Vol.2020-SLP-133 No.14
2020/10/23

組み込みシステムでの利用を想定したテキスト音声合成APIの設計

西澤 信行[1,a)]　小原 朋広[1]　服部 元[1]

概要：筆者らは組み込みシステムでの利用を想定し，マイクロコントローラ（MCU）上で単体動作する日本語テキスト音声合成ソフトウェアの開発を進めてきた．我々の音声合成ソフトウェアの特長の 1 つに，リアルタイム OS 等に依存しないシンプルな API（Application Program Interface）セットの提供があり，ソフトウェアには，この API のために設計を変更した箇所や，逆に使い勝手等の観点から API の動作に関する規定の方を変更した箇所がある．本論文では我々の音声合成 API について説明するとともに，実際のソフトウェア実装との関係についても述べる．

Design of text-to-speech API suitable for embedded systems

Nobuyuki Nishizawa[1,a)]　Tomohiro Obara[1]　Gen Hattori[1]

Abstract: We have developed Japanese text-to-speech software that can be executed on microcontroller units (MCUs) for embedded systems. One of the features of our software is the provision of a simple API (Application Program Interface) set that does not depend on the OS environment. In the development, the structure of some parts of the software has been changed for this API set, and conversely, the definitions of some APIs have also been modified to improve software usability. This paper explains our text-to-speech API set and the relationship between the API set and our implementation.

1. はじめに

近年，スマートフォンやスマートスピーカからの合成音声出力は一般的なものとなっている．スマートフォンやスマートスピーカでは Linux のような比較的高機能な OS を用いたシステムで実現されていることも多く，アプリケーション開発者から見て，比較的容易に音声合成機能を利用できるようになっているが，同様に音声合成システムの開発者にとっても，音声合成を行い，合成した音声を出力するようなソフトウェアの開発は容易になっている．例えば音声出力に必要なオーディオデバイスはデバイスドライバーと関連するオーディオ処理ライブラリによって高度に抽象化されていることから，OS 上で動くソフトウェアの開発では，実際のオーディオデバイスの差異をそれほど気にする必要はない．音声合成ソフトウェアでは，その処理に関するノウハウの秘匿等を目的に顧客にソースコードは開示せず，バイナリ（実行ファイル）のみを提供する場合も多いが，Linux 等の OS ではハードウェアの制御が抽象化されており，例えばテキスト入力に対してオーディオデバイスからその読み上げ音声が出力されるような，いわゆる高水準の API（application program interface）を提供するソフトウェアをバイナリ提供することも可能である．

これに対し，バッテリ駆動のウェアラブル機器や小型 IoT（Internet of Things）機器では，マイクロコントローラ（microcontroller unit, MCU）と呼ばれる，汎用のプロセッサコアを中心に SRAM やフラッシュメモリ，その他の周辺回路を集積した低コスト・低消費電力の半導体デバイスを用いることが多いが，MCU 向けのソフトウェアでは，ハードウェアに直接アクセスすることが普通である．筆者らは既に 32 ビットの MCU 上で動作する日本語テキスト音声合成ソフトウェアを開発している [1] が，MCU 向けの音声合成ソフトウェアで先述のような高水準な API を提供しようとするとソフトウェアがハードウェア設計に強く依存してしまうため，音声合成ライブラリ（ミドルウェア）の API では低水準の機能のみ提供し，ハードウェア制御部分はライブラリの利用者であるアプリ開発者が開発する形が基本である．また，MCU 向けのライブラリの開発言語には通常 C 言語 [2] が用いられ，API も C 言語の関数としてアプリケーションと結合させることが多いが，この際，ライブラリ側もアプリケーション側と同じ言語処理系を使わなければならない場合があり，ライブラリの処理

[1] 株式会社 KDDI 総合研究所
KDDI Research, Inc., Japan
[a)] no-nishizawa@kddi-research.jp

系間での移植性は重要な要素となる．ちなみに，実際にはPC やスマートフォンでも超小型・超軽量の音声合成ライブラリの需要は存在し，組み込みシステム向けの音声合成ライブラリをスマートフォンや PC に移植して用いることがあるため，その点でも移植性は重要である．

本稿では，特に MCU を使った組み込みシステムを対象とした音声合成 API 設計事例の 1 つとして，我々が設計した組み込みシステム向け音声合成ライブラリの API について説明する．また設計した API の動作が実際のシステムで問題になり得るとの懸念から，ライブラリの実装も含めて API 設計について検討した箇所があり，それらについても紹介する．

