AT&T Natural Voices™ Text-To-Speech Engines Version 5.1.1 for Linux

Developer's Guide

Table of Contents

1. Introduction

About the TTS Engine

AT&T Natural Voices Text-to-Speech (TTS) Engines provide synthesis services in multiple languages for application builders. The TTS engine runs on both single and multi-processor computers.

The AT&T Natural Voices TTS product is currently available in two configurations: the Server Edition and the Desktop Edition. The Server Edition provides a scalable, client/server architecture that's ideal for supporting many simultaneous client connections to a single server or a large server farm of TTS servers. The Desktop Edition provides the same AT&T Natural Voices TTS technology on a desktop computer and limits the TTS engine to a single channel.

The AT&T Natural Voices supports US English and Spanish. You can mix languages in a single input file by specifying the voice to use to speak the text. Note that any voice will attempt to speak whatever text is presented, i.e. Rosa (Spanish) will attempt to synthesize English text as Spanish words.

AT&T Natural Voices TTS engines allow you to mix languages on a single server simply by specifying a different voice. Users can specify a voice and language in an SSML control tag that is included in the input text or make C++ API calls to switch between voices and languages seamlessly. This architecture also allows you to deliver new voices in a variety of languages with a single TTS engine executable.

8 KHz and 16 KHz versions of all voices are available separately. The 8 KHz voices are ideally suited for telephony applications. The 16 KHz voices offer higher quality output for desktop applications including web site applications and wav files. You can install any combination of 8 KHz and 16 KHz voices on a single server.

For Developers

Both editions include a C++ language SDK that allows application developers to control the TTS engine. The Client SDK provided with the product includes source code for sample applications that demonstrates how applications interact with the TTS engine.

The TTS engines support the Speech Synthesis Markup Language (SSML) component of the Voice XML standard. This markup language allows client applications to include special instructions within the input text that may change the default behavior of the text synthesizer (not all features are supported in all languages).

2. Minimum Requirements

Hard Drive Space

All editions of the AT&T Natural Voices require less than 50 MB of disk space (without voice fonts).

Disk requirements of voice fonts vary. Generally, 8 KHz voice fonts require between 150 MB and 200 MB of hard drive space per font, while 16 KHz voice fonts require between 500 MB and 900 MB per font.

You will require some additional space to store server logs if running the Server Edition.

Memory

All editions require 256 MB RAM minimum but 512MB RAM is recommended. Extra memory will provide better performance for all editions if you use the TTS engine while other applications are running.

Also note that each voice requires about 75 MB of data to be accessed by the server so we recommend that you have at least 1 GB of memory per server CPU if you start more than one voice.

The Server Edition memory requirements grow depending on a number of channels – each additional channel requires an additional 10 MB RAM.

Test Results

Our test results show that a Server Edition running on a 500 MHz Intel Pentium-III processor with 512 MB of memory can support around 32 simultaneous channels of TTS with reasonable performance.

The Server Edition running on a 1 GHz Intel Pentium-4 with 1 GB of memory supports about 48 simultaneous channels of TTS.

In both cases, the TTS server computer is not running any other applications.

3. Installation

You may keep either edition of AT&T Natural Voices in any directory. However, samples in this manual assume that AT&T Natural Voices reside under `/usr/local`.

Desktop Edition

The tar file will unpack into a single directory called `att_naturalvoices_v5.1_desktop`. You may safely unpack the tar file in any directory. To extract the files from the tar file, use the command:

```
tar -xvf att_naturalvoices_v5.1_desktop_live.tar , or
tar -xvf att_naturalvoices_v5.1_desktop_trial.tar
```

Server Edition

The tar file will unpack into a single directory called `att_naturalvoices_v5.1_server`. You may safely unpack the tar file in any directory. To extract the files from the tar file, use the command:

```
tar -xvf att_naturalvoices_v5.1_server_live.tar , or
tar -xvf att_naturalvoices_v5.1_server_trial.tar
```

The server edition has both server and client components within. If you need to access the engine from another computer, copy the following client components onto that computer:

1. File: `<INSTALL_DIR>/bin/TTSClientPlayer`
2. File: `<INSTALL_DIR>/bin/TTSClientFile`
3. Directory: `<INSTALL_DIR>/sdk` (if you build C applications)

Evaluation Version

The evaluation version of AT&T Natural Voice TTS Engines limits usage to a certain number of days. Every time when it is used, the TTS engine will output a message telling you how many trial days are left. An Internet connection is needed for the evaluation version to work.

If you have the evaluation version of either the desktop or server edition, you have to do the following before you attempt to use AT&T Natural Voices for a first time:

1. Locate the order number, which can be found on the invoice sent by Wizzard Software Corp.
2. Create a file named `order.txt` and type the order number (10 digits) into that file.
3. Save `order.txt` into the directory `<INSTALL_DIR>/bin.`

Please note that if you will be executing AT&T Natural Voices binaries from directories other than `<INSTALL_DIR>/bin`, then you will need to copy `order.txt` into CWD as well.

Install Voice Font(s)

Your desktop and/or server edition should have one or more voice fonts installed before you attempt to use it for a first time. Each voice font is packaged separately. The desktop and server editions use the same fonts.

The voice font tar file will unpack files into a single directory. To install a voice font, simply copy all the extracted files into the appropriate directory (see Table of Fonts, below).

Here is an installation example for the voice font Mike 8 KHz (US English) and Rosa 16 KHz (Spanish):

```
tar -xvf att_naturalvoices_v5.1_mike8.tar
tar -xvf att_naturalvoices_v5.1_rosa16.tar
cp mike8/* /usr/local/att_naturalvoices_v5.1_server/data/en_us/mike8
cp rosa16/* /usr/local/att_naturalvoices_v5.1_server/data/es_us/rosa16
```

To complete a voice font installation, you will need to adjust the configuration file named `tts.cfg` and located in the directory `<INSTALL_DIR>/data`. Uncomment the font that you have installed. Commented lines have double pound (`##)` in the beginning of the line, while uncommented lines have a single pound (`#)` in the beginning of the line.

Continuing the above example, you would need to replace following lines:

```
      ##include "en_us/mike8/mike8.cfg"
      ##include "es_us/rosa16/rosa16.cfg"
```

With this:

```
      #include "en_us/mike8/mike8.cfg"
      #include "us_us/rosa16/rosa16.cfg"
```

Table of Fonts

Language	**Name**		**Location**
US English	Mike	M	`<INSTALL_DIR>/data/en_us/mike*`
	Crystal	F	`<INSTALL_DIR>/data/en_us/crystal*`
	Rich	M	`<INSTALL_DIR>/data/en_us/rich*`
	Ray	M	`<INSTALL_DIR>/data/en_us/ray*`
	Claire	F	`<INSTALL_DIR>/data/en_us/claire*`
	Julia	F	`<INSTALL_DIR>/data/en_us/julia*`
	Lauren	F	`<INSTALL_DIR>/data/en_us/lauren*`
	Mel	M	`<INSTALL_DIR>/data/en_us/mel*`
Spanish	Alberto	M	`<INSTALL_DIR>/data/es_us/alberto*`
	Rosa	F	`<INSTALL_DIR>/data/es_us/rosa*`

Replace * with either 8 or 16, depending on voice font type.

Distribution Components

Directory	Distribution
`<INSTALL_DIR>/bin`	Distribute only files which you use within your application. Always use files from the live version. Always distribute `TTSServer` (applies to Server Edition only).
`<INSTALL_DIR>/lib`	Always distribute files from this directory. These files are the same for evaluation and live versions.
`<INSTALL_DIR>/data`	Always distribute this directory and all files and sub-directories within. Make sure you install purchased voice fonts before distribution.
`<INSTALL_DIR>/docs`	Do not distribute this directory.
`<INSTALL_DIR>/sdk`	Do not distribute this directory.

Please note that if you have created an application using our client C++ SDK then you will need to recompile your executables with the libraries from the live version. Those libaries could be found in:

```
<INSTALL_DIR>/sdk/lib
```

64-bit Components

Your desktop and/or server edition includes 64-bit components. Following list represents the location of such files. You can replace 32-bit components with their 64-bit versions if you prefer to use and/or redistribute your program on a 64-bit machine.

32-bit Components	64-bit Components
`<INSTALL_DIR>/bin`	`<INSTALL_DIR>/bin64`
`<INSTALL_DIR>/lib`	`<INSTALL_DIR>/lib64`
`<INSTALL_DIR>/sdk/lib`	`<INSTALL_DIR>/sdk/lib64`

4. Using the Server Edition

The AT&T Natural Voices Server Edition supports a client/server architecture where the client and server executables run on the same or different computers.

