

Common Text-to-Speech API (draft, 28.4.2006)

Hynek Hanke, Brailcom <hanke@brailcom.org>

Milan Zamazal, Brailcom <zamazal@brailcom.org>

Willie Walker, GNOME, Sun Microsystems <willie.walker@sun.com>

Olaf Jan Schmidt, KDE <ojschmidt@kde.org>

Gary Cramblitt, KDE <garycramblitt@comcast.net>

Copyright © 2006 Brailcom, o.p.s. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code of this document must retain the above copyright notice and this list of conditions.
2. Redistributions in binary form and in printed form must reproduce the above copyright notice, this list of conditions and/or other materials provided with the distribution.
3. The names of the authors may not be used to endorse or promote products derived from this document without specific prior written permission.

Table of Contents

1	Introduction	1
2	Interface Description	2
2.1	General Points	2
2.2	Speech Synthesis Driver Discovery	3
2.3	Voice Discovery	6
2.4	Speech Synthesis Commands	7
2.5	Speech Control Commands	10
2.6	Parameter Settings	11
2.6.1	Driver Selection and Parameters	11
2.6.2	Voice Selection	11
2.6.3	Prosody parameters	12
2.6.4	Style parameters	13
2.6.5	Dictionaries	15
2.6.6	Audio Settings	15
2.7	Audio Retrieval	16
2.8	Event Callbacks	19
3	Notes About the Interface	20
Appendix A	Requirements on the API	21
A.1	Design Criteria	21
A.2	Synthesizer Discovery Requirements	21
A.3	Synthesizer Configuration Requirements	22
A.4	Synthesis Process Requirements	22
A.5	Performance Guidelines	24
Appendix B	Extended SSML Markup	25
Appendix C	Key Names	29
C.1	General Rules	29
C.2	List of symbolic key names	29
Appendix D	Requirements on the synthesizers	31
Appendix E	Recommended Sound Icons	33
Appendix F	Related Specifications	34
	Index of Functions	35

1 Introduction

The purpose of this document is to define a common low-level interface for access to the various speech synthesizers on Free Software and Open Source platforms. It is designed to be used by applications that do not need advanced functionality like message management (such as `txt2wave`) and by applications providing high-level interfaces (such as `SpeechDispatcher`, `GnomeSpeech`, `KTTS` etc.) The purpose of this document is not to define and force an API on the speech synthesizers. The synthesizers might use different interfaces that will be handled by their drivers.

This interface will be implemented by a simple layer integrating available speech synthesis drivers and in some cases emulating some of the functionality missing in the synthesizers themselves.

Advanced capabilities not directly related to speech, like message management, prioritization, synchronization etc. are left out of scope for this low-level interface. They will be dealt with by higher-level interfaces. Such a high-level interface (not necessarily limited to speech) will make good use of the already existing low-level interface.

It is desirable that simple applications can use this API in a simple way. However, the API must also be complex enough so that it doesn't limit more advanced applications in use of the synthesizers.

Requirements on this interface have been gathered between various accessibility projects, most notably KDE, GNOME, Emacspeak, Speakup and Free-b-Soft. They are summarized in Appendix A and Appendix B of this document. Appendix A deals with general requirements and required functionality, while Appendix B describes the extended SSML subset in use and thus also defines required parameter settings. The interface definition contained in chapter 2 was composed based on these requirements.

Temporary Note: A goal is a real implementation of this interface in the near future. The next step will be merging the available engine drivers in the various accessibility projects under this interface and using this interface. For this reason, we need all accessibility projects who want to participate in this common effort to make sure all their requirements on a low-level speech output interface are met and that such an interface is defined so that it is suitable for their needs.

Temporary Note: Any comments about this draft are welcome and useful. But since the goal of these requirements is a real implementation, we need to avoid endless discussions and keep the comments focused and to the point.

2 Interface Description

This section defines the low-level TTS interface for use by all assistive technologies on free software platforms.

2.1 General Points

- The definition of this interface is not meant to imply that the final interface will be provided by C library calls. C syntax is only used for convenience.
 - The general design of the API must be respected in every implementation of it. Regardless of language of implementation, the full set of functions defined here must be available.
 - Function definitions (return value, arguments) must be respected.
 - Data types must be respected in their meaning and possible values.
 - The names of functions, data types and the names of the values for enumeration and set data types should be the same as given here, except for writing them in the form most appropriate for the language in use and prepending/appending them with namespace identifiers, class names etc.
 - This API definition uses numerical values as error return values for functions. A given implementation might choose a better mechanism for reporting errors, appropriate for the language or communication mechanism used (for example exceptions, three digit error codes for a TCP protocol etc).
 - Where this API speaks about callbacks, other means of asynchronous notification can be used. (Such as asynchronous messages for a TCP text protocol.)
- The interface is designed such that both a simple library interface and a serialized language-independent character-based protocol for use over sockets and pipes can be provided.
- This interface is meant to be provided by a simple library or process running the various synthesis drivers and emulating **MUST HAVE** functionality where possible and needed. It can also try to emulate some of the **SHOULD HAVE** and **NICE TO HAVE** capabilities at implementor's discretion.
- The interface between this library or process and the synthesis drivers themselves, hardware or software, will be a subset of this interface.
- The interface definition uses the type `bool_t` not present in C for variables whose value can be either `TRUE` or `FALSE`.
- For strings that can possibly contain characters not present in the ASCII table (for example various language specific characters), the data type used in this interface definition is `wchar_t` and the corresponding expected format for Unicode is UTF-32. Strings marked with type `char*`, it must not contain any wide characters. Only when the particular implementation of the interface (for example a text protocol) does not allow for different types, UTF-8 encoding should be used for values marked as `wchar_t`.
- When the word 'strings' is used in this interface description, it means a NULL-terminated array of characters of type `char` for single-byte encoding or a NULL-terminated array of type `wchar_t` for UTF-32 encoding (in other words, four

consecutive zero bytes at the end). More appropriate data types can be used in other languages.

2.2 Speech Synthesis Driver Discovery

This section deals with the discovery of the synthesis drivers available behind this interface. It also covers discovery of the capabilities and voices provided by the drivers.

`driver_capabilities_t` [Variable Type]

`driver_capabilities_t` is a structure data type intended for carrying information about driver capabilities.

```
typedef struct {
    /* Voice discovery */
    bool_t can_list_voices;
    bool_t can_set_voice_by_properties;
    bool_t can_get_current_voice;

    /* Prosody parameters */
    bool_t can_set_rate_relative;
    bool_t can_set_rate_absolute;
    bool_t can_get_rate_default;

    bool_t can_set_pitch_relative;
    bool_t can_set_pitch_absolute;
    bool_t can_get_pitch_default;

    bool_t can_set_pitch_range_relative;
    bool_t can_set_pitch_range_absolute;
    bool_t can_get_pitch_range_default;

    bool_t can_set_volume_relative;
    bool_t can_set_volume_absolute;
    bool_t can_get_volume_default;

    /* Style parameters */
    bool_t can_set_punctuation_mode_all;
    bool_t can_set_punctuation_mode_none;
    bool_t can_set_punctuation_mode_some;
    bool_t can_set_punctuation_detail;

    bool_t can_set_capital_letters_mode_spelling;
    bool_t can_set_capital_letters_mode_icon;
    bool_t can_set_capital_letters_mode_pitch;

    bool_t can_set_number_grouping;

    /* Synthesis */
}
```

```

bool_t can_say_text_from_position;
bool_t can_say_char;
bool_t can_say_key;
bool_t can_say_icon;

/* Dictionaries */
bool_t can_set_dictionary;

/* Audio playback/retrieval */
bool_t can_retrieve_audio;
bool_t can_play_audio;

/* Events and index marking */
bool_t can_report_events_by_sentences;
bool_t can_report_events_by_words;
bool_t can_report_custom_index_marks;

/* Performance guidelines */
int honors_performance_guidelines;

/* Deferring messages */
bool_t can_defer_message;

/* SSML Support */
bool_t can_parse_ssml;

/* Multilingual utterances */
bool_t supports_multilingual_utterances;
} driver_capabilities_t;

```

*can_set_rate_**, *can_set_pitch_**, *can_set_pitch_range_** and *can_set_volume_** variables indicate whether the corresponding prosody parameter setting commands are supported. See (Section 2.6.3 [Prosody Parameters], page 12).

