サウンドエフェクタ
ソフトウェア編(2)
～入力信号を加工して出力してみる～
取扱説明書

-本書を必ずよく読み、ご理解された上でご利用ください

株式会社北斗電子
REV.1.0.0.0

ー目　次ー

注意事項

本書を必ずよく読み、ご理解された上でご利用ください

【ご利用にあたって】

1. 本製品をご利用になる前には必ず取扱説明書をよく読んで下さい。また、本書は必ず保管し、使用上不明な点がある場合は再読し、よく理解して使用して下さい。
2. 本書は株式会社北斗電子製マイコンボードの使用方法について説明するものであり、ユーザシステムは対象ではありません。
3. 本書及び製品は著作権及び工業所有権によって保護されており、全ての権利は弊社に帰属します。本書の無断複写・複製・転載はできません。
4. 弊社のマイコンボードの仕様は全て使用しているマイコンの仕様に準じております。マイコンの仕様に関しましては製造元にお問い合わせ下さい。弊社製品のデザイン・機能・仕様は性能や安全性の向上を目的に、予告無しに変更することがあります。また価格を変更する場合や本書の図は実物と異なる場合もありますので、御了承下さい。
5. 本製品のご使用にあたっては、十分に評価の上ご使用下さい。
6. 未実装の部品に関してはサポート対象外です。お客様の責任においてご使用下さい。

【限定保証】

1. 弊社は本製品が頒布されているご利用条件に従って製造されたもので、本書に記載された動作を保証致します。
2. 本製品の保証期間は購入戴いた日から1年間です。

【保証規定】

保証期間内でも次のような場合は保証対象外となり有料修理となります

1. 火災・地震・第三者による行為その他の事故により本製品に不具合が生じた場合
2. お客様の故意・過失・誤用・異常な条件でのご利用で本製品に不具合が生じた場合
3. 本製品及び付属品のご利用方法に起因した損害が発生した場合
4. お客様によって本製品及び付属品へ改造・修理がなされた場合

【免責事項】

弊社は特定の目的･用途に関する保証や特許権侵害に対する保証等、本保証条件以外のものは明示・黙示に拘わらず一切の保証は致し兼ねます。また、直接的・間接的損害金もしくは欠陥製品や製品の使用方法に起因する損失金・費用には一切責任を負いません。損害の発生についてあらかじめ知らされていた場合でも保証は致し兼ねます。
ただし、明示的に保証責任または担保責任を負う場合でも、その理由のいかんを問わず、累積的な損害賠償責任は、弊社が受領した対価を上限とします。本製品は「現状」で販売されているものであり、使用に際してはお客様がその結果に一切の責任を負うものとします。弊社は使用または使用不能から生ずる損害に関して一切責任を負いません。
保証は最初の購入者であるお客様ご本人にのみ適用され、お客様が転売された第三者には適用されません。よって転売による第三者またはその為になすお客様からのいかなる請求についても責任を負いません。
本製品を使った二次製品の保証は致し兼ねます。

安全上のご注意

製品を安全にお使いいただくための項目を次のように記載しています。絵表示の意味をよく理解した上でお読み下さい。

表記の意味

警告 取扱を誤った場合、人が死亡または重傷を負う危険が切迫して生じる可能性がある事が想定される

注意 取扱を誤った場合、人が軽傷を負う可能性又は、物的損害のみを引き起こすが可能性がある事が想定される

絵記号の意味

	**一般指示** 使用者に対して指示に基づく行為を強制するものを示します		**一般禁止** 一般的な禁止事項を示します
	**電源プラグを抜く** 使用者に対して電源プラグをコンセントから抜くように指示します		**一般注意** 一般的な注意を示しています

警告

以下の警告に反する操作をされた場合、本製品及びユーザシステムの破壊・発煙・発火の危険があります。マイコン内蔵プログラムを破壊する場合もあります。

1. 本製品及びユーザシステムに電源が入ったままケーブルの抜き差しを行わないでください。
2. 本製品及びユーザシステムに電源が入ったままで、ユーザシステム上に実装されたマイコンまたはIC等の抜き差しを行わないでください。
3. 本製品及びユーザシステムは規定の電圧範囲でご利用ください。
4. 本製品及びユーザシステムは、コネクタのピン番号及びユーザシステム上のマイコンとの接続を確認の上正しく扱ってください。

発煙・異音・異臭にお気付きの際はすぐに使用を中止してください。

電源がある場合は電源を切って、コンセントから電源プラグを抜いてください。そのままご使用すると火災や感電の原因になります。

⚠注意

以下のことをされると故障の原因となる場合があります。

1. 静電気が流れ、部品が破壊される恐れがありますので、ボード製品のコネクタ部分や部品面には直接手を触れないでください。
2. 次の様な場所での使用、保管をしないでください。

   ホコリが多い場所、長時間直射日光があたる場所、不安定な場所、衝撃や振動が加わる場所、落下の可能性がある場所、水分や湿気の多い場所、磁気を発するものの近く
3. 落としたり、衝撃を与えたり、重いものを乗せないでください。
4. 製品の上に水などの液体や、クリップなどの金属を置かないでください。
5. 製品の傍で飲食や喫煙をしないでください。

ボード製品では、裏面にハンダ付けの跡があり、尖っている場合があります。

取り付け、取り外しの際は製品の両端を持ってください。裏面のハンダ付け跡で、誤って手など怪我をする場合があります。

CD メディア、フロッピーディスク付属の製品では、故障に備えてバックアップ(複製)をお取りください。

製品をご使用中にデータなどが消失した場合、データなどの保証は一切致しかねます。

アクセスランプがある製品では、アクセスランプ点灯中に電源の切断を行わないでください。

製品の故障の原因や、データの消失の恐れがあります。

本製品は、医療、航空宇宙、原子力、輸送などの人命に関わる機器やシステム及び高度な信頼性を必要とする設備や機器などに用いられる事を目的として、設計及び製造されておりません。

医療、航空宇宙、原子力、輸送などの設備や機器、システムなどに本製品を使用され、本製品の故障により、人身や火災事故、社会的な損害などが生じても、弊社では責任を負いかねます。お客様ご自身にて対策を期されるようご注意ください。

概要

当社製品、サウンドエフェクタ向けのプログラムを作成する際の手順、注意点等に関して記載を行っている資料となります。製品のハードウェアに関しては、「サウンドエフェクタ　取扱説明書」に記載がありますので、そちらも合わせて参照頂きたく。

個別のプログラムに関しては、プログラム毎のドキュメント（アプリケーションノート）を参照ください。

ソフトウェア編の資料は、

・サウンドエフェクタ　ソフトウェア編(1)　取扱説明書　～波形を生成してみる～
・サウンドエフェクタ　ソフトウェア編(2)　取扱説明書[本書]　～入力信号を加工して出力してみる～
・サウンドエフェクタ　ソフトウェア編(3)　取扱説明書　～FFT/逆 FFT をライブラリ関数で処理してみる～
・サウンドエフェクタ　ソフトウェア編(4)　取扱説明書　～デバッグ、ゼロからプログラムを作成する場合に～

に分かれています。

ベースとなるテンプレートは、

RX65_SOUND_EFFECTOR_TEMPLATE

です。このテンプレート（ベースとなるプロジェクト）を基に、サウンドエフェクタのプログラムを作成する上で、共通となる部分に関して説明します。

参考文献

プログラムの音声処理の部分に関しては、

C 言語ではじめる音のプログラミング　青木　直史著
オーム社　ISBN978-4-274-20650-4

http://floor13.sakura.ne.jp/book03/book03.html

サウンドプログラミング入門　青木　直史著
技術評論社　ISBN978-4-7741-5522-7

http://floor13.sakura.ne.jp/book06/book06.html

上記文献、公開されているソースコードをベースに、当該製品向けに修正する場合の手順等を示しますので、音声処理のプログラムについて学びたい方は、参考にして頂きたく。

本書で説明するソースコードに関して

本書では、CD 内の

TEMPLATE¥RX65_SOUND_EFFECTOR_TEMPLATE

以下の、プロジェクトをベースに説明を行っています。

上記を、PC 内のストレージにコピーし、変更を行いながら動作を確認頂きたく。

1. マイコンでの処理

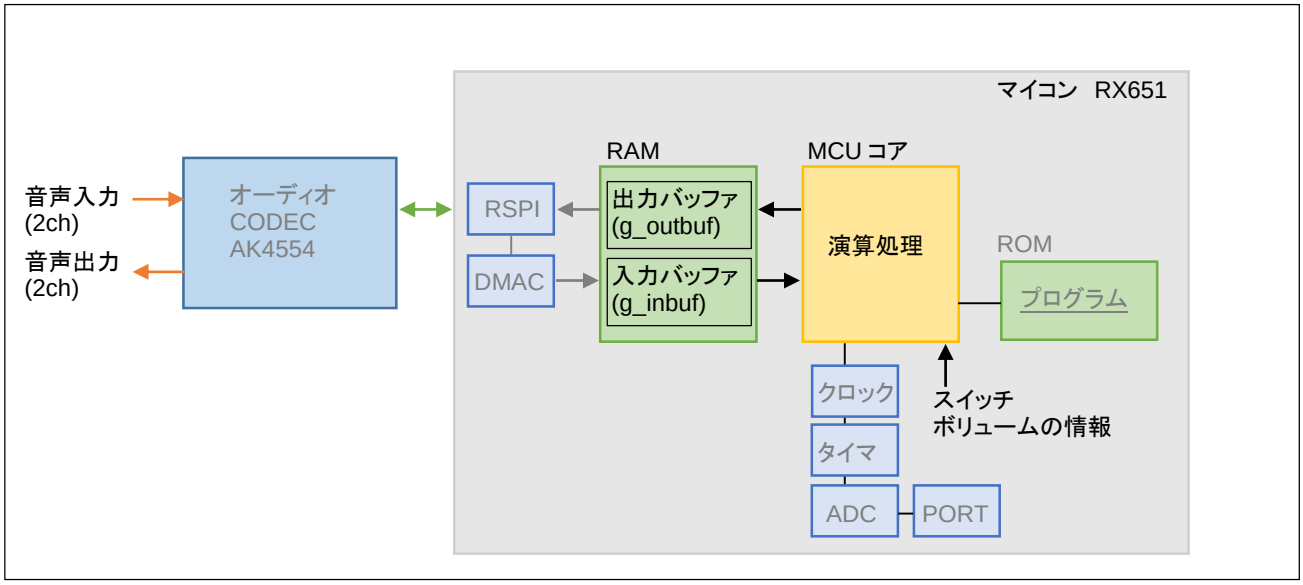


図 1-1 マイコン処理系

本製品で作成するソフトウェアは、マイコン(RX651)を制御するためのプログラムで、マイコン内部の ROM に格納され、マイコンを制御します。

マイコンでは、オーディオ CODEC チップとの通信（RSPI モジュール）や、クロック、タイマ、ADC 等の機能を使用して、本製品の目的である音声データの加工・出力を行っていますが、本書では、上図でグレーの部分（RSPI, DMAC, クロック, タイマ, ADC）の部分はブラックボックスとして扱い、以下の部分に関して解説を行います。

・入力バッファ(g_inbuf)

A/D 変換された音声データが格納されるメモリです。データは、16bit リニア PCM の生データです。

・出力バッファ(g_outbuf)

マイコンで加工済みのデータを格納する領域です。ここに格納されたデータが、音声出力として出力されます。

・演算処理部

入力バッファからデータを読み取り、加工後、出力バッファに格納する処理を行う部分です。

※本書で解説を行っていない部分は、「サウンドエフェクタ　ソフトウェア編(4)　取扱説明書」で記載しています。プログラムを、ゼロから作成する場合や、入力バッファ、出力バッファの構成を変更したい場合は、そちらも合わせて参照頂きたく。

2. 入力バッファ、出力バッファ

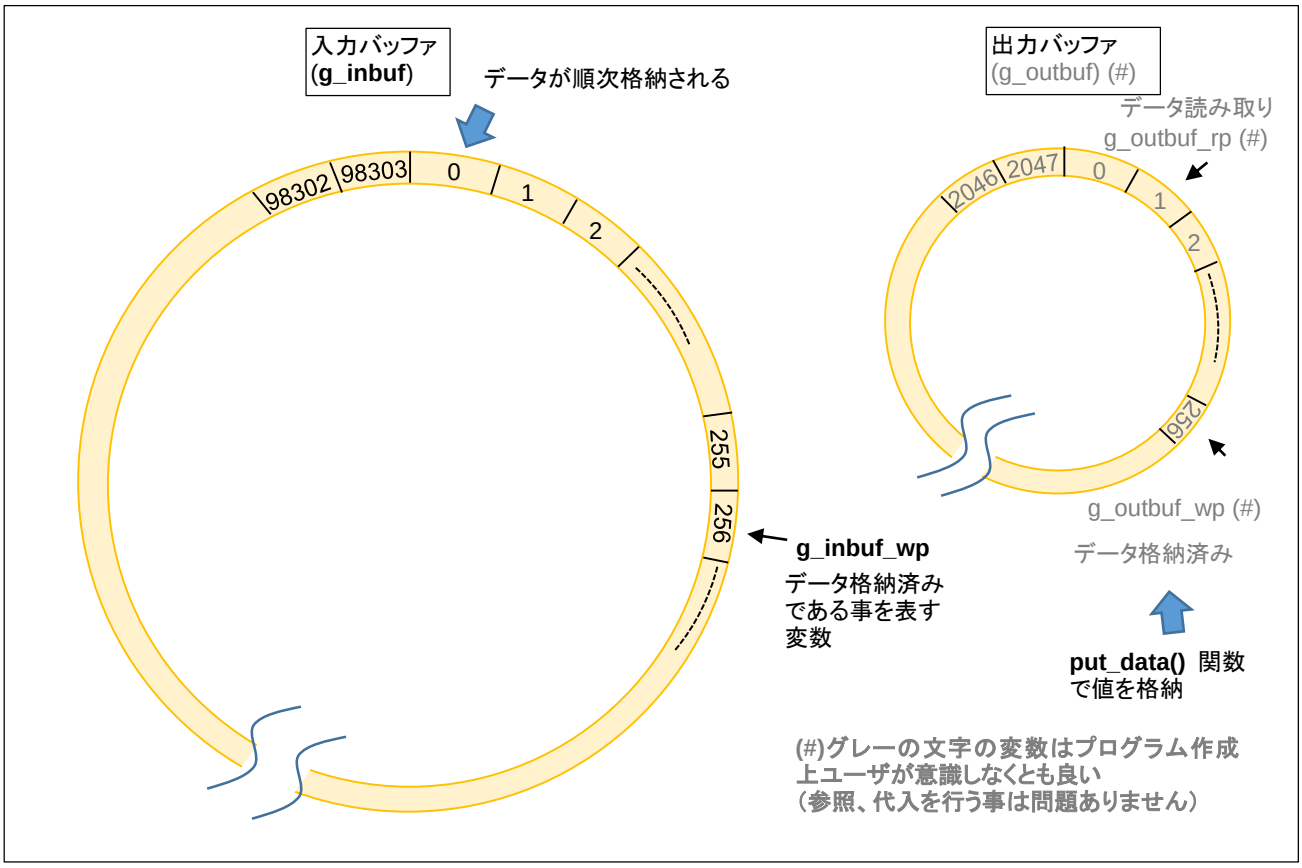


図 2-1 リングバッファ構成

入力バッファ(g_inbuf)と出力バッファ(g_outbuf)は、リングバッファの構成となっています。

リングバッファとは、0～n 個の領域に、順次データを格納していき、n までデータを格納した後は、再度 0 からデータを格納（上書き）する構成のバッファです。

入力バッファ側は、音声データを逐次 A/D 変換し、A/D 変換結果の PCM データ(48kHz, 16bit, 2ch)を、g_inbuf[n]に、順次格納します。格納処理は、バックグラウンドで行われていますので、現時点では自動的にデータが格納されれるものと考えてください。