2. 組み込みシステム向けテキスト音声合成ソフトウェア [1]

本稿が対象とする組み込みシステムは先述のように主に MCU を用いたシステムであり，ソフトウェア環境としては，リアルタイム OS（RTOS）を使ったシステムや，OS を使わない（ベアメタル環境の）システムを想定している．

本節では，音声合成 API に関する議論の前提として，まず我々が開発した組み込みシステム向けテキスト合成ライブラリ（ミドルウェア）の概要と，その速度性能について述べる．

2.1 開発したテキスト音声合成ソフトウェアの概要

日本語テキスト音声合成システムの入力は漢字仮名交じり文であり，最終的な出力である音声合成波形を出力するためには，まず日本語テキストの解析処理が必要になる．我々が開発したシステムでは，漢字仮名交じり文を JEITA IT-4006 日本語テキスト音声合成用記号 [3] で規定されている仮名レベル表記に相当する音声合成用記号に変換し，この音声合成用記号から HMM 音声合成技術 [4] により合成音声を生成する構成としている．また，HMM 音声合成処理は，決定木による HMM パラメータの推定，パラメータ生成，音声波形生成の大きく 3 つの部分で構成される．

ところで，MCU に内蔵される RAM（他の論理回路と混載となるため通常は SRAM）のサイズは高性能品でも数百キロバイトから 1 メガバイトであり，内蔵フラッシュメモリのサイズも 2 メガバイトから 4 メガバイト程度である．テキスト音声合成処理を行うにはメモリサイズが不足しているため，通常，MCU に RAM やフラッシュメモリを追加で接続する必要がある．このうちフラッシュメモリについては，アクセス速度性能面での影響はあるものの，シリアル接続の NOR フラッシュ（最大で数百メガバイト程度）を使うことができる．一方でシリアル接続の大容量 RAM は（恐らくはその需要の問題から）普及しておらず，実際のシステムでは，MCU と外部 RAM との間がパラレルバス接続となる．この配線数が MCU の選定や設計のコスト，システムのハードウェア全体のサイズ，製造コストに大きく影響し，実際，外部 RAM の接続に対応していない MCU も多いため，MCU を使ったテキスト音声合成システムの普及には，MCU 内蔵の RAM だけで処理を行えることが重要な要素となる．

そこで我々は，HMM 音声合成におけるパラメータ生成処理において，従来の発話全体の最適化ではなく，ブロック単位で最適化を行う手法 [5] を導入することで，パラメータ生成処理以降の処理で，疑似的なストリーム処理を実現した．これにより，中間結果や出力結果を一時的に保持するために必要な RAM 領域を大幅に削減でき，シリアルバス接続のフラッシュメモリを外部に追加した MCU だけで，テキスト音声合成処理が可能となった．

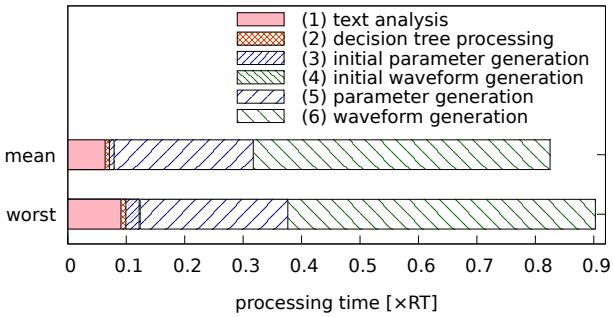

図 1 テキスト音声合成における各処理の処理速度．
Fig. 1 Processing speed of each processing of the TTS system.

2.2 処理速度性能

我々の先行研究 [1] では，実際に超小型（14 ピン DIP サイズ，17.78 mm × 7.62 mm）の MCU ボード（CPU コア: ARM Cortex-M4F 80MHz，RAM サイズ: 320KiB）を製作し，CPU のクロックサイクルカウンタを使った処理時間測定を行った．

図 1 は，女声声質の音声モデルで，ATR 音素バランス文 [6]503 文と同じテキストから音声を合成した場合の各処理の処理時間の平均値と最悪値であり，図の横軸は合成した音声の時間長を 1 とした場合の相対値（RTF）である．なお，ここでは音声合成処理を 6 つの処理に分けている．具体的には，先述のパラメータ生成と，音声波形生成の処理について，ストリーム処理を行った際に，最初の波形サンプル値を出力するまでの処理と，それ以降の処理の 2 つに分けて考えている．これは，最初の波形サンプル値を出力するまでの処理時間までがシステムの遅延時間（レイテンシ）に影響するためである．

