallback) (
    SWIttsPort ttsPort,
    SWItts_cbStatus status,
    void *data,
    void *userData
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx() or SWITTS_INVALID_PORT (-1) if the callback is called from within SWIttsInit(), SWIttsOpenPortEx(), or SWIttsTerm().
status	These are enumerated types that are used to inform the callback function of the status of the void *data variable. See table below.
data	Pointer to a structure containing data generated by the server. This pointer is declared as void * because the exact type varies. The status parameter indicates the exact type to which this pointer should be cast.
userData	This is a void * in which the application programmer may include any information that he wishes to be passed back to the callback function. A typical example is a thread ID that is meaningful to the application. The userData variable is a value you pass to these functions: <ul style="list-style-type: none"> ❑ SWIttsInit() for errors during SWIttsInit() ❑ SWIttsTerm() for errors during SWIttsTerm() ❑ SWIttsOpenPortEx() otherwise

This table lists the values of SWItts_cbStatus:

Status code	Description
SWItts_cbAudio	Audio data packet. The data structure is a SWIttsAudioPacket shown below.
SWItts_cbBookmark	User-defined bookmark. The data structure is a SWIttsBookMark as shown below.
SWItts_cbDiagnostic	Diagnostic message. The data structure is a SWIttsMessagePacket as shown below. You only receive this message if the SWITTSLOGDIAG environment variable is defined. See “Speechify Logging” on page 27 for more information about logging.
SWItts_cbEnd	End of audio packets from the current SWIttsSpeak() command. The data is a NULL pointer.

Status code	Description
SWItts_cbError	<p>Asynchronous error message. This message is received if an asynchronous API function encounters an error when trying to perform an asynchronous operation such as reading from the network. If you receive this message, consider it fatal for that port. You are free to call SWItts functions from the callback but you should consider the receipt of SWItts_cbError fatal and call SWIttsClosePort() to properly clean up the port. This event is always preceeded with a SWItts_cbLogError event that indicates the error reason.</p> <p>The ttsPort argument is SWItts_INVALID_PORT (-1) and the userData argument could be NULL if the failure occurred during SWIttsInit(), SWIttsOpenPortEx(), or SWIttsTerm(). Make sure you check for these possibilities before your code dereferences userData or uses the port number for a lookup.</p>
SWItts_cbLogError	<p>Error message. The data structure is a SWIttsMessagePacket which contains error information, and is described below. The callback may receive the cbLogError and cbDiagnostic events at anytime, whether inside a synchronous or asynchronous function. The user is not allowed to call any SWItts function at this time. If you receive this message, log it to a file, console, etc., and continue execution.</p> <p>The ttsPort argument is SWItts_INVALID_PORT (-1) and the userData argument could be NULL if the failure occurred during SWIttsInit(), SWIttsOpenPortEx(), or SWIttsTerm(). Make sure you check for these possibilities before your code dereferences userData or uses the port number for a lookup.</p>
SWItts_cbPhonememark	Represents information about a phoneme boundary in the input text. The data structure is a SWIttsPhonemeMark shown below.
SWItts_cbPortClosed	The port was successfully closed after a call to SWIttsClosePort(). The data is a NULL pointer.
SWItts_cbStart	Represents the commencement of audio packets from the current SWIttsSpeak() command. The data is a NULL pointer.
SWItts_cbStopped	SWIttsStop() has been called and recognized. There is no SWItts_cbEnd notification. The data is a NULL pointer.
SWItts_cbWordmark	Represents information about a word boundary in the input text. The data structure is a SWIttsWordMark shown below

Structures

The audio packet data structure is described here:

```
typedef struct {
    void *samples;
    unsigned int numBytes;
    unsigned int firstSampleNumber;
} SWIttsAudioPacket;
```

Member	Description
samples	The buffer of speech samples. You must copy the data out of this buffer before the callback returns as the client library may free it or overwrite the contents with new samples.
numBytes	The number of bytes in the buffer. This number of bytes may be larger than the number of samples, e.g., if you've chosen a sample format of 16-bit linear, the number of bytes would be twice the number of samples.
firstSampleNumber	The accumulated number of samples in the current SWIttsSpeak() call. The first packet has a sample number of zero.

The message packet data structure is described here:

```
typedef struct {
    time_t messageTime;
    unsigned short messageTimeMs;
    unsigned int msgID;
    unsigned int numKeys;
    const wchar_t **infoKeys;
    const wchar_t **infoValues;
    const wchar_t *defaultMessage;
} SWIttsMessagePacket;
```

Member	Description
messageTime	The absolute time at which the message was generated.
messageTimeMs	An adjustment to messageTime to allow millisecond accuracy.
msgID	An unique identifier. A value of 0 is used for SWItts_cbDiagnostic messages.
numKeys	The number of key/value pairs (the number of entries in the infoKeys and infoValues arrays). For SWItts_cbLogDiagnostic messages, this is always 0. For SWItts_cbLogError messages, this may be 0 or greater.
infoKeys/infoValues	Additional information about message, in key/value pairs of 0-terminated wide character string text. These members are only valid for SWItts_cbLogError messages.
defaultMessage	A pre-formatted 0-terminated wide character message. This member is only valid for SWItts_cbDiagnostic messages.

See “Speechify Logging” on [page 27](#) for information about how to map msgID into a meaningful message.

The bookmark data structure is described here:

```
typedef struct {  
    const wchar_t *ID;  
    unsigned int sampleNumber;  
} SWIttsBookMark;
```

Member	Description
ID	A pointer to the bookmark 0-terminated wide character string. It corresponds to the user-defined string specified in the bookmark tag.
sampleNumber	The bookmark location, specified by an accumulated number of samples for the current SWIttsSpeak() call. A bookmark placed at the beginning of a string has a timestamp of 0. The sampleNumber always refers to a sample number in the future (that has not yet been received).

The wordmark data structure is described here:

```
typedef struct {  
    unsigned int sampleNumber;  
    unsigned int offset;  
    unsigned int length;  
} SWIttsWordMark;
```

Member	Description
sampleNumber	The sample number correlating to the beginning of this word. The sampleNumber always refers to a sample number in the future (that has not yet been received).
offset	The index into the input text of the first character where this word begins. Starts at zero.
length	The length of the word in characters not bytes.

The phoneme-mark data structure is described here:

```
typedef struct {
    unsigned int sampleNumber;
    const char *name;
    unsigned int duration;
    unsigned int stress;
} SWIttsPhonemeMark;
```

Member	Description
sampleNumber	The sample number correlating to the beginning of this phoneme. The sampleNumber always refers to a sample number in the future (that has not yet been received).
name	The name of the phoneme as a NULL-terminated US-ASCII string. (The phoneme names are described in the Speechify supplements for each language.)
duration	The length of the phoneme in samples.
stress	Indicates whether the phoneme was stressed or not. A 0 indicates no stress, a 1 indicates primary stress, and a 2 indicates secondary stress.

Notes

The callback function is user-defined but is called by the SWItts library, i.e., the user writes the code for the callback function, and a pointer to it is passed into the SWIttsOpenPortEx() function. The client calls this function as needed when data arrives from the Speechify server. It is called from a thread created for the port during the SWIttsOpenPortEx() function.

The SWItts_cbStatus variable indicates the reason for invoking the callback and also what, if any, type of data is being passed. The SWIttsResult code returned by the callback is not currently interpreted by Speechify, but may be in the future, thus the callback function should always return SWItts_SUCCESS.



NOTE

Because the callback function is user-defined, the efficiency of its code has a direct impact on system performance – if it is inefficient, it may hinder the client's ability to service the network connection and data may be lost.

The Speechify server throttles its transmission of audio data to the client at two times real-time. Even with this throttling, this means that sending a large amount of text to the SWIttsSpeak() function may cause the server to send back a large amount of audio before the application needs to send it to an audio device or telephony card.

On average, expect about one second of audio for every ten characters of text input. For example, if you pass 10 KB of text to the `SWIttsSpeak()` function, your callback may receive about 1000 seconds of audio samples. That is 8 MB of data if you chose to receive 8-bit μ -law samples and 16 MB of data if you chose to receive 16-bit linear samples. This amount of text may require more buffering than you want to allow for, especially in a scenario with multiple TTS ports.

A common technique to avoid a buffering load is to divide your input text into pieces, sending a new speak request for one piece when the previous is finished playing. For best results, you should divide the text at sentence boundaries such as periods.

SWIttsClosePort()

Mode: Asynchronous

Closes a TTS port which frees all resources and closes all communication with the server.

```
SWIttsResult SWIAPI SWIttsClosePort (
    SWIttsPort ttsPort
);
```

Parameter	Description
ttsPort	Port handle returned by SWIttsOpenPortEx().

After closing, SWIttsClosePort() sends a SWItts_cbPortClosed message to the callback upon successful closing of the port. Once a port is closed, you cannot pass that port handle to any SWItts function.

See also

“SWIttsOpenPortEx()” on [page 155](#)

SWIttsDictionaryActivate()

Mode: Synchronous

Activate a dictionary for subsequent SWIttsSpeak() requests.

```
SWIttsResult SWIAPI SWIttsDictionaryActivate (
    SWIttsPort ttsPort,
    const SWIttsDictionaryData *dictionary,
    unsigned int priority
);
```

Parameter	Description
ttsPort	Port handle returned by SWIttsOpenPortEx().
dictionary	Object containing the URI and fetch parameters, or a string.
priority	Priority to assign to this dictionary compared to other active dictionaries. Values: Integers 1–2 ³¹ . Lowest priority: 1.

See SWIttsDictionaryLoad() for the specification of the SWIttsDictionaryData data structure.

Applications must use SWIttsDictionaryLoad() to load a dictionary before activating it.

Activating the dictionary never triggers a reload of the dictionary. To refresh a loaded dictionary that may be changed, call SWIttsDictionaryFree() followed by SWIttsDictionaryLoad(), and then activate the dictionary. See “SWIttsDictionaryLoad()” on [page 148](#) for more information.

If you want dictionaries to be active for a speak request, load and activate them before calling SWIttsSpeak(). (SWIttsSpeak() does not require any activated dictionaries.) Once activated, dictionaries are active until they are explicitly deactivated. More than one dictionary can be activated at any given time. When you are finished using all dictionaries, call SWIttsDictionariesDeactivate() and then SWIttsDictionaryFree() to clean up the resources.

The dictionary priority is a unique integer ranking the priority of this dictionary compared to all other activated dictionaries of the same language and type. During speak requests, the engine performs a lookup in the dictionary of the appropriate type with the highest priority. If the lookup fails, it tries the dictionary of the appropriate type with the next highest priority, until there are no more dictionaries of that type to try.