2.1. 入力バッファのアクセス手法

データは、1/48kHz のタイミング（20.83us 毎）に、0→1→2…のバッファに格納されていきます。用意されているテンプレートのプログラムでは、入力バッファは、98304 個確保されていますので、約 2 秒（2.048 秒）で、98303 個目のバッファまでデータが埋まります。その後は、0 からデータを上書きしていきますので、約 2 秒（2.048 秒）以上前のデータは残りません。逆に言えば、約 2 秒前までのデータであれば、任意のデータにアクセス可能です。

入力バッファのデータがどこまで格納されたかを示す変数(g_inbuf_wp)を読み出せば、現在どのデータが最新かを判断できます。例えば、g_inbuf_wp=256 の時、g_inbuf[255]が最新のデータで、g_inbuf[256]が 2.048 秒前のデータです。入力バッファは、ある時点では、2.048 秒前から現在（インデックス=g_inbuf_wp）までの時系列のデータが直線的に配置されている構成であると考える事もできます。

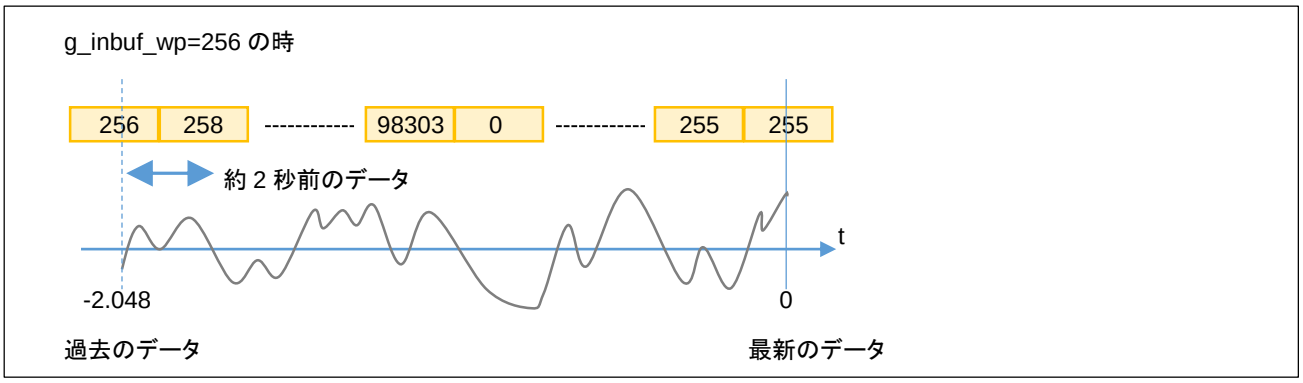


図 2-2 入力バッファ構成

g_inbuf_wp は、バックグラウンドで更新される変数です（ユーザプログラムで書き換えを行わないでください）。

（g_inbuf_wp は、データを格納：write, を指し示す：pointer の意図で、_wp と命名していますが、ポインタ型ではなく、unsigned long 型の変数です）

※g_inbuf_wp=256 に変化した際は、256 が一番古いデータとなりますが、20.83us(=1/48kHz)後には、256 に最新のデータが格納されますので、約 2 秒前のデータ（上図の←→の領域）のデータを使用する際には、データが上書きされる前に読み出せているか注意してください

入力バッファ(g_inbuf)は、PCM データ(pcm_data)の構造体となっており、以下で定義されています。

```
#define INBUF_SIZE(98304)//約2秒, 384kB

typedef struct{
    short ch0;
    short ch1;
} pcm_data;

pcm_data g_inbuf[INBUF_SIZE];
```

インデックスが 0 の ch0(Lch)のデータは、
g_inbuf[0].ch0

インデックスが 256 の ch1(Rch)のデータは、
g_inbuf[256].ch1

となります。

データは、short 型で、-32768～32767 のデータを 16bit(2bytes)の変数で扱います。

g_inbuf のインデックス（[ ]内の数値, 0~98303）は、g_inbuf_wp の値から算出してください。この計算は、ユーザプログラム側で行う必要があります。

※構造体のままでデータを取り扱う場合は、g_inbuf[index]の形で使用可能です

バッファのサイズは、テンプレートでは、98304 となっています。これは、拡張 RAM(384kB)のサイズで決まっており、減らす事は単純にできますが、増やす場合は工夫が必要です。（当面は、このサイズで使用してください）

2.2. 出力バッファのアクセス手法

出力バッファも、入力バッファ同様のリングバッファ構成となっています。但し、こちらはテンプレートでは、サイズが 2048 となっています。（入力データ側は、ディレイやエコーの処理で、過去のデータを使用する場面があるため、約 2 秒分確保していますが、出力側は未来のデータを予め計算、格納しておきたいというニーズは薄いと判断したため、このようなサイズの構成としています。2048 は、変更が可能です。）

出力バッファ側は、どこまでデータを格納したか(g_outbuf_wp)と、どこまで読み取ったか（音声データとして出力したか）を示す(g_outbuf_rp)の 2 つの変数でバッファを管理していますが、この変数の値はユーザが意識する必要がない様にしています。

また、出力バッファ変数(g_outbuf)に関しても、ユーザがアクセスしなくても良いです（アクセスしても問題ありません）。

出力バッファにデータを格納する場合は、テンプレートで用意されている関数、put_data() を使用してください。

・出力を 2ch 使用するモードの場合 [#define OUT_CH (2)]

```
int put_data( pcm_data data );
```

pcm_data data;

data.ch0 = 1000;
data.ch1 = 2000;

put_data(data); //引数として、pcm_data 構造体を与える

・出力を 1ch 使用するモードの場合 [#define OUT_CH (1)]

```
int put_data( short data );
```

short data;

data = 1000;

put_data(data); //引数として、short 型の変数を与える

または、

put_data(2000); //引数として、値を与える

put_data() 関数を呼び出す事により、出力バッファにデータが格納され、次のデータ出力のタイミング（20.83us 毎）で、セットしたデータがオーディオ CODEC を経由して出力されます。

※入力を使用する本テンプレートでは、入力 2ch に合わせ、出力側も 2ch のモードで使用する設定をデフォルトとしています（本書での説明も出力 2ch がベースです、設定により出力を 1ch で使用する事もできると認識して頂ければ問題ありません）

入力を使用しないテンプレートとの相違点

入力を使用しないテンプレート(RX65_SOUND_EFFECTOR_TEMPLATE_NOINPUT)の「バッファモード」と、本書で解説しているテンプレートの動作は似通っていますが、次の点が異なります。

入力を使用しないテンプレートでは、出力バッファが約 4 秒(1ch), 2 秒(2ch)でしたが、このテンプレートでは、2048 個（約 42ms）となっています。このテンプレートでは、入力バッファにメモリを割り振っていて、出力バッファの容量は最低限の値としています。（増やす事はできます）

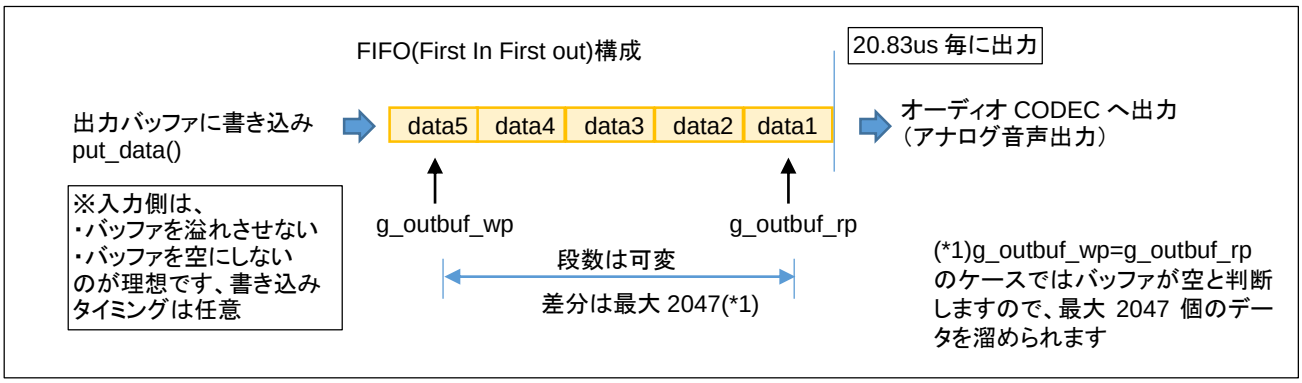


図 2-3 出力バッファ構成

出力バッファは、形式上のリングバッファの形を取っていますが、FIFO（先入れ、先出し）の構成のバッファです。

入り口側は、put_data() で、ユーザがデータを入力します。出口側は、20.83us 毎に、バックグラウンドでデータが出力されていきます。

put_data() でデータを入力するタイミングとしては、平均で 20.83us に 1 データであれば理想です。20.83us 毎に 1 データでも良いですし、333.3us（=20.83us x 16)毎に 16 個のデータを連続して入力する様な使い方でも問題ありません。

バッファを空にした場合は、無音データが出力されます。
（無音データ、有効データが繰り返されると、「プツプツ」音となりますので、注意ください）

バッファが満タン（2047 データが溜まっている）状態で、put_data() を呼び出すと、データの格納は行われません。

データの格納が成功した場合は、put_data() の戻り値は 0 となり、バッファが満タンの場合は、戻り値 1 となりますので、戻り値 1 の場合は時間を空けて再度 put_data() を呼び出す等、プログラムで工夫してください。
（タイミング的にバッファが溢れない様に工夫されていれば、put_data() の戻り値を確認する必要はありません）

2.3. データの入力単位の変更

入力バッファ(g_inbuf)は、20.83us(=1/48kHz)毎に更新されます。

入力バッファ(g_inbuf)をどこまで更新したかを示す、g_inbuf_wp に関しては、

(1)20.83us(=1/48kHz)毎に、1 ずつ増分
(2)ある程度まとめて増分

を選択できる様にしています。

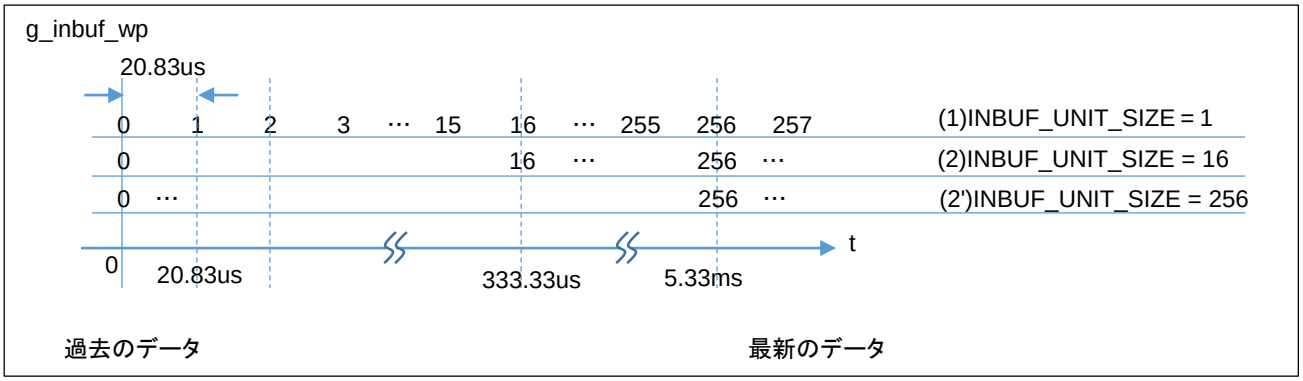


図 2-4 入力バッファインデックス変数増分値

(1)の場合は、データが格納される度（20.83us 毎）に、g_inbuf_wp が、1 ずつ増加します。

例えば(2)の場合は、16 個のデータ格納される度に、g_inbuf_wp が、0→16→32→…と、333.33us 毎に 16 刻みで増加します。(2')の場合は、256 個のデータ格納される度に、g_inbuf_wp が、0→256→512→…と、5.33ms 毎に 256 刻みで増加します。

これは、
・FFT(*1)処理等で、複数のデータをまとめて処理する必要がある
・まとまったデータを処理する事で効率化したい(*2)
・20.83us で演算が終わらない(*3)ケース
で使用する事を想定しています。

(*1)FFT：高速フーリエ変換、時間ドメインのデータを周波数ドメインのデータに変換するため、ある程度の個数の時系列のデータが入力として必要です

(*2)増分値を n とする事で、g_inbuf_wp 変数のアップデートに掛かる処理時間を 1/n に削減可能、また各種演算をまとめて行った方が効率化できるケースがあります

(*3)例えば、256 個のデータを処理する場合、トータルで 5.33ms に納まらないときは、リアルタイム処理不可（出力バッファが空となり、無音データが出力されるタイミングが生じる）となるが、最初のデータの出力までに掛かる時間が、20.83us に納まらないが、トータルでは 5.33ms に納まるケースでは、256 個のデータをまとめて処理する手法は有効です

データの処理単位を 1 とすると、20.83us 毎に 1 つのデータを出力する必要があるが、データの処理単位を 256 とすると、5.33ms 毎に 256 つのデータを出力する必要があります。

データ処理に、初回のオーバヘッドがあるケースや、データの処理速度にばらつきがあるケースでは、

(1)1 データの処理時間 ＜ 20.83us
(2)1 データの平均処理時間 ＜ 20.83us

(1)は満たせないが、(2)であれば満たせるケースが考えられます。

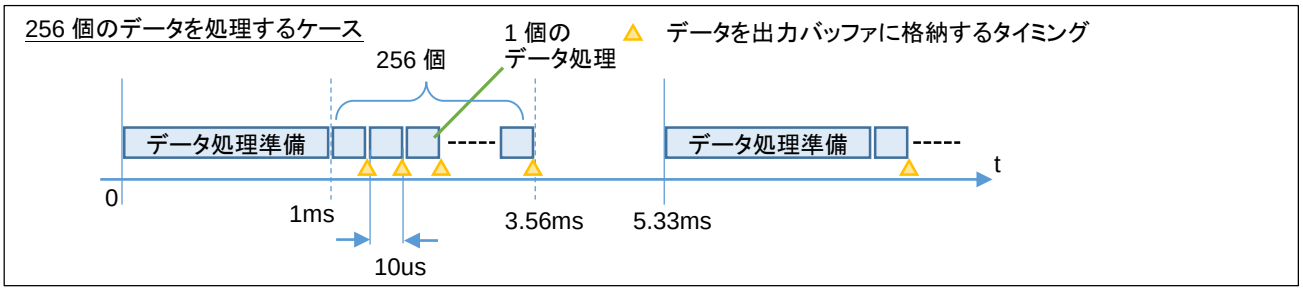


図 2-5 複数データをまとめて処理する手法

この、入力データの処理単位を変える場合、プロジェクトフォルダの

src¥user¥sound_effector¥sound_effector_userdef.h

内の、

```
//データ読み込みサイズ
#defineINBUF_UNIT_SIZE(256)//n(=256)データ毎に g_inbuf_wp を nインクリメントします
//INBUF_UNIT_SIZE は、INBUF_SIZE(=98304) % INBUF_UNIT_SIZE = 0
//となるサイズである必要があります
```

