TTS API Provider

Hynek Hanke, Brailcom <hanke@brailcom.org>
Jan Buchal, Brailcom <buchal@brailcom.org>
Gary Cramblitt, KDE <garycramblitt@comcast.net>

Copyright © 2006, 2007 Brailcom, o.p.s. All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License."

A copy of the license is included in the section entitled "GNU General Public License"

Table of Contents

1 Introduction . . . 1

2 Design . . . 2
- 2.1 General Design . . . 2
- 2.2 TTS API Provider (core) . . . 3
- 2.3 Drivers . . . 4
- 2.4 Driver Template Library . . . 5
- 2.5 Convenience Libraries for Applications . . . 5

3 Server Implementation . . . 6
- 3.1 TTS API Provider core (implementation) . . . 6
- 3.2 Audio Subsystem (implementation) . . . 7

4 Device Driver Implementation . . . 8
- 4.1 Driver Interface . . . 8
- 4.2 Drivers in Python . . . 9
  - 4.2.1 Driver Design (Python) . . . 9
  - 4.2.2 Driver skeleton (Python) . . . 9
  - 4.2.3 Core (Python) . . . 9
  - 4.2.4 Controller (Python) . . . 10
  - 4.2.5 RetrievalSocket (Python) . . . 10
  - 4.2.6 Other tools (Python) . . . 11
- 4.3 Drivers in C . . . 11

5 TTS API Implementations . . . 12
- 5.1 Basic Usage of TTS API . . . 12
  - 5.1.1 Understanding the API . . . 12
  - 5.1.2 Initializing and closing a connection . . . 12
  - 5.1.3 Parameter settings . . . 12
  - 5.1.4 Speaking and audio retrieval . . . 12
  - 5.1.5 Callbacks and events . . . 12
  - 5.1.6 Serialized and simultaneous speech . . . 13
- 5.2 Text Protocol TTS API . . . 13
  - 5.2.1 General Rules (text protocol) . . . 14
  - 5.2.2 Return Codes . . . 14
  - 5.2.3 Driver Discovery (text protocol) . . . 15
  - 5.2.4 Voice Discovery . . . 17
  - 5.2.5 Speech Synthesis Commands (text protocol) . . . 17
  - 5.2.6 Speech Control Commands (text protocol) . . . 20
  - 5.2.7 Parameter Settings (text protocol) . . . 21
    - 5.2.7.1 Driver Selection and Parameters (text protocol) . . . 21
    - 5.2.7.2 Voice Selection (text protocol) . . . 21

5.2.7.3 Prosody Parameters (text protocol) 21
5.2.7.4 Style Parameters (text protocol) 23
5.2.7.5 Dictionaries 23
5.2.7.6 Audio Settings 23
5.2.8 Event Callbacks (text protocol) 24
5.2.9 Other Commands (text protocol) 24
5.3 Python TTS API 25

1 Introduction

TTS API Provider is a system service providing applications with low-level access to various synthesizers via TTS API (See TTS API specification, http://www.freebsoft.org/tts-api). It manages the synthesizers available on the system and provides a unified interface to them.

This API was developed in cooperation between the biggest Free Software accessibility projects at that time (Gnome Accessibility Project, KDE Accessibility, Free-b-Soft and others). It is made available publicly and separately in the hope to unify our efforts, revise the speech output architecture and put it on solid, well designed, well documented and de-facto standardized grounds. Its target is on accessibility, however we believe it is powerful enough to be useful in other areas too where speech synthesis is useful.

Our publishing of the API and making it available on the system for easy use should however not be seen as an encouragement for developers to use this low-level API directly! Most applications, especially accessibility applications, should never contact this low-level API directly. They should use a high-level API such as one of the Speech Dispatchers APIs, Gnome Speech API or KTTS API doing the necessary coordination of messages from various applications. To put it simple, one should only use TTS API Provider directly and fully if he is sure he is authorized to take complete control over all speech synthesis audio output on the system. As mentioned, this is mostly never the case of ordinary applications. In the audio world, the issue could be compared to applications taking control over the /dev/dsp device (under OSS) or intentionally bypassing the mixer (under ALSA) and thus blocking others.[1] An exception to this rule are applications on embedded systems, highly specialized applications like telephony servers, speech synthesis research and development applications and applications who only retrieve the synthesized audio samples but do not play them on the sound card (e.g. a text2wav converter). If in doubt, please contact us.

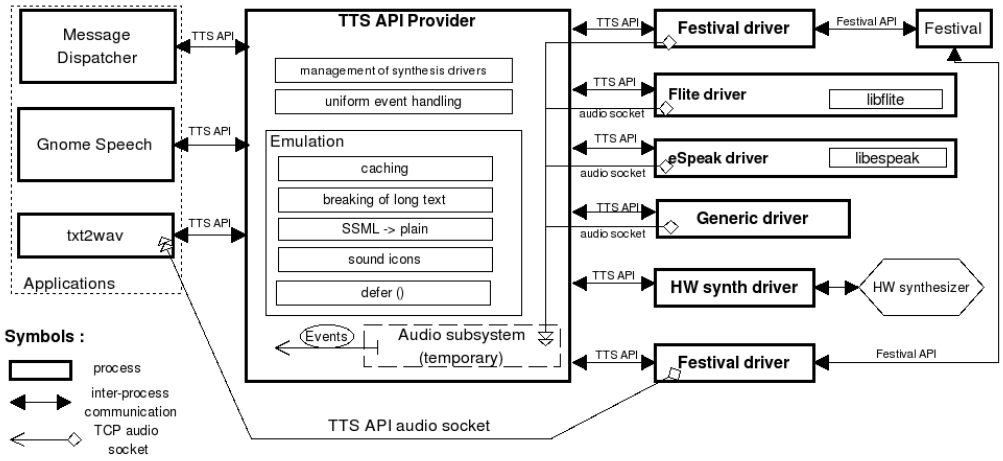
[1] While sound output for various concurrent speech streams is not a problem any longer, if it is done without any attempt at coordination and control, the result will likely be that the user can't understand any of the streams.