図より，ストリーム処理を行える（ストリーム処理が可能になってて以降の，パラメータ生成処理と音声波形生成処理の処理時間の和が RTF で 1 以下となる）ことが分かるが，動作に必要な RAM サイズを削減するために，1 発話分の中間結果を保持しなければならないような API を設けることはできず，ストリーム処理に対応した API が必要となる．

3. 組み込みシステムに対応した音声合成 API の前提条件

先に述べたように，多くの組み込みシステムでは，ソフトウェアから見てハードウェアの抽象化が限定的なため，音声合成だけでなく音声出力も行うような，いわゆる高水準の API を提供する音声合成ソフトウェアは，対象のハードウェアに強く依存したものとなる．ソフトウェアとしての汎用性を持たせるためには，ハードウェアを制御する部分は別のソフトウェアとして分離させる必要があり，音声合成ライブラリの場合，汎用性を持たせられるのは，合成した音声の波形データ (PCM データ) を返す機能までとなる．本稿ではそのような API を低水準の API と呼び，オーディオデバイス等から合成した音声波形を出力するような機能は利用者側が作成するものとする．

また，音声処理のライブラリでは，プリエンプティブなマルチタスク機能の利用を前提に非同期で動作する API を設けているものもあるが，音声合成ライブラリが特定のマルチスレッドライブラリに依存してしまうと，ライブラリの移植性が悪化する．そこで我々は移植性をより重視し，全ての API が呼び出し側と同じスレッドで同期的に動作するような API を提供することとした．全ての処理は，API

を呼び出してから，API が処理を返すまでの間のみ，API を呼び出したスレッド上で処理が行われる．また，前述のようにプリエンプティブなマルチタスク機能が使える前提とはしないことから，1 回の API 呼び出しの処理時間が長くならないようにする必要もある[*1]．そして，これらの条件と利用者にとっての利便性を両立させるため，API の動作は状態依存（ステートフル）なものとし，その状態はインスタンス単位で管理される．

ところで，API の設計上，ライブラリ実装でヒープ領域 (具体的には malloc と free) を使える前提とした．後述する create API はインスタンスハンドルと呼称されるポインタを返すが，インスタンスハンドルはヒープ領域に確保された音声合成ライブラリのインスタンス変数領域を指すポインタを想定したものである．

4. 音声合成 API の構成

設計した音声合成 API では，音声合成ライブラリをオブジェクトファイルや共有ライブラリの形で提供できるように，リンク互換 (ここでは，音声合成ライブラリのオブジェクトファイルを置き換え再リンクする，もしくはダイナミックリンクライブラリを単に置き換えることで，ライブラリ実装の置き換えが可能なことをいう) となるよう意識している．

基本となる API は，以下の 12 種類である．

- create
- destroy
- addLexicon
- releaseAllLexicons
- addVoice
- releaseAllVoices
- setParam
- getParam
- setText
- synthesize
- getLength
- getPosition

ただし，各 API は C 言語のグローバルな関数として定義されるため，実際のライブラリ実装では上記の API 名に対してさらにライブラリを識別するプレフィックスを付ける．例えば，音声合成ソフトウェア「N2」[7] が提供する API セットでは，上記の API 名に `n2tts_`というプレフィックスを付けたもの (例えば `n2tts_create`) が，実際に定義される関数名である．

また，テキスト音声合成処理には上記の API 全てが必要ではなく，サブセットの API セットを提供することも考えられる．ただし，現在我々が作成している音声合成ライブラリでは上記 12 種類の API を全て提供している．

この API セットは利便性と保守性のバランスを考えて定義したものである．例えば，リンク互換性を重視する立場では，異なる機能を異なる API で提供するより，同じ API で引数により機能を切り替える方が互換性をより容易に維持できると考えられるが，必要な引数セットが機能間で異なる場合，API の引数設定が面倒なものになり，利用者の利便性が損われる．また，API の呼び出し回数を最小化するようなより高水準寄りな API の提供は，特定の利用者にとっては便利かもしれないが，API セットがさらに複雑なものになってしまう危険性がある．