*can_set_punctuation_mode_** variables indicate which parameters are supported for `set_punctuation_mode()`. See ([`set_punctuation_mode()`], page 13).

can_set_punctuation_detail indicates whether the function `set_punctuation_detail()` is supported. See ([`set_punctuation_detail()`], page 14).

*can_set_capital_letters_mode_** variables indicate which parameters are supported for `set_capital_letters_mode()`. See ([`set_capital_letters_mode()`], page 14).

can_set_number_grouping indicates whether the function `set_number_grouping` is supported. See ([`set_number_grouping()`], page 15).

can_say_text_from_position indicates whether the capability to start synthesis at a given position in the text is supported, as described in ([`say_text()`], page 8).

Other *can_say_** variables indicate whether the corresponding `say_` synthesis command is supported. See (Section 2.4 [Speech Synthesis Commands], page 7).

can_set_dictionary indicates whether the function `set_dictionary()` is supported.

can_play_audio and *can_retrieve_audio* variables indicate whether the corresponding audio output methods are allowed for `set_audio_output`. See (Section 2.7 [Audio Retrieval], page 16).

*can_report_** variables indicate which kind of audio events and index marks are supported. See (Section 2.8 [Event Callbacks], page 19).

honors_performance_guidelines variable is 0 if performance guidelines are not honored, 1 if performance guidelines are honored on the (SHOULD HAVE): level and 2 if performance guidelines are honored on the (NICE TO HAVE): level.

can_defer indicates whether the defer capability is supported. If this variable is true, `defer()` and `say_deferred` must be supported. It is expected the synthesizer will be able to defer multiple messages at the same time. See (`[defer()]`, page 10), (`[say_deferred()]`, page 9).

can_parse_ssml indicates whether the synthesizer is able to parse SSML. It doesn't indicate which SSML elements and attributes are supported.

supports_multilingual_utterances indicates whether the synthesizer supports multilingual utterances (utterances containing multiple languages).

`driver_description_t` [Variable Type]

`driver_description_t` is a structure containing information about a single driver.

```
typedef struct {
    char*      driver_id;
    char*      driver_version;
    char*      synthesizer_name;
    char*      synthesizer_version;
} driver_description_t;
```

synthesizer_id is the identification string of the driver.

synthesizer_version carries information about the synthesizer version in use in a human readable form. There is no strict rule for formatting the version information inside the string as the versioning schemes of the various synthesizers differ significantly. If it is not possible to determine the synthesizer version, this string should be NULL.

synthesizer_name is a full name of the synthesizer engine.

driver_version carries information about the driver version in use for the given synthesizer. It has the form "major.minor" where `major` is the major version number for the driver and `minor` is the minor version number for the driver.

driver_capabilities contains information about the support of the driver for functions and features defined in this interface. See (`[driver_capabilities_t]`, page 3) for a list of the available information.

Example:

```
driver_id = "festival"
synthesizer_name = "Festival Speech Synthesis System"
synthesizer_version = "1.94beta"
driver_version = "1.2"
```

`driver_description_t** list_drivers (void)` [Function]

`list_drivers()` returns a newly allocated null-terminated array of available synthesizer drivers. Each of the items in the array is of the type `driver_description_t*`, ([`driver_description_t`], page 5), and must carry a properly filled in variable `driver_id`. In case of an error, the value `NULL` is returned..

`driver_capabilities_t* driver_capabilities (char* driver_id)` [Function]

`driver_capabilities` returns information about the capabilities of the driver in a `driver_capabilities_t` structure.

Under this API, each driver is not guaranteed to support all of the functionality as defined in this document. It must however provide the full set of functions. Whether the functions will have the described effect can be discovered by examining the entries of the `driver_capabilities_t` structure and comparing them with the documentation for the given functions.

`driver_id` is the unique identification string for the synthesizer driver whose capabilities should be reported. See ([`list_drivers()`], page 6).

This function returns a properly filled `driver_capabilities_t` structure on success. In case of an error, the value `NULL` is returned..

2.3 Voice Discovery

`voice_description_t` [Variable Type]

`voice_description_t` is a structure containing the description of a voice.

```
typedef struct {
    wchar_t *name;
    char *language;
    wchar_t *dialect;
    voice_gender gender;
    unsigned int age;
} voice_description_t;
```

`name` is the name of the voice as recognized by the synthesizer.

`language` is an ISO 639 language code represented as a character string. Examples are `en`, `fr`, `cs`.

`dialect` is a string describing the language dialect or `NULL` if unknown or not applicable. Examples are `american` or `british` with English language or `moravian` with Czech language.

Open Issue: Is there a standard way of describing dialects?

`gender` indicates the gender of the voice. The values `MALE`, `FEMALE` and `UNKNOWN` are permitted.

`age` gives the approximate age of the voice in years. A value of 0 means the age is unknown.

`voice_description_t** list_voices (char* driver_id)` [Function]

For a given driver specified as `driver_id`, `driver_list_voices()` returns a newly allocated null-terminated array of describing the available voices in `voice_description_t*` items, one for each voice.

driver_id is the identification string of the driver as returned by `list_drivers()` (`list_drivers()`), page 6).

In case of an error, the value `NULL` is returned..

2.4 Speech Synthesis Commands

Functions defined in this section generally accept a message to synthesize, with driver, voice and other parameters according to the current settings at the time when the function is called. Several types of messages are handled by this API. It can be either a text message, containing plain text or SSML, or it can be a 'key' or 'character' event or any general event.

The functions defined in this section can only block the calling process for as long as is necessary to fully receive and/or transfer the message, which should generally be a very short time. These functions will not block the calling process for the time of synthesis of the message and audio output.

The result of these commands will either be that the resulting audio stream is played on the audio device or that the audio stream is returned via the registered communication channel. Please see [Section 2.6.6 \[Audio Settings\]](#), page 15.

`message_format_t` [Variable Type]

`message_format_t` is an enumeration type to indicate the type of the content of a message.

```
typedef enum {
    MESSAGE_TYPE_SSML,
    MESSAGE_TYPE_PLAIN
} message_format_t;
```

`MESSAGE_TYPE_SSML` means the content of the message is text formatted according to the Speech Synthesis Markup Language. See ([\[SSML\]](#), page 34).

`MESSAGE_TYPE_PLAIN` means the content of the message is plain text.

`message_id_t` [Variable Type]

```
typedef signed int message_id_t;
```

A positive value represents the identification number of the message. The value of 0 means 'no message' and -1 means an error occurred.

`event_type_t` [Variable Type]

`event_type_t` is used to describe the type of an event both in the original text and in the synthesized audio data.

```
typedef enum {
    EVENT_MESSAGE_BEGIN,
    EVENT_MESSAGE_END,
    EVENT_SENTENCE_BEGIN,
    EVENT_SENTENCE_END,
    EVENT_WORD_BEGIN,
    EVENT_WORD_END,
    EVENT_NONE
} event_type_t;
```

EVENT_MESSAGE_BEGIN and EVENT_MESSAGE_END are events corresponding to the begin and end of the message.

EVENT_SENTENCE_BEGIN and EVENT_SENTENCE_END are events corresponding to the begin and end of a sentence.

