サウンドエフェクタ
ソフトウェア編(3)
～FFT/逆 FFT をライブラリ関数で処理してみる～
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[本書]　～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。

本資料では、FFT/逆 FFT をライブラリ関数を使用して処理する事に関して説明します。

ベースとなるテンプレートは、

RX65_SOUND_EFFECTOR_TEMPLATE_LIB

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、ライブラリ関数を用いて、サウンドエフェクタのプログラムを作成する手法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。

使用するライブラリに関して

本機器で採用している CPU は、ルネサスエレクトロニクス製、RX651 マイコンとなりますが、ルネサスエレクトロニクスでは、RX マイコン向けに、「RX ファミリ用 DSP ライブラリ」というライブラリを提供しています。

ルネサスエレクトロニクス Web より、

デザイン/サポート　－　サンプルコード検索
「RX ファミリ用 DSP ライブラリ」 [検索]

RX ファミリ用 DSP ライブラリ　Version5.0

（an-r01an4359　で検索しても、見つかると思います）

an-r01an4359jj0100-rx-dsplib.zip （2019/11 現在）

を予め、ダウンロードを行っておいてください。

上記を展開すると、いくつかのフォルダに展開されますが、使用するのは、

dsplib-rxv2

以下のファイルです。

reference_document

以下には、ライブラリのマニュアルもありますので、参照してください。

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_LIB

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。

ライブラリのコピー先

上記テンプレートには、RX ファミリ用 DSP ライブラリが含まれておりません。再配布条件により、このライブラリを使用する際は、ユーザ側でダウンロードして頂く必要があり、展開した

dsplib-rxv2

フォルダを

[path]¥RX65_SOUND_EFFECTOR_TEMPLATE_LIB¥lib

以下にコピーしてください。

ライブラリファイル（拡張子.lib）は、何種類か用意されていますが、使用するのは

RX_DSP_FPU_LE_Check.lib

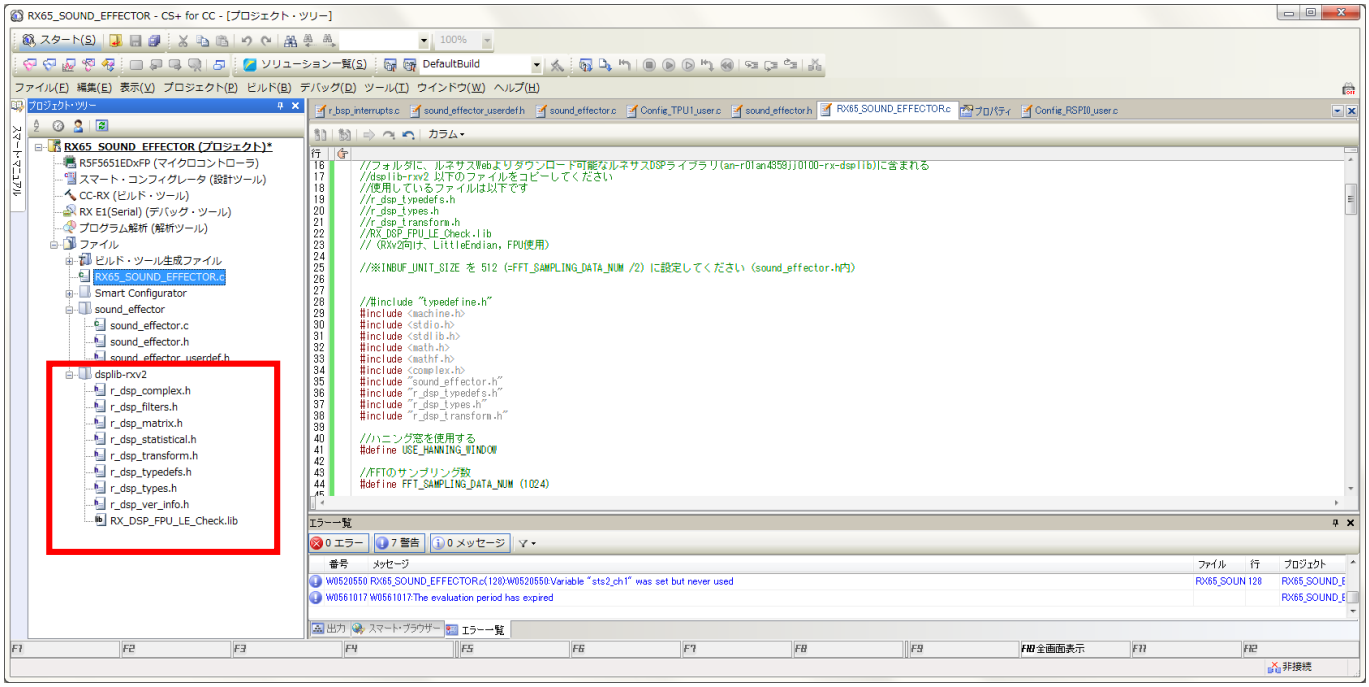
です。その他、ヘッダファイル（拡張子.h）も使用します。

※dsplib-rxv2 フォルダをコピーしないと、プロジェクトのビルド時に、ファイルが無い
[Could not open source file]のエラーが出ます

[path]¥RX65_SOUND_EFFECTOR_TEMPLATE_LIB¥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。