SWIttsDictionaryActivate() may return the following error codes:

Error code	Problem
SWItts_DICTIONARY_ACTIVE	The dictionary is already activated.
SWItts_DICTIONARY_NOT_LOADED	The dictionary is not loaded.
SWItts_DICTIONARY_PRIORITY_ALREADY_EXISTS	No duplicate priorities are allowed in dictionaries of the same type and language.
SWItts_INVALID_PARAMETER	The ttsPort or dictionary parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active on this port.

See also

“SWIttsDictionariesDeactivate()” on [page 146](#)

“SWIttsDictionaryFree()” on [page 147](#)

“SWIttsDictionaryLoad()” on [page 148](#)

SWIttsDictionariesDeactivate()

Mode: Synchronous

Deactivates all activated dictionaries for subsequent speak requests.

```
SWIttsResult SWIAPI SWIttsDictionariesDeactivate (
    SWIttsPort ttsPort
);
```

Parameter	Description
ttsPort	Port handle returned by SWIttsOpenPortEx().

When you are finished using a dictionary, call SWIttsDictionaryFree() to clean up the resources used by the dictionary data.

SWIttsDictionariesDeactivate() does not deactivate the default dictionaries that are configured in the Speechify server configuration file.

Active dictionaries remain active until they are explicitly deactivated by SWIttsDictionariesDeactivate(). They are not automatically deactivated after each speak request. SWIttsDictionariesDeactivate() deactivates all active dictionaries. There is no way to deactivate individual dictionaries. To deactivate only some of the currently active dictionaries, use this function to deactivate all dictionaries, then re-activate the desired dictionaries with SWIttsDictionaryActivate().

SWIttsDictionariesDeactivate() may return the following error codes:

Error code	Problem
SWItts_INVALID_PARAMETER	The ttsPort parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active on this port.

See also

“SWIttsDictionaryActivate()” on [page 144](#)

“SWIttsDictionaryFree()” on [page 147](#)

“SWIttsDictionaryLoad()” on [page 148](#)

SWIttsDictionaryFree()

Mode: Synchronous

Signals the engine that the dictionary is no longer needed.

```
SWIttsResult SWIAPI SWIttsDictionaryFree (
    SWIttsPort ttsPort,
    const SWIttsDictionaryData *dictionary
);
```

Parameter	Description
ttsPort	Port handle returned by SWIttsOpenPortEx().
dictionary	Object containing the URI and fetch parameters, or a string.

When you are finished using a dictionary, call SWIttsDictionaryFree() to clean up the resources used by the dictionary data.

SWIttsDictionaryFree() *cannot* be used to free the default dictionaries that are configured in the Speechify server configuration file.

SWIttsDictionaryFree() may return the following error codes:

Error code	Problem
SWItts_DICTIONARY_ACTIVE	Dictionary cannot be freed until deactivated.
SWItts_DICTIONARY_NOT_LOADED	Dictionary is not loaded.
SWItts_INVALID_PARAMETER	The ttsPort or dictionary parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active on this port.

See also

“SWIttsDictionaryActivate()” on [page 144](#)
 “SWIttsDictionariesDeactivate()” on [page 146](#)
 “SWIttsDictionaryLoad()” on [page 148](#)

SWIttsDictionaryLoad()

Mode: Synchronous

Load a complete dictionary from a URI or string to prepare it for future activation.

```
SWIttsResult SWIAPI SWIttsDictionaryLoad (
    SWIttsPort ttsPort,
    const SWIttsDictionaryData *dictionary
);
```

Parameter	Description
ttsPort	Port handle returned by SWIttsOpenPortEx().
dictionary	Object containing the URI and fetch parameters, or a string.

A dictionary must be loaded before it can be activated. If you want the engine to apply dictionaries to text passed in the SWIttsSpeak() and SWIttsSpeakEx() functions, you must load and activate them before calling SWIttsSpeak() or SWIttsSpeakEx().

SWIttsDictionaryLoad() blocks until dictionary loading and parsing is complete.

The SWIttsDictionaryData structure is defined as follows:

```
typedef struct SWIttsDictionaryData {
    unsigned int version;
    const char *uri;
    const unsigned char *data;
    unsigned int lengthBytes;
    const char *contentType;
    const VXIMap *fetchProperties;
    VXIMap *fetchCookieJar;
} SWIttsDictionaryData;
```

Field	Description
version	Use the constant SWItts_CURRENT_VERSION, defined in SWItts.h.
uri	URI to the dictionary content; <i>contentType</i> may be NULL. Pass NULL when <i>data</i> is non-NULL. The URI may be one of the following: <ul style="list-style-type: none">❑ HTTP/1.1 web server access, where the URL is fetched by the Speechify server: http://myserver/mydict.xml❑ Simple file access, where the file is resolved on the Speechify server, for example: file:/users/mydict.xml /users/mydict.xml

Field	Description
<code>data</code>	In-memory dictionary content; <i>contentType</i> must be non-NULL. Pass NULL when <i>uri</i> is non-NULL.
<code>lengthBytes</code>	Length of the in-memory dictionary content in bytes. Pass 0 when <i>uri</i> is non-NULL.
<code>contentType</code>	MIME content type to identify the dictionary format. One of the following: <ul style="list-style-type: none"> ❑ NULL: only valid when type is "uri". Automatically determines the content type from the URI. For http: URIs, the MIME content type returned by the web server is processed using the rules that follow. For file: URIs, files with a .xml extension are treated as SpeechWorks XML lexicon format dictionaries, otherwise an error results. ❑ application/octet-stream: assume a SpeechWorks XML lexicon format dictionary (this is the default MIME content type returned by web servers for unknown data types) ❑ text/xml: assume a SpeechWorks XML lexicon format dictionary ❑ application/x-swi-dictionary: SpeechWorks XML lexicon format dictionary ❑ Other: a SWItts_INVALID_MEDIATYPE error is returned
<code>fetchProperties</code>	Optional VXIMap to control Internet fetch operations (particularly the base URI and fetch timeouts). May be NULL to use defaults. These settings apply to the fetch of the dictionary when <i>uri</i> is non-NULL.
<code>fetchCookieJar</code>	Optional VXIMap to provide session or end-user-specific cookies for Internet fetch operations, modified to return the updated cookie state on success. May be NULL to disable cookies (web server cookies are refused).

If an application asks `SWIttsDictionaryLoad()` to load a dictionary that is already loaded, Speechify returns the non-fatal error code `SWItts_DICTIONARY_LOADED`. To refresh Speechify's copy of a dictionary that has been updated or changed elsewhere, call `SWIttsDictionaryFree()` then `SWIttsDictionaryLoad()` to force Speechify to reload the dictionary.

`SWIttsDictionaryLoad()` may return the following error codes:

Error code	Problem
<code>SWItts_DICTIONARY_LOADED</code>	Dictionary is already loaded.
<code>SWItts_DICTIONARY_PARSE_ERROR</code>	Any error during dictionary parsing.
<code>SWItts_INVALID_MEDIATYPE</code>	Unsupported MIME content type for the dictionary format.
<code>SWItts_INVALID_PARAMETER</code>	The <code>ttsPort</code> or dictionary parameter is NULL or invalid.
<code>SWItts_MUST_BE_IDLE</code>	A speak operation is active on this port.
<code>SWItts_UNKNOWN_CHARSET</code>	Character set for the dictionary is invalid or unsupported.

Error code	Problem
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_URI_NOT_FOUND or SWItts_URI_TIMEOUT.
SWItts_URI_NOT_FOUND	URI was not found (file does not exist or the web server does not have a matching URI).
SWItts_URI_TIMEOUT	Timeout during web server URI access.

See also

“SWIttsDictionaryActivate()” on [page 144](#)
“SWIttsDictionariesDeactivate()” on [page 146](#)
“SWIttsDictionaryFree()” on [page 147](#)

SWIttsGetParameter()

Mode: Synchronous

Retrieves the value of a parameter from the server.

```
SWIttsResult SWIAPI SWIttsGetParameter (
    SWIttsPort ttsPort,
    const char *name,
    char *value
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
name	The name of the parameter to retrieve.
value	Takes a preallocated buffer of size SWITTS_MAXVAL_SIZE.

The following table describes the parameters that can be retrieved. All parameters have a default value from the server XML configuration file, and certain parameters are read-only.

Name	Possible values	Read-only	Description
tts.audio.packetsize	Even number 64–102400	no	Maximum size of the audio packets, in bytes, that the Speechify client extracts from the server connection and sends to the user supplied callback function. (All packets will be this size except the last packet, which may be smaller.)
tts.audio.rate	33–300	no	Port-specific speaking rate of the synthesized text as a percentage of the default rate.
tts.audio.volume	0–100 (See description for caveat.)	no	Port-specific volume of synthesized speech as a percentage of the default volume: 100 means maximum possible without distortion and 0 means silence. Values greater than 100 are permitted, but output might have distortion.
tts.audioformat.encoding	ulaw, alaw, linear	yes	Encoding method for audio generated during synthesis. This value can be set via the mimetype.

Name	Possible values	Read-only	Description
tts.audioformat.mimetype	audio/basic audio/x-alaw-basic audio/L16;rate=8000 audio/L16;rate=16000	no ^a	<p>The audio format of the server:</p> <ul style="list-style-type: none"> ❑ audio/basic corresponds to 8 kHz, 8-bit μ-law; ❑ audio/x-alaw-basic corresponds to 8 kHz, 8-bit A-law; ❑ audio/L16;rate=8000 corresponds to 8 kHz, 16-bit linear; ❑ audio/L16;rate=16000 corresponds to 16 kHz, 16-bit linear. <p>All other values generate a SWItts_INVALID_PARAM return code.</p> <p>In all cases, audio data is returned in network byte order.</p>
tts.audioformat.samplerate	8000, 16000	yes	Audio sampling rate in Hz.
tts.audioformat.width	8, 16	yes	<p>This value can be set via the mimetype.</p> <p>Size of individual audio sample in bits.</p> <p>This value can be set via the mimetype.</p>
tts.client.version	Current Speechify version number	yes	<p>The returned value is a string of the form major.minor.maintenance. For example, 3.0.0 or 3.0.1.</p> <p>This parameter reflects the client version, and can be retrieved after SWIttsInit() is called but before SWIttsOpenPortEx() is called. Use SWITTS_INVALID_PORT for the first argument to SWIttsGetParameter().</p>
tts.engine.id	positive integer	yes	Speechify server logical channel ID that is handling speak requests for this client port. The Speechify server logs diagnostics, errors, and events to the server diagnostic and error log using this logical channel ID, so including this logical channel ID in application logs can help in cross-referencing application and Speechify server logs.
tts.engine.version	Current Speechify server version number	yes	The returned value is a string of the form major.minor.maintenance. For example, 3.0.0 or 3.0.1.
tts.marks.phoneme	true, false	no	Controls whether phoneme marks are reported to the client.
tts.marks.word	true, false	no	Controls whether wordmarks are reported to the client.
tts.network.timeout	positive integer	no	Timeout, in seconds, for the connection to the Speechify client. If a send operation to the client fails to complete within this duration, or if a heartbeat is not received from a client in this duration, the client connection is presumed to be dead and the connection is dropped.

Name	Possible values	Read-only	Description
tts.product.name	Speechify, Speechify Solo	yes	"Speechify" for the main Speechify product; "Speechify Solo" for the Speechify Solo product.
tts.server.licensingMode	default, explicit	yes	Modes for controlling license allocation to a Speechify port object: <ul style="list-style-type: none"> ❑ default: Automatically when SWIttsOpenPortEx() is called ❑ explicit: As decided by the platform developer. Use SWIttsResourceAllocate() and SWIttsResourceFree() to control allocation and de-allocation of licenses.
tts.voice.gender	male, female	yes	Synthesis voice gender.
tts.voice.language	server	yes	Synthesis language.
tts.voice.name	server	yes	Unique name identifying the voice.
tts.voice.version	Current Speechify server voice version number	yes	The returned value is a string of the form major.minor.maintenance.voicebuild. For example, 3.0.0.0 or 3.0.0.1.

- a. tts.audioformat.mimetype values may be switched between audio/basic, audio/x-alaw-basic, and audio/L16;rate=8000 if the server has been instantiated with the 8 kHz voice database. If the server is instantiated with the 16 kHz voice database, this parameter has the read-only value of audio/L16;rate=16000

See also

"SWIttsSetParameter()" on [page 159](#)

SWIttsInit()

Mode: Synchronous

Initializes the client library so that it is ready to open ports.

