サウンドエフェクタ
ソフトウェア編(1)
～波形を生成してみる～
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[本書]　～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　～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　～デバッグ及び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。

本資料では、サイン波やのこぎり波等の波形を生成して出力する手法を説明します。

ベースとなるテンプレートは、

RX65_SOUND_EFFECTOR_TEMPLATE_NOINPUT

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

を参考にして作成を行っています。

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。

開発環境に関して

サウンドエフェクタでのプログラム開発は、ルネサスエレクトロニクス RX マイコンがターゲットとなります。

・統合開発環境

CS+

e2Studio

統合開発環境としては、ルネサスエレクトロニクスでは 2 種類用意されており、本書での説明では CS+を使用しています。e2studio は eclipse ベースのツールとなっており、eclipse の環境に慣れている方は、e2studio を使用する事も可能です。（どちらのツールも、ルネサスエレクトロニクスの Web よりダウンロード可能です。※要ユーザ登録）

・コンパイラ（リンカ）

CC-RX（無償版）

CC-RX（有償版）

GNURX

コンパイラ、リンカは、CS+の場合には CC-RX が同梱されています。e2studio の場合は、別途インストールする必要があります。

CC-RX（無償版）では、評価期間（60 日）経過後は、プログラムのリンクサイズが 128kB(max)に制限されます。

本製品に採用している、RX651 マイコンは、ROM サイズが 2MB あり、128kB 以上のプログラムを作成する場合は、有償版のコンパイラか、GNURX を使用する必要があります。

但し、出荷時に書き込まれているデモプログラムのサイズは、16kB 程度であり、余程大規模なプログラムを作成しない限りは、CC-RX の無償版の制約に掛かる事はないものと考えます。

・プログラム書き込みツール

RenesasFlashProgrammer

開発したプログラムを、装置に書き込む際、RenesasFlashProgrammer（以下 RFP）を使用します。RFP は、無償評価版と有償版がありますが、趣味で使用する分には、無償評価版で問題ないと考えます（書込みが行えるサイズ等に関して、無償評価版故の制約はありません）。

・デバッガ

ルネサス E1（生産終了、市場在庫のみ）

ルネサス E2 Lite

ルネサス E2, E20（趣味で使用する場合、高価な装置となります）

凝ったプログラムの作成にチャレンジする方は、デバッガを用意する事を推奨します。E2 Lite であれば、8,000 円程度で購入できます。

また、ルネサス RX マイコンのプログラムに慣れていない方は、「スマート・コンフィグレータ」を使用する事を推奨致します（製品付属のサンプルプログラムでも使用しています）。スマート・コンフィグレータは、マイコンを動かすために必要なコードを自動生成してくれるツールです。GUI でパラメータを入力すると、パラメータに応じたソースコードが生成されますので、プログラムを一から作成する場合に対して、大幅に時間を節約できます。

プログラムの開発にあたり、必要なツールとしまして、ルネサス Web (https://www.renesas.com/jp/ja/) より

【無償評価版】統合開発環境 CS+ for CC V8.xx.xx（一括ダウンロード版）（2019/10 現在 V8.02.00）
または
【無償評価版】統合開発環境 CS+ for CC V8.xx.xx（分割ダウンロード版）1~n

CS+ RX スマート・コンフィグレータ通信プラグイン V1.xx.xx　（2019/10 現在 V1.02.05）
RX スマートコンフィグレータ V2.xx.xx　（2019/10 現在 V2.2.1）

を予め、ダウンロード、インストールを行っておいてください。

マイコンのリソースに関して

サウンドエフェクタで使用してるマイコンは、ルネサスエレクトロニクス社の RX65 グループのマイコンで、

・最大動作周波数　120MHz（本製品では、98.304MHz で駆動）
・ROM（フラッシュ）　2MB
・RAM(SRAM)　640kB（256kB+384kB）
・FPU 内蔵
・DSP 内蔵

となっています。

RAM に関して、256kB 側をワークメモリとして使用。384kB を、音声データのバッファとして割り当てると、

384kB / (16bit(2bytes) x 48kHz x 2ch) = 2.048 [s]

となり、2ch でそれぞれ約 2 秒分のデータをバッファ可能です。

1ch のバッファとして利用すると、単純に倍の約 4 秒（4.096[s]）のデータをバッファ可能です。

（256kB 側のメモリもバッファに割り当てる等の対応を行えば、もっと長時間のデータをバッファする事も可能です。）

RX651 は、組み込み用のマイコンとしては大容量の RAM を搭載しているのですが、PC に比べると使用可能なメモリや演算リソースは限られます。よって、プログラムを作成する際は、「マイコン」のリソースの限界を意識する必要があります。（特に、普段 PC 上でプログラミングを行っている方は、ご注意ください。）

ビルド時に出力されるファイルに関して

マイコン上で動作するプログラムは、C(C++)言語、(RX マイコン向け)アセンブリ言語で記載しますが、コンパイル、ビルドを行った結果出力されるのは、

MOT ファイル(拡張子.mot)
ABS ファイル(拡張子.abs)

となります。(ルネサスのコンパイラ CC-RX を用いた場合のデフォルト)

MOT ファイルは、モトローラ S 形式と呼ばれる汎用フォーマットのファイルです。

「ヘッダーアドレスーデーターチェックサム」の順に、hex のデータがテキスト形式で並んでおり、マイコンの ROM にプログラムを書き込む際には、このファイルを使用します。

CS+の環境では、デフォルトでは

DefaultBuild

フォルダの下に出力されますので、RenesasFlashProgrammer を使用してサウンドエフェクタに、プログラムの書き込みを行う際は、このファイルを使用してください。

ABS ファイルは、アブソリュートファイルと呼ばれるファイルで、マシン語のプログラムとデバッグ情報が含まれたバイナリ形式のファイルです。デバッガを用いてデバッグを行う際は、このファイルが使用されます。

その他、拡張子が.map のファイルはマップファイルというもので、プログラムコードや変数のアドレスマップの情報が出力されているファイルです。変数が RAM の領域から溢れている等の情報が拾えますので、必要に応じて参照頂きたく。

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_NOINPUT

以下の、プロジェクトをベースに説明を行っています。

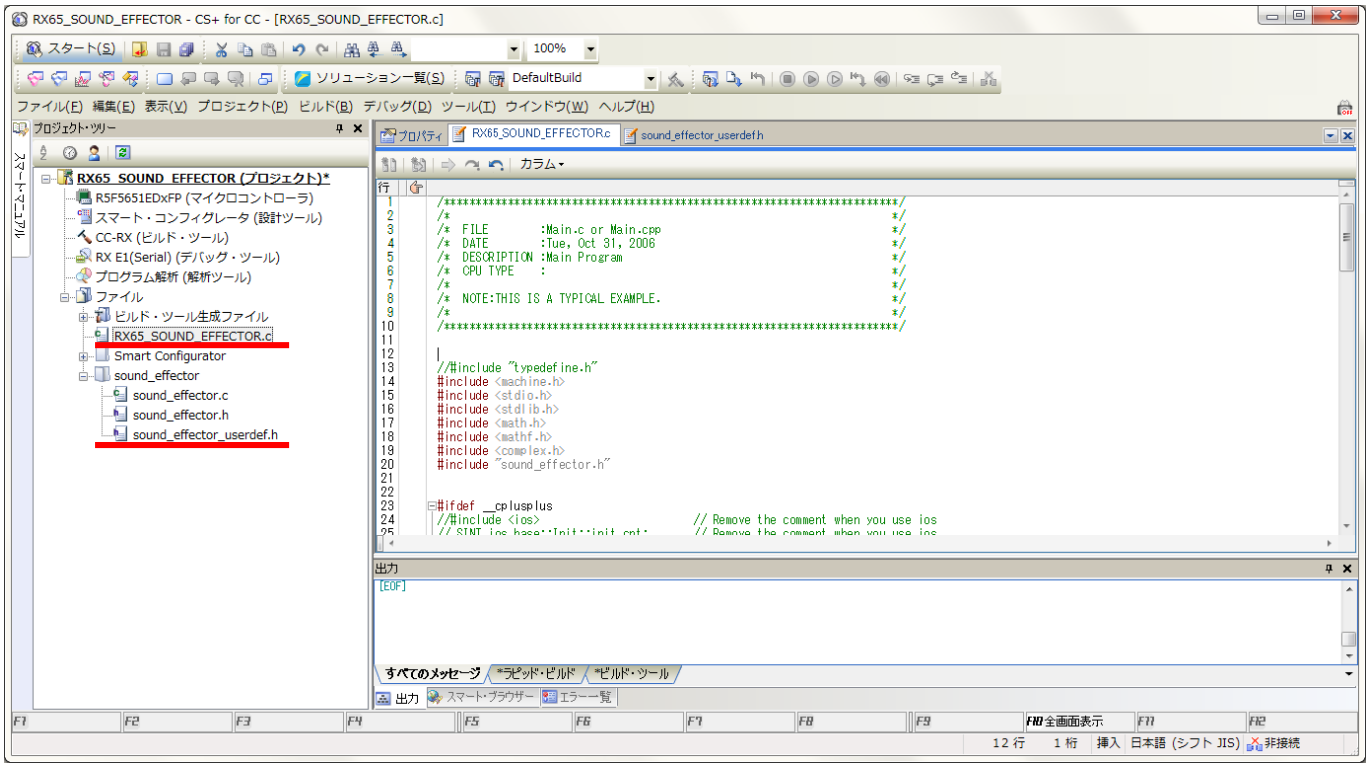
上記を、PC 内のストレージにコピーし、変更を行いながら動作を確認頂きたく。

ツールの起動とビルド

[path]¥RX65_SOUND_EFFECTOR_TEMPLATE_NOINPUT¥RX65_SOUND_EFFECTOR.mtpj

をダブルクリックして、CS+forCC を起動してください。（拡張子の関連付けがなされていない場合は、「開発環境に関して」の章を参照し、必要なツール（CS+forCC）をインストールしてください。