INBUF_UNIT_SIZE

の値を変更してください。

この数値は、98304 の 1/n（n:整数）である必要があります。

また、OUTBUF_SIZE(=2048)より大きくする場合は、OUTBUF_SIZE も変更する必要があります（あるかもしれません…1 回の処理で出力バッファに出力する回数に拠ります）。

テンプレートでは、256 としており、この場合は 256 個のデータが溜まらないと、g_inbuf_wp が変化しないので、一般的にはデータ処理を始めるまで、5.33ms の時間が掛かることを意味します。入力－出力間のレイテンシー時間が重要なアプリケーションでは、この値を小さく（最低は 1）してください。

コラム　データ型に関して　～　RX マイコンにおける、int 型とは？～

RX マイコンにおける、int 型は、何バイト消費して、どの範囲の数値を扱えるのでしょうか。

これは、処理系（CPU, OS　やコンパイラ）に依存する話かと思います。サウンドエフェクタでは、OS　は使用しておらず、コンパイラは標準では CC-RX コンパイラです。int のサイズを決めている要因は、マイコンの仕様に基づきます。

RX マイコンは、32bit マイコンなので、int 型は 32bit となります。

・整数型

変数型	sizeof	扱えるデータ範囲
unsigned char	1	0 ~ 255
char	1	-128 ~ 127
unsigned short	2	0 ~ 65535
short	**2**	**-32768 ~ 32767**
unsigned long (=unsigned int)	4	0 ~ 4294967296
long (=int)	4	-2147483648 ~ 2147483647
unsigned long long	8	0 ~ 2^64
long long	8	-2^64 ~ 2^64 -1

int 型は、4 バイトとなりますので、-32768~32767 の数値（16bit の PCM データ）を多数扱う場合は、short 型の方がメモリの使用効率に優れます。

・浮動小数点型

変数型	sizeof	備考
float	4	単精度
double	4	デフォルト
long double	4	デフォルト
double	8	-dbl_size=8 指定時
long double	8	-dbl_size=8 指定時

RX マイコン、CC-RX 環境では、デフォルトで float 型と double 型は同じ（4 バイト）です。
（float, double のどちらを使用しても等価です）
コンパイルオプションで、double 型を 8 バイトの倍精度として取り扱う事も可能ですが、その場合 double の演算がハードウェアではなく、ソフトウェアライブラリでの計算となるため、極力 double を単精度で取り扱う事を推奨致します。
（数値の精度や桁数が必要な場合は、-dbl_size=8 を指定して、double と float を使い分けるようにしてください。）

3. スイッチとボリュームの情報の読み取り

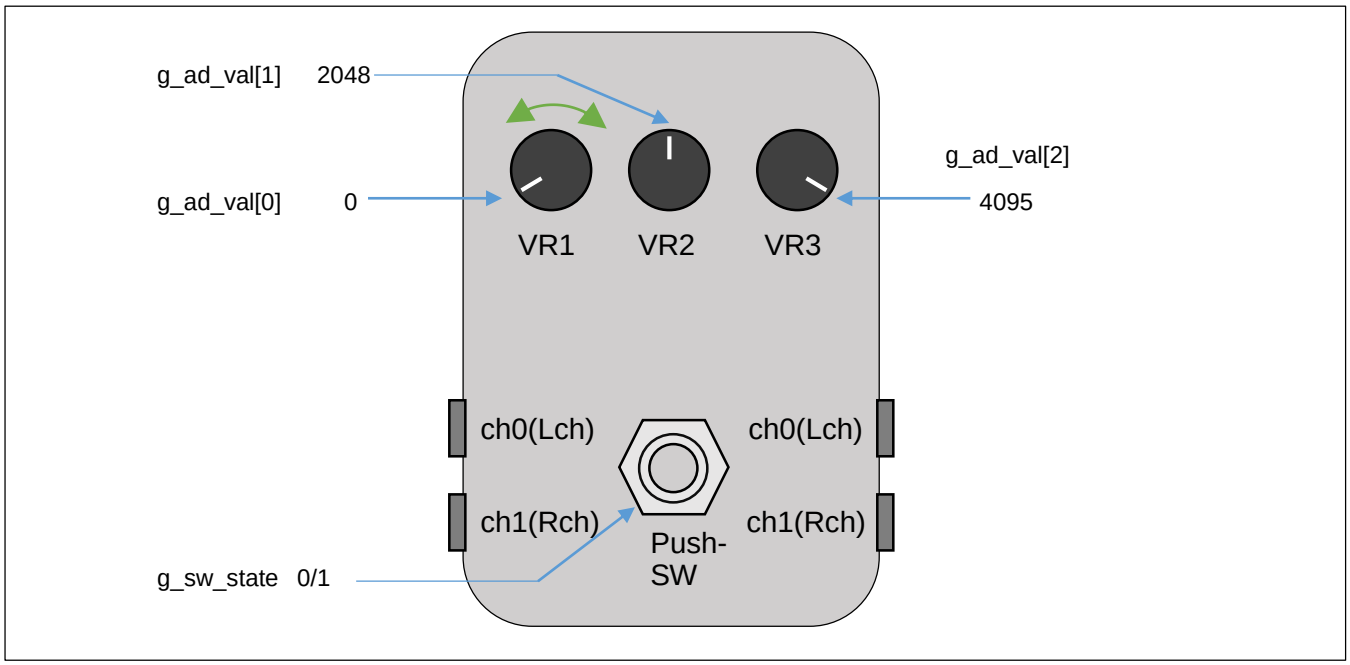


図 3-1 スイッチとボリューム

サウンドエフェクタには、ロック式のプッシュスイッチと、3 つのボリューム(VR1~VR3)があります。これらのスイッチとボリュームは、バックグラウンドで 50ms 毎に読み取り処理を行い、グローバル変数に反映されます。

プッシュスイッチは、

g_sw_state

0:スイッチが押ささっている
1:スイッチは押ささっていない

変数に、状態が反映されます。

ボリュームは、

VR1 : g_ad_val[0]
VR2 : g_ad_val[1]
VR3 : g_ad_val[2]

に、回転角度に応じた値が入ります。ボリュームは、反時計回りに目一杯回した際に、0。時計回りに目一杯回した場合、4095 となります。(センター付近で、2048)

4. 演算処理部

演算処理の基本は、入力データ(g_inbuf)の数値を演算し、出力（put_data() 関数）に渡す事です。

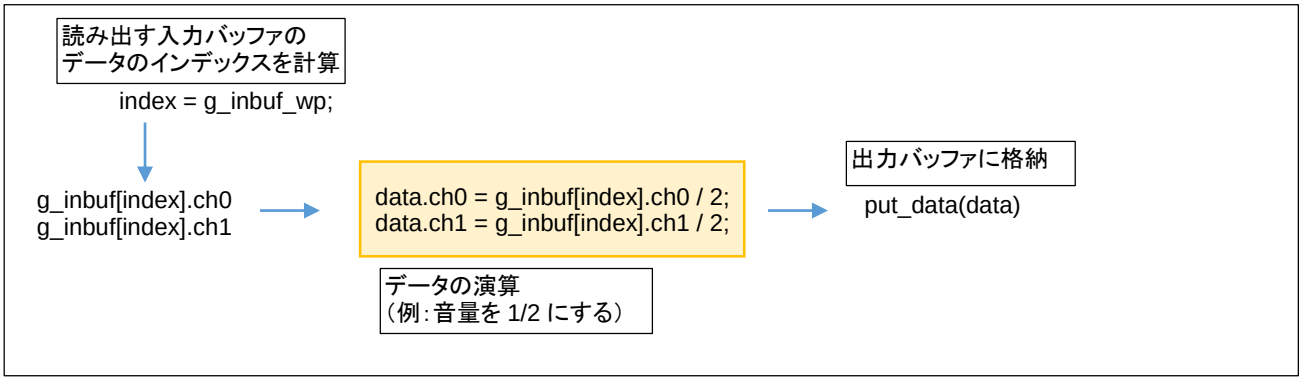


図 4-1 演算処理部の基本構成

4.1. メイン関数

テンプレートのプログラムのメイン関数は、

RX65_SOUND_EFFECTOR.c

内の、main() となります。

組み込み系のプログラムでは、メイン関数から抜けない様にプログラムを組むのが流儀となります。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{
    while(1)
    {

        //演算処理内容

    }
    //ここには来ない様に！
}
```

メイン関数内に、while 等の無限ループを設定し、メイン関数から抜ける事の無い様にします。

PC 向けのアプリケーションとは、異なる点ですので、注意ください。

4.2. 入力データがあった事を検知する方法

(1)1 データずつ処理する方法

src¥user¥sound_effector¥sound_effector_userdef.h

```
//データ読み込みサイズ
#defineINBUF_UNIT_SIZE(1)//n(=256)データ毎に g_inbuf_wp を nインクリメントします
//INBUF_UNIT_SIZE は、INBUF_SIZE(=98304) % INBUF_UNIT_SIZE = 0
//となるサイズである必要があります
```

INBUF_UNIT_SIZE を 1 としてください。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;

（中略）

    prev_inbuf_wp = 0;

    while(1)
    { //ここから
        if(prev_inbuf_wp != g_inbuf_wp)
       {
            put_data(g_inbuf[prev_inbuf_wp]);    //g_inbuf をそのまま put_data で出力
            prev_inbuf_wp = g_inbuf_wp;

        }
    } //ここまで
}
```

上記は、入力された音声を、スルーで出力するプログラム例です。

prev_inbuf_wp という変数を定義し、この変数と g_inbuf_wp（自動的に更新される変数）を比較し、変化していれば put_data() を呼び出すというものです。

while(1)内のコード（「ここから」「ここまで」で囲まれた部分）は、常にループしています。

g_inbuf_wp は、20.83us 毎に g_inbuf にデータ格納後、インクリメント(+1)されます。g_inbuf_wp が変化したタイミングで、

put_data(g_inbuf[prev_inbuf_wp]);

の行が実行され、格納済みの最新のデータ(g_inbuf[prev_inbuf_wp])が、put_data() で、出力バッファに格納されます。

その後、

prev_inbuf_wp = g_inbuf_wp;

で、現在の g_inbuf_wp をprev_inbuf_wp に保存します。

次に、g_inbuf_wp が変化したタイミングで、再度同じ動作が実行され、この動作が永遠に続く事となります。

入力バッファにデータが格納された事を、保存しておいた g_inbuf_wp と現在の g_inbuf_wp を比較する事により行う方式です。

(2)複数のデータをまとめて処理する方法

src¥user¥sound_effector¥sound_effector_userdef.h

```
//データ読み込みサイズ
#defineINBUF_UNIT_SIZE(256)//n(=256)データ毎に g_inbuf_wp を nインクリメントします
//INBUF_UNIT_SIZE は、INBUF_SIZE(=98304) % INBUF_UNIT_SIZE = 0
//となるサイズである必要があります
```

テンプレートのデフォルト値のままです。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;
    int n;

（中略）

    prev_inbuf_wp = 0;

    while(1)
    { //ここから
        if(prev_inbuf_wp != g_inbuf_wp)
       {
            for(n=0; n<INBUF_UNIT_SIZE; n++) put_data(g_inbuf[prev_inbuf_wp+n]);
                                        //g_inbuf をそのまま put_data で出力
            prev_inbuf_wp = g_inbuf_wp;

        }
    } //ここまで
}
```

複数データをまとめて処理する方法は、(1)と基本的には変わりません。

・put_data() を呼び出す部分を一度に扱うデータ個数分ループさせる
・g_inbuf のインデックスを prev_inbuf_wp + n と設定する

というだけです。

データを 1 個ずつ処理するか、複数まとめて処理するかは、2 章でも説明していますが、本製品のプログラムを行う上で重要な部分ですので、再度説明しています。データの処理単位は、使用するアプリケーションに応じて選択できますので、アプリケーションに適した処理単位を選択してください。

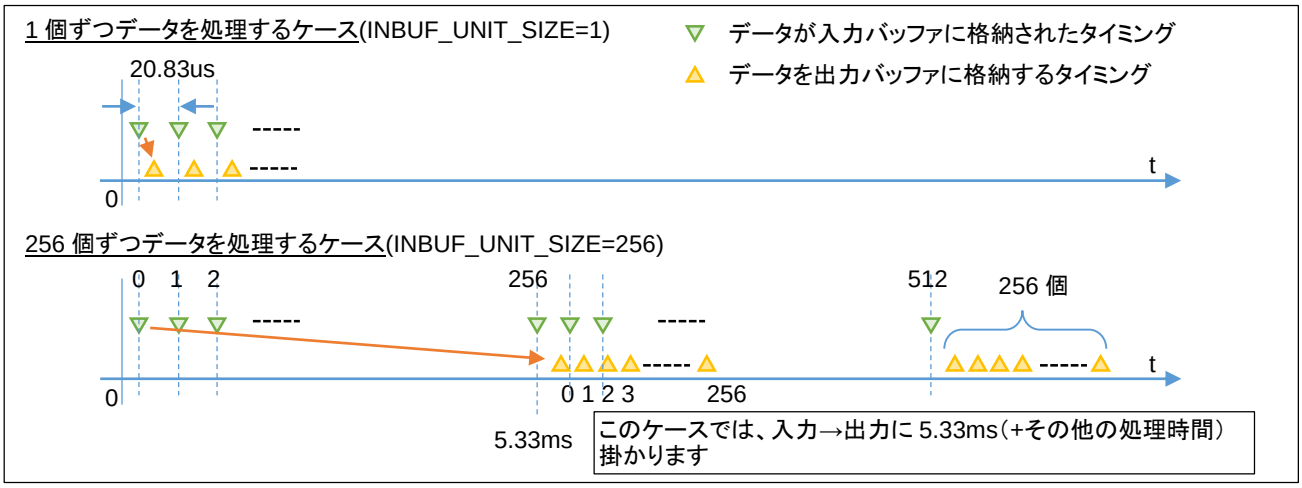


図 4-2 1 データ毎に出力する場合と複数データをまとめて処理する場合

表 4-1 1 データ毎に出力する場合と複数データをまとめて処理する場合

	1 個ずつデータを処理	複数のデータをまとめて処理
入力→出力間のレイテンシー	○最小化できる	△INBUF_UNIT_SIZE に応じたレイテンシーが掛かる
バックグラウンド処理のオーバヘッド	×大	○小
データ処理の時間余裕	△データ入力から処理を終えるまで 20.83us の時間的余裕しかない	○トータルで(256 データの処理で、5.33ms)時間的余裕を守れれば良い

(1)(2)では、

・g_inbuf_wp の値を保存
・保存した g_inbuf_wp と g_inbuf_wp が不一致となったタイミング

という手法を採っています。

(3)1 データずつ処理する方法(2)

