Windows プログラミングによる音声入力システムについて

松原 篤

製作技術課

1 はじめに

Windows プログラミングは，開発言語において，Windows が提供している API（Application Programming Interfaces）と呼ばれるアプリケーションから当該 OS を操作することが可能なインターフェースを運用することにより，実現される．

Windows API は，C，Python 等，現在主流のプログラム開発言語により，容易に呼び出すことができる反面，プログラム開発者には，API がサポートしている関数と開発言語との間を接続するデータ型に対するインターフェース整合の知識が要求される．

本報告では，Window API を活用した各種プログラミング言語による音声入力システムの構築について述べる．開発言語としては，C++，Python，また Visual Studio 系である C#の合計 3 種を採用しているが，各開発言語の流儀を活かしながら，それぞれ同種のシステム構築を行っている．API の中で，音声情報を扱えるものはマルチメディア API として提供されているが，本システムでは，デバイスの認識・開閉，メモリの確保，データ集録の開始・停止等において，それぞれ専用の関数を，適宜活用している．

なお，対象となるデータの型については，同一のプログラミング言語においても，複数の表記(型名)が同義的に用いられているため，この辺りの等価・互換性については，「附録」の章を参照されることが望ましい．

2 システムの概要

音声入力には，外部デバイスとした USB 接続タイプのエレクトレットコンデンサ型マイクロフォン(Sony 製 ECM-PCV80U；周波数特性：80～15kHz；感度: -40dB±3.5dB)を採用し，システムの骨子としては，ウィンドウ画面上に配置した各ボタンコントロールのクリックイベントを通じ，音声データの集録に関わる一連の処理を行うものである．

本章では主に，プログラミング言語独自の規則や定義に関わらないシステム設計に関する共通的な事項(考え方)について述べる．

図 1. コンデンサ型マイクロフォン

2.1 音声入力に関する API 関数 [1)]

システムの構築にあたり，代表的かつ，不可欠となる API 関数およびデータ型：構造体を表 1 に示す．表より，上位 9 つの関数は，それぞれ戻り値を有しており，そのデータ型は「符号なし 32 ビット整数型」である．また，waveInGetNumDev 関数を除く 8 つの関数，それぞれにおける処理の成功は，ゼロ値とした戻り値により確認される．

これらの関数や構造体には，複数の引数やメンバが定義されているが，本章では，以降，特に主要な項目に対してのみ触れることとし，それらのデータ型の表記については，一般的な C 言語のス

表 1. 音声入力用 API 関数/構造体

関数/構造体 名	用途
waveInGetNumDev	音声入力可能なデバイス数を取得
waveInGetDevCaps(A)	デバイス情報の確認
waveInOpen	デバイスのオープン(ハンドルの取得)
waveInPrepareHeader	音声データ格納用バッファの準備
waveInAddBuffer	データ格納のためのバッファ登録
waveInStart	データ入力の開始
waveInStop	データ入力の停止
waveInUnprepareHeader	バッファのクリーンアップ
waveInClose	デバイスのクローズ
WAVEINCAPSA	デバイス情報の取得
WAVEFORMATEX	音声データの形式を定義
WAVEHDR	バッファを識別するためのヘッダを定義

タイルを踏襲している．また，これらの API 関数を活用するためには，専用のライブラリ(winmm)をインポートする必要があるが，インポート・参照の方法については，プログラミング言語に対する開発環境やコンパイラに依存する．

```
MMRESULT waveInOpen (
  LPHWAVEIN phwi,                    // デバイスのハンドル(ポインタ)
  UINT        DeviceID,              // デバイス ID
  LPWAVEFORMATEX pwfx,               // WAVEFORMATEX のポインタ
  DWORD_PTR wCallback,               // ウィンドウハンドル
  DWORD_PTR dwCallbackInstance,      // インスタンス情報
  DWORD       fdwOpen                // デバイスオープン用フラグ
);
```

構築に際し最初の手続きとしては，waveInGetNumDev 関数により，内臓および外部接続された利用可能な音声入力デバイス数を取得する．次に，waveInGetDevCaps(A)関数により，WAVEINCAPSA 構造体を通じてデバイスに関する情報が得られるため，目的のデバイスに対し割りあてられた ID(変数 *DeviceID* に格納)を，以後導入する API 関数に適用する．

```
typedef struct WAVEFORMATEX_tag{
    WORD    wFormatTag;                // オーディオフォーマットタイプ
    WORD    nChannels;                 // チャネル数
    DWORD nSamplesPerSec;              // サンプルレート[Hz]
    DWORD nAvgBytesPerSec;             // 平均データ転送速度[bytes/s]
    WORD    nBlockAlign;               // ブロックサイズ
    WORD    wBitsPerSample;            // サンプルあたりのビット数[bit/Sample]
    WORD    cbSize;                    // フォーマットに対する追加情報サイズ
} WAVEFORMATEX, * LPWAVEFORMATEX;
```

waveInOpen 関数では，ID 情報をもとに，独自のハンドル(デバイスハンドル)を取得することにより，対象デバイスのオープン状態を実現する．waveInOpen 関数では，第 2 引数(変数名：*DeviceID*)において ID の指定により，デバイスハンドルが取得される(第1引数：ポインタ変数名 *phwi*)．第 3 引数のポインタ指定の WAVEFORMATEX 構造体では，フォーマットタイプやサンプルレート等，音声データの集録に必要となる各種パラメータを定義している．また，第 4 引数では GUI として作成するウィンドウ画面のハンドルを指定している．

```
MMRESULT waveInPrepareHeader (
  HWAVEIN   hwi,                     // デバイスのハンドル
  LPWAVEHDR pwh,                     // WAVEHDR のポインタ
  UINT    cbwh                       // WAWEHDR のサイズ
);
```

```
typedef struct WAVEHDR_tag{
    LPSTR   lpData;                    // バッファのポインタ
    DWORD   dwBufferLength;            // バッファの長さ
    DWORD   dwBytesRecorded;
    DWORD_PTR   dwUser;                // ウィンドウハンドル
    DWORD   dwFlags ;                  // 値：0
    DWORD   dwLoops;
    struct WAVEHDR_tag *lpNext;
    DWORD_PTR   reserved;
} WAVEHDR, *LPWAVEHDR;
```

デバイスをオープンした後，waveInPrepareHeader および waveInAddBuffer 関数により，データ格納用とするバッファの準備および登録を行う．双方の関数の引数条件は同等であるが，ポインタ指定の WAVEHDR 構造体では，データを格納するバッファを識別するためのヘッダが定義されている．

以上の手続きを踏まえ，waveInStart 関数によるデータ入力の開始，waveInStop 関数によるデータ入力の停止，また waveInClose 関数によるデバイスのクローズを順に行うことができる．なお，waveInUnprepareHeader 関数は，準備した上記バッファに関する設定条件をクリーンアップするため，活用する．

2.2 GUI

実際の API 関数の呼び出しについては，図 2 に示すようなウィンドウ画面に配置したボタンクラスのオブジェクト(ボタンコントロール)のクリックイベントの発生に伴い，逐次行えるようにしている．具体的には，①「Setting」ボタンにより waveInOpen 関数を，②「Start」ボタンにより waveInPrepareHeader，waveInAddBuffer，waveInStart 関数の順で，③「Stop」ボタンにより waveInStop 関数を，④「Close」ボタンにより waveInClose

図 2. ウィンドウ画面例

関数を，それぞれ呼び出すことができるよう実装している．また，「Save」ボタンにより，集録時間分のデータを，時系列としたテキストデータ(CSV ファイル)および音声データ(WAVE ファイル)として保存できるようにし，「End」ボタンにより，プログラムは終了する．

表 2．ボタンコントロールの有効/無効(○/×)状況

	起動時	クリック直後				
		Setting	Start	Stop	Save	Close
Setting	○	×	×	×	×	○
Start	×	○	×	○	○	×
Stop	×	×	○	×	×	×
Save	×	×	×	○	×	×
Close	×	○	×	○	○	×
End	○	○	○	○	○	○

なお，本システムでは，データ処理上において，イベント発生に伴う誤動作等の不具合の発生を回避するため，ボタンコントロールには，機能に関する有効/無効の切り替えが行えるようにしている(表 2 参照;図 2 は「起動時」状態)．

2.3 システム設計に関して

2.3.1 規約・設定条件

音声入力に関する各種パラメータの設定条件を表 3 に示す．サンプルレートは人の日常会話程度の音域(<1kHz)を想定し設定している．バッファのサイズはサンプルレートとブロックサイズ(指定したフォーマットの最小単位のデータサイズ)の積により，32,000 としたが，このサイズは 1 秒分のデータに相当する．集録時間分を考慮した場合，集録されるデータ点数は合計 160,000 (=32,000×5)となり，また全体のバッファ数も 5 個必要となるため，本システムでは，バッファ1個分を定義している WAVEHDR 構造体については，要素数 5 とした構造体配列として準備した．

表 3．設定条件

項目	値	主な変数名
フォーマットタイプ	WAVE_FORMAT_PCM(=1)	*wFormatTag*
チャネル数	1(=モノラル)	*nChannels*
サンプルレート[Hz]	16,000	*nSamplesPerSec*
平均データ転送速度[bytes/s]	32,000 (=*nSamplePerSec* ×*nBlockAlign*)	*nAvgBytesPerSec*
ブロックサイズ[バイト単位]	2	*nBlockAlign*
サンプルあたりのビット数[bit/Sample]	16	*wBitsPerSample*
フォーマットに対する追加情報のサイズ	0	*cbSize*
バッファサイズ	32,000 (=*nSamplePerSec* ×*nBlockAlign*)	*dwBufferLength*，(new)*BUFFER_SIZE*
集録時間[sec]	5	(new) *N_BUFFERS*

変数名：WAVEFORMATEX，WAVEHDR のメンバ
new: 新規に定義

2.3.2 ウィンドウメッセージについて

Windows の OS は，ユーザやプログラム由来のイベントに反応し処理を行うイベントドリブンであるが，発生したイベントはメッセージの通知として，認識される．表 4 に本システムで利用した音声入力 API 関数に対するメッセージを示す．表より，上位 3 つの関数に対するメッセージは既存に割り当てられているものであり，関数を呼び出す行為により自動的に通知されるものである．

表 4．音声入力 API に関するメッセージ

関数名	メッセージ	
	名称	hex.(dec.) 値
waveInOpen	WIN_OPEN	0x3BE (958)
waveInClose	WIN_CLOSE	0x3BF (959)
waveInStart	WIN_DATA	0x3C0 (960)
waveInStop	WIN_STOP	0x3C1 (961)

一方，waveInStop 関数では，通知されるメッセージがないため，独自設定とし通知も明示的に行っている．waveInStart 関数の呼び出しに伴うメッセージ「WIN_DATA」は，バッファの容量が上限に達した場合に通知される．本システムでは，1 秒ごとにメッセージの通知が行われることになるため，通知回数は 5 回である．

また，表 5 に，その他，必要に応じて活用しているメッセージを列挙しておく[2]．

表 5. システム構築に活用したウィンドウメッセージ

名称	値 hex. (dec.)	概要
WM_DESTROY	0x002 (2)	ウィンドウの破棄
WM_QUIT	0x000 (0)	プログラム終了
WM_PAINT	0x00F (15)	ウィンドウ内一部の再描画
WM_CLOSE	0x010(16)	ウィンドウのクローズ
WM_COMMAND	0x111 (273)	ウィンドウ配置のコントロールよりイベント発生

メッセージに応じた処理については，原則，アルゴリズム 1 に示したような疑似コード(pseudo code)にしたがう形となっている．WndProc は，ウィンドウメッセージに応じた処理を実装できる関数として知られているが，WndProc 関数のポインタは，ウィンドウ画面とメッセージの処理との関連付けを成立させる手段として活用される．

アルゴリズム 1. メッセージに応じた処理に関する疑似コード

```
Argument: hwnd is a pointer to an untyped
          msg is an unsigned integer
          wParam is a pointer casted to an unsigned integer
          lParam is a pointer casted to a signed long integer
  Return: ptr is a pointer casted to a signed long integer
 Require: num ⇐ number of button instance
          N   ⇐ number of messages required for API
function WndProc(hwnd, msg, wParam, lParam) : ptr
    if  msg=MESSAGE (e.g. WIN_OPEN)  then
        Processing_1
    ⋮
    else if  msg=WM_COMMAND  then
        for all k ∈ [0, num)
            If  wParam=buttonID[k] then
                Processing k
            end if
        end for
    else
        Processing_N
    end if
    Return DefWindowProc (hwnd, msg, wParam, lParam)
DefWindowProc : function that processes unrelated messages
```

WndProc 関数に対し，①*hwnd* はウィンドウ画面のハンドルを，②*msg* はメッセージの種類を，③*wParam* および *lParam* はメッセージに応じて内容が異なるものの，メッセージの詳細な情報を，それぞれ提供している変数である．各ボタン共に，クリックイベントの発生に伴いメッセージ「WM_COMMAND」が通知されることになるが，ボタンの識別は，変数 *wParam* に格納された ID 情報により，行える仕組みとなっている．

2.3.3　データの確保および保存

WAVEHDR 構造体を配列として準備するにあたり，当該構造体と，バッファデータの格納用メモリ領域(配列)とをまとめて「クラス」として定義することにより，実装上では，クラスオブジェクトの配列化を行っている．

データの保存については，時系列データ(テキストファイル：拡張子「csv」)として保存する場合，バッファデータはバイト型配列(バイト列)であるため，数値への型変換が必要となるが，本システムでは，ブロックサイズや 1 サンプルあたりのバイト数の条件により，1 つの数値データは 2 バイト分のデータより抽出される．

表 6. WAVE ファイルのヘッダ

名称	サイズ (バイト)	値
チャンク ID	4	"RIFF"
チャンクサイズ	4	*DataSize*-8
フォーマット	4	"WAVE"
サブチャンク 1 ID	4	"fmt␣"
サブチャンク 1 サイズ	4	16
オーディオフォーマット	2	1 (=*wFormatTag*)
チャネル	2	1(=*nChannels*)
サンプルレート	4	16,000(=*nSamplesPerSec*)
バイトレート	4	32,000(=*nAvgBytesPerSec*)
ブロックサイズ	2	2(=*nBlockAlign*)
サンプル当たりのビット数	2	16(=*wBitsPerSample*)
サブチャンク 2 ID	4	"data"
サブチャンク 2 サイズ	4	160,000(=*BUFFER_SIZE*×*N_BUFFERS*)

HeaderSize=44，　*DataSize=HeaderSize+BUFFER_SIZE*×*N_BUFFERS*

1 サンプルあたりのバイト数 *BytesPerSample= wBitsPerSample* / 8

␣：空白文字

また，音声データ(WAVE ファイル：拡張子「wav」)として保存する場合は，バイト型データそのものを利用するが，表 6 に示す音声入力に対するフォーマット情報を「ヘッダ」部として付加する必要がある[3]．ヘッダはパラメータ 13 項目，合計 44 バイトのサイズから成るが，ファイルの構成上，ヘッダ部の後続に集録時間分の音声データが位置付けされることになる．

なお，音声データは，”RIFF(Resource Interchange File Format)”と呼ばれるタグ付きデータを格納するためのメタファイル形式として保存され，ファイルはチャンク(情報のひとかたまりの意)の並びとして解釈される．

最後に，本システムでは，音声データの型変換や保存等，比較的計算量がかかる処理については，専用のスレッドにおいて行えるよう工夫している(マルチスレッド化)．マルチスレッドの導入は，システムの改良・拡張といった今後を見据え，全体的な処理量の増加に伴い要求が想定される複数タスクの並列的な処理などに有益である．

3　プログラミング言語による構築

本章では，前章の概要に基づき，実装について述べるが，章立てした趣旨を，「節」では言語の種類に，「項」では生成した GUI に，それぞれ設定している．参考のため，採用したプログラミング言語に対する開発条件を表 7 に示す．

表 7. 開発条件

言語	開発環境	機械語翻訳・実行法	GUI
C++	Sublime Text 3	g++ (MinGW6.3.0)	Windows API(Win32API)
	Qt Creator 7.0.0	g++ (MinGW8.1.0)	Qt 5.15.0
Python	Spyder 5	interpreter Python 3.9.12	Windows API(Win32API)
			Kivy 2.1.0
C#	Visual Studio 2019	csc v4.0.30319	WPF

3.1　C++

C++言語によるウィンドウ画面の生成については，現在，様々な方法(フレームワーク)があるが，本節では，①32 ビットプログラムに対応した Windows API(Win32API)と，②Windows 以外の OS(プラットホーム)においても動作が可能な Qt を利用した 2 通りを紹介する．開発環境に応じたライブラリの参照・インポート方法についても併せて示す．なお，コンパイラは伝統的な g++を採用している．

3.1.1　Windows API [4),5)]

〈画面・イベント〉

C/C++言語において Windows API を利用する方法として，ヘッダファイル「windows.h」の指定がよく知られている．「windows.h」を指定した場合，「__stdcall」として定義されている「WINAPI」という名の呼び出し規約の効果により，「WinMain」関数が実行時に最初に呼び出される関数となる．関数の引数の 1 つである「*hInstance*」はウィンドウ画面に関わるインスタンスハンドルであり，特に本システムのように，ボタンコントロールを，親コントロールであるウィンドウ画面に配置する子コントロールという立場で，ウィンドウ画面に対し関連付けを行う際，必要となる．

```
int WINAPI WinMain(
  HINSTANCE   hInstance,        // インスタンスハンドル
  HINSTANCE   hPrevInstance,    // 値:0
  LPSTR   lpCmdLine,            //
  int       nCmdShow            //   ウィンドウ画面に関するフラグ
);
```

ウィンドウ画面の生成は，WNDCLASSEX 構造体，RegisterClassEx 関数，CreateWindowEx 関数の組み合わせにより行うことができる．WNDCLASSEX 構造体は描画スタイル，アイコン，カーソル，背景色など画面の体裁・動作に関わる基本的な事項をウィンドウクラスのパラメータとして定義したものであり，これらのクラスパラメータを RegisterClassEx 関数により登録し，CreateWindowEx 関数によりウィ

```
typedef struct WNDCLASSEX_tag{
    UINT    cbSize;             // 本構造体のサイズ
    UINT    style;              // 描画スタイル
    WNDPROC    lpfnWndProc;     // メッセージ受信用関数へのポインタ
    int    cbClsExtrea;         //
    int    cbWndExtrea;         //
    HANDLE    hInstance;        // インスタンスハンドル
    HICON    hIcont;            // アイコンハンドル
    HCURSOR hCursor;            // カーソルハンドル
    HBRUSH hbrBackground;       // 背景色ハンドル
    LPCTSTR lpszMenuName;       //
    LPCTSTR lpszClassName;      // ウィンドウクラス名
    HICON    hIconSm;           //
} WNDCLASSEX;
```

ンドウ画面の実体化を行うという手続きとなる.

```
HWND CreateWindowEx(
    DWORD    dwExStype,          // 拡張ウィンドウスタイル
    PCTSTR   lpszClassName,      // ウィンドウクラス名
    PCTSTR   lpszWindowName,     // ウィンドウ画面の表示名
    DWORD    dwStyle,            // 描画スタイル
    int      x,                  // ウィンドウ画面の表示位置(水平)
    int      y,                  // ウィンドウ画面の表示位置(垂直)
    int      nWidth,             // ウィンドウサイズ(幅)
    int      nHeight,            // ウィンドウサイズ(高さ)
    HWND     hWndParent,         // 親ウィンドウのハンドル
    HMENU    hMenu,              // メニューハンドル or 子ウィンドウ ID
    IINSTANCE  hInstance,        // インスタンスハンドル
    PVOID    pParam;             //
);
```

WNDCLASSEX 構造体や CreateWindowEx 関数における各種設定値についてはデフォルト値や汎用的な値を採用しているため, 便宜上, 詳細は略するが, 構造体メンバ「*lpfnWndProc*」には前述したウィンドウメッセージに対する処理関数「WndProc」を, また同メンバ「*hInstance*」には前記インスタンスハンドルを, それぞれ指定している. また, CreateWindowEx 関数では, 画面の生成に成功した場合, 戻り値としてウィンドウハンドルが獲得される.

ボタンコントロールの生成・配置については, ウィンドウ画面と同様, CreateWindowEx 関数を利用する. この際, 図 2 のように合計 6 個のボタンを配置することになるため, 関数はループ適用とし, ボタン共通のパラメータに対しては, ①ウィンドウクラス名「*lpszClassName*」にはクラス名"BUTTON"を, ②描画スタイル「*dwStyle*」には子コントロールとした「WS_CHILD」および, ボタンのクリックイベントに伴いメッセージ「WM_COMMAND」の通知を行うことができる「BS_PUSHBUTTON」等を, ③「*hWndParent*」には親であるウィンドウのハンドルを, それぞれ指定し, 各ボタン固有のパラメータ値としては, ①表示名「*lpszWindowName*」にはそれぞれ文字列「Setting」～「End」を, ②「*hMenu*」にはそれぞれ数値「101」～「106」を割り当て HANDLE 型(型なしポインタ)に変換した「ボタン ID」を, 指定している.

アルゴリズム 2. メッセージループ

```
Msg msg;
while(GetMessage(&msg,NULL,0,0) !=0){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

ウィンドウ画面の表示は, ShowWindow 関数で行うことになるが, 画面の表示を維持するためには, GetMessage 関数によるメッセージループの実装が不可欠となる(アルゴリズム 2 参照). GetMessage 関数では Msg 構造体のポインタを指定することにより, メッセージの取得が可能となるが, メッセージ「WM_QUIT(=0)」を受信しない限り, ウィンドウ画面は終了しない(表示の維持)ようにしている. 詳細に言えば, 「End」ボタンを押した際, SendMessage 関数によりメッセージ「WM_CLOSE」を送信し取得できるメッセージ「WM_DESTOROY」に対し, PostQuitMessage 関数を用いてメッセージ「WM_QUIT」を明示的に送信することにより, メッセージループ, 延いてはプログラム(アプリケーション)が安全に終了するという仕組みを適用している.

参考のため, 図 2 のように, ボタンの表示名のフォントを書体「Elephant」タイプにカスタマイズしているが, フォントの変更には, gdi32 という名のライブラリにおいて定義されている CreateFontA 関数を利用している. 表 5 の「WM_PAINT」は, ウィンドウ画面が生成された直後に送信されるメッセージであるが, このメッセージの受信をフォントの変更のタイミングとしている. 以上により, ウィンドウ画面は, ボタン名のフォントが変更された状態で初出する.

〈環境設定〉

開発環境は「Sublime Text3」としたが, ライブラリファイルを参照するには, コンパイル・ビルド条件をカスタマイズする必要がある. 本システムではアルゴリズム 3 のように構成したが, この構成をファイル化し, コンパイル時に呼び出すようにしている. 手続きとしては, 開発環境画面において, 「Tools」メニュー→「Build System」→「New Build System」により作成・ファイル保存→コンパイル・実行時に「Build System」として選択設定, により, 可能となる. ライブラリは, ①音声入力に関わるものは「winmm」に, ②GUI 上のフォント・背景色に関するものは「gdi32」に, ③ウィンドウ画面(or ボタン)の生成・登録・表示や, コントロールの有効/無効等に関するものは「user32」に, それぞれ定義されているが, これら

のライブラリは実行コマンドにおけるリンカオプション「-l（エル）」を通じて，参照される．

〈データ確保〉

バッファあたりの音声データはBYTE型配列 *data* として確保している．配列 *data* は WAVEHDR 構造体を含むクラス「MyWavebuffer」のメンバとして定義し，MyWavebuffer クラスのオブジェクトを配列 *WBUF*[]として準備することにより，データの確保領域はバッファ数とバッファサイズからなる 2 次元配列と等価となる．本システムでは，メッセージ「WIN_DATA」の受信ごとに，配列 *data* の容量が順次上限に達することになる．

集録時間分のデータは，memcpy 関数により 2 バイト分のデータを short 型で確保し，short 型の値の有効範囲の絶対値最大(値：32768)による正規化を経てfloat型($0 \leq f1 \leq 1$)にしたものを，データ保存用に提供している(アルゴリズム 4)．

バッファごとのデータ確保および集録時間分のデータ保存に際しスレッドの確立には，

「_beginthreadex / _endthreadex」関数を利用している．関数の利用は「process.h」の指定により可能である．コードの骨子を疑似コードとしてアルゴリズム 5 に示すが，「_beginthreadex」関数が受け取る関数には，呼び出し規約「__stdcal1」がついており，スレッドが確立した場合は，スレッドハンドルを戻り値として得ることになる．なお，WaitForSignal Object 関数の働きは対象スレッド上の処理が終了するまで，スレッドを待機させることであり，CloseHandle 関数は対象スレッドのハンドルをクローズする．

アルゴリズム 3．コンパイル・ビルド条件

```
{
    "shell_cmd": "g++  \"${file}\" -o  \"${file_path}/${file_base_name}\"
    -lgdi32 -luser32 -lwinmm",
    "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
    "working_dir": "${file_path}",
    "selector": "source.c++",
   "encoding": "Shift_JIS",
   "variants":
    [
      {
         "name": "Run",
         "shell_cmd":"g++\"${file}\" -o  \"${file_path}/${file_base_name}\"
        -lgdi32 -luser32 -lwinmm &&  \"${file_path}/${file_base_name}\""
      }
    ]
}
```

```
BYTE *data;
data = (BYTE *) calloc((int)BUFFER_SIZE, sizeof(BYTE));
```

```
MyWaveBuffer *WBUF;
WBUF = new MyWaveBuffer[(int)N_BUFFERS];
```

アルゴリズム 4．データの確保

```
WAVEFORMATEX WFE;
0≤ BuffCounter < N_BUFFERS
for (int k=0; k<WFE. nSamplesPerSec; k++){
   short ddd;
   memcpy(&ddd, &WBUF[BuffCounter].data[(int)WFE.nBlockAlign*k], sizeof(short));
   float fff=(float) ddd/(float) 32768;
   Data[k+BufferCounter* WFE. nSamplesPerSec]=fff;
}
```

アルゴリズム 5．スレッド

Callee

unsigned __stdcal1 *function name* (void *p){
Processing
_endthreadex(0);
return;
}

Caller
thr_handle=(HANDLE) _beginthreadex(NULL, 0,*function name*,NULL,0,NULL) WaitForSignalObject(*thr_handle*,INFINITE) CloseHandle(*thr_handle*)

3.1.2 Qt [6)]

〈環境設定〉

開発環境「Qt Creator」では，ファイルの管理は，①C++ソースファイル，②ヘッダファイル，③GUI 開発用 UI ファイル(拡張子「ui」)，④前記①～③のファイルの指定や，GUI や API に関わるライブラリのインポート等，プログラムの構築に関わる各種設定を行うプロファイル(拡張子「pro」)を束ねた形のプロジェクトファイルとしてなされている．また，プログラムのコンパイル・機械語翻訳については，「qmake」と呼ばれる Makefile を自動生成するビルドシステムが採用されている．

本システムでは，必要となるライブラリを，プロファイル内にリンカオプションを介して追加・指定している．また，ヘッダファイルでは，コントロールのための「Widgets」，データの読み込み/書き出しなどの各種ヘッダ，広域変数といったものの他に，ボタンコントロールのクリックイベントに対する処理関数(スロット)の定義・宣言も行っている．

```
LIBS  += −lwinmm
LIBS  += −luser32
```

〈画面・イベント〉

開発環境の観点より，ウィンドウ画面については静的に生成可能な状態にあるため，踏襲することとし，ボタンコントロールについては，QPushButton クラスのオブジェクトを配列として動的に生成することにした．ウィンドウ画面の外観は図 2 に準拠したものとしている．なお，ウィンドウ画面のハンドルは「this–>winId()」メソッドを利用することにより，取得できる．

ボタンのクリックイベントによる処理は，「シグナル」としたクリック操作と，それに応じ処理を行う関数「スロット」とを接続する connect 関数の活用により，実現している(アルゴリズム 6 参照：connect 関数部以外は疑似コード表現)．スロット内での(イベント発生元の)ボタン情報の取得に関する手続きを，アルゴリズム 7 に示すが，QString 型変数「*btn_name*」に，イベント発生元である「Setting」～「End」のいずれかのボタン名が格納される．

アルゴリズム 6．シグナルとスロットの接続

```
Require :  num        ⇐ number of button instance
           btn[num]  ⇐  array for button instance
           for all k ∈ [0, num)
              connect(btn[k], SIGNAL(clicked()), SLOT(function name));
           end for
```

アルゴリズム 7．ボタン情報の取得

```
QPushButton *p = dynamic_cast<QPushButton *>(sender());
QString str = p–>text();
QString btn_name = str.toStdString();
```

ところで，本システムのウィンドウ画面では，前項紹介の「Windows API」システムと異なり，ウィンドウメッセージ処理用の WndProc 関数との関連付けが有効に行えなかったため，メッセージの取得に関しては，アルゴリズム 8 に示すように，メッセージループ内での GetMessage 関数の導入により，実現している(対象関数に対するメッセージの受信により，ループ終了)．

アルゴリズム 8．メッセージに応じた処理に関する疑似コード

```
while true do
    Message obtained by GetMessage()
    if Message=message assigned to the API func.
       break
    end if
    API func. for wave Input
end while
```

その他，音声入力に関わる各種設定・処理およびデータの確保・保存等に関しては，前項の「Windows API」システムを踏襲している．

3.2　Python

Python 言語では，ライブラリ「windll」の LoadLibrary メソッドを利用することにより，ライブラリ「winmm.dll」のインポートを容易に行うことができるが，言語の流儀・規則上，API 関数の運用に必要なデータ型やポインタを明示的に指定することができない．

そこで，システムの構築に際し，Python 言語においてもデータの型変換やポインタの指定が行える「ctypes」と呼ばれる外部関数ライブラリを導入した．ctypes は C 言語と互換性のあるデータ型を提供し，ダイナミックリックや共有ライブラリ内の関数を呼び出すことができる．ctypes の導入に関するコードを右に示すが，ライブラリ「windll」は ctypes のメンバであり，モジュール「wintypes」は WORD,HWND 等 Windows 固有(Win32API 型)のデータ型を提供する．

```
from ctypes import windll
from ctypes import wintypes
winmm=windll.LoadLibrary("winmm.dll")
```

開発環境である「Spyder」において，デフォルトの実行モード「Execute in current console」では，再実行の際，デバイスハンドルが取得できなくなる等，不具合が生じるため，実行モードについては外部コンソールによる実行を意味す

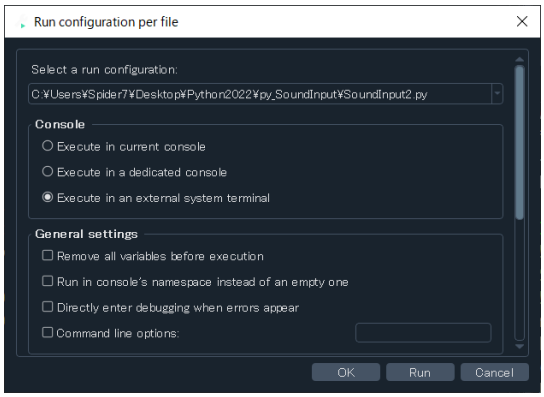


図 3. コンソールモードの変更

```
class WNDCLASSEX(ctypes.Structure):
    _fields_=[("cbSize",wintypes.UINT),
              ("style",wintypes.UINT),
              ("lpfnWndProc",WNDPROCTYPE),
              ("cbClsExrta",ctypes.c_int),
              ("cbWndExrta",ctypes.c_int),
              ("hInstance",wintypes.HANDLE),
              ("hIcon",wintypes.HANDLE),
              ("hCursor",wintypes.HANDLE),
              ("hbrBackground",wintypes.HANDLE),
              ("lpszMenuName",wintypes.LPCWSTR),
              ("lpszClassName",wintypes.LPCWSTR),
              ("hIconSm",wintypes.HANDLE)]
```

る「Execute in an external system console」」に切り替えるとよい(図 3 参照:画面は「Run」メニュー→「Configuration per file…」で表示可能).

3.2.1 Windows API

設計全般に際しては, 原則, 前節 C++言語による「Windows API」システムに準拠している.

〈画面・イベント〉

ウィンドウ画面に関しては, 定義は①WND CLASSEX 構造体, 登録は②windll.user32 ライブラリ定義の RegisterClassExW 関数, 生成は③同ライブラリ定義の CreateWindowExW 関数により, それぞれ行っている. ただし, Python には構造体がないため, 基本的にはクラスの概念により代用される. WNDCLASSEX 構造体を例に挙げると, 構造体の各メンバ(フィールド)が「_field_」シーケンスにより指定される様子がわかる.

アルゴリズム 9. WndProc 関数(定義, ポインタ, 処理関数)

```
WNDPROCTYPE=ctypes.WINFUNCTYPE(ctypes.c_uint,
                               wintypes.HWND,
                               ctypes.c_uint,
                               wintypes.WPARAM,
                               wintypes.LPARAM)

WndProc=WNDPROCTYPE(pyWNDPROC)

def pyWNDPROC(hWnd,Msg, wParam,lParam):
        Processing to Message
```

```
info=windll.user32.CreateWindowExW
info.restype=wintypes.HWND
info.argtypes=(wintypes.DWORD, wintypes.LPCWSTR,
               wintypes.LPCWSTR, wintypes.DWORD,
               ctypes.c_int, ctypes.c_int, ctypes.c_int, ctypes.c_int,
               wintypes.HANDLE, wintypes.HANDLE,
               wintypes.HANDLE, ctypes.c_void_p)
hwnd=info(各引数に値を設定)
```

アルゴリズム 9 に, ウィンドウメッセージを受け取る WndProc 関数に対する定義, 関数ポインタ, 処理関数を, それぞれ順に示す. 関数ポインタ WndProc が WNDCLASSEX 構造体の第 3 メンバ *lpfnWndProc* に利用される.

例に挙げた CreateWindowExW 関数のように, 本システムで扱うすべての API 関数に対しては, 生成した関数オブジェクトに対し restype/argtypes プロパティを利用することにより, 戻り値および引数に対する型の宣言を忠実に行うことができる.

ボタンコントロールを配置したウィンドウ画面の表示は windll.ShowWindow 関数により行えるが, 画面表示の維持にはメッセージループが必要となる(アルゴリズム 2).

〈データ確保〉

アルゴリズム 10 に示したデータ確保

アルゴリズム 10. データの確保

```
WFE= WAVEFORMATEX()
WBUF=[0 for k in range(N_BUFFERS)]
0≤ BufferCounter < N_BUFFERS
for k in range(WFE. nSamplesPerSec):
    list=WBUF[BuffCounter].data[k*WFE.nBlockAlign:(k+1)*WFE.nBlockAlign]
    shortVal=array.array('h',bytes(list))
    val=shortVal[0]
    dat=val/32768
    Data.append(dat)
```

に関する一連の手続きはアルゴリズム 4 と等価である. リスト変数 *list* に格納できた 2 バイト分のデータに対する目的の数値変換は, short 型を意味する型モード「h」の指定を通じ array クラスのコンストラクタにより, 順次達成される. 集録時間分のデータはリスト変数 Data に格納される.

最後に, スレッドの確立は, threading.Thread クラスを利用しているが, コンストラクタの引数にデータ確保や保存等, 目的の処理を行う関数を指定することにより, 行える.

3.2.2 Kivy

Kivy [7]は, 近年, GUI 開発によく利用されているオープンソースライブラリであり, pip コマンドによりインストールを行い, ライブラリ「kivy」のインポートを経てプログラミングに活用できる.

アルゴリズム 11. ウィンドウ生成の基本スタイル

```
from kivy.app import App
from kivy.core.window import Window
from kivy.uix.button import Button
Window.size=(300,600)
class WindowCreate(App):
    def build(self):
        btn=Button()
            ⋮
        return btn
WindowCreate().run()
```

ウィンドウ画面の生成法の基本は, アルゴリズム 11 に示すように, モジュール「kivy.app」に属する App クラスを反映させた独自クラス「WindowCreate」に, build メソッドを実装し, クラスオブジェクトに対する run メソッドを働かせるところにある. 本システムでは, この生成法により図 2 のような画面構成を実現した. なお, ウィンドウ画面のハンドルは ctypes.windll.user32.FindWindowW 関数により取得できる.

ボタンコントロールについては, 生成したボタンオブジェクトに対し, bind メソッドにより, クリックイベントに関連づける処理関数を指定している. その他, 音声入力システムの実装については, 前項の「Windows API」システムと等価であるが, ウィンドウメッセージの受信に関しては, C++言語による Qt システムと同様, アルゴリズム 8 のループ処理系を継承している.

3.3 Visual Studio～C#

〈画面・イベント〉

WPF におけるウィンドウ画面のハンドル「*handle*」の取得は, アルゴリズム 12 に示したように, WindowInteropHelper クラスを利用し get メソッドにより, 行うことができる. ボタンコントロールのクリックイベントの実装は, 「RoutedEventHandler」と呼ばれるイベントハンドラ用デリゲートにより, 処理関数との関連付けを行い, 処理関数内ではボタンの名前「Setting」～「End」に応じた処理が行えるようにしている.

アルゴリズム 12. ウィンドウハンドルの取得

```
Public IntPtr handle
{
    get
    {
        IntPtr Helperhandle = new WindowInteropHelper(this).Handle;
        return Helperhandle;
    }
}
```

〈ウィンドウメッセージ〉

ウィンドウメッセージの受信については, WPF アプリケーションではメッセージを処理する関数をフックする方法がとられる(アルゴリズム 13). 対象のウィンドウ画面のハンドル「*handle*」を指定し生成した HwndSource クラスのオブジェクトに対し AddHook メソッドが適用される.

アルゴリズム 13. メッセージ処理関数へのフック手続き

```
HwndSource source=HwndSource.FromHwnd(handle);
source.AddHook(new HwndSourceHook(WndProc));
```

〈API 関数〉

```
[DllImport("winmm.dll", CharSet = CharSet.Unicode, SetLastError = true)] public static extern Int32 waveInOpen(arg1, …);
```

音声入力に関わる API 関数の導入は, DllImport 属性を利用した宣言により, 達成される. また, 音声入力の形式や各種パラメータの設定に活用した WAVEFORMATEX や WAVEHDR 等の構造体の宣言に関しては, StructLayout

属性により，メンバのメモリ領域に関するレイアウトを整えており，LayoutKind 列挙型の Sequntial フィールドの指定はレイアウトがメンバの宣言順にしたがったものとなる(「附録」④の事例を参照).

〈スレッド〉

スレッドの確立は，Threading.Thread クラスのコンストラクタにおいて，別スレッドで実行させたい処理関数を指定し，Thread クラスの Start メソッドにより行っている.

```
Thread thr = new Thread (new ThreadStart(function));
thr.Start();
```

余談であるが，WPF アプリケーションとしたプログラミング言語 VB による音声入力システムが C#システムの置換型になることも，実際の構築を通じて確認している.

4　動作例

構築したシステムの動作検証については，いずれの言語によるシステムでも同等であることを確認しているため，本章では，データの集録例として C++による Windows API システムの場合を紹介する.

著者による母音「あ」～「お」の発声音に対し集録した時系列データの全体図(0.5 秒分)を，昇順に図 4 に示す. 発声音は声帯の振動を空気に伝播させることにより，音圧として捕らえたものであるが，図より，いずれの発声音も，一定時間持続した，見かけ上，紡錘形を成している様子がわかる. また，図 5 に，各発声音信号において，特定時間帯(0.05 秒分：全体の 1/10 に相当)の時系列信号，および持続時間を含む上記 0.5 秒分による周波数解析結果を示す.

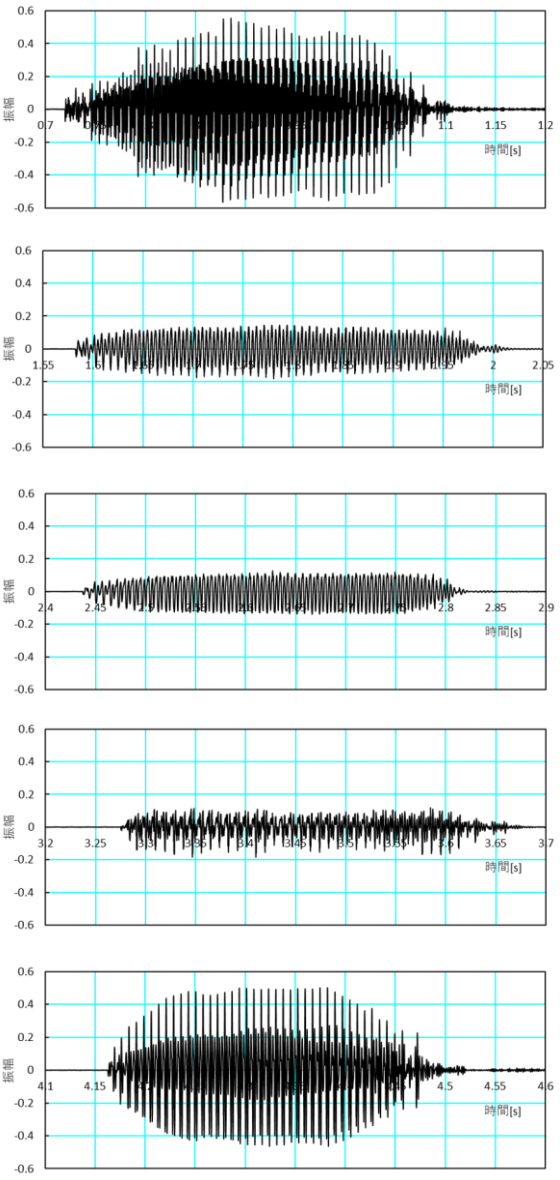


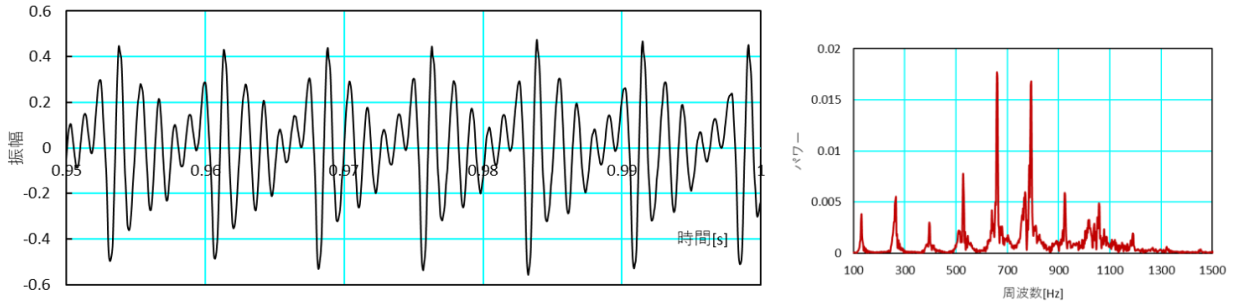
図 4.　音声信号例

図(a)より，「あ」については，660Hz，あるいは 790Hz 近傍を最大ピークとする周波数成分が固有振動，換言すれば，コンデンサ型マイクロフォンの動特性に起因する減衰振動として(周波数にして)132Hz 程度の間隔毎に繰り返し現れることがわかる(ピーク周波数間隔が固有振動間の時間間隔の逆数に相当).

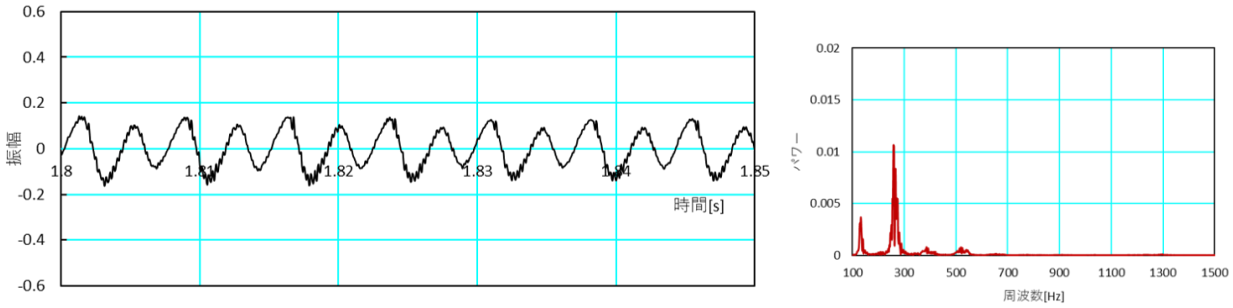
「い」および「う」は，「あ」に比べ，狭帯域にピーク周波数が存在し固有振動の周波数は低いものの，ピーク周波数の間隔は 130Hz 程度で同じである. また，双方の発声音は，周波数解析の結果や全体図(図 4)より，非常に類似した傾向をもつように見えるが，詳細に波形を観察すると，「い」の信号は正弦波状であるのに対し，「う」は半周期ごとに周期が異なっている様子が伺える[同図(b)および(c)].

「え」は，「い」および「う」と同様，500Hz 以下の帯域に複数のピークが見られるものの，大きさにさほど差がなく均等に含まれていることもあり，他の音声信号に比べ，複雑な信号波形となっている[(d)参照].

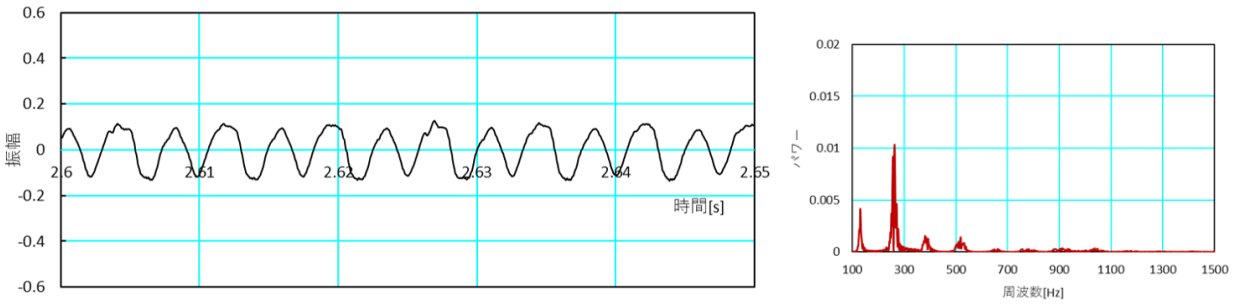
図(e)の発声音「お」については，固有振動は 400～600Hz にピーク周波数をもつ，「あ」に比べ，多少低い周波数成



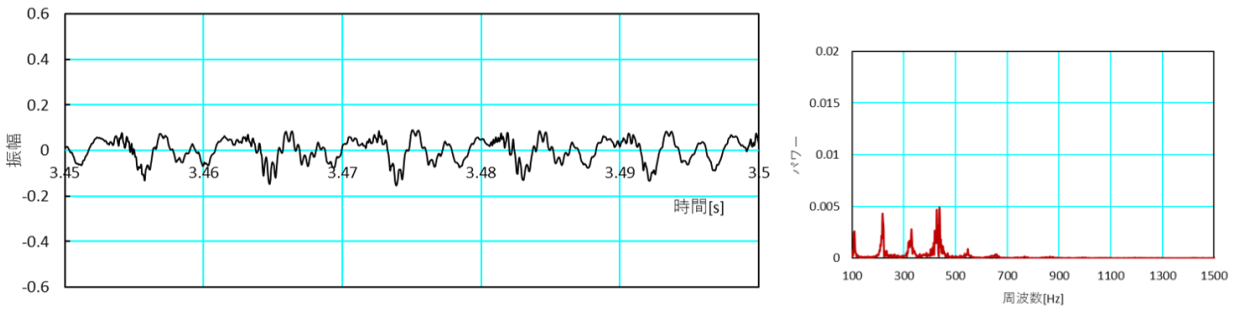
(a)「あ」



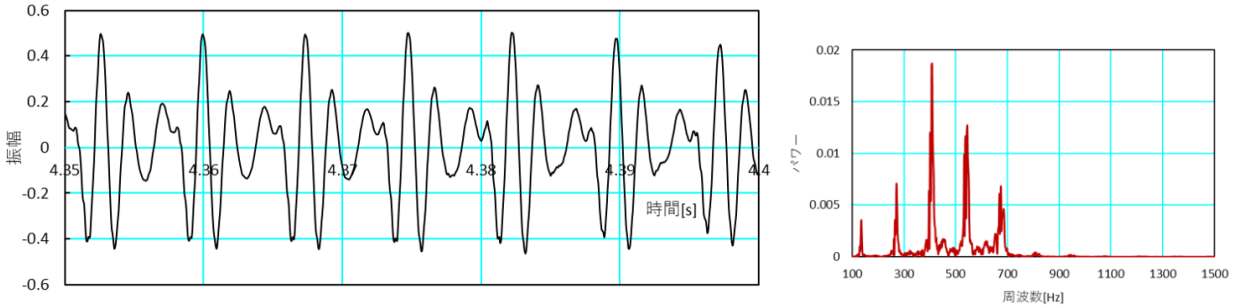
(b)「い」



(c)「う」



(d)「え」



(e)「お」

図 5. 音声信号(一部拡大)および周波数解析結果例

分であるが，大きな振動であり，132Hz 程度ごとに出現する様子は同レベルである．また，パワーの分布が比較的，最大ピーク周波数に対し前後対称の山型となる傾向も見て取れる．

以上，共通的にみられる固有振動の繰り返しは，発声音に対し固有振動に多少の違いがあるものの，総じて，発声音が時間的に等間隔で持続するパルス列となっていることを意味している，と言える．

5　補足

音声情報を扱うシステムの構築に際しては，各プログラミング言語において現在は，簡素に，かつスピーディに実装可能なツールがライブラリに提供されている．例を挙げると，①C++言語では「QAudio」，②Python では「pyAudio」，③C#では「NAudio」，などである．

Python 言語の「pyAudio」に焦点を当てた場合，アルゴリズム 14 に示したような PyAudio クラスの open メソッドでは，先述した複数の API 関数の導入とは異なり，デバイスのオープンからデータ集録まで音声入力に必要な一連の手続きを一手に担うことができる．open メソッドにおける設定パラメータ「frames_per_buffer」はバッファごとのデータサイズであり，open メソッドによる戻り値である Stream 型のインスタンス *fp* に対し read メソッドを活用することにより，バッファごとのデータを取得することができる．リスト変数「*Data*」は集録時間分(5秒)のデータを格納している．

アルゴリズム 14．データ集録に関するコード例

```
Chunk=BUFFER_SIZE
au=pyaudio.PyAudio()
fp=au.open(format=pyaudio.paInt16,channels=cha,
           rate=SampleRate,input=True,
           frames_per_buffer=Chunk)
for k in range(0,int(N_BUFFERS))
    data=fp.read(Chunk)
    Data.append(data)
fp.stop_stream()
fp.close()
au.terminate()
```

また，集録データの型変換(バイト型から数値へ)に関わる一連の処理については，本報告のシステム同様(e.g. アルゴリズム 10)，必須事項ではあるが，音声データ(WAVE ファイル)の保存は，表 6 に示したヘッダ情報に必要以上に注意を払うことなく，サンプルレート等，最低限のパラメータ情報をもとに wave モジュール定義の wave クラスにより，簡潔に行える．

6　おわりに

音声認識技術の目覚ましい進展に伴い，音声入出力に関する技術は基礎レベルとし簡素化されている．本報告の API 関数を用いた技術は，工数も多く煩雑であり，いまや"伝統芸"に近い感も否めないが，音声データの構成や処理過程等，詳細なメカニズムを習得できるため，更なる高機能のシステム作りには有益である．「バッファごとのデータサイズ」を例として挙げた場合においても，集録時間分を 1 つのバッファで賄えるよう設定することも可能であるが，本システムのように，小さいサイズのバッファの複数利用は，①バッファごとのデータ保存により，集録時間分の保存に要する処理時間を分割できる，②バッファの容量が上限に達したタイミング(メッセージ「WIN_DATA」の受信)で，特別な処理を行うなどの仕掛け作りができる，といった視点からも有効と思われる．

各プログラミング言語共に，充実したライブラリの積極的な活用により，この種のシステム開発に対する閾(レベル)も低くなっている現状を踏まえ，本報告が興味ある読者への一助となれば幸甚である．

参考文献

1) Windows App Development：mmeapi.h header，https://docs.microsoft.com/en-us/windows/win32/api/mmeapi/
2) ウィンドウメッセージリスト，https://chokuto.ifdef.jp/urawaza/message/index.html
3) 通信用語の基礎知識：WAV，https://www.wdic.org/w/TECH/WAV
4) Windows App Development：winuser.h header，https://docs.microsoft.com/en-us/windows/win32/api/winuser/

5) WinMain 関数からメインウィンドウハンドルを手に入れるまでのおさらい, http://marupeke296.com/WINT_GetWindowHandle.html
6) Qt プログラミング入門, https://densan-labs.net/tech/qt/
7) Kivy Tutorial, https://www.geeksforgeeks.org/kivy-tutorial
8) Windows App Development : Windows Data Types, https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types
9) VB.NET/VB6.0/CLR/C/C++/Win32API 型一覧表, https://www.nda.co.jp/memo/dim.html
10) ctypes の変数についてのメモ, https://qiita.com/maiueo/items/b2093ba78cde988bb111

附録～データ型対応・互換表 [8),9),10)]

実際に利用したデータ型に対する対応・互換性について, 一覧化した. なお, Python 言語については, あくまでも API 関数の適用に際し導入した「ctypes」ライブラリに基づくものとしている.

① 基本データ型

型名	バイト長	C++		Python	C#
		標準	Windows API		
バイト型	1	unsigned char	BYTE	ctypes.c_ubyte	byte
文字型	2	char	CHAR	ctypes.c_char	char
符号付き短整数型	2	short	SHORT	ctypes.c_short	short
符号なし短整数型	2	unsigned short	WORD ATOM	ctypes.c_ushort wintypes.WORD	UInt16
符号付き長整数型	4	int long	INT LONG	ctypes.c_int ctypes.c_long	int
符号なし長整数型	4	unsigned int unsigned long	UINT DWORD	ctypes.c_uint ctypes.c_ulong wintypes.DWORD	UInt32
単精度浮動数点型	4	float		ctypes.c_float	float
倍精度浮動小数点型	8	double		ctypes.c_double	double

②ハンドル, ポインタ

型名	C++		Python	C#	備考
	標準	Windows API			
ハンドル	void *	HANDLE	wintypes.HANDLE	IntPtr	型なしポインタ
ハンドル	void *	HWND	wintypes.HWND	IntPtr	ウィンドウハンドル
ハンドル	void *	HINSTANCE	wintypes.HANDLE	IntPtr	インスタンスハンドル
型なしポインタ	void *	LPVOID	ctypes.c_void_p wintypes.LPVOID	IntPtr	
符号なし長整数型	unsigned long	MMRESULT	ctypes.c_ulong	Int32	音声 API 関数の戻り値
文字型ポインタ	char *	PCHAR LPSTR	wintypes. LPSTR	string	string 型
符号なし長整数型のポインタ	unsigned long *	PDWORD	wintypes.PDWORD	UIntPtr	
定数文字列へのポインタ	const wchar_t *	LPCWSTR	wintypes.LPCWSTR	—	WCHAR 型：16bit ワイド(Unicode)文字型
符号なし int 型へ cast	unsigned int	UINT_PTR WPARAM	wintypes.WPARAM	UIntPtr	メッセージパラメータ(ボタンの識別)
符号付き long 型へ cast	long	LONG_PTR LPARAM	wintypes.LPARAM	IntPtr	メッセージパラメータ
符号なし long 型へ cast	unsigned long	DWORD_PTR ULONG_PTR	ctypes.c_ulong wintypes.DWORD	UIntPtr	

③ 関数例～waveInOpen 関数

	C++	Python	C#
宣言・定義		*info*=winmm.waveInOpen	[DllImport("winmm.dll", CharSet = CharSet.Unicode, SetLastError = true)]
	MMRESULT waveInOpen (	*info*.restype=ctypes.c_uint	public static extern Int32 waveInOpen(
	LPHWAVEIN *phwi*,	*info*.argtypes=(ctypes.POINTER(wintypes.HANDLE),	ref IntPtr *hwi*,
	UINT *DeviceID*,	wintypes.UINT,	Int32 *DeviceID*,
	LPWAVEFORMATEX *pwfx*,	ctypes.POINTER(WAVEFORMATEX),	ref WaveFormatEx *_wfx*,
	DWORD_PTR *wCallback*,	wintypes.DWORD,	IntPtr *dwCallback*,
	DWORD_PTR *dwCallbackInstance*,	wintypes.DWORD,	Int32 *dwCallbackInstance*,
	DWORD *dwOpen*);	wintypes.DWORD)	Int32 *fdwOpen*);
導入・呼び出し	UINT *resp*=waveInOpen(	*ret=info*(	int ret = NativeMethods.waveInOpen(
	&*hwi*,	ctypes.pointer(*hwi*),	ref *hwi*,
	DeviceID,	wintypes.UINT(*DeviceID*),	*DeviceID*,
	&*WFE*,	ctypes.pointer(*WFE*),	ref *WFE*,
	handle,	wintypes.DWORD(*handle*),	*handle*,
	0,	wintypes.DWORD(0),	0,
	(DWORD) CALLBACK_WINDOW);	wintypes.DWORD (CALLBACK_WINDOW))	NativeMethods.CALLBACK_WINDOW);

④ 構造体例～WAVEHDR

C++	Python	C#
		[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
typedef struct WAVEHDR_tag{	class WAVEHDR(ctypes.Structure):	public struct WaveHdr {
LPSTR *lpData*;	_fields_=[("*lpData*",wintypes.LPSTR),	public IntPtr *lpData*;
DWORD *dwBufferLength*;	("*dwBufferLength*",wintypes.DWORD),	public Int32 *dwBufferLength*;
DWORD *dwBytesRecorded*;	("*dwBytesRecorded*",wintypes.DWORD),	public Int32 d*wBytesRecorded*;
DWORD_PTR *dwUser*;	("*dwUser*",wintypes.PDWORD),	public IntPtr *dwUser*;
DWORD *dwFlags* ;	("*dwFlags*",wintypes.DWORD),	public Int32 *dwFlags*;
DWORD *dwLoops*;	("*dwLoops*",wintypes.DWORD),	public Int32 *dwLoops*;
struct WAVEHDR_tag **lpNext*;	("*lpNext*",wintypes.LPVOID),	public IntPtr *lpNext*;
DWORD_PTR *reserved*;	("*reserved*",wintypes.PDWORD)	public Int32 *reserved*;
} WAVEHDR, *LPWAVEHDR;	]	}