CS+が起動すると、下記の様画面となります。



このプロジェクトは、色々なプログラムを作成するテンプレートとなっています。

基本的に、新規に入力端子を使用しないプロジェクトを作成する場合は、このテンプレートをコピーして使用してください。

CS+は、左側にプロジェクトツリーが表示されており、

ファイル

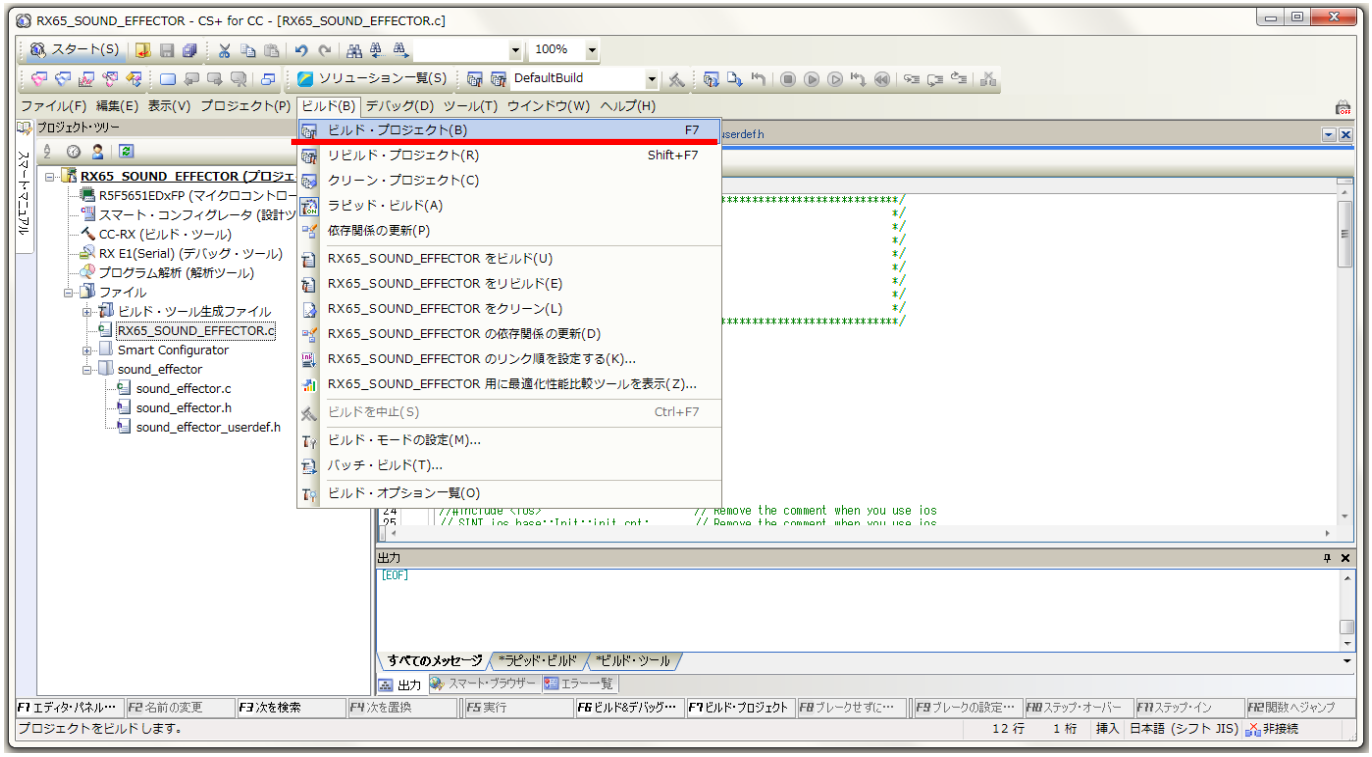
ツリーの下に、ファイルが表示されます。

まずは、

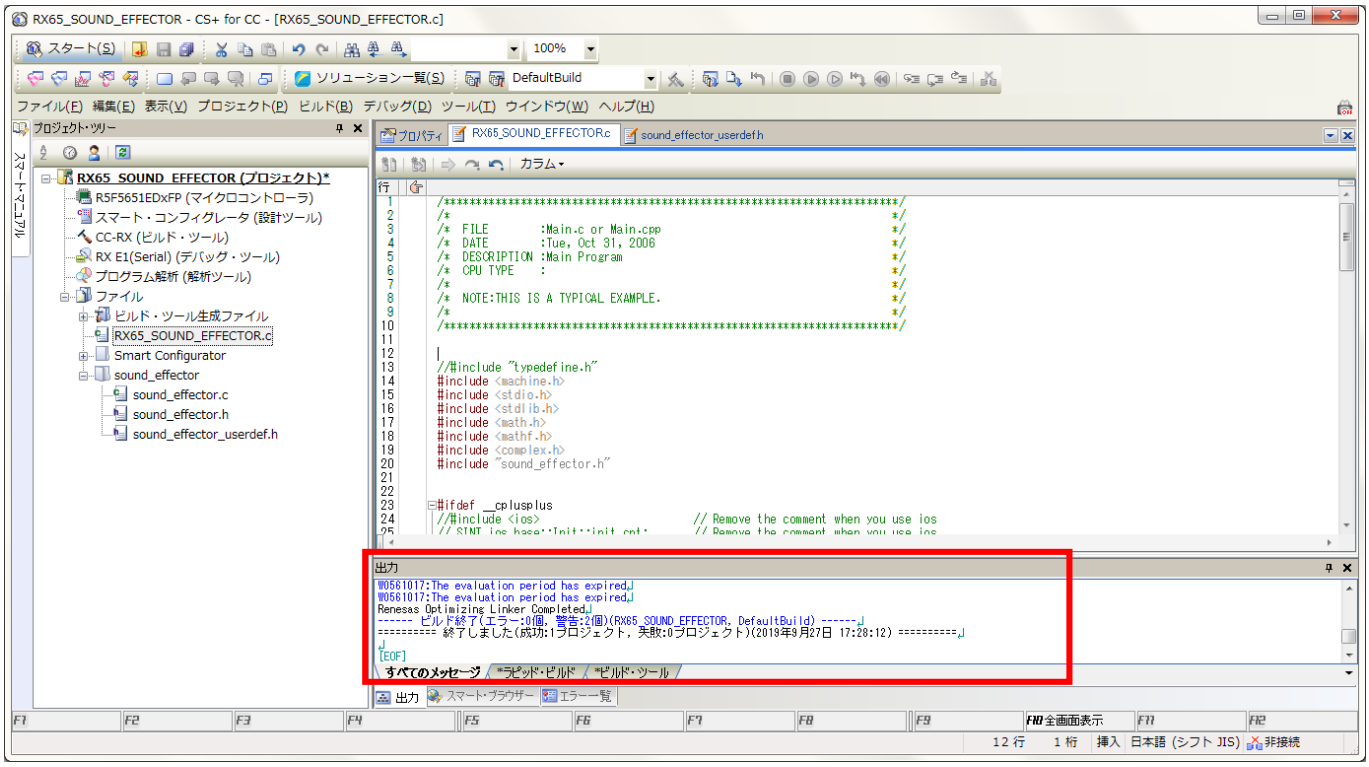
RX65_SOUND_EFFECTOR.c　メイン関数を含むプログラムの処理
sound_effector_userdef.h　プログラムの挙動を変える定義値を定義

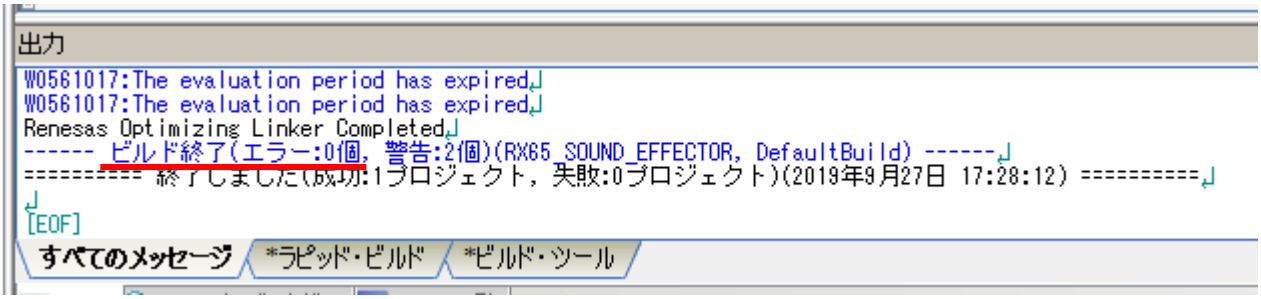
の 2 つのファイルのみ、変更する様にしてください。他のファイルは、通信やバッファの処理を行っているもので、現時点ではブラックボックスとお考え頂きたく。

ファイルの変更後、ビルド（ソースファイルのコンパイル＆リンク）は、メニューの
ビルドービルド・プロジェクト（もしくは、F7 を押す）



で行ってください。





エラーが無ければ、ビルドは成功しています。
（警告は、内容を確認して、問題有無を判断してください）
※「W0561017:The evaluation period has expired」の警告は、無償評価版を 60 日以上使用している場合に出力される警告で、無視しても問題ありません

ビルド結果のファイル（サウンドエフェクタに書き込むファイル）は、

[path]¥RX65_SOUND_EFFECTOR_TEMPLATE_NOINPUT¥DefaultBuild¥RX65_SOUND_EFFECTOR.mot

となります。（テンプレートを変更しなければ、33kB 程度のファイルです）

実際に実機で動作させるためには、この mot ファイルを取扱説明書に記載の手順でサウンドエフェクタに書き込みを行ってください。

1. マイコンでの処理

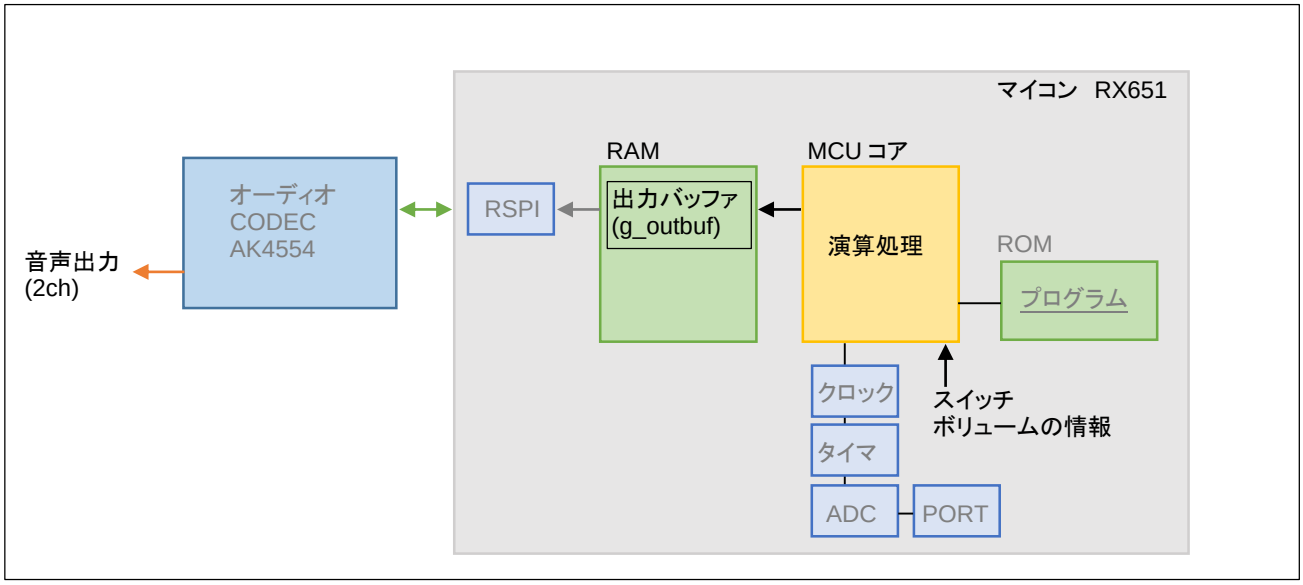


図 1-1 マイコン処理系

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, 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。

・出力バッファ(g_outbuf)