```
SWIttsResult SWIAPI SWIttsInit (
    SWIttsCallback *callback,
    SWIttsCallback *userData
);
```

Parameter	Description
callback	A pointer to a callback function that may receive SWItts_cbLogError and/or SWItts_cbDiagnostic messages during the SWIttsInit() call. If this callback is called, the ttsPort parameter is -1. This may be the same callback that is passed to SWIttsOpenPortEx() or SWIttsTerm().
userData	User information passed back to callback. It is not interpreted or modified in any way by the client.



NOTE

This must be the first API function called, and it should only be called once per process, not once per call to SWIttsOpenPortEx().

SWIttsInit() may return the following error codes:

Error code	Problem
SWItts_ALREADY_INITIALIZED	The Speechify library is already initialized (from a prior SWIttsInit() call).

See also

“SWIttsOpenPortEx()” on [page 155](#)

“SWIttsTerm()” on [page 168](#)

SWIttsOpenPortEx()

Mode: Synchronous

This function opens and connects to a Speechify server port. Call this function after `SWIttsInit()`.

```
SWIttsOpenPortEx(
    SWIttsPort *ttsPort,
    const char *parameters,
    SWIttsResources *resources,
    SWIttsCallback *callback,
    void *userdata
);
```

Parameter	Description
ttsPort	Address of a location to place the new port's handle.
parameters	Key/value parameter list in <key1>=<value1>;<key2>=<value2>[...] form. For the Speechify client, the keys are "hostname" and "hostport."
resources	Reserved for future use, pass NULL.
callback	A pointer to a callback function that receives audio buffers and other notifications when the server sends data to the client. If an error occurs during the call to <code>SWIttsOpenPortEx()</code> , the callback is called with a <code>SWItts_cbLogError</code> message and a <code>ttsPort</code> of -1.
userData	User information passed back to callback.

Notes

`SWIttsOpenPortEx()` may return the following error codes:

Error code	Problem
<code>SWItts_INVALID_PARAMETER</code>	One of the parameters passed to the function was invalid.
<code>SWItts_NO_LICENSE</code>	There are no purchased licenses available.

Example

Below is an example of how you would use this API function:

```
SWIttsPort port;
SWIttsOpenPortEx(&port, "hostname=localhost;hostport=5555",
    NULL, callback, NULL);
```

See also

“SWIttsClosePort()” on [page 143](#)

“SWIttsInit()” on [page 154](#)

SWIttsResourceAllocate()

Explicitly assign a license for a specified Speechify port.

```
SWIttsResult SWIttsResourceAllocate (
    SWIttsPort ttsPort,
    const wchar_t *feature,
    void *reserved
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
feature	Use the constant SWItts_LICENSE_SPEAK defined in SWItts.h for licensing functionality.
reserved	This parameter is reserved for future use. Pass in NULL.

Notes

The tts.server.licensingMode configuration parameter must be set to “explicit” for SWIttsResourceAllocate() to work. You can use SWIttsGetParameter() to retrieve the value of tts.server.licensingMode and find out whether you need to call this function (and explicitly allocate and free licenses) or not. If the licensing mode is set to “default,” the SWIttsOpenPortEx() function implicitly allocates a license for the application.

See the *SpeechWorks Licensing Handbook* for more information on licensing.

SWIttsResourceAllocate() may return the following error codes:

Error code	Problem
SWItts_INVALID_PARAMETER	An invalid feature parameter was specified.
SWItts_LICENSE_ALLOCATED	A license has already been allocated for this port.
SWItts_MUST_BE_IDLE	A speak operation is active.
SWItts_NO_LICENSE	There are no purchased licenses available.
SWItts_UNSUPPORTED	The tts.server.licensingMode parameter is not set to explicit.

See also

“SWIttsResourceFree()” on [page 158](#)

SWIttsResourceFree()

Explicitly free the user license for the specified Speechify port.

```
SWIttsResult SWIttsResourceFree (
    SWIttsPort ttsPort,
    const wchar_t *feature,
    void *reserved
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
feature	Use SWItts_LICENSE_SPEAK to free a license.
reserved	This parameter is reserved for future use. Pass in NULL.

Notes

The tts.server.licensingMode configuration parameter must be set to explicit for SWIttsResourceFree() to work.

Note that SWIttsClosePort() also frees the license for a port while freeing all other resources for that port.

SWIttsResourceFree() may return the following error codes:

Error code	Problem
SWItts_INVALID_PARAMETER	An invalid feature parameter was specified.
SWItts_LICENSE_FREED	A license has already been freed for this port.
SWItts_MUST_BE_IDLE	A speak operation is active.
SWItts_NO_LICENSE	There are no purchased licenses available.
SWItts_UNSUPPORTED	The tts.server.licensingMode parameter is not set to explicit.

See the *SpeechWorks Licensing Handbook* for more information on licensing.

See also

“SWIttsClosePort()” on [page 143](#)
“SWIttsResourceAllocate()” on [page 157](#)

SWIttsSetParameter()

Mode: Synchronous

Sends a parameter to the server.

```
SWIttsResult SWIAPI SWIttsSetParameter (
    SWIttsPort ttsPort,
    const char *name,
    const char *value
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
name	A parameter name represented as a NULL-terminated US-ASCII string.
value	A parameter value represented as a NULL-terminated US-ASCII string.

Notes

If SWIttsSetParameter() returns an error, the parameter is not changed. Setting a parameter is not a global operation, it only affects the TTS port passed to the call.

The following table describes the parameters that can be set. All parameters have a default value from the server XML configuration file. SWIttsGetParameter() lists the read-only parameters. If you try to set a read-only parameter, SWIttsSetParameter() returns SWItts_READ_ONLY.

Name	Possible values	Description
tts.audio.packetsize	Even number 64–102400 Recommended values: 1024, 2048, or 4096	Maximum size of the audio packets, in bytes, that the Speechify client extracts from the server connection and sends to the user supplied callback function. (All packets are this size except the last packet, which may be smaller.)
tts.audio.rate	33–300	Port-specific speaking rate of the synthesized text as a percentage of the default rate.
tts.audio.volume	0–100 (See description for caveat.)	Port-specific volume of synthesized speech as a percentage of the default volume: 100 means maximum possible without distortion and 0 means silence. Values greater than 100 are permitted, but output might have distortion.

Name	Possible values	Description
tts.audioformat.mimetype	audio/basic audio/x-alaw-basic audio/L16;rate=8000 audio/L16;rate=16000 ^a	<p>The audio format of the server:</p> <ul style="list-style-type: none">❑ audio/basic corresponds to 8 kHz, 8-bit μ-law;❑ audio/x-alaw-basic corresponds to 8 kHz, 8-bit A-law;❑ audio/L16;rate=8000 corresponds to 8 kHz, 16-bit linear;❑ audio/L16;rate=16000 corresponds to 16 kHz, 16-bit linear. <p>All other values generate a SWItts_INVALID_PARAM return code.</p> <p>In all cases, audio data is returned in network byte order.</p>
tts.marks.phoneme	true, false	Controls whether phoneme marks are reported to the client.
tts.marks.word	true, false	Controls whether wordmarks are reported to the client.
tts.network.timeout	positive integer	Timeout, in seconds, for the connection to the Speechify client. If a send operation to the client fails to complete within this duration, or if a heartbeat is not received from a client in this duration, the client connection is presumed to be dead and the connection is dropped.
tts.reset	none	Command which causes all parameters controllable via SWIttsSetParameter() to revert to their default values; the value is ignored.

a. tts.audioformat.mimetype values may be switched between audio/basic, audio/x-alaw-basic, and audio/L16;rate=8000 if the server has been instantiated with the 8 kHz voice database. If the server is instantiated with the 16 kHz voice database, this parameter has the read-only value of audio/L16;rate=16000

See also

“SWIttsGetParameter()” on [page 151](#)

SWIttsSpeak()

Mode: Asynchronous

Sends a text string to be synthesized. Call this function for every text string to synthesize.

```
SWIttsResult SWIAPI SWIttsSpeak (
    SWIttsPort ttsPort,
    const unsigned char *text,
    unsigned int lengthBytes,
    const char *content_type
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
text	The text to be synthesized: an array of bytes representing a string in a given character set.
lengthBytes	The length of the text array in bytes; note that this means any NULL in the text is treated as just another character.
content_type	Description of the input text according to the MIME standard (per RFC-2045 Sec. 5.1 and RFC 2046). Default (if set to NULL): text/plain;charset=iso-8859-1.

Notes

See SWIttsSpeakEx() to do either of these tasks:

- ❑ Have the Speechify server fetch the document to speak from a web server (instead of the text being in memory)
- ❑ Specify internet fetch controls for <audio> elements within W3C SSML documents

The content types that are supported are text/* and application/synthesis+ssml.

Any subtype may be used with "text". However, only the subtype "xml" is treated specially: the text is assumed to be in W3C SSML and if it is not, an error is returned. All other subtypes are treated as "plain".

The application/synthesis+ssml content type is used to indicate W3C SSML content, which is parsed accordingly. If W3C SSML input is not signaled via the content type parameter, it is pronounced as plain text.

The only meaningful `content_type` parameter is “charset,” which is case-insensitive. (See www.iana.org/assignments/character-sets for more details.) All other parameters are ignored. If “charset” is not specified, it is assumed to be ISO-8859-1. An example of a valid content type:

```
text/plain;charset=iso-8859-1
```

The following charsets are supported for Western languages:

- ❑ ISO-8859-1 (default)
- ❑ US-ASCII (synonym: ASCII)
- ❑ UTF-8
- ❑ UTF-16
- ❑ `wchar_t`

The “`wchar_t`” character set is not a MIME standard. It indicates that the input is in the form of the client platform’s native wide character array (i.e., `wchar_t*`). Note that input length must still be specified in bytes (i.e., the number of wide characters in the input times the number of bytes per wide character).

The supported charsets vary for Asian languages. For example, Japanese does not support ISO-8859-1 (and thus, `content_type` becomes required). Japanese supports:

- ❑ UTF-8
- ❑ UTF-16
- ❑ EUC
- ❑ Shift-JIS
- ❑ `wchar_t`

Also for Japanese, the text-to-speech engine ignores any white space in the input text. For example, this allows correct processing when text begins on one line and is continued on the next. A result of this behavior, which you might not anticipate, is that numeric values separated by spaces, tabs, or returns are pronounced as single units: “1 2 3”, “1<tab>2<tab>3”, and “1<return>2<return>3” are all pronounced as “123”. To speak the digits individually, separate them with commas: “1,2,3”.

Note that for Speechify Japanese, the default byte order is set to big endian when the byte order mark is missing.

If text contains byte sequences that do not represent valid characters when converted to the server’s preferred character set, the server still performs the synthesis. It does this by removing the invalid characters and speaking the remaining text.

`SWIttsSpeak()` may return the following error codes:

Error code	Problem
<code>SWItts_NO_LICENSE</code>	There are no purchased licenses available.

See also

"SWIttsSpeakEx()" on [page 164](#)

"SWIttsStop()" on [page 167](#)

SWIttsSpeakEx()

Mode: Asynchronous

Sends a text URI or string to be synthesized. Call this function for every text URI or string to synthesize.

```
SWIttsResult SWIAPI SWIttsSpeakEx (  
    SWIttsPort ttsPort,  
    const SWIttsSpeakData *speakData  
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().
speakData	Object containing the URI and fetch parameters, or a string.

Notes

The SWIttsSpeakData structure is defined as follows:

```
typedef struct SWIttsSpeakData {  
    unsigned int version;  
    const char *uri;  
    const unsigned char *data;  
    unsigned int lengthBytes;  
    const char *contentType;  
    const VXIMap *fetchProperties;  
    VXIMap *fetchCookieJar;  
} SWIttsSpeakData;
```

Field	Description
version	Use the constant SWItts_CURRENT_VERSION defined in SWItts.h.
uri	URI to the text; <i>contentType</i> may be NULL. Pass NULL when <i>data</i> is non-NULL. The URI may be one of the following: <ul style="list-style-type: none">❑ HTTP/1.1 web server access, where the URL is fetched by the Speechify server: http://myserver/mytext.txt❑ Simple file access, where the file is resolved on the Speechify server, for example: file:/users/mytext.txt /users/mytext.txt
data	In-memory text; <i>contentType</i> must be non-NULL. Pass NULL when <i>uri</i> is non-NULL.
lengthBytes	Length of the in-memory text in bytes. Pass 0 when <i>uri</i> is non-NULL.

Field	Description
contentType	<p>MIME content type to identify the text format. One of the following:</p> <ul style="list-style-type: none"> ❑ NULL: only valid when type is "uri". Automatically determines the content type from the URI. For http:// URIs, the MIME content type returned by the web server is processed using the rules that follow. For file: URIs, files with a .xml extension are treated as W3C SSML documents, and files with a .txt extension are treated as ISO-8859-1 text documents, otherwise an error results. ❑ text/*: Subtype xml text is assumed to be in W3C SSML. If it is not, an error is returned. Other subtypes are treated as "plain." ❑ application/synthesis+ssml: indicates W3C SSML content. If W3C SSML input is not indicated via contentType, it is pronounced as plain text. ❑ Other: a SWItts_INVALID_MEDIATYPE error is returned
fetchProperties	<p>Optional VXIMap to control Internet fetch operations (particularly the base URI and fetch timeouts). May be NULL to use defaults. These settings apply to the fetch of the top-level document when <i>uri</i> is non-NULL, and also to any fetches for <audio> elements within W3C SSML documents (whether the W3C SSML document was fetched by URI or provided in-memory).</p>
fetchCookieJar	<p>Optional VXIMap to provide session or end-user-specific cookies for Internet fetch operations, modified to return the updated cookie state on success. May be NULL to disable cookies (web server cookies are refused).</p>

The only meaningful contentType parameter is "charset," which is case-insensitive. (See www.iana.org/assignments/character-sets for more details.) All other parameters are ignored. If "charset" is not specified, it is assumed to be ISO-8859-1. An example of a valid contentType:

```
text/plain; charset=iso-8859-1
```

The following charsets are supported for Western languages:

- ❑ ISO-8859-1 (default)
- ❑ US-ASCII (synonym: ASCII)
- ❑ UTF-8
- ❑ UTF-16
- ❑ wchar_t

The "wchar_t" character set is not a MIME standard. It indicates that the input is in the form of the client platform's native wide character array (i.e., wchar_t*). Note that input length must still be specified in bytes (i.e., the number of wide characters in the input times the number of bytes per wide character).

The supported charsets vary for Asian languages. For example, Japanese does not support ISO-8859-1 (and thus, content_type becomes required). Japanese supports:

- ❑ UTF-8

- ❑ UTF-16
- ❑ EUC
- ❑ Shift-JIS
- ❑ wchar_t

Also for Japanese, the text-to-speech engine ignores any white space in the input text. For example, this allows correct processing when text begins on one line and is continued on the next. A result of this behavior, which you might not anticipate, is that numeric values separated by spaces, tabs, or returns are pronounced as single units: “1 2 3”, “1<tab>2<tab>3”, and “1<return>2<return>3” are all pronounced as “123”. To speak the digits individually, separate them with commas: “1,2,3”.

Note that for Speechify Japanese, the default byte order is set to big endian when the byte order mark is missing.

If *text* contains byte sequences that do not represent valid characters when converted to the server's preferred character set, the server still performs the synthesis. It does this by removing the invalid characters and speaking the remaining text.

SWIttsSpeakEx() may return the following error codes:

Error code	Problem
SWItts_INVALID_MEDIATYPE	Unsupported MIME content type for the speak data.
SWItts_INVALID_PARAMETER	The ttsPort or speakData parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active.
SWItts_NO_LICENSE	There are no purchased licenses available.
SWItts_UNKNOWN_CHARSET	Character set for speakData is invalid or unsupported.
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_URI_NOT_FOUND and SWItts_URI_TIMEOUT.
SWItts_URI_NOT_FOUND	URI was not found (file does not exist or the web server does not have a matching URI).
SWItts_URI_TIMEOUT	Timeout during web server URI access.

See also

“SWIttsStop()” on [page 167](#)

SWIttsStop()

Mode: Asynchronous

Interrupts a call to SWIttsSpeak().

```
SWIttsResult SWIAPI SWIttsStop (
    SWIttsPort ttsPort
);
```

Parameter	Description
ttsPort	The port handle returned by SWIttsOpenPortEx().

Notes

When the currently active speak request is completely stopped and the port is idle, the SWItts library calls the port's callback with a status of SWItts_cbStopped. The callback is called with SWItts_cbStopped only if the SWIttsStop() function returns with a SWItts_SUCCESS result.

If there is no SWIttsSpeak() function in progress, or if a currently active speak request is already stopping due to a previous call to SWIttsStop(), this function returns an error.

See also

“SWIttsSpeak()” on [page 161](#)
“SWIttsSpeakEx()” on [page 164](#)

SWIttsTerm()

Mode: Synchronous

Closes all ports, terminates their respective threads, shuts down the API library, and cleans up memory usage.

```
SWIttsResult SWIAPI SWIttsTerm(  
    SWIttsCallback *callback,  
    void *userData  
);
```

Parameter	Description
callback	A pointer to a callback function that may receive SWItts_cbError, SWItts_cbLogError, and/or SWItts_cbDiagnostic messages during the SWIttsTerm() call.
userData	User information passed back to callback.

Notes

If SWIttsTerm() closes one or more open TTS ports, you receive SWItts_cbPortClosed messages in their respective callbacks.

See also

“SWIttsInit()” on [page 154](#)



Configuration Files

Speechify uses XML configuration files to configure the server. A default system-wide configuration file is included with Speechify, and defines basics such as the Internet fetch proxy server and the default audio format. Each installed voice includes a voice configuration file. A voice configuration file overrides settings in the system-wide configuration file such as voice name and sampling rate.

Configuration files are loaded from the Speechify server command line and modified by hand or, on Windows, with the Speechify MMC.

This chapter describes the Speechify server configuration file format and each parameter in detail. It also provides some examples of editing the configuration file.

For information on how configuration files are loaded, see “Using configuration files” on [page 24](#). For information on using the Speechify MMC to edit certain parameters, see “Using the MMC snap-in” on [page 19](#).

In This Chapter

- ❑ “Configuration file format” on [page 170](#)
- ❑ “Modifying a configuration file” on [page 174](#)
- ❑ “Configuration parameters” on [page 175](#)
 - “Audio parameters” on [page 175](#)
 - “Cache parameters” on [page 176](#)
 - “Engine parameters” on [page 177](#)
 - “Environment variable parameters” on [page 178](#)
 - “Internet fetch parameters” on [page 179](#)
 - “Log parameters” on [page 180](#)
 - “Marks parameters” on [page 182](#)
 - “Network parameters” on [page 182](#)
 - “Preprocessor parameters” on [page 183](#)

- “Server parameters” on [page 183](#)
- “SSML parameters” on [page 185](#)
- “Text format parameters” on [page 185](#)
- “Voice parameters” on [page 186](#)

Configuration file format

The format of a configuration file is XML that lists parameter/value pairs in language blocks. Here is a sample configuration file followed by a description of the elements and attributes:

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="SWIttsConfig.xsl"?>
<!DOCTYPE SWIttsConfig PUBLIC "-//SpeechWorks//DTD SPEECHIFY
  CONFIG 1.0//EN" "SWIttsConfig.dtd">

<SWIttsConfig version="1.0.0">
  <lang name="Default">
    <param name="tts.server.port">
      <value> 5555 </value>
    </param>
  </lang>
</SWIttsConfig>
```

The sample configuration file consists of the following parts:

❑ XML declaration:

```
<?xml version="1.0" ?>
```

❑ Document type declaration:

```
<!DOCTYPE SWIttsConfig PUBLIC "-//SpeechWorks//DTD
  SPEECHIFY CONFIG 1.0//EN" "SWIttsConfig.dtd">
```

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

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

❑ The root element of the document (as specified in the document type declaration), i.e., the container element for <lang> elements:

```
<SWIttsConfig version="1.0.0">
```


- ❑ One or more `<lang>` elements containing `<param>` elements which, in turn, contain `<value>` or `<namedValue>` elements.