Running the Server

```
TTSServer -c connectionPort [-r path] [-x voice] [-v[0|1|2|3]] [-config file]
[-m max_clients] [-y port_no] [-temp path] [-if interface] [-wd #] [-l file]
[-rollover N [unit]]
```

Option	Argument	Action
-c	Connection port	Required field. Identifies the port on which TTSServer will be listening for client connections. Note that the client application must be configured to use this same port.
-r	Path name	The directory path of the `data` subdirectory. Default path is `$NaturalVoicesPath`
-x	Voice font	Specify the voice: mike8, crystal16, etc.
-v0	Trace level 0	Prints only error messages.
-v1	Trace level 1	Prints error messages, voice information, and process start/stop information. This is a default.
-v2	Trace level 3	Prints error messages, voice information, process start/stop information, and request and reply packets. Beware that v2 mode generates extensive information and should be used sparingly.
-v3	Trace level 3	Prints error messages, voice information, process start/stop information, request and reply packets, and all notifications. Beware that v3 mode generates extensive information and should generally not be used.
-config	Configuration file	Default is `<INSTALL_DIR>/data/tts.cfg`
-m	Number of clients	Maximum number of simultaneous clients. Attempts to allocate additional channels will fail. Default is 32.
-y	SNMP Port	Access this port for SNMP.
-temp	Path name	Use the given path for temporary files (default is `/tmp`)
-if	Interface	Bind to specified network interface
-wd	#	Allow # timeout intervals before disconnecting client
-l	Log file path	File where messages are logged. Default is stdout.
-rollover	N [unit]	Rollover log file after N minutes (or specify unit of time (hours, minutes, days, weeks))

Examples:

```
/usr/local/att_naturalvoices_v5.1_server/bin/TTSServer -c 7000 -l tts.log -x
mike8 -r /usr/local/att_naturalvoices_v5.1_server/data

NaturalVoicesPath=/usr/local/att_naturalvoices_v5.1_server/data
export NaturalVoicesPath
/usr/local/att_naturalvoices_v5.1_server/bin/TTSServer -c 7000
```

Running the Client

Once the server is up and running, you can send text to the TTS server to try it out. You can write your own application in C++ or you can use the sample applications TTSClientPlayer and TTSClientFile, which can be found in the <INSTALL_DIR>/bin directory. TTSClientPlayer allows you to synthesize text files and play them through a sound card. TTSClientFile synthesizes text files and writes the linear 16 bit PCM output as a wav file.

```
TTSClientPlayer -p connectionPort -s TTSServer [-v[0|1|2|3]] [-transcribe]
[-a] [-ssml] [-latin1] [-f inputFile | -l fileList] [-du userDictionary] [-da
applicationDictionary] [-mulaw] [-normalize] [-queue]
```

```
TTSClientFile -p connectionPort -s TTSServer [-v[0|1|2|3]] [-transcribe] [-a]
[-ssml] [-latin1] [-f inputFile | -l fileList] [-du userDictionary] [-da
applicationDictionary] [-mulaw] [-normalize] [-queue] -o outputFile
```

Option	**Argument**	**Action**
-p	Connection port	Required field. Identifies the port number on which `TTSServer` will be listening for client connections.
-s	Server name	Required field. Specify the name or IP address of the TTS server. Use `localhost` if server is on the same PC.
-v	Verbose output (0-3)	3 is default. Use -v0 for no output; -v3 for max output.
-transcribe	None	Perform a transcription instead of speaking.
-normalize	None	Same as above, but with fully normalized text
-a	None	Use asynchronous engine behavior.
-x	Default voice font	Specify the voice: mike8, crystal16, etc.
-ssml	None	Parse the input files as SSML.
-latin1	None	Assume Latin-1 input (default encoding is UTF-8).
-f	Input file name	Path of the text file to synthesize.
-l	File with a list of files	Specify a file with a list of text files to synthesize.
-du	User dictionary file	Specify a user dictionary file.
-da	Application dictionary file	Specify an application dictionary file.
-o	Audio output file	The output audio file (TTSClientFile only).
-mulaw	None	Generate mulaw audio output
-queue	None	Queue all speak requests at once.

Following sample synthesizes sample.txt using the TTS server running on the local machine and listening on port 7000 and writes the audio to a sound card:

```
/usr/local/att_naturalvoices_v5.1_server/bin/TTSClientPlayer -p 7000 -s
localhost -f sample.txt
```

Following sample synthesizes the text in sample.txt, capturing the audio output to sample.wav:

```
/usr/local/att_naturalvoices_v5.1_server/bin/TTSClientFile -p 7000 -s
localhost -f sample.txt -r 22050 -o sample.wav
```

Generated Output Messages

Server Output and Error Messages

The TTS Server writes all output and error messages to the log file specified with the –l option.
All messages have the format

Wire#*child time message*

e.g. Wire#1 171.90 : wireline started
where *child#* is a sequential number of speak requests, where each speak request creates a new child process and *time* is the number of seconds since the server started.

Initialization Messages

The following messages are generated by the TTS Server during initialization. These set of messages indicate that a wireline server started up. Wireline is the application itself, server engine is the thread that receives wireline requests and sends wireline replies, the asynch notify server is the thread that sends wireline notifications, and the asynch standalone is the thread that processes speak requests.

```
Wire#1 171.90 : wireline started
Wire#1 171.91 : server engine started
Wire#1 172.81 : asynch notify server started
Wire#1 172.85 : asynch standalone started
```

Shutdown Messages

The following messages are generated by the TTS Server during shutdown and map directly to the corresponding Server Initialization Messages.

```
Wire#1 177.56 : asynch standalone finished
Wire#1 177.56 : asynch notify server finished
Wire#1 177.56 : server engine finished
Wire#1 177.60 : wireline finished
```

Voice Inventory Messages

The following messages enumerate the voices that are available on the Server, the default voice and the current voice:

```
Wire#1 172.93 : Installed voices
mike8;en_us;male;adult;8000;16;pcm;
klara16;de_de;female;adult;16000;16;pcm;
Wire#1 172.94 : Default voice: mike8;en_us;male;adult;8000;16;pcm;
Wire#1 172.94 : Current voice: klara16;de_de;female;adult;16000;16;pcm;
```

Client Messages

These messages may be displayed when a client disconnects from the server unexpectedly.

```
Wire#1 368.83 : receive error 10054 ::Connection reset by peer
Wire#1 368.83 : sending notify packet::Connection reset by peer
Wire#1 368.90 : wireline finished::Connection reset by peer
```

Periodically, the server will clean up resources for child processes that have exited.

```
TTSWireServer 395.02 : reaped child 1
```

These messages are simply housekeeping messages and can be ignored.

5. Using the Desktop Edition

TTSStandalonePlayer and TTSStandaloneFile are included in the `<INSTALL_DIR>/bin` directory. TTSStandalonePlayer allows you to synthesize text files and play them through a sound card. TTSStandaloneFile synthesizes text files and writes output to a file.

```
TTSStandalonePlayer [-v[0|1|2|3]] [-transcribe] [-a] [-ssml] [-latin1] [-f
file | -l filelist] [-du userdictionary] [-da appdictionary] [-x voice_name]
[-normalize] [-mulaw] [-data path] [-root path] [-config path] [-queue]
```

```
TTSStandaloneFile [-v[0|1|2|3]] [-transcribe] [-ssml] [-latin1] [-f file | -l
filelist] [-du userdictionary] [-da appdictionary] [-x voice_name] [-mulaw] [-
a] [-normalize] [-data path] [-root path] [-config path] [-queue] -o file
```

Option	Argument	Action
-v	Verbose output (0-3)	3 is default. Use -v0 for no output; -v3 for max output.
-transcribe	None	Perform a transcription instead of speaking.
-normalize	None	Same as above, but with fully normalized text
-a	None	Use asynchronous engine behavior.
-ssml	None	Parse the input files as SSML.
-latin1	None	Assume Latin-1 input (default encoding is UTF-8).
-f	Input file name	Path of the text file to synthesize.
-l	File with a list of files	Specify a file with a list of text files to synthesize.
-du	User dictionary file	Specify a user dictionary file.
-da	Application dictionary file	Specify an application dictionary file.
-x	Default voice font	mike8, crystal16, etc.
-o	Audio output file	The output audio file (TTSStandaloneFile only).
-mulaw	None	Generate mulaw audio output
-data	Path name	The directory path of the data subdirectory
-root	Path name	The directory path of the bin subdirectory
-config	Configuration file	Default is `<INSTALL_DIR>/data/tts.cfg`
-queue	None	Queue all speak requests at once.

Following sample synthesizes the text in sample.txt, capturing the audio output to sample.wav.

```
/usr/local/att_naturalvoices_v5.1_desktop/bin/TTSStandaloneFile -x rosa8 -f
sample.txt -data /usr/local/att_naturalvoices_v5.1_desktop/data -o sample.wav
-root /usr/local/att_naturalvoices_v5.1_desktop/bin
```

Following sample synthesizes the text in sample.txt and writes the audio to a sound card:

```
/usr/local/att_naturalvoices_v5.1_desktop/bin/TTSStandalonePlayer -x mike8 -f
sample.txt -data /usr/local/att_naturalvoices_v5.1_desktop/data -root
/usr/local/att_naturalvoices_v5.1_desktop/bin
```

6. Client C++ SDK

The AT&T Natural Voices TTS package includes a Software Developers Kit (SDK) for building client applications that work with the TTS server. The SDK contains C++ class definitions, libraries and sample applications that demonstrate the use of the SDK. The Server and Desktop Editions of the AT&T Natural Voices use the same SDK. The only difference between using the SDK with the two Editions is in selecting the engine model when creating the engine.

6.1. Getting Started

SDK Headers

There is only one header required for inclusion in a speech synthesis application. The header file TTSApi.h contains all the necessary C++ classes. Alternatively, an application can include an operating system-specific header, TTSWin32API.h for Windows or TTSUnixAPI.h for UNIX, which contains some additional classes that may be useful for building applications. TTSWin32API.h and TTSUnixAPI include the header file TTSApi.h.