CS+が起動すると、下記の様画面となります。



赤枠の部分が、RX ファミリ用 DSP ライブラリのファイルです。

1. テンプレート・プロジェクトの説明

テンプレートのプログラムは、

・Push-SW が OFF のとき、入力信号をスルーで出力（VR1 で音量調整）[Write/ErrLED は消灯]
・Push-SW が ON のとき、入力信号を FFT 処理し、逆 FFT 処理して出力する（要はスルー）[Write/ErrLED が点滅]

というサンプルとなっています。

見た目（聞いた感じ）では、Push-SW が ON でも OFF でも相違はほぼありません（Push-SW が OFF の際は、ボリューム調整が有効なので、VR1 を絞ると、出力レベルが小さくなります）。

※スルーの場合は、入力端子から出力端子までのレイテンシーが 15.8ms, FFT/逆 FFT 出力とした場合は、26.6ms となります

FFT を掛けた後、逆 FFT を掛けて時間領域のデータに戻しているので、複雑な計算は行っていますが、結果は入力信号がそのまま出力に伝播されるという内容です。

RX65_SOUND_EFFECTOR.c

```
#include <machine.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mathf.h>
#include <complex.h>
#include "sound_effector.h"
#include "r_dsp_typedefs.h"
#include "r_dsp_types.h"
#include "r_dsp_transform.h"

//FFTのサンプリング数
#define FFT_SAMPLING_DATA_NUM (1024)

void main(void)
{

    unsigned long prev_inbuf_wp;

    long inbuf_index;

    pcm_data pcm_tmp;

    float data_tmp_ch0, data_tmp_ch1;

    float volume1_val;

    unsigned short i;

    static float fft_input_ch0[FFT_SAMPLING_DATA_NUM];//time-domain, FFT入力データ
    static float fft_input_ch1[FFT_SAMPLING_DATA_NUM];

    static float prev_output_data_ch0[FFT_SAMPLING_DATA_NUM/2];//1世代前のIFFT結果
    static float prev_output_data_ch1[FFT_SAMPLING_DATA_NUM/2];

    static cplxf32_t fft_output_ch0[FFT_SAMPLING_DATA_NUM/2];//frequency-domain, FFT出力
データ
    static cplxf32_t fft_output_ch1[FFT_SAMPLING_DATA_NUM/2];

    static float ifft_output_ch0[FFT_SAMPLING_DATA_NUM];//time-domain, IFFT出力データ
    static float ifft_output_ch1[FFT_SAMPLING_DATA_NUM];

    static vector_t vtime_ch0 = {FFT_SAMPLING_DATA_NUM, (void *)fft_input_ch0};
    static vector_t vtime_ch1 = {FFT_SAMPLING_DATA_NUM, (void *)fft_input_ch1};

    static vector_t vfreq_ch0 = {FFT_SAMPLING_DATA_NUM/2, (void *)fft_output_ch0};
    static vector_t vfreq_ch1 = {FFT_SAMPLING_DATA_NUM/2, (void *)fft_output_ch1};

    static vector_t vtime2_ch0 = {FFT_SAMPLING_DATA_NUM, (void *)ifft_output_ch0};
    static vector_t vtime2_ch1 = {FFT_SAMPLING_DATA_NUM, (void *)ifft_output_ch1};

    static cplxf32_t fft_twiddles1[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];
    static uint32_t fft_bitrev1[240];     //1024 pointのバッファサイズ
```

本プログラムで使用しているライブラリ関数のヘッダ

FFT，逆 FFT で使用するワーク変数

```
    static float window[FFT_SAMPLING_DATA_NUM];//窓関数

    static cplxf32_t fft_twiddles2[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];
    static uint32_t fft_bitrev2[240];   //1024 pointのバッファサイズ

    static r_dsp_fft_t h1 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles1, fft_bitrev1,
NULL, window};//FFT向け 窓関数を含める

    static r_dsp_fft_t h2 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles2, fft_bitrev2,
NULL, NULL};//IFFT向け

    r_dsp_status_t sts1 = R_DSP_STATUS_OK;//FFT初期化
    r_dsp_status_t sts2 = R_DSP_STATUS_OK;//IFFT初期化
    r_dsp_status_t sts1_ch0 = R_DSP_STATUS_OK;//FFT
    r_dsp_status_t sts1_ch1 = R_DSP_STATUS_OK;
    r_dsp_status_t sts2_ch0 = R_DSP_STATUS_OK;//IFFT
    r_dsp_status_t sts2_ch1 = R_DSP_STATUS_OK;

    /* initialize required buffer sizes for twiddles etc. */
    size_t ntwb1; // bytes for twiddle array
    size_t nbrb1; // bytes for bit-reverse table
    size_t nwkb1; // bytes for working area

    size_t ntwb2; // bytes for twiddle array
    size_t nbrb2; // bytes for bit-reverse table
    size_t nwkb2; // bytes for working area
```