マイコンで生成した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

波形データを演算、出力バッファに格納する処理を行う部分です。

2. 動作モード

テンプレートのプログラムでは、動作モードを 3 種類用意しています。

(1)リピートモード

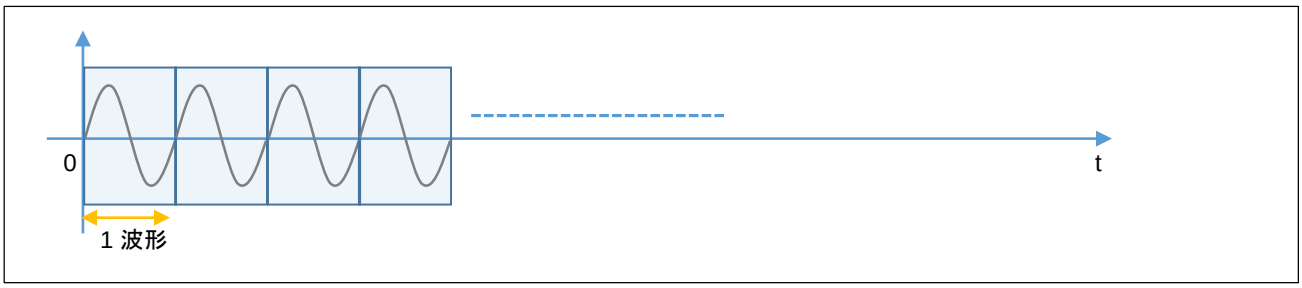


図 2-1 リピートモード

1 つの波形を繰り返し出力するモードです。

1 波形の長さは、2ch 使用する場合最大 2.048 秒（1ch 使用する場合は、最大 4.096 秒）で、繰り返し周期は任意の長さを指定できます。

(2)ワンショットモード

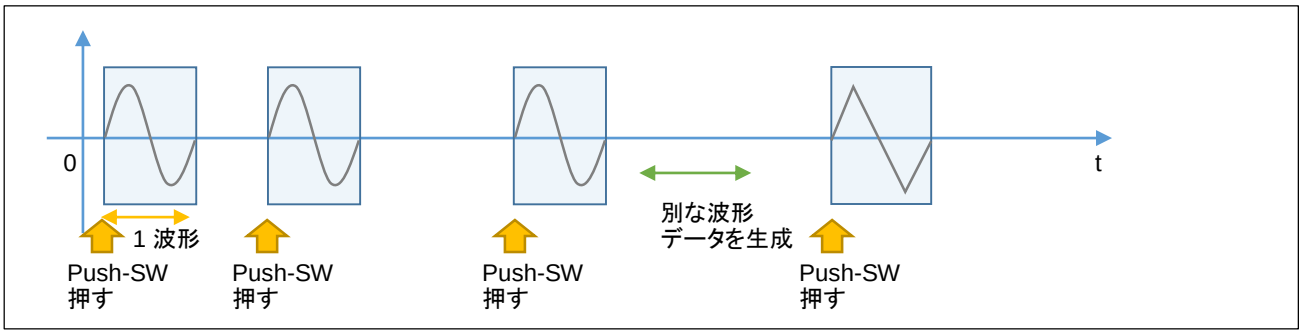


図 2-2 ワンショットモード

Push-SW を押す度に 1 波形を出力するモードです。1 回出力すると、音データの出力は停止します。

1 波形の長さは、2ch 使用する場合最大 2.048 秒（1ch 使用する場合は、最大 4.096 秒）です。1 波形の長さは任意の値を指定できます。

同じ波形の繰り返しの他、途中で別なデータを生成して、波形を変える事も可能です。

※リピートモードにおいても、ボリュームや、Push-SW の状態を読み取って動作中に波形を変える事が可能です。

(3)バッファモード

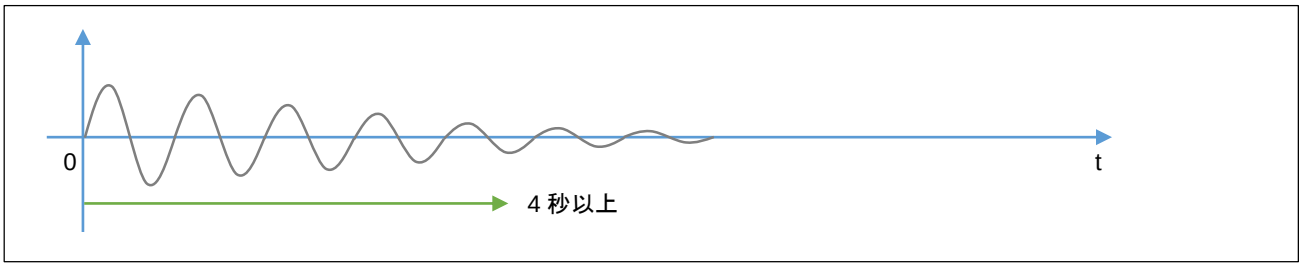


図 2-3 バッファモード

任意の波形をリアルタイムで生成し、バッファに格納。バッファに格納されたデータが、順次出力されるモードです。（図では、長周期の減衰するサイン波の例としています。リピート、ワンショットモードでは、4 秒を超える長さの波形を出力する事ができませんが、バッファモードですと、出力バッファに入りきらない長い周期の波形でも出力可能です。）
※バッファモードでも、Push-SW やボリュームの値を演算に使用する事が可能です
※出力を途切れさせない様にするためには、平均で 1 データ／20.83us のレートでデータを出力バッファに入れていく必要があります

プログラムでは、(1)~(3)の 3 種の動作モードから 1 つを選択する形となります。

3. 出力バッファ

プログラムでは、波形データを作成し出力バッファに格納する事により、本機器からは音が出力されます。

本機器には、2 つの出力端子がありますが、1ch を使用するモードに設定すると、ch0, ch1 の出力端子から同じ音が出力されます。2ch を使用するモードとすると、ch0, ch1 の出力端子から別々な音を出力可能です。

出力バッファは、g_outbuf という名称の配列変数として定義されており、1ch 使用時と 2ch 使用時で構造が異なります。

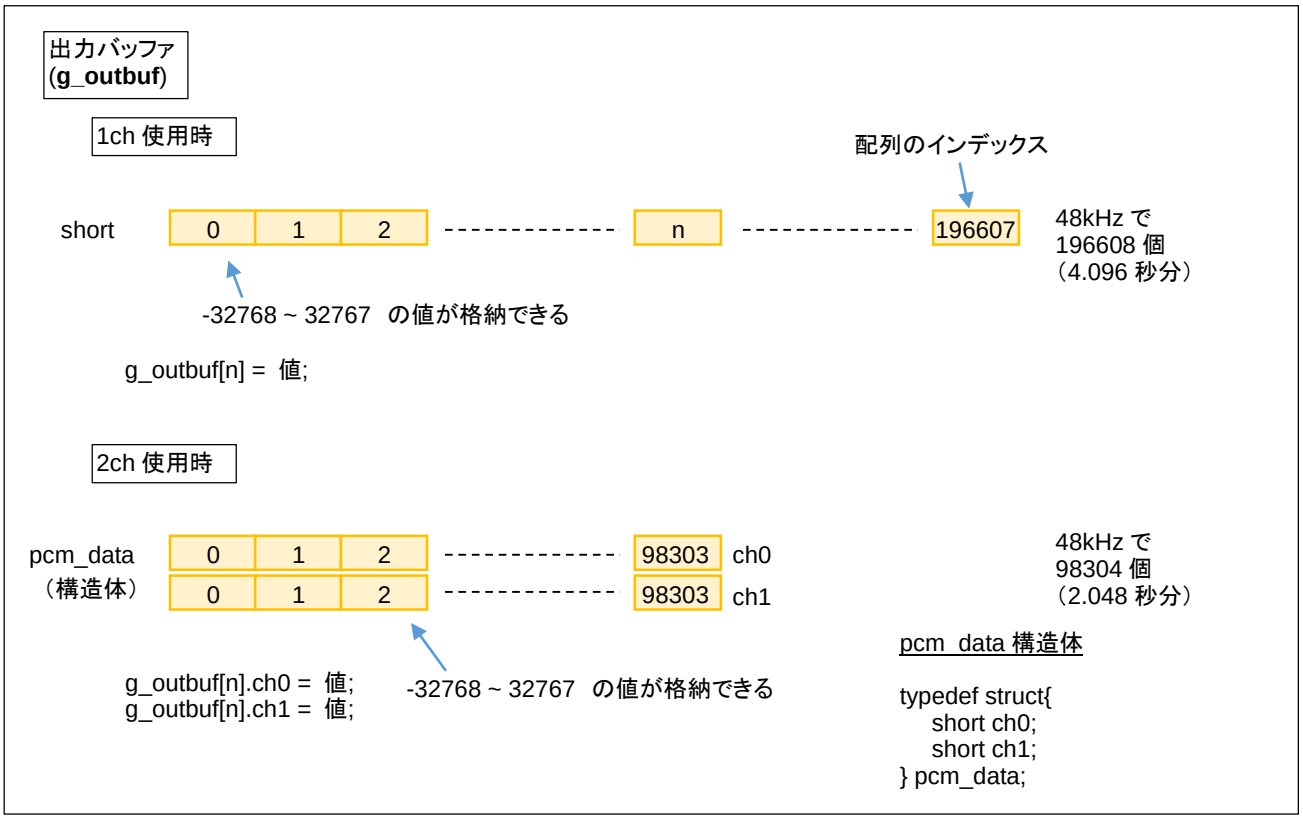


図 3-1 バッファ構成(1)

1ch 使用時は、g_outbuf は short 型の配列変数となります。

2ch 使用時は、pcm_data 構造体（メンバ、ch0, ch1）となります。

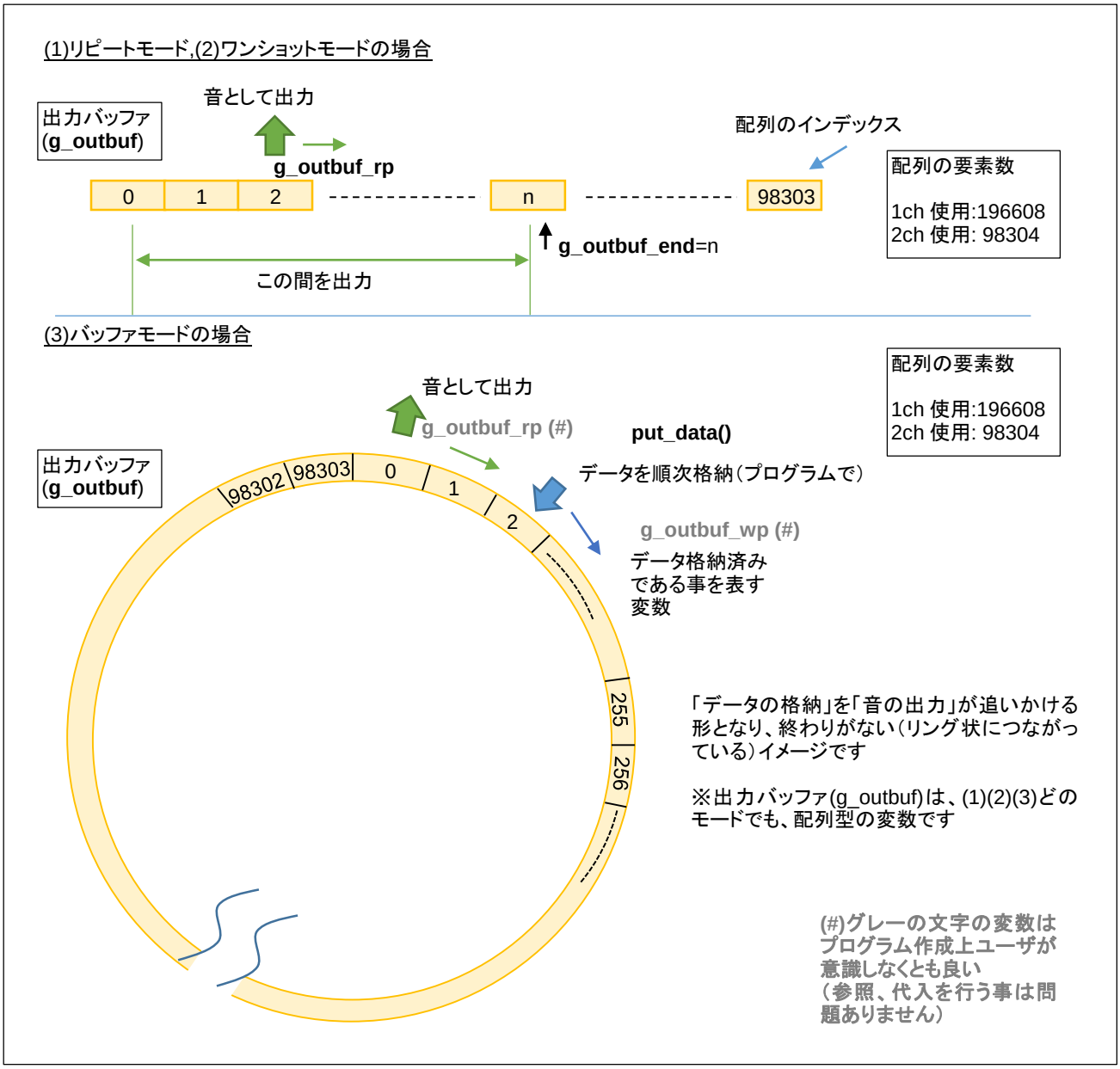


図 3-2 バッファ構成(2)