Header	Description
TTSApi.h	Contains all the necessary C++ classes, definitions and functions. Also includes the headers TTSResult.h and TTSOSMacros.h
TTSWin32API.h	Contains some optional Windows-specific C++ classes, headers and definitions. Includes TTSApi.h
TTSUnixAPI.h	Contains some optional Unix-specific C++ classes, headers and definitions. Includes TTSApi.h
TTSResult.h	This file contains the result codes that are returned from the SDK C++ classes and functions
TTSOSMacros.h	This file contains operating system-specific macros necessary to provide thread-safe access to the SDK C++ classes.
TTSUTF8.h	This file contains string functions that handle UTF-8 strings.

Compiler Definitions

The following preprocessor definitions should be used when integrating the SDK with an application.

Operating System	Definitions
Windows	WIN32
Linux	UNIX

SDK Libraries

Those libraries are built with the GNU g++ compiler and use pthread functionality. An application must also link with libpthread.a.

Library	AT&T Natural Voices
libTTSAPICS.a	Server Edition
libTTSAPISA.a	Desktop Edition

6.2. Programming Conventions

Return Codes

The header TTSResult.h defines a result code type called TTS_RESULT and all the possible values it may contain. Most of the C++ global and class member functions return a TTS_RESULT code. An application may use the macros FAILED() and SUCCEEDED() to determine if a function failed or succeeded. There is also a static class CTTSResult that will return a descriptive English-language string for the result code. For example:

```
TTS_RESULT result = ttsCreateEngine(...);
if (FAILED(result)) {
      cout << CTTSResult::GetErrorString(result);
}
```

CTTSRefCntObject

All of the SDK C++ classes derive from the CTTSRefCntObject class. This class is used to provide reference counting for classes and manage pointers to the classes. When a class object is created, its reference count is zero. An application should call CTTSRefCntObject::AddRef for every pointer to a class object it needs to obtain. CTTSRefCntObject::AddRef increases the reference count of the class by one.

An application should call CTTSRefCntObject::Release when it is finished with a pointer to a class object. CTTSRefCntObject::Release decreases the reference count of the object by one. When the reference count is zero, the object will free itself from memory. These calls are symmetrical, for every CTTSRefCntObject::AddRef there should be a CTTSRefCntObject::Release.

UTF-8 Strings

The SDK supports only UTF-8 strings as both input and output. UTF-8 is a multibyte character encoding. Its major advantage is that it allows encoding of 7-bit US-ASCII characters in a single byte, thus allowing existing US-ASCII text data to be read as UTF-8 without modification. The application should be conscious of functions within the SDK that receive or return text in UTF-8 format. The header TTSApi.h defines some type definitions for UTF-8 strings.

```
typedef       unsigned char   UTF8String;
typedef       unsigned char * PUTF8String;
typedef const unsigned char * PCUTF8String;
```

C++ Standard Template Library

Some of the TTS SDK class functions use C++ Standard Template Library (STL) list, vector, and string container classes. Developers should be familiar with using STL classes.

6.3. Creating an Engine

The SDK is the same for the Server and Desktop Editions of the AT&T Natural Voices TTS engines. The only difference in the SDK between Editions is the engine model that is specified during engine configuration. Creating an engine involves building a TTSConfig structure and calling the library function ttsCreateEngine().

Build a Configuration Structure

The TTSConfig structure contains the information necessary for the SDK library to create a CTTSEngine object.

TTSConfig	Description
M_eEngineModel	Indicates which engine model to create: TTSENGINEMODEL_CLIENTSERVER – a client/server engine model. The SDK will determine what protocol the server is running. Server Edition only. TTSENGINEMODEL_STANDALONE – a standalone engine model. This supports only one instance of the Natural Voices engine per process. Desktop Edition only.
M_eEngineBehavior	Determines the behavior of the engine: TTSENGINE_SYNCHRONOUS – the behavior is the same for all function calls as in the asynchronous mode except the CTTSEngine::Speak()call which will not return until all the audio and other notifications have been sent to the application's CTTSSink class. TTSENGINE_ASYNCHRONOUS – the engine will return after fully completing a function call. Most affected is the CTTSEngine::Speak() call which will return before all audio and other notifications have been sent to the applications CTTSSink class. TTSENGINE_MESSAGEBASED – the engine will return immediately from all function calls. The application should rely on the CTTSSink notification to determine if the call was successful or not.
M_strDataDirectory	This field is only applicable to the TTSENGINEMODEL_STANDALONE engine models. It is an override to where the voice data is located.
M_strTempDirectory	This is an optional field that tells the SDK where to store temporary files.

The TTSServerConfig structure is a description of a server, port and timeout that the client/server engine will use to connect to a TTS server. This structure is appropriate only for the TTSENGINEMODEL_CLIENTSERVER engine models.

TTSServerConfig	Description
M_nServerPort	The TTS server engine port to which the SDK should connect.
m_tvTimeout	The timeout value the SDK should use while communicating with a TTS server. If the application sets the timeval to {0, 0} the SDK will use an appropriate default.
m_szServer	The TTS server to which the SDK should connect. It can either be a DNS name or an explicit IP address.

All the engine modes and behaviors are multi-thread safe. The TTSENGINE_ASYNCHRONOUS engine is for applications that need to ensure that function calls return the result from the TTS server before returning control to the application. In addition, they rely on separate threads in the SDK to process notification and audio data from the TTS server. The TTSENGINE_SYNCHRONOUS engine is for applications that do not want extra threads running in the system and need CTTSEngine::Speak() calls to run to completion before returning. The TTSENGINE_MESSAGEBASED engine is for applications that run in a state machine and rely on the asynchronous notifications in the CTTSSink to drive the application.

Creating an Engine

The process for creating an engine is slightly different for the AT&T Natural Voices TTS Server Edition than for the Desktop Editions.

Creating an Engine for the Server Edition

Invoking the ttsCreateEngine() function will result in a CTTSEngine pointer being returned to the application. If the call is successful, the application should AddRef()the returned CTTSEngine pointer. When the application no longer needs the CTTSEngine pointer, it should call its Release() member function. The format of the ttsCreateEngine() function is:

```
TTS_RESULT ttsCreateEngine(CTTSEngine **ppEngine, const TTSConfig &ttsConfig);
CTTSEngine *pEngine = 0;
TTSConfig ttsConfig;
TTSServerConfig ttsServerConfig;
// Setup our server configuration
ttsServerConfig.m_nServerPort = nPort;
ttsServerConfig.m_szServer = pszServer;
ttsServerConfig.m_tvTimeout.tv_sec = 15;
ttsServerConfig.m_tvTimeout.tv_usec = 0;
// Setup our configuration
ttsConfig.m_eEngineModel = TTSENGINEMODEL_CLIENTSERVER;
ttsConfig.m_eEngineBehavior = TTSENGINE_SYNCHRONOUS;
ttsConfig.push_back(ttsServerConfig);
// Create the engine
result = ttsCreateEngine(&pEngine, ttsConfig);
// Success?
if (result == TTS_OK && this->m_pEngine) {
	// AddRef() the engine
	pEngine->AddRef();
	// application continues
	. . . .
	// Release the engine
	pEngine->Release();
}
```

Creating an Engine for the Desktop Edition

Invoking the ttsCreateEngine() function will result in a CTTSEngine pointer being returned to the application. If the call is successful, the application should AddRef()the returned CTTSEngine pointer. When the application no longer needs the CTTSEngine pointer, it should call its Release() member function. The format of the ttsCreateEngine() function is:

```
TTS_RESULT ttsCreateEngine(CTTSEngine **ppEngine, const TTSConfig &ttsConfig);
```

Standalone engine runs within the process space of the application that called ttsCreateEngine(). It is limited to a single engine instance per process. An attempt to call ttsCreateEngine() for a second standalone engine will fail, unless the first engine has been released.

```
CTTSEngine *pEngine = 0;
TTSConfig ttsConfig;
// Setup our configuration
ttsConfig.m_eEngineModel = TTSENGINEMODEL_STANDALONE;
ttsConfig.m_eEngineBehavior = TTSENGINE_SYNCHRONOUS;
// Create the engine
result = ttsCreateEngine(&pEngine, ttsConfig);
// Success?
if (result == TTS_OK && this->m_pEngine) {
	// AddRef() the engine
	pEngine->AddRef();
	// application continues
	. . . .
	// Release the engine
	pEngine->Release();
}
```

6.4. Receiving Engine Notifications and Messages

CTTSSink Notifications

An application should implement a CTTSSink-derived class and send a pointer to it to the CTTSEngine::SetSink() function. A CTTSSink object receives TTS notifications such as audio, bookmarks, word information, and phoneme information. When a CTTSEngine::Speak() function is called, the CTTSEngine object will send CTTSNotification objects to the application through its CTTSSink pointer.

```
class CSimpleEngine : public CTTSSink
{
public:
      CSimpleEngine(void)
      :
      CTTSSink()
{
}
virtual TTS_RESULT onNotification(CTTSNotification *pNotification)
{
            switch (pNotification->Notification()) {
                  ………
      }
      return TTS_OK;
}
protected:
      virtual ~CSimpleEngine(void)
      {
      }
};
```

An application tells the SDK which notifications it is interested in receiving by using the CTTSEngine::SetNotifications() function.