FFT，逆 FFT で使用するワーク変数

```
    //1世代前のIFFT結果の初期化
    for(i = 0; i< (FFT_SAMPLING_DATA_NUM / 2); i++)
    {
        prev_output_data_ch0[i] = 0;
        prev_output_data_ch1[i] = 0;
    }

    //窓関数定義 ハニング窓
    for(i=0; i<FFT_SAMPLING_DATA_NUM; i++)//FFT_SAMPLING_DATA_NUMは偶数
    {
        window[i] = 0.5 - 0.5 * cosf(2.0 * M_PI *
((float)i/(float)FFT_SAMPLING_DATA_NUM));
    }

    //FFT構造体初期化
    sts1 = R_DSP_FFT_BufSize_f32cf32(&h1, &ntwb1, &nbrb1, &nwkb1);

    if((sizeof(fft_twiddles1) < ntwb1) || (sizeof(fft_bitrev1) < nbrb1))
    {
        while(1); /* not enough memory size */
    }

    sts1 = R_DSP_FFT_Init_f32cf32(&h1);

    //IFFT構造体初期化
    sts2 = R_DSP_FFT_BufSize_cf32f32(&h2, &ntwb2, &nbrb2, &nwkb2);

    if((sizeof(fft_twiddles2) < ntwb2) || (sizeof(fft_bitrev2) < nbrb2))
    {
        while(1); /* not enough memory size */
    }

    sts2 = R_DSP_FFT_Init_cf32f32(&h2);
```

```
    //初期化関数
    sound_effector_init();

    //動作開始
    sound_effector_start();

    prev_inbuf_wp = 0;

    while(1)
    {

        if(g_sw_state == 0)
        {
            //スルー（VR1で音量調整）
            volume1_val = (float)g_ad_val[0] / 4096.0;//ボリューム値を正規化(0 ～ 1)

            if(prev_inbuf_wp != g_inbuf_wp)
            {
                for(i=0; i<INBUF_UNIT_SIZE; i++)
                {

                    data_tmp_ch0 = (float)g_inbuf[prev_inbuf_wp + i].ch0 / 32768.0; //入力
値を正規化(-1 ～ 1)
                    data_tmp_ch1 = (float)g_inbuf[prev_inbuf_wp + i].ch1 / 32768.0;

                    data_tmp_ch0 *= volume1_val;//ボリューム値を乗算
                    data_tmp_ch1 *= volume1_val;

                    //クリッピング処理
                    pcm_tmp = clipping_data_norm(data_tmp_ch0, data_tmp_ch1);

                    //出力バッファに格納
                    put_data(pcm_tmp);
                }
                prev_inbuf_wp = g_inbuf_wp;
            }
        }
```

Push-SW OFF のとき（スルー＋音量調整）

```
        else
        {
            //FFT
            if(prev_inbuf_wp != g_inbuf_wp)
            {
                //1回目 (-512)-511のデータを処理，ダミー
                //2回目 0-1023データを処理
                //3回目 512-1535のデータを処理
                //4回目 1024-2047のデータを処理
                //→512ずつオーバラップさせてデータを処理する

                //入力データをFFT構造体内の配列変数にコピー
                for(i=0; i<FFT_SAMPLING_DATA_NUM; i++)
                {
                    //この時点で、inbuf_wp(=prev_inbuf_wp + (FFT_SAMPLING_DATA_NUM / 2))までは
                    //入力データが格納済み
                    //→prev_inbuf_wp - (FFT_SAMPLING_DATA_NUM / 2) ～ prev_inbuf_wp -
(FFT_SAMPLING_DATA_NUM / 2) + FFT_SAMPLING_DATA_NUM
                    // (prev_inbuf_wp-512 ～prev_inbuf_wp+512(=inbuf_wp) の範囲のデータを読み込む事は
問題がない

                    inbuf_index = prev_inbuf_wp - (FFT_SAMPLING_DATA_NUM / 2) + i;
                    if(inbuf_index < 0) inbuf_index += INBUF_SIZE;
                    else if(inbuf_index >= INBUF_SIZE) inbuf_index -= INBUF_SIZE;

                    fft_input_ch0[i] = (float)g_inbuf[inbuf_index].ch0 / 32768.0;
                    fft_input_ch1[i] = (float)g_inbuf[inbuf_index].ch1 / 32768.0;
                }

                //FFT vtime -> vfreq, 時間領域から周波数領域に変換
                sts1_ch0 = R_DSP_FFT_f32cf32(&h1, &vtime_ch0, &vfreq_ch0);
                sts1_ch1 = R_DSP_FFT_f32cf32(&h1, &vtime_ch1, &vfreq_ch1);

                //IFFT vfreq -> vtime2, 周波数領域から時間領域に逆変換
                sts2_ch0 = R_DSP_IFFT_CCS_cf32f32(&h2, &vfreq_ch0, &vtime2_ch0);
                sts2_ch1 = R_DSP_IFFT_CCS_cf32f32(&h2, &vfreq_ch1, &vtime2_ch1);

                //IFFT結果の連結
                for(i = 0; i < (FFT_SAMPLING_DATA_NUM / 2); i++)
                {
                    //前半の512データは、前回の演算結果と加算して出力バッファに格納

                    data_tmp_ch0 = (prev_output_data_ch0[i] + ifft_output_ch0[i]);
                    data_tmp_ch1 = (prev_output_data_ch1[i] + ifft_output_ch1[i]);

                    //クリッピング処理
                    pcm_tmp = clipping_data_norm(data_tmp_ch0, data_tmp_ch1);

                    put_data(pcm_tmp);
                }

                for(i = 0; i < (FFT_SAMPLING_DATA_NUM / 2); i++)
                {
                    //後半の512データは、変数に一時保存
                    prev_output_data_ch0[i] = ifft_output_ch0[i + (FFT_SAMPLING_DATA_NUM / 2)];
                    prev_output_data_ch1[i] = ifft_output_ch1[i + (FFT_SAMPLING_DATA_NUM / 2)];
                }

                prev_inbuf_wp = g_inbuf_wp;
            }
        }

    }//while
}
```