動作モードが変わっても、出力バッファ自体は変わりませんが、出力バッファの使用方法が多少変わってきます。

(1)(2)の場合は、データを出力バッファに格納後、データの終わりの点を、

g_outbuf_end

変数に設定します。プログラムでは、0~g_outbuf_end の区間のデータを出力します。

現在、出力しているデータの要素（インデックス）は、

g_outbuf_rp

の値のデータです。(1)のリピートモードでは、g_outbuf_rp は、0 からインクリメントされていき、g_outbuf_end まで達すると、再び 0 に戻り、データ出力を継続します。

(2)のワンショットモードでは、g_outbuf_rp は、0 からインクリメントされていき g_outbuf_end+1 で止まります。g_outbuf_rp のインクリメントが止まった後は、無音データが出力されます。音データの出力終了後（または、出力中）に、g_outbuf_rp=0 を代入すると、再びデータの先頭から音が出力されます。

(3)のバッファモードでは、出力バッファ(g_outbuf)は、(1)(2)同様、配列型の変数ですが、FIFO（First In First Out, 先入れ、先出し方式）バッファとして動作します。
プログラムとしては、put_data() 関数を使用し、データを出力バッファにコピーする使い方となります。出力バッファから、音の出力のところはバックグラウンドで処理され、20.83us(1/48kHz)に 1 データずつ出力されていきます。
（図 3-2 の緑の矢印（出力バッファからの読み取り）は、20.83us 毎に時計回りに自動的に進んでいきます）
データのコピーが間に合わない場合、（FIFO バッファが空になった場合＝緑の矢印が青の矢印に追いついた場合）無音データが出力されます。
FIFO バッファの長さは、2ch 使用時 2.048 秒、1ch 使用時 4.096 秒です。
※どこまで読み取った（音の出力を行った）かを示す変数 g_outbuf_rp と、どこまで出力バッファに格納したかを示す変数 g_outbuf_wp は、プログラム作成上意識しなくとも良い
（g_outbuf_rp はバックグラウンドで勝手に進んでいく変数で、g_outbuf_wp は put_data() 関数でデータ格納時にインクリメントされる変数です）

※g_outbuf_rp　g_outbuf 向けの read pointer の意味合いで_rp という変数名としています
※g_outbuf_wp　g_outbuf 向けの write pointer の意味合いで_wp という変数名としています

4. スイッチとボリュームの情報の読み取り

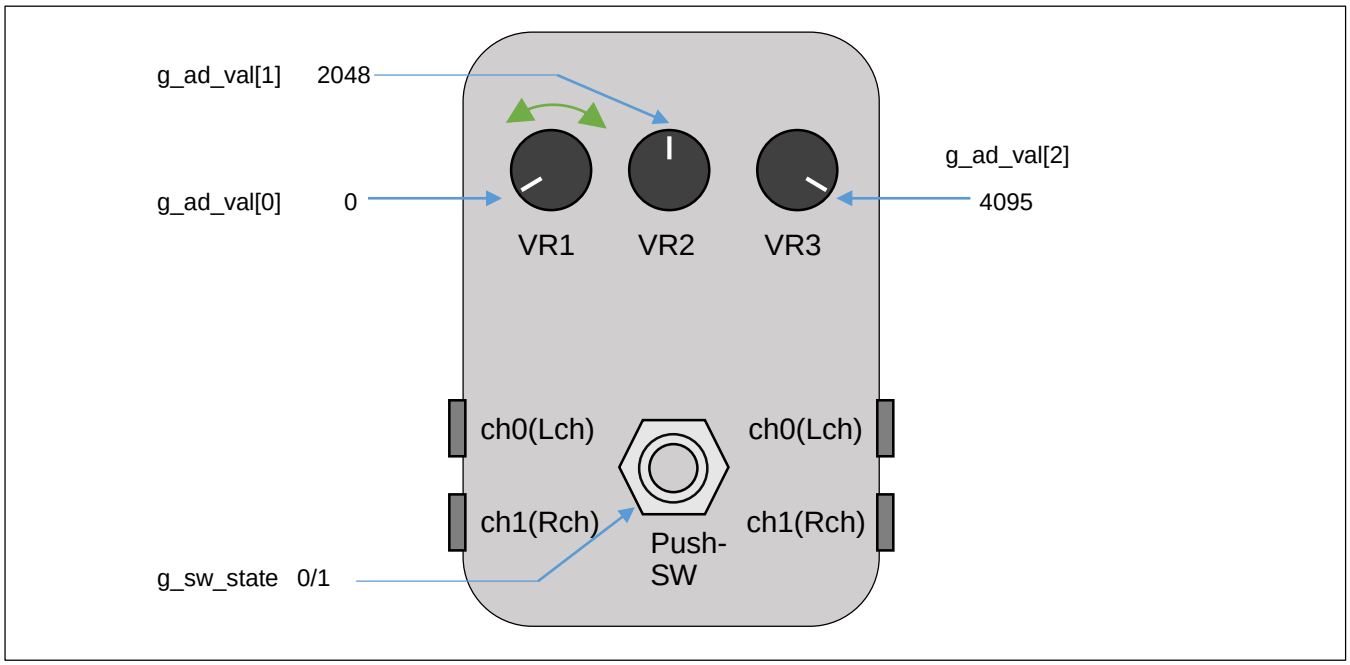


図 4-1 スイッチとボリューム

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)

5. プログラムの流れ

実際に、どの様にプログラムを作成するかを示します。

・使用チャンネル数の決定

2ch の出力から別々なデータを出力するか、同じデータを ch0, ch1 から出力するかを選びます。

・動作モードの決定

2 章の 3 種類の動作モードからどれを選ぶかを決定します。

(1)~(3)の動作モードと使用チャネル数は、ユーザ定義ファイル(sound_effector_userdef.h)で設定します。

src¥user¥sound_effector¥sound_effector_userdef.h

```
//出力ch
#define OUT_CH (1)//"1"（ch0, ch1に同じ信号を出力）または"2"（ch0, ch1に別々の信号を出力）を定義
//※コンパイルオプション、実行中にch数の変更は不可


//動作モード//"REPEAT(=0)", "ONCE(1)", "BUFFER(2)"のいずれかを定義
#define OUT_MODE REPEAT//REPEAT: 0~g_outbuf_end間を繰り返し出力
//#define OUT_MODE ONCE//ONCE: Push-SWを押すたびに0~g_outbuf_end間を1回出力
//#define OUT_MODE BUFFER//BUFFER: 出力バッファをFIFOで使用するモード
//※コンパイルオプション、実行中に動作モードを切り替える事は不可
```

定義値

OUT_CH
を
(1) ：1ch 使用（ch0, ch1 の出力端子から同じデータが出力される）
(2) ：2ch 使用（ch0, ch1 から別々のデータを出力可）
のいずれかに設定してください。

1ch 使用の際は、g_outbuf は、g_outbuf[0] ~ g_outbuf[196607]まで使用可能です。
2ch 使用の際は、g_outbuf は、g_outbuf[0] ~ g_outbuf[98303]まで使用可能です。

OUT_MODE

を、
REPEAT　:リピートモード
ONCE　　:ワンショットモード
BUFFER　:バッファモード

のいずれかに設定してください。

これらの値は、コンパイル時に、コードの選択コンパイルに利用されますので、実行中の動作モードの変更はできません。(変更後は、プログラムの再コンパイルが必要になります)

5.1. プログラム本体(メイン関数)

(1)リピートモード

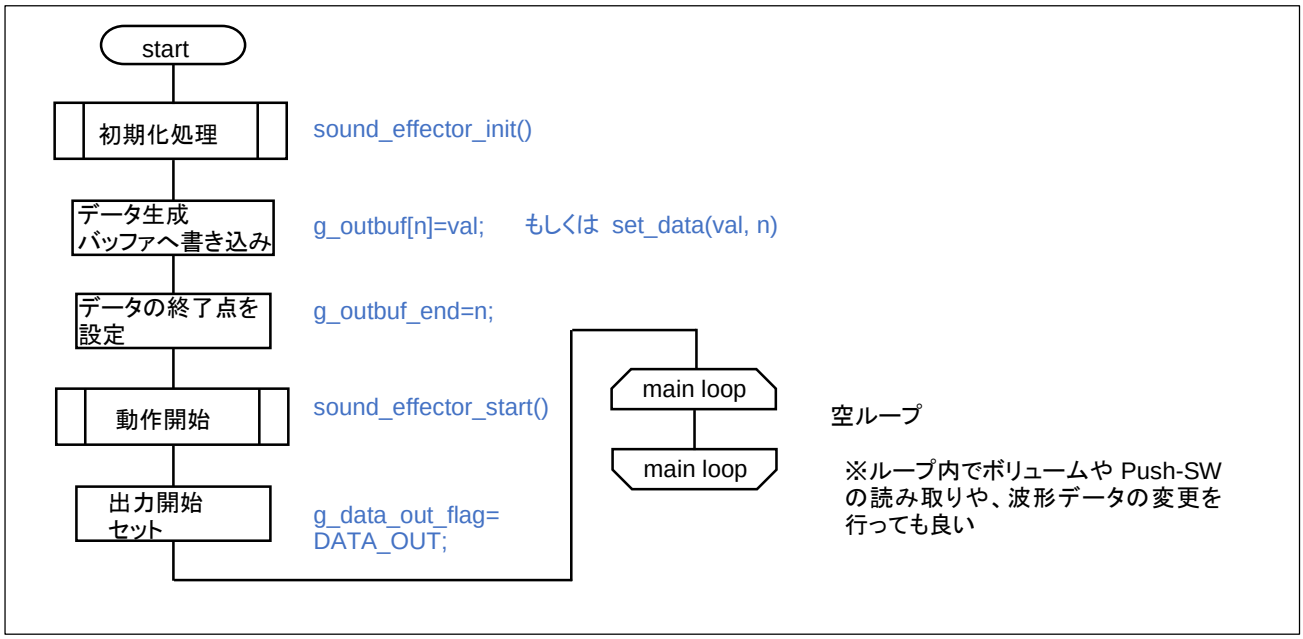


図 5-1 リピートモード フローチャート

リピートモードでの、メイン関数の処理例は上記となります。

・g_outbuf への出力データ設定
・データの終了点の設定

後、本機器の動作を開始(sound_effector_start() を呼び出す)し、g_data_out_flag を DATA_OUT に設定し、音の出力を開始します。

g_data_out_flag は、音の出力を制御する変数で、

g_data_out_flag = DATA_OUT_HOLD;　出力を行わない（データ格納中等）
g_data_out_flag = DATA_OUT;　出力を行う