EVENT_WORD_BEGIN and EVENT_WORD_END are events corresponding to the begin and end of a word.

```
message_id_t say_text (message_format_t format, wchar_t* text)           [Function]
message_id_t say_text_from_event (message_format_t format,                [Function]
    wchar_t* text, unsigned int position, event_type_t position_type)
message_id_t say_text_from_index_mark (message_format_t format,           [Function]
    wchar_t* text, char* index_mark)
message_id_t say_text_from_character (message_format_t format,            [Function]
    wchar_t* text, size_t character_position)
```

`say_text` accepts a text message to synthesize and starts synthesis at the given position.

`position` and `position_type` describe the position in the message where synthesis should be started. `position_type` can be either a word or sentence event. `position` is a positive counter of events of type `position_type` from the beginning of the message. So for example the position 2 of event `EVENT_WORD_START` describes the start of the second word. In a similar way, `index_mark` specifies the name of the index mark where synthesis should start and `character` gives a position in the text as a positive number in characters.

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 is returned.

When a markup language, such as SSML, is being used as the format of the text, this markup may or may not be checked for validity, according to users settings. If a validity check is performed and the text is found to be invalid, an error code is returned and the text is not processed further.

Errors found during processing the document, as for example a markup request to set a language which is not available for the synthesizer, are not reported.

If the position requested through `say_text_from_char` falls in the middle of a markup tag, the synthesis should begin with the text following the tag. If the position is in the middle of a word, the synthesizer can either synthesize from the exact position or it can start from the beginning of the word. Neither of these is considered an error.

`format` is a format of the message according to (`[message-format-t]`, page 7).

`text` is the text to be synthesized in the form according to the value of the `format` argument.

`position` is a positive number counting the events of the given type. If `position_type` is set to `EVENT_MESSAGE_BEGIN`, the value of this argument is irrelevant and is conventionally set to 0.

`position_type` is one of `EVENT_MESSAGE_BEGIN`, `EVENT_SENTENCE_START`, `EVENT_SENTENCE_END`, `EVENT_WORD_BEGIN` and `EVENT_WORD_END`.

index_mark is the name of the index mark where synthesis should begin.
character_position is a positive number of the character where synthesis should begin.
 On success, a positive value – a unique message identifier – is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

For example calling `say_text()` with the following arguments

```
say_text(MESSAGE_TYPE_PLAIN, "This is an example.", 3, EVENT_WORD_BEGIN)
```

should result in audio which starts with the word 'an' and continues to the end of the sentence.

Note: For longer and more complicated texts, it will not be possible to say in advance where the audio will start, given just the original text of the message and the position description. The placing of events across the original text may be ambiguous and depends on the synthesizer. However, this capability is designed for purposes like rewinding (rewind 5 sentences forward) or context pause (resume speaking from a place which we already got event information about when we executed pause). The application must not try to guess where exactly the events are and rely on that guess if it did not receive the information from the synthesizer earlier.

```
message_id_t say_deferred (message_id_t message_id) [Function]
message_id_t say_deferred (message_id_t message_id, signed int
    position_from, PositionType position_type) [Function]
message_id_t say_deferred_from_index_mark (message_id_t
    message_id, char* index_mark) [Function]
message_id_t say_deferred_from_character (message_id_t
    message_id, size_t character) [Function]
```

`say_deferred` works just like `say_text`, except it works on messages which were previously deferred and if position is set to 0, this has an additional meaning of "start where speech was interrupted last time". Please see ([\[defer\(\)\]](#), page 10).

message_id is the id of the message to synthesize, as obtained by `defer()`.

On success, a positive value – a unique message identifier – is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

```
message_id_t say_key (wchar_t* key_name) [Function]
say_key accepts a key name to synthesize. The command is intended to be used for speaking keys pressed by the user.
```

key_name is a valid key name as defined in [\[appendix-C\]](#), page 29.

On success, a positive value – a unique message identifier – is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

```
message_id_t say_char (wchar_t character_name) [Function]
say_char accepts a letter (or syllable if the language doesn't have individual letters) to synthesize. The command is intended to be used for speaking single character messages, produced when the user is moving the cursor over a word.
```

character_name is the character to synthesize.

On success, a positive value – a unique message identifier – is returned. In case of an error, the value `-1` is returned. When this function is not supported by the driver, `-2` is returned.

`message_id_t say_icon (char* icon_name)` [Function]

`say_icon` accepts a general sound icon to synthesize. The command is intended to be used for general events like ‘new-line’, ‘message-arrived’, ‘question’ or ‘new-email’. The exact sound produced or text synthesized depends on user’s configuration.

The name for the icon can be one of the names given in ([recommended-sound-icons], page 33) or any other name. If the icon name is not recognized by the synthesizer, the synthesizer tries to synthesize the name of the event itself.

If the icon name is not recognized by the synthesizer, the synthesizer tries to synthesize the name of the icon itself.

icon_name is the name of the icon to synthesize. It must not contain any whitespace characters.

On success, a positive value – a unique message identifier – is returned. In case of an error, the value `-1` is returned. When this function is not supported by the driver, `-2` is returned.

2.5 Speech Control Commands

`void cancel (void)` [Function]

`cancel` immediately stops synthesis and audio output of the current message. When this function returns, the audio output is fully stopped and the synthesizer is ready to synthesize a new message.

If this function is called during the transfer of audio data to the application, the data block currently being transferred is completed and no further data block is sent.

Calling this command when no message is being processed is not considered an error. In case of an error, the value `-1` is returned. When this function is not supported by the driver, `-2` is returned.

`void defer (void)` [Function]

`defer` is similar to `cancel` except after stopping the synthesis process and audio playback the message is not thrown away in the synthesizer, but data that might be useful for future working with the message (such as rewinding, repeating or resuming the synthesis process) are preserved. This might or might not include the original text of the message. In any case, enough information must be preserved so that the synthesizer is able to fully reproduce the audio data for the message.

If this function is called during the transfer of audio data to the application, the data block currently being transferred is completed and no further data block is sent.

This function can also be called after all the audio has been already transferred to the application, but before another synthesis request is issued, with no `cancel()` request in between, the data for the previous message are stored.

There is no explicit upper limit on the number of messages that can be simultaneously postponed by `defer()`. There might however be a limit imposed by the administrator

or forced by available system resources. In case such a limit is passed, `defer()` will return with an error.

After the message is no longer needed, the application must make sure to discard it through `discard()`, otherwise system resources will be wasted.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

int `discard` (*message_id_t message*) [Function]

Discards a previously deferred message. The driver/engine will drop all information about this message and the message will be removed from the list of paused messages.

See ([`defer()`], page 10).

message is the message ID of the message to discard. Passing an ID of a message that is not paused is considered an error.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

2.6 Parameter Settings

2.6.1 Driver Selection and Parameters

int `set_driver` (*char* driver_id*) [Function]

Set the synthesis driver. See [`list_drivers()`], page 6.

driver_id is the unique ID of the driver as returned by `list_drivers()`.

2.6.2 Voice Selection

Setting parameters in this section only has effect until the synthesizer driver in use is changed by the application.

int `set_voice_by_name` (*wchar_t* voice_name*) [Function]

`set_voice_by_name` selects the voice with the given name.

voice_name is the name of the desired voice. It must be one of the names returned by `list_voices()`. See [`list_voices`], page 6.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

int `set_voice_by_properties` (*voice_description_t *voice_description, unsigned int variant*) [Function]

`set_voice_by_properties` selects a voice under the current driver most closely matching the given description. The exact voice selected might be subject to user preference settings for voice selection inside the synthesizer.

There is no guarantee that any of the given parameters will be respected, although language generally is supposed to be respected, unless impossible or unless the user wishes otherwise.