4.1 API 共通仕様

先述したように，API を呼び出したスレッドと同じスレッドで処理が行われる．

また，後述する create を除き，API はその戻り値として符号付きの整数値を返し，エラー生じた場合は −1 を除く負の値として定義されるエラーコードを返す．

ちなみに，C 言語において `stdint.h` ヘッダファイルで定義される `intN_t` 型や `uintN_t` 型は，特に組み込みシステムを対象とした処理系において，特定の N に対してサポートされない可能性があることから，API 定義での利用を避けた．

4.2 音声合成ライブラリのインスタンスの生成と破棄

以下，API の定義については，基本的に C 言語の記法を用いる．

まず，

```
typedef struct instance_struct instance;
```

と定義し，この `instance` 型のポインタをインスタンスに関するデータ構造を抽象的に扱う変数とする．以下ではこれをインスタンスハンドルと呼ぶ．

音声合成ライブラリを初期化し，そのインスタンスを返す API である create と，インスタンスを破棄する API である destroy は，それぞれ以下の様に定義される．

```
instance *create();
int destroy(instance *ih);
```

create は音声合成ライブラリのインスタンスを初期化し，そのインスタンスハンドルを返す．もしインスタンスの初期化に失敗した場合は値 NULL を返す．一方，destroy はインスタンスハンドルをその引数とし，必要な終了処理を行い，インスタンス変数のためのメモリ領域を開放する．その戻り値は，正常終了時は 0，エラー時はエラーコードの負の値となる．

API の設計上は，create により複数のインスタンスを作成し，異なるインスタンスに対しては異なるスレッドから同時に API を呼ぶことができる実装を想定している．実際に現在の我々の実装は，malloc，free の実装がスレッドセーフであれば，このような API の利用が可能である．

4.3 語彙辞書データの登録と解放

addLexicon はテキスト音声合成処理に必要な，語彙辞書 (lexicon[*2]) のデータ領域を設定する API である．API 設計時の想定では，この API はデータ領域のアドレスを渡すためのもので，辞書データのコピーは行わない．このため，destroy あるいは後述する releaseAllLexicons の呼び出しまで，設定したデータ領域の書き換えは原則禁止となる (厳密には書き換え可否はライブラリ実装に依存する)．

API の定義は以下の通りである．

```
int addLexicon(
        instance *ih, int type,
        const void *lexicon, size_t size);
```

ここで，引数 `type` は辞書形式を示す整数値，`lexicon` は辞書データ領域の先頭アドレス，`size` は辞書データ領域のサイズである．また，戻り値は，正常終了時は 0，エラー時はエラーコードの負の値となる．

API の名称を add としているのは，addLexicon を複数回呼び出し，同時に複数の辞書をロードするような使い方を想定していることによる．実際我々の実装では，いわゆ

[*1] プリエンプティブなマルチタスク機能を備えた RTOS 環境で音声合成ライブラリを使い，API の処理途中に RTOS の機能により別のタスクに切り替えることはできる．

[*2] 音声認識および音声合成のための発音辞書フォーマットに関する W3C (World Wide Web Consortium) 勧告である Pronunciation Lexicon Specification (PLS)[8] を意識し，lexicon という用語を用いている．ただし現在の実装は PLS を直接サポートしていない．

るユーザ辞書データの登録にも本 API を用いる．

一方 releaseAllLexicons は addLexicon で登録された lexicon のデータ領域を全て解放する API で，

```
int releaseAllLexicons(instance *ih);
```

と定義される．また，AddLexicon で複数の lexicon を登録した場合に，一部の lexicon だけを解放する API は定義していない．これは，先述のように addLexicon ではデータ領域のコピーが行われないので，一部の辞書だけを無効化したい場合に，releaseAllLexicons して登録されている全ての辞書を一度解放し，それから使う辞書を addLexicon し直しても，そのオーバヘッドは大きくない，という前提があることによる[*3]．

4.4 音声モデルデータの登録と解放

音声モデル (声質モデルと呼ぶこともある) は，例えば HMM 音声合成においては音響モデルと呼ばれる，合成音声の音響的特徴を言語情報から予測するモデルに相当する．システムによってはこのデータをボイスフォントと呼ぶこともある．

音声モデルデータの登録と解放に対応する API はそれぞれ addVoice, releaseAllVoices で，データの扱いに関する考え方は，先述の語彙辞書データに関する API（addLexicon, releaseAllLexicons）と同様である．実際の API は以下の形で定義される．

```
int addVoice(
        instance *ih, int type,
        const void *voice, size_t size);
int releaseAllVoices(instance *ih);
```