となります。データ準備中や、データの差し替え時に、DATA_OUT_HOLD にセットすると、音の出力を無音にできます。

(2)ワンショットモード

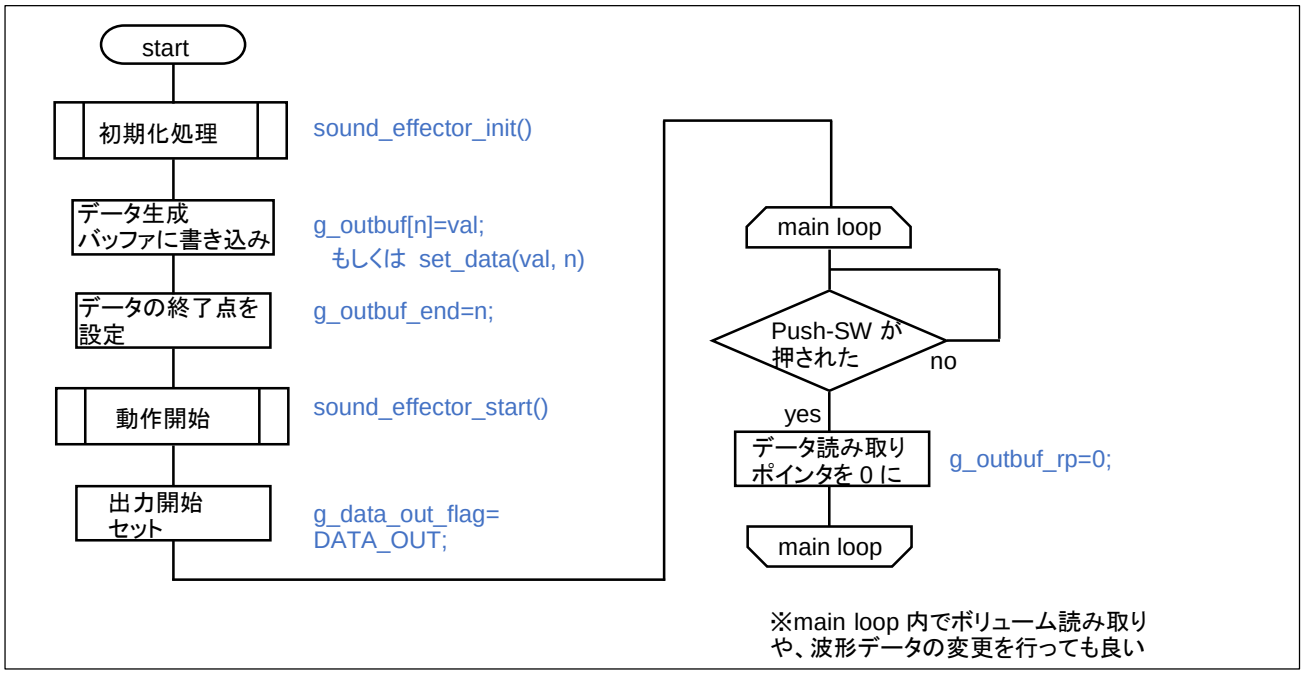


図 5-2 ワンショットモード フローチャート

ワンショットモードでは、メインループ内に、スイッチの読み取りのコードをユーザ側で記載する必要があります。具体的な記載例は、後述します。

(3)バッファモード

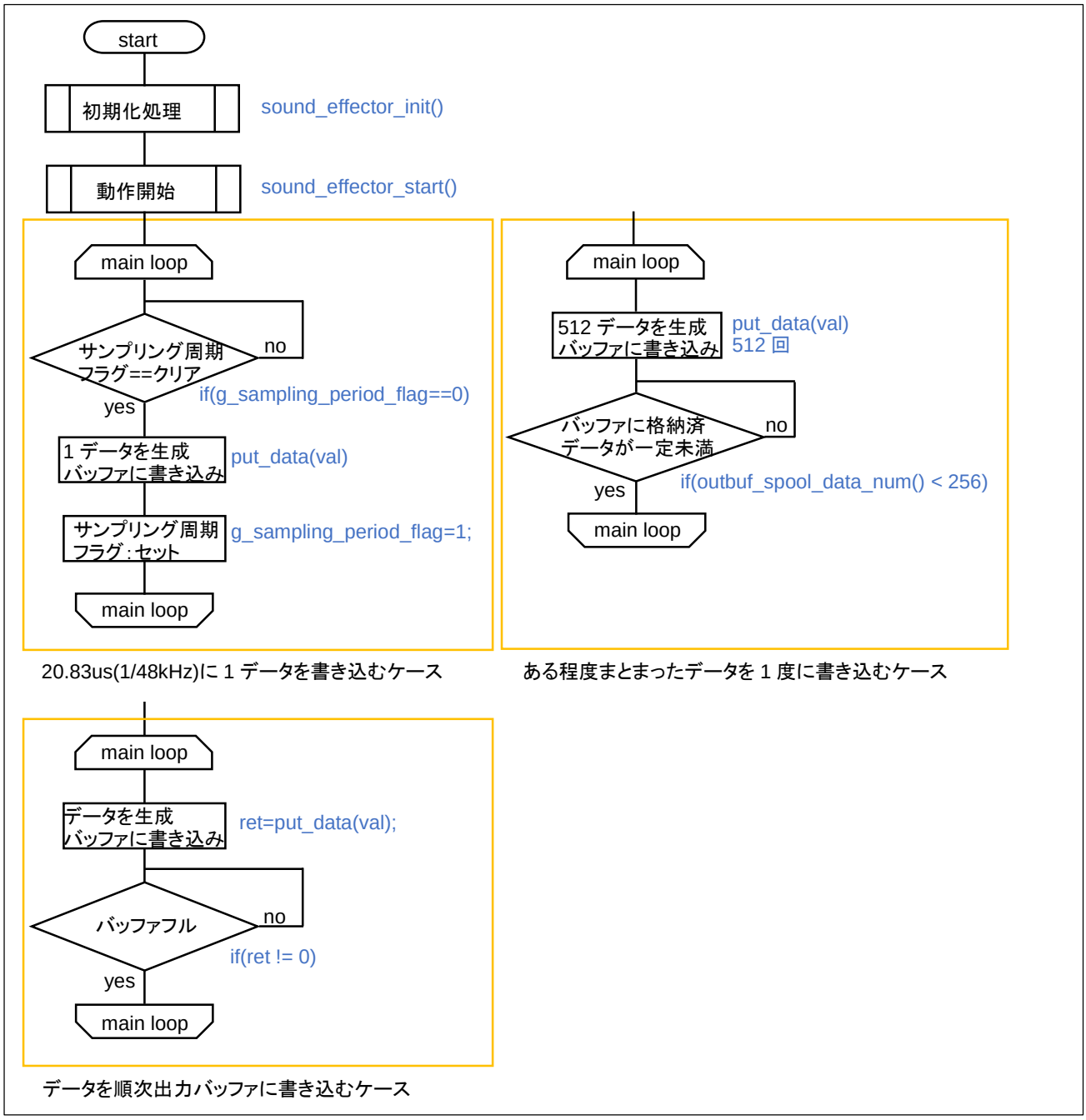


図 5-3 バッファモード フローチャート

バッファモードでは、

・波形をリアルタイムで生成し出力

・波形データを演算、順次出力バッファに格納して出力

等、の動作を行わせる事ができます。

音の出力部はバックグラウンドで処理されていて、出力バッファに格納されているデータが 20.83us に 1 データずつ読み出されて、出力されていきます。

FIFO 構成の出力バッファが空になった場合（書き込んだデータが全て出力されてしまった場合）は、無音が出力されます。

※「出力バッファへ格納」、「出力バッファが空になる」を繰り返すと、プツプツ音が生じる事となります

データのリアルタイム生成では、平均 20.83us に 1 データを出力バッファに書き込むことができれば、音の出力が途切れる事はありません。

6. チュートリアル

6.1. リピートモード

TEMPLATE¥RX65_SOUND_EFFECTOR_TEMPLATE_NOINPUT

上記テンプレートは、

・1ch 使用
・リピートモード
・440Hz のサイン波を出力

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、

・リピートモード
・「サイン波」「のこぎり波」「矩形波」「三角波」「無音」を出力する

プログラムを作成してみます。

※Push-SW を押す度に出力される波形が変わるプログラムで、順番が判る様、5 回目のシーケンスでは無音出力とします

テンプレートを、コピーし、

[folder]¥RX65_SOUND_EFFECTOR_WAVE_SEQUENCE

というフォルダ名に変更する事とします。

[folder]¥RX65_SOUND_EFFECTOR_WAVE_SEQUENCE¥RX65_SOUND_EFFECTOR.c

が、メイン関数を含むファイルで、今回テンプレートからの変更は、上記ファイルのみです。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    int n, i;
    int triangle_phase;

    float gain;

    int wave_sequence=0;//0:サイン波, 1:のこぎり波, 2:矩形波, 3:三角波, 4:無音

    unsigned char prev_sw_state;


    float *sin_table;//サイン波のテーブル
    float *saw_table;//のこぎり波のテーブル
    float *rect_table;//矩形波のテーブル
    float *triangle_table;//三角波のテーブル
    unsigned short table_size=(unsigned short)(PCM_FS / 440);//440Hzのテーブルを作成
（440Hz, 1周期分とする）

    //初期化関数
    sound_effector_init();

    //メモリの確保
    sin_table = calloc(table_size, sizeof(float));
    saw_table = calloc(table_size, sizeof(float));
    rect_table = calloc(table_size, sizeof(float));
    triangle_table = calloc(table_size, sizeof(float));

    if(sin_table == NULL || saw_table == NULL || rect_table == NULL || triangle_table
== NULL)
    {
        while(1);
        //メモリの確保に失敗
    }

    //サイン波の生成
    for(n=0; n<table_size; n++)
    {
        sin_table[n] = sinf(2.0 * M_PI * (float)n / (float)table_size);
    }

    gain = 0.5;
    for(n=0; n<table_size; n++)
    {
        sin_table[n] *= gain;
    }
```

table size -> 109 となります

0~108 の 109 点のデータで波形を構成

1 周期分のみ波形データを生成し繰り返し出力させます

生成された-1 ~ 1 の波形の振幅を 1/2 とします
（他の波形で、-1~1 を超える場合があるため、一律 1/2 とする）

```
    //ノコギリ波の生成
    for(i=1; i<=48; i++)
    {
        for(n=0; n<table_size; n++)
        {
            saw_table[n] += 1.0 / (float)i * sinf(2.0 * M_PI * (float)i * (float)n /
(float)table_size);
        }
    }

    gain = 0.5;
    for(n=0; n<table_size; n++)
    {
        saw_table[n] *= gain;
    }

    //矩形波の生成
    for(i=1; i<=48; i+=2)
    {
        for(n=0; n<table_size; n++)
        {
            rect_table[n] += 1.0 / (float)i * sinf(2.0 * M_PI * (float)i * (float)n /
(float)table_size);
        }
    }

    gain = 0.5;
    for(n=0; n<table_size; n++)
    {
        rect_table[n] *= gain;
    }