2 Design

TTS API Provider is composed of four independent parts. The *provider itself*, running on the system and interfacing applications with speech synthesizers and doing the necessary emulations, the various *drivers* that translate the APIs of the synthesizers into TTS API, *driver template library* to help with developement of drivers and *convenience libraries* which provide easy access to TTS API from various languages.

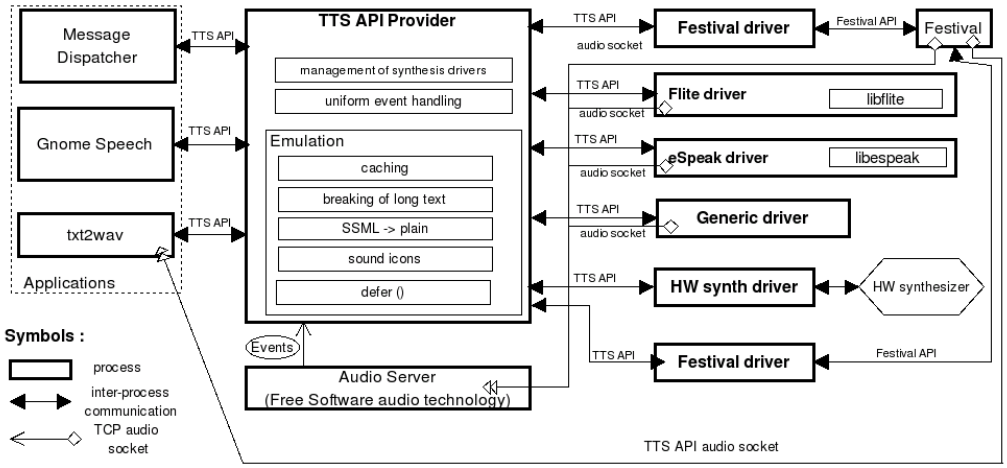
2.1 General Design

The intended middle-term design, as also described in other sections in this chapter, is demonstrated on the following picture. Please keep in mind the picture was not meant to be exhaustive and only contains the main ideas and some examples.



Such an architecture is not ideal, however. The optimal future design is demonstrated on the picture bellow. Most important, TTS API Provider should not have to take care of directly handling audio devices and doing its own audio output. This is a task unrelated to the Text-to-Speech process, necessary for many other applications outside the domain of speech synthesis and accessibility, and as such would best be left to an independent component. Only because currently we do not know of any general Free Software audio

component that could handle our needs, we will temporarily continue to provide our own basic audio output.



Also, ideally, audio should not be transported through the engine driver, but could be directed from the synthesizer itself directly to the destination, lowering the latency and transport overhead, as is illustrated on the picture in case of Festival.

2.2 TTS API Provider (core)

Functionality

- manage all drivers
- emulate necessary functionality not supported by the drivers
- provide interface to applications

Implementation

- License: GNU GPL
- Language: Python
- Dependencies: Python

TTS API Provider will be a separate process for the following reasons:

- There will be only one instance of it running on the system at a given time, which simplifies things like rereading configuration after configuration change, maintaining context on simple non-context synthesizers (!), solving of conflicts in access to hardware synthesizers. It will make it possible not to lunch unnecessarily many drivers for one synthesizer (!).
- Applications won't have compile dependency on TTS API Provider.
- Applications won't be affected by TTS API Provider stability.
- The language of implementation of TTS API Provider is not important for the applications.
- Problems with TTS API Provider will be easier to diagnose for the user and its debugging will be easier for the developers.
- Communication can be led over network.

TTS API Provider communication layer will provide a TCP version of TTS API. This TCP inter-process communication will be wrapped in convenience libraries (4) for easy use in the application. Care will be taken to avoid dependency on this particular communication method so that in the future, other IPC methods like Corba or DBUS can be added if necessary.

TTS API Provider will automatically launch available engine drivers. In order to achieve context independency of clients, to allow more clients to perform synthesis at once and to keep the synthesizer drivers simple (not require them to explicitly handle various clients), it might sometimes be necessary to have several instances of a driver running to serve the various clients.

TTS API Provider will temporarily handle audio output if necessary. Audio output code should not be a part of TTS API Provider, but currently no audio output framework is available on the Free Software platform which would fit our needs. The audio handling part is to be removed as soon as such a framework is available.

TTS API Provider will emulate the necessary functionality that is reported by the synthesizer drivers through TTS API as non-available. The most important examples are: SSML to plain-text conversion, context switching and breaking of long text into smaller chunks.

TTS API Provider will provide the defer() and say_deferred() mechanism for all synthesizers (maintaining the necessary heap of messages for synthesizers who do not support it). This will significantly release implementation burden on the applications.

2.3 Drivers

Functionality

- initialization of and communication with synthesizers
- conversion between TTS API and synthesizer API
- only emulate functionality where necessary for conversion between TTS API and synthesizer API

Implementation

License: various

Language: various, most often C

Dependencies: various, driver template library

Drivers will be separate processes as is completely necessary for legal reasons and for the reasons of stability.

Drivers can be built using the driver template library (3), although this is not necessary.

Drivers will communicate in the same TCP version of TTS API through pipes, reading from their standard input and writing to their standard output.

2.4 Driver Template Library

Functionality:

- take care of IPC communication (TTS API + audio socket)
- provide the necessary infrastructure (threads etc.) so that the driver author can only implement the necessary TTS API functions
- provide tools to make conversion between TTS API and the various synthesizer APIs easier

Implementation

License: GNU LGPL

Language: Python, C

Dependencies: python, glibc

2.5 Convenience Libraries for Applications

Functionality

- expose a simple library version of TTS API and take care of IPC communication

Implementation

License: GNU LGPL

Language: C, Python

Dependencies: libc, Python

3 Server Implementation

This chapter roughly describes the implementation of TTS API provider, including the related subsystems (driver templates, audio subsystem etc.). This chapter should rather be seen more as an introduction guide for new developers than an exhaustive technical code documentation. Please note that this description is not necessarily up to date. Ultimate documentation for the implementation are the abundant comments in the source code itself.