TTSNOTIFY_* definition	**Description**
TTSNOTIFY_STARTED	A Speak has started
TTSNOTIFY_NOTIFYCHANGED	A notification value has changed
TTSNOTIFY_VOICECHANGED	The voice has changed
TTSNOTIFY_AUDIOFORMATCHANGED	The audio format has changed
TTSNOTIFY_FINISHED	A Speak has finished
TTSNOTIFY_AUDIO	Audio data notification
TTSNOTIFY_WORD	Word notification
TTSNOTIFY_PHONEME	Phoneme notification
TTSNOTIFY_VISEME	Viseme notification
TTSNOTIFY_BOOKMARK	Bookmark notification
TTSNOTIFY_VOLUME	Volume value change
TTSNOTIFY_RATE	Speaking rate change
TTSNOTIFY_PITCH	Pitch change

When performing a CTTSEngine::SetNotifications() function call, the application specifies which notifications it is interested in modifying and whether the notification should be turned on or off. For example, the following code tells the engine that the application wishes to turn on audio and end-of-speech notifications, and to turn off phoneme notifications:

```
//Turn audio and finished notifications on and turns phoneme notifications off
TTS_RESULT result = pEngine->SetNotifications(
TTSNOTIFY_AUDIO | TTSNOTIFY_FINISHED | TTSNOTIFY_PHONEME,
TTSNOTIFY_AUDIO | TTSNOTIFY_FINISHED);
```

Turning off phoneme and viseme notifications can make a dramatic performance improvement because the events are not generated by the server and then processed by the client. If your application does not require phonemes or visemes, you can disable them with the following code:

```
// Turn audio notifications on and turns phoneme and viseme notifications off
TTS_RESULT result = pEngine->SetNotifications(
TTSNOTIFY_AUDIO | TTSNOTIFY_PHONEME | TTSNOTIFY_VISEME,
TTSNOTIFY_AUDIO);
```

CTTSNotification Object

The CTTSNotification object contains all the information for any type of notification. To determine the type of the notification, the application should call the CTTSNotification::Notification() function which returns a TTSNOTIFY_* code.

```
TTS_RESULT
CmySink::onNotification(CTTSNotification *pNotification)
{
   switch (pNotification->Notification()) {
   case TTSNOTIFY_AUDIO: {
      long lLength;
      if (pNotification->AudioData(&pAudioData, &lLength) == TTS_OK) {
         . . play the audio . .
         }
         break;
      }
   case TTSNOTIFY_FINISHED:
      . . signal that it is finished with a speak call. .
      break;
   case TTSNOTIFY_WORD: {
      TTSWordNotification wordNotification;
      if (pNotification->Word(&wordNotification) == TTS_OK) {
         cout << "WORD: ";
         cout << wordNotification.m_lCharacterOffset;
         }
      break;
      }
   case TTSNOTIFY_BOOKMARK: {
      PCUTF8String szBookmark = 0;
      if (pNotification->Bookmark(&szBookmark) == TTS_OK && szBookmark) {
         cout << "BOOKMARK: " << szBookmark << "\n";
         }
      break;
      }
   case TTSNOTIFY_PHONEME: {
      TTSPhonemeNotification phoneme;
      if (pNotification->Phoneme(&phoneme) == TTS_OK) {
         vector<PCUTF8String>::const_iterator iPhonemes =
                                          phoneme.m_vPhonemeIDs.begin();
         while (iPhonemes != phoneme.m_vPhonemeIDs.end()) {
            cout << *iPhonemes++;
            }
         cout << " " << phoneme.m_lDuration;
         cout << " " << phoneme.m_lStressLevel;
         }
      break;
      }
   return TTS_OK;
```

```
    }
```

TTSWordNotification structure

The TTSWordNotification structure contains all the information for a word notification.

TTSWordNotification members	Description
m_lCharacterOffset	Offset in bytes of the word from the beginning of text input.
m_lWordLength	The length of the word in bytes.
m_lSentenceLength	If this is the first word of a sentence, this is the length of the sentence in bytes.
m_lWordFlags	A bit-field of flags indicating what the word is: TTSWORDFLAG_SENTENCE TTSWORDFLAG_PARAGRAPH TTSWORDFLAG_NOUNPHRASE
m_ucPartOfSpeech	The part of speech of the word

TTSPhonemeNotification structure

The TTSPhonemeNotification structure contains all the information for a phoneme notification.

TTSPhonemeNotification members	Description
m_vPhonemeIDs	This is a vector containing one or more phoneme strings.
m_lDuration	The duration of the phoneme string in milliseconds.
m_lStressLevel	The stress level of the phoneme string (value between 1 and 3).

CTTSErrorInfoObject

An application may *optionally* implement a CTTSErrorInfoObject-derived class and send a pointer to it to the CTTSEngine::SetErrorInfoObject() function. A CTTSErrorInfoObject receives error, information, and trace messages from the SDK classes and can be useful for debugging and logging information.

```
class CSimpleEngine : public CTTSErrorInfoObject
{
public:
      CSimpleEngine(void)
      :
      CTTSErrorInfoObject()
      {
}
      virtual TTS_RESULT      onErrorMessage(TTS_RESULT error,
const char *pszErrorMessage)
      {
            try {
                  cout << pszErrorMessage << "\n";
                  cout << CTTSResult::GetErrorString(error) << "\n";
            }
            catch(...) {
            }
            return TTS_OK;
      }
      virtual TTS_RESULT      onInformationMessage(
const char *pszInformationMessage)
      {
            try {
```

```
			cout << pszInformationMessage << "\n";
		}
		catch(...) {
		}
		return TTS_OK;
	}
	virtual TTS_RESULT	onTraceMessage(int nTraceLevel,
const char *pszTraceMessage)
	{
		try {
			cout << pszTraceMessage << "\n";
		}
		catch(...) {
		}
		return TTS_OK;
	}
protected:
	virtual ~CSimpleEngine(void)
	{
	}
};
```

6.5. Initializing and Shutting Down the Engine

After creating a CTTSEngine object, the application must call the CTTSEngine::Initialize() function. If the CTTSEngine::Initialize() call returns a successful status code, then the application must call CTTSEngine::Shutdown() when it is finished with the CTTSEngine object.
These calls must be symmetrical, for every CTTSEngine::Initialize() there should be a CTTSEngine::Shutdown().

```
// Initialize the engine
TTS_RESULT result = pEngine->Initialize();
if (SUCCEEDED(result)) {
	. . . speak etc.

	// we are done, so shut down the engine
	pEngine->Shutdown();
}
```

6.6. Setting the Voice

TTSVoice Structure

An application uses a TTSVoice structure to enumerate, set or query the current voice the engine is using.

TTSVoice member	**Description**
m_nGender	Indicates the gender. The possible values: TTSGENDER_DEFAULT – use the default gender TTSGENDER_MALE – male voice TTSGENDER_FEMALE – female voice
m_nAge	Indicates the TTSAge enumeration. The possible values are: TTSAGE_CHILD – child voice TTSAGE_TEEN – teenage voice TTSAGE_ADULT – adult voice

	TTSAGE_SENIOR – senior voice
m_szName	The name of the voice. If this is "", then the default voice will be used.
m_szLocale	This is the language locale ("en_us", "es_us", etc) for the voice. If this is "", then the default locale will be used.
m_nSampleRate	This the sample rate of the voice, either 8000Hz or 16000Hz. Both are 16 bit PCM.

Enumerating the Voices

An application can find out how many voices are available by calling the CTTSEngine::NumVoices function and can then enumerate their characteristics using the CTTSEngine::EnumVoice function.

```
// How many voices are there?
int nVoices = 0;
TTS_RESULT result = pEngine->NumVoices(&nVoices);
if (SUCCEEDED(result) && nVoices) {
      for (int i = 0; i < nVoices; i++) {
            TTSVoice voice;
            result = pEngine->EnumVoice(i, &voice);
            if (SUCCEEDED(result)) {
                  cout << voice.m_szName;
            }
      }
}
```

Setting the Voice

An application can set the voice using the CTTSEngine::SetVoice function. The application should be aware that the returned voice may not be exactly what the application desired. The SDK will select the closest supported voice for the supplied TTSVoice structure. The application may check the returned TTSVoice structure to determine the characteristics of the actual voice selected by the SDK.

```
// Set the male voice
TTSVoice voice, voiceReturned;
voice.m_nGender = TTSGENDER_MALE;
TTS_RESULT result = pEngine->SetVoice(&voice, &voiceReturned);
if (SUCCEEDED(result)) {
      // Did it really change it?
      if (voiceReturned.m_nGender == TTSGENDER_MALE) {
            // Yes!
      }
}
```

6.7. Setting the Audio Format

TTSAudioFormat Structure

An application uses a TTSAudioFormat structure to enumerate, set or query the current audio format the engine is using.

TTSAudioFormat member	**Description**
m_nType	Indicates the type of audio format. Possible values are: TTSAUDIOTYPE_DEFAULT – the default format. For the current server, the default format is TTSAUDIOTYPE_LINEAR.