    //三角波形の生成
    triangle_phase=0;
    for(i=1; i<=48; i+=2)
    {
    for(n=0; n<table_size; n++)
    {
        if(triangle_phase % 2 == 0)//i=3,7,11,...では位相を180度ずらした（反転させた）波形を重
ね合わせる
            {
                triangle_table[n] += 1.0 / (float)i / (float)i * sinf(2.0 * M_PI *
(float)i * (float)n / (float)table_size);
            }
            else
            {
                triangle_table[n] -= 1.0 / (float)i / (float)i * sinf(2.0 * M_PI *
(float)i * (float)n / (float)table_size);
            }
        }
        triangle_phase++;
    }

    gain = 0.5;
    for(n=0; n<table_size; n++)
    {
        triangle_table[n] *= gain;
    }

    g_outbuf_end=table_size-1;//リピート周期を設定（0～table_size-1まで出力されます）

    sound_effector_start();//動作開始

    prev_sw_state = g_sw_state;//Push-SWの状態を保存
```

サイン波以外の波形は、サイン波の高調波の重ね合わせで生成しています

ノコギリ波の計算式です

矩形波の計算式です

三角波の計算式です

出力の周期を設定

オーディオ CODEC にデータ送出を開始

```
    while(1)
    {
        if(prev_sw_state != g_sw_state)
        {
            //Push-SWが押された

            g_data_out_flag=DATA_OUT_HOLD;//出力停止
            g_outbuf_rp = 0;//出力開始点を0に

            switch(wave_sequence)
            {
                case 0:
                    for(n=0; n<table_size; n++)
                    {
                        g_outbuf[n] = clipping_data_norm(sin_table[n]);//出力バッファに出力値
をセット
                    }
                    break;

                case 1:
                    for(n=0; n<table_size; n++)
                    {
                        g_outbuf[n] = clipping_data_norm(saw_table[n]);//出力バッファに出力値
をセット
                    }
                    break;

                case 2:
                    for(n=0; n<table_size; n++)
                    {
                        g_outbuf[n] = clipping_data_norm(rect_table[n]);//出力バッファに出力値
をセット
                    }
                    break;

                case 3:
                    for(n=0; n<table_size; n++)
                    {
                        g_outbuf[n] = clipping_data_norm(triangle_table[n]);//出力バッファに
出力値をセット
                    }
                    break;

                case 4:
                    for(n=0; n<table_size; n++)
                    {
                        g_outbuf[n] = 0;//無音データ
                    }
                    break;
            }

            g_data_out_flag=DATA_OUT;//出力スタート

            wave_sequence++;
            if(wave_sequence >= 5) wave_sequence = 0;

            prev_sw_state = g_sw_state;//スイッチの状態を保存

            }

            nop();

        }//while
}
```

Push-SW の状態が変化した

サイン波のデータを出力バッファにコピー

のこぎり波のデータを出力バッファにコピー

矩形波のデータを出力バッファにコピー

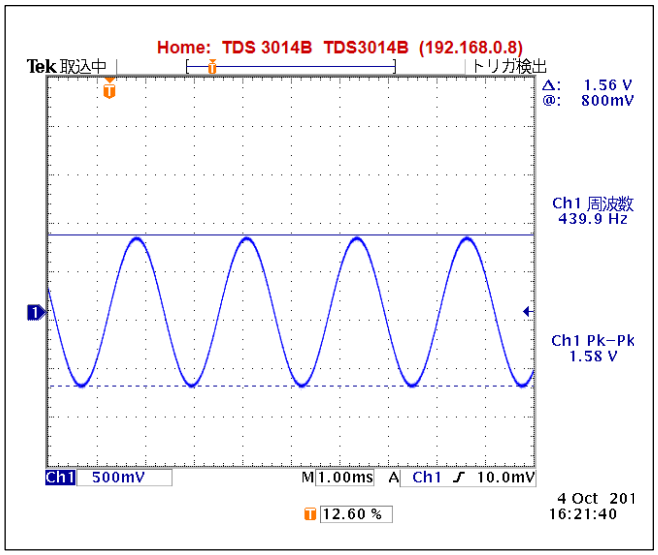
三角波のデータを出力バッファにコピー

出力される音を無音とする

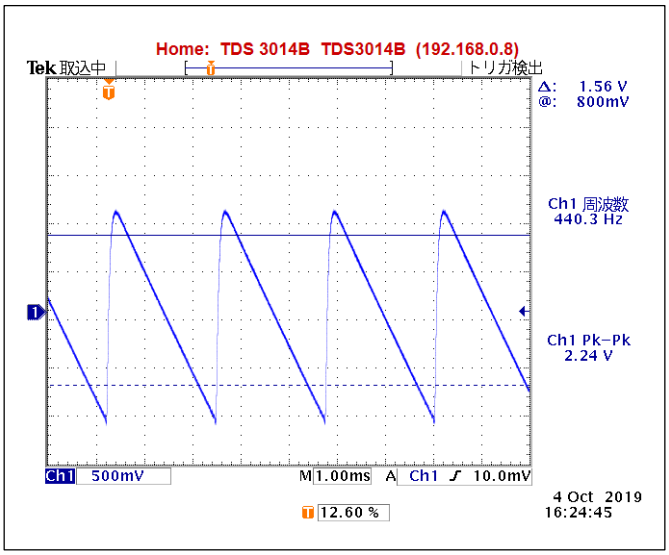
4 種+無音のシーケンスを切り替える

上記のコードを実行し、出力をオシロスコープで観測した場合、以下の様な出力が得られます。

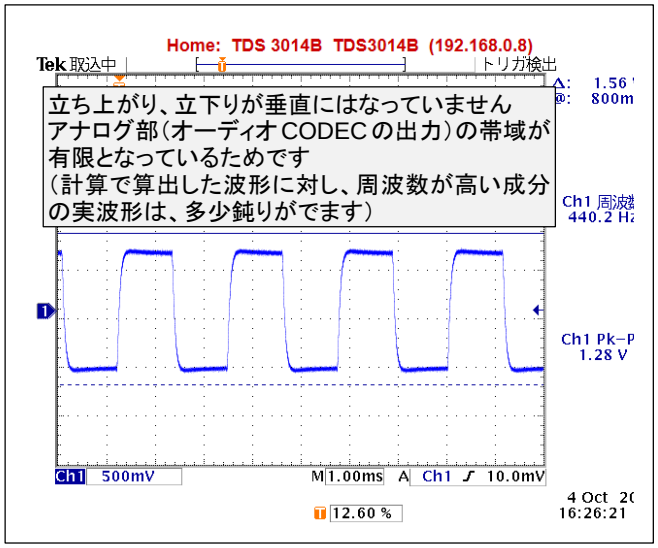
(1)サイン波



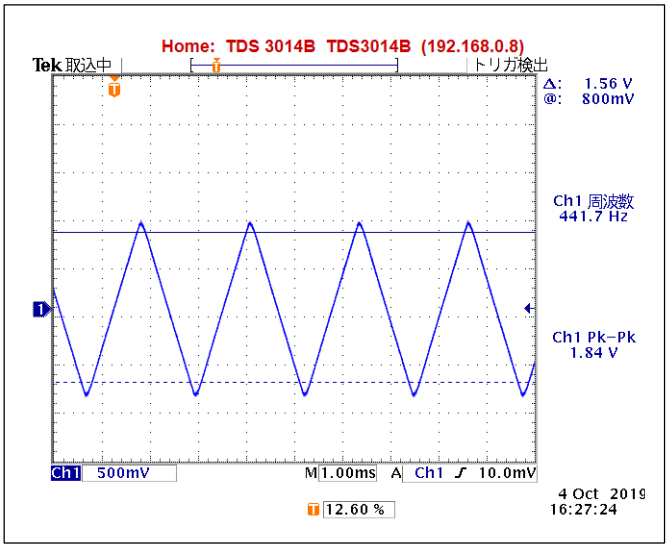
(2)ノコギリ波



(3)矩形波



(4)三角波



本例では、単純なサイン波の重ね合わせで作成できる波形を生成するサンプルを示しましたが、波形データさえ作成できれば、どのような波形でも生成可能です。

なお、本チュートリアルで作成したプログラムは、CD の SAMPLE フォルダ内に格納されています。

各種波形の生成に関しては、参考文献「サウンドプログラミング入門」の 2,3 章
ex2_1.c, ex3_1.c, ex3_2.c, ex3_3.c
を参考にしていますので、参照頂きたく。

6.2. ワンショットモードモード

同じテンプレートをベースに、

・ワンショットモード
・ボリュームで、周波数を変更可能
・「減衰するサイン波（エンベロープ変化）」を出力する

プログラムを作成してみます。

※Push-SW を押す度に 1 回出力されます

テンプレートを、コピーし、

[folder]¥RX65_SOUND_EFFECTOR_ENVEROPE

というフォルダ名に変更する事とします。

src¥user¥sound_effector_userdef.c

```
//動作モード//"REPEAT(=0)", "ONCE(1)", "BUFFER(2)"のいずれかを定義
//#define OUT_MODE REPEAT//REPEAT: 0~g_outbuf_end間を繰り返し出力
#define OUT_MODE ONCE//ONCE: Push-SWを押すたびに0~g_outbuf_end間を1回出力
//#define OUT_MODE BUFFER//BUFFER: 出力バッファをFIFOで使用するモード
//※コンパイルオプション、実行中に動作モードを切り替える事は不可
```

動作モードを、sound_effector_userdef.c ファイル内で変更します。デフォルトでは、REPEAT の行が有効になっていますので、ONCE の行を有効化します。

RX65_SOUND_EFFECTOR.c

```
void main(void)
    {

    int n;

    float f0;

    float a, a0, a_end;
    //オリジナルで使用している *a は、振幅に相当する係数となるが、メモリ容量の関係上、配列として確保せず、ループ内で計算する

    int length;

    float volume1_val=0;

    unsigned char prev_sw_state;//1世代前のスイッチの状態

    //初期化関数
    sound_effector_init();

    length = PCM_FS * 4;/* 音データの長さ（4秒） */

    //音データは、出力バッファに直接書き込んでいくのでpcm.sのcallocに相当する処理は不要
    //（pcm.sに相当する領域は、g_outbufとして確保済み）

    a0 = 0.5;//スタート時の振幅
    a_end = 0.0;//最後の振幅

    g_outbuf_end=length-1;//波形周期を設定（4秒）

    sound_effector_start();//動作開始

    prev_sw_state = g_sw_state;//スイッチの状態を保存
```

生成するデータの長さを 4 秒とし、振幅を 0.5→0 にするプログラムです。

```
    while(1)
    {
        if(prev_sw_state != g_sw_state)
        {
            //スイッチの状態が変化した

            g_data_out_flag=DATA_OUT_HOLD;//出力停止

            volume1_val = (float)g_ad_val[0] / 4096.0;//VR1のA/D変換結果を 0-1 に正規化

            f0 = (8000.0 - 20.0) * volume1_val + 20.0;//20-8000Hzの周波数に変換

            //減衰するエンベロープを持つsin波形の生成

            for(n=0; n<length; n++)
            {
                a = a0 + (a_end - a0) * n / (length -1);//振幅
                g_outbuf[n] = clipping_data_norm( a * (sinf(2.0 * M_PI * f0 * n /
PCM_FS)));//sin波形の生成及び出力バッファへの格納
            }

            g_outbuf_rp = 0;//波形をデータの最初から出力する
            g_data_out_flag=DATA_OUT;//出力開始

            prev_sw_state = g_sw_state;//現在のスイッチの状態を保存
            }

        nop();