3.1 TTS API Provider core (implementation)

TTS API Provider is a thread-based server. Handling of each client is done in a separate thread which in turn operates a separate set of output drivers. This is highly convenient as by a design decision, there are made no assumptions on interaction between the various clients. In other words, the design must allow for multiple clients handling their synthesis requests and speaking at the same time. This allows higher level layers using TTS API to do a very sophisticated message coordination including serialized AND concurent speech, 3D aural speech from multiple sources etc.

The main server with its daemon functionality (pidfile, signals, terminal detaching, global state of the server and global audio events receiver) is implemented in the source file '`src/server.py`'. The server is currently based on TCP communication and TCP communication should remain a part of the functionality. However, it is by no means restricted to TCP communication. '`src/server.py`' can be extended to include also other means of communication (DBUS, etc.) without loosing the Provider functionality provided by the underlaying modules. For such reason, care is taken to distinguish between the different modes of communication in paramters passed forward even though currently text-based TCP (or pipes) is the only option.

Connection handling

When a new connection is obtained, a new thread is created running the `serve_client` function found in the same source file. `serve_client` starts a new `provider.Provider` object. The global logging, configuration, audio (subsystem) are passed to the Provider object on initialization.

A second object of the family `ttsapi.server.` is then created to handle the connection. It is given the new `provider.Provider` object as its parameter. Its method `connection.process_input()` is then being called in a loop. This connection object will process the input from the communication channel and call the appropriate communication-mechanism independent method of the object `Provider`. `provider.Provider` is then responsible for providing the TTS API functionality, emulations and communication with the device drivers. It roughly implements the Python version of TTS API.

Audio Event Delivery Thread

One thread is launched by the TTS API Provider right on its start. It is the Audio Event Delivery Thread running as function `audio_event_delivery` implemented in '`src/server.py`'. This thread listens for all audio events delivered to TTS API Provider from the audio subsystem and dispatches them to the various `provider.Provider` objects associated with the appropriate connection to which this audio event belongs.

Please note that it is important not to slide into the temptation of delivering audio events from the audio subsystem to the provider objects directly as it is a design decision to eventually split the audio subsystem and TTS API provider into different servers. (The idea is to use some global Free Software audio subsystem when one appears that fulfills the Accessibility Audio Framework Requirements and drop our own codebase). Once such a transition is achieved, events will necessarily be delivered into the server as a whole, not directly to the `provider.Provider` instances, so it will only be necessary to modify the `audio_event_delivery` method if this strict separation is maintained.

For more information about the audio subsystem, please see See Section 3.2 [Audio Subsystem (implementation)], page 7.

Provider Object

The `Provider` object is implemented in the file '`src/provider.py`' and implements the core functionality of TTS API Provider in a client-server communication method independent way. On initialization, it starts its own set of device driver processes. Communication with the device drivers is currently led in the text-based pipe version of TTS API, the same code in '`src/ttsapi/server.py`' is used as for communication between clients and provider. Please refer to TTS API definition document, See Section 5.3 [Python TTS API], page 25, and the source code itself for description of the available methods and their function.

With regards to `emulation` of missing capabilities in the drivers: Ideally, all emulation should be done in the provider object by cooperation with its subsystems whenever the driver reports that a capability is not available. Generally useful emulations like character caching, defer, text-substitution (punctuation, capital letters) and index marking emulation etc. should not be done in the drivers as this unnecessarily blows their codebase, reduces readability and necessarily leads to duplication of code. Emulations that do not emulate TTS API `MUST HAVE` functionality must be configurable.

The provider object recognizes three audio output methods:

- retrieval – audio data will be retrieved to client application
- playback – the synthesizer will play the audio data on speakers
- emulated_playback – audio data will be passed to the TTS API Provider audio subsystem for playback

Methods `retrieval` and `emulated_playback` are only available if the device can return the synthesized data to the caller. For simple hardware synthesizers, only method playback is available. However, when both playback and retrieval are available by the device itself and the client requests playback, Provider will use 'emulated_playback' so that we have full control over the playback and we do not have to deal with the (usually numerous) bugs in synthesizers implementation of audio output (device blocking etc.) according to the TTS API Provider design decisions.

If method `emulated_playback` is being used, the `say_` functions need to notify the audio subsystem about the new message identification and tell it to expect new audio data in its audio data bin. Also request for playback must be sent whenever necessary. Please see the method `provider._prepare_for_message` in '`src/provider.py`'.

3.2 Audio Subsystem (implementation)

4 Device Driver Implementation

Device drivers are implemented as separate processes that are being launched by the TTS API Provider Core. All communication between the core and the drivers happens through the text protocol TTS API via pipes (stdin, stdout and stderr of the said process). As such, drivers can be implemented in any programming language as long as they conform to the prescribed interface.

We provide libraries for Python and C which make implementation of the device drivers quite simple by handling the questions of communication, logging and internal structure (such as threading) automatically. Their use is recommended for ease and consistency, programmers are however free to implement their device drivers in a different way or in a different programming language.

4.1 Driver Interface

The drivers communicate using the standard TTS API implemented as a text protocol, with a few exceptions listed bellow. Input/output happens on standard input and standard output of the device driver process. All logging messages should be written in the selected level of verbosity to the standard error output.

Differencies from the standard TTS API:

- INIT The very first command sent to the driver process must be the `INIT` command. On receiving this command, the device driver should initialize itself (connect to the synthesizer, test if the connection is working etc.) and report success or error together with a reason for the error in a human readable form.

  If the driver reports an error during initialization, the provider core is responsible for sending the QUIT command subsequently and thus terminating the driver process.