```
void main(void)
{

 （中略）

    while(1)
    { //ここから
        if(g_sampling_period_flag == 0)
       {
            data = .....;//data の演算
            put_data(data);
            g_sampling_period_flag = 1;//フラグのセット
        }
    } //ここまで
}
```

g_dampling_period_flag は、20.83us に 1 回フラグがクリアされますので、この変数を用いて 20.83us に 1 回、put_data() を呼び出すようにする手法も考えられます。

(4)複数のデータをまとめて処理する方法(2)

```
void main(void)
{

 （中略）

    while(1)
    { //ここから
        if((g_sampling_counter % 256) == 0)
       {
            data[i] = .....;//data の演算
            for(i=0; i<256; i++) put_data(data[i]);
        }
    } //ここまで
}
```

g_sampling_counter は、20.83us に 1 ずつインクリメントされます。この変数の増分や余剰をみて、処理する手法も考えられます。

その他にも、「タイマを使用する」等の手法も考えられます。4.2 で説明している手法は、あくまでいくつか考えられる手法の一つであるとお考えください。

4.3. 2 つの入力に別々な演算を施す方法

本製品は、2ch の入出力がありますが、ch0 と ch1 に別々に取り扱いたいケースがあるかと思います。

ch0 の音量を 1/2 とし、ch1 の音量を 2 倍とするケースを考えてみます。INBUF_UNIT_SIZE は 1 でも 2 以上でも対応出来る様、n でループ処理する様にしています。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;
    int n;
    pcm_data tmp_data;
    float ch0_data, ch1_data;

 （中略）

    prev_inbuf_wp = 0;

    while(1)
    {
        if(prev_inbuf_wp != g_inbuf_wp)
       {
            for(n=0; n<INBUF_UNIT_SIZE; n++)
            {
                ch0_data = (float)inbuf[prev_inbuf_wp+n].ch0/32768.0;  //1 に正規化
                ch1_data = (float)inbuf[prev_inbuf_wp+n].ch1/32768.0;  //1 に正規化

                ch0_data /= 2.0;
                ch1_data *= 2.0;

                tmp_data = clipping_data_norm(ch0_data, ch1_data);

                put_data(tmp_data);
            }
            prev_inbuf_wp = g_inbuf_wp;
        }
    }
}
```

g_inbuf[index] は、pcm_data 構造体なので、

g_inbuf[index].ch0　ch0 の pcm データ（生データ）
g_inbuf[index].ch1　ch1 の pcm データ（生データ）

.ch0, .ch1 を後ろに付ける事により、ch0, ch1 別々のデータにアクセスできます。（4.2 章のプログラムでは、別々には取り扱っていませんでした）

ch0_data = (float)inbuf[prev_inbuf_wp+n].ch0/32768.0;

g_inbuf[index].ch0 は、short 型なので、-32768 ~ 32767 の値しか格納できないので、そのまま 2 倍するとオーバフローにより値が化ける事が起こりえるので、-1 ~ 1 のデータに正規化して、float 型の変数(ch0_data)に格納します。

その後、任意の演算（ここでは、/=2.0 と*=2.0）を施し、

tmp_data = clipping_data_norm(ch0_data, ch1_data);

clipping_data_norm() という関数を呼び出しています。この関数は、-1 ~ 1 から外れたデータをクリッピング処理、及び 16bit の PCM データに伸長（-1 ~ 1 → -32768 ~ 32767 に変換）するものです。

```
pcm_data clipping_data_norm(float ch0_data, float ch1_data)
{
//クリッピング処理, short型への変換を行う関数
//（入力データは1に正規化されたデータ）

//引数：
//  float ch0_data  : ch0のデータ
//  float ch0_data  : ch1のデータ

//戻り値：
//  pcm データ（short 型の構造体）

（関数本体は省略）
```

float 型の引数を 2 つ受け取り、pcm_data 構造体を返します。

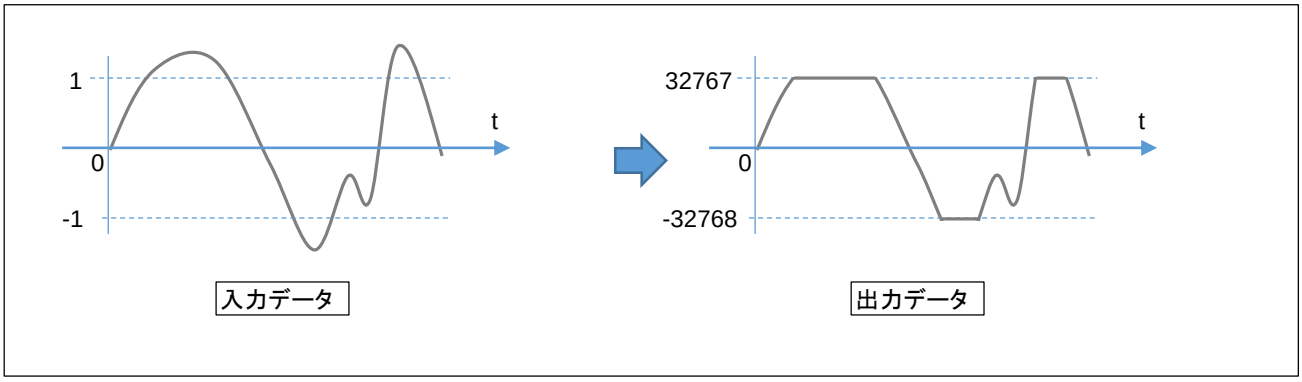


図 4-3 クリッピング処理（＋伸長）のイメージ

本サンプルを参考に、ch0 と ch1 のデータを扱う際は、構造体のメンバを参照する様にしてください。

4.4. 過去のデータを扱う方法

ディレイやエコーのプログラムでは、過去のデータを扱うケースがでてきます。

ここでは、現在の音声信号に、50ms 前のデータを 1/2 の強度で加算するケースを考えてみます。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;
    int n;
    pcm_data tmp_data;
    float ch0_data, ch1_data;
    long past_index;

 （中略）

    prev_inbuf_wp = 0;

    while(1)
    {
        if(prev_inbuf_wp != g_inbuf_wp)
       {
            for(n=0; n<INBUF_UNIT_SIZE; n++)
            {
                ch0_data = (float)inbuf[prev_inbuf_wp+n].ch0/32768.0;  //1 に正規化
                ch1_data = (float)inbuf[prev_inbuf_wp+n].ch1/32768.0;  //1 に正規化

                past_index = prev_inbuf_wp + n - 2400;  //50ms 前のデータ
                if(past_index < 0) past_index += INBUF_SIZE;
                else if(past_index >= INBUF_SIZE) past_index -= INBUF_SIZE;

                ch0_data += (float)inbuf[past_index].ch0/32768.0/2.0;  //50ms 前の 1/2 を加算
                ch1_data += (float)inbuf[past_index].ch1/32768.0/2.0;  //50ms 前の 1/2 を加算

                tmp_data = clipping_data_norm(ch0_data, ch1_data);

                put_data(tmp_data);
            }
            prev_inbuf_wp = g_inbuf_wp;
        }
    }
}
```

ここで、ポイントとなるのは、past_index, 現在より 50ms 前のデータを示すインデックスです。

2400 という数値は、50ms / 20.83us より来ています。現時点のデータ(prev_inbuf_wp+n)から 2400 前のデータが、50ms 前のデータとなります。

ここで、注意が必要なのは、リングバッファ構成です。リニアなバッファ構成であれば、2400 を減じた値を考えれば良いのですが、リングバッファの場合は 0 をまたぐ場合を考慮に入れる必要があります。

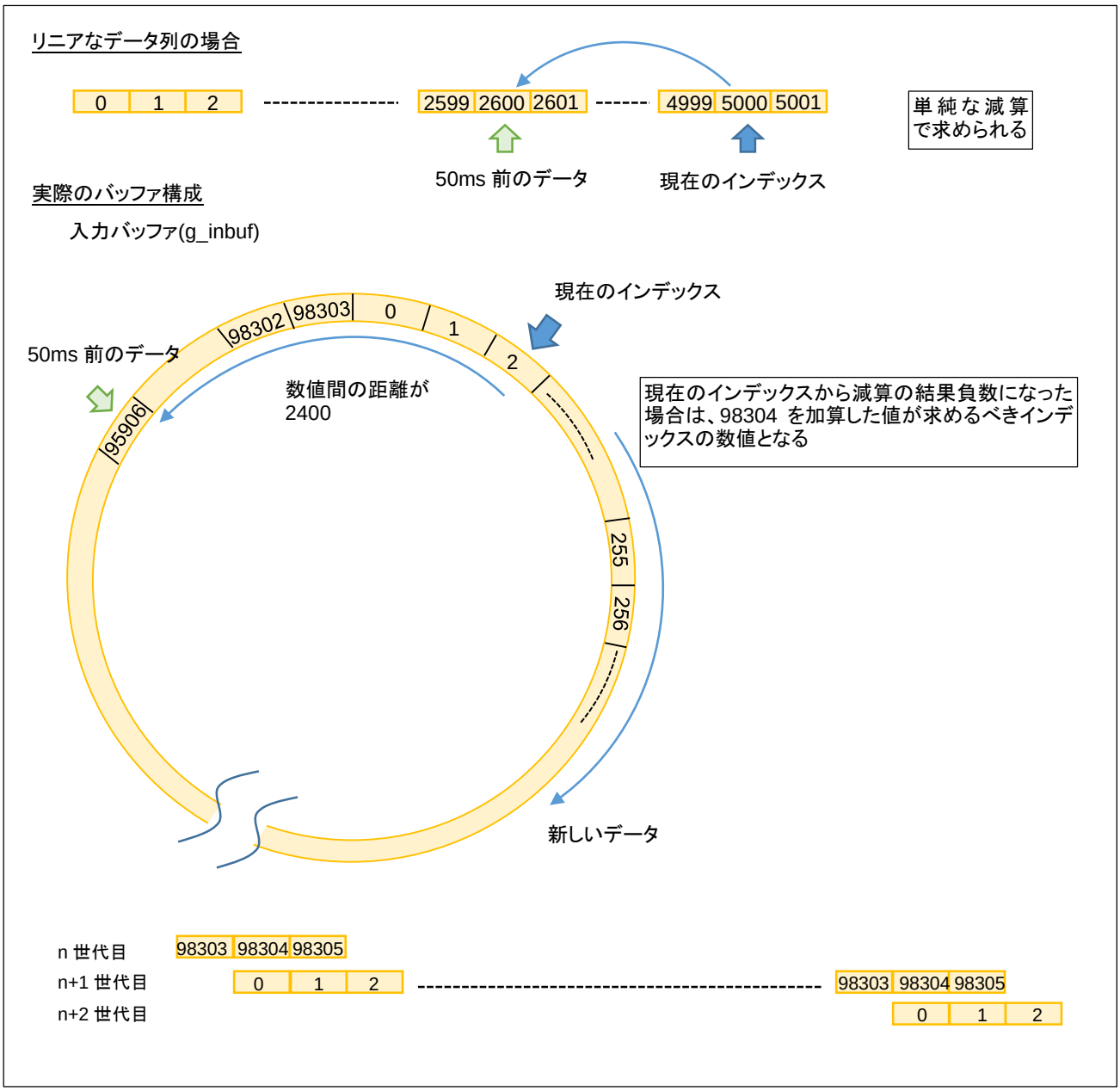


図 4-3 リングバッファ構成のインデックス

```
past_index = prev_inbuf_wp + n - 2400;
```

基本的には、現在のインデックスから 2400 前（50ms 前）のデータが欲しい