```
<lang name="Default">
  <param name="tts.server.port">
    <value> 5555 </value>
  </param>
</lang>
```

The `<lang>` element

The `<lang>` element is used to support language-specific parameter values. The `<lang>` element has one required attribute, "name."

The name attribute

The value of the name attribute is a language in ISO form (e.g., "en-US") or "Default." The "Default" value is used to specify language independent parameter values.

When the Speechify server starts, it retrieves the `tts.voice.language` parameter inside the `<lang name="Default">` element to determine the active language. Speechify then fetches all parameter values from the `<lang>` element whose name value matches that of the active language. It falls back to the value in the default `<lang>` element if the language-specific parameter value does not exist.



NOTE

In almost all cases, you only need to modify the default `<lang>` element. The only real use for multiple `<lang>` blocks is a Speechify installation on a server that hosts multiple languages each of which is customized to load language-specific dictionaries via the `tts.engine.dictionaries` parameter.

The <param> element

The <param> element is used to specify a parameter along with its value. The <param> element has one required attribute, name

The name attribute

The value of the name attribute is a configuration parameter. Most of the configuration parameters have the following format:

```
tts.[subsystem].[value_name]
```

[subsystem] is a logical subsystem name specified in all lowercase letters, for example, "voice," "network," and "log.event."

[value_name] is the name of the value within the subsystem. It is specified in mixed case, starting with a lowercase letter and using capital letters to separate words. Example value names include "fileMimeType," "reportErrorText," and "phoneme."

Other parameter names do not follow the above format, but match a corresponding environment variable. Example environment variable parameter names include TMPDIR and USER.

The <value> element

The <value> element is used to specify a value for a parameter. It may only appear within a <param> element, and there may only be one <value> with each <param> element. It has no attributes. The element data may be empty for some parameters. The data may include references to environment variables and/or other parameters by enclosing the environment variables or parameters in \${}. For example:

```
<param name="tts.cache.dir">  
  <value> ${SWITTSSDK}/cache_${tts.voice.name}_  
    ${tts.voice.format} </value>  
</param>
```

This example states that the Internet fetch cache is located in the Speechify installation area (SWITTSSDK environment variable) in a directory named after the current voice name and voice format, e.g., C:\Program Files\Speechify\cache_tom_8.

Variables are resolved at the time they are needed, rather than at configuration file load time. This simplifies configuration using overlaid configuration files by avoiding order dependencies: there is no need to re-specify variables such as the cache directory in a voice-specific configuration files merely due to the voice name changing.

The <namedValue> element

The <namedValue> element is used to specify a list of values for a parameter. Each <namedValue> element has a required name attribute, whose value specified a key name. The data in the <namedValue> element is handled exactly the same as <value> element data. Here is a typical entry:

```
<param name="tts.inet.extensionRules">
  <namedValue name=".alaw"> audio/x-alaw-basic </namedValue>
  <namedValue name=".ulaw"> audio/basic </namedValue>
  <namedValue name=".wav"> audio/x-wav </namedValue>
  <namedValue name=".L16"> audio/L16;rate=8000 </namedValue>
  <namedValue name=".txt"> text/plain </namedValue>
  <namedValue name=".xml"> text/xml </namedValue>
  <namedValue name=".ssml"> application/synthesis+ssml
  </namedValue>
</param>
```

In this case, <param> is used as before. However, a series of one or more <namedValue> elements may be used within <param> instead of using a single <value> element. Each <namedValue> has one required attribute, name, which specifies a key name. The data in the <namedValue> element is the value of the key/value pair, and is handled exactly the same as <value> element data.

Modifying a configuration file

To modify a configuration file, load it in a text or XML editor. You can add a new `<language>` element or change existing `<value>` or `<namedValue>` elements.

For example, to change the `tts.network.timeout` value from 60 seconds to 120 seconds for all Speechify servers regardless of language, change the `tts.network.timeout` `<param>` element in the default `<lang>` element from:

```
<param name="tts.network.timeout">
  <value> 60 </value>
</param>
```

to:

```
<param name="tts.network.timeout">
  <value> 120 </value>
</param>
```

To automatically load and activate two French dictionaries at startup, add a French `<lang>` element containing a `tts.engine.dictionaries` `<param>` element. Within the `tts.engine.dictionaries` `<param>` element, add two `<namedValue>` elements, one for each dictionary:

```
<lang name="fr-FR">
  <param name="tts.engine.dictionaries">
    <namedValue name="1"> c:\dicts\globalDict.xml </
    namedValue>
    <namedValue name="5"> c:\dicts\localDict.xml </
    namedValue>
  </param>
</lang>
```

To override the value of the `SWITTSSDK` environment variable, uncomment the `SWITTSSDK` `<param>` element and add the path to the `<value>` element:

```
<!-- <param name="SWITTSSDK">
  <value> </value>
</param> -->
```

becomes:

```
<param name="SWITTSSDK">
  <value> c:\mydir </value>
</param>
```

Configuration parameters



NOTE

Some parameters can be set and retrieved through the API functions “SWIttsSetParameter()” on [page 159](#) and “SWIttsGetParameter()” on [page 151](#).

Audio parameters

Parameter	Description
tts.audio.packetsize	Maximum size of the audio packets, in bytes, that the Speechify client extracts from the server connection and sends to the user supplied callback function. (All packets are this size except the last packet, which may be smaller.) May be overridden on a port basis via SWIttsSetParameter(). Possible values: any even number 64–102400. Recommended values: <ul style="list-style-type: none">❑ 1024❑ 2048❑ 4096 Default: 4096
tts.audio.rate	Speaking rate of the synthesized text as a percentage of the default rate. May be overridden on a port basis via SWIttsSetParameter(). Must be in the range 33–300. Default: 100

Parameter	Description
tts.audio.volume	Volume of synthesized speech as a percentage of the default volume: 100 means maximum possible without distortion and 0 means silence. May be overridden on a port basis via <code>SWIttsSetParameter()</code> . Must be in the range 0–100. Default: 100
tts.audioformat.mimetype.8kHz tts.audioformat.mimetype.16kHz	Audio format of the server, where the 8kHz version is used if <code>tts.voice.format</code> is set to 8, and the 16kHz version is used if <code>tts.voice.format</code> is set to 16. May be overridden on a port basis via <code>SWIttsSetParameter()</code> : <ul style="list-style-type: none"> ❑ audio/basic: 8 kHz, 8-bit μ-law ❑ audio/x-alaw-basic: 8 kHz, 8-bit A-law ❑ audio/L16;rate=8000: 8 kHz, 16-bit linear ❑ audio/L16;rate=1600: 16 kHz, 16-bit linear In all cases, audio data is returned in network byte order. Defaults to audio/basic for 8 kHz server configurations, and audio/L16;rate=16000 for 16 kHz server configurations.

Cache parameters

Parameter	Description
tts.cache.dir	Cache directory name, used for caching fetched audio. Must be unique for each server instance on a host. Default: <code>\${TMPDIR}/cache_\${tts.voice.name}_\${tts.voice.format}_\${USER}</code>
tts.cache.entryExpTimeSec	Maximum amount of time any individual cache entry remains in the cache, in seconds. If set is too small, cache entries are discarded too soon, reducing system performance. If set too large, cache entries may accumulate rapidly for applications that use dynamic URLs, resulting in the system running out of memory and/or disk space. The optimal value depends on the application and amount of disk space and RAM available on the system. Default: 3600 (1 hour)

Parameter	Description
tts.cache.entryMaxSizeMB	Maximum size of any individual cache entry, in megabytes. Default: 200
tts.cache.totalSizeMB	Maximum size of the data in the cache directory, in megabytes. Default: 200

Engine parameters

Parameter	Description
tts.engine.dictionaryDefaultPriorityBase	If the priority is omitted from the dictionary tag, use this base priority. The first dictionary specified in a speak request without a priority is loaded with this base value. The next dictionary specified without a priority is loaded with priority base+1, etc. See “Specifying user dictionaries” on page 75 and “SWIttsDictionaryActivate()” on page 144 .
tts.engine.dictionaries	<p>Optional. List of dictionaries for the server to automatically load and activate at startup. These dictionaries are never deactivated by SWIttsDictionariesDeactivate(), and cannot be freed by SWIttsDictionaryFree().</p> <p>If tts.engine.dictionaries exists, it must contain at least one <namedValue> element. The name attribute of the <namedValue> element must specify the priority of the dictionary, and the data must be a URI specifying the dictionary to load. (See “SWIttsDictionaryActivate()” on page 144 for details on priorities.)</p> <p>If one or more of the URIs cannot be found, the server logs an error but proceeds to start. Other errors cause startup to fail.</p> <p>Default: empty list</p>

Environment variable parameters

Environment variable parameters are used to override environment variable settings in the configuration file. If the corresponding environment variable is not set or the parameter is not specified in the configuration file, Speechify supplies a default value.

Parameter	Value
SWITTSSDK	The directory where the Speechify SDK is installed. On Windows this defaults to the directory that contains the directory where Speechify server is located. On UNIX, this defaults to /usr/local/SpeechWorks/Speechify.
TMPDIR	Temporary directory that is the default destination for server output files (used as the root path for several tts.log[...] parameters and the tts.cache.dir parameter). On Windows this defaults to the temporary directory used by the Windows operating system. On UNIX this defaults to /tmp.
USER	Current username, used in conjunction with TMPDIR, the voice name, and the language name to configure server output files in order to help eliminate conflicts when multiple server instances are run on a single host. Defaults to the effective username for the process as returned by the operating system.

Internet fetch parameters

Parameter	Description
tts.inet.acceptCookies	<p>True to accept cookies if the user called SWIttsSpeakEx() with a cookie jar. False to reject all cookies and ignore any cookie jar arguments.</p> <p>Default: true</p>
tts.inet.extensionRules	<p>Rules for mapping file name extensions to MIME content types, specified as a sequence of <namedValue> elements where the name attribute is the extension and the value is the MIME content type.</p> <p>Used to determine the audio format for audio files accessed via the W3C SSML <audio> element, and used to determine the content type for documents fetched during SWIttsDictionaryLoad() or SWIttsSpeakEx().</p> <p>Default:</p> <ul style="list-style-type: none"> <input type="checkbox"/> .alaw=audio/x-alaw-basic <input type="checkbox"/> .ulaw=audio/basic <input type="checkbox"/> .wav=audio/x-wav <input type="checkbox"/> .L16=audio/L16;rate=8000 <input type="checkbox"/> .txt=text/plain <input type="checkbox"/> .xml=text/xml <input type="checkbox"/> .ssml=application/synthesis+ssml
tts.inet.proxyPort	<p>Port number for accessing the HTTP proxy server. See tts.inet.proxyServer below.</p> <p>Default: 1234</p>
tts.inet.proxyServer	<p>Name of a HTTP proxy server to use for HTTP access, sometimes used by customers for security or firewall purposes. Set the server name or IP address, or leave this empty to disable using a proxy server.</p> <p>Default: empty (no proxy server used)</p>
tts.inet.userAgent	<p>HTTP user agent name sent in all HTTP requests. (Do not edit this parameter.)</p>

Log parameters

Parameter	Description
tts.log.contentDir	Directory used for writing binary diagnostic or event logging data as individual files, such as large blocks of input text for speak requests when certain diagnostics or configuration parameters are enabled. Must be unique for each server instance on a host. Default: \${TMPDIR}/SWIttsContent_\${tts.voice.name}_\${tts.voice.format}_\${USER}
tts.log.diagnostic.errorMapFile	XML error map file used to lookup the error text for reported error numbers, specified as a file path. Default: \${SWITTSSDK}/doc/SpeechifyErrors.en-US.xml
tts.log.diagnostic.file	Controls whether diagnostic and error messages are written to a text file, specified as the path of the diagnostic and error log file, or with an empty value to disable logging to a text file. Must be unique for each server instance on a host. Default: \${TMPDIR}/SWIttsDiagnosticLog_\${tts.voice.name}_\${tts.voice.format}_\${USER}.txt
tts.log.diagnostic.fileMaxSizeMB	Controls the maximum size of the diagnostic and error log file specified by tts.log.diagnostic.file before it rolls over (renames the existing log to have a ".old" suffixed, then opens a new diagnostic log file). Specified in megabytes. Default: 50 (MB)
tts.log.diagnostic.fileMimeType	Controls the format of the diagnostic and error log file specified by tts.log.diagnostic.file. <ul style="list-style-type: none">❑ text/plain;charset=UTF-8: Unicode UTF-8❑ text/plain;charset=ISO-8859-1: 8-bit Latin-1 Default: text/plain;charset=UTF-8
tts.log.diagnostic.reportErrorText	Controls whether errors are reported by error number alone (set to false), or by error number with the accompanying error text from the XML error map file (true). Default: true
tts.log.diagnostic.toStdout	Controls whether diagnostic and error messages are reported to the console (standard output), true or false. Default: true

Parameter	Description
tts.log.diagnostic.toSystemLog	Controls whether error messages are reported to the system logging service, true or false. On Unix this is syslog. On Windows, this is the Windows event log. Default: true
tts.log.event.file	Controls whether an event log file is written that can be used for reporting system usage. Specified as a file path, or with an empty value to disable event reporting. Must be unique for each server instance on a host. Default: empty value (event reporting is disabled)
tts.log.event.generatedPronunciations	Controls whether a pronunciation generated via text-to-phoneme rules is logged in the event log. Event logging must be enabled. See "Logging generated pronunciations" on page 130 for more information. Values are true or false. Default: false
tts.log.event.fileMaxSizeMB	Controls the maximum size of the event log file specified by tts.log.event.file before it rolls over (renames the existing log to have a ".old" suffix, then opens a new diagnostic log file). Specified in megabytes. Default: 50
tts.log.event.fileMimeType	Controls the format of the event log file specified by tts.log.event.file. <ul style="list-style-type: none"> ❑ text/plain;charset=UTF-8: Unicode UTF-8 ❑ text/plain;charset=ISO-8859-1: 8-bit Latin-1 Default: text/plain;charset=UTF-8

Marks parameters

Parameter	Description
tts.marks.phoneme	Controls whether phoneme marks are reported to the client. May be overridden on a port basis via <code>SWIttsSetParameter()</code> . Values are true or false. Default: false
tts.marks.word	Controls whether word marks are reported to the client, may be overridden on a port basis via <code>SWIttsSetParameter()</code> . Values are true or false. Default: false

Network parameters

Parameter	Description
tts.network.timeout	Timeout, in seconds, for the connection to the Speechify client. If a send operation to the client fails to complete within this duration, or if a heartbeat is not received from a client in this duration, the client connection is presumed to be dead and the connection is dropped. This value may be overridden on a port-by-port basis through <code>SWIttsSetParameter()</code> . Default: 60

Preprocessor parameters

Parameter	Description
tts.preprocessor.map128ToEuroInISO8859-1	Controls whether the character code 128 is mapped to the euro character for ISO-8859-1 input text. If it is, the euro character gets spoken, otherwise this character is discarded (treated as a space). Default: true

Server parameters

Parameter	Description
tts.server.licenseServerList	License servers to contact to acquire licenses, specified as a list where the <namedValue> "name" attribute values are ignored and the values of each <namedValue> indicates the host and port number in <port>@<host> format. In many cases, only a single license server is configured. If multiple license servers are configured, Speechify looks for valid Speechify licenses on each of the servers in the order they are listed, going to the next server in the list only if valid licenses could not be obtained from the previous server (due to that server being down, or being out of licenses, or some other error). Default: 27000@localhost
tts.server.licensingMode	Modes for controlling license allocation to a Speechify port object: <ul style="list-style-type: none"> ❑ default: Automatically when SWIttsOpenPortEx() is called ❑ explicit: As decided by the platform developer. Use SWIttsResourceAllocate() and SWIttsResourceFree() to control allocation and de-allocation of licenses. Default: default

Parameter	Description