  It is an error to call any other TTS API command before `INIT` or to call `INIT` more than once.

  Example with successful initialization:

  ```
  INIT
  200 OK INITIALIZED SUCCESFULLY
  ```

  Example with error during initialization:

  ```
  INIT
  304-"Festival driver not loaded, server not running."
  304 DRIVER NOT LOADED
  QUIT
  ```

- LIST DRIVERS This command provides only one entry in its reply which is corresponding to the name and version of the synthesizer handled by this driver.
- SET DRIVER This command is not used. Attempts to call it should result in an `INVALID COMMAND` error reply.
- HELP This command is not used by the Provider. If is up to the driver author whether he implements it.

4.2 Drivers in Python

A convenience library and a prepared skelton for developing Python device drivers can be found in '`tts-api-provider/src/provider/driver.py`'. This file is basically a full implementation of a driver that does nothing :)

4.2.1 Driver Design (Python)

There are two possible ways how to create a device driver:

- Bottom-up: By overriding some methods of the provided classes that should contain code specific for your drivers while using the provided skeleton.
- Top-Down: By creating your own skeleton of the driver process while using some of the provided classes for some of the tasks that you need to accomplish.

Unless your driver is very specific, we recommend using the bottom-up approach, which should be considerably easier to implement.

4.2.2 Driver skeleton (Python)

To use the provided driver skeleton, create a new script which calls the method `driver.main_loop(DriverCore, DriverController)` where `DriverCore` and `DriverController` are instances of classes defining the driver functionality. Normally they are derived from the base classes `driver.Core` and `driver.Controller` and reimplement only those methods specific to the driver in question. For that reason, if you consider writing a device driver in Python, you are very much encouraged to study the contents of '`src/provider/driver.py`' carefully.

Basically, `DriverCore` is the main ,,provider" object which implements all the TTS API functionality methods, like `Core.set_rate`, `Core.say_text` or `Core.say_key`. Some of the methods like `Core.say_text`, `Core.say_key` or `Core.cancel` are by default preprogrammed so that they accept the request only, pass it for processing to the appropriate `DriverController` method which runs in a separate thread, and return. While the `DriverCore` methods must be non-blocking (i.e. `SAY TEXT` must return immediatelly as per definition of TTS API), there is no such restriction on the `DriverController` methods, which are launched in a separate thread, and can run during the whole time of the synthesis and/or audio playback of the requested message.

It is enough for a programmer to override the `driver.Core.say_text()` method if he doesn't need the asynchronous functionality of the `driver.Controller` object. If the synthesizer is blocking however, the driver programmer can make advantage of the provided mechanism by overriding the `driver.Controller.say_text()` method instead without having to create his own threads and locks for that purpose.

4.2.3 Core (Python)

The `Core` object provides the main functionality of the driver. It implements all driver TTS API methods. Every method has a set of parameters and a return value as defined in TTS API and documented in '`src/provider/driver.py`'. If not overriden, most of these methods raise the `ttsapi.error.ErrorNotSupportedByDriver` exception.

1. It is not necessary to implement all the methods, the driver author must however allways make sure that the state of implementation of the `DriverCore` methods is consistent with the capabilities list returned by the `DriverCore.capabilities()` method.

Notes:

2. It is necessary to run `super(Core, self).init()` and `super(Core, self).quit()` at the end of your own code if you override the `Core.init` or `Core.quit` methods so that the `DriverController` thread is handled correctly.

Please see inline documentation in '`src/provider/driver.py`' for more information about the class and its methods and '`src/provider/festival.py`' for an example.

4.2.4 Controller (Python)

The controller object contains the following TTS API methods:

- say_text
- say_key
- say_char
- say_icon
- cancel
- defer
- discard

Instead of overriding these methods in `DriverCore`, they can be overriden in the `DriverController` object. The methods in this later object are called one after each other in a lateral thread. The execution mechanism is implemented in the `run` method which we recommend to study as well.

The parameters and return values of the `DriverController` TTS API methods are exactly the same as those for the `DriverCore` methods.

Please note that if you redefine the primary `DriverCore` method, e.g. `say_text`, without calling the `super` method of the parent, the appropriate `DriverController` method will never get called. It is thus allways only reasonable to redefine one of those two methods.

Example: A very common case will be the `say_text` method implemented in the `DriverController` object while the `cancel` method is implemented in the `DriverCore` object. This way, the `say_text` method can be blocking (in the lateral thread) during the whole time of synthesis playback, while the `cancel` method in the `DriverCore` object in the main thread can still be called to stop the 'blocking' synthesis code.

Please see inline documentation in '`src/provider/driver.py`' for more information about the class and its methods and '`src/provider/festival.py`' for an example.

4.2.5 RetrievalSocket (Python)

The `RetrievalSocket` class allows the module to open an audio retrieval socket connection (as defined by TTS API) to the given host and port and send blocks of audio data via the `send_data_block` method.

Drivers that support the 'retrieval' method of audio output but the underlaying synthesizer doesn't support it, must retrieve the data from the synthesizer via the offered mechanism and send them to their desired destination via the TTS API retrieval socket. Creating an object of the `RetrievalSocket` class is the prefered way to accomplish the task.

Please see inline documentation in '`src/provider/driver.py`' for more information about the class and its methods and '`src/provider/festival.py`' for an example.

4.2.6 Other tools (Python)

- The `main_loop` function creates a global logger (with output to stderr as defined by the Driver guidelines). You can access it via `driver.log.`. For example `driver.log.debug(‘‘Hello world’’)`
- The `main_loop` registers its own callback via `DriverCore.register_callback()` which reports all events via the standardiz TTS API text protocol mechanism. If your driver implements the playback method (handles playback itself instead of retrieving the audio data), you should implement the `register_callback()` method so that your provider registers the correct callback reporting function at start. Your implementation of `DriverCore` is then responsible to call this callback function on receiving every callback/event.