```
if(past_index < 0) past_index += INBUF_SIZE;
```

計算の結果負数となった場合は、98304 を加算した数値が、求めるべきインデックスとなります。

else if(past_index >= INBUF_SIZE) past_index -= INBUF_SIZE;

本プログラムでは上記、計算後のインデックスが 98034 を超える事はありませんが、インデックスの取り扱いによって超える事がある様であれば、減算が必要なケースも考えられます。

5. プログラムを移植する上の注意点

4 章では、単純な演算に関して、ソースコード例を示して説明を行いました。

本章では、参考文献のプログラムを本機器に移植する場合の注意点に関して記載します。

5.1. 入出力の相違

参考文献「C 言語ではじめる　音のプログラミング」「サウンドプログラミング入門」では、概ね以下の構成となっています。

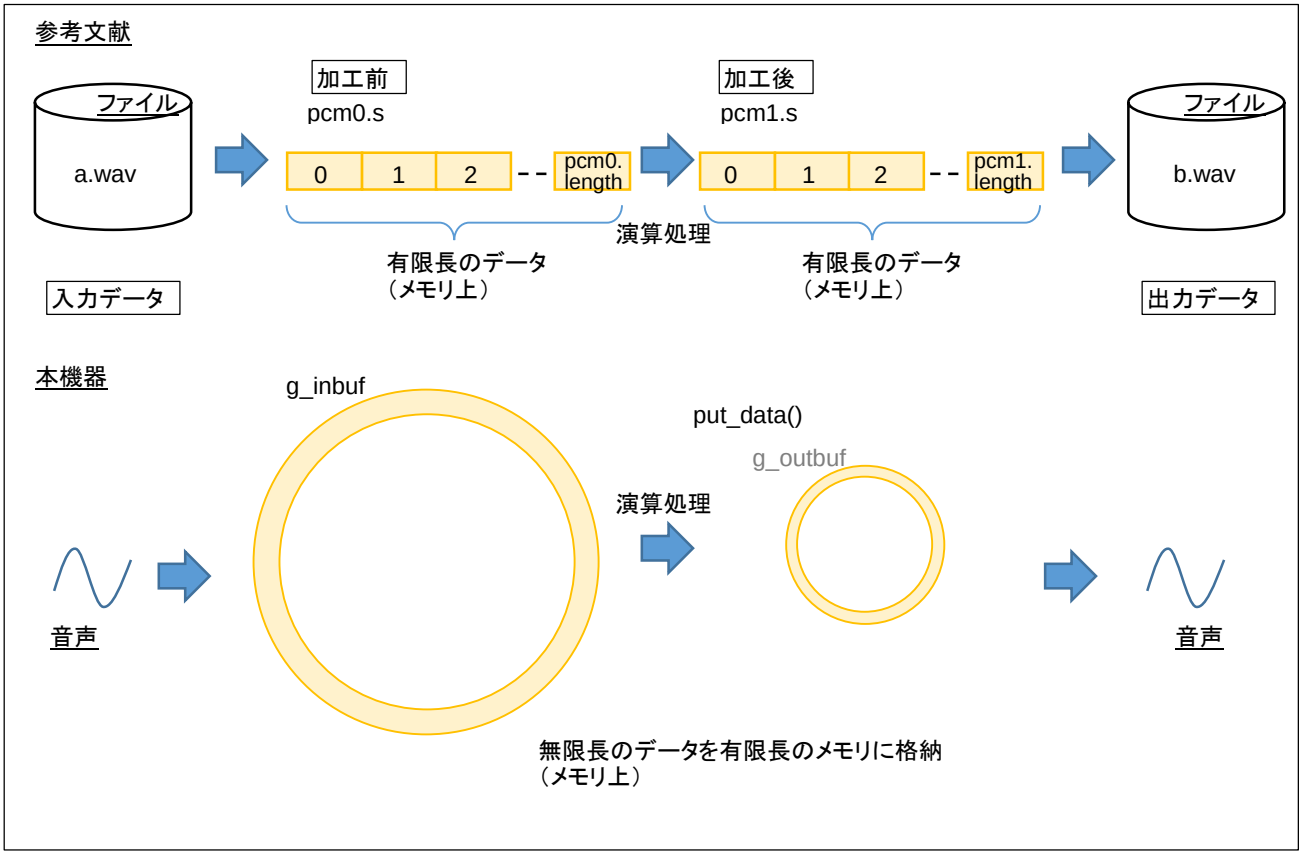


図 5-1 参考文献との相違点

表 5-1 参考文献との相違点

	参考文献	本機器	備考
入力データの読み取り	pcm0.s[n] の値	g_inbuf[n] の値	本機器では過去約 2 秒分のデータしかアクセス不可
出力データの書き込み	pcm1.s[n] に代入	put_data() を使用	参考文献ではファイルに出力するまで何度でも pcm1.s[n]の値を変更可能(*1)

※pcm0, pcm1 が入力で pcm2 が出力のケースもあります

参考文献では、ファイルに対して変更を加えるのに対し、本機器では音声のストリームデータを編集するという、根本的な相違があります。

プログラム作成上の相違としては、入力データのアクセス方法は参考文献でも本機器でも、大きく変わりません。

参考文献　pcm0.s[n]　n:0~pcm0.length
本機器　　g_inbuf[n]　n:0~98303

本機器では、

・リングバッファ構成となっており、n=0 が必ずしも古いデータでない事が注意点
・2 秒以上前のデータは失われている

となります。

本機器でプログラムを作成する際は、n の計算方法で工夫する必要があります。

出力データの書き込みは、

参考文献　pcm1.s[n] = val;
本機器　　put_data(pcm_data_val);

参考文献では、構造体に値を代入していますが、本機器では関数に値を与える様になります。

参考文献では、pcm1.s[n]は、ファイルへの出力処理を行うまでは、加算（書き換え）や読み出しが可能です。

本機器では、書き換えは基本的には不可となります。(*1)

(*1)正確には、値をセットした後、オーディオ CODEC にデータを渡す前であれば書き換え可能です。しかし、オーディオ CODEC にデータを渡すタイミングは put_data() 呼び出し直後の事もありますので、基本的には、一度 put_data() で出力バッファにデータ格納後は書き換えできないものとお考えください

put_data() で値を出力バッファに書き込んだ後の読み出しに関してですが、テンプレートのプログラムでは 2048 データまでは可能です（2048 という個数を増やす事も可能です）。

本機器では、無限長のデータを取り扱いますので、処理する内容によっては、有限長のデータを一括処理する手法が適用できない場合があります。

・サイン波や矩形派の出力

入力を使用せず、機器内部で波形を生成する場合は、時系列で出力されるデータを順次出力バッファに書き込んでいくという処理となりますので、無限長に関して意識する必要はないかと考えます。但し、出力バッファへの書き込みは、出力のサンプリングレート（48kHz, 20.83us に 1 データ）となるように調整が必要です。

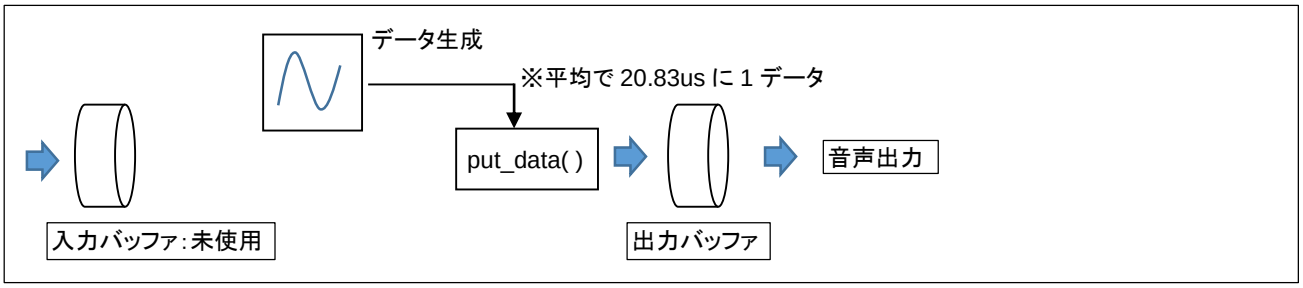


図 5-2 内部で波形を生成するケース

・リミッタ、コンプレッサ、ゲイン、クリッピング

これらは、入力データと出力データの時系列にずれがない処理ですので、入力データを演算し順次出力バッファに書き込んでいく処理となります。無限長であることの考慮は不要であると考えます。

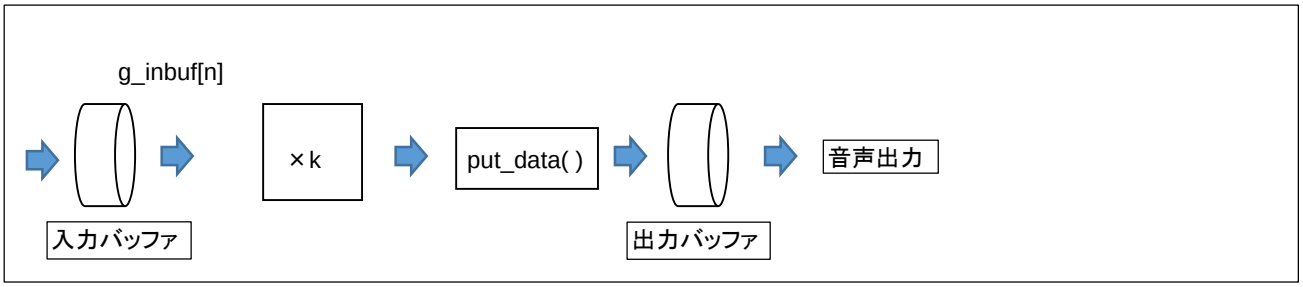


図 5-3 時系列的なずれがない系

・ディレイ、エコー、リバーブ

これらは、過去のデータを現在のデータに加減算する処理となります。4.4 節で説明している手法で配列のインデックス計算を行えば、有限長のデータを取り扱う場合と同様の処理が可能です。

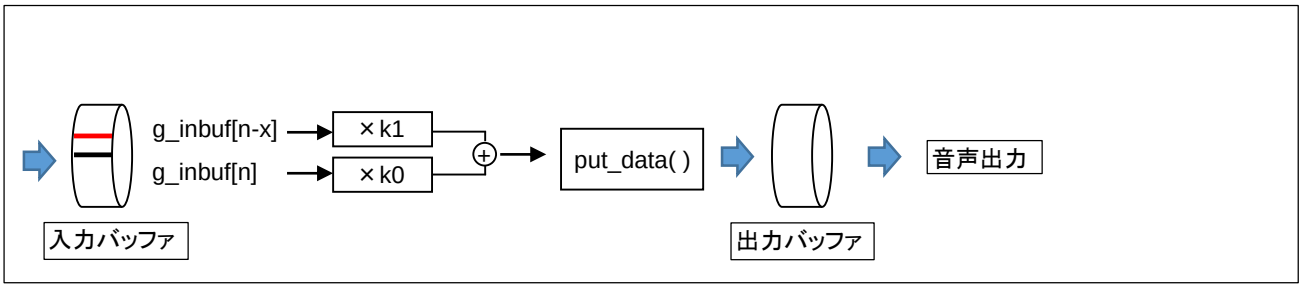


図 5-4 過去のデータを利用する系

・フィルタ

LPF, BPF 等のフィルタ処理を本機器に適用する場合、フレーム単位での処理を行う必要があります。参考文献「C 言語ではじめる音のプログラミング」6.3「フレーム単位でのフィルタリング」に詳細な手法が記載されていますので、参照ください。

・FIR フィルタの処理例

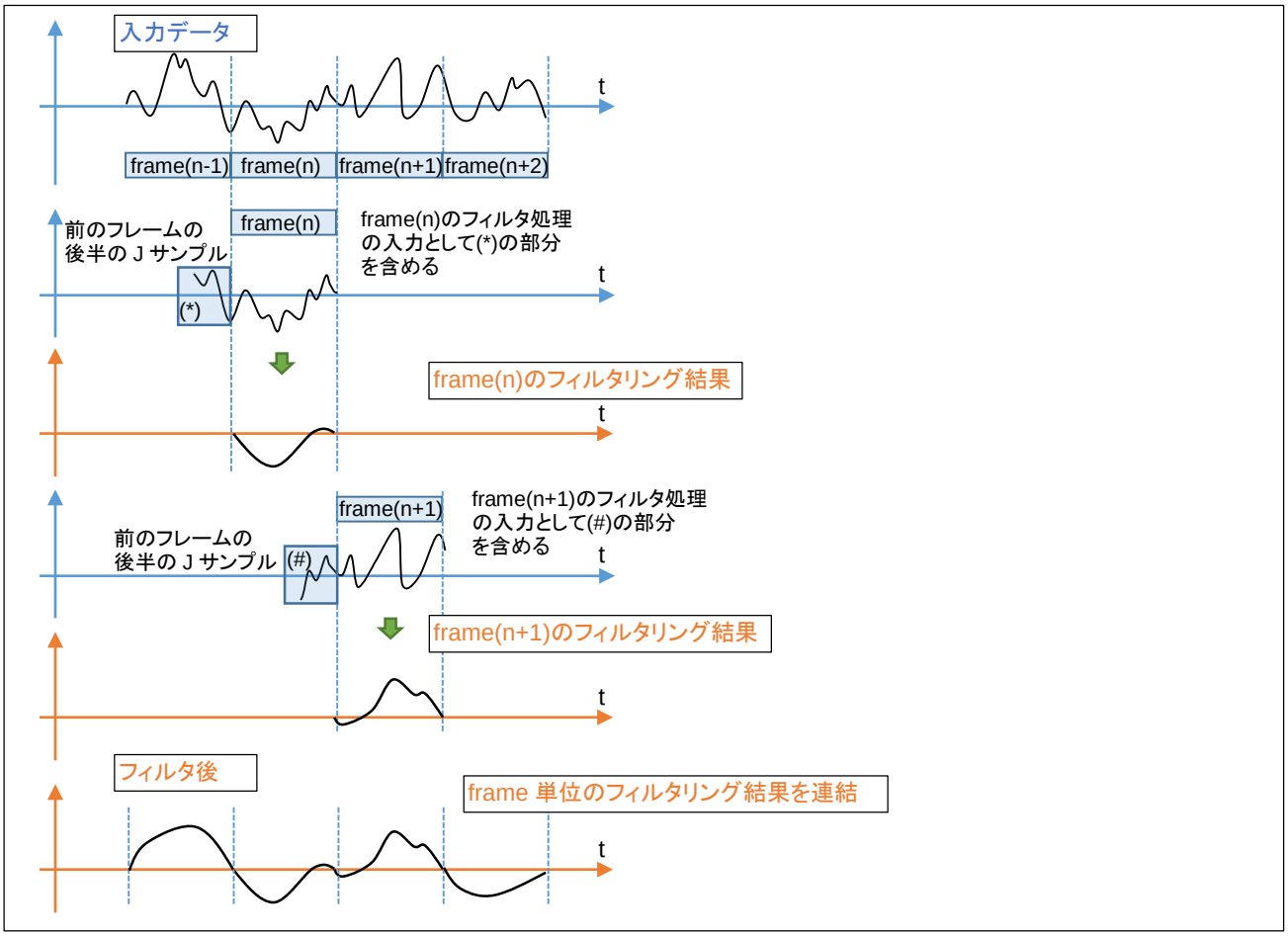


図 5-5 フレーム単位でのフィルタ処理(FIR フィルタ)

FIR フィルタの場合は、フィルタリングの入力として、処理対象フレームより「時間的に前のデータ」を含めてフィルタリング処理を行い、フィルタリング結果を連結して、フィルタの出力結果を得ます。

・IIR フィルタの処理例

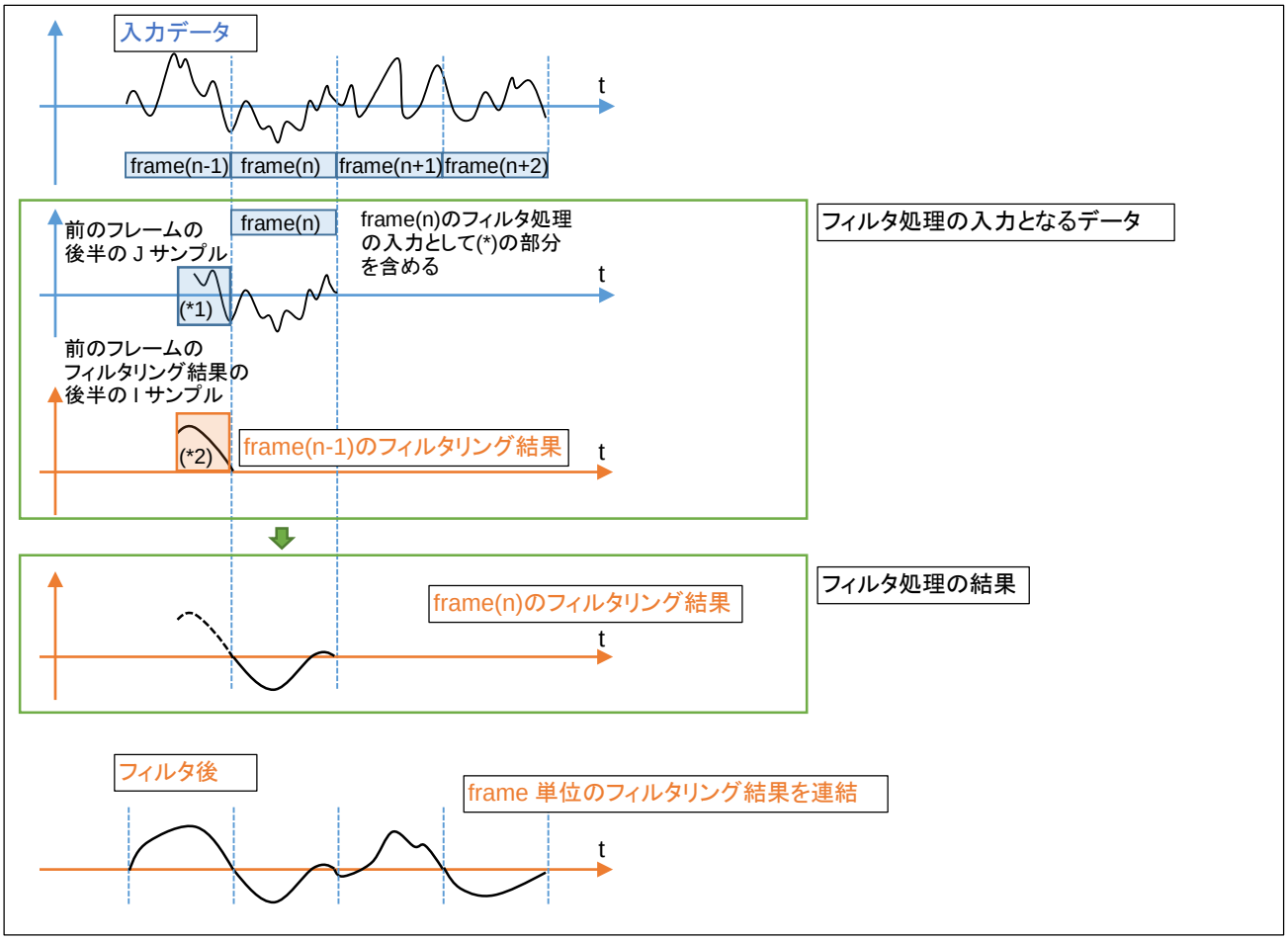


図 5-6 フレーム単位でのフィルタ処理(IIR フィルタ)

IIR フィルタの場合は、

処理対象フレームより「時間的に前のデータ」(*1)

に加え、

前フレームのフィルタリング結果の後半のデータ (*2)

を含めて、フィルタリング処理を行い、結果を連結する事により、フィルタの出力結果を得ます。

・FFT（高速フーリエ変換）

FFT は、有限のサンプル数のデータに対して適用されますので、本機器で FFT を適用する場合、データをフレーム単位に切り出して、処理する必要があります。

・FFT, IFFT の処理例

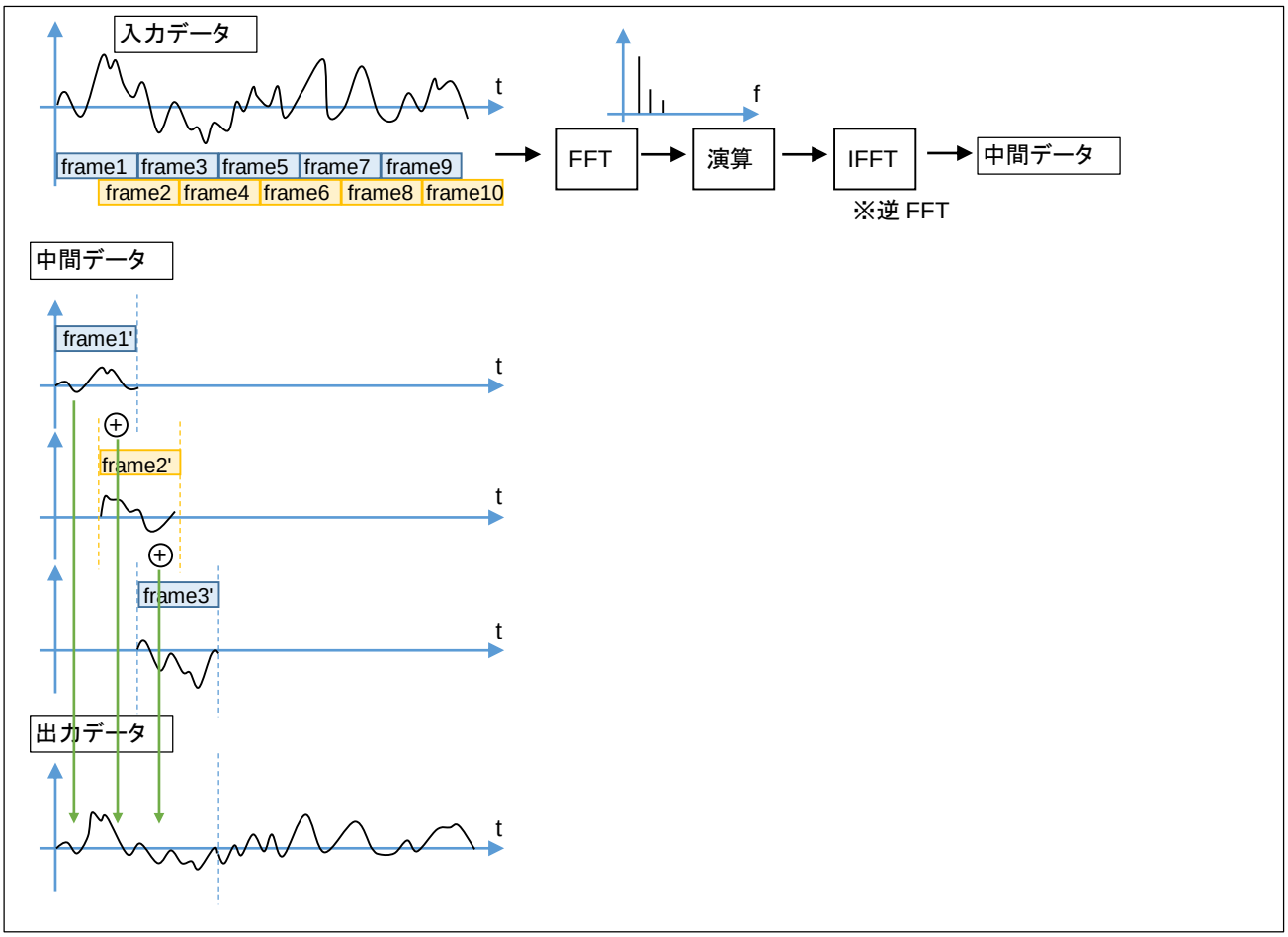


図 5-7 フレーム単位でのオーバラップアド処理

FFT の処理では、入力データがオーバラップする様にフレーム化（切り出し）を行い、処理を行う手法が考えられます。

フレーム長が 1024 の場合、各フレームを 512 ずつオーバラップさせて切り出しを行い、フレーム毎に FFT，演算処理, IFFT を行い、結果を加算していく事により、連続的な出力を得る事ができます。

無限長のデータをフレームに分割して取り扱う際は、処理内容により、手法を考慮する必要があります。

その他、本機器では 2ch の入出力を取り扱います。

参考文献　pcm0[n].s　　…1ch
本機器　　g_inbuf[n].ch0, g_inbuf[n].ch1　　…2ch

2ch 使用する必要がない場合は、g_inbuf[n].ch0 側のみを使用する使い方でも問題ありません。

出力側は、OUT_CH 定数の設定により、1ch（ch0 と ch1 から同じ音声が出力される）化することも可能です。

※入力側を 1ch 化する（1ch 化により、約 4 秒のデータを保持可能）事も可能です
（入力 1ch 化は、ソフトウェア編(4)で説明します）

量子化ビット数は、参考文献、本機器とも、16bit です。

サンプリング周波数は、

参考文献　pcm0.fs　　…可変
本機器　PCM_FS　　(48kHz)　　…固定

です。参考文献のプログラムで、pcm0.fs（入力ファイルのサンプリング周波数）を使用している箇所は、PCM_FS（定数）を参照する様にしてください。

5.2. メモリ

参考文献では、入力とするファイルの全データを、pcm0[n].s に格納して処理を行っているものが多いですが、本機器で使用可能なメモリには制限があります。

※PC 上で動かすプログラムにも、メモリの制限はありますが、最近の PC で扱えるメモリ量に対し、本機器で扱えるメモリは少ないものとなります。

入力バッファ(g_inbuf)　384kB （ch あたり、98304 データ、2ch）
その他（出力バッファ、ワーク領域等） 256kB

合計、640kB のメモリ内で処理可能な様にプログラムを組む必要があります。

5.3. 演算速度

本機器に搭載されているマイコンは、一般的な最近の PC に搭載されている CPU と比べると計算能力が低く抑えられています。

また、PC 上でファイルに加工を施すプログラムの場合、CPU が非力なマシンで実行した場合、ファイルが出力されるまでの待ち時間が長くなるというだけ（出力結果は変わらない）ですが、本機器の場合 20.83us で 1 データ処理出来ない場合は、音声出力が途切れたり、期待と全く異なる出力となる事も考えられます。

よって、一定の時間内に演算処理を終わらせる様、工夫が必要なケースがでてきます。

・整数で計算を行う

一般に、float, double 等の浮動小数点型の変数を使用したケースより、int, short, long 等の整数型の変数の演算の方が高速に処理可能です。

※但し、本機器で採用されているマイコン(RX651)は、浮動小数点ユニットを持っていますので、それ程整数演算にこだわる必要はありません

・ループ外で計算可能なものは、ループの外に追いやる

ループ内で計算している式で、共通部分がある場合は、ループに入る前に計算を済ませておく。

・PCM データの正規化が必須でない場合は省略する

入力データ（-32768 ~ 32767, 16bit 符号付き整数）を、-1 ~ 1 に正規化を行い処理する事が必須ではない場合、正規化の処理と、16bit 符号付き整数に戻す処理を省略する事が考えられます。

・速度の最適化例

参考文献「C 言語ではじめる音のプログラミング」のリバーブの処理 ex3_2.c を最適化する例を示します。

－最適化前－

オリジナルのソースコードをできるだけ忠実に移植した場合です。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;

    int n, m, i, repeat;
    double a, d;

    double out_ch0, out_ch1;

    pcm_data pcm_tmp;

    long inbuf_index;

    const int pcm_fs = PCM_FS;//サンプリングレート 48kHz

    a = 0.5;/* 減衰率 */
    d = pcm_fs * 0.05;/* 遅延時間 */
    repeat = 10; /* 繰り返し回数 */

    //debug

    PORTE.PDR.BIT.B3 = 1;
    PORTE.PODR.BIT.B3 = 0;

    //初期化関数
    sound_effector_init();

    prev_inbuf_wp = 0;

    while(1)
    {
        if(g_sw_state == 0)//スルー
        {
            if(prev_inbuf_wp != g_inbuf_wp)
            {
                for(n=0; n<INBUF_UNIT_SIZE; n++)
                {
                    put_data(g_inbuf[prev_inbuf_wp + n]);
                }
                prev_inbuf_wp = g_inbuf_wp;
            }
        }
        else
```