FFT+逆 FFT 処理

FFT の入力

FFT 処理

FFT ライブラリ関数

逆 FFT 処理

逆 FFT ライブラリ関数

オーバラップアド処理

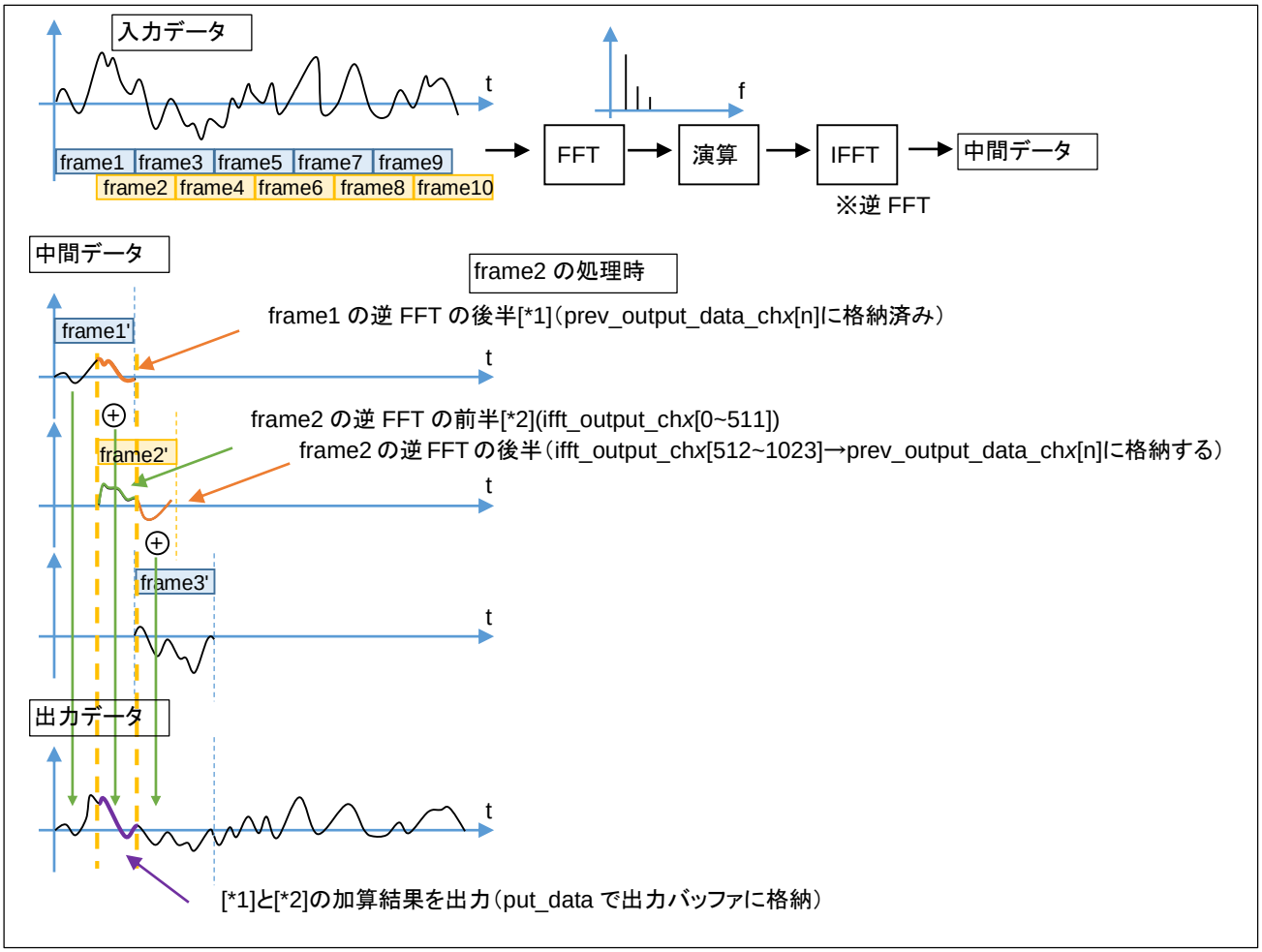
1 周期前の後半データ

現周期の前半データ

出力バッファに格納

現周期の後半データ

・フレーム単位でのオーバラップアド処理



本プログラムは、データを 512 サンプルずつオーバラップして切り出しを行い、FFT、逆 FFT を掛けて、逆 FFT 結果をオーバラップアド処理しています。

FFT を掛ける処理は、以下の手順となります。

(1)FFT の入力となるデータの配列を用意する（ここでは、float 型で、サンプリング数(1024)の配列）
static float fft_input_ch0[FFT_SAMPLING_DATA_NUM];

(2)FFT の出力となるデータの配列を用意する（ここでは、float 型の複素数で、サンプリング数/2(=512)の配列）
static cplxf32_t fft_output_ch0[FFT_SAMPLING_DATA_NUM/2];

(3)FFT の入力のベクタを用意する
static vector_t vtime_ch0 = {FFT_SAMPLING_DATA_NUM, (void *)fft_input_ch0};

FFT のライブラリ関数は、データとサンプル数等をまとめたベクタを引数として扱うため、入力のベクタを定義します。

(4)FFT の出力のベクタを用意する
static vector_t vfreq_ch0 = {FFT_SAMPLING_DATA_NUM/2, (void *)fft_output_ch0};

(5)FFTで使用するワーク変数を用意する
static cplxf32_t fft_twiddles1[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];
static uint32_t fft_bitrev1[240];

(6)FFT で使用する窓関数の配列を用意する[オプション]
static float window[FFT_SAMPLING_DATA_NUM];