4.3 Drivers in C

The C library and driver skeleton have not been implemented yet, they are planned for very near future however.

5 TTS API Implementations

5.1 Basic Usage of TTS API

5.1.1 Understanding the API

The set of functions available in TTS API Provider, called TTS API, is defined in a general way at http://www.freebsoft.org/doc/tts-api/tts-api.html. We highly recommend you to study this document carefully before proceeding further. The interface itself has various implementations. Currently, there is the TCP text protocol implementation and a python library implementation. These implementations of the interface differ in the coding syntax, in the way how functions are called and in the parameter types. They however should not differ in the functionality provided, so your best and most accurate guide to the exact meaning of the functions provided is the description of TTS API itself mentioned above, to which the various implementations must conform.

Bellow follows a brief overview of the API and some examples of its proper usage.

5.1.2 Initializing and closing a connection

TTS API does not require any `init` command at the beginning of each session. Each connection becomes fully operational directly after connecting on the given socket or creating the appropriate object according to the communication method in use. When some kind of an `init()` function is necessary for a given API, this is mentioned in the API documentation.

At the end of each session, client program should call the `close()` function to notify TTS API Provider about session termination and to close the socket/connection.

5.1.3 Parameter settings

TTS API allows the controlling application to set various speech and controll parameters. Please see http://www.freebsoft.org/doc/tts-api/tts-api.html#Parameter-Settings for a detailed overview.

5.1.4 Speaking and audio retrieval

The client application can request synthesis of text message into audio stream. Depending on the configuration and synthesizer capabilities, this audio stream can be played by the synthesizer, by the TTS API Provider audio subsystem or retrievaed to the client application for further processing.

Please read http://www.freebsoft.org/doc/tts-api/tts-api.html#Speech-Synthesis-Commands for information about the available speech synthesis commands. If you are interested in audio retrieval, please also read http://www.freebsoft.org/doc/tts-api/tts-api.html#Audio-Retrieval.

5.1.5 Callbacks and events

It is essential for proper synchronization in client application and for further sound processing that the synthesized audio stream is accompanied with marking information about the former text (sentence and word boundaries). In case playback of the audio stream is done on the TTS API Provider side, it is also necessary that some kind of callbacks is

provided so that the client application knows when speech is started or stopped. Both of these mechanisms are supported by the TTS API Provider.

The mechanism of reporting this information differs based on whether playback or audio retrieval is requested. In case of audio retrieval, the information about various events and their timing in the given audio stream is sent in a well defined format along with the audio data. On the other side, if playback is requested, the events are reported in a form of callbacks at the time when they are reached by the audio playback. The exact mechanism may differ according to the API implementation.

For further information about events in audio retrieval mode, please see http://www.freebsoft.org/doc/tts-api/tts-api.html#Audio-Retrieval.

For further information about in-playback callbacks, please see http://www.freebsoft.org/doc/tts-api/ and read the documentation specific to the API implementation you use to learn about the exact mechanism.

5.1.6 Serialized and simultaneous speech

Where there are more than one message to be synthesized and played, the client application may want to achieve either serialized or simultanous speech, or a combination of them. Serialized speech means that the messages are spoken one after another without any overlaps, while in simultanous speech messages are spoken all at the same time. Both approaches might be useful in certain situations.

TTS API is a low-level interface and for this reason, it doesn't attempt to solve the synchronization and playback timing of messages. The client application needs to take care of that. Nor TTS API Provider connection nor the API itself is designed to process more than one synthesis request at a time and such attempts will be rejected.

Thus if the client application wants to synthesize/ speak two or more messages in parallel, it must open the corresponding number of independent connections. Each connection to TTS API Provider gets in turn its own independent connection to the synthesizer (or its own instance of the synthesizer), so that it is able to fulfill synthesis requests really in parallel.

When serialized speech (in other words, one message after another) is desired, the calling program must ensure this using callbacks. A new synthesis request can only be sent after the message_end callback is delivered to the program.

5.2 Text Protocol TTS API

This section documents the text protocol in use for communication over sockets, pipes and other channels where serialized textual protocol is a convenient interface.

In the protocol description bellow, accent is put on form. The exact expected behavior of all the commands and the exact meaning of the arguments is described in TTS API specifications available from http://www.freebsoft.org/doc/tts-api/ and is not repeated in this document. All commands or functions and their arguments have an identical or very similar name to those in the original TTS API specifications.

5.2.1 General Rules (text protocol)

The text protocol version of TTS API is defined as a set of text commands in the usual manner for common Internet protocols. All the characters are encoded using the UTF-8 encoding.

Each command, unless specified otherwise, consists of exactly one line. The line is sent in the following format:

```
command arg ...
```

where *command* is a case insensitive command name and *arg*s are its arguments separated by spaces. The command arguments which come from a defined set of values are case insensitive as well. The number of arguments is dependent on the particular command and there can be commands having no arguments.

All input and output lines must be ended with a pair of carriage return and line feed characters, in that order.

A connection is preferably closed by issuing the `QUIT` command, see Section 5.2.9 [Other Commands (text protocol)], page 24.

The protocol defined here is synchronous — you send commands and only after a complete response arrives back are you allowed to send the next command. The only exceptions to synchronous communication are event and index mark notifications sent by the server in order to inform the client about a task in progress. Such notifications (but only if requested) are sent asynchronously to the connection.

Usually, the connection remains open during the whole run of the particular client application. If you close the connection and open it again, you must set all the previously set parameters again, session parameters are not stored between connections.

Replies have the following format:

```
ccc-line 1
ccc-line 2
...
ccc-line n-1
ddd line n
```

where *n* is a positive integer, and *ccc* and *ddd* are three-digit long numeric codes identifying the result of the command. The last line determines the overall result of the command. The result code is followed by an English message describing the result of the action in a human readable form.

5.2.2 Return Codes

Each line of the output starts with a three-digit numeric code of the form *NXX* where *N* determines the result group and *xx* denotes the finer classification of the result.