In case no voice matches the given language, the synthesizer should pick the general default voice (if applicable) or choose an arbitrary voice. This alone is not considered an error and must not be a reason for the synthesizer to refuse further synthesis

requests unless for some other related reason (as for example the voice being unable to handle the given Unicode character range).

The application can check which voice was selected and how closely (if at all) it matches the given description.

voice_description is a description of the desired voice. Any of its entries except language can be filled in or left blank (NULL for strings, 0 for integer values, UNKNOWN for VoiceGender). Please see [voice_description_t], page 6 for more information about the format and allowed values.

variant is a positive (1,2,3...) number specifying which of the voices matching the description and assigned equal priority inside the synthesizer should be selected. Please see ([SSML], page 34) for more details.

Note: This function is different from performing `voice_list` and following that with `set_voice_by_name` as user settings about voice selection inside the synthesizer are respected.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

```
voice_description_t* get_current_voice (void) [Function]
get_current_voice returns a voice_description_t structure filled in with all
known information about the voice currently in use.
In case of an error, the value NULL is returned..
```

2.6.3 Prosody parameters

Setting parameters in this section only has effect until the synthesizer is changed.

```
int set_rate_relative (signed int rate_relative) [Function]
int set_rate_absolute (unsigned int rate_absolute) [Function]
unsigned int get_rate_absolute_default (void) [Function]
Set/get the rate of speech.
```

rate_relative represents the relative change with respect to the default value for the given voice. For example 0 means the default value for the given voice while -50 means a fifty percent lower rate with respect to the default.

rate_absolute is the desired rate in words per minute.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

```
int set_pitch_relative (signed int pitch_relative) [Function]
int set_pitch_absolute (unsigned int pitch_absolute) [Function]
unsigned int get_pitch_absolute_default (void) [Function]
Set/get the voice base pitch.
```

pitch_relative represents the relative change with respect to the default value for the given voice. For example 0 means the default value for the given voice while -50 means a fifty percent lower pitch with respect to the default.

pitch_absolute is the desired pitch in Hz.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

int set_pitch_range_relative (*signed int range*) [Function]
 Set voice pitch range in relative units. Pitch range is how much pitch changes in intonation with respect to the base pitch.
pitch represents the relative change with respect to the default value for the given voice. For example 0 means the default value for the given voice while -50 means a fifty percent lower pitch range with respect to the default.

int set_pitch_range_absolute (*unsigned int range*) [Function]
 Open Issue: How should this work? It is not clear from the SSML specs.
 On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

int set_volume_relative (*signed int volume_relative*) [Function]
int set_volume_absolute (*unsigned int volume_absolute*) [Function]
unsigned int get_volume_absolute_default () [Function]
 Set/get the volume of speech.
volume_relative represents the relative volume change with respect to the default value for the given voice. For example 0 means the default value for the given voice while -50 means a fifty percent lower volume with respect to the default.
volume_absolute is a number from the range 0 to 100 where the value of 0 means silence and 100 means maximum volume.
 On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

2.6.4 Style parameters

punctuation_mode_t [Variable Type]
punctuation_mode_t is an enumeration type containing information about punctuation signalling mode.

```
typedef enum {
    PUNCTUATION_NONE,
    PUNCTUATION_ALL,
    PUNCTUATION_SOME
} punctuation_mode_t;
```

PUNCTUATION_NONE means no punctuation is signalled.
PUNCTUATION_ALL means all punctuation characters are signalled.
PUNCTUATION_SOME means only selected punctuation characters are signalled. (See [\[set_punctuation_detail\(\)\]](#), page 14).

int set_punctuation_mode (*punctuation_mode_t mode*) [Function]
 Set punctuation reading mode. In other words, this influences which punctuation characters will be signalled while reading the text. Signalling means either synthesizing their name (e.g. ‘question mark’) or playing the appropriate sound icon, according to user settings inside the synthesizer.
 For example the ‘.’ (dot) and ‘?’ (question mark) are not normally pronounced and their presence only influences the intonation of the sentence. However, in some cases

such as copyediting text or editing program source code, it is desirable to have them spoken or otherwise indicated.

mode is one of PUNCTUATION_NONE, PUNCTUATION_ALL and PUNCTUATION_SOME (See [set_punctuation_detail()], page 14) as defined in [punctuation_mode_t], page 13.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

`int set_punctuation_detail (wchar_t *detail)` [Function]

`set_punctuation_detail` influences which punctuation characters should be signalled when the punctuation mode is set to PUNCTUATION_SOME (See [set_punctuation_mode()], page 13)

detail is a string enumerating the punctuation characters that should be signalled without any spaces.

Example:

```
set_punctuation_detail("?!.#");
```

`capital_letters_mode_t` [Variable Type]

`capital_letters_mode_t` is an enumeration type containing information about selected mode for signalling capital letters.

```
typedef enum {
    CAPITAL_LETTERS_NO,
    CAPITAL_LETTERS_SPELLING,
    CAPITAL_LETTERS_ICON,
    CAPITAL_LETTERS_PITCH,
} capital_letters_mode_t;
```

CAPITAL_LETTERS_NO means no signalling of capital letters.

CAPITAL_LETTERS_SPELLING means that each capital letter is prepended with the word “capital” or similar appropriate for the given language. Alternatively, the whole word containing the capital letter may be spelled. These two approaches may be combined.

For example the text “My name is John” would be read as “Capital my name is capital John.” or “Capital m way name is capital j ou age en.”.

CAPITAL_LETTERS_ICON means that each capital letter is prepended with a sound icon.

The above example text “My name is John” would be read as “*ding* My name is *ding* John” where *ding* is the appropriate sound for capital letter signalling as provided by the synthesizer or configured by the user.

CAPITAL_LETTERS_PITCH is a method where capital letters are indicated by raising pitch of the voice when reading them.

Open Issue: How exactly does CAPITAL_LETTERS_PITCH work?

`int set_capital_letters_mode (capital_letters_mode_t mode)` [Function]

`set_capital_letters_mode` sets the capital letters speaking mode as requested.

When the engine is not able to set the requested mode, but it is able to set some other mode, this should be done.

mode is one of `CAPITAL_LETTERS_NO`, `CAPITAL_LETTERS_SPELLING`, `CAPITAL_LETTERS_ICON` and `CAPITAL_LETTERS_PITCH` as defined in [\[capital_letters_mode_t\]](#), page 14.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

`int set_number_grouping (unsigned int grouping)` [Function]

Sets how many digits should be grouped together when reading a number. See [\[tts:digits\]](#), page 26 for a detailed description of the functionality.

grouping a positive number indicating how many digits should be grouped together or 0 for reading numbers as a whole.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

2.6.5 Dictionaries

`int set_dictionary (Dictionary dictionary)` [Function]

Open Issue: How should this work? What is the Dictionary type?

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

2.6.6 Audio Settings

Generally, there are two ways of dealing with audio. Either the application can ask this API to send the synthesized audio samples as data or it can ask for it to be played (e.g. on computer audio device or the internal speakers of hardware devices). Of course both options are not available in every synthesizer.

In the case where the application asks for the audio to be played on the audio device, the means of handling audio events and index marking will be callbacks (handled either as function callbacks or asynchronous socket notifications). This way, event signalling and/or index marking callbacks can be provided by every synthesizer which supports synchronization and/or index marking, regardless of whether it plays audio itself or it gives data to its driver.

If the application asks for audio data to be returned to the application, then event marks and custom index marks are embedded as additional information in the retrieved audio data blocks. This is more accurate and is very useful when the application doesn't want to play the audio immediately, but it wants to store it either as a file or in memory. However, this is only possible with synthesizers that can give audio data to its driver.