[次ページへ続く]

```
        {
            //リバーブ
            if(prev_inbuf_wp != g_inbuf_wp)//データが INBUF_UNIT_SIZE(=256) 溜まった
            {
                PORTE.PODR.BIT.B3 = 1;//ポートで処理時間をモニタ（処理スタート）★

                for(n=0; n<INBUF_UNIT_SIZE; n++)
                {

                    /* 現在の時刻の音データ */
                    //オリジナルはwave.hで、ファイル読み込み時に1に正規化
                    out_ch0 = (double)g_inbuf[prev_inbuf_wp + n].ch0 / 32768.0;
                    out_ch1 = (double)g_inbuf[prev_inbuf_wp + n].ch1 / 32768.0;

                    for (i = 1; i <= repeat; i++)
                    {
                        //過去のデータのインデックス（0からの絶対位置）
                        m = (int)((double)n - (double)i * d);
                        //現在位置(prev_inbuf_wp)からの相対位置のインデックスに変換
                        inbuf_index = prev_inbuf_wp + m;
                        if(inbuf_index < 0) inbuf_index += INBUF_SIZE;//負数になった場合の処理
                        else if(inbuf_index > INBUF_SIZE) inbuf_index -= INBUF_SIZE;
                        //入力バッファ数をオーバした場合の処理

                        /* 過去の音データをミックスする */
                        out_ch0 += pow(a, (double)i) * ((double)g_inbuf[inbuf_index].ch0 / 32768.0);
                        out_ch1 += pow(a, (double)i) * ((double)g_inbuf[inbuf_index].ch1 / 32768.0);
                    }

                    //クリッピング処理
                    pcm_tmp = clipping_data_norm(out_ch0, out_ch1);

                    //出力バッファに格納
                    put_data(pcm_tmp);
                }

                PORTE.PODR.BIT.B3 = 0;//ポートで処理時間をモニタ（処理エンド）★

                prev_inbuf_wp = g_inbuf_wp;
            }
        }
    }//while
}
```

この処理を実行した場合、256 個のデータを処理する時間（コードの★～★間）は、実測で 14.2ms となりました。

256 データの処理に許容される時間は、5.33ms ですので、このケースでは 256 個のデータを 3 回処理する内 2 回は処理できていない事（○××○××…のイメージ）となります。

次に、このコードを本機器向けに速度的なチューニングを行ってみます。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    unsigned long prev_inbuf_wp;
        (1)
    int d, n, m, i, repeat;
    double a, a0;
              (2)
    double out_ch0, out_ch1;

    pcm_data pcm_tmp;

    const int pcm_fs = PCM_FS;//サンプリングレート 48kHz

(2) a0 = 0.5;/* 減衰率 */
(1) d = (int)(pcm_fs * 0.05);/* 遅延時間 */
    repeat = 10; /* 繰り返し回数 */

    //debug

    PORTE.PDR.BIT.B3 = 1;
    PORTE.PODR.BIT.B3 = 0;

    //初期化関数
    sound_effector_init();

    prev_inbuf_wp = 0;

    while(1)
    {
        if(g_sw_state == 0)//スルー
        {
            if(prev_inbuf_wp != g_inbuf_wp)
            {
                for(n=0; n<INBUF_UNIT_SIZE; n++)
                {
                    put_data(g_inbuf[prev_inbuf_wp + n]);
                }
                prev_inbuf_wp = g_inbuf_wp;
            }
        }
        else
```

[次ページへ続く]

```
        {
        //リバーブ
        if(prev_inbuf_wp != g_inbuf_wp)//データが INBUF_UNIT_SIZE(=256) 溜まった
        {
            PORTE.PODR.BIT.B3 = 1;//ポートで処理時間をモニタ（処理スタート）★

            for(n=0; n<INBUF_UNIT_SIZE; n++)
            {

                /* 現在の時刻の音データ */
                out_ch0 = (double)g_inbuf[prev_inbuf_wp + n].ch0;//正規化を行わない
                out_ch1 = (double)g_inbuf[prev_inbuf_wp + n].ch1;     (3)

            (2)  a = a0;//減衰率の1次の係数を設定
            (1)  m = prev_inbuf_wp;//現時点のインデックスを基準（スタート）とする

                for (i = 1; i <= repeat; i++)
                {
              (1)  m -= d;//ループの度にインデックスを50ms前に
                    if(m < 0) m += INBUF_SIZE;//負数になった場合の処理

                    out_ch0 += a * ((double)g_inbuf[m].ch0);/* 過去の音データをミックスする */
                    out_ch1 += a * ((double)g_inbuf[m].ch1);      (3)

               (2)  a *= a0;//減衰率を 1/a^i に設定
                }

                //クリッピング処理
                //正規化していないデータ向けのクリッピング処理関数
                pcm_tmp = clipping_data_raw(out_ch0, out_ch1);  (3)