ここで引数 `type` は，addLexicon の同名の引数と同様に，データフォーマットを指定する引数である．

語彙辞書と同様に複数の音声モデルデータの同時登録を想定しているが，その具体的な用途としては，例えばモデル間補間による声質制御が考えられる．

4.5 システム動作パラメータの設定と取得

各インスタンスには，音声合成処理に関する動作の制御等を目的とした整数値の仮想的な配列があり，その値を設定する API が setParam，値を取得する API が getParam であり，それぞれ以下の形で定義される．

```
int setParam(
        instance *ih, int index, int value);
int getParam(
        instance *ih, int index,
        int *value);
```

ここで引数 `index` は仮想的な配列のインデックスであり，例えば現在の我々のライブラリ実装では

0: 波形振幅に関する係数（256 を 1 倍とする相対値）
1: 話速に関する係数（256 を 1 倍とする相対値）
2: 基本周波数シフト量（+768 で 1 オクターブ上がる）
3: 基本周波数変化に関する係数（256 を 1 倍とする相対値）
256: 出力波形のサンプリングレート

と定義されている．また getParam で取得された値はポインタ `value` が指す領域に格納される．なお，このように汎用的なインタフェースとしたのは，設定項目ごとに API を定義してしまうと API の数が増え，ライブラリの保守性が低下すると考えたためである．

ところで，パラメータとして実際に設定可能な値の範囲は，`int` 型の範囲よりも狭いことが多く，またサンプリングレートのようにある決まったいくつかの値しか設定できない場合もあるが，本来設定できない値を setParam で設定した場合の動作として，エラーを返す場合と，設定可能な範囲の値に丸める場合の 2 つがあり，パラメータの種類（すなわち `index`）毎に動作は規定され，値が丸められる場合，getParam により丸められた結果の値を取得できる．

なお，将来のライブラリ実装で，整数型以外の値のパラメータを設ける必要が生じた場合は，その型の値を設定，取得する API を追加することで対応する予定である．

4.6 音声合成対象テキストの指定

テキストの文字集合には Unicode[10] を用いるとして，API の設計段階では，そのテキスト表現として（1）UTF-8 による 8 ビット整数値の配列を用いる，（2）UTF-16 による 16 ビット整数値の配列を用いる，（3）符号化形式は実装依存とし，API ではこちらも実装依存の抽象化した型を定義してその配列とするか，汎用的なワイド文字列として int 型や unsigned int 型の配列を用いる，の主に 3 つが考えられた．これに対し，ライブラリ内部のテキスト処理で UTF-16 に類似した 16 ビット値単位の処理を行っており UTF-16 との親和性が高いこと，ライブラリの主なターゲットとして組み込みシステムを想定しており，複雑な変換処理をライブラリ内部に包含すべきではなく，またメモリ利用効率もより高めるべきと考えられたことから，上記（2）を基本とした．ただし API の引数としては，先述したように `uint16_t` 型の利用を避け，UTF-16 のテキストを unsigned short 型の配列に格納する[*4]．

API の定義は以下の通りである．

```
int setText(
        instance *ih, int texttype,
        const unsigned short *text,
        unsigned int length,
        unsigned int opt);
```

ここで引数 `texttype` は `text` のテキスト形式を指定する整数値で，例えば現在の我々のライブラリ実装では指定可能な値として

1: プレーンテキスト（漢字仮名交じり文）
4: JEITA IT-4006 日本語テキスト音声合成用記号 仮名レベル表記

がある．`text` は音声合成対象となるテキストの UTF-16 文字列であり，`length` でその長さを指定する．`text` がヌル文字により終端されている場合，`length` に `(unsigned int)-1` を指定しても良い．`opt` は将来の拡張のための引数で，現在の我々の実装では 0 を指定する．先述の setParam 経由で内部変数を設定する形でも，この引数 `opt` に相当するライブラリ実装への情報伝達が可能だが，setParam API が存在しない API サブセットを用いる場合を想定し，引数 `opt` を設けている．一方戻り値は，実際に設定された文字数であり，エラーが生じた場合はエラーコードを表す負の値を返す．ちなみに，引数 `length` の型を，addLexicon や addVoice でサイズを表す引数に使われている `size_t` 型ではなく unsigned int 型としている