tts.server.networkInterface	<p>Instructs Speechify to only listen for connections on a specific network interface. This can be used for security purposes, such as ensuring the server only accepts connections originating from a network interface that is only accessible via a secure LAN. For example, set this parameter to 127.0.0.1 to make the Speechify server accept connections only from Speechify clients that are running on the same host.</p> <p>By default this parameter is not set, resulting in the server accepting connections from all network interfaces.</p>
tts.server.numPorts	<p>Number of Speechify client connections to support (number of ports opened via <code>SWIttsOpenPortEx()</code> against this server instance).</p> <p>Default: 100</p>
tts.server.port	<p>Specifies the sockets port where the Speechify server should listen for incoming connections. If you are running multiple instances of Speechify on one server machine, this number should be different for each instance. Here the term “port” refers to a sockets (networking) port, not to a single connection created between the client and the server by the <code>SWIttsOpenPortEx()</code> function; you can have multiple client connections to the same sockets port. This affects the parameters you pass to <code>SWIttsOpenPortEx()</code>: any change to the port number must be reflected in the <code>connectionPort</code> parameter you pass to <code>SWIttsOpenPortEx()</code>. (See “<code>SWIttsOpenPortEx()</code>” on page 155 for more details about this function.)</p> <p>Default: 5555</p>

SSML parameters

Parameter	Description
tts.ssml.doStrictValidation	Controls whether strict W3C SSML parsing is done, where the W3C SSML DTD is used to ensure the SSML document only contains valid elements and attributes. Use false for the Speechify 2.x behavior where unknown elements and attributes are merely ignored. Default: true
tts.ssml.failOnAudioFetchError	Controls whether a W3C SSML parse fails if there are <audio> elements with no fallback where the audio source cannot be fetched. Use false for the Speechify server to log an error but otherwise proceed as if the <audio> element were not present. Use true for SWIttsSpeak() or SWIttsSpeakEx() to fail with a SSML parse error code, with no audio generated. Default: false

Text format parameters

Parameter	Description
tts.textformat.mimetype	MIME content type to use for speak requests when the content_type parameter to SWIttsSpeak() is NULL. See “SWIttsSpeakEx()” on page 164 for a list of supported content types. Default: text/plain;charset=ISO-8859-1

Voice parameters

Parameter	Description
tts.voice.dir	Specifies the directory where Speechify can find the voice database. Default: \${SWITTSSDK}, the value of the SWITTSSDK environment variable which is set during installation.
tts.voice.format	Specifies the audio format of the voice database that Speechify loads. <ul style="list-style-type: none">❑ 8: 8 kHz, 8-bit μ-law❑ 16: 16 kHz, 16-bit linear Default: 8
tts.voice.language	Specifies the language of the voice database that Speechify loads. The value consists of a 2-letter language identifier, a hyphen, and a 2-letter country code. For example: <ul style="list-style-type: none">❑ en-US: US English❑ fr-FR: Parisian French Default: en-US
tts.voice.name	Specifies the name of the voice that Speechify loads. Default: tom



Appendices and Index

The chapters in this part cover these topics:

[Appendix A “SAPI 5”](#) describes Speechify’s compliance with the SAPI 5 interface.

[Appendix B “Frequently Asked Questions”](#) answers frequently asked questions about using Speechify.

[Index](#)



SAPI 5

The SAPI 5 interface for Speechify is compliant with the Microsoft SAPI 5 text-to-speech specification. Although certain SAPI 5 features are not currently supported, those most commonly used by SAPI 5 developers are.

In This Appendix

- ❑ “Compliance” on [page 189](#)
- ❑ “SAPI voice properties” on [page 191](#)

Compliance

The SAPI 5 interface for Speechify:

1. Provides an in-process COM server implementing the following interfaces:

- IspTTSEngine
- IspObjectWithToken

If you use the SAPI IspVoice interface to manipulate a specific voice, ScanSoft does not support the following methods:

- Skip

In doing so, Speechify fails the following SAPI Compliance Tool tests:

- TTS Compliance Test\ISpTTSEngine\Skip
- TTS Compliance Test\Real Time Rate/Vol Tests\Real Time Rate Change
- TTS Compliance Test\Real Time Rate/Vol Tests\Real Time Volume Change

2. Supports the following SAPI-defined speak flags:

Speak flags	Action
SPF_DEFAULT	This is the default speak flag. The engine renders text to the output side under normal conditions.
SPF_PURGEBEFORESPEAK	The engine can stop rendering when passed an <i>abort</i> request.
SPF_IS_XML	The engine correctly interprets and acts upon a number of SAPI XML tags (defined below).
SPF_ASYNC	The engine uses SAPI 5 thread management to support the asynchronous synthesis of audio output without blocking the calling application.

3. Supports the following output formats natively, depending on the voice selected:

Output formats	Definition
SPSF_CCITT_uLaw_8kHzMono	Single channel 8 kHz μ -law with an 8-bit sample size
SPSF_CCITT_ALaw_8kHzMono	Single channel 8 kHz A-law with an 8-bit sample size
SPSF_8kHz16BitMono	Single channel 8 kHz linear PCM with a 16-bit sample size
SPSF_16kHz16BitMono	Single channel 16 kHz linear PCM with a 16-bit sample size

By default, Speechify SAPI 5 voices return audio to SAPI in these formats:

- Speechify 8kHz: SPSF_8kHz16BitMono
- Speechify 16kHz: SPSF_16kHz16BitMono

To set an audio format in the table to be Speechify's native engine format (before instantiating that voice), use the registry to set the `Attributes\MimeType` subkey of the appropriate voice key to one of Speechify's MIME types. (The MIME types are documented in "`SWIttsSetParameter()`" on [page 159](#).) If `MimeType` is not present or is set to an empty string, Speechify voices use their default native formats.

Thus, you can set a default in the registry (e.g., if you're using an 8 kHz voice and want μ -law instead of 8 kHz linear PCM), which is equivalent to using `SWIttsSetParameter()` to set `tts.audioformat.mimetype` for each port. SAPI has no equivalent of `SWIttsSetParameter()`; setting parameters is combined with creating an instance.

Note that SAPI 5 can convert to other audio formats not listed in this table.

4. Supports SAPI 5's User and Application Lexicons.

SAPI 5 has a separate client/application-specific dictionary called the Lexicon. *Speechify for SAPI 5* interacts with the SAPI lexicon allowing clients to override the default word pronunciation of the Speechify Server. Speechify ignores the user-supplied part-of-speech field in the Lexicon.

5. Supports the following SAPI XML tags. All other SAPI XML tags are ignored by Speechify:

- <SILENCE>
- <PRON>
- <BOOKMARK>
- <SPELL>
- <RATE>
- <VOLUME>

By ignoring other XML tags, Speechify fails the following tests in SAPI Compliance Tool:

- TTS Compliance Test\TTS XML Markup\Pitch (uses <PITCH> tag)
- Features\PartOfSp (uses <PARTOFSP> tag)

6. Synthesizes the following SAPI events:

Bookmark names beginning with “!SWI” are now reserved for internal SAPI processing and cannot be used as names of user-provided SAPI bookmarks.

- SPEI_TTS_BOOKMARK
- SPEI_WORD_BOUNDARY
- SPEI_PHONEME
- SPEI_VISEME (for US English only)

7. Supports the instantiation and simultaneous use of multiple TTS Engine instances within a single client process.

SAPI voice properties

To invoke the properties dialog box, use Control Panel >> Speech and click the Text To Speech tab. From the Voice Selection list, choose a Speechify voice. (You may see a message about not being able to initialize the voice. Ignore this message; it is

innocuous.) Once the Speechify voice is selected, the button labeled **Settings...** should be enabled. Click this button to open the Speechify SAPI Voice Properties dialog box.

The properties dialog box displays the voice name at the top of the box. The next section displays attributes of the voice including the language, gender, age, and the native audio format of the server. None of these values are editable. Although SAPI can provide an application with almost any audio format it requests, the application should request audio in the voice's native audio format for the best audio quality.

The next section of the dialog box lets you configure the Speechify server's address and port number. You can select either **Host name** and enter a server's host name or select **Host IP** and enter a server's IP address. Below those fields, enter the server's port number.

The final section of the dialog box is for diagnostic logging. If you experience problems with Speechify's SAPI 5 interface and require technical support, ScanSoft may ask you to turn on diagnostic logging within the interface. Unless required for technical support, you should leave logging turned off to avoid unnecessary CPU and disk activity. To turn on diagnostic logging, click the arrows to set the logging level:

- ☐ 0: logging is turned off
- ☐ 1: log high-level information
- ☐ 2: log verbose information

When logging is turned on, information is logged to the Speechify /bin directory, in a file named Speechify-SAPI5.log.

When you are done modifying the settings in the Voice Properties dialog box, click OK to accept the changes. The changes go into effect the next time an engine instance is created.



Frequently Asked Questions

This appendix answers frequently asked questions about using Speechify.

- ❑ **Question types:** “Why is the output the same, whether I end a sentence with a question mark or a period?”
- ❑ **Changing rate or volume:** “Can the application change the speaking rate or volume of the audio output while Speechify is synthesizing?”
- ❑ **Rewinding and fast-forwarding:** “Can the application move forward or backward in the audio output (e.g. go back 10 seconds)?”
- ❑ **Java interface:** “Does Speechify or Speechify Solo offer a Java interface?”
- ❑ **E-mail preprocessor and SAPI 5:** “Is it possible to use the e-mail preprocessor with SAPI 5?”
- ❑ **W3C SSML and SAPI 5:** “How can I use input text containing W3C SSML with SAPI 5?”
- ❑ **Error codes 107 and 108:** “What are the error codes 107 and 108? (For example: ERROR 108 received in myTTSCallBack.)”
- ❑ **Connecting to the server:** “When trying to run the Speechify “spfyDemo” sample application, you may get the following error message box: “ERROR: The ScanSoft TTS library could not connect to the server.””
- ❑ **Port types:** “What is the relationship between the SWIttsOpenPortEx() function and the port number used to configure a voice on the Speechify server?”
- ❑ **OpenPort performance:** “Is there a significant performance hit if SWIttsOpenPortEx() is called for every speak request?”

Question types

Why is the output the same, whether I end a sentence with a question mark or a period?

There are many different types of questions, including:

- ❑ Yes/No questions, e.g. Did you see the film yesterday?
- ❑ Tag questions, e.g. You saw the film yesterday, didn't you?
- ❑ Wh-questions, e.g. Where did you see the film?

Different question types have fundamentally different prosodic patterns. It is a common misconception that all a TTS engine needs to do is generate rising intonation at the end of every question. For certain question types, the intonation produced by Speechify (and many other TTS systems) might be inappropriate. In order to get the most natural speech possible out of Speechify, we have concentrated on generating optimal output for statements and wh-questions (who, how, what, when, where, why, which). If you design your user interactions carefully, it's possible to have your text consist entirely of statements and wh-questions. For example, "You can choose one of options a, b, or c. Which would you like?"

Changing rate or volume

Can the application change the speaking rate or volume of the audio output while Speechify is synthesizing?

Once it has asked Speechify to synthesize some text, the application cannot change the rate or volume of the output. However, the application can tell Speechify to change to different rates or volumes in the output by embedding rate and/or volume control tags in the input text. See "Volume control" on [page 81](#) and "Speaking rate control" on [page 82](#) for details.

Rewinding and fast-forwarding

Can the application move forward or backward in the audio output (e.g. go back 10 seconds)?