Of course it is possible to discover the capabilities of each driver in advance. See [\(Section 2.2 \[Speech Synthesis Driver Discovery\], page 3\)](#).

`output_method_t` [Data Type]

`output_method_t` is an enumeration type for selecting the audio output method for the synthesizer.

```
typedef enum {
    AUDIO_OUTPUT_PLAYBACK,
    AUDIO_OUTPUT_RETRIEVAL,
} audio_output_method_t;
```

`OUTPUT_AUDIO_PLAYBACK` means the audio should be played on the synthesizer or automatically sent to playback.

`OUTPUT_AUDIO_RETRIEVAL` means the audio should be returned to the application from the synthesizer.

`int set_audio_output (output_method_t method)` [Function]

This option deals with the output of the synthesizer. The two possibilities are to have the audio played (which is the only possibility for some synthesizers) or have audio retrieved over a socket as a series of data blocks (either synchronously or asynchronously).

method is either `AUDIO_OUTPUT_PLAYBACK` or `AUDIO_OUTPUT_RETRIEVAL`. See ([[output_method_t](#)], page 15).

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

`int set_audio_retrieval_destination (char *host, unsigned int port)` [Function]

Sets the TCP socket where audio data should be sent. See ([Section 2.7 \[Audio Retrieval\]](#), page 16) for more details.

host is the IP address of the machine where audio data should be delivered.

port is the port on the machine where audio data should be delivered.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

2.7 Audio Retrieval

This section deals with the situation when the application wants to retrieve audio data from the synthesizer as described in ([Section 2.6.6 \[Audio Settings\]](#), page 15).

The audio data are delivered to a TCP socket on the address specified by the application using `set_audio_retrieval_destination()`.

For each message sent to the synthesizer, one or more data blocks are delivered asynchronously over the socket. Each data block contains the identification of the original message and the serial number of the block, information about the audio format in use, events and custom index marks in the given block and the audio data itself.

Each data block is composed of four sections: `BLOCK` acting as a header specifying which message this data belongs to, `PARAMETERS` carrying information about the parameters of the audio data, `EVENTS` as a list of events and custom index marks reached in this audio data, and `DATA` containing the data itself.

The following syntax is used for each block. Where arguments are provided, they are separated by one or more spaces (except for the `PARAMETERS` section where spaces are not allowed).

```

BLOCK msg_id block_number
PARAMETERS
  data_format=data_format
  data_length=data_length
  audio_length=audio_length
  sample_rate=sample_rate
  channels=channels
  encoding=encoding_string
END OF PARAMETERS
EVENTS
  type      text_position      time_position
END OF EVENTS
DATA
  audio_data
END OF DATA

```

BLOCK

msg_id is the unique identification number of the message this audio data belongs to.

block_number is positive string-represented number indicating the position of this audio chunk in the resulting audio for the message. *block_number* one means the first part of the data.

PARAMETERS

The parameters section contains the following parameters (not all of them are always used).

data_format is a string identification for the format of the audio data. Recognized names are: “raw”, “wav” and “ogg”. This parameter is required.

Open Issue: Is there any specification that we could refer here so that we do not need to enumerate the possible values? The goal of this specification is not to dictate which data format should be used.

data_length is an unsigned string-represented number indicating the length of the data contained in the DATA section in bytes. This parameter is required.

audio_length is an unsigned string-represented number indicating the length of the audio data contained in the DATA section in milliseconds. This parameter is required.

sample_rate and *channels* are only used for the “raw” data format and they are string-represented numbers describing the three common audio parameters. *sample_rate* is the sampling frequency in Hz and *channels* is the number of channels in the audio data.

encoding is a string describing the encoding details of the audio data. This parameter is only used for the “raw” audio output. It has the following usual form:

```
signed/unsigned_bits-per-word_endian
```

where *signed/unsigned* is either S or U for signed or unsigned data type, *bits-per-word* is a two digit number representing word data width and *endian* is either LE for little endian or BE for big endian.

EVENTS

The events section contains zero or more lines, each of them representing an event or a custom index mark which is reached in the sent audio data chunk. Each gives its order and position in both the original message text and the synthesized audio.

The synthesizer or synthesizer driver will only report as much information as is possible. The `message_start` and `message_end` events must always be signalled, though.

- Each line for the ‘message’ event has the following form

```
message_start
```

or

```
message_end
```

- Each line for a ‘sentence / word event’ has the following form

```
type    n    pos_text    pos_audio
```

type is the type of the event, one of: `message_start`, `message_end`, `word_start`, `word_end`, `sentence_start`, `sentence_end`.

n is the number of the event starting from one.

pos_text is the position of the event in the original text represented as number of characters.

Note: It is important *pos_text* is given in characters rather than bytes, as a value in bytes is encoding dependent. For example it is different for a text encoded in UTF-8 versus the same text encoded in UTF-32, since in the first case, variable width encoding is being used. This way, it is also easy to deal with by synthesizers that do not support multibyte encodings.

pos_audio is the position of the event in the returned audio in milliseconds. This position is given with respect to the beginning of the message, not the current audio block.

- Each line for the ‘custom index mark’ event has the following form

```
index_mark "name" pos_text pos_audio
```

name is the name of the index mark as included in the SSML mark element by the application. It must be enclosed in quotes because it can contain whitespace characters.

pos_text and *pos_audio* have exactly the same meaning as defined above for the sentence / word event.

DATA

The section `DATA` contains audio data in exactly the length as is specified by the `data_length` parameter in the `PARAMETERS` section for the given block.

Example

Below is an example of audio data for a message being sent in a single block:

```
BLOCK 142 1
PARAMETERS
data_format=raw
```

```

data_length=109368 (bytes)
audio_length=1240 (ms)
sample_rate=44100
channels=1
architecture=S16_LE
END OF PARAMETERS
EVENTS
message_start
word_start      1          0          12
sentence_start  1          0          12
index_mark      "my-1"      14         123
word_start      1          19         442
index_mark      "my-2"      31         821
word_start      2          31         821
message_end
END EVENTS
DATA
here are the audio data
END OF DATA

```

2.8 Event Callbacks

If the output method is set for audio playback, meaning the audio is being played on the audio device behind this API, events and custom index marks are reported through callbacks.

`callback_function_t` [VariableType]

Type for a function to be used as a callback for reporting events and custom index markers.

```

typedef int callback_function_t(event_type_t event,
    signed int n, size_t text_pos, char *name);

```

event is the type of the event reported. ([[event_type_t](#)], [page 7](#))

n and *text_pos* are defined in ([[EVENTS](#)], [page 17](#)). Where not applicable (*n* for index marks and all three for message events), these variables are set to -1.

name is only used when the event is of type custom index mark and contains the name of the index mark. Otherwise its value is set to NULL.

`int register_callback` (*callback_function_t* callback_function*) [Function]

This function registers a function to be called whenever an event or a custom index mark is reached during playing the audio for the synthesized message on the audio device.

callback_function is the function to be used as a callback. Please see ([[callback_function_t](#)], [page 19](#)) for details about the exact form.

On success, the value 0 is returned. In case of an error, the value -1 is returned. When this function is not supported by the driver, -2 is returned.

3 Notes About the Interface

Intended use

The primary use for this interface is access of applications to a low level layer, provided either by a process or a library, managing the synthesizer drivers.

A subset of this interface can however be used to interface this low level layer with the synthesizer drivers themselves. Even the capabilities provided by a driver itself and those provided by the interface using this same driver can differ as some functionality can be emulated by this low level library or process. Notably SSML conversion or stripping, [interfacing with] audio output and callbacks.

The audio retrieval method is designed in such a way that it will bypass this middle layer when the application wants to receive audio, and it can also possibly bypass the driver if the synthesizer supports it, resulting in better performance.