	TTSAUDIOTYPE_MULAW - this is CCITT g.711 mulaw TTSAUDIOTYPE_LINEAR – this is linear PCM TTSAUDIOTYPE_ALAW - this is CCITT g.711 alaw
m_nSampleRate	This is the sample rate. Set it to 0 to use the default sample rate in samples per second for the selected audio type. This will be either 8000 or 16000.
m_nBits	This is the number of bits per sample. Set it to 0 to use the default number of bits. For the current server, this will either be 8 or 16 bits.
m_nReserved	This is reserved field, please do not use it.

Enumerating the Audio Formats

An application can find out how many audio formats there are by calling the CTTSEngine::NumAudioFormats function and can then enumerate their characteristics using the CTTSEngine::EnumAudioFormat function.

```
// How many audio formats are there?
int nFormats = 0;
TTS_RESULT result = pEngine->NumAudioFormats(&nFormats);
if (SUCCEEDED(result) && nFormats) {
      for (int i = 0; i < nFormats; i++) {
            TTSAudioFormat format;
            result = pEngine->EnumAudioFormat(i, &format);
            if (SUCCEEDED(result)) {
                  cout << format.m_nType;
            }
      }
}
```

Setting the Audio Format

An application can set the audio format using the CTTSEngine::SetAudioFormat function. The application should be aware that the returned audio format may not be exactly what the application desired. The SDK will select the closest supported audio format for the TTSAudioFormat structure supplied by the application. The application may check the returned TTSAudioFormat structure to determine the characteristics of the actual audio format selected by the SDK.

```
// Set to linear PCM
TTSAudioFormat format, formatReturned;
format.m_nType = TTSAUDIOTYPE_LINEAR;
TTS_RESULT result = pEngine->SetAudioFormat(&format, &formatReturned);
if (SUCCEEDED(result)) {
      // Did it really change it?
      if (formatReturned.m_nType == TTSAUDIOTYPE_LINEAR) {
            // Yes!
      }
}
```

6.8. Speaking Text

An application speaks text using the CTTSEngine::Speak() function. If the application has created a TTSENGINE_SYNCHRONOUS engine then the CTTSEngine::Speak() call will not return until all the audio and other notifications have been sent to the CTTSSink object. If the application has created a TTSENGINE_ASYNCHRONOUS engine then the CTTSEngine::Speak() call will return immediately. Applications using the asynchronous model optionally get a TTSSPEAKHANDLE when making a CTTSEngine::Speak call which can be used to associate Notifications with a particular asynchronous Speak call.

An application can queue multiple speak requests by calling the CTTSEngine::Speak() function multiple times.

Speaking Text

The SDK requires UTF-8 character input for internationalization. The SDK supports SSML text markup languages. The application uses the CTTSEngine::SpeakFlags argument of the CTTSEngine::Speak() call to tell the SDK what kind of text the application is sending.

CTTSEngine::Speak Flags	Description
sf_default	Default text string with no tags or XML information.
sf_sapixml	SSML formatted text.

```
// Speak some text
string szText = “hello!”;
TTS_RESULT result = pEngine->Speak((PUTF8String) szText.c_str(),
CTTSEngine::sf_default);
if (FAILED(result)) {
      cout << CTTSResult::GetErrorString(result);
}
```

Speaking CTTSTextFragment objects

The SDK also supports speaking CTTSTextFragment objects. The application implements a CTTSTextFragment derived object and passes a list of CTTSTextFragment objects to the SDK through the text fragment version of the CTTSEngine::Speak(const list<CTTSTextFragment *> &lstFragments) call. When passed a list of fragments, the SDK will interpret the CTTSTextFragment object and call its functions to retrieve information pertaining to the fragment.

A CTTSTextFragment object describes a fragment of text. Each fragment has an action which determines how to interpret the data within the fragment.

CTTSTextFragment::Actions	Description	
a_speak	The fragment contains text to speak. The member functions are interpreted as follows:	
	Text()	The text to speak.
	TextLength()	The length of the text.
	TextOffset()	The offset of the text from the beginning of the original text buffer.
	PartOfSpeech()	The part of speech for the text.
	EmphasisAdjustment()	Is the text emphasized or not?
	RateAdjustment()	Speech rate adjustment for the text above or below normal rate.
	VolumeAdjustment()	Speech volume adjustment for the text above or below normal rate.
	PitchAdjustment()	Speech pitch adjustment for the text above or below the normal pitch.
	Context()	Context flags for interpreting the text.
a_silence	The fragment contains silence. The member functions are interpreted as follows:	
	Silence()	Milliseconds of silence to insert.
a_pronounce	The fragment contains text with an associated phonemic pronunciation. The member functions are interpreted as for a_speak with the addition of:	

	PhonemeIDs()	DARPA phonemes that indicate the pronunciation to use for the given text.
a_bookmark	The fragment contains a bookmark. The member functions are interpreted as follows:	
	Text()	The ID of the bookmark.
	TextLength()	The length of the bookmark.
	TextOffset()	The offset in bytes from the beginning of the original text buffer.
a_spell	The fragment contains text to spell. The member functions are interpreted as for a_speak.	
a_section	The fragment contains a section of text. Not currently implemented.	

6.9. Setting Volume and Rate

An application can change the default speaking volume and rate with the CTTSEngine::SetVolume() and CTTSEngine::SetRate() functions. These functions may be called from a separate thread or from within the same thread.
Volume is expressed as a percentage of the normal speaking volume of a voice. The default value is 100, but can range from 0 to 500. *Note: Setting the volume above 200 may result in distortion of the voice.*
Speaking rate is expressed on a logarithmic scale from –18 to 18, representing 1/8 the normal rate to 8 times the normal rate. The default value of 0 indicates the normal speaking rate of the voice.

```
// Rate and volume adjustment
pEngine->SetRate(-10);        // Approximately 1/3 normal rate
pEngine->SetVolume(25); // ¼ normal speaking volume
pEngine->Speak((PCUTF8String)”A slow whisper”, 14, CTTSEngine::sf_default);
pEngine->SetRate(10);   // Approximately 3 times normal rate
pEngine->SetVolume(200);      // twice normal speaking volume
pEngine->Speak((PCUTF8String)”A quick shout!”, 14, CTTSEngine::sf_default);
```

6.10. Stopping the Engine

An application may call the CTTSEngine::Stop() function to stop the engine from speaking text. All queued speaking requests will be stopped. This function may be called from a separate thread or from within the same thread.

```
// Stop speaking
TTS_RESULT result = pEngine->Stop();
```

6.11. Retrieving Phonetic Transcriptions

You may wish to find the phonetic transcription of a word or phrase to use to modify or create a new dictionary item. If you’re not a linguist, it may be easiest to retrieve the phonetic transcription of a word or phrase from the TTS engine.

```
TTSTranscription myTranscription;
myTranscription.m_utfWord = "frog";
TTS_RESULT res = pEngine->Transcribe(myTranscription);
if (SUCCEEDED(res)) {
      cout << myTranscription.m_utfTranscription;
}
```

6.12. Custom Dictionaries

An application can specify pronunciations to be used while speaking text through the use of custom dictionaries. A dictionary is a list of entries consisting of a word, and one or more phonemic pronunciations for that word. The format of a dictionary entry is as follows:

Word {phone1-1 phone1-2 phone1-3 …}(/part-of-speech) {phone2-1 …}

The "Word" is the textual representation of the word whose pronunciation is being specified. It is followed by one or more transcriptions. A transcription consists of an open brace ({) followed by a space-separated list of DARPA phoneme names for English words or SAMPA phoneme names for words of all other languages. The last phoneme is followed by a closing brace (}). This may be optionally followed by a slash (/) and a part of speech designation, (noun, verb, modifier, function, or interjection). If a part of speech is present, that pronunciation will only be used for words whose part of speech matches. Here is a small sample dictionary:

```
// Sample Dictionary
tomato {t ow - m ey 1 t ow}
lead {l iy d} {l eh d}/noun
```

Dictionaries are managed through the following set of member functions in the CTTSEngine class.