[*3] 最初期の API 設計案には releaseAllLexicons が存在せず，destroy と create によるインスタンス再生成のみが辞書データ領域の唯一の解放手段だったが，インスタンス再生成により，addLexicon だけでなく，後述する addVoice，setParams の再呼び出しも必要となることから，それを避けるために releaseAllLexicons（および後述の releaseAllVoices）を追加した．

[*4] C11[9] で導入された `char16_t` 型への変更は今後の課題である．

のは，API の戻り値を他の API と同じ int 型に揃えるためである．

なお，setText は有効なインスタンスに対して任意の時点で呼ぶことができ，実際に setText を呼ぶと，前の setText 呼び出しの際に行われた内部状態設定が全てリセットされる．また，setParam で指定したパラメータ設定の多く（現在の我々の実装では全て）は setText を呼んだ時点でコピーされ，その後で setParam で設定を変更しても，後述する synthesize の動作には影響しない．

4.7 合成音声波形の出力

synthesize は合成した音声波形を返す API である．主に組み込みシステムでの用途を考えた場合に，合成音声結果として 16bit の PCM データが返されればほぼ全てのケースで充分と考えられることから，API の引数の設計で，出力形式の汎用性は目指さないことにした．

API の定義は以下の通りである．

```
int synthesize(
        instance *ih, short *buffer,
        unsigned int buflen,
        unsigned int timehint);
```

API は，setText で設定したテキストに対応する合成音声波形について，ポインタ `buffer` が指すメモリ領域に，それまでに返した合成音声波形の次のサンプルから順に最大で `buflen` 個の波形サンプル値 (振幅の値) を格納する．実際に `buffer` に格納されたサンプル数が正常終了時の API の戻り値になり，エラーが生じた場合はエラーコードを表す負の値を返す．ただし，全てのサンプルを出力し終えて 1 サンプルも出力できない状態になると，戻り値は −1 になる．また，引数 `timehint` は API が処理を返すまでの時間を決める引数であり，値が 1 のときはその時点で可能な最小の処理を行い，`(unsigned int)`−1 のときは，`buflen` 個の波形サンプルを出力するか，最後の波形サンプルを出力するまで処理を返さない．0 のときは何も行わず，0 または −1 を返す．それ以外の値についてはライブラリ実装依存であり，基本的には値が大きいほど処理を返すまでの時間が長くなる．この API は利用者が繰り返し呼ぶ想定である．

ちなみに，synthesize の設計当初の名前は getWaveform だったが，音声合成処理を行うような名前の API が無いと音声合成処理を行う別の API があるような誤解を与えるとの意見があり，現在の名称に変更した経緯がある．

4.8 合成音声波形サンプル数の取得

getLength は setText で設定したテキストを音声合成した結果の長さ (合成波形の合計サンプル数) を返す API で，定義は以下の通りである．

```
int getLength(instance *ih);
```

正常終了時の API の戻り値については当初 2 種類の動作に関する規定があり，そのどちらになるかは実装依存としていたが，現在の我々の実装では動作 2 となっている．

動作 1: API の呼び出し時点で合成音声の長さが確定している場合はその長さ (サンプル数) を返す．長さが確定していない場合は −1 を返す．

動作 2: 常に合成音声の長さ (サンプル数) を返す．

動作 2 とすることで，この API が処理を返すまでの時間が極端に長くなる場合がある．これに関しては 5.2 節にて述べる．

4.9 合成音声波形サンプル数の取得

getPosition は現在の波形出力位置を返す API で，定義は以下の通りである．

```
int getPosition(instance *ih);
```

正常終了時の API の戻り値は現在の出力位置を示す波形サンプル数で，0 以上の値である．

この API はそれまでに synthesize で出力した波形サンプル数を返すだけなので，システムの動作に必須の API ではない．ライブラリの内部変数として保持している値を，ライブラリ外で重複して保持しなくても良いように設けた API である．

4.10 音声合成 API の使用方法

音声合成 API の基本的な呼び出し順序は以下の通りになる．

(1) create でインスタンスを作成する．
(2) addLexicon で語彙辞書データを設定する．
(3) addVoice で音声モデルデータを設定する．
(4) 必要な場合は，setParam で追加のパラメータ設定を行う．
(5) setText で音声合成したいテキストを設定する
(6) synthesize で合成音声波形データを取得する．synthesize は全ての合成音声波形データを取得し終えるまで繰り返し呼び出す．
(7) destroy でインスタンスを解放する．