FFT で窓関数を使用する場合は、窓関数のテーブルを用意する。窓関数を使用しない場合は不要。
※窓関数のテーブルの値を定義する事も必要

(7)ワーク変数、窓関数をまとめた構造体を定義する（ここでは"h1"という名称で構造体を定義）
static r_dsp_fft_t h1 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles1, fft_bitrev1, NULL, window};

(8)ワーク変数のサイズが足りているかの確認[オプション]
size_t ntwb1; // bytes for twiddle array
size_t nbrb1; // bytes for bit-reverse table
size_t nwkb1; // bytes for working area
sts1 = R_DSP_FFT_BufSize_f32cf32(&h1, &ntwb1, &nbrb1, &nwkb1);

使用するワークエリアのサイズの確認。（容量が足りている事が明らかであれば不要）

(9)FFT 構造体の初期化
sts1 = R_DSP_FFT_Init_f32cf32(&h1);

FFT の入力として、f32（32bit 浮動小数点）。FFT の出力として、cf32（32bit 浮動小数点、複素数）。
※入力、出力は様々な型があり、ここでは、入力 f32、出力 cf32 を使用する

(10)FFT の実行
sts1_ch0 = R_DSP_FFT_f32cf32(&h1, &vtime_ch0, &vfreq_ch0);

構造体、入力ベクタ（横軸：時間のデータ）、出力ベクタ（横軸：周波数のデータ）。
※入力、出力は様々な型があり、ここでは、（初期化時と同じ）入力 f32、出力 cf32 を使用する

出力に、vfreq_ch0 を指定しているので、(4)で定義したベクターが出力となります。(4)で定義したベクターの実体は、(2)で定義しているので、FFT の結果は
fft_output_ch0[n].re （実部）
fft_output_ch0[n].im （虚部）
に格納されます。

(1)~(9)の部分は、準備に相当しますので、最初に 1 回実行してください。

FFT を繰り返して実行する場合は、(1)の入力の配列

fft_input_ch0[n]

にデータを格納して、(10)の FFT 実行を行ってください。

逆 FFT(IFFT)を掛ける処理は、以下の手順となります。

(1)IFFT の入力となるデータの配列を用意する（ここでは、float 型の複素数で、サンプリング数/2(=512)の配列）

static cplxf32_t ifft_input_ch0[FFT_SAMPLING_DATA_NUM/2];

テンプレートのプログラムでは、FFT の出力と IFFT の入力は兼ねているため fft_output_ch0 を使用している

(2)IFFT の出力となるデータの配列を用意する（ここでは、float 型で、サンプリング数(1024)の配列）

static float ifft_output_ch0[FFT_SAMPLING_DATA_NUM];

(3)IFFT の入力のベクタを用意する

static vector_t vfreq_ch0 = {FFT_SAMPLING_DATA_NUM/2, (void *)ifft_input_ch0};

テンプレートのプログラムでは、FFT の出力と IFFT の入力は兼ねているため fft_output_ch0 を使用している

(4)IFFT の出力のベクタを用意する

static vector_t vtime2_ch0 = {FFT_SAMPLING_DATA_NUM, (void *)ifft_output_ch0};

(5)IFFT で使用するワーク変数を用意する

static cplxf32_t fft_twiddles2[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];

static uint32_t fft_bitrev2[240];　　//1024 point のバッファサイズ

(6)ワーク変数をまとめた構造体を定義する（ここでは"h2"という名称で構造体を定義）

static r_dsp_fft_t h2 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles2, fft_bitrev2, NULL, NULL};

(7)ワーク変数のサイズが足りているかの確認[オプション]

size_t ntwb2; // bytes for twiddle array

size_t nbrb2; // bytes for bit-reverse table

size_t nwkb2; // bytes for working area

sts2 = R_DSP_FFT_BufSize_cf32f32(&h2, &ntwb2, &nbrb2, &nwkb2);

使用するワークエリアのサイズの確認。（容量が足りている事が明らかであれば不要）

(8)IFFT 構造体の初期化
sts2 = R_DSP_FFT_Init_cf32f32(&h2);

IFFT の入力として、cf32（32bit 浮動小数点、複素数）。IFFT の出力として、f32（32bit 浮動小数点）。
※入力、出力は様々な型があり、ここでは、入力 cf32、出力 f32 を使用する

(9)IFFT の実行
sts2_ch0 = R_DSP_IFFT_CCS_cf32f32(&h2, &vfreq_ch0, &vtime2_ch0);

構造体、入力ベクタ（横軸：周波数のデータ）、出力ベクタ（横軸：時間のデータ）。
※入力、出力は様々な型があり、ここでは、（初期化時と同じ）入力 cf32、出力 f32 を使用する

出力に、vtime2_ch0 を指定しているので、(4)で定義したベクターが出力となります。(4)で定義したベクターの実体は、(2)で定義しているので、FFT の結果は
ifft_output_ch0[n]
に格納されます。

(1)~(8)の部分は、準備に相当しますので、最初に 1 回実行してください。

IFFT を繰り返して実行する場合は、(1)の入力の配列（複素数）

ifft_input_ch0[n].re
ifft_input_ch0[n].im

にデータを格納して、(9)の IFFT 実行を行ってください。

ここでは、入力、出力とも浮動小数点型の FFT/IFFT を使用しています。（他にも、入出力の型違いのバリエーションがあります）。

出力に、vtime2_ch0 を指定しているので、(4)で定義したベクターが出力となります。(4)で定義したベクターの実体は、(2)で定義しているので、IFFT の結果は
ifft_output_ch0
に格納されます。