Repeat, rewind, context pause

The rewind and context pause functionality can be implemented in applications for every synthesizer that supports some kind of event notification for plain text messages. For SSML messages, support for the variable position start inside the message, as described in ([`say_text()`], page 8) is needed. The least granularity for rewind and context pause is determined by the granularity with which the synthesizer supports events notification.

Rewind can work as follows: The event notification mechanism is used to determine the current position in the spoken text. The message is first canceled or deferred. The synthesis process is started again from a position *n* words or sentences forward or backward. If the synthesizer does not support this functionality, this can be emulated for plain text by simply sending only the desired part of the text. The application can possibly take advantage of the audio data already received.

The working of context pause is similar.

If supported by the synthesizer, the higher level can also make use of the `defer()` ([`defer()`], page 10) functionality for better performance. This way, the text of the message does not need to be transferred again after each pause or resume and the synthesizer can make use of the already computed results, particularly SSML parsing and syntax analysis.

Audio formats in use

This interface does not enforce any particular audio format to be used by the synthesizer. The API used to interface synthesizers should not limit the synthesizers or the applications in the formats used to transfer audio.

Limits will however be given by the implementation of the audio output mechanism in use. Any audio output format fits these requirements, but synthesizer and synthesizer driver authors must be aware that output in a format not supported by the audio technology in use will be useless for the user.

Appendix A Requirements on the API

This section defines a set of requirements on the interface and on speech synthesizer drivers that need to support assistive technologies on free software platforms.

A.1 Design Criteria

The Common TTS Driver Interface requirements will be developed within the following broad design criteria:

A.1.1 Focus on supporting assistive technologies first. These assistive technologies can be written in any programming language and may provide specific support for particular environments such as KDE or GNOME.

A.1.2 Simple and specific requirements win out over complex and general requirements.

A.1.3 Use existing APIs and specs when possible.

A.1.4 All language dependent functionality with respect to text processing for speech synthesis should be covered in the synthesizers or synthesis drivers, not in applications.

A.1.5 Requirements will be categorized in the following priority order: (MUST HAVE);, (SHOULD HAVE);, and (NICE TO HAVE);.

The priorities have the following meanings with respect to the drivers available under this API:

- (MUST HAVE): All drivers must satisfy this requirement.
- (SHOULD HAVE): The driver will be usable without this feature, but it is expected the feature is implemented in all drivers intended for serious use.
- (NICE TO HAVE): Optional features.

Regardless of the priority, full interface will be provided by the API, even when the given functionality is actually not implemented behind the interface.

A.1.6 Requirements outside the scope of this document will be labelled as (OUT OF SCOPE);.

A.1.7 An application must be able to determine if (SHOULD HAVE); and (NICE TO HAVE); features are supported for a given driver.

(See API ([\[list_drivers\(\)\]](#)), [page 6](#)).

A.2 Synthesizer Discovery Requirements

A.2.1. (MUST HAVE): An application will be able to discover all speech synthesizer drivers available to the machine.

See API ([Section 2.2 \[Speech Synthesis Driver Discovery\]](#), [page 3](#)).

A.2.2. (MUST HAVE): An application will be able to discover all possible voices available for a particular speech synthesizer driver.

See API ([\[list_voices\]](#), [page 6](#)).

A.2.3. (MUST HAVE): An application will be able to determine the supported languages, possibly including also a dialect or a country, for each voice available for a particular speech synthesizer driver.

Rationale: Knowledge about available voices and languages is necessary to select proper driver and to be able to select a supported language or different voices in an application. See API (Section 2.2 [Speech Synthesis Driver Discovery], page 3) and (Section 2.3 [Voice Discovery], page 6).

A.2.4. (MUST HAVE): Applications may assume their interaction with the speech synthesizer driver does not cause inappropriate blocking of system resources or affect other operating system components in any unexpected way. Especially, the synthesizer must not block audio output for other applications.

A.2.5. (OUT OF SCOPE): Higher level communication interfaces to the speech synthesizer drivers. Exact form of the communication protocol (text protocol, IPC etc).

Note: It is expected they will be implemented by particular projects (GnomeSpeech, KTTTS, SpeechDispatcher) as wrappers around the low-level communication interface defined below.

A.3 Synthesizer Configuration Requirements

A.3.1 (MUST HAVE): An application will be able to specify the default voice to use for a particular synthesizer, and will be able to change the default voice in between ‘speak’ requests.

See API (Section 2.6.2 [Voice Selection], page 11).

A.3.2 (SHOULD HAVE): An application will be able to specify the default prosody, voice attributes and style settings for a voice for a given message by calling explicit functions for setting these parameters. These settings will match those defined in the SSML specification (see [appendix-B], page 25), and the synthesizer may choose which attributes it wishes to support. Note that prosody, voice and style elements specified in SSML sent as a ‘speak’ request will temporarily override the default values.

A.3.3 (SHOULD HAVE): An application should be able to provide the synthesizer with an application-specific pronunciation lexicon addenda. Note that using ‘phoneme’ element in SSML is another way to accomplish this on a very localized basis, and will override any pronunciation lexicon data for the synthesizer.

Rationale: This feature is necessary so that the application is able to speak artificial words or words with explicitly modified pronunciation (e.g. "the word ... is often mispronounced as ... by foreign speakers").

See API (Section 2.6.5 [Dictionaries], page 15).

A.3.4. (MUST HAVE): Applications may assume they have their own local copy of a synthesizer and voice. That is, one application’s configuration of a synthesizer or voice should not conflict with another application’s configuration settings.

A.3.5. (MUST HAVE): Changing the default voice or style and prosody settings does not affect a ‘speak’ in progress.

A.4 Synthesis Process Requirements

A.4.1 (MUST HAVE): The speech synthesizer driver is able to process plain text (i.e. text that is not marked up via SSML) encoded in one of the Unicode character encodings.

See API ([say_text()], page 8).

A.4.2 (MUST HAVE): The speech synthesizer driver is able to process text formatted using extended SSML markup defined in (see [appendix-B], page 25) of this document and encoded in Unicode. The synthesizer may choose to ignore markup it cannot handle or even to ignore all markup as long as it is able to process the text inside the markup.

A.4.3 (SHOULD HAVE): The speech synthesizer driver is able to properly process the extended SSML markup defined in the (see [appendix-B], page 25). of this document as SHOULD HAVE. Analogously for NICE TO HAVE.

A.4.4 (MUST HAVE): An application must be able to cancel a synthesis operation in progress. In case of hardware synthesizers, or synthesizers that produce their own audio, this means cancelling the audio output as well.

See API (Section 2.5 [Speech Control Commands], page 10).

A.4.5 (REMOVED: Moved to performance guidelines.)

A.4.6. (SHOULD HAVE): The speech synthesizer driver should honor the Performance Guidelines described below.

A.4.7. (NICE TO HAVE): It would be nice if the interface supported "rewind" and "repeat" functionality for an utterance.

Rationale: This allows moving over long texts without the need to synthesize the whole text and without losing context.

See API (Section 2.5 [Speech Control Commands], page 10)

A.4.8 (NICE TO HAVE): It would be nice if a synthesizer were able to support multi-lingual utterances.

A.4.9 (MUST HAVE): If the synthesized audio is being played, it must be possible to discover when the playback started and when it terminated.

A.4.10 (NICE TO HAVE): It would be nice if the synthesizer supported notification of custom index marks inserted as the 'mark' element, and if the application was able to align these events with the synthesized audio.

A.4.11 (NICE TO HAVE): It would be nice if a synthesizer supported "word started", "word ended", "sentence started" and "sentence ended" events and allowed alignment of the events similar to that in 4.9.

Rationale: This is useful to update cursor position as a displayed text is spoken. It is also essential for rewinding and context pause capabilities.