    }//while
}
```

ボリューム 1 の回転角度に応じて、周波数を 20~8,000Hz に設定し、振幅が減衰（直線状のエンベロープを持つ）する波形を生成します。

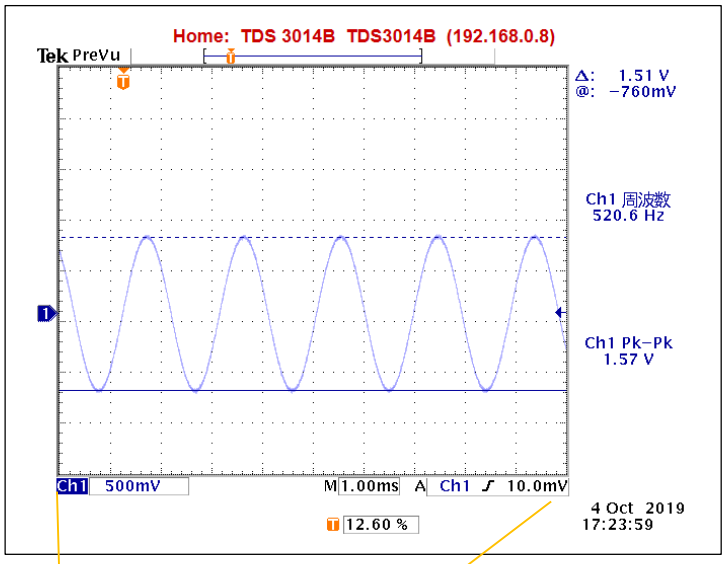
波形データ生成後に、出力を開始します。

このプログラムでは、波形データの生成に約 0.5 秒程度掛かります。データ生成後に、出力を開始するようになっていますので、スイッチを押した後、音が出るまでには約 0.5 秒掛かります。
（リピートモードやワンショットモードでは、波形生成に演算時間が掛かっても問題ありません）

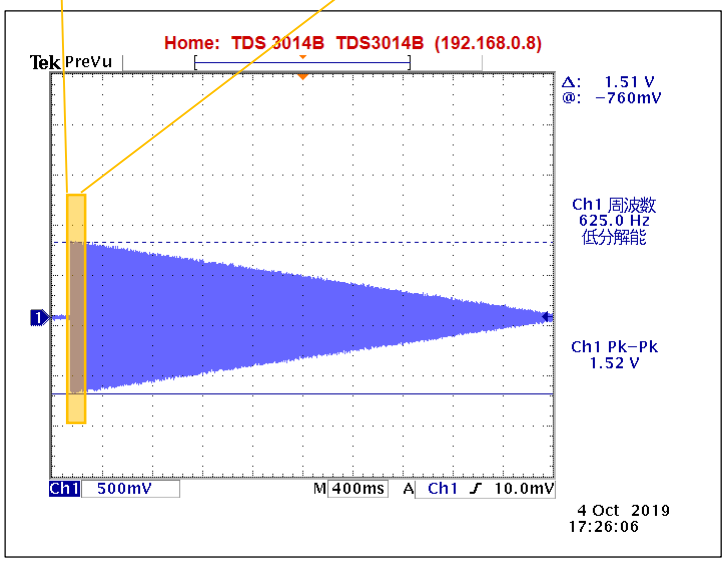
波形の生成に関しては、参考文献「サウンドプログラミング入門」の 6 章
ex6_1.c
を参考にしていますので、参照頂きたく。

上記のコードを実行し、出力をオシロスコープで観測した場合、以下の様な出力が得られます。

(1)出力開始時を 1ms/div で観測



(2)全体を 400ms/div で観測



4 秒で、出力が 100%から 0%に直線的に変化（直線のエンベロープを持つ波形）となります。

6.3. バッファモード

同じテンプレートをベースに、

・リアルタイムモード
・ボリュームで、時間エンベロープの変化時間(ADSR)を制御(*1)
・Push-SW を押すと「出力スタート」、もう一度押すと「出力ストップ」
・440Hz のこぎり波＋LPF の音を出力する

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_ADSR

というフォルダ名に変更する事とします。

src¥user¥sound_effector_userdef.c

```
//動作モード//"REPEAT(=0)", "ONCE(1)", "BUFFER(2)"のいずれかを定義
//#define OUT_MODE REPEAT//REPEAT: 0~g_outbuf_end間を繰り返し出力
//#define OUT_MODE ONCE//ONCE: Push-SWを押すたびに0~g_outbuf_end間を1回出力
#define OUT_MODE BUFFER//BUFFER: 出力バッファをFIFOで使用するモード
//※コンパイルオプション、実行中に動作モードを切り替える事は不可
```

動作モードは、sound_effector_userdef.c 内で、BUFFER を有効化します。

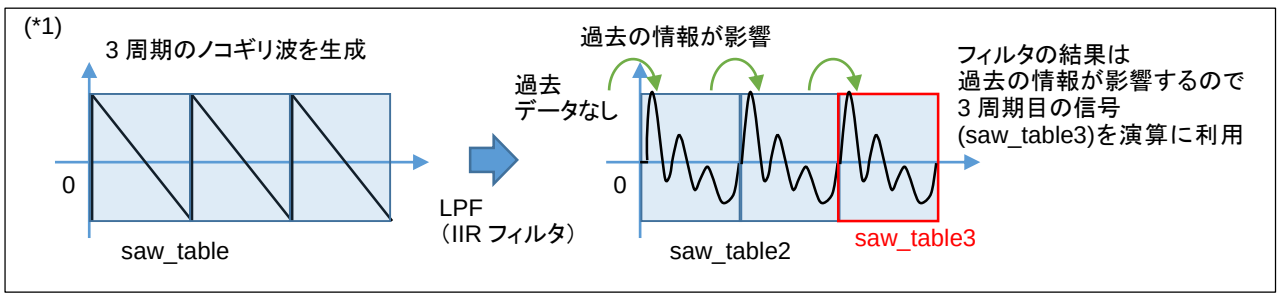
(*1)音の大きさのエンベロープ（時間変化）を、「A，アタックタイム」「D，ディケイタイム」「S，サスティンレベル」「R，リリースタイム」の 4 つのパラメータで制御する方式です
3 つのボリュームで、
VR1: A アタックタイム
VR2: D ディケイタイム
VR3: R リリースタイム
を可変できる様にしています。

RX65_SOUND_EFFECTOR.c

```
void main(void)
{

    int n, m, t0, Ix, J, A, D, R, gate;
    float vco, vcf, vca, gain, S, Q, a[3], b[3];

    int spool_data_num;
    unsigned char prev_sw_state;

    float *saw_table;//ノコギリ波のテーブル
    float *saw_table2;//ノコギリ波のテーブル（LPF適用後）
    float *saw_table3;//ノコギリ波のテーブル（LPF適用後, 1周期）

    //初期化関数
    sound_effector_init();

    /* ノコギリ波 */
    vco = 440.0;/* 基本周波数 */
    t0 = PCM_FS / vco;/* 基本周期 */

    saw_table = calloc(t0 * 3, sizeof(float));
    saw_table2 = calloc(t0 * 3, sizeof(float));
    saw_table3 = calloc(t0, sizeof(float));


    if((saw_table == NULL) || (saw_table2 == NULL) || (saw_table3 == NULL))
    {
        while(1);
        //メモリの確保に失敗
    }

    //ノコギリ波の生成（3周期のテーブルを作成）
    m = 0;
    for(n = 0; n < (t0 * 3); n++)
    {
        saw_table[n] = 1.0 - 2.0 * m / t0;
        m++;
        if(m>=t0)
        {
            m=0;
        }
    }

    vcf = 1500.0;/* 遮断周波数 */
    Q = 5.0;/* レゾナンス */
    Ix = 2;/* 遅延器の数 */
    J = 2;/* 遅延器の数 */

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

オリジナルのソースでは、I を使用しているが、complex.h で、I をマクロとして使用しているので、Ix にリネームする

(*1)
LPF 適用前のノコギリ波:3 周期分生成
LPF 適用後のノコギリ波:3 周期分とする
波形出力に使用するノコギリ波:1 周期分とする



```
/* フィルタリング （3周期の元信号をフィルタリングし saw_table2 に出力 */
for(n = 0; n < (t0 * 3); n++)
{
    for(m = 0; m <= J; m++)
    {
        if(n - m >= 0)
        {
            saw_table2[n] += b[m] * saw_table[n - m];
        }
    }

    for(m = 1; m <= Ix; m++)
    {
        if (n - m >= 0)
        {
            saw_table2[n] += -a[m] * saw_table2[n - m];
        }
    }
}

/* 3周期の信号の最後の周期を抜き出す */
for(n = 0; n < t0; n++)
{
    saw_table3[n] = saw_table2[n + t0 * 2];
}

A = 0;//VR1の値を反映（最大1秒）
D = 0;//VR2の値を反映（最大1秒）
S = 0.5;
R = 0;//VR3の値を反映（最大1秒）

gain = 0.5;

gate = 0;//鍵盤=OFF（押されていない）

sound_effector_start();//動作開始

prev_sw_state = g_sw_state;//Push-SWの状態を保存
```

(*2)本プログラムでは、S の値を 0.5 としています

```
while(1)
{
    if(prev_sw_state != g_sw_state)
    {
        //Push-SWが押された

        if(gate == 0)//出力停止中ならば、鍵盤が押されたと判断
        {
            gate = 1;//鍵盤=ON（押している状態）

            A = g_ad_val[0] * PCM_FS / 4096;//VR1の回転に応じ最大1秒
            D = g_ad_val[1] * PCM_FS / 4096;//VR2の回転に応じ最大1秒

            m=0;

            //A
            for (n = 0; n < A; n++)
            {
                vca = 1.0 - exp(-5.0 * n / A);

                put_data(clipping_data_norm(saw_table3[m] * vca * gain));

                m++;
                if(m >= t0) m = 0;
            }

            //D
            for (n = 0; n < D; n++)
            {
                vca = S + (1 - S) * exp(-5.0 * n / D);

                put_data(clipping_data_norm(saw_table3[m] * vca * gain));

                m++;
                if(m >= t0) m = 0;
            }
        }
        else if(gate == 1)//出力中ならば、鍵盤が離された判断
        {
            gate = 0;//鍵盤=OFF（離している状態）

            R = g_ad_val[2] * PCM_FS / 4096;//VR3の回転に応じ最大1秒

            //R
            for (n = 0; n < R; n++)
            {
                vca = S * exp(-5.0 * (n + 1) / R);

                put_data(clipping_data_norm(saw_table3[m] * vca * gain));

                m++;
                if(m >= t0) m = 0;
            }
        }
        prev_sw_state = g_sw_state;//スイッチの状態を保存
    }
```

A（アタック）の部分のデータ生成

D（ディケイ）の部分のデータ生成

R（リリース）の部分のデータ生成

```
        if(gate == 1)
        {
            //出力中
                                                    S(サステイン)の部分のデータ生成
            //S
            spool_data_num = outbuf_spool_data_num();//出力バッファに溜まっているデータ個数を取
得
            if(spool_data_num < (int)(PCM_FS * 0.01))//出力バッファに格納済みのデータが0.01秒未
満となった場合
            {
                vca = S;
                for(n = 0; n < (int)(PCM_FS * 0.01); n++)//0.01秒分のデータを出力バッファに追加
                {
                    put_data(clipping_data_norm(saw_table3[m] * vca * gain));

                    m++;
                    if(m >= t0) m = 0;
                }
            }
        }

        nop();

    }//while
}
```

