

0. はじめに

鍵盤を弾けなくてもツマミを回したりボタンを押しただけで「いい感じ」の音が出てくる。

これが「ぴゅんぴゅん 3 号」の目標です。

開発に必要なソースコード等、資料は

¥PSoC-PyunPyun-3rdLPF-master ¥

に収録しました。

また、ぴゅんぴゅん 3 号で作った音をライン録りして Wave ファイルにしたものを

¥Wave¥

に収録しました。

1. ぴゅんぴゅんマシンとは

最近はまだすっかり話題にものぼらなくなった気もしますが「初音ミク」が発売されるか少し前、若い子に「どんな音楽聴いてる？」と聞くと「レゲエ！」とか「トランス！」とか「V 系！」とか割とちゃんとした答えが返ってきました。

レゲエと言ってもおっさん世代はボブ・マーリーで止まっていたわけがよくわからなかったのですが、貸してもらった CD のコンピレーションアルバムに、2 ちゃんねるでたまたま見かけて気になっていた「ぴゅんぴゅんマシン」の音が入っていました。

「ぴゅんぴゅんマシン」とは言葉で説明すると難しいのですが、「ぴゅ〜ん」とか「ぴゅんぴゅんぴゅんぴゅん」とか、とてもなまけない音がする「電子楽器」です。

海外では「Siren Machine」と呼ばれているようです。

原理としては、オシレーターで発生させる波形に LFO で周波数変調を掛けるというもので、ミニマルなシンセサイザーと言えるかもしれません。

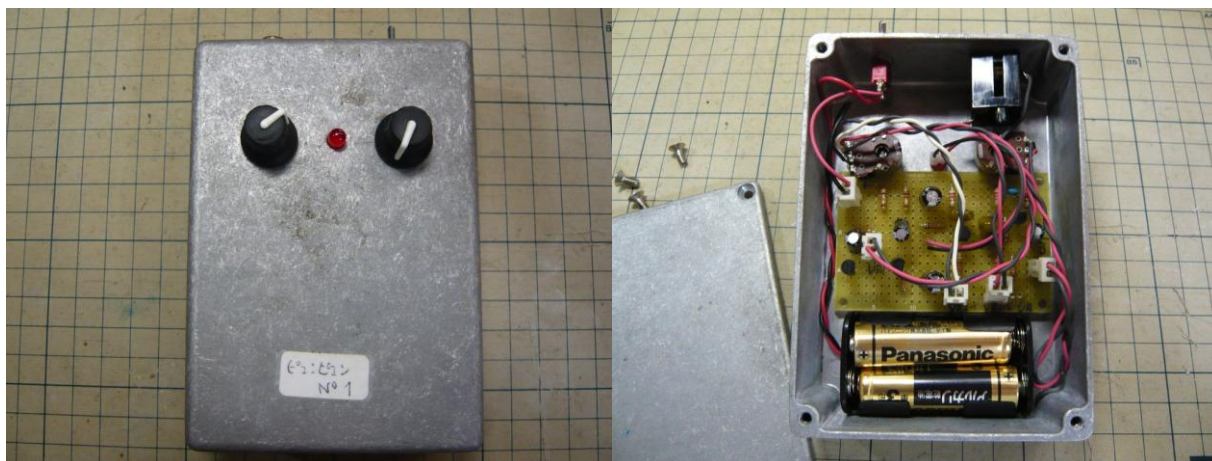
エロゲ会社で仕事をさぼりながらループ音源で曲を作っていた身にとっては「ぴゅんぴゅんマシン」は大ヒットでした。