A.4.12 (REMOVED: not directly important for accessibility)

The former version: It would be nice if a synthesizer supported timing information at the phoneme level and allowed alignment of the events similar to that in 4.9. Rationale: This is useful for talking heads.

A.4.13. (NICE TO HAVE): The application must be able to pause and resume a synthesis operation in progress while still being able to handle other synthesis requests in the meantime. In case of hardware synthesizers, this means pausing and if possible resuming the audio output as well.

See API (Section 2.5 [Speech Control Commands], page 10).

A.4.14 (REMOVED: not clear purpose and against SSML specification) The synthesizer should not try to split the contents of the 's' SSML element into several independent

pieces, unless required by a markup inside. Rationale: An application may have better information about the synthesized text and perform its own splitting of sentences.

A.4.15 (OUT OF SCOPE): Message management (queueing, ordering, interleaving, etc.).

A.4.16 (OUT OF SCOPE): Interfacing software synthesis with audio output.

A.4.17 (OUT OF SCOPE): Specifying the audio format to be used by a synthesizer.

A.5 Performance Guidelines

In order to make the speech synthesizer driver actually usable with assistive technologies, it must satisfy certain performance expectations. The following text provides a clue to the driver implementors to get a rough idea about what is needed in practice.

Typical scenarios when working with a speech enabled text editor:

A.5.1 Typed characters are spoken (echoed).

Reading of the characters and cancelling the synthesis must be very fast, to catch up with a fast typist or even with autorepeat. Consider a typical autorepeat rate 25 characters per second. Ideally within each of the 40 ms intervals synthesis should begin, produce some audio output and stop. To perform all these actions within 100 ms (considering a fast typist and some overhead of the application and the audio output) on a common hardware is very desirable.

Appropriate character reading performance may be difficult to achieve with contemporary software speech synthesizers, so it may be necessary to use techniques like caching of the synthesized characters. Also, it is necessary to ensure there is no initial pause ("breathing in") within the synthesized character.

A.5.2 Moving over words or lines, each of them is spoken.

The sound sample needn't be available as quickly as in case of the typed characters, but it still should be available without clearly noticeable delay. As the user moves over the words or lines, he must hear the text immediately. Cancelling the synthesis of the previous word or line must be instant.

A.5.3 Reading longer messages

The speech synthesizer driver must be able to process longer input texts in such a way that the audio output starts to be available as soon as possible. An application must not be required to split long texts into smaller pieces.

A.5.4 Reading a large text file.

In such a case, it is not necessary to start speaking instantly, because reading a large text is not a very frequent operation. One second long delay at the start is acceptable, although not comfortable. Cancelling the speech must still be instant.

Appendix B Extended SSML Markup

This section defines the set of extended SSML markup and special attribute values for use in input texts for the drivers. The markup consists of two namespaces: 'SSML' (default) [SSML], page 34 and 'tts', where 'tts' introduces several new attributes to be used with the 'say-as' element and a new element 'style'.

If an SSML element is supported, all its mandatory attributes by the definition of SSML 1.0 ([SSML], page 34) must be supported even if they are not explicitly mentioned in this document.

This section also defines which functions the API needs to provide for default prosody, voice and style settings, according to [A.3.2], page 22.

Note: According to available information, SSML is not known to suffer from any IP issues.

B.1 (SHOULD HAVE): The following elements are supported

1. `speak`
2. `voice`
3. `prosody`
4. `say-as`

B.1.1 These SPEAK attributes are supported

1. (SHOULD HAVE): `xml:lang`

B.1.2 These VOICE attributes are supported

1. (SHOULD HAVE): `xml:lang`
2. (SHOULD HAVE): `name`
3. (NICE TO HAVE): `gender`
4. (NICE TO HAVE): `age`
5. (NICE TO HAVE): `variant`

B.1.3 These PROSODY attributes are supported

1. (SHOULD HAVE): `pitch` (with +/- %, "default")
2. (SHOULD HAVE): `rate` (with +/- %, "default")
3. (SHOULD HAVE): `volume` (with +/- %, "default")
4. (NICE TO HAVE): `range` (with +/- %, "default")
5. (NICE TO HAVE): `pitch`, `rate`, `range` with absolute value parameters

Note: The corresponding global relative prosody settings commands (not markup) in TTS API represent the percentage value as a percentage change with respect to the default value for the given voice and parameter, not with respect to previous settings.

B.1.4 The `say-as` attribute `interpret-as` is supported with the following values

1. (SHOULD HAVE): characters The format `glyphs` is supported.

Rationale: This provides capability for spelling.

2. (SHOULD HAVE): `tts:char`

Indicates the content of the element is a single character and it should be pronounced as a character. The element's contents (CDATA) should only contain a single character.

This is different from the interpret-as value `characters` described in [B.1.3], page 25. While `characters` is intended for spelling words and sentences, `tts:char` means pronouncing the given character (which might be subject to different settings, as for example using sound icons to represent symbols).

If more than one character is present as the contents of the element, this is considered an error.

Example:

```
<say-as interpret-as="tts:char">@</say-as>
</speak>]
```

Rationale: It is useful to have a separate attribute for "single characters" as this can be used in TTS configuration to distinguish the situation when the user is moving with cursor over characters from the situation of spelling. As well as in other situations where the concept of "single character" has some logical meaning.

3. (SHOULD HAVE): `tts:key` The content of the element should be interpreted as the name of a keyboard key or combination of keys. See section (C) for possible string values of content of this element. If a string is given which is not defined in section (C), the behavior of the synthesizer is undefined.

Example:

```
<say-as interpret-as="tts:char">shift_a</say-as>
</speak>
```

4. (NICE TO HAVE): `tts:digits` Indicates the content of the element is a number. The attribute "detail" is supported and can take a numerical value, meaning how many digits should the synthesizer group for reading. The value of 0 means the number should be pronounced as a whole appropriate for the language, while any non-zero value means that a groups of so many digits should be formed for reading, starting from left.

Example: The string "5431721838" would normally be read as "five billion four hundred thirty seven million ..." but when enclosed in the above say-as with detail set to 3, it would be read as "five hundred forty three, one hundred seventy two etc." or "five, four, three, seven etc." with detail 1.

Note: This is an extension to SSML not defined in the format itself, introduced under the namespace 'tts' (as allowed in SSML 'say-as' specifications).

B.2 (NICE TO HAVE): The following elements are supported

1. `mark`
2. `s`
3. `p`
4. `phoneme`
5. `sub`

B.2.1. (NICE TO HAVE): These P attributes are supported:

1. `xml:lang`

B.2.2. (NICE TO HAVE): These S attributes are supported

1. `xml:lang`

B.3. (SHOULD HAVE): An element `'tts:style'` (not defined in SSML 1.0) is supported.

This element can occur anywhere inside the SSML document. It may contain all SSML elements except the element `'speak'` and it may also contain the element `'tts:style'`.

It has two mandatory attributes `'field'` and `'mode'` and an optional string attribute `'detail'`. The attribute `'field'` can take the following values

1. `punctuation`
2. `capital_letters`

defined below.

If the parameter field is set to `'punctuation'`, the `'mode'` attribute can take the following values

1. `none`
2. `all`
3. (NICE TO HAVE): `some`

When set to `'none'`, no punctuation characters are explicitly indicated. When it is set to `'all'`, all punctuation characters in the text should be indicated by the synthesizer. When set to `'some'`, the synthesizer will pronounce those punctuation characters enumerated in the additional attribute `'detail'` or will only speak those characters according to its settings if no `'detail'` attribute is specified.

The attribute `detail` takes the form of a string containing the punctuation characters to read.