The following result groups are defined:

`1xx` Informative response — general information about the protocol, help messages.

`2xx` Operation was completely successful.

`3xx` Server side error, problem on the server side or in the driver.

`300 UNKNOWN ERROR`
: Unknown error.

`301 NOT SUPPORTED BY DRIVER`
: Not supported by the driver.

`302 NOT SUPPORTED BY SERVER`
: Not supported by the server (implementation incomplete).

`303 DRIVER ACCESS DENIED`
: Cannot access driver.

`304 INTERNAL ERROR`
: Internal error in server.

`4xx`
: Client error, invalid arguments or parameters received, invalid commands syntax, unparseable input.

`400 INVALID COMMAND`
: Invalid command, wrong formating of parameters etc.

`401 INVALID ARGUMENT`
: Invalid command argument value given

`402 MISSING ARGUMENT`
: Missing mandatory command argument.

`403 INVALID PARAMETER`
: Trying to set invalid parameter.

`404 ENCODING ERROR`
: Invalid UTF-8 encoding.

`7xx`
: Events and index marks notifications.

`701` Message event.

`702` Sentence or word event.

`703` Index mark event.

Result groups *1xx* and *2xx* correspond to successful actions, groups to *3xx* to *5xx* unsuccessful actions. Only the groups defined here may be returned in a valid TTS API connection.

Currently, for return codes in the range `100`–`299` and `302`–`399`, only the meaning of the first digit of the result code is defined. The last two digits are insignificant and can be of any value. Clients shouldn't rely on the unspecified digits in any way.

However, the return codes in the range `700`–`800`, reserved for events notification, are well defined in the appropriate section of the documentation and client applications can rely on them.

In the future, these return codes should be fixed so that clients can rely on them.

5.2.3 Driver Discovery (text protocol)

`LIST DRIVERS`
: Lists the available drivers.

The reply contains several lines of the following form, each one for a different driver.

```
201-driver-id "synthesizer-name" "synthesizer-version" "driver-
version"
```

Example of usage:

```
LIST DRIVERS
201-festival "Festival Speech Synthesis System" "1.94beta" "1.2"
201-flite "Festival Lite" "1.2" "1.1"
201 OK LIST SENT
```

`DRIVER CAPABILITIES` *driver-id*

Return information about the capabilities of the given driver.

The reply takes the following form. Each of the lines must be present in the following order and carry one of the specified values. {a|b} means either a or b (but not both) is possible, while [a,b,c] means a, b, c or any subset where items are separated by spaces.

```
202-can_list_voices {true|false}
202-can_set_voice_by_properties {true|false}
202-can_get_current_voice {true|false}
202-rate_settings [absolute, relative]
202-can_get_default_rate {true|false}
202-pitch_settings [absolute, relative]
202-can_get_default_pitch {true|false}
202-pitch_range_settings [absolute, relative]
202-can_get_pitch_range_default {true|false}
202-volume_settings [absolute, relative]
202-can_get_volume_default {true|false}
202-punctuation_modes [all, none, some]
202-can_set_punctuation_detail {true|false}
202-capital_letters_modes [spelling, icon, pitch]
202-can_set_number_grouping {true|false}
202-can_say_text_from_position {true|false}
202-can_say_char {true|false}
202-can_say_key {true|false}
202-can_say_icon {true|false}
202-can_set_dictionary {true|false}
202-audio_methods [playback, retrieval]
202-events [by_sentences, by_words, by_index_marks]
202-performance_level {none|good|excelent}
202-can_defer_message {true|false}
202-can_parse_ssml {true|false}
202-supports_multilingual_utterances {true|false}
202 OK DRIVER CAPABILITIES SENT
```

Example of usage (incomplete reply indicated by '[...]')

```
DRIVER CAPABILITIES festival
202-can_list_voices true
202-can_set_voice_by_properties true
202-can_get_current_voice true
```

```
202-rate_settings relative absolute
[...]
202-honors_performance_guidelines excelent
202-can_defer_message false
202-can_parse_ssml true
202-supports_multilingual_utterances false
202 OK DRIVER CAPABILITIES SENT
```

5.2.4 Voice Discovery

`LIST VOICES` *`driver-id`*

List voices available for a given driver.

The reply contains zero or more lines of the following form.

`203-"`*`name`*`" `*`language`*` "`*`dialect`*`" {MALE|FEMALE} `*`age`*

Example usage:

```
LIST VOICES festival
201-"kal" en nil MALE 30
201-"ked" en nil MALE 30
201-"czech_ph" cs nil MALE 30
201-"el_diphone" es nil MALE 48
201-"lp_diphone" it nil MALE 30
201-"pc_diphone" it nil FEMALE 30
201-OK LIST SENT
```

5.2.5 Speech Synthesis Commands (text protocol)

Commands listed in this section are actual request for synthesis (and possibly playback) of a textual or sound message.

`SAY TEXT` *`format`*

Start receiving a text message and synthesize it. After sending a reply to the command, the server waits for the text of the message. The text can spread over any number of lines and is finished by an end of line marker followed by the line containing the single character `.` (dot). Thus the complete character sequence closing the input text is `CR LF . CR LF`. If any line within the sent text starts with a dot, an extra dot is prepended before it.

During reception of the text message, server doesn't send responses for the lines sent. The response line is sent only immediately after the `SPEAK` command and after receiving the closing dot line. Server can start input processing or speech synthesis as soon as a sufficient amount of the text arrives; it generally needn't (but may) wait until the end of data marker is received.

There is no explicit upper limit on the size of the text, but the server administrator may set one in the configuration or the limit can be enforced by available system resources. If the limit is exceeded, the whole text is accepted, but the excess is ignored and an error response code is returned after processing the final dot line.

The content of the message can be either a plain text or a SSML (Speech Synthesis Markup Language) text according to the *format* argument. *format* can be either `SSML` or `PLAIN`.