                //出力バッファに格納
                put_data(pcm_tmp);
                }

            PORTE.PODR.BIT.B3 = 0;//ポートで処理時間をモニタ（処理エンド）★

            prev_inbuf_wp = g_inbuf_wp;
            }
        }
    }//while
}
```

(1) このケースでは m が INBUF_INDEX を超えるケースは生じないため超えた場合の処理は記載しない

チューニング後のコードが上記です。
修正点は、主に 3 点です。

(1)ディレイの係数 d を int 型で取り扱い、過去のインデックスを示す m の計算を可能な限り repeat ループ外で行う
(2)a^{-i} の計算を pow（math ライブラリ関数）から、乗算での計算に変更
(3)PCM データの正規化を行わない

この 3 点の変更で、処理時間（★～★）が、14.2ms から 2.75ms に短縮されました。2.75ms は、5.33ms に納まっていますので、データの取りこぼしなく処理可能です。

なお、(1)~(3)のそれぞれの時間短縮効果ですが、

(1)0.9ms
(2)9.2ms
(3)1.3ms

程度となります。

このケースで、(2)では、repeat ループ内のべき乗計算を、乗算に変更しており、ループ回数が多いと誤差が蓄積されるため、数値の精度を考えると変更前の方が良いかと考えます。しかし、リアルタイム処理を行う上ではある程度の割り切りが必要かと思います。

※(2)のチューニングで、a *= a0;の 10 回目のループでは値を計算しているものの値を使用していない等、まだチューニングの余地はあります。

5.4. まとめ

プログラム移植上の注意は、

・入力バッファの構造の理解
・全て演算後に出力バッファに書き込み(put_data)
・リアルタイム処理を心がける（可能な限り効率的に）

といった点となります。

※ここで算出した処理時間は、コンパイラの最適化 OFF 設定の場合です（コンパイラの最適化を ON にするだけで、かなりの高速化が見込まれる場合もあります）

6. チュートリアル

6.1. ディレイ

SAMPLE¥RX65_SOUND_EFFECTOR_TEMPLATE

上記テンプレートは、

・Push-SW が OFF のとき、入力信号をスルーで出力
・Push-SW が ON のとき、
　VR1 が ch0 側の音量を調整
　VR2 が 1ch 側の音量を調整

というサンプルとなっています。

このテンプレートをベースに、

・Push-SW が OFF のとき、入力信号をスルーで出力
・Push-SW が ON のとき、
　VR1 に応じたディレイ（最大 2 秒）を掛ける

プログラムを作成してみます。

テンプレートを、コピーし、

[folder]¥RX65_SOUND_EFFECTOR_DELAY

というフォルダ名に変更する事とします。

RX65_SOUND_EFFECTOR.c

```
#define AD_VAL_CHANGE_THRESHOLD 3//A/D変換値がn以上変化した際に、新しい値を読み取る

void main(void)
{

    unsigned long prev_inbuf_wp;

    int i;

    pcm_data pcm_tmp;

    long delay_points;
    long delay_index;

    unsigned short volume1_val=0;//VR1の読み取り値（AD_VAL_CHANGE_THRESHOLD以下の揺らぎは反映させない）
    unsigned short mute=0;//VR1変化中は出力をミュートとする

    //初期化関数
    sound_effector_init();

    sound_effector_start();//動作開始

    prev_inbuf_wp = 0;

    while(1)
    {
        if(g_sw_state == 0)
        {
            //スルー
            if(prev_inbuf_wp != g_inbuf_wp)
            {
                for(i = 0; i < INBUF_UNIT_SIZE; i++)//未処理データが INBUF_UNIT_SIZE 溜まった
                {
                    put_data(g_inbuf[prev_inbuf_wp + i]);//入力バッファの値を未加工で出力バッファに格納
                }
                prev_inbuf_wp = g_inbuf_wp;//入力バッファのポインタ値を保存
            }
        }
        else
        {
            //ディレイ
            if(prev_inbuf_wp != g_inbuf_wp)//未処理データが INBUF_UNIT_SIZE 溜まった
            {

                //ボリューム値読み取り
                if( ((volume1_val + AD_VAL_CHANGE_THRESHOLD) <= g_ad_val[0]) || ((volume1_val - AD_VAL_CHANGE_THRESHOLD) >= g_ad_val[0]) )
                {
                    //g_ad_val[0] が AD_VAL_CHANGE_THRESHOLD 以上変化した場合
                    volume1_val = g_ad_val[0];
                    mute = 20;//出力の音量を下げる（20周期、5.33ms x 20 = 約0.1秒）
                }
                else
                {
                    //ボリュームの変化がない場合は、徐々にミュートを解除
                    if(mute != 0) mute--;
                }
```

スルーの部分は、テンプレートから変更なし

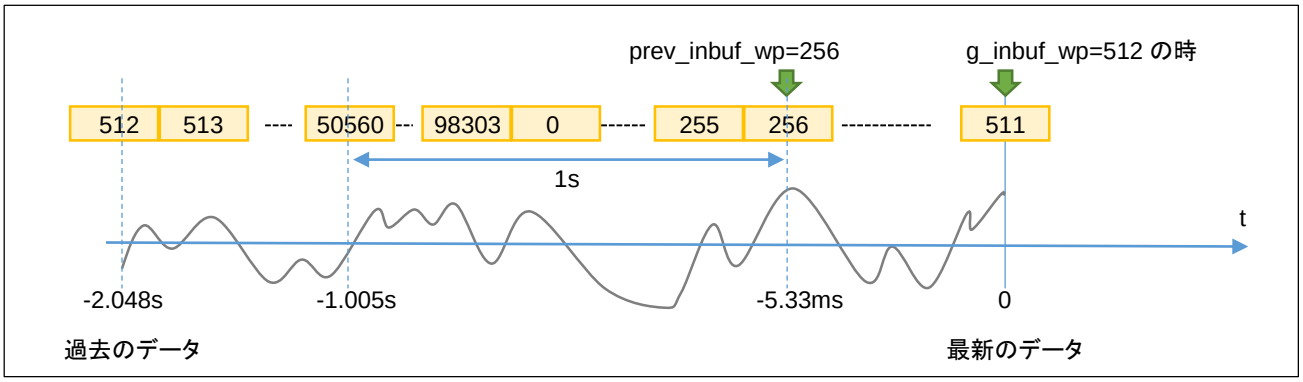
volume1_val = 0~4095, VR1 に応じた値
PCM_FS=48000

```
                //ディレイ値の算出（最大2秒）
                delay_points = (long)(((float)volume1_val / 4096.0) * (float)PCM_FS *
2.0);

                delay_index = prev_inbuf_wp - delay_points;//ディレイ量(g_delay_points)に応
じたインデックスを計算，prev_inbuf_wp を基準に delay_points 分過去のデータにアクセス
                if(delay_index < 0) delay_index += INBUF_SIZE;

                for(i = 0; i < INBUF_UNIT_SIZE; i++)
                {

                    if(mute == 0)
                    {
                        put_data(g_inbuf[delay_index++]);//通常
                    }
                    else
                    {
                        //ボリューム調整中はプツプツ音を防ぐため、音量をミュートする（ボリュームが変化し
ない場合は徐々に音量を回復させる）
                        pcm_tmp.ch0 = g_inbuf[delay_index].ch0 / mute;
                        pcm_tmp.ch1 = g_inbuf[delay_index].ch1 / mute;
                        delay_index++;
                        put_data(pcm_tmp);
                    }

                    if(delay_index >= INBUF_SIZE) delay_index=0;//INBUF_UNIT_SIZE が
INBUF_SIZE の 1/n（n:整数）になっている場合は不要なはずであるが一応
                }

                prev_inbuf_wp = g_inbuf_wp;
            }
        }
    }//while
}
```

本プログラムは、ディレイを掛けるもので、入力バッファ（最大約 2 秒）に蓄えられている過去のデータを出力するサンプルです。

過去のデータにアクセスする手法は、フィルタのプログラムを作成する際に必ず必要となりますので、g_inbuf の構成を理解する事が重要となります。

読み込み単位(INBUF_UNIT_SIZE)が 256（デフォルト）の時を考えます。



g_inbuf_wp が 512 に変化した際、prev_inbuf_wp は 256 となっています。

このとき、g_inbuf[512]が一番古いデータです。

prev_inbuf_wp を基準にして、1 秒前のデータを取得する場合、インデックスは 48000（1 秒あたりのサンプリングレート）前のものとなります。

256 - 48000 = -47744

計算結果が、負なので、INBUF_SIZE(=98304)を加算します。

-47744 + 98304 = 50560

g_inbuf[50560] が、prev_index_wp（現時点より、5.33ms 前）を基準にした場合の、1 秒前のデータとなります。

ここで、アクセスを行う、g_inbuf のインデックスに関して注意すべき点が 2 点あります。

(1)-2.048 秒近傍のデータにアクセス

上図の時点では、g_inbuf[512]が一番古いデータとなりますが、20.83us 後には、g_inbuf[512]は最新のデータで上書きされます。一番古いデータは、順次上書きされていきますので、アクセスするタイミングには注意が必要です。

※この例で、一番古いデータは、(prev_inbuf_wp)でない事に注意ください

一番古いデータは、g_inbuf_wp です

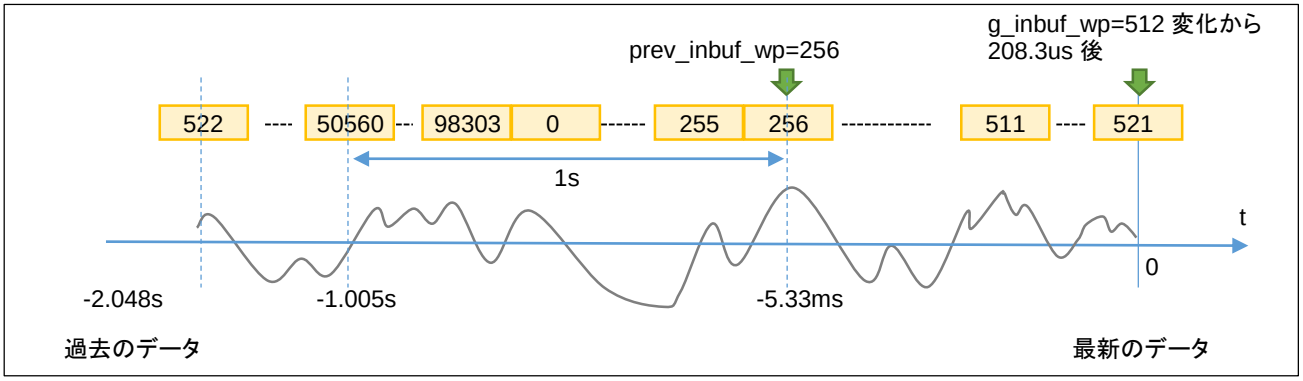
(2)g_inbuf_wp を基点とするデータアクセス

g_inbuf_wp-1 までは、データの格納が完了しています。この時点で、prev_index_wp と g_inbuf_wp のどちらを基点にすべきかですが、g_inbuf_wp を基点にデータにアクセスした場合、時系列的におかしな事となる事が考えられます。

g_inbuf_wp が 512 に変化後、208.3us 後にデータアクセスする場合で、下記の様にデータにアクセスした場合、

```
for(i=0; i<INBUF_UNIT_SIZE; i++)
{
   g_inbuf[g_inbuf_wp + i];
}
```

※INBUF_UNIT_SIZE=256



g_inbuf[g_inbuf_wp + 0] ~ g_inbuf[g_inbuf_wp + 9] (g_inbuf[512] ~ g_inbuf[521])
と
g_inbuf[g_inbuf_wp + 10] ~ (g_inbuf[522] ~)

は、時間的につながらないデータとなります。

```
for(i=0; i<INBUF_UNIT_SIZE; i++)
{
   g_inbuf[prev_index_wp + i];
}
```

であれば、g_inbuf[prev_index_wp + 0] ~ g_inbuf[prev_index_wp + 255]は、時系列的につながっているデータとなります。

g_inbuf は、常に更新される変数であることを意識して、[ ]内のインデックスの値を決めてください。

6.2. IIR フィルタ(LPF)

SAMPLE¥RX65_SOUND_EFFECTOR_TEMPLATE

上記テンプレートをベースに、

・Push-SW が OFF のとき、入力信号をスルーで出力
・Push-SW が ON のとき、
　1kHz の LPF として働く

プログラムを作成してみます。

テンプレートを、コピーし、

[folder]¥RX65_SOUND_EFFECTOR_FILTER

というフォルダ名に変更する事とします。

RX65_SOUND_EFFECTOR.c

```
//インクルード
//#include "typedefine.h"
#include <machine.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mathf.h>
#include <complex.h>
#include "sound_effector.h"
#include "iir_filter.h"

void main(void)
{

    unsigned long prev_inbuf_wp;

    int n, m, Ix, J, L;

    //<math.h>をインクルードすると（正しくは<complex.h>内で定義）、"I" -> "((float
_Imaginary)((float _Complex)__I__))"にマクロ展開されるので、
    //オリジナルソースの変数名"I"を"Ix"に変更しています

    double fc, Q, a[3], b[3], *x_ch0, *x_ch1, *y_ch0, *y_ch1;

    pcm_data pcm_tmp;

    long inbuf_index, outbuf_index;

    unsigned short i;

    fc = 1000.0/ (double)PCM_FS;/* 遮断周波数 (PCM_FS = 48000)*/
    Q = 1.0/ sqrt(2.0);/* クオリティファクタ */
    Ix = 2;/* 遅延器の数 */
    J = 2;/* 遅延器の数 */

    IIR_LPF(fc, Q, a, b);/* IIRフィルタの設計 */

    L = INBUF_UNIT_SIZE;/* フレームの長さ = 256 */

    x_ch0 = calloc((L + J), sizeof(double)); /* メモリの確保 */
    x_ch1 = calloc((L + J), sizeof(double)); /* メモリの確保 */
    y_ch0 = calloc((L + Ix), sizeof(double)); /* メモリの確保 */
    y_ch1 = calloc((L + Ix), sizeof(double)); /* メモリの確保 */

    if((x_ch0 == NULL) || (x_ch1 == NULL) || (y_ch0 == NULL) || (y_ch1 == NULL))
    {
        //メモリの確保に失敗
        while(1);
    }

    //初期化関数
    sound_effector_init();

    //動作開始
    sound_effector_start();

    prev_inbuf_wp = 0;
```

iir_filter.h は、参考文献のサポートサイト(http://floor13.sakura.ne.jp)からダウンロードしたものです

```
    while(1)
    {
        if(g_sw_state == 0)//スルー
        {
            //スルー
            if(prev_inbuf_wp != g_inbuf_wp)
            {
                for(i=0; i<INBUF_UNIT_SIZE; i++)
                {
                    put_data(g_inbuf[prev_inbuf_wp + i]);
                }
                prev_inbuf_wp = g_inbuf_wp;
            }
        }
        else
        {
            //LPF
            if(prev_inbuf_wp != g_inbuf_wp)//データが INBUF_UNIT_SIZE(=256) 溜まった
            {