2. チュートリアル

TEMPLATE¥RX65_SOUND_EFFECTOR_TEMPLATE_LIB

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、
ch0：シンセサイザ入力
ch1：ボーカル入力
ボーカルをロボットボイスに変換して、シンセサイザに合わせる（ボコーダ）

というプログラムを作成してみます。

ch0 側の入力にはシンセサイザ等の楽器を接続し、ch1 側の入力にはマイクでボーカルを入力してみてください。（ch1 側にダイナミックマイクを接続する場合は、ゲイン切り替えで高ゲインを選択してください。
出力は、モノラル出力（ch0 と ch1 から同じ音声出力）となりますので、片方のみをモニタスピーカに接続しても構いません。

信号処理関しては、参考文献「サウンドプログラミング入門」の 7 章
7.5 ボコーダ
ex7_4.c
を参考にしていますので、参照頂きたく。

テンプレートを、コピーし、

[folder]¥RX65_SOUND_EFFECTOR_VOCORDER

というフォルダ名に変更する事とします。

[folder]¥RX65_SOUND_EFFECTOR_VOCORDER¥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)
{

    unsigned long prev_inbuf_wp;

    long inbuf_index;

    pcm_data pcm_tmp;

    int n, k, offset;
    int band, number_of_band, band_width;

    double a;

    float data_tmp;

    static float fft_input_ch0[FFT_SAMPLING_DATA_NUM];//time-domain, FFT入力データ
    static float fft_input_ch1[FFT_SAMPLING_DATA_NUM];

    static float prev_output_data[FFT_SAMPLING_DATA_NUM/2];//1世代前のIFFT結果

    static cplxf32_t fft_output_ch0[FFT_SAMPLING_DATA_NUM/2];//frequency-domain, FFT出力
データ
    static cplxf32_t fft_output_ch1[FFT_SAMPLING_DATA_NUM/2];

    static cplxf32_t ifft_input[FFT_SAMPLING_DATA_NUM/2];//frequency-domain, IFFT入力デー
タ

    static float ifft_output[FFT_SAMPLING_DATA_NUM];//time-domain, IFFT出力データ

    static float fft_output_mag_ch1[FFT_SAMPLING_DATA_NUM/2];//ch1 FFT結果の振幅
(sqrt(re^2+im^2))

    static vector_t vtime_ch0 = {FFT_SAMPLING_DATA_NUM, (void *)fft_input_ch0};
    static vector_t vtime_ch1 = {FFT_SAMPLING_DATA_NUM, (void *)fft_input_ch1};

    static vector_t vfreq_ch0 = {FFT_SAMPLING_DATA_NUM/2, (void *)fft_output_ch0};
    static vector_t vfreq_ch1 = {FFT_SAMPLING_DATA_NUM/2, (void *)fft_output_ch1};

    static vector_t vfreq2 = {FFT_SAMPLING_DATA_NUM/2, (void *)ifft_input};
    static vector_t vtime2 = {FFT_SAMPLING_DATA_NUM, (void *)ifft_output};

    static vector_t vfreq_mag_ch1 = {FFT_SAMPLING_DATA_NUM/2, (void
*)fft_output_mag_ch1};

    static cplxf32_t fft_twiddles1[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];
    static uint32_t fft_bitrev1[240];     //1024 pointのバッファサイズ

    static float window[FFT_SAMPLING_DATA_NUM];//窓関数

    static cplxf32_t fft_twiddles2[FFT_SAMPLING_DATA_NUM + FFT_SAMPLING_DATA_NUM/2];
    static uint32_t fft_bitrev2[240];   //1024 pointのバッファサイズ

    static r_dsp_fft_t h1 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles1, fft_bitrev1,
NULL, window};//FFT向け 窓関数を含める

    static r_dsp_fft_t h2 = {FFT_SAMPLING_DATA_NUM, 0, fft_twiddles2, fft_bitrev2,
NULL, NULL};//IFFT向け
```

FFT 入力データ(ch0, ch1)

FFT 出力データ(ch0, ch1)

IFFT 入力データ(1ch)

IFFT 出力データ(1ch)

ベクタ等の定義

```
    r_dsp_status_t sts1 = R_DSP_STATUS_OK;//FFT初期化
    r_dsp_status_t sts2 = R_DSP_STATUS_OK;//IFFT初期化
    r_dsp_status_t sts1_ch0 = R_DSP_STATUS_OK;//FFT
    r_dsp_status_t sts1_ch1 = R_DSP_STATUS_OK;
    r_dsp_status_t sts2_mono = R_DSP_STATUS_OK;//IFFT

    /* initialize required buffer sizes for twiddles etc. */
    size_t ntwb1; // bytes for twiddle array
    size_t nbrb1; // bytes for bit-reverse table
    size_t nwkb1; // bytes for working area

    size_t ntwb2; // bytes for twiddle array
    size_t nbrb2; // bytes for bit-reverse table
    size_t nwkb2; // bytes for working area

    //1世代前のIFFT結果の初期化
    for(n=0; n<FFT_SAMPLING_DATA_NUM/2; n++)
    {
        prev_output_data[n] = 0;
    }

    //窓関数定義 ハニング窓
    for(n = 0; n < FFT_SAMPLING_DATA_NUM; n++)//FFT_SAMPLING_DATA_NUMは偶数
    {
        window[n] = 0.5 - 0.5 * cosf(2.0 * M_PI *
((float)n/(float)FFT_SAMPLING_DATA_NUM));
    }