波形の生成に関しては、参考文献「サウンドプログラミング入門」の 9 章
ex9_4.c
を参考にしていますので、参照頂きたく。

※オリジナルのプログラムで、変数 I(大文字・アイ)を使用している部分を、Ix(大文字・アイ,小文字・エックス)に変更しています。これは、I が、コンパイラ標準のヘッダファイル complex.h でマクロとして定義されているためです。
(I は、complex.h をインクルードする限り予約語の様な扱いとなってしまいます)

オリジナルのプログラムでは、4 秒分のデータを一括して計算後に、ファイルに書き込む処理となりますが、本プログラムでは Push-SW が押される度に波形データの都度計算、出力を行っています。

このバッファモードでは、データの生成に時間が掛かると、出力するデータが空になってしまうという問題があります。(空になった際は無音データが出力されるだけですので、空になる事が問題というわけではないのですが)

「リピート」「ワンショット」のモードでは、波形データの生成に時間が掛かっても問題ありませんでしたが、「バッファモード」では、出力バッファから出て行くタイミング(20.83us に 1 データ)、出力バッファにデータを格納するタイミングを意識する必要があります。

※本プログラムでは、最初に出力バッファに格納されるデータは、A（アタック）の部分のデータです。例えば A=1 秒に設定すると、約 48,000 個のデータを出力バッファに格納するのに、173ms 掛かりました。（コンパイラの最適化 OFF 時）

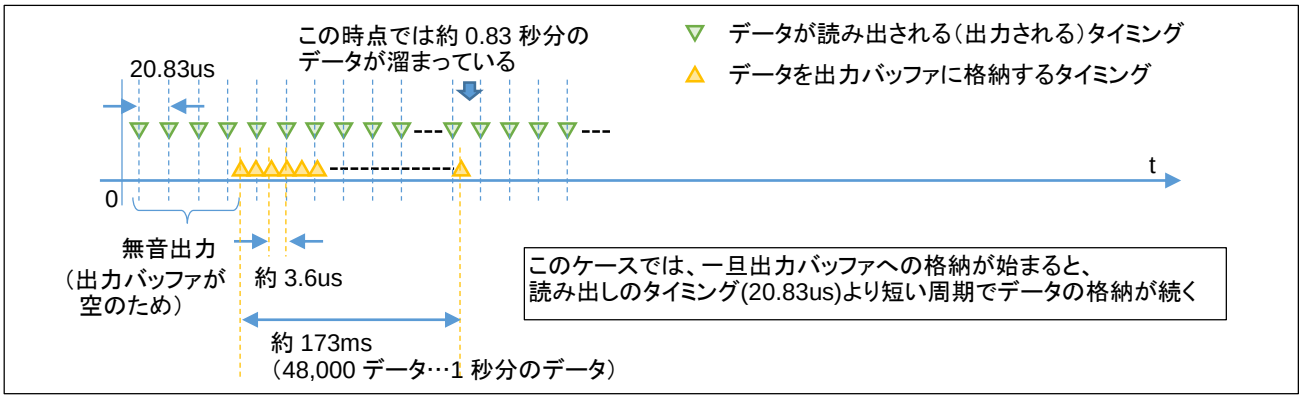


図 6-1 出力バッファへのデータ格納とデータ出力

データの格納と読み出しのタイミングを図にすると、上図の様な関係となります。

このケースでは、
データを出力バッファに格納するタイミング 3.6us ＜ 読み出しタイミング 20.83us
となっているので問題ありませんが、データを出力バッファに格納するタイミングが 1 データあたり 20.83us を超えない様に考慮する必要があります。
（データ生成に掛かる演算、処理時間に関する意識が必要です）

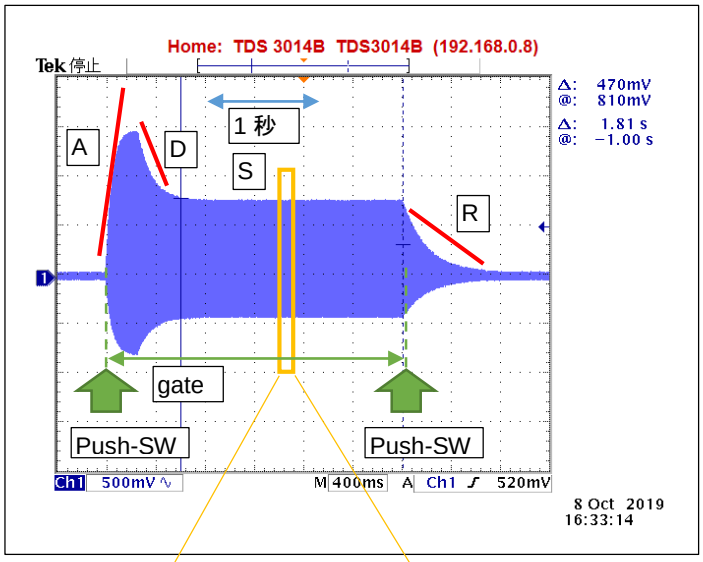
本プログラムでは、ADR のデータは、ボリュームの回転角度に応じ 0~1 秒（0~48,000 データ）分生成して、出力バッファに格納します。AD のデータは連続して生成、R のデータは Push-SW を押したタイミングで生成となります。

S のデータは、outbuf_spool_data_num() 関数を用いて、出力バッファに溜まっているデータが 0.01 秒（480 データ）未満となった場合、0.01 秒分のデータを生成し、出力バッファに格納する様にしています。

※本プログラムでは、ADR のデータが連続して生成された場合でも、最大 3 秒分のデータとなりますので、出力バッファが溢れる事を想定した処理を入れていません。出力バッファが溢れる可能性がある場合は、バッファの空き容量に応じた処理や、バッファフル時（put_data() の戻り値が 1）の際の処理を入れてください。

上記のコードを実行し、出力をオシロスコープで観測した場合、以下の様な出力が得られます。

(1)全体のエンベロープ

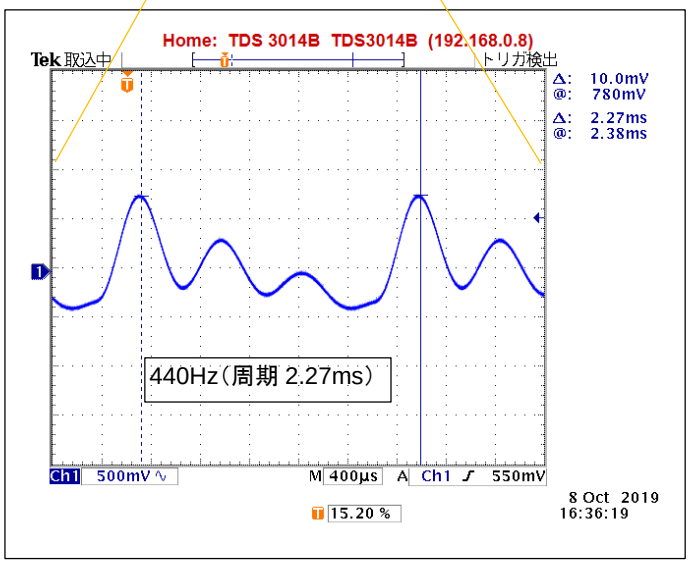


「1 度目の Push-SW 押す」タイミングから「2 度目の Push-SW 押す」タイミングまでが gate タイム（鍵盤を押している間）となるイメージです

A : 1/4 回転→0.25 秒
D : 1/2 回転→0.5 秒
S : ゲイン=0.5
R : 1 回転→1 秒
に設定
※ADR の変化は指数関数のカーブとしています

※波形の上下がアンバランスですが、元信号（ノコギリ波+LPF）がアンバランスのためです

(2)波形の切り出し



ノコギリ波＋LPF

(2)波形の切り出し

この波形では、

A=0.25s

D=0.5s

S=0.5（A のピークの 1/2） (*2)

R=1s

の設定としています。（A,D,R のカーブは指数関数）ADR の時間は、3 つのボリュームの回転角で可変します。

(*2)オリジナルのプログラムは 1 としていますが、AD の効果を見るため本サンプルでは、0.5 としています。

以上、3 つの動作モードを簡単なサンプルプログラムで説明しました。

色々な波形（データ）を生成し、出力バッファに格納していくというのが、本機器の使い方となります。

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

set_data

概要:データ出力バッファ格納関数

宣言:

int set_data(short data, unsigned long index)　[出力 1ch 向け]

int set_data(pcm_data data, unsigned long index)　[出力 2ch 向け]

説明:

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（リピートモード、ワンショットモード)

引数[出力 2ch 向け]:

short data　音声データ（16bitPCM データ）

unsigtned long index　出力バッファの配列のインデックス

引数[出力 2ch 向け]:

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

unsigtned long index　出力バッファの配列のインデックス

戻り値:

0: 出力バッファにデータ格納する処理が成功

1: index が出力バッファの範囲を超えているため、データ格納処理が失敗

補足:

出力バッファ(g_outbuf[ ])に直接データを書き込んでも等価です、インデックスの範囲のチェックが追加されているだけです

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:
　FIFO 構成の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

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。「リピートモード」「ワンショットモード」の際には、本変数を書き換えるか、set_data() 関数を使用して、出力バッファへの格納を行う。「バッファモード」の際は、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。「バッファモード」の際に使用される。

unsigned long g_outbuf_end

出力バッファの、終了位置を示す変数。「リピートモード」「ワンショットモード」の際に使用される。

unsigned short g_data_out_flag

「出力許可」「出力停止」を制御する変数。

DATA_OUT_HOLD(0): 出力停止
DATA_OUT(1): 出力許可

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 1 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 OUT_MODE REPEAT

動作モードの指定。
REPEAT(=0)： リピートモード
ONCE(=1)： ワンショットモード
BUFFER(=2)： バッファモード
※カッコ内の数値は、REPEAT 等の定義値です REPEAT を指定するのと、0 を指定するのは等価です

#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 OUTBUF_SIZE (98304) [2ch]
#define OUTBUF_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、「音を作り出す」ケースを説明致しました。

リピートモード、ワンショットモードでは、最大 4 秒分のデータを扱う事が可能で、データの生成（計算）に時間が掛かっても良い（数値計算の最適化等は考慮しなくとも良い）ですので、まずはこれらのモードで音を出してみる事を推奨致します。

バッファモードでは、出力バッファの読み出しタイミング（20.83us）が関係してきますので、ある程度リアルタイム処理が必要になってきます。時間の掛かる処理は、所定の時間で計算が終わる様工夫が必要になってきます。そのため、バッファモードの方が使い方が多少難しい面があるものと考えます。

次のマニュアル、
「サウンドエフェクタ　ソフトウェア編(2)　取扱説明書　～入力信号を加工して出力してみる～」
では、入力信号を加工して出力する手順が説明されています。

入力信号を加工して出力する場合、本マニュアルのバッファモードでのリアルタイム処理の考え方が重要になってきますので、「サウンドエフェクタ　ソフトウェア編(2)」で詰まった場合は、入力を使用しない、バッファモードのプログラムで試行してみる事をお勧めします。

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(1)
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)