ぴゅんぴゅんマシンが欲しくなってネットで調べると売っているけど、めっちゃ高いではないか！

自作しようと思って探してもぴったり来る解説や回路図が全然見つからない！！

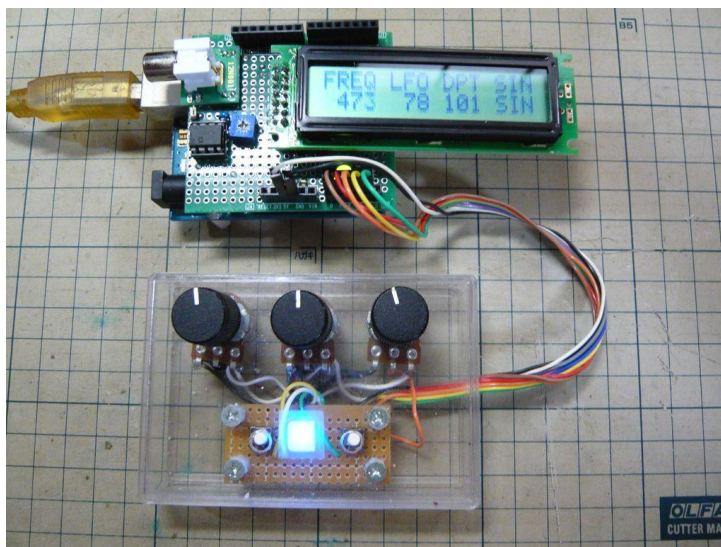
似たような音はがんばれば出せるかもしれないので、一から自分で作ってみようというのがそもそもの動機でした。

フル・ディスクリットで作ったぴゅんぴゅん 1 号



トランジスタ回路で作成しました。「太い」音がします。

Arduino で作ったぴゅんぴゅん 2 号



パラメーターをいじっていろいろな音を出せることを目標に製作しました。

ぴゅんぴゅん 1 号と 2 号については、去年の夏コミで頒布した「はじめてのぴゅんぴゅんマシン」で紹介しています。

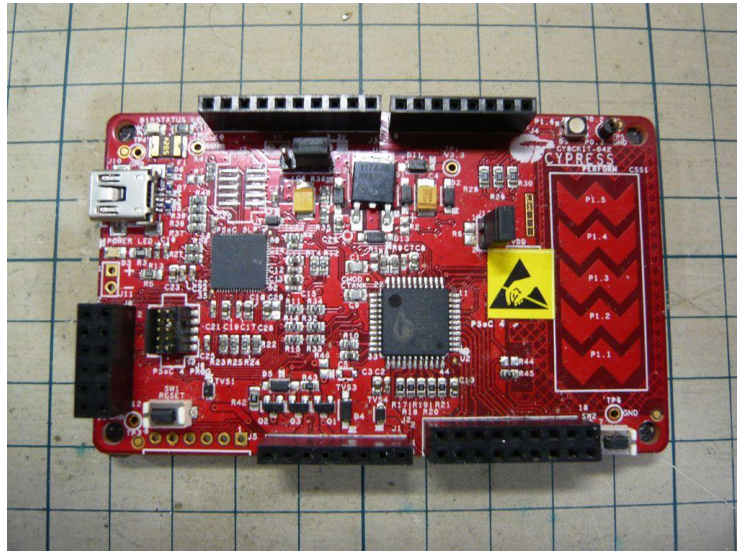
<https://github.com/ryood/C86>

2. PSoC とは

ぴゅんぴゅん 3 号は、PSoC ベースで作ることにしました。

PSoC とは Cypress 社が出しているマイコン+プログラム可能なデジタル・ブロックとアナログ・ブロックという構成のなかなか変態な IC（マイコン?）です。

シリーズとしては、PSoC1、PSoC3、PSoC4、PSoC5LP とあり、それぞれ特徴がありますが、2〜3 年ぐらい前に「PSoC 4 Pioneer Kit」という開発ボードが発売されました。



このボードの特徴は3つ。

- Arduino 互換のソケットが出ているので Arduino のシールドが使える
- PSoC のカスタマイズ可能な周辺機能が使える
- ARM (32bit CPU) のパワーの恩恵に預かれる

という点です。

コアは ARM の Cortex-M0 なので気にせず 32bit 演算できて、そのうえデジタル・ブロックやアナログ・ブロックを自分好みにカスタマイズして使えるというわけです。これは使わない手はないなと思って、ぴゅんぴゅん 3 号は「PSoC 4 Pioneer Kit」で製作しました。