実際には，create で作成したインスタンスに対して，destroy を含めその他の API を任意の順序で呼ぶことができる．全ての API は同期処理を行うので，音声合成を中断したい場合は単に synthesize を繰り返し呼ぶことを中断すれば良く，同じインスタンスで別の文を新規に音声合成する場合も setText を呼べば良い．特別な中断処理等は不要である．

また，create でライブラリ実装に埋め込まれた語彙辞書データや音声モデルデータが自動的に設定され，追加のパラメータ設定をサポートせず，さらにインスタンスの解放を必要としない場合には，create，setText，synthesize の 3 つの API 実装があれば良い．さらに，PC 等，メモリがある程度潤沢に使える環境では，synthesize の出力先に充分に大きなバッファを設けるか，setText の直後に getLength で必要なバッファサイズを取得し，malloc 等で確保したそのサイズのバッファを出力先として，synthesize の `timehint` 引数に −1 を設定して 1 回の synthesize 呼び出しで全ての合成音声波形を取得するといった，利用者からみて比較的簡単と思われる API の使い方も可能であり，API の使い方について，ある程度の自由度を有している．

5. API 動作に関する検討

前節で説明した API の中で，その設計上，処理を返すまでの処理時間が極端に長くなる可能性を考慮しているのは，synthesize のみである．synthesize では `timehint` 引数により，API が処理を返すまでの時間をある程度制御できるようにしているが，これが可能であるのは，1 回の synthesize 呼び出しで `buffer` に格納されるサンプル数が，0 個も含めた可変の個数になっていることによる．

しかし，実際の実装では，設計上配慮している synthesize の他に，setText と getLength で比較的長時間の処理が生じる可能性があり，意図的にそれを許容している．以下ではそれぞれの背景や許容した理由について説明する．

これらは音声合成技術全体から見ると些細な点にも思われるが，特にプリエンプティブなマルチタスク機能を使え

ない組み込みシステムでは，システム全体の動作にも大きな影響を及ぼし得るため無視できない部分である．

5.1 setText と入力テキストに対するエラー処理

当初は，setText で設定されたテキストに対する実際の解析処理をできるだけ遅延させ，テキスト解析処理のほとんどを synthesize 内部で行うようにしていた．これにより setText 自体の処理はすぐに終了するものの，利用者より，setText が正常終了した後で synthesize がテキスト処理に関連したエラーを結果を返した場合に，そのエラーの原因が分かりにくいとの指摘があった．

これに対し，その後，図 1 で示したようにテキスト解析処理の処理時間が短縮されたこともあり，より詳細な入力テキストのチェック結果を setText の戻り値に反映させるため，現在の我々の実装では，setText の内部処理で入力テキストに関する解析処理までを全て行う形に setText の内部処理を変更している．しかし，MCU 上でのテキスト解析処理時間は RTF で約 0.1 程度あるため，比較的長いテキストを設定した場合に，setText が処理を返すまでの処理時間が長くなる．そのため現在は，あまり長いテキストを設定しないよう利用者側に求める形としている[*5]．

しかし，フォーマット的に不正なテキストが設定された場合を除くと，テキスト音声合成が，その入力テキストに起因して処理を断念すること自体が不適切であると考えることもできる．また，setText の処理において，不正なテキストフォーマット（プレーンテキスト入力に対しては，Unicode として不正なシーケンスを設定した場合等，また，JEITA IT-4006 仮名レベル表記については，規格上不正なシーケンスを設定した場合等）をテキスト処理の前処理として行うことは現実的に可能であり，setText の内部処理としてテキスト解析処理を行わなくても，不正な入力に対して適切なエラーを返すことができる．

setText の内部処理については現在も検討を継続しており，テキスト解析精度を向上させるために，より高コストな手法を導入する場合等に備えて，今後，setText の内部処理では上述したようなフォーマットチェックだけ行う形に変更する予定である．

5.2 getLength に関連する動作

4.8 節で説明した動作 2 のように getLength が常に正しい結果を返す仕様とすると，setText を呼んだ後，例えば synthesize が全く呼ばれていない状態でも正しい値を返す必要がある．例えば HMM 音声合成方式を用いたテキスト音声合成システムにおけるおける一連の処理を（1）テキスト処理，（2）HMM パラメータ推定，（3）音声合成パラメータ生成，（4）音声波形生成の 4 つの部分に分けたとき，getLength が返す値である setText で設定したテキストに対応する合成音声の長さを返すためには，(2) の少なくとも発話全体に対する音素継続長に関するパラメータを求める処理までを終える必要がある．(2) 以降の処理は synthesize の内部で行われる処理のため，当初は，getLength の動作は，4.8 節で説明した動作 1（発話全体の長さが決まる処理を終えるまでは，長さ不明を表す値 −1 を返す）としていた．