Example:

```
<tts:style field="punctuation" mode="some" detail="?!">
```

If the parameters field is set to `'capital_letters'`, the `'mode'` attribute can take the following values

1. `no`
2. `spelling`
3. (NICE TO HAVE): `icon`
4. (NICE TO HAVE): `pitch`

When set to `'no'`, capital letters are not explicitly indicated. When set to `'spell'`, capital letters are spelled (e.g. "capital a"). When set to `'icon'`, a sound is inserted before the capital letter, possibly leaving the letter/word/sentence intact. When set to `'pitch'`, the capital letter is pronounced with a higher pitch, possibly leaving the letter/word/sentence intact.

Rationale: These are basic capabilities well established in accessibility. However, SSML does not support them. Introducing this additional element does not break the possibility of outside applications to send valid SSML into TTS API.

B.4 (NICE TO HAVE): Support for the rest of elements and attributes defined in SSML 1.0. However, this is of lower priority than the enumerated subset above.

Open Issue: In many situations, it will be desirable to preserve whitespace characters in the incoming document. Should we require the application to use the 'xml:space' attribute for the speak element or should we state 'preserve' is the default value for 'xml:space' in the root 'speak' element in this case?

Appendix C Key Names

C.1 General Rules

Key name may contain any character excluding control characters (the characters in the range 0 to 31 in the ASCII table and other “invisible” characters), spaces, dashes and underscores.

The recognized key names are:

- Any single Unicode character, excluding the exceptions defined above.
- Any of the symbolic key names defined below.
- A combination of key names defined below using the ‘_’ (underscore) character for concatenation.

Examples of valid key names:

```
A
shift_a
shift_A
$
enter
shift_kp-enter
control
control_alt_delete
```

C.2 List of symbolic key names

Escaped keys

```
space
underscore
dash
```

Auxiliary Keys

```
alt
control
hyper
meta
shift
super
```

Control Character Keys

```
backspace
break
delete
down
```

end
enter
escape
f1
f2 ... f24
home
insert
kp-*
kp-+
kp--
kp-.
kp-/
kp-0
kp-1 ... kp-9
kp-2
kp-enter
left
menu
next
num-lock
pause
print
prior
return
right
scroll-lock
space
tab
up
window

Appendix D Requirements on the synthesizers

This section gives guidelines to the synthesizer authors and driver implementors about what capabilities should be supported by the synthesizers accessible under this API.

The requirements are sorted into three categories: (MUST HAVE):, (SHOULD HAVE):, (NICE TO HAVE): with meaning analogous to that specified in ([appendix-A], page 21). A synthesizer which does not fit all of the (MUST HAVE): requirements cannot be accessed under this interface.

1. General points
 1. (MUST HAVE): Interaction with the synthesizer must not cause inappropriate blocking of system resources or affect other operating system components in an unexpected way. Especially, the synthesizer must not block audio output for other applications.
2. Discovery of available voices
 1. (NICE TO HAVE): It would be nice if it was possible to discover all available voices.
 2. (NICE TO HAVE): It would be nice to have the possibility of discovering languages and possibly also countries or dialects supported by each voice.
3. Synthesizer configuration requirements
 1. The synthesizer should (would be nice to if) support configuration options as defined in the interface description under (Section 2.6 [Parameter Settings], page 11). The relevant priorities for these capabilities are specified as points A.3.1-A.3.3, A.3.5 of the requirements on the API ([appendix-A], page 21) and in the extended SSML subset in use specifications ([appendix-B], page 25).
4. Synthesis process requirements
 1. (MUST HAVE): The synthesizer must be able to process plain text as input.
 2. (NICE TO HAVE): If the synthesizer can't process Unicode encoding for the text, it would be nice if possible to determine the encoding used for a given voice and language.
 3. (SHOULD HAVE): The synthesizer should be able to process text formatted using extended SSML markup defined in (see [appendix-B], page 25) of this document and encoded in Unicode. The synthesizer may choose to ignore markup it cannot handle or even to ignore all markup as long as it is able to process the text inside the markup.
 4. (SHOULD HAVE): The speech synthesizer should be able to properly process the extended SSML markup defined in the (see [appendix-B], page 25). of this document as SHOULD HAVE. Analogously for NICE TO HAVE.
 5. (NICE TO HAVE): It would be nice if the synthesizer was able to start the synthesis process from a position in the text where an event (word or sentence boundary) occurs, as described in ([say_text()], page 8).
 6. (NICE TO HAVE): It would be nice if the synthesizer supported the ([defer()], page 10) capability or a similar compatible mechanism how to achieve good performance when rewinding and pausing/resuming inside long texts.

7. (MUST HAVE): An application must be able to cancel a synthesis operation in progress. In case of hardware synthesizers, or synthesizers that produce their own audio, this means cancelling the audio output as well.
 8. (MUST HAVE): If the synthesized audio is being played, it must be possible to discover when the playback started and when it terminated.
5. Audio retrieval
 1. (NICE TO HAVE): It would be nice if the synthesizer could retrieve audio data rather than play them itself. Preferably through the mechanism described in the interface definition ([Section 2.7 \[Audio Retrieval\], page 16](#)).
 6. Performance guidelines
 1. (SHOULD HAVE): The speech synthesizer driver should honor the Performance Guidelines described in ([\[appendix-A\], page 21](#)).
 2. (NICE TO HAVE): It would be nice if the synthesizer was able to process long input texts in such a way that the audio output starts to be available for playing as soon as possible. The driver is not required to split long texts into smaller pieces.
 7. Other requirements
 1. (NICE TO HAVE): It would be nice if a synthesizer were able to support multi-lingual utterances.
 2. (NICE TO HAVE): It would be nice if the synthesizer supported notification of events and custom index marks as defined in ([\[event_type-t\], page 7](#)) and if the application was able to align these events with the synthesized audio as in ([Section 2.7 \[Audio Retrieval\], page 16](#))

Rationale: This is useful to update cursor position as a displayed text is spoken. It is also essential for rewinding and context pause capabilities.

Appendix E Recommended Sound Icons

This appendix specifies a set of recommended names of sound icons to be used by the application and recognized by the synthesizer. The set is divided in groups according to their purpose.

[...]

Appendix F Related Specifications

1. [*SSML*], Speech Synthesis Markup Language, W3C,
<http://www.w3.org/TR/2004/REC-speech-synthesis-20040907/>
2. [*SSML-req*], SSML Requirements, W3C,
<http://www.w3.org/TR/2004/REC-speech-synthesis-20040907ref-reqs>
3. [*SSML-say-as*], SSML 'say-as' Element Attribute Values, W3C,
<http://www.w3.org/TR/2005/NOTE-ssml-sayas-20050526/>
4. [*MRCP*], MRCP, ,
<http://www.ietf.org/html.charters/speechsc-charter.html>

Index of Functions

C

cancel 10

D

defer 10

discard 11

driver_capabilities 6

G

get_current_voice 12

get_pitch_absolute_default 12

get_rate_absolute_default 12

get_volume_absolute_default 13

L

list_drivers 6

list_voices 6

R

register_callback 19

S

say_char 9

say_deferred 9

say_deferred_from_character 9

say_deferred_from_index_mark 9

say_icon 10

say_key 9

say_text 8

say_text_from_character 8

say_text_from_event 8

say_text_from_index_mark 8

set_audio_output 16

set_audio_retrieval_destination 16

set_capital_letters_mode 14

set_dictionary 15

set_driver 11

set_number_grouping 15

set_pitch_absolute 12

set_pitch_range_absolute 13

set_pitch_range_relative 13

set_pitch_relative 12

set_punctuation_detail 14

set_punctuation_mode 13

set_rate_absolute 12

set_rate_relative 12

set_voice_by_name 11

set_voice_by_properties 11

set_volume_absolute 13

set_volume_relative 13