3. PSoC 4 Pioneer Kit を使ってみる

インストール方法

必要環境：

- PSoC 4 Pioneer Kit
- Windows の動くパソコン

まずは「**PSoC 4 Pioneer Kit**」を購入。3000 円ぐらいです。Arduino Uno と同じぐらいの値段です。「PSoC 4 **Prototyping Kit**」という 600 円ぐらいで売っているものもあって、これはこれでコスパが高いのですが、ドキュメントがいまいちなので少し敷居が高いです。

樂をしたいなら「Pioneer Kit」です。

PSoC の開発には Cypress から無償で提供されている「PSoC Creator」が必要です。

「PSoC 4 Pioneer Kit」の箱に書いてあるとおり

www.cypress.com/go/CY8CKIT-042

にアクセスして

CY8CKIT-042 Kit Setup

をダウンロードしてインストールします。

※途中で CYPRESS のアカウントの作成を要求されたり、「Akamai NetSession Interface」のインストールを要求されますので従いましょう。

インストール後しばらくして「Cypress Update Manager」が起動して Update を促されると思いますが、ドキュメントと機能が乖離するので、わからなければ慣れるまで Update はキャンセルしておいた方が無難だと思います。

PSoC 4 Pioneer Kit で使われている PSoC4200 CY8C4245AXI-433 のデータシートは、別途

<http://www.cypress.com/documentation/datasheets/psoc-4-psoc-4200-family-datasheet-programmable-system-chip-psoc>

からダウンロードできます。

PSoC 4 Pioneer Kit はボード上のジャンパの設定で、5V 駆動と 3.3V 駆動を切り替えられます。ジャンパーの位置は「CY8CKIT-042 Kit Setup」に入っている「CY8CKIT-042 Quick Start Guide」の 2 ページ目に載っている画像の「System Power Supply Jumper [J9]」のところで、ボード上にもシルクスクリーンで表示されています。

今回は 3.3V 駆動の LCD を使うため、3.3V 駆動に設定しました。

4. 音を作る

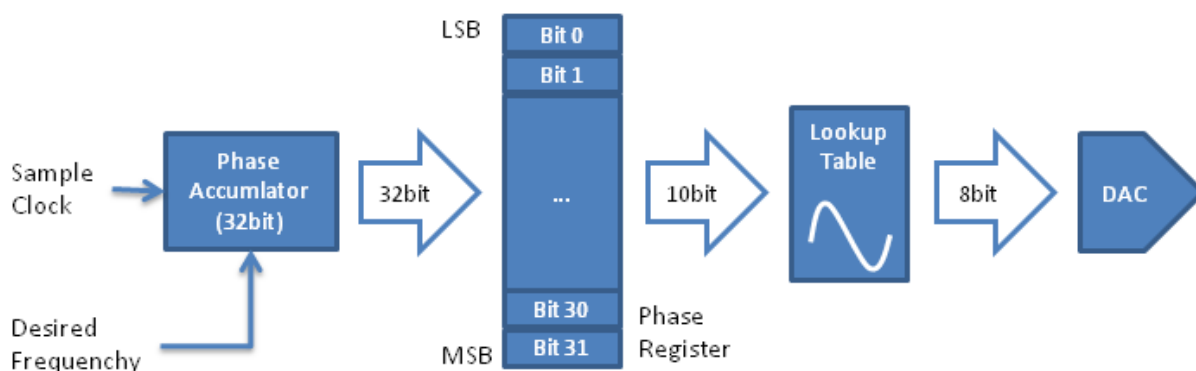
デジタル回路には DDS がいちばん？

デジタル的に波形（音）を生成する方法はいろいろありますが、ぴゅんぴゅん 3 号では DDS（Direct Digital Synthesizer）という手法をとっています。

DDS のメリットは演算量が非常に少なくて済むという点です。

「**楽器**」として使うならリアルタイムに波形を生成しないといけないので複雑な演算をしていては波形生成が追いつかなくなります。

Direct Digital Synthesis(DDS)



一見、複雑そうに見えますが、あらかじめ周波数(Desired Frequency)を「tuningWord」に変換したり、「waveTable」(Lookup Table)をメモリ上に準備しておけば、周波数を指定して波形を簡単に生成できます。