Speechify and Speechify Solo do not support this directly, but an application could buffer the audio output and provide that functionality.

Java interface

Does Speechify or Speechify Solo offer a Java interface?

Speechify and Speechify Solo currently do not offer a Java interface. A Java developer will need to write the JNI (Java Native Interface) wrapper to access the C API. If you would like ScanSoft to provide a Java interface, please contact your ScanSoft sales representative or technical support.

E-mail preprocessor and SAPI 5

Is it possible to use the e-mail preprocessor with SAPI 5?

It is possible to use the e-mail preprocessor and SAPI together. The application must process the e-mail via the e-mail preprocessor first, independently of SAPI, and then pass the resulting text into SAPI.

W3C SSML and SAPI 5

How can I use input text containing W3C SSML with SAPI 5?

To use W3C SSML input with the SAPI 5 interface: encode the SSML text in UTF-16, pass your SSML text to `ISpVoice::Speak`, and ensure that the `SPF_IS_NOT_XML` flag is set. Please note that this is non-standard behavior and may not be supported by other SAPI engine vendors.

Error codes 107 and 108

What are the error codes 107 and 108? (For example: ERROR 108 received in myTTSCallBack.)

Error codes 107 and 108 are both related to networking:

Error 107 indicates that the connection between the client and the server existed at one point but was disconnected unexpectedly. There are several likely causes:

- ❑ The Speechify server stopped, either by an operator shutting it down or by a crash.
- ❑ The machine that the server was running on was shut down.
- ❑ The network connection between the client and server was broken, either by unplugging a cable or shutting down a switch.

Error 108 indicates that some network operation between the client and server timed out. There are several common causes:

- ❑ The Speechify server stopped, either by an operator shutting it down or by a crash.
- ❑ The server machine is under heavy load and the Speechify server could not respond quickly enough to the client.
- ❑ The Speechify server is not started.

Connecting to the server

When trying to run the Speechify “spfyDemo” sample application, you may get the following error message box: “ERROR: The ScanSoft TTS library could not connect to the server.”

Common causes:

- ❑ The Speechify service has not yet been started. Start the Speechify service and try running the sample again.
- ❑ You have started the Speechify server but it has not completed initialization so is not ready to accept connections.
- ❑ You entered an incorrect host-name/port-number pair in the Server Address box. Restart the sample after checking that you have the correct address.

Port types

What is the relationship between the SWIttsOpenPortEx() function and the port number used to configure a voice on the Speechify server?

A common mistake is to think that each call that the application makes to SWIttsOpenPortEx() must specify a different port number for the server address. In fact, the TTS “port” handle returned from the SWIttsOpenPortEx() function has nothing to do with the “port” number passed to the function. SWIttsOpenPortEx() creates an instance of a Speechify voice and returns a handle to that voice. The port number passed to the function and used on the server to configure the voice refers to a different type of port: a networking/sockets port.

When you configure a voice on the server for a certain port number, that number is part of the network address on which the server listens for connections from a client application. That application creates an instance of a voice by passing the server's address, i.e., the host name and the port number, to SWIttsOpenPortEx(). To create multiple TTS ports for that voice, always pass the same host name/port number combination to the function. Even though a single Speechify server listens on only one network address, it can support multiple voice instances.

OpenPort performance

Is there a significant performance hit if `SWIttsOpenPortEx()` is called for every speak request?

There is a performance hit to establish and tear down a connection from the client to the server for each speak request. The price you pay depends a little on the operating system and a lot on the network configuration.

Network latency for the `SWIttsOpenPortEx()` request totally depends on your network setup. In our testing, it can take up to 150 ms to return from `SWIttsOpenPortEx()` or as little as 20 ms. Somewhere in the middle is typical, unless the client and server are running on the same box (usually 10–20 ms for round trips).

Index

Symbols

!SWI 191
\\! 72
\\![SPR] 75
\\!bm 79
\\!eos 73
\\!ny0 78
\\!ny1 78
\\!p 72
\\!rdN 82
\\!rdr 82
\\!rpN 82
\\!rpr 82
\\!ts0 76
\\!tsa 76
\\!tsc 72, 76
\\!tsr 76
\\!vdN 81
\\!vdr 81
\\!vpN 81
\\!vpr 81

A

abbreviation dictionary 111, 115
 uses for 120
abbreviations
 processing of annotations 74
ambiguous expressions 84
 homographs 126
annotations
 entering SPRs 74
 pause 72
 pronouncing numbers and years 77
 quick reference table 7
API function overview 133
API result codes 135
application outline 49
audio buffer underflow 42
audio SSML tag 100

B

bookmarks
 inserting 79
break SSML tag 100

C

callback function 50, 141
calling convention 134
cardinal numbers 84
character set
 for Japanese 162, 165
 ISO-8859-1 134, 162, 165

preferred 134
Shift-JIS 134
supported 162, 165
US-ASCII 162, 165
UTF-16 162, 165
UTF-8 162, 165
wchar_t 162, 165
client SDK on Windows
 hardware requirements 4
 software requirements 4
commas, role in phrase breaks 118
concatenative synthesis, definition of 129
configuration file
 layers 24, 25
 voice 25
configuring Speechify 3
copyrights for third party software xiv
CPU utilization 42
currencies 85

D

dates 85
<definition> dictionary element 67
desc SSML tag 100
diagnostic message syntax 31
dictionaries
 abbreviation 111
 main 111
 mainext 113
 processing of abbreviation annotations 74
 root 112
dictionary element
 <definition> 67
 <entry> 67
 <lexicon> 65
digits
 floating point 84

E

email preprocessor, use of 120
Embedded tags
 use of 121
end of sentence 73, 112
<entry> dictionary element 67

F

features 48
floating point digits 84
fractions 84

G

guidelines

implementation 58

H

homographs

overview of 126

I

implementation guidelines 58

International Radio Alphabet 76

J

Japanese, supported charsets 162, 165

K

key, dictionary 61, 110

L

latency 41

<lexicon> dictionary element 65

lexicon SSML tag 101

M

main dictionary 111, 115

uses for 120

mainext dictionary 113

mark SSML tag 101

metadata SSML tag 101

MIME types 149, 152, 165, 176, 179, 180, 181, 185,
190

monetary expressions 85

N

negative numbers 84

normalization of text

as part of text analysis 119

numbers

cardinal 84

negative 84

ordinal 84

numbers and years

annotations 77

O

order of API functions 49

ordinal numbers 84

P

paging file 4

paragraph SSML tag 101

pauses

inserting 72

performance 39

period, trailing abbreviation dictionary) 112

phone numbers 85

phoneme SSML tag 101

prosody SSML tag 101

R

result codes 135

root dictionary 112, 114

S

sample time line 52

SAPI 7, 189

say-as SSML tag 101

sentence SSML tag 101

server CPU utilization 42

sizing 39

social security numbers 85

speak SSML tag 102

Speechify

configuring 3

features 48

installing 3

SpeechifyErrors.en-US.xml 33

spellout modes 76

SPF_IS_NOT_XML 196

SPR tag 74

SPR tags

use of 74, 120

SSML tags

use of 121

structure

SWIttsAudioPacket 51, 139

SWIttsBookMark 140

SWIttsMessagePacket 139

SWIttsPhonemeMark 141

SWIttsWordMark 140

support services xiii

SWIrecResourceAllocate() 157

SWIrecResourceFree() 158

SWItts_ALREADY_EXECUTING_API 135

SWItts_ALREADY_INITIALIZED 135

SWItts_cbAudio 137

SWItts_cbBookmark 137

SWItts_cbDiagnostic 29, 137

SWItts_cbEnd 137

SWItts_cbError 138

SWItts_cbLogError 29, 138

SWItts_cbPhonememark 138

SWItts_cbPortClosed 138

SWItts_cbStart 138

SWItts_cbStatus 50, 137

SWItts_cbStopped 138

SWItts_cbWordmark 138

SWItts_CONNECT_ERROR 135

SWItts_DICTIONARY_ACTIVE 135

SWItts_DICTIONARY_LOADED 135

SWItts_DICTIONARY_NOT_LOADED 135

SWItts_DICTIONARY_PARSE_ERROR 135

- SWItts_DICTIONARY_PRIORITY_
 - ALREADY_EXISTS 135
 - SWItts_ERROR_PORT_ALREADY_STOPPING 135
 - SWItts_ERROR_STOP_NOT_SPEAKING 135
 - SWItts_FATAL_EXCEPTION 135
 - SWItts_HOST_NOT_FOUND 135
 - SWItts_INVALID_MEDIATYPE 135
 - SWItts_INVALID_PARAMETER 135
 - SWItts_INVALID_PORT 135
 - SWItts_INVALID_PRIORITY 135
 - SWItts_LICENSE_ALLOCATED 135
 - SWItts_LICENSE_FREED 135
 - SWItts_MUST_BE_IDLE 135
 - SWItts_NO_LICENSE 135
 - SWItts_NO_MEMORY 135
 - SWItts_NO_MUTEX 135
 - SWItts_NO_THREAD 135
 - SWItts_NOT_EXECUTING_API 135
 - SWItts_PORT_ALREADY_SHUT_DOWN 136
 - SWItts_PORT_ALREADY_SHUTTING_DOWN 136
 - SWItts_PORT_SHUTTING_DOWN 136
 - SWItts_PROTOCOL_ERROR 136
 - SWItts_READ_ONLY 136
 - SWItts_SERVER_ERROR 136
 - SWItts_SOCKET_ERROR 136
 - SWItts_SSML_PARSE_ERROR 29, 99, 136
 - SWItts_SUCCESS 136
 - SWItts_UNINITIALIZED 136
 - SWItts_UNKNOWN_CHARSET 136
 - SWItts_URI_FETCH_ERROR 136
 - SWItts_URI_NOT_FOUND 136
 - SWItts_URI_TIMEOUT 136
 - SWItts_WINSOCK_FAILED 136
 - SWIttsAudioPacket structure 51, 139
 - SWIttsBookMark structure 140
 - SWIttsCallback() 137
 - SWIttsClosePort() 143, 144, 146, 147, 148
 - SWIttsDictionariesDeactivate() 146
 - SWIttsDictionaryActivate() 144
 - SWIttsDictionaryData 148
 - SWIttsDictionaryFree() 147
 - SWIttsDictionaryLoad() 57, 148
 - SWIttsGetParameter() 56, 151
 - SWIttsInit() 151, 154
 - SWITTSLOGDIAG 30
 - SWITTSLOGTOFILE 31
 - SWITTSLOGTOSTD 30
 - SWITTSMAXLOGSIZE 31
 - SWIttsMessagePacket structure 139
 - SWIttsOpenPort() 155
 - SWIttsOpenPortEx()
 - stack size 58
 - SWIttsPhonemeMark structure 141
 - SWIttsResult 135
 - SWITTSSDK 7, 174, 186
 - SWIttsSetParameter() 56, 159
 - SWIttsSpeak() 57, 161, 164
 - SWIttsSpeakData 164
 - SWIttsSpeakEx() 57
 - SWIttsStop() 167
 - SWIttsTerm() 168
 - SWIttsWordMark structure 140
 - syllable boundaries 97
 - syllable stress 97
 - Symbolic Phonetic Representation 74
- T**
- third-party software xiv
 - time line
 - sample 52
 - times 85
 - translation value, dictionary 61, 110
 - tts 181
 - tts.audio.packetsize 151, 159, 175
 - tts.audio.rate 151, 159, 175
 - tts.audio.volume 151, 159, 176
 - tts.audioformat.encoding 151
 - tts.audioformat.mimetype 152, 160
 - tts.audioformat.mimetype.16kHz 176
 - tts.audioformat.mimetype.8kHz 176
 - tts.audioformat.samplerate 152
 - tts.audioformat.width 152
 - tts.cache.dir 22, 176
 - tts.cache.entryExpTimeSec 176
 - tts.cache.entryMaxSizeMB 177
 - tts.cache.totalSizeMB 177
 - tts.client.version 152
 - tts.engine.dictionaries 62, 171, 174, 177
 - tts.engine.dictionaryDefaultPriorityBase 101, 177
 - tts.engine.id 152
 - tts.engine.version 152
 - tts.inet.acceptCookies 179
 - tts.inet.extensionRules 23, 179
 - tts.inet.proxyPort 179
 - tts.inet.proxyServer 179
 - tts.inet.userAgent 179
 - tts.log.contentDir 180
 - tts.log.diagnostic.errorMapFile 33, 180
 - tts.log.diagnostic.file 22, 32, 180
 - tts.log.diagnostic.fileMaxSizeMB 180
 - tts.log.diagnostic.fileMimeType 180
 - tts.log.diagnostic.reportErrorText 180
 - tts.log.diagnostic.toStdout 32, 180

tts.log.diagnostic.toSystemLog [32](#), [181](#)
tts.log.event.file [22](#), [28](#), [181](#)
tts.log.event.fileMaxSizeMB [181](#)
tts.log.event.fileMimeType [181](#)
tts.log.event.generatedPronunciations [130](#), [181](#)
tts.marks.phoneme [152](#), [160](#), [182](#)
tts.marks.word [152](#), [160](#), [182](#)
tts.network.timeout [152](#), [160](#), [174](#), [182](#)
tts.preprocessor.map128ToEuroInISO8859-1 [183](#)
tts.product.name [153](#)
tts.reset [160](#)
tts.server.licenseServerList [183](#)
tts.server.licensingMode [153](#), [157](#), [158](#), [183](#)
tts.server.networkInterface [184](#)
tts.server.numPorts [22](#), [184](#)
tts.server.port [22](#), [184](#)
tts.ssml.doStrictValidation [99](#), [185](#)
tts.ssml.failOnAudioFetchError [185](#)
tts.textformat.mimetype [185](#)
tts.voice.dir [22](#), [186](#)
tts.voice.format [22](#), [176](#), [186](#)
tts.voice.gender [153](#)
tts.voice.language [22](#), [153](#), [171](#), [186](#)
tts.voice.name [22](#), [153](#), [186](#)
tts.voice.version [153](#)

U

underflow rate [42](#)

V

voice configuration file [25](#)

voice SSML tag [102](#)

W

white space, in Japanese text [162](#), [166](#)