Creating a Dictionary

An application creates a dictionary with the CTTSEngine::CreateDictionary() function. The application supplies a name by which the dictionary can be referenced, a weight, used to specify the order in which dictionaries are sorted, (the higher the weight, the earlier the dictionary is searched), and the contents of the dictionary.

```
// Creating a dictionary - Be sure to surround the phonemes with curly braces
PCUTF8String pszMyDictionary = "\n\
tomato {t ow - m ey 1 t ow}\n\
lead {l iy d} {l eh d}/noun\n\
";

PCUTF8String pszMyDictName = "MyDictionary";

TTS_RESULT result = pEngine->CreateDictionary(pszMyDictName, 0, pszMyDictionary,
(PCUTF8String)"att_darpabet_english");
```

Updating a Dictionary

Once a dictionary has been created by an application, it can add and replace transcriptions using the CTTSEngine::UpdateDictionary() function.

```
// Updating a dictionary
PCUTF8String pszMyDictionaryUpdate = "\n\
tomato {t ow - m aa 1 t ow}\n\
potato {p ow - t ey 1 t ow}\n\
";

TTS_RESULT result = pEngine->UpdateDictionary(pszMyDictName,
szMyDictionaryUpdate);
```

Clearing and Deleting a Dictionary

An application can clear all entries from a dictionary with the CTTSEngine::ClearDictionary() function. This leaves the dictionary in place with its current weight, but removes all entries from the dictionary. This call is useful if the application plans to completely replace the contents of a dictionary without losing its position in the dictionary search list. If an application is completely finished using a dictionary, it can be completely removed with the CTTSEngine::DeleteDictionary() function.

```
// Clearing and deleting a dictionary
pEngine->ClearDictionary(pszMyDictName);
pEngine->UpdateDictionary(pszMyDictName, pszMyDictionaryUpdate);
…
pEngine->DeleteDictionary(psz8MyDictName);
```

Changing Search Order

An application can change the order in which its custom dictionaries are searched by changing the weight of a dictionary. Increasing the weight will move the dictionary closer to the beginning of the list. Decreasing the weight will move it closer to the end of the list. Dictionary weights are set when the dictionary is created, and can be modified later with the CTTSEngine::ChangeDictionaryWeight() function.

```
// Change search order
pEngine->CreateDictionary((PCUTF8String)”Dict1”, 5, 0);
pEngine->CreateDictionary((PCUTF8String)”Dict2”, 5, 1);
// Current search order is dict2 (weight 1) followed by dict1 (weight 0)
pEngine->ChangeDictionaryWeight((PCUTF8String)”Dict1”, 2);
// New search order is dict1 (weight 2) followed by dict2 (weight 1)
```

6.13. Optional Operating System Specific Classes

There are some classes defined that an application may use if desired.

CTTSWin32AudioPlayer

A Win32 audio player class is provided and used by the TTSDesktopPlayer application. It plays audio using the Microsoft WAVE API.

CTTSWin32AudioWriter

A Win32 RIFF WAVE file writer class is provided and used by the TTSDesktopFile application. It will write .WAV files for an application.

CTTSUnixAudioPlayer

A LINUX audio player class is provided and used by the TTSClientPlayer application.

CTTSUnixAudioWriter

A UNIX audio writer class is provided and used by the TTSClientFile class. It writes audio out to raw files.

7. SSML Control Tags

The AT&T Natural Voices TTS engine does a great job of synthesizing most text without special instructions, but there may be special circumstances where you wish to fine-tune the pronunciation of certain words or phrases. The AT&T Natural Voices TTS engine allows users to mark up the text to be spoken to include special control tags that change the way the text is pronounced. The AT&T Natural Voices TTS engine supports a subset of the SSML control tags and adds a few extras. Not all tags are supported in all languages. Control tags as case-insensitive.

XML Tag	US	ES
<ATT_Ignore_Case>	Yes	Yes
<speak> text </speak>	Yes	Yes
<speak xml:lang=”en_us”> text </speak>	Yes	Yes
<paragraph> text </paragraph>	Yes	Yes
<p> text </p>	Yes	Yes
<sentence> text </sentence>	Yes	Yes
<s> text </s>	Yes	Yes
<say-as type=”ATT_Literal”> text </say-as>	Yes	No
<say-as type=”ATT_Math”> text </say-as>	Yes	No
<say-as type=”ATT_Measurement”> text </say-as>	Yes	No
<say-as type=”acronym”> text </say-as>	Yes	No
<say-as type=”number”> text </say-as>	Yes	Yes
<say-as type=”number:ordinal”> text </say-as>	Yes	Yes
<say-as type=”number:digits”> text </say-as>	No	No
<say-as type=”date”> text </say-as>	Yes	Yes
<say-as type=”date:dmy”> text </say-as>	No	No
<say-as type=”date:mdy”> text </say-as>	Yes	Yes
<say-as type=”date:ymd”> text </say-as>	No	No
<say-as type=”date:ym”> text </say-as>	No	No
<say-as type=”date:my”> text </say-as>	Yes	Yes
<say-as type=”date:md”> text </say-as>	Yes	Yes
<say-as type=”date:y”> text </say-as>	Yes	No
<say-as type=”date:m”> text </say-as>	Yes	Yes
<say-as type=”date:d”> text </say-as>	No	No
<say-as type=”time”> text </say-as>	Yes	No
<say-as type=”time:hms”> text </say-as>	Yes	No
<say-as type=”time:hm”> text </say-as>	Yes	No
<say-as type=”time:h”> text </say-as>	Yes	No
<say-as type=”duration”> text </say-as>	No	No
<say-as type=”duration:hms”> text </say-as>	No	No
<say-as type=”duration:hm”> text </say-as>	No	No
<say-as type=”duration:h”> text </say-as>	No	No

<say-as type=”duration:m”> text </say-as>	No	No
<say-as type=”duration:s”> text </say-as>	No	No
<say-as type=”currency”> text </say-as>	Yes	Yes
<say-as type=”telephone”> text </say-as>	Yes	No
<say-as type=”name”> text </say-as>	Yes	Yes
<say-as type=”net”> text </say-as>	Yes	Yes
<say-as type=”net:email”> text </say-as>	Yes	Yes
<say-as type=”net:url”> text </say-as>	Yes	Yes
<say-as type=”address”> text </say-as>	Yes	Yes
<say-as sub=”sub”> text </say-as>	Yes	Yes
<phoneme alphabet=”att_darpabet_english” ph=”*phonemes*”> *orthography* </phoneme>	Yes	No
<phoneme alphabet=”att_sampa_spanish” ph=”*phonemes*”> *orthography* </phoneme>	No	Yes
<emphasis> text </emphasis>	No	No
<voice gender=”male, female, neutral” >…</voice>	Yes	Yes
<voice age=“integer” >…</voice>	Yes	Yes
<voice category=“child, teenager, adult, elder” >…</voice>	Yes	Yes
<voice variant>…</voice>	No	No
<voice name=”mike8, crystal16, rosa8, etc”>…</voice>	Yes	Yes
<break/>	Yes	Yes
<break size=“none, small, medium, large” />	Yes	Yes
<break time=”*integer*”/>	Yes	Yes
<prosody pitch=”high, medium, low, default”> … </prosody>	No	No
<prosody contour=”…”> … </prosody>	No	No
<prosody range=”…”> … </prosody>	No	No
<prosody rate=”fast, medium, slow, default” > … </prosody>	Yes	Yes
<prosody duration=”…”> … </prosody>	No	No
<prosody volume=”silent, soft, medium, loud, default” > … </prosody>	Yes	Yes
<!ENTITY … !>	No	No
Audio	No	No
Mark	Yes	Yes

ATT_Ignore_Case

Tag tells the TTS engine to ignore the capitalization of words within the specified context. This mode is useful to override the default behavior which is to spell all capitalized words.

Syntax: <ATT_IGNORE_CASE> *text* </ATT_IGNORE_CASE>
Example: <ATT_ignore_case> THIS CONTRACT </ATT_ignore_case>
Note: Pronounced as “this contract”

Break

The tag instructs the TTS engine to insert a pause in the synthesized text in one of three ways.

Syntax #1: <BREAK/>
Example: Time for a pause <Break/> Okay, keep going.
Note: Inserts a brief break after the word “pause”.

Syntax #2: <BREAK Size=”none | small | medium | large”/>
Example: No time for a pause <Break size=”none”/> Keep going.
Note: Inserts no break after the word “pause”.
Example: Time for a pause <Break size=”medium”/> Okay, keep going.
Note: Inserts a brief silence, the equivalent of the silence following a sentence, after the word “pause”.
Example: Time for a pause <Break size=”large”/> Keep going.
Note: Inserts only the default break after the word “pause”.
Example: Time for a pause <Break size=”medium”/> Okay, keep going.
Note: Inserts the equivalent of a paragraph break of silence after the word “pause”.

Syntax #3: <BREAK time=” duration ”/>
Example: Break for 100 milliseconds <Break time=”100ms”/> Okay, keep going.
Note: Inserts 100 milliseconds of silence after the word “milliseconds”.
Example: Break for 3 seconds <Break time=”3s”/> Okay, keep going.
Note: Inserts 3 seconds of silence after the word “seconds”.

Mark

The application may insert a user bookmark in the text. The TTS Engine will optionally inform the application via a notification when that bookmark is reached in audio stream. Any number of bookmarks may be inserted anywhere in the text.

Syntax: <Mark Name=”text_label”/> text
Example: The quick <Mark Name=”mark 1”/> fox jumped over the lazy <Mark Name=”mark 2”/> dog.
Note: The TTS engine will provide a notification just before speaking the word “fox” and another notification just before speaking the word “dog”.

Paragraph

This tag tells the TTS engine to change the prosody to reflect the end of a paragraph, regardless of the surrounding punctuation.

Syntax: <PARAGRAPH> text </PARAGRAPH> or <P> text
Example: <P> The paragraph tag can be abbreviated as just the letter P.
Note: The TTS engine changes the prosody to reflect the paragraph boundaries.

Phoneme

This tag allows the user to specify pronunciations explicitly in the input text. Including the orthography, i.e. the word or words represented by the phonemes, are optional, but recommended, because some modules in the front end depend on orthography. The pronunciations must be represented using the DARPA or SAMPA phoneme set when using the AT&T Natural Voices TTS Client SDK. Complete description of the DARPA and SAMPE phoneme sets is later in this manual.