ぴゅんぴゅん 3 号のコードで実際に基本波形を生成している部分は

```
// Caluculate Wave Value
//
phaseRegister += tuningWord;

// 32bit の phaseRegister をテーブルの 10bit(1024 個)に丸める
index = phaseRegister >> 22;
waveValue = *(waveTables[waveShape] + index);
```

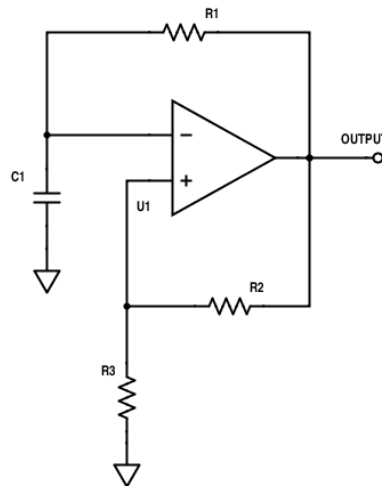
これだけです。

計算して出てきた「waveValue」は出力波形の電位のデジタル値なので、DAC でアナログ値の電位に変換すれば「音」として扱える様になります。

LFO について

「Low Frequency Oscillator」の略で日本語に訳すと「低周波発振器」という意味になります。LFO と言われても「別に普通のオシレーターでいいやん?!」と思ってましたが、1 つの発振回路構成で可聴帯域の発振と、可聴帯域以下 (0.1Hz~10Hz あたり) の発振を担わせようと思うと困難な問題に直面することがあります。

アナログ回路だと CR 発振器の C (コンデンサ) の値の取り方でまともに発振してくれる R (可変抵抗器) の値の範囲が決まります。



この回路は OPAMP を使った矩形波の発振回路で、発振周波数は、

$$f_0 \approx \frac{1}{2C_1 R_1 \ln\left(1 + \frac{2R_3}{R_2}\right)} [\text{Hz}]$$

で決まりますが、現実世界では C1 の容量や特性で使える R の値の範囲が決まり、発振周波数の範囲が制限されます。

なので、作りたい周波数帯域に合わせてちょうどいい C1 を選択する必要があります。

デジタル回路でも 1 回で計算できる桁数は固定なので、ムリをすると破綻してめちゃくちゃな計算結果が出てくることがあります。

なので、「聴こえる周波数」と「うねりを作る周波数」を分けて回路やプログラムを作ってやればいいのか?

ということで低周波専用の発振器が登場することになります。

DDS なら簡単で、tuningWord を算出するときのサンプリング・レートを下げてやれば LFO は実現できます。

DDS の最低出力周波数は

$f_{out} = (\text{tuningWord} / 2^n) * \text{SAMPLING_RATE};$

(f_{out} : 出力周波数, n : tuningWord のビット長)

で求められます。

($\text{tuningWord} / 2^n$)が負の値を取ると出力周波数が負の値になっておかしいことになります。

2^n は正の整数で tuningWord は整数なので、tuningWord を 0 以上の整数の最小値とすると

$\text{tuningWord} = 1$

となり、16bit 演算で、CD と一緒の 44.1kHz の場合で考えると、 $\text{SAMPLING_RATE} = 44,100$, $n = 16$ なので、生成できる一番低い周波数は

$f_{out} = (1 / 2^{16}) * 44100 = 0.673[\text{Hz}]$

となります。

つまり 16bit 演算でサンプリング・レートが 44.1kHz だと 0.673Hz 以下の周波数は生成できないということになります。

サンプリング・レートを 1/10 の 4,410Hz に下げれば

$f_{out} = (1 / 2^{16}) * 4410 = 0.0673[\text{Hz}]$

まで下の周波数が生成できることになります。

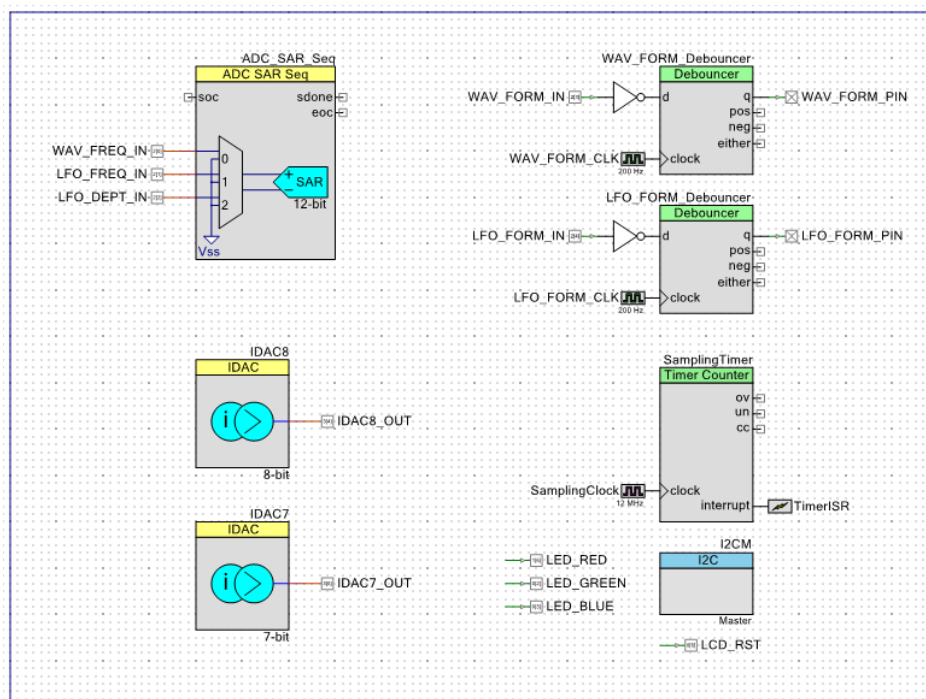
ただか 1/10 ですが、「桁足らず？」で演算できなくなるのをフォローするためには 2 系統作るのがよさそうです。

PSoC4 は 32bit の CPU なのでそれ程厳密に考えなくていいと思いますが、ぴゅんぴゅん 3 号でも余裕を持って Wave 系と LFO 系の 2 系統に分けて演算しています。

5. PSoC 4 Pioneer Kit でプログラミング

コンポーネントの使い方

PSoC のプログラミング方法は普通のマイコンとは少し違います。



PSoC Creator では上図のような「TopDesign」で「コンポーネント」と呼ばれるパーツを組み合わせて普通のマイコンに備わっている周辺機能や、PLD のデジタル回路みたいなものを定義します。

オブジェクト指向で使う UML 図やデジタル回路のシンボルのようにグラフィカルに定義できるので、プログラム言語で記述するよりもパッと見た感じ把握しやすいと思います。

各コンポーネントをダブルクリックすると「Configure」というダイアログが現れてコンポーネントの機能の詳細を設定できます。

コンポーネントの使い方を習得するのが PSoC を使う上で重要事項になります。

ぴゅんぴゅん 3 号で使ったコンポーネント

TIMERCOUNTER

いわゆる Timer です。正確な間隔で割り込みをかけるコンポーネントです。

このタイミングを利用して DDS のサンプリングの再生を行っています。

やっていることは 12MHz のクロックを受けて、250/65535 で分周しているので

$$12(\text{MHz}) * (250 / 65535) = \text{だいたい } 45.777(\text{kHz})$$

あ！やばい！プログラムのサンプリング・レートと合っていない！まあいいか

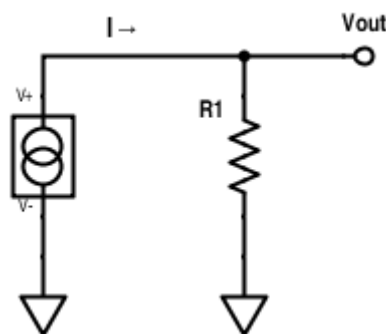
このコンポーネントは波形を生成する大事なクロックを司っているのですが、バグも音の味ということで、そのうち直したいと思います（^ q ^ ;

IDAC

電流出力型の DAC (Digital / Analog Converter) のコンポーネントです。プログラムで生成した波形をアナログ値に変換するために使用しています。

普通の DAC は電圧出力型ですが、PSoC4 には IDAC という電流出力型のコンポーネントしか用意されていません。

電流出力型と言われて最初びびりましたが、電流値を電圧に変換すればいいだけなので基本的には抵抗 1 本でできます。



図の左側の「∞」みたいなシンボルのパーツは電流源です

$$V_{out} = I * R1$$

なので R1 の値で電流出力を電圧に変換できます。

ですが、このままでは後段の回路と干渉するので何か対策する必要があります。

PSoC4 の IDAC の精度は 7bit か 8bit で出力電流は「0-306uA (1.2uA/bit)」か「0-612uA (2.4uA/bit)」を選ぶようになっていて、今回は消費電力を節約するために、「0-306uA」を指定しました。

I2C

パラメーターを表示する LCD との通信に使いました。

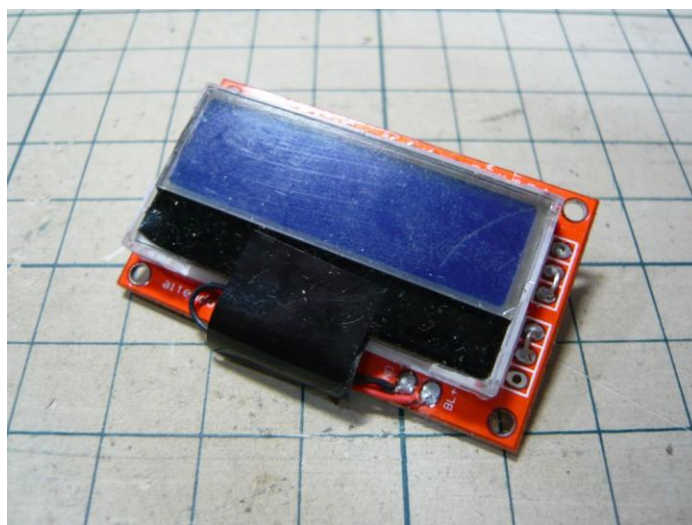
LCD は HD44780 という規格がよく使われていて製作例も多くて使いやすいのですが、接続が平行なのでマイコンから出ているピンをたくさん使ってしまうのが難点です。

また、サイズの小さいものもあまり見かけません。

ぴゅんぴゅん 3 号は Arduino 互換のシールドとして実装するため線数が少なく済んでサイズの小さいものもある I2C の LCD を使うことにしました。

バックライトは暗いところでも視認できるという意味で必須です。

16 桁 2 行、バックライト付でできるだけサイズの小さいものということで aitendo の「SPLC792-I2C-M」を使いました。



この LCD は 375 円と値段もかなりお手頃なのですが、通販だと 1 回に 2 個しか買えません。また、ちょいちょい売り切れます。（なぜだか割と早めに補充される）

同じ LCD モジュールと基板のセットで自分ではんだ付けしないといけないものもありますが、はんだ付けに自信がある人は別として、モジュールと基板がもともとはんだ付けされている「SPLC792-I2C-M」が使いやすいと思います。

「SPLC792-I2C-M」もピンヘッダのほか多少はんだ付けしないといけないので使い方は Blog の記事を参照してください。

aitendo の SPLC792-I2C を Arduino で使う方法

<http://dad8893.blogspot.jp/2015/02/aitendosplc792-i2carduino.html>

PSoC4 では「I2C LCD」用のコンポーネントもありますが「SPLC792-I2C-M」はそのままでは使えない様なので、「I2C」というプリミティブなコンポーネントを使いました。

PSoC4 では「I2C(SCB mode)」というコンポーネントになります。

I2C コンポーネントの設定は「Data rate」ぐらいしかないので 100kbps にしました。

LCD の制御はコンポーネントではなくプログラムに記述しました。

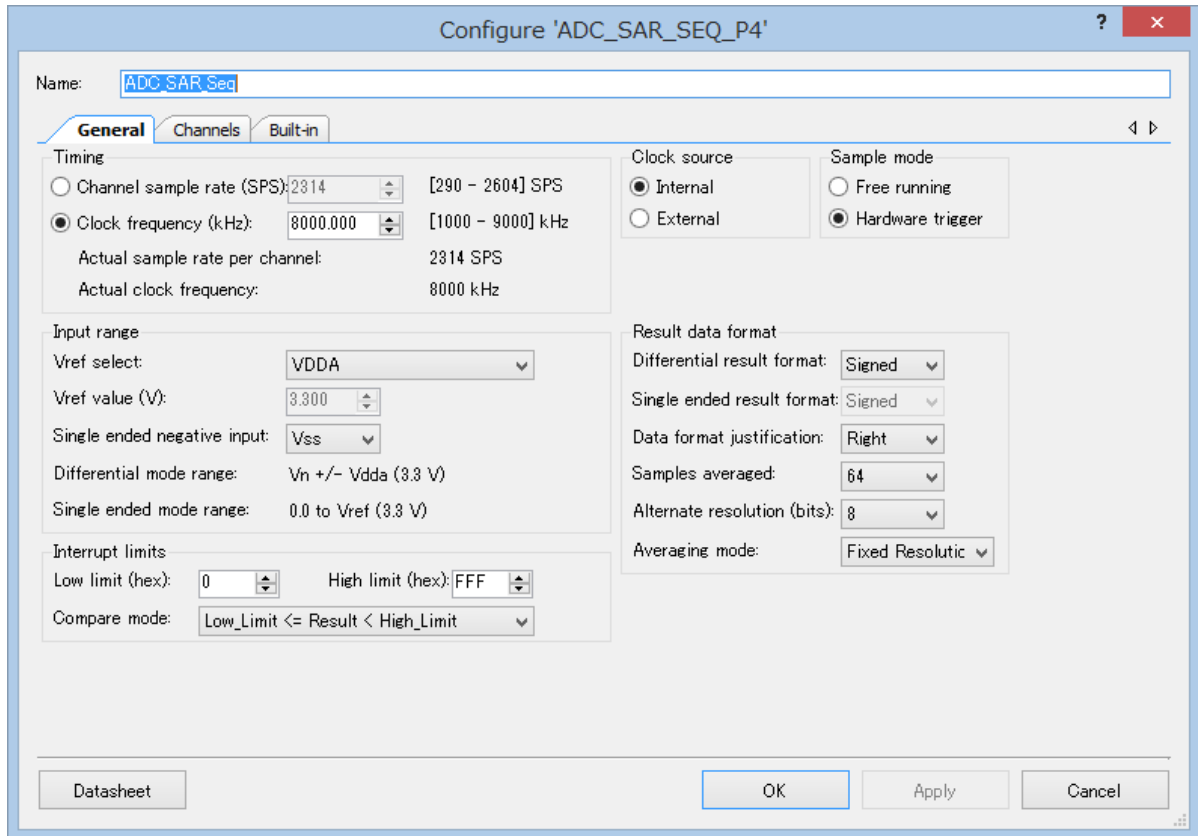
ADC

ADC とは「アナログ→デジタル・コンバーター」の略です。

パラメーター設定用の可変抵抗の値を読み取るのに使用しています。

PSoC4 で使えるコンポーネントは「Sequencing SAR ADC」です。

今回使ったコンポーネントの設定で「ADC」が一番把握するのに苦労しました。



ADC の入力は 8 個まで使えますが、実態は ADC は 1 個しかなく順次処理をしています。使っている入力チャネルの数で処理が分配されるので、入力チャネルが増えると処理能力が落ちます。

ダイアログの左上の「Timing」の「Clock frequency」は 8000kHz (8MHz) を指定していますが、分配した結果としてそれぞれのチャネルで 2314 サンプル/秒の処理ができます、ということになります。

「Input range」の「Vref select」は基準とする上側の電圧で、「VDDA」（アナログの VDD）を指定しているので今回は（だいたい）3.3V になります。

「Single ended negative input」は下側の電圧で「Vss」なので GND レベルになります。

なので制御用の可変抵抗の両端にそれぞれ 0V、3.3V を与えれば ADC でちょうどいい変換ができるということになります。

「Interrupt limits」は、割り込みの閾値の設定で、今回は ADC の割り込みを使っていないので関係ありません。