ところで，実際に setText の直後，syntheszie よりも前に getLength で合成音声の長さを得たい理由の 1 つとして，WAV 形式 [11] のような音声データフォーマットで，そのヘッダ部にデータサイズを格納していることがある．合成音声波形を全て取得するまでサンプル数が決まらない場合，synthesize で音声波形を全て取得し終えるまで WAV 形式のデータを出力できず，WAV 形式のデータを返すようなテキスト音声合成システムでは，そのレイテンシが著しく悪化するという問題が生じる．さらに，合成した音声波形データを利用者側で一時的に保存する必要もある．そこで我々のライブラリ実装では，先述の（1）と（2）は 1 発話全体に対する一括処理としてより早期に getLength が正しい値を返せるようにしている一方で，(3) と (4) はレイテンシを短縮するためにストリーム処理として実装している．しかし，実際の利用を考えた場合，あまり CPU の性能が高くない組み込みシステムでは，全体のサイズをあらかじめ知らなくても良い場合が多く，一方，WAV 形式の出力が必要な場合のほとんどは PC 等で利用であり，その場合，複数のプロセスやマルチスレッドを使った処理とすることが一般的で，API が処理を返すまでに多少時間が掛かってもそれほど問題が生じないことが多い．そのため，getLength については API の使いやすさの方を重視し，その応答時間が長くなる場合があることを許容して，常に正確な値を返す現在の動作（動作 2）に変更した．その内部の処理は，合成音声の長さが得られるまで，synthesize と同じ処理を行うだけである．

なお，getLength が長時間動作を返さないと問題が生じる場合，実装依存の動作となるものの，setText の後，最初のサンプルが取得できるまで synthesize を引数 `timehint` に例えば 1 を設定して繰り返し呼び最初のサンプルが取得できた後で getLength を呼べば良い．

[*5] 処理時間とは別に，ヒープメモリ消費の点で問題が生じることもこの理由の 1 つである．

6. おわりに

本稿では，組み込みシステム向けのテキスト音声合成システムのための API 設計について，我々の API を一例として紹介した．また，この音声合成 API を用いたシステムで問題になり得る 2 つの事例について，その検討内容及び実際の対応方法を説明した．

参考文献

[1] 西澤信行, 小原朋広, 服部元: マイクロコントローラを用いた日本語テキスト音声合成システムの開発，信学論, vol. J103–D (7), pp. 566–574 (2020.7).
[2] “ISO/IEC 9899:1990, Programming Languages – C,” International Organization for Standardization (1990.12).
[3] “日本語テキスト音声合成用記号，” 電子情報技術産業協会規格，JEITA IT-4006 (2010.3).
[4] Tokuda, K., Nankaku, Y., Toda, T., Zen, H., Yamagishi, J. and Oura, K.: “Speech synthesis based on hidden markov models,” Proc. of IEEE, vol. 101(5), pp. 1234-1252 (2013.5).
[5] 西澤信行，小原朋広，服部元: “音声合成におけるブロック処理によるパラメータ軌跡生成の検討，” 音講論集, 1-P-34，pp.1049–1052 (2019.9).
[6] 阿部匡伸，匂坂芳典，梅田哲夫，桑原尚夫: ” 研究用日本語データベース利用解説書 (連続音声データ編)，” TR–I–0166，ATR 自動翻訳電話研究所 (1990.8).
[7] KDDI 総合研究所: 音声合成ソフトウェア「N2」(online), 入手先 ⟨http://www.kddi-research.jp/products/n2.html⟩.
[8] “Pronunciation Lexicon Specification (PLS) Version 1.0,” W3C Recommendation (2008.10)
[9] “ISO/IEC 9899:2011, Information technology – Programming Languages – C,” International Organization for Standardization (2011.12).
[10] “The Unicode Standard Version 13.0,” Unicode Consortium (2020.3).
[11] IBM Corporation and Microsoft Corporation: *Multimedia Programming Interface and Data Specifications 1.0* (1991.8).