    //FFT構造体初期化
    sts1 = R_DSP_FFT_BufSize_f32cf32(&h1, &ntwb1, &nbrb1, &nwkb1);

    if((sizeof(fft_twiddles1) < ntwb1) || (sizeof(fft_bitrev1) < nbrb1))
    {
        while(1); /* not enough memory size */
    }

    sts1 = R_DSP_FFT_Init_f32cf32(&h1);

    //IFFT構造体初期化
    sts2 = R_DSP_FFT_BufSize_cf32f32(&h2, &ntwb2, &nbrb2, &nwkb2);

    if((sizeof(fft_twiddles2) < ntwb2) || (sizeof(fft_bitrev2) < nbrb2))
    {
        while(1); /* not enough memory size */
    }

    sts2 = R_DSP_FFT_Init_cf32f32(&h2);

    band_width = 8;
    number_of_band = FFT_SAMPLING_DATA_NUM / 2 / band_width;

    //初期化関数
    sound_effector_init();

    //動作開始
    sound_effector_start();
```

FFT の入力にハニング窓を適用する

```
    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
        {
            //FFT
            if(prev_inbuf_wp != g_inbuf_wp)
            {

                //1回目 (-512)-511のデータを処理，ダミー
                //2回目 0-1023データを処理
                //3回目 512-1535のデータを処理
                //4回目 1024-2047のデータを処理
                //→512ずつオーバラップさせてデータを処理する

                //入力データをFFT構造体内の配列変数にコピー
                for(n=0; n<FFT_SAMPLING_DATA_NUM; n++)
                {
                    //この時点で、inbuf_wp(=prev_inbuf_wp +(FFT_SAMPLING_DATA_NUM/2))までは
                    //入力データが格納済み
                    //→prev_inbuf_wp - (FFT_SAMPLING_DATA_NUM/2) ～ prev_inbuf_wp -
(FFT_SAMPLING_DATA_NUM/2) + FFT_SAMPLING_DATA_NUM
                // (prev_inbuf_wp-512 ～ prev_inbuf_wp+512(=inbuf_wp) の範囲のデータを読み込む
事は問題がない

                    inbuf_index = prev_inbuf_wp - FFT_SAMPLING_DATA_NUM/2 + n;
                    if(inbuf_index < 0) inbuf_index += INBUF_SIZE;
                    else if(inbuf_index >= INBUF_SIZE) inbuf_index -= INBUF_SIZE;

                    fft_input_ch0[n] = (float)g_inbuf[inbuf_index].ch0 / 32768.0;

                    fft_input_ch1[n] = (float)g_inbuf[inbuf_index].ch1;

                    /* プリエンファシス処理（Vocal側） */
                    //1サンプル前の信号
                    inbuf_index--;
                    if(inbuf_index < 0) inbuf_index += INBUF_SIZE;
                    fft_input_ch1[n] -= (float)g_inbuf[inbuf_index].ch1 * 0.98;

                    fft_input_ch1[n] /= 32768.0;
                }

                //FFT vtime -> vfreq, 時間領域から周波数領域に変換
                sts1_ch0 = R_DSP_FFT_f32cf32(&h1, &vtime_ch0, &vfreq_ch0);
                sts1_ch1 = R_DSP_FFT_f32cf32(&h1, &vtime_ch1, &vfreq_ch1);
```

Push-SW が押されているときは、スルーで出力

ch0（シンセサイザ側）は無加工（正規化のみ）で FFT 入力変数にコピー

ch1（ボーカル側）はプリエンファシス＋正規化で FFT 入力変数にコピー

ch0 と ch1 の両方に FFT 処理を実行

```
                //信号処理

                //sqrt(re^2 + im^2)の計算（高速型か精密型のどちらか一方を使用）
                //R_DSP_VecCplxMag_Fast_cf32f32(&vfreq_ch1, &vfreq_mag_ch1,
FFT_SAMPLING_DATA_NUM/2);//ライブラリ関数（高速型）を使用（sqrtを使用せず近似計算）
                R_DSP_VecCplxMag_cf32f32(&vfreq_ch1, &vfreq_mag_ch1, FFT_SAMPLING_DATA_NUM/2);
//ライブラリ関数（精密型）を使用
```

ch1 側の FFT 結果の振幅(√(実部 ^{2}+虚部 ^{2}))を算出(*1)

```
                //0-511のバンドから、0-7, 8-15, ..., 504-511の64のバンドの周波数エンベロープを算出
                for(band=0; band<number_of_band; band++)//number_of_band=64
                {
                    offset = band_width * band;//band_width=8