                /* 直前のフレームの後半のJサンプルをつけ加える */
                for(n = 0; n < L + J; n++)
                {
                    inbuf_index = prev_inbuf_wp - J + n;
                    if(inbuf_index < 0) inbuf_index += INBUF_SIZE;
                    else if(inbuf_index >= INBUF_SIZE) INBUF_INDEX -= INBUF_SIZE;

                    x_ch0[n] = (double)g_inbuf[inbuf_index].ch0 / 32768.0;
                    x_ch1[n] = (double)g_inbuf[inbuf_index].ch1 / 32768.0;
                }

                /* 直前のフィルタリング結果の後半のIサンプルをつけ加える */
                for (n = 0; n < Ix; n++)
                {
                    outbuf_index = g_outbuf_wp - Ix + n;
                    if(outbuf_index < 0) outbuf_index += OUTBUF_SIZE;
                    else if(outbuf_index >= OUTBUF_SIZE) OUTBUF_INDEX -= OUTBUF_SIZE;

                    y_ch0[n] = (double)g_outbuf[outbuf_index].ch0 / 32768.0;
                    y_ch1[n] = (double)g_outbuf[outbuf_index].ch1 / 32768.0;
                }

                for(n = 0; n < L; n++)
                {
                    y_ch0[n+Ix] = 0;
                    y_ch1[n+Ix] = 0;
                }

                /* フィルタリング */
                for (n = 0; n < L; n++)
                {
                    for (m = 0; m <= J; m++)
                    {
                        y_ch0[Ix + n] += b[m] * x_ch0[J + n - m];
                        y_ch1[Ix + n] += b[m] * x_ch1[J + n - m];
                    }
                    for (m = 1; m <= Ix; m++)
                    {
                        y_ch0[Ix + n] += -a[m] * y_ch0[Ix + n - m];
                        y_ch1[Ix + n] += -a[m] * y_ch1[Ix + n - m];
                    }
                }
```

スルーの部分は、テンプレートから変更なし

g_inbuf は直前のフレームのデータは残っているので、prev_inbuf_wp より古いデータにアクセスしても問題がない

g_outbuf は
書き込みアクセス：データが未出力の保証なし
読み出しアクセス：現時点より 2048 データであれば過去のデータが残っている事が保証される

y の残りの部分は、0 を初期データとする

```
                /* フィルタリング結果の連結 */
                for (n = 0; n < L; n++)
                {
                    //クリッピング処理
                    pcm_tmp = clipping_data_norm(y_ch0[Ix + n], y_ch1[Ix + n]);

                    //出力バッファに格納
                    put_data(pcm_tmp);
                }

                prev_inbuf_wp = g_inbuf_wp;
            }
        }

    }//while
}
```

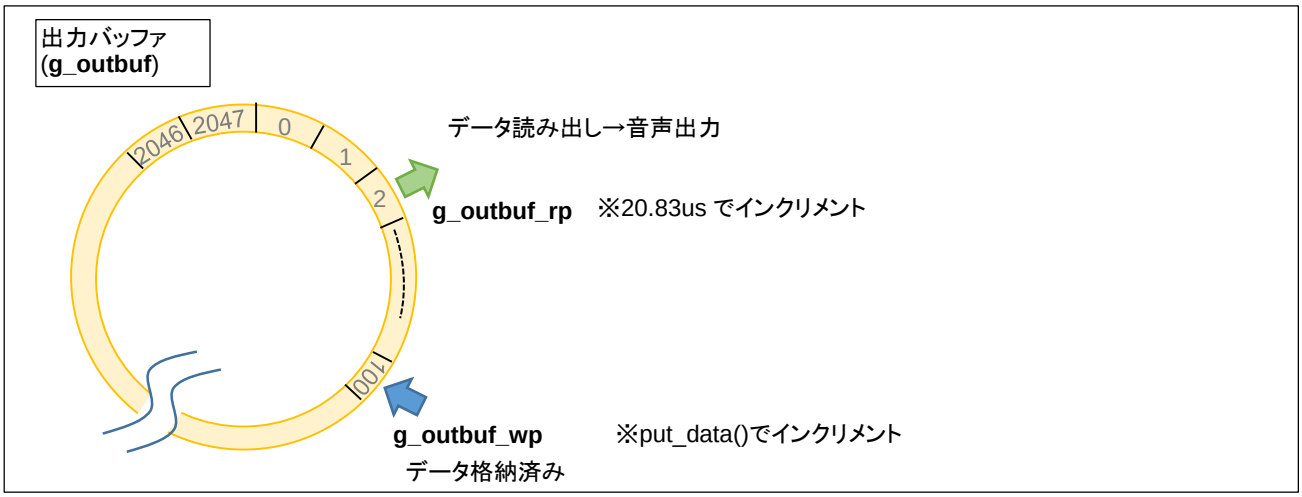
本プログラムは、参考文献「C 言語ではじめる音のプログラミング」の 6 章
ex6_4.c
を参考にしていますので、参照頂きたく。

http://floor13.sakura.ne.jp/book03/book03.html
からダウンロードできる
chapter06.zip
を展開し、

chapter06¥ex6_4¥iir_filter.h

を本プログラムではインクルードしています。

このプログラムでは、g_outbuf の値を参照しています。g_outbuf にデータを書き込む際は、データ格納関数
put_data()
を使用します。



put_data() で、g_outbuf にデータ格納後、どのタイミングで読み出されて音声出力されるかは、出力バッファにどのぐらいデータが溜まっているかで決まります。場合によっては、格納後時間を空けずに読み出されて、音声出力されるケースもあり得ます。

put_data() でデータ格納後に、g_outbuf の値を書き換える事は、出力前であれば可能ですが、非推奨です。

※出力データが確定していないのであれば、一旦ワーク変数に格納し、書き換え、データ確定後に、put_data() で、g_outbuf に格納してください。

それに対し、g_outbuf に格納されているデータの読み取りアクセスに関しては、行って頂いて構いません。g_outbuf には、デフォルトで 2048 個のデータが格納されていますので、インデックス g_outbuf_wp-1（最新データ）を基準に、過去に格納（出力）したデータを参照する事は問題ありません。

※g_outbuf_wp は、格納済みデータ位置+1 を示しています

IIR フィルタのプログラムでは、（出力バッファに格納した）直前のフィルタリングの結果の、Ix サンプル（プログラムでは 2 サンプル）のデータを計算に使用しています。

7. 関数、グローバル変数、定数仕様

sound_effector.c, sound_effector.h 内で定義されている、関数、変数に関して仕様を示します。

7.1. 関数

sound_effector_init

概要:初期化関数
宣言:
　　void sound_effector_init(void)
説明:
　　各種変数の初期化を行います
（出力バッファ g_outbuf にデータ格納後に、本関数を呼ぶと、データがクリアされますので注意ください）
引数:
　　なし
戻り値:
　　なし

sound_effector_start

概要:動作開始関数
宣言:
　　void sound_effector_start(void)
説明:
　　各種タイマのスタート、オーディオ CODEC との通信動作を開始します、基本的には各種変数等の初期化後、メインループ開始前に実行してください
引数:
　　なし
戻り値:
　　なし

put_data

概要:データ出力バッファ格納関数

宣言:

int put_data(short data)　[出力 1ch 向け]

int put_data(pcm_data data)　[出力 2ch 向け]

説明:

FIFO 構成の出力バッファに対してデータを格納する処理を行います（バッファモード）

引数[出力 1ch 向け]:

short data　音声データ（16bitPCM データ）

引数[出力 2ch 向け]:

pcm_data data　音声データ（16bitPCM データ, 2ch）

戻り値:

0: 出力バッファにデータ格納する処理が成功

1: 出力バッファがフルのため、データ格納処理が失敗

clipping_data_norm

概要:正規化データのクリッピング処理関数

宣言:

short clipping_data_norm(float data)　[出力 1ch 向け]

pcm_data clipping_data_norm(float ch0_data, float ch1_data)　[出力 2ch 向け]

説明:

正規化（-1~1 の範囲）されたデータのクリッピング処理（-1~1 の範囲から外れるデータを-1, 1 にクリッピング)を行い、32768 を乗じた値を返します

引数[出力 1ch 向け]:

float data　音声データ（16bitPCM データ）

引数[出力 2ch 向け]:

float ch0_data　ch0 音声データ（16bitPCM データ）

float ch1_data　ch1 音声データ（16bitPCM データ）

戻り値[出力 1ch 向け]:

short 型　16bitPCM データ

戻り値[出力 2ch 向け]:

pcm_data 構造体　16bitPCM データ, 2ch （put_data() の引数とすることを想定）

clipping_data_raw

概要:PCM データのクリッピング処理関数

宣言:

short clipping_data_raw(float data)　[出力 1ch 向け]

pcm_data clipping_data_raw(float ch0_data, float ch1_data)　[出力 2ch 向け]

説明:

PCM 生データ(-32768~32767 の範囲)のクリッピング処理(-1~1 の範囲から外れるデータを-1, 1 にクリッピング)を行います

(引数は、float 型である事に注意、正規化の処理を省略したい場合に使用する関数です)

引数[出力 1ch 向け]:

float data　音声データ(16bitPCM データ)

引数[出力 2ch 向け]:

float ch0_data　ch0 音声データ(16bitPCM データ)

float ch1_data　ch1 音声データ(16bitPCM データ)

戻り値[出力 1ch 向け]:

short 型　16bitPCM データ

戻り値[出力 2ch 向け]:

pcm_data 構造体　16bitPCM データ, 2ch (put_data の引数とすることを想定)

outbuf_spool_data_num

概要:出力バッファのデータ格納数取得関数

宣言:

unsigned long outbuf_spool_data_num(vold)

説明:

出力バッファに格納済みで、未出力のデータ数を取得します

引数:

なし

戻り値:

unsigned long　データ数

0 : 出力バッファに溜まっているデータなし

7.2. グローバル変数

・ユーザがアクセスする事を想定している変数
pcm_data g_inbuf[ ]

入力バッファ変数。入力端子から入力された音声が PCM データに変換されたものが格納される変数。16bit の PCM データ(-32768~32768)の値を格納する。

unsigned long g_inbuf_wp

入力バッファの、書き込み位置を示す変数。INBUF_UNIT_SIZE 単位でインクリメントされる。
※正確には、g_inbuf_wp -1 までデータ格納済みで、g_inbuf_wp は次にデータを格納するインデックス

unsigned short g_outbuf[ ] [出力 1ch 向け]
pcm_data g_outbuf[ ] [出力 2ch 向け]

出力バッファ変数。音声出力を行いたいデータを格納する変数。16bit の PCM データ(-32768~32768)の値を格納する。put_data() 関数を用いてデータの格納を行う。

unsigned long g_outbuf_rp

出力バッファの、読み込み位置を示す変数。20.83us(1/48kHz)毎にインクリメントされる。

unsigned long g_outbuf_wp

出力バッファの、書き込み位置を示す変数。put_data() 関数実行時にインクリメントされる。
※正確には、g_outbuf_wp -1 までデータ格納済みで、g_outbuf_wp は次にデータを格納するインデックス

unsigned short g_zero_data [出力 1ch 向け]
pcm_data g_zero_data [出力 2ch 向け]

ゼロデータ。無音データを出力バッファに格納したい場合、put_data() の引数として指定する。
put_data(g_zero_data);
※値を書き換える事は可能となっています

unsigned short g_sampling_period_flag

20.83us(=1/48kHz)毎にクリア（0 を代入）される変数。プログラム中で、20.83us 毎に行いたい処理がある場合に使用。

unsigned long g_sampling_counter

20.83us(=1/48kHz)毎インクリメントされる変数。プログラム中で、参照する事により、時間差（サンプリング周期数）を把握する事が可能。（最大値は、2^{32}-1 です。最大値に達した後（約 50 日）は、再度 0 からカウントアップします）

unsigned char g_sw_state

Push-SW の状態が格納される変数。スイッチが押されていない時は 1。押されているときは 0 が入ります。50ms 毎に更新されます。

unsigned short g_ad_val[3]

ボリュームの状態が格納される変数。ボリュームを反時計回りに目一杯回した際、0。時計回りに目一杯回した場合、4095 の値が入ります。g_ad_val[0]が VR1 に対応し、g_ad_val[1], g_ad_val[2]が、VR2, VR3 に対応します。50ms 毎に更新されます。

7.3. 定数定義

#define DEBUG

定義時、デバッグ用の変数を有効化する。

#define DEBUG2

定義時、詳細なデバッグ用の変数を有効化する。

#define DEBUG_PORT

定義時、汎用 I/O ポートを使用したデバッグを有効化する。

#define OUT_CH (1)

出力に使用するチャンネル数を指定。
(1): 出力を 1ch 使用（ch0 と ch1 からは同じ音が出力される）
(2): 出力を 2ch 使用
のいずれかが定義可能です。

#define INBUF_UNIT_SIZE (256)

データ読み込み単位。g_inbuf_wp が、この定数単位でインクリメントされる。
（g_inbuf への格納は、20.83us(1/48kHz)周期で行われ、g_inbuf_wp は、20.83us × INBUF_UNIT_SIZE の周期で、INBUF_UNIT_SIZE 分インクリメントされます）

#define OUTBUF_SIZE (2048)

出力バッファサイズ。一度にまとめて 2048 個以上のデータを出力バッファに書き込む必要がある場合は、この値を増加させるか、put_data() の戻り値を観測しながらデータの格納を行う必要があります。

#define CLIPPING_NORM_MAX (0.9999)
#define CLIPPING_NORM_MIN (-0.9999)

クリッピング処理で使用する、最大、最小値です（正規化値）。データがこの値から外れた場合は、この値にクリッピングされます。clipping_data_norm() 関数で使用。

#define CLIPPING_NORM_RAW (32767)
#define CLIPPING_NORM_RAW (-32768)

クリッピング処理で使用する、最大、最小値です。データがこの値から外れた場合は、この値にクリッピングされます。clipping_data_raw() 関数で使用。

#define M_PI (3.14159265358979)

円周率の値です。

以下の値は、定義値の変更は不可です（参照は可）。

#define INBUF_SIZE (98304) [2ch]
#define INBUF_SIZE (196608) [1ch]

入力バッファサイズ。

#define PCM_FS (48000)

サンプリング周波数。

8. まとめ

本マニュアルでは、音を加工するケースを説明致しました。

入力データを用い、音を加工する事ができれば、本機器の使用バリエーションも大きく拡がりますので、音を加工する手法を活用して頂きたいと思います。

9. 付録

取扱説明書改定記録

バージョン	発行日	ページ	改定内容
REV.1.0.0.0	2019.11.30	—	初版発行

お問合せ窓口

最新情報については弊社ホームページをご活用ください。

ご不明点は弊社サポート窓口までお問合せください。

株式会社 北斗電子

〒060-0042 札幌市中央区大通西 16 丁目 3 番地 7

TEL 011-640-8800　FAX 011-640-8801

e-mail：support@hokutodenshi.co.jp (サポート用)、order@hokutodenshi.co.jp (ご注文用)

URL:http://www.hokutodenshi.co.jp

サウンドエフェクタ　ソフトウェア編(2)
取扱説明書

株式会社 北斗電子

 2019 年 11 月 30 日改訂 REV.1.0.0.0 (191130)