Position where to start synthesis is specified as a non-negative number *position* and the type of the event *position_type* as specified in TTS API with one of the following values: `MESSAGE_BEGIN`, `MESSAGE_END`, `SENTENCE_BEGIN`, `SENTENCE_END`, `WORD_BEGIN`, `WORD_END`.

The reply for the `SAY` command has the form

```
204 OK RECEIVING DATA
```

and the reply to the end of text marker `CR LF . CR LF` completing the whole composed command is

```
203-message-id
204 OK MESSAGE RECEIVED
```

where *message-id* is a positive number representing the unique message identification.

Example usage:

```
SAY TEXT PLAIN
203 OK RECEIVING DATA
Hello world!
.
204-67
204 OK MESSAGE RECEIVED
```

`SAY TEXT FROM POSITION` *position position_type*

Same as (see [SAY TEXT], page 17) except synthesis is started from a given event of type *position_type* on *position* specified as a positive number.

position_type is one of `SENTENCE_BEGIN`, `SENTENCE_END`, `WORD_BEGIN`, `WORD_END`.

Example usage:

```
SPEAK PLAIN FROM POSITION 2 WORD_BEGIN
203 OK RECEIVING DATA
Hello, world.
204-68
204 OK MESSAGE RECEIVED
```

`SAY TEXT` *format* `FROM CHARACTER` *character_position*

Same as (see [SAY TEXT], page 17) except synthesis is started from a given character position *character_position* specified as a non-negative number.

Example usage:

```
SPEAK PLAIN FROM CHARACTER 7
203 OK RECEIVING DATA
Hello, world.
204-69
204 OK MESSAGE RECEIVED
```

`SAY TEXT` *`format`* `FROM INDEX MARK "`*`index_mark`*`"`

Same as (see [SAY TEXT], page 17) except synthesis is started from a client supplied index mark *index_mark*.

Example usage:

```
SPEAK SSML FROM INDEX_MARK "test"
203 OK RECEIVING DATA
<speak>
Hello, <mark name="test">world.
</speak>
204-70
204 OK MESSAGE RECEIVED
```

`SAY DEFERRED` *`message-id`*

Similar to see [SAY TEXT], page 17 except this commands accepts no text.

`SAY DEFERRED` *`message-id`* `FROM POSITION` *`position position_type`*

Similar to see [SAY TEXT FROM POSITION], page 18 except this commands accepts no text.

`SAY DEFERRED` *`message-id`* `FROM CHARACTER` *`character_position`*

Similar to see [SAY TEXT FROM CHARACTER], page 18 except this commands accepts no text.

`SAY DEFERRED` *`message-id`* `FROM INDEX MARK "`*`index_mark`*`"`

Similar to see [SAY TEXT FROM INDEX MARK], page 18 except this commands accepts no text.

`SAY CHAR` *`char`*

Speak letter *char*. *char* can be any character representable by the UTF-8 encoding. The only exception is the character space ( ); that can't be sent directly. In this case, a string `space` must be sent instead.

Example usage:

```
SAY CHAR e
204-71
204 OK MESSAGE RECEIVED

SAY CHAR \
204-72
204 OK MESSAGE RECEIVED

SAY CHAR space
204-73
204 OK MESSAGE RECEIVED

SAY CHAR &
204-74
204 OK MESSAGE RECEIVED
```

This command is intended to be used for speaking single letters, e.g. when reading a character under cursor or when spelling words.

`SAY KEY` *`key_name`*

Example usage:

```
SAY KEY shift_A
204-75
204 OK MESSAGE RECEIVED
```

Accept a key identified by *key_name* as message. The command is intended to be used for speaking keys pressed by the user.

`SAY ICON` *`icon_name`*

Accept a general sound icon identified by *icon_name*.

Example usage:

```
SAY ICON new-line
204-76
204 OK MESSAGE RECEIVED
```

5.2.6 Speech Control Commands (text protocol)

`CANCEL`

Immediately stop synthesis and audio output of the current message, throw away all the data about this message and prepare the synthesizer to receive a new message.

Example usage:

```
CANCEL
209 OK CANCELED
```

`DEFER`

If synthesis and/or audio output are in progress, immediately stop them. Keep the original text and as much data (possibly also audio) as is needed to resume the message later via `SAY DEFERRED`.

Reply has the following form

```
209-message_id
209 OK DEFERRED
```

where *message_id* is a unique positive number as defined in TTS API.

Example usage:

```
DEFER
209-47
209 OK DEFERRED
```

`DISCARD` *`message_id`*

Discards a previously deffered message.

Example usage:

```
DISCARD 47
210 OK MESSAGE DISCARDED
```

5.2.7 Parameter Settings (text protocol)

All settings except for driver selection only have effect until the driver is changed.

Success return codes for all SET commands are

```
211 OK PARAMETER SET
```

5.2.7.1 Driver Selection and Parameters (text protocol)

`SET DRIVER` *`driver_id`*

Set the synthesis driver.

Example usage:

```
SET DRIVER festival
211 OK PARAMETER SET
```

5.2.7.2 Voice Selection (text protocol)

`SET VOICE BY NAME "`*`voice_name`*`"`

Set voice by name for the synthesis driver in use.

Example usage:

```
SET VOICE BY NAME "kal"
211 OK PARAMETER SET
```

`SET VOICE BY PROPERTIES` *`language "dialect" gender age variant`*

Set voice by the given properties.

Example usage:

```
SET VOICE BY PROPERTIES cs nil FEMALE nil 0
211 OK PARAMETER SET
```

`GET CURRENT VOICE`

Return information about the currently used voice. The output contains exactly two lines of this form:

```
212-"name" language "dialect" {MALE|FEMALE} age
212 OK VOICE DESCRIPTION SENT
```

Example usage:

```
GET CURRENT VOICE
212-"kal" en nil MALE 30
203-OK LIST OF VOICES SENT
```

5.2.7.3 Prosody Parameters (text protocol)

`SET {`*`RELATIVE|ABSOLUTE`*`} RATE` *`rate`*

Set relative or absolute rate. *rate* is a positive or negative number representing percents for relative changes, it is a positive number representing words per minute for absolute changes.