                    a = 0.0;
                    for(k=0; k<band_width; k++)
                    {
                        a += fft_output_mag_ch1[offset + k];
                    }
                    a /= band_width;
                    for(k=0; k<band_width; k++)
                    {
                        fft_output_mag_ch1[offset + k] = a;
                    }
                }
                fft_output_mag_ch1[0] = 0.0;//fft振幅の最初の項をゼロとする
```

ch1 側の FFT 結果を 64 の帯域に分割してそれぞれの振幅を算出

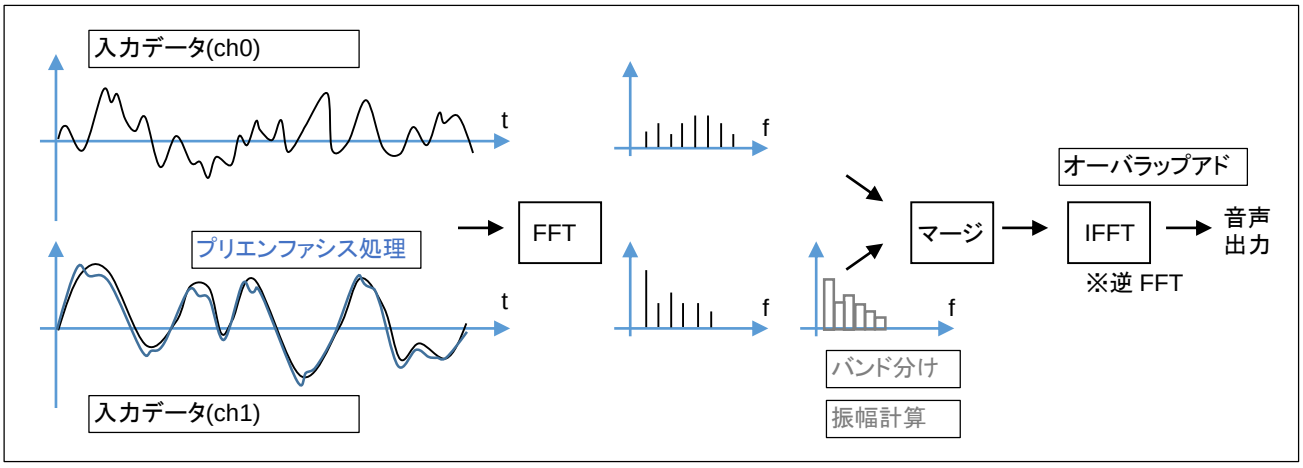
```
                /* フィルタリング */
                for(k=0; k<FFT_SAMPLING_DATA_NUM/2; k++)
                {

                    ifft_input[k].re = fft_output_ch0[k].re * fft_output_mag_ch1[k];//虚部の演算
を省略
                    ifft_input[k].im = fft_output_ch0[k].im * fft_output_mag_ch1[k];//虚部の演算
を省略
                }
```

ch0(シンセサイザ)と ch1(ボーカル)の加工結果をマージ

```
                //IFFT vfreq -> vtime2, 周波数領域から時間領域に逆変換
                sts2_mono = R_DSP_IFFT_CCS_cf32f32(&h2, &vfreq2, &vtime2);

                //IFFT結果の連結
                for(n=0; n<FFT_SAMPLING_DATA_NUM/2; n++)
                {
                    //前半の512データは、前回の演算結果と加算して出力バッファに格納

                    data_tmp = (prev_output_data[n] + ifft_output[n]);

                    //クリッピング処理，モノラルデータをch0，ch1に出力
                    pcm_tmp = clipping_data_norm(data_tmp, data_tmp);

                    put_data(pcm_tmp);
                }

                for(n=0; n<FFT_SAMPLING_DATA_NUM/2; n++)
                {
                    //後半の512データは、変数に一時保存
                    prev_output_data[n] = ifft_output[n + FFT_SAMPLING_DATA_NUM/2];
                }

                prev_inbuf_wp = g_inbuf_wp;
            }
        }

    }//while
}
```

逆 FFT を実行して、時間軸のデータ(音声)に変換

オーバラップアドの処理は、テンプレートと同じ
(このサンプルでは、扱うデータは 1ch)



大まかな処理の流れは上図となります。

本チュートリアルでは、FFT, IFFT 以外に、振幅の計算(*1)でもライブラリ関数を使用しています。

//sqrt(re^2 + im^2)の計算（高速型か精密型のどちらか一方を使用）

//R_DSP_VecCplxMag_Fast_cf32f32(&vfreq_ch1, &vfreq_mag_ch1, FFT_SAMPLING_DATA_NUM/2);
//ライブラリ関数（高速型）を使用（sqrtを使用せず近似計算）

R_DSP_VecCplxMag_cf32f32(&vfreq_ch1, &vfreq_mag_ch1, FFT_SAMPLING_DATA_NUM/2);
//ライブラリ関数（精密型）を使用

この関数は、入力(vfreq_ch1)の、振幅

$$\mathrm{mag} = \sqrt{Re^2 + Im^2}$$

を計算する関数です。（Re:FFT 結果の実部、Im：FFT 結果の虚部）Fast が付く関数は、高速型で時間の掛かる平方根の計算を省き高速に処理が行える関数です。

今回のプログラムでは、平方根の計算を行う精密型でも、時間内（512 サンプル毎のデータを 10.66ms 以内に処理）できたため、精度の良い精密型を使用しています。

※FFT, IFFT で処理するデータ数は 1024 としていますが、512 個のデータが入力される毎に 1024 の半サイクルずらしでフレーム切り出しを行い、データ処理を行っています。そのため、512 データ毎（20.83us x 512=10.66ms 毎）に 1 フレーム（1024 データ）を処理する必要があります。

3. 関数、グローバル変数、定数仕様

TEMPLATE¥RX65_SOUND_EFFECTOR_TEMPLATE_LIB

で定義されている関数等は

TEMPLATE¥RX65_SOUND_EFFECTOR_TEMPLATE

のものと同一ですので、サウンドエフェクタ ソフトウェア編(2)を参照してください。

4. まとめ

本マニュアルでは、ルネサスエレクトロニクスで公開されている、「RX ファミリ用 DSP ライブラリ」を使用したケースを説明致しました。

FFT や IFFT 等の処理を、「簡単に」「高速で」処理できるのがライブラリを使用するメリットです。

フィルタや、DFT、FFT の他、本マニュアルのチュートリアルで使用した、複素数の振幅を求める関数等が用意されています。

速度的に計算が間に合わない場合や、手軽に計算結果を得たい場合は、ライブラリの活用を検討してみてください。

5. 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(3)
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)