Syntax: <PHONEME alphabet="att_darpabet_english" ph=" phoneme$^+$ "/>
Syntax: <PHONEME alphabet="att_sampa_spanish" ph=" phoneme$^+$ "> orthography </PHONEME>

Example: <Phoneme alphabet="att_darpabet_english" ph="b ow t 1"/>
Example: <Phoneme alphabet="att_sampa_english" ph="b ow t 1"/>
Example: <Phoneme alphabet="att_sampa_spanish" ph="p a 1 D r e"> padre </Phoneme>

Prosody

We support only the Rate and Volume attributes of the SSML Prosody tag. The Pitch and Contour tags are not supported.

Prosody > Rate

The Rate attribute of the Prosody tag changes the rate at which the text is spoken. You can specify either the absolute rate or a relative change in the current speaking rate. This release supports a range of up to eight times faster or slower than the default speed.

Syntax: <PROSODY RATE="fast | medium | slow | default"> content </PROSODY>
Syntax: <PROSODY RATE=" RelativeChange "> text </PROSODY>
Note: This changes the speaking rate which is expressed in Words per Minute (WPM). RelativeChange is a floating point number that is added to the current rate. Using a RelativeChange < 0 decreases the speed.
Example: This is the default speed <prosody rate="slow"> this is speaking slowly <prosody rate="fast"> this is speaking fast </prosody> back to slow </prosody> back to the default rate.
Example: This is the default speed <prosody rate="-50%"> this is 50% slower <prosody rate="+100%"> this is 50% faster than normal </prosody> back to 50% slower </prosody> back to the default rate.

Prosody > Volume

The Volume attribute of the Prosody tag allows the application to change the volume of the TTS voice. Note that this does not change the volume of the output device, but it does raise or lower the volume of the text spoken within the context of the tag.

Syntax: <PROSODY VOLUME=" level "> text </PROSODY>
Note: Level is a value from 0.0 to 200.0. A value of 100 is the voice's default volume, a value of 0 changes the volume to 0 and a value of 200 doubles the volume. The volume changes linearly.
Syntax: <PROSODY VOLUME=" silent | soft | medium | loud | default "> text </PROSODY>
Syntax: <PROSODY VOLUME=" delta "> text </PROSODY>
Example: This is the default volume <prosody volume="silent"> silence </prosody> <prosody volume="soft"> Now I'm whispering </prosody> <prosody volume="+20%"> a little louder </prosody> <prosody volume="medium"> medium volume</prosody> <prosody volume="+20%"> a little louder </prosody> <prosody volume="loud"> very loud </prosody> <prosody volume="default"> back to default volume </ prosody>.

Say-As

Say-As tags provide contextual hints to the TTS engine about how text should be pronounced. The TTS engine supports a number of different contexts that can be used to fine-tune the pronunciation of words.

Say-As > Acronym

The Acronym context tells the TTS engine to treat the text as an acronym and to pronounce the text as the letters in the words. This tag is especially useful if your text is mostly upper case and you use the ATT_Ignore_Case tag but then encounter an acronym.

Syntax: <SAY-AS Type="Acronym"> text </SAY-AS>
Example: MADD <Say-as type="acronym"> MADD </Say-as>
Note: Pronounced as "mad M-A-D-D".

Say-As > Address

The Address context tells the TTS engine to treat the text as an address.

Syntax: <SAY-AS Type="Address"> text </SAY-AS>
Example: <Say-as Type="Address"> 123 Main St. , New York, NY 10017 </Say-as>
Note: Will be pronounced "one twenty three main street, New York, New York one zero zero one seven"

Say-As > ATT_Literal

The ATT_Literal context instructs the TTS engine to pass the string through literally without applying additional context processing such as abbreviations, addresses, dates, math symbols, measurements, names, numbers, times, or phone numbers.

Syntax: <SAY-AS Type="ATT_Literal"> text </SAY-AS>
Example: <Say-as type="ATT_Literal"> misc. </Say-as> misc.
Note: Pronounced as "misk miscellaneous"

Say-As > ATT_Math

The ATT_Math context tells the TTS engine to treat the text as a mathematical expression.

Syntax: <SAY-AS Type="ATT_Math"> text </SAY-AS>
Example: <Say-as Type="ATT_Math"> 3+4=7 </Say-as> 3+5=8
Note: Pronounced as "three plus four equals seven three plus sign five equal sign eight"

Say-As > ATT_Measurement

The ATT_Measurement context tells the TTS engine to treat the text as a measurement, e.g. single quotes are pronounced "feet" and double quotes are pronounced "inches".

Syntax: <SAY-AS Type="ATT_Measurement"> text </SAY-AS >
Example: <Say-as Type="ATT_Measurement"> 5' 3" </Say-as> 7' 9"
Note: Pronounced as "five feet three inches seven single quote nine double quote"

Say-As > Currency

The Currency context tells the TTS engine to treat the text as currency and expand $ and decimal numbers appropriately. The Currency Context works only with US currency and not other currencies.

Syntax: <SAY-AS Type="Currency"> text </SAY-AS>
Example: <Say-as Type="Currency"> $25.32 </Say-as>
Note: Pronounced as "twenty five dollars and thirty two cents"

Say-As > Date

The Date context tells the TTS engine to treat the text as date. You may also add qualifiers to provide even more information to the TTS engine but in general the extra qualifiers are not needed.

Syntax: <SAY-AS Type=”Date”> text </SAY-AS>
Example: <Say-as Type=”Date”> Dec 25, 2001 </Say-as>
Note: Pronounced as “December twenty fifth two thousand one”
Syntax: <SAY-AS Type=”Date:M”> text </SAY-AS>
Example: <Say-as Type=”Date:M”> Dec </Say-as>
Note: Pronounced as “December”
Syntax: <SAY-AS Type=”Date:MD”> text </SAY-AS>
Example: <Say-as Type=”Date:MD”> Dec 25</Say-as>
Note: Pronounced as “December twenty fifth”
Syntax: <SAY-AS Type=”Date:MDY”> text </SAY-AS>
Example: <Say-as Type=”Date:MDY”> Dec 25, 2001 </Say-as>
Note: Pronounced as “December twenty fifth two thousand one”
Syntax: <SAY-AS Type=”Date:MY”> text </SAY-AS>
Example: <Say-as Type=”Date:MY”> Dec, 2001 </Say-as>
Note: Pronounced as “December two thousand one”
Syntax: <SAY-AS Type=”Date:Y”> text </SAY-AS>
Example: <Say-as Type=”Date:Y”> 2001 </Say-as>
Note: Pronounced as “two thousand one”

Say-As > Name

AT&T Natural Voices TTS engine does a great job on names without XML tags but you can tell the engine to expect an name with the SAY-AS name tag.

Syntax: <SAY-AS Type=”Name”> text </SAY-AS>
Example: <Say-as Type=”Name”> Mark Beutnagel </Say-as>
Note: Pronounced as “Mark Beutnagel”

Say-As > Net

The Net type tells the engine to expect either an email address or a URL.

Syntax: <SAY-AS Type=”Net”> text </SAY-AS>
Example: <Say-as Type=”Net”> help@naturalvoices.att.com </Say-as>
Note: Pronounced as “help at natural voices dot ATT dot com”
Example: <Say-as Type=”Net”> http://naturalvoices.att.com </Say-as>
Note: Pronounced as “H T T P natural voices dot ATT dot com”
Syntax: <SAY-AS Type=”Net:email”> text </SAY-AS>
Example: <Say-as Type=”Net:email”> help@naturalvoices.att.com </Say-as>
Note: Pronounced as “help at natural voices dot ATT dot com”
Syntax: <SAY-AS Type=”Net:URL”> text </SAY-AS>
Example: <Say-as Type=”Net:URL”> help@naturalvoices.att.com </Say-as>
Note: Pronounced as “help at natural voices dot ATT dot com”

Say-As > Number

The Number type tells the engine to expect a number.
Syntax: <SAY-AS type=”Number”> text </SAY-AS>
Example: <Say-as type=”Number”> 10,243 </Say-as>
Note: Pronounced as “ten thousand two hundred forty three”
Syntax: <SAY-AS type=”Number:Decimal”> text </SAY-AS>
Example: <Say-as type=”Number:decimal”> 3.14159 </Say-as>
Note: Pronounced as “three point one four one five nine”

Syntax: <SAY-AS type=”Number:Fraction”> text </SAY-AS>
Example: <Say-as type=”Number:Fraction”> 5 3/4 </Say-as>
Note: Pronounced as “five and three fourths”
Syntax: <SAY-AS type=”Number:Ordinal”> text </SAY-AS>
Example: <Say-as type=”Number:ordinal”> VI </Say-as>
Note: Pronounced as “sixth”

Say-As > Sub

The SUB tag allows you to substitute spoken text for the written text.

Syntax: <SAY-AS sub=”spoken”> text </SAY-AS>
Example: <Say-as sub=” Mothers Against Drunk Driving”> MADD </Say-as>
Note: Pronounced as “mothers against drunk driving”

Say-As > Telephone

The Telephone context tells the TTS engine to treat the text as a telephone number.

Syntax: <SAY-AS type=”Telephone”> text </SAY-AS>
Example: <Say-as type=”telephone”> (212)555-1212 </Say-as>
Note: Pronounced as “two one two five five five one two one two”

Say-As > Time

The Time context tells the TTS engine to treat the text as a time.

Syntax: <SAY-AS Type=”Time”> text </SAY-AS>
Example: <SAY-AS type=”Time”> 12:34 PM </SAY-AS>
Note: Pronounced as “twelve thirty four P M”
Syntax: <SAY-AS Type=”Time:HMS”> text </SAY-AS>
Example: <SAY-AS type=”Time”> 12:34:56 PM </SAY-AS>
Note: Pronounced as “twelve thirty four and fifty six seconds P M”
Syntax: <SAY-AS Type=”Time:HM”> text </SAY-AS>
Example: <SAY-AS type=”Time”> 12:34 PM </SAY-AS>
Note: Pronounced as “twelve thirty four P M”
Syntax: <SAY-AS Type=”Time:H”> text </SAY-AS>
Example: <SAY-AS type=”Time”> 12 PM </SAY-AS>
Note: Pronounced as “twelve P M”

Sentence

This tag tells the TTS engine to change the prosody to reflect the end of a sentence, regardless of the surrounding punctuation. The TTS engine changes the prosody to reflect the sentence boundaries.

Syntax: <SENTENCE> text </SENTENCE> or <S> text </S>
Example: <Sentence> This text is a sentence. </Sentence>
Example: <S> The sentence tag can be abbreviated as just the letter S. </S>

Speak

The SPEAK tag tells the TTS engine to synthesize the specified text. You need not include this tag to have your text synthesized.

Syntax: <SPEAK> text </SPEAK>
Example: <Speak> This text is synthesized. </Speak>

Note: Pronounced as “this text is synthesized.”

Voice

The Voice tag allows the application to change the voice of the TTS speaker from the input text. You can use this feature to change voices, e.g. you might use different voices to speak different sections of an email message or carry on a conversation between two different voices. The default voice for a server is specified when the server process is started. You choose a voice by specifying one or more of the following attributes:

Gender: male, female, or neutral
Age: an integer value, e.g. 30
Category: child, teen, adult, or elder
Name: mike8, crystal8, rosa16, rich16, or any other voices you’ve installed.
Language: “en_us” “English” for US English
“es_us” or “Spanish” for Spanish

You can specify several attributes but in the current release it is best to specify the speaker by name. You can purchase additional voices which may make more use of the additional attributes. Note that switching voices forces the server to load the data files for the each voice that is specified which may result in noticeable delays. Voice switches will happen instantaneously once the voice data is in place.

Syntax: <VOICE Gender=”male | female | neutral” Age=”integer” Category=”child | teen | adult | elder” Language=”en_us | …” Name=”mike8 | …” </VOICE>
Example: <Voice Name=”mike8”> Hi, I’m Mike </Voice>
Note: Pronounced as “Hi, I’m Mike” using Mike’s 8 KHz voice.
Example: <Voice Name=”crystal8”> I’m Crystal </Voice>
Note: Pronounced as “I’m Crystal” using Crystal’s 8 KHz voice.
Example: <Voice Name=”rosa8”> Hola, me llamo Rosa </Voice>
Note: Pronounced as “Hola, me llamo Rosa” using Rosa’s 8KHz voice.
Example: <voice name=”mike16”> This is Mike <Voice Name=”crystal16”> This is Crystal </voice> This is Mike again. </voice>
Note: Pronounced as: in Mike’s voice “This is Mike”, then in Crystal’s voice, “This is Crystal”, then in Mike’s voice, “This is Mike again”.

8. Custom Dictionaries

The AT&T Natural Voices TTS engine allows you to define custom dictionaries that change the default pronunciation of words. The AT&T Natural Voices TTS engine does a great job pronouncing most words and even most names correctly but there are some words that it doesn't get quite right. Custom dictionaries allow you to change pronunciations of words for your users or application.

When searching for a transcription, the TTS Server first searches custom dictionaries and then its own internal dictionaries to find some pronunciation for the word. The search stops as soon as a pronunciation is found.

Defining Custom Pronunciations

The first step in building custom dictionaries is to define new pronunciations for words or phrases using the DARPA phonemes for English or the SAMPA phonemes for all other languages. Finding just the right set of phonemes to make a word can be tricky. To find word that sounds like the word you wish to say, experiment using the SSML <PHONEME> tag.

For example, say that you want the TTS engine to say "toe MAH toe" rather than "toe MAY toe".

```
      <pron sym=" t ow m ey 1 t ow" /> is pronounced “toe MAY toe”
      <pron sym=" t ow m aa 1 t ow" /> is pronounced “toe MAH toe”
```

Adding stress marks can also help the TTS engine pronounce the word the right way. You can specify that phonemes should be unstressed (0), have primary (1) or secondary stress (2), e.g.
<pron sym="f ax 0 n aa 1 m aa 0 n aa n 2"> phenomenon </pron>.

Finding the proper set of phonemes to say a word in just the right way can be challenging. A colleague with expertise in linguistics is a great resource for this problem. The TTS engine does not expose the system lexicon to applications.

Adding Custom Pronunciations to a Dictionary File

Once you have defined your words and their pronunciations, you need to put them into a dictionary. A Dictionary is a flat text file which holds one or more "word { phonemes }" combinations. It is a good idea to maintain the same file extension (we suggest `.dict`) for all your custom dictionaries and keep those files in the directory according to the following table:

Language	Directory	Configuration File
US English	`<INSTALL_DIR>/data/en_us`	`en_us.cfg`
Spanish	`<INSTALL_DIR>/data/es_us`	`es_us.cfg`

Following is an example of custom dictionary file:

```
DOS {d ao s 0}
forget { f er 0 g eh t 1  }
linux { l ih n 1 ax k s }
```

Letting TTS Engine Accept Your Custom Dictionary

Locate a language model configuration file (last column from the table in previous paragraph). This is how typical English language model configuration file looks like:

```
Language                en_us
LanguageLocale          en_us
LanguageDictionary      ../data/en_us/en_us.dict att_darpabet_english
LanguageTextAnalysis    ../lib/ft_en_us.so
TextAnalysisLoad        dynamic
```

To use a dictionary file `new_custom_dictionary.dict`, add a new line as follows:

```
Language                en_us
LanguageLocale          en_us
LanguageDictionary      ../data/en_us/en_us.dict att_darpabet_english
LanguageTextAnalysis    ../lib/ft_en_us.so
UserDictionary          ../data/en_us/new_custom_dictionary.dict
TextAnalysisLoad        dynamic
```

Save the file. The TTS engine will use the transcriptions in `new_custom_dictionary.dict`.

9. Phonetic Alphabets

DARPA US English Phonetic Alphabet [att_darpabet_english]

Phoneme	Example	Transcription
Aa	Bob	b aa b 1
Ae	bat	b ae t 1
Ah	but	b ah t 1
Ao	bought	b ao t 1
Aw	down	d aw n 1
Ax	about	ax 0 b aw t 1
ay	bite	b ay t 1
b	bet	b eh t 1
ch	church	ch er ch 1
d	dig	D ih g
dh	that	dh ae t 1
dx	butter	b ah 1 dx er 0
eh	bet	b eh t 1
em	Chatham	ch ae 1 dx em 0
en	satin	s ae 1 q en 0
er	bird	b er d 1
ey	bait	b ey t 1
f	fog	f ao g 1
g	got	g aa t 1
hh	hot	hh aa t 1
ih	bit	b ih t 1
iy	beat	b iy t 1
jh	jump	jh ah m p 1
k	cat	k ae t 1
l	lot	l aa t 1
m	Mom	m aa m 1
n	nod	n aa d 1
ng	sing	s ih ng 1
ow	boat	b ow t 1
oy	boy	b oy 1
p	pot	p aa t 1
q	button	b ah 1 q en 0
r	rat	r ae t 1
s	sit	s ih t 1
sh	shut	sh ah t 1
t	top	t aa p 1
th	thick	th ih k 1
uh	book	b uh k 1
uw	boot	b uw t 1
v	vat	v ae t 1
w	won	w ah n 1
y	you	y uw 1
z	zoo	z uw 1
zh	measure	m eh 1 zh er 0
0	Unstressed	
1	Primary stress	
2	Secondary stress	
&	Word boundary	
pau	Silence	

SAMPA Spanish Phonetic Alphabet [att_sampa_spanish]

Transcription	Phoneme	Example
a 1 g w a 0	a	agua
e s 1 t e 0	e	este
i 0 g w a l 1	i	igual
o 1 s o 0	o	oso
u 1 b e 0	u	uve
o j 1 g o 0	j	oigo
f w e 1 r a 0	w	f w e 1 r a 0
p a r 1 t e 0	p	parta
t o 1 m a 0	t	toma
k o 0 x e r 1	k	coger
b a 1 k a 0	b	vaca
k a 1 d a 0	d	cada
a 0 g a 1 l o 0	g	hagalo
f l a 1 k o 0	f	flaco
s i 1	s	si
x e n 1 t e 0	x	gente
k o 1 tS e 0	tS	coche
a 0 jj e r 1	jj	ayer
jj a 1 b e 0	jj	llave
m i 1 jj a 0	jj	milla
m a 1 n o 0	m	mano
n a 1 d a 0	n	nada
e s 0 p a 1 J a 0	J	españa
l o 1 k o 0	l	loco
p e 1 r o 0	r	pero
p e 1 rr o 0	rr	perro