Example usage

```
SET RELATIVE RATE +300
211 OK PARAMETER SET
```

```
SET RELATIVE RATE -20
211 OK PARAMETER SET

SET RELATIVE RATE 150
211 OK PARAMETER SET
```

`GET DEFAULT ABSOLUTE RATE`

Get absolute value of default rate for the voice in use.

Reply is in the form:

```
213-absolute_rate
213-OK ABSOLUTE RATE IN WPM SENT
```

where *absolute_rate* is a positive number representing the rate in words per minute.

`SET {RELATIVE|ABSOLUTE} PITCH pitch`

Set relative or absolute rate. *pitch* is a positive or negative number representing percents for relative changes, it is a positive number representing Hertzs for absolute changes.

Examples are analogous to those for see [SET RATE], page 21

`GET DEFAULT ABSOLUTE PITCH`

Get default value of absolute pitch for the voice in use.

Reply is in the form:

```
214-pitch
214-OK ABSOLUTE PITCH IN HZ SENT
```

where *pitch* is a positive number representing the pitch in Hertzs.

`SET {RELATIVE|ABSOLUTE} PITCH_RANGE pitch_range`

Set relative or absolute pitch range. *pitch_range* is a positive or negative number representing percents for relative changes, it is a positive number representing Hertzs absolute changes.

Examples are analogous to those for see [SET RATE], page 21.

`SET {RELATIVE|ABSOLUTE} VOLUME volume`

Set relative or absolute volume. *volume* is a positive or negative number representing percents for relative changes, it is a positive number between 0 (silence) and 100 (max volume) for absolute changes.

Examples are analogous to those for see [SET RATE], page 21.

`GET DEFAULT ABSOLUTE VOLUME`

Get absolute value of default volume for the voice in use.

Reply is in the form:

```
215-pitch
215-OK ABSOLUTE VOLUME IN DB SENT
```

where *volume* is a positive number.

5.2.7.4 Style Parameters (text protocol)

`SET PUNCTUATION MODE` *`punctuation-mode`*

Set punctuation mode to *punctuation-mode*. Allowed values are `NONE`, `ALL`, `SOME`.

Example usage:

```
SET PUNCTUATION MODE ALL
211 OK PARAMETER SET
```

`SET PUNCTUATION DETAIL` *`detail`*

Set the detail for punctuation reading when punctuation mode is set to `SOME`. Detail is a string enumerating all punctuation characters that should be explicitly pronounced. The string must not contain any whitespace characters.

Example usage:

```
SET PUNCTUATION DETAIL ?!.#
211 OK PARAMETER SET
```

`SET CAPITAL LETTERS MODE` *`cap-let-mode`*

Set capital letters reading mode. Allowed values for the *cap-let-mode* parameter are: `NO`, `SPELLING`, `ICON`, `PITCH`.

Example usage:

```
SET CAPITAL LETTERS MODE ICON
211 OK PARAMETER SET
```

`SET NUMBER GROUPING` *`grouping`*

Set grouping of digits for reading numbers. The parameter *grouping* is a non-negative number.

5.2.7.5 Dictionaries

5.2.7.6 Audio Settings

`SET AUDIO OUTPUT` *`method`*

Sets audio output method. Available values of the *method* argument are `PLAYBACK` and `RETRIEVAL`.

Example usage:

```
SET AUDIO OUTPUT PLAYBACK
211 OK PARAMETER SET
```

`SET AUDIO RETRIEVAL DESTINATION` *`host port`*

Sets destination for audio retrieval. *host* is the IP address of the machine where audio data should be delivered. The IP address is written as groups of three digits separated by dots. *port* is a positive number of the desired port.

Example usage:

```
SET AUDIO RETRIEVAL DESTINATION 127.0.0.1 1315
211 OK PARAMETER SET
```

5.2.8 Event Callbacks (text protocol)

Event are reported on the main connection asynchronously and only if the audio output method is set to `PLAYBACK`. (If output method is set to `RETRIEVAL`, information about events reached is sent together with the audio data on the appropriate side channel).

Asynchronous nature of the event reports means such messages in the protocol are not a result of a command being sent by the client and may come at any time after a request for speaking (`SAY`) is sent. Such notifications can be sent even the `CANCEL` or `DEFER` command is issued.

Information about each event is sent in this form:

- Message Events

  ```
  701-type n pos_text
  701 MESSAGE EVENT
  ```

- Sentence and Word Events

  ```
  702-type n pos_text
  702 SENTENCE OR WORD EVENT
  ```

- Index Mark Events

  ```
  703-event-type "name" pos-text
  703 EVENT SENT
  ```

the exact meaning and format of the parameters is explained in TTS API specifications under section Audio Retrieval.

5.2.9 Other Commands (text protocol)

`QUIT` Close the connection. No reply is sent and the connection is closed on server side.

Example usage:

```
QUIT
```

`HELP` Print a short list of all available commands as a multi-line message.

The following format is used for reply:

```
800-line 1
800-line 2
800 HELP SENT
```

Example usage:

```
HELP
800-SAY
800-[...]
800-CANCEL
800-[...]
800-HELP
800 HELP SENT
```

5.3 Python TTS API

Python API is documented through docstrings and embedded comments. Please see '`src/ttsapi/client.py`' in the source tree. This documentation however includes only facts specific for the python implementation and only a very brief description of the functionality provided by the offered methods.

Please read first see Section 5.1 [Basic Usage of TTS API], page 12 for a general overview of how to use the API.

Please also refer to `http://www.freebsoft.org/doc/tts-api/tts-api.html` for the exact general description of the functionality provided by the API functions and for the description of the event/callback and playback/audio retrieval mechanisms in use.