Vol.2015-GN-93 No.40
Vol.2015-CDS-12 No.40
Vol.2015-DCC-9 No.40
2015/1/27

リアルタイム音声処理を5行で実現するためのライブラリの開発

竹渕 瑛一[†1,a)]　速水 治夫[†2,b)]

概要：一般的にオンボードでマイクやヘッドフォンなどの音声機器を入出力する場合，カーネル・ミキサーと呼ばれる OS に内蔵されたサウンド・システムが用いられる．また，DTM など作曲のための録音・再生環境では，音が遅れて聴こえてくる現象を防ぐため，音声入出力を 10 ミリ秒以下で実現することが望まれている．これを実現する規格として，Steinberg 社によって Audio Stream Input/Output（ASIO）が策定されている．Steinberg 社は ASIO によるオーディオ・アプリケーションなどの開発を支援するため，COM インターフェースの仕様や開発キットを配布している．しかし，現状では ASIO を用いたオーディオ・アプリケーション開発の敷居は高いと言わざるを得ない．本研究では，ASIO の COM インターフェースを最短で 5 行程度で利用可能なライブラリを開発した．本ライブラリと既存ライブラリの実装例を比較すると，Halstead Volume が 2.5〜9.6 倍縮小し，実装時間が 2.4〜13.0 倍短縮された．

1. はじめに

一般的なコンピュータには，オーディオ・インターフェース（以下，Audio I/F）がオンボードで搭載されている．Audio I/F とは，マイクやヘッドフォンなどとコンピュータを繋ぎ，デジタル信号とアナログ信号を相互に変換する機能を持った音響機器の一つである．例えば，マイクであれば入力されたアナログ信号をデジタル信号に変換し，ヘッドフォンであればデジタル信号をアナログ信号に変換する．

オンボードの Audio I/F を利用する場合，特にサウンド・システムを選ばなければカーネル・ミキサー [2] によって，音声の入出力が行われる．一方で，DTM などのような録音・再生環境であれば，Audio Stream Input/Output（以下，ASIO[1]）が用いられる．

カーネル・ミキサーは OS によって制御され，各種アプリケーションから鳴らされる音声を，内部で決められたビット深度やサンプリング周波数にまとめる役割を担っている．また，リサンプリングによって発生するエイリアシングを抑制する機能も持ち合わせている．これらの機能により，様々なアプリケーションから鳴らされた音声を違和感なく聞くことが可能となる．

一方で，カーネル・ミキサーによってリサンプリングされることにより，音質の劣化が現れることがある．また，リサンプリング処理を経ることで，音声が出力されるまでに 400 ミリ秒程度の遅延が見られる．これらの音質の劣化は DTM などのような録音・再生環境であれば致命的である．そこで，音質を劣化させることなく，10 ミリ秒以下の遅延で音声の入出力を可能とする規格が ASIO[11] である．ASIO はカーネル・ミキサーとは異なり，Audio I/F のドライバとアプリケーションを直接通信することが可能である．ASIO は特別な処理を行うことがないため，音質を劣化させることなく，極めて短い遅延で音声の入出力ができる．

ASIO の開発キットは Steinberg 社によって提供されている．しかしながら，ASIO の開発キットを用いたオーディオ・アプリケーションの開発は，敷居が高いため積極的に行われていない．このようなこともあり，DTM 以外において，Audio I/F が使われることは数少ない．

本研究では，ASIO を用いたオーディオ・アプリケーション開発の敷居を下げるため，5 行程度で ASIO を用いることが可能なライブラリを開発した．また，本ライブラリについて定量的に評価するため，既存のライブラリ 3 種について 2 つの実装例で比較を行った．評価指標として，Halstead Volume[13][23]，保守容易性指数 [14][15][16]，実装時間 [19] を用いた．

本論文では，本ライブラリの実装とその評価について述

†1 現在，神奈川工科大学大学院博士前期
Presently with Master's Course of Kanagawa Institute of Technology
†2 現在，神奈川工科大学大学
Presently with Kanagawa Institute of Technology
a) nanashi4129@gmail.com
b) hayami.haruo@gmail.com

べる．第 1 章では本論文の概要，第 2 章では本研究の背景と現状及び研究目的，第 3 章では関連研究，第 4 章では本ライブラリの概要，第 5 章では本ライブラリの評価，第 6 章では評価に関する考察，第 7 章では本論文のむすびについて述べる．

2. 研究の背景と現状

本章では，Audio I/F とリアルタイム音声処理の開発と現状について述べる．

2.1 カーネル・ミキサー

Windows では，Multi Media Extension（以下，MME）や Direct Sound などのドライバ・インターフェースが提供されている．しかし，Windows では，カーネル・ミキサーを経由してそれらのドライバ・インターフェースと通信を行っている [2]．

カーネル・ミキサーとは，各種アプリケーションで鳴らされた音声や，各種マイクや楽器などの音声を，あるビット深度やサンプリング周波数にリサンプリングする役割を担っている．また，カーネル・ミキサーはリサンプリング時に発生するエイリアシングの発生を抑制するための機能を持っているため，アップサンプリングやダウンサンプリングに伴う音質の劣化は，一般的には聞き分けることが難しい．

一方で，カーネル・ミキサーのリサンプリングが原因で元波形に変化が生じる．また，リサンプリング処理は全ての波形に対して行われるため，CPU に対して高い負荷をかけることがある．これにより，音声が実際に記録，もしくは再生されるのにわずかながら遅延が発生する [8]．

この問題は，新しく低遅延なオーディオ・システムを開発したり [9]，Audio I/F と ASIO を組み合わせるなどで解決することができる．一般的には，後者の方法が用いられる．

2.2 Audio Interface

Audio I/F とは，アナログ信号とデジタル信号を相互に変換し，なおかつデジタル信号をコンピュータに取り込むための音響機器の一つである．図 1 は Audio I/F とコンピュータ及び入出力機器の模式図である．

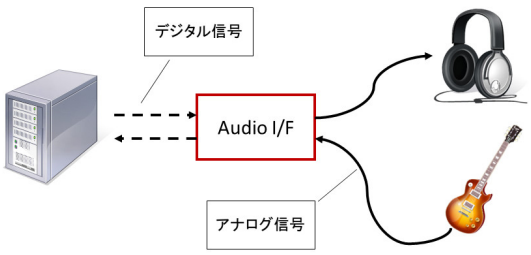

図 1 Audio I/F と各種機材の模式図

図の矢印は入力と出力を表している．図の Audio I/F は，楽器から入力されたアナログ信号をデジタル信号に変換（A/D 変換）し，コンピュータに取り込んでいる．また，デジタル信号をアナログ信号に変換（D/A 変換）し，それを音声として出力している．

Audio I/F は一般的に販売されているマザーボードにも搭載されているが，DTM などで扱われるような専門的な機器と比べると，遅延や A/D 及び D/A 変換の性能は極めて低い．演奏内容をリアルタイムで録音し，リアルタイムで聴音したい場合や，高いビットレートや高ビット深度，正確な A/D 及び D/A 変換を行いたい場合において，DTM 等で扱われる専門的な Audio I/F が必要となる [4][5]．

2.3 Audio Stream Input/Output

ASIO とは，ドイツの Steinberg 社によって規格化されたドライバ・インターフェースの一つである．実質的に，Audio I/F とアプリケーションを繋ぐための標準的な規格である．

ASIO を利用することで，カーネル・ミキサーを経由せずに，直接アプリケーションとオーディオ・ドライバで通信できるようになる．カーネル・ミキサーを経由しないため，遅延が最短 2 ミリ秒程度と極めて小さい．また，リサンプリングをしないため音質の劣化も起こらない．DTM において ASIO は録音や聴音に欠かせない存在である．

2.4 ASIO SDK

ASIO SDK とは，アプリケーション側で ASIO に準拠したオーディオ・ドライバを利用するための開発キットである．Steinberg 社によって提供されている．

アプリケーションとオーディオ・ドライバの通信には，Component Object Model（以下，COM）[6] が用いられる．ASIO SDK では，COM の初期化やドライバの取得，COM インターフェースの提供などを行っている．また，これらの処理をある程度簡略化するための関数なども提供している．

ASIO SDK はオーディオ・ドライバを ASIO という規格上で利用するための開発キットである．しかし，ASIO SDK は事実上オーディオ・ドライバと COM で通信を行うためのサンプルコードである．その理由として，COM の初期化を内部で行っている，C 言語と C++のコードが入り混じっている，グローバル変数が多く宣言されている，それらのグローバル変数において排他制御 [7] が行われていない，列挙型ではなくマクロ定義を多用している，COM がクラッシュするリスクを利用者に転嫁している点などが挙げられる．また，Visual C++ 6.0 のみ対応しているため，ASIO SDK 本体の修正やプロジェクトの設定を試行錯誤する必要があり，導入直後でもビルドすることが難しい．ASIO SDK を利用する場合，予期せぬバグや ASIO SDK

本体の修正が必要である．

これらの問題があるため，ASIO SDK を直接利用することは少ない．開発では主に，ASIO ドライバから提供されている COM インターフェースのみを利用して開発する．そのため，ASIO での開発を行う際，COM の開発経験や OS の知識などが必要となり，開発者に要求されるスキルが多くなる問題がある．

2.5 研究目的

ASIO SDK における問題点として，開発の敷居が高いことについて述べた．ASIO の開発の敷居が高いままであると，特にゲームやインタラクション分野においてリアルタイム音声処理が用いられることは望めない．

本研究の目的は，ASIO を用いたリアルタイム音声処理の敷居を下げることである．ASIO を用いた開発の敷居を下げることで，楽器やマイク等のインターフェースを用いたアプリケーションの普及・発展を後押しする．

3. 関連研究

本章では，関連研究となる RtAudio 及び PortAudio について述べる．

3.1 RtAudio

RtAudio[3] とは，Gary P. Scavone によって開発された，クロスプラットフォームで利用可能な，音声入出力ストリーミングライブラリである．C++で開発され，オブジェクト指向に対応している．対応しているオーディオ・システムとして，Windows の Direct Sound や Linux の ALSA API，OS-X の Core Audio 等が挙げられる．また，ASIO の利用も可能である．

RtAudio の特徴として，各種プラットフォームとオーディオ・システムの差異を吸収できる点が挙げられる．通常，クラスプラットフォーム対応の音声処理を記述する場合，プラットフォームごとかつ，オーディオ・システムごとに実装しなければならない．例えば，Windows では Direct Sound の他に MME なども存在する．これらを全て記述するのは大変手間がかかり，冗長である．RtAudio はこれらの実装の手間を削減することが可能である．

3.2 PortAudio

PortAudio[10] とは，Ross Bencina らによって開発された，クラスプラットフォームで利用可能な，音声入出力ストリーミングライブラリである．C 言語で開発されている．Windows MME，Direct Sound，Core Audio，UNIX 及び Linux で利用可能である．また，ASIO や EASI，独自で開発したサウンド・システムにも対応している．

PortAudio は RtAudio と同様に，オーディオ・システムの差異を吸収するために開発されたライブラリである．ただし，C 言語で書かれているため，RtAudio と比べると，C 言語の使用経験が要求される．

4. 開発システム

本研究では，ASIO の COM インターフェースを利用し，新しい ASIO ライブラリの開発を行った．本章では開発システムである TinyASIO[12] について述べる．

4.1 設計方針

第 2 章にて，ASIO を用いた音声入出力の開発は容易でないことについて述べた．関連研究である RtAudio と PortAudio は，多種多様なプラットフォームとサウンド・システムに対応していることもあり，網羅性や確実性が高い一方で簡潔さに欠けている．

本研究では，ASIO を用いたリアルタイム音声入出力の開発の敷居を低くすることを目的としている．従って，利用者が扱うソースコードは短く簡潔であるべきである．また，システムを利用するために必要な変数，引数，関数の数も極力抑えることで，扱いやすさを向上させる．

具体的な方法として，4 種類挙げられる．

ASIO ではサンプリング周波数やバッファサイズを指定することができる．1 つ目として，これらの設定を ASIO ドライバ側に任せることで API を簡潔にする．

2 つ目として，ASIO のサンプルデータ型は 21 種類存在するが本ライブラリは 32 ビット符号付き整数型のみサポートを行う．これは ASIO ドライバが 21 種類のサンプルデータ型全てに対応しているとは限らず，対応していない場合は利用者の環境においてバグが発生することが予想できるからである．32 ビット符号付き整数型は数多くの ASIO ドライバでサポートしていることを考慮すると，21 種類のサンプルデータ型すべてに対応する必然性はあまりないものと考えられる．

3 つ目として，ASIO では 4 種類のコールバック関数を登録する必要がある．その中で利用されるのは主に 1 種類だけである．利用者はこの 1 種類のコールバック関数のみ実装を行うことで，利用頻度の少ない 3 種類のコールバック関数の実装を省く．

4 つ目として，覚える要素を極力減らすため，多くの部分をブラックボックス化することである．利用者にとって最も関心のあることは，正常に指定したチャンネルで音声信号を入出力できるかどうかである．その点を考慮し，利用者にとって興味のない部分に関してはインターフェースを提供しない方針とした．例えば，アプリケーション終了時における解放処理などである．C++においては，基幹となるクラスのデストラクタにて解放処理を行えば，改めて利用者が開放処理を記述する必要がなくなる．

これらを念頭にライブラリの実装を行うことで，利用者に対して簡潔で使いやすいインターフェースを提供するこ

とが可能になると考えられる．

4.2 動作環境

本ライブラリの動作環境を，表 1 に示す．

表 1 TinyASIO の動作環境

OS	Windows 7 64bit
開発環境	Visual Studio 2013
コンパイラ	Visual C++ 12.0

また，利用可能な ASIO ドライバの設定及び Audio I/F の仕様を表 2 に示す．

表 2 TinyASIO で利用可能な ASIO ドライバの設定及び Audio I/F の仕様

接続方法	USB2.0
サンプリング周波数	ASIO ドライバの設定に依存
バッファサイズ	ASIO ドライバの設定に依存
サンプルデータ型	32 ビット符号付き整数型

4.3 クラス設計の概要

本ライブラリのクラス設計を図 2 に示す．

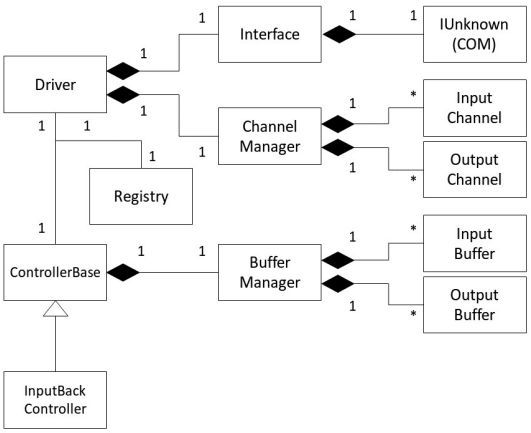

図 2 TinyASIO のクラス設計

本ライブラリでは, 主に本ライブラリの利用者が意識的に用いる部分として，おおまかに Driver クラスと Controller クラスに分けられている．

Driver クラスでは，COM インターフェースの初期化や，入出力チャンネルの管理などが行われる．Controller クラスとは，バッファリングを管理するためのクラス群である．なお，Controller クラスは，ControllerBase クラスを含む，ControllerBase クラスを継承したクラスのことを指す．従って，本論文では，InputBackController クラスを Controller クラスと呼ぶことがある．

ChannelManager クラスとは，Audio I/F に存在する入出力チャンネルの情報を管理するためのクラスである．Driver クラスによって所有され，Driver クラスが初期化された後に生成されるクラスである．主に Controller クラスを初期化する際に用いられる．

BufferManager クラスは，BufferManager クラスとは，チャンネルごとにバッファリングを行い，管理するためのクラスである．Contreller クラスによって所有される．Controller クラスの拡張を行う場合を除いて，BufferManager インスタンスは Controller クラスにおいてカプセル化される．

Registry クラスとは，COM インターフェースの CUID を取得するために用いられるユーティリティ・クラスである．

図 3 は本ライブラリの利用者，レジストリ，デバイスと，各種クラスとの関係性を示す図である．この図では，ユースケースとアクターとの強い関連を示す図である．矢印は依存関係を表している．

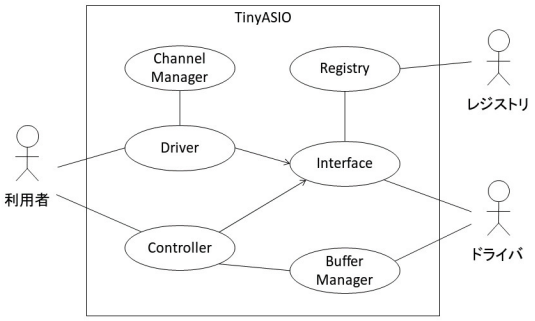

図 3 TinyASIO の利用者，レジストリ，ASIO ドライバと各種クラスとの関係

実質的に利用者が関わるのは Driver クラスと Controller クラスのみである．一方で，レジストリと直接的な関連があるのは Registry クラスのみである．また，Interface クラスと BufferManager クラスでは，COM インターフェースの呼び出しや，非同期処理などの関係で ASIO ドライバと関連がある．ChannelManager クラスが利用者との関連がないのは，Driver クラスから利用者に向けて提供されており，利用者から生成できない仕組みとなっているからである．

4.4 開発システムの流れ

本節では，本ライブラリの処理の流れや利用方法などについて述べる．

4.4.1 基本的な利用方法

本ライブラリの大まかな流れは以下の通りである．

(1) ASIO ドライバの初期化
(2) コントローラの生成
(3) バッファリングの開始
(4) バッファ内のストリームから値を取得，もしくは蓄積
(5) バッファリングの停止

この流れに沿ったソースコードを以下に示す．このソー

スコードは，ドライバの初期化からバッファリングの停止までを行っている．

```
1  #include <Windows.h>
2  #include <TinyASIO.hpp>
3
4  int main() {
5    // レジストリにあるドライバ名を指定
6    asio::Driver::Init("AudioBox");
7
8    asio::InputBackController controller;
9    controller.Start(); // バッファリング開始
10   while (true)
11   {
12     // ここで蓄積された音声を取得できる
13     auto fetchedWaves = controller.Fetch();
14   }
15   controller.Stop(); // バッファリング停止
16   return 0;
17 }
```

6 行目に ASIO ドライバの初期化を行っている．Driver::Init 関数では，文字列で ASIO ドライバの名前を指定することが可能である．ドライバ名があらかじめわかっている場合の開発を簡略化するためである．

8 行目では Controller クラスの生成を行っている．ここでは，本ライブラリで用意されている InputBackController クラスを用いている．InputBackController クラスでは，入力信号を出力信号にそのまま流し，入力信号を入力バッファに蓄積するだけのクラスである．このサンプルでは，0 番の入力チャンネルと 0 番の出力チャンネルを用いている．

9 行目でバッファリングを開始している．

13 行目でバッファリングされた入力信号を取り出し，fetchedWaves 変数に格納している．Fetch 関数は，StreamingVector 型のオブジェクトを返す．

15 行目でバッファリングを停止している．

なお，Driver クラスや Controller クラスは，デストラクタでバッファや COM インターフェース等の解放を自動的に行なっている．従って，本ライブラリではアプリケーション終了時に解放処理を行う必要がない．

4.4.2 レジストリの取得，変更

Registry クラスでは，文字列から ASIO ドライバの CUID を取得したり，レジストリにおけるサブキーの取得などを行える．Registry クラスを用いることで，たとえ未知の環境でも，インストールされている ASIO ドライバの列挙が可能となる．

また，COM における ThreadingModel[21] を修正する関数も用意されている．万が一，対象の ASIO ドライバがシングルスレッドで動作していても，その関数を利用することで，瞬時にマルチスレッドに対応することが可能である．

インストールされている ASIO ドライバのレジストリのサブキーは，以下のソースコードで取得できる．

```
1  auto pathes = asio::Registry::GetDriverPathes();
2  asio::Driver(pathes.Items(0));
```

このソースコードでは，インストールされている ASIO ドライバのサブキーを取得し，それを用いて ASIO ドライバの初期化を行っている．

また，COM が COINIT_APARTMENTTHREADED ではなく，COINIT_MULTITHREADED によって初期化されていて，かつ ThreadingModel のレジストリ値が Apartment だった場合，COM は基本的に初期化に失敗する仕組みとなっている．

ThreadingModel が原因で初期化に失敗する事象を防ぐため，Registry クラスには対象の ASIO ドライバの ThreadingModel を変更するための関数が用意されている．以下のソースコードは，ThreadingModel を変更しているコードである．

```
1  asio::Registry::ChangeThreadingModel(
       ThreadingModel::Both);
2  asio::Registry::ChangeThreadingModel(
       ThreadingModel::Apartment);
3  asio::Registry::ChangeThreadingModel(
       ThreadingModel::Free);
```

Apartment はシングルスレッドの設定である．Free はマルチスレッドの設定である．Both は両方で動作する設定である．もし，マルチスレッドとシングルスレッドで共存する可能性がある場合は，Both を設定する．

4.4.3 チャンネルの取得

指定した Audio I/F に搭載されているチャンネルは，Driver インスタンスから取得することができる．個別のチャンネルについては，Inputs 関数及び Outputs 関数で取得することができる．これらの関数が返却するのは，std::vector 型の参照であるため，通常の配列と同様に扱うことができる．

以下のソースコードは初期化した ASIO ドライバから，ChannelManager を取得するためのコードである．

```
1  auto& driver = asio::Driver::Init("AudioBox");
2  auto& ch_mng = driver.ChannelManager();
3
4  // 入力チャンネル配列の取得
5  auto& inputs = ch_mng.Inputs();
6  // 出力チャンネル配列の取得
7  auto& outputs = ch_mng.Outputs();
```

4.4.4 バッファリング

図 4 は，ASIO におけるバッファと，本ライブラリの Controller クラスとの関係を示した図である．

ASIO では，Audio I/F の入出力チャンネルごとに，音声信号を蓄積するためのバッファが 2 つずつ用意される．入力チャンネルの場合は，ASIO ドライバがバッファに音

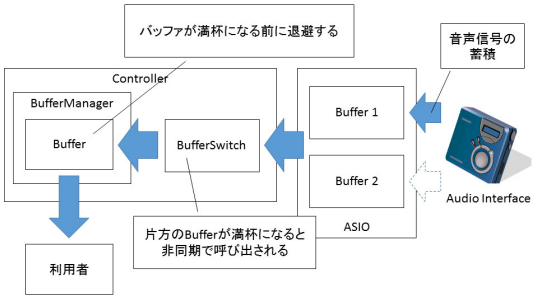

図 4 Controller と ASIO のバッファの関係図

声信号を書き込む．出力チャンネルの場合は，利用者がそのバッファに音声信号を書き込む．

ASIO ドライバは自動的にバッファの読み取り，もしくは書き込みを行っている．片方のバッファに対して読み書きが終了すれば，もう片方のバッファに対して読み書きを行う仕組みとなっている．読み書きが終了すると，ASIO ドライバが自動的に BufferSwitch 関数を呼び出す．BufferSwitch 関数が呼び出されると，即座にもう片方のバッファに対して読み書きを行う．

このとき，もう片方のバッファの読み書きが終わるまでに僅かな隙が生じる．利用者は，そのバッファの読み書きが終わるまでに，BufferSwitch 関数内で音声信号の読み書きを行う．

ASIO はこの仕組みにより，音が途切れずに音声信号の入出力を行うことができる．なお，ASIO では，この仕組を Double Buffering[22] と呼んでいる．

4.4.5 Controller の拡張

Controller クラスの拡張は，ControllerBase クラスを継承し，コールバック関数を実装することで解決することができる．以下のソースコードは，ControllerBase クラスを継承した場合のサンプルコードである．入力チャンネルから受け取った音声信号を，出力チャンネルにコピーしている．

```
1  #include <TinyASIO.hpp>
2
3  class OriginController : public ControllerBase
4  {
5    static InputBuffer* input;
6
7    static void BufferSwitch(long index, long
         directProcess)
8    {
9      void* outBuf = output->GetBuffer(index);
10     void* inBuf = input->GetBuffer(index);
11
12     memcpy(outBuf, inBuf, BufferSize());
13
14     input->Store(inBuf, BufferLength());
15   }
16
17 public:
18   OriginController() : ControllerBase()
19   {
20     CreateBuffer({
21       channelManager->Inputs(0),
22       channelManager->Outputs(0)}, &BufferSwitch
             );
23
24     // 入力チャンネル 0 => 入力バッファ 0
25     input = &bufferManager->Inputs(0);
26   }
27
28   StreamingVector Fetch()
29   {
30     return input->Fetch();
31   }
32 };
33
34 InputBuffer* OriginController::input = nullptr;
```

BufferSwitch 関数は，ASIO ドライバがバッファの読み書きが終了したときに呼び出すための関数である．input 変数は，ASIO ドライバが書き込んだバッファを蓄積するための変数である．

outBuf 及び inBuf は，ASIO ドライバがバッファリング対象のメモリ空間のアドレスである．inBuf には，入力チャンネルから音声信号が書き込まれる．outBuf には出力チャンネルに出力したい音声信号を書き込む．それぞれ，BufferSize 関数の戻り値と同じだけの領域が確保されている．

Buffer クラスの Store 関数は，引数で与えられたバッファを，Buffer クラス内部のバッファに蓄積するための関数である．これは，Buffer クラスの Fetch 関数が呼び出されるまで蓄積され続ける．

Buffer クラスの Fetch 関数は，Buffer クラス内部で蓄積されたバッファを返すための関数である．なお，Fetch 関数が呼び出された時，Buffer クラス内部のバッファはクリアされる．

Buffer クラスの Store 関数及び Fetch 関数では，マルチスレッドのために排他制御が行われている．そのため，OriginalController クラスにおける BufferSwitch 関数と Fetch 関数が別スレッドで動作していたとしても，デッドロックなどによってアプリケーションがクラッシュする心配がない．

5. ライブラリの評価

本章では，本ライブラリの評価について述べる．本章では，ASIO SDK 及び RtAudio，PortAudio の 3 種類のライブラリと，本ライブラリのソースコードを比較した．比較方法としては，各種ライブラリにおいて実装例を 2 つ用意し，コード・メトリクス [18] において代表的な Halstead Volume，保守容易性指数，実装時間を求め，比較を行った．

5.1 実装例の比較

各ライブラリと本ライブラリについて，実装例を 2 つ用意することで，それぞれのライブラリで必要最小限のソースコードの量と，利用者にとってそれぞれ理解し，保守するのに必要な労力を定量化した．本節では，その結果について述べる．

5.1.1 比較方法

実装例の比較方法として，各ライブラリと本ライブラリについて，必要最小限の実装例を 2 つ用意し，それぞれステップ数，保守容易性指数 [14][15]，Halstead Volume[13][23]，実装時間について比較する．

Halstead Volume とは，プログラムの体積を表すための指標である [20]．Halstead Volume は，演算子や予約語，変数や関数の数に影響される指標であるため，この数値が高いほどソースコードの分量は増大する．Halstead Volume を式 1 で表す．使用されている演算子や予約語などの種類は η_1，その合計の数は N_1，使用されている関数名や変数名などの数は η_2，その合計の数は N_2 である．

$$V = N_1 \log_2 \eta_1 + N_2 \log_2 \eta_2 \tag{1}$$

保守容易性指数とは，Halstead Volume 及び循環的複雑度 [17] からなる，ソースコードの保守のしやすさを表す指標である．保守容易性指数は 0 から 100 までの値を取り，値が高いほど，保守がしやすく，読みやすいコードである．20 以下で保守が困難なソースコードとなり，10 以下でリファクタリングをする必要がある．保守容易性指数を式 2 に示す [16]．LoC とは，ソースコードの行数である．CP とは，コード行数に対するコメントの割合である．

$$\begin{aligned} I = {} & \max(0, 171 - 5.2 \ln(V) - 0.23M \\ & -16.2 \ln(LoC) + 50.0 \sin \sqrt{2.46CP} \end{aligned} \tag{2}$$

実装時間 [19] とは，ソースコードを実装もしくは理解するために必要な時間を表した値である．単位は秒である．この数値が高ければ高いほど，ソースコードを理解，実装するために時間を要する．実装時間を式 4 に示す．E は実装に必要な努力値である．

$$E = \frac{\eta_1 N_2 V}{2\eta_2} \tag{3}$$

$$T = \frac{E}{18} \tag{4}$$

実装例 A は，入力バッファから受け取った音声信号を，出力バッファに書き込み，その音声を出力するソースコードである．

実装例 B は，入力バッファから受け取った音声信号をランダムに変化させるソースコードである．

なお，エラー処理に関しては，それぞれ冗長であるため行わないものとする．

5.1.2 比較結果

実装例の比較結果を表 3 及び表 4 に示す．

表 3 実装例 A の比較

比較項目	A	B	C	X
Statements	63	58	73	8
Halstead Volume	2876.94	2951.0	3206.0	297.25
Maintainability	59.2	11.07	66.87	100.0
Time to Implement	1917	4918	3116	140

表 4 実装例 B の比較

比較項目	A	B	C	X
Statements	63	57	71	24
Halstead Volume	2892.08	2871.0	3164.0	1123.97
Maintainability	59.13	14.75	68.86	75.63
Time to Implement	2169	4784	3076	874

表中の Statements の単位はステップ数である．また，Time to Implement の単位は秒である．

表 3 及び表 4 の比較対象の A は ASIO SDK，B は RtAudio，C は PortAudio，X は本ライブラリである．

6. ライブラリの評価に関する考察

実装例 A の Halstead Volume において，他のライブラリと本ライブラリを比べると，最大で約 9.6 倍の差が開いている．他のライブラリでは，初期化やバッファリングの準備に関して，多くの行数を割いているため，Halstead Volume の値が大きくなっている．その点を考慮すると，本ライブラリは他のライブラリと比べ，実装に必要な演算子や予約語，変数名や関数名などが少なくなるように設計及び実装されていると考えられる．

一方で，本ライブラリの実装例 B については，Controller を自作する必要があった．その拡張部分について行数が増加していることから，本ライブラリの Halstead Volume も相対的に増加している．しかし，他のライブラリと比べて最大で約 2.5 倍の開きがあることから，拡張部分を比べても，本ライブラリのほうが簡潔であることを示している．

保守容易性指数に関しては，本ライブラリの実装例 A は 100.0 であるため，保守する点に関しては問題にならない．実装例 B では本ライブラリの保守容易性指数が低下しているが，保守が困難である 20 以上を保っているため，保守は困難ではないことが示されている．

これまで述べた Halstead Volume 及び保守容易性指数における本ライブラリの評価も，実装時間の長さにおいて裏付けられている．実装例 A では，本ライブラリを実装及び理解するのにかかる時間は，およそ 2 分半程度であるが，他のライブラリは 30 分を超えている．最も長いものであれば，RtAudio が 1 時間以上かかることが示されている．また，実装例 B についても，本ライブラリは 10 分程度で

理解及び実装できることが示されている．他のライブラリは実装例 A と同様に，理解及び実装に 30 分以上かかることが示されている．

Halstead Volume 及び保守容易性指数を総合的に見ると，本ライブラリの API は，他のライブラリの API と比べると理解しやすく，実装もしやすいことが定量的に示された．本ライブラリの設計方針は API の実装として正しかったものと考えられる．

7. おわりに

本論文では，本研究で実装した TinyASIO の設計とその評価について述べた．

第 5 章において，本ライブラリと既存のライブラリについて，コードメトリクスによるライブラリ実装と実装例について定量的な評価を行った．最も簡単な実装例では，コード量が 9.6 倍縮小し，実装時間が 13.0 倍短縮された．また，本ライブラリを拡張する場合，同等の機能を既存ライブラリで実装した内容と比較すると，コード量が 2.5 倍縮小され，実装時間は 2.4 倍短縮された．

今後の課題として，本ライブラリの動作する環境を網羅することである．現状で，OS は Windows7 64bit，Audio I/F は Audio Box 及び ASIO4ALL のみテストに成功している．Windows8 や，他の Audio I/F でも動作できるように，ユーザテストなどでログの可視化を行いながら拡張を続けていく方針である．

謝辞 本ライブラリの動作実験において，神奈川工科大学情報メディア学科徳弘研究室にご協力を賜った．感謝する．

本ライブラリの使用方法等について Siv3D 勉強会及び未踏交流会 Vol.15 にて発表を行った．発表の場を賜った Siv3D の開発者である鈴木遼氏及び未踏関係者に感謝する．

本論文について速水研究室のメンバー数名に校正ミスの指摘を頂いた．深く感謝する．

参考文献

[1] Jonas Ekeroot and Jan Berg.: *Audio software development - an audio quality perspective.*, Audio Engineering Society, Convention Paper 7438, 2008.
[2] Michael B. Jones, John Regehr and Stefan Saroiu.: *Two Case Studies in Predictable Application Scheduling Using RialtoNT.*, Seventh Real-Time Technology and Applications Symposium (RTAS 2001), pp.157-164, 2001.
[3] Gary P. Scavone.: *RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output.*, International Computer Music Conference 2002 (ICMC 2002), pp. 196-199, 2002.
[4] 赤堀肇, 石川智治, 小林幸夫, 宮原誠：ディジタル・オーディオ・インタフェース (AES/EBU) の jitter と音質の関係, 電子情報通信学会技術研究報告. EA, 応用音響 99(260), 1-8, 1999-08-27.
[5] 西村 明，小泉 宣夫：ディジタル・オーディオ機器におけるサンプリング・ジッターの諸様相とその要因, 東京情報大学研究論集 7(2). 79-92, 2004-02-20.
[6] Williams, Sara, and Charlie Kindel.: *The component object model: A technical overview.*, Microsoft Technical Report, 1994.
[7] Dalessandro, Luke, et al.: *Transactional mutex locks.*, Euro-Par 2010-Parallel Processing, pp. 2-13, 2010.
[8] 長嶋洋一, 中村文隆：メディアアートにおける画像系の制御について，情報処理学会研究報告. 2000.76 (2000)，pp. 31-36，2000.
[9] Juillerat, Nicolas, Stefan Mller Arisona, and Simon Schubiger-Banz.: *Real-time, low latency audio processing in Java.*, Proceedings of the international computer music conference, Immerced Music, vol., pp. 99-102, 2007.
[10] Bencina, Ross, and Phil Burk.: *PortAudioan open source cross platform audio API.*, Proceedings of the International Computer Music Conference, pp. 263-266, 2001.
[11] 高澤嘉光，徳弘一路：ASIO(Audio Stream Input Output) の有効性についての考察，電子情報通信学会技術研究報告. EA, 応用音響 108(115), pp. 31-36, 2008-06-20.
[12] 竹 渕 瑛 一：TinyASIO(online)，入 手 先 ⟨https://github.com/GRGSIBERIA/TinyASIO⟩ (2014.12.13).
[13] Halstead M. H.: *Elements of Software Science.*, Operating, and Programming Systems Series Volume 7, Elsevier, 1977.
[14] Land, Rikard.: *Measurements of software maintainability.*, Proceedings of ARTES Graduate Student Conference, ARTES. pp. 1-7, 2002.
[15] Heitlager, Ilja, Tobias Kuipers, and Joost Visser.: *A practical model for measuring maintainability.*, Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the. IEEE, pp. 30-39, 2007.
[16] Coleman D, Ash D, Lowther B and Oman P.: *Using Metrics to Evaluate Software System Maintainability.* , Computer, IEEE, Vol. 27, Issue. 8, pp. 44-49, 1994.
[17] McCabe, Thomas J.: *A complexity measure.*, Software Engineering, IEEE Transactions on 4, pp. 308-320, 1976.
[18] Jiang, Yue, et al.: *Comparing design and code metrics for software quality prediction.*, Proceedings of the 4th international workshop on Predictor models in software engineering. ACM, pp. 11-18, 2008.
[19] Halstead, Maurice H.: *Toward a theoretical basis for estimating programming effort.*, Proceedings of the 1975 annual conference. ACM, pp 222-224, 1975.
[20] GRGSIBERIA.: halstead.rb(online), available from ⟨https://gist.github.com/GRGSIBERIA/a7c35bb69d32a506ef44⟩(2014.12.21).
[21] Wang, Yi-Min, and P-YE Chung.: *Customization of distributed systems using COM.* Concurrency, IEEE, Vol. 6, No. 3, pp. 8-12, 1998.
[22] Chaudhary, Amar, Adrian Freed, and Matthew Wright.: *An open architecture for real-time audio processing software.* Audio Engineering Society Convention 107. Audio Engineering Society, 1999.
[23] Samoladas, Ioannis, et al.: *Open source software development should strive for even greater code maintainability.* Communications of the ACM, Vol. 47, No. 10, pp. 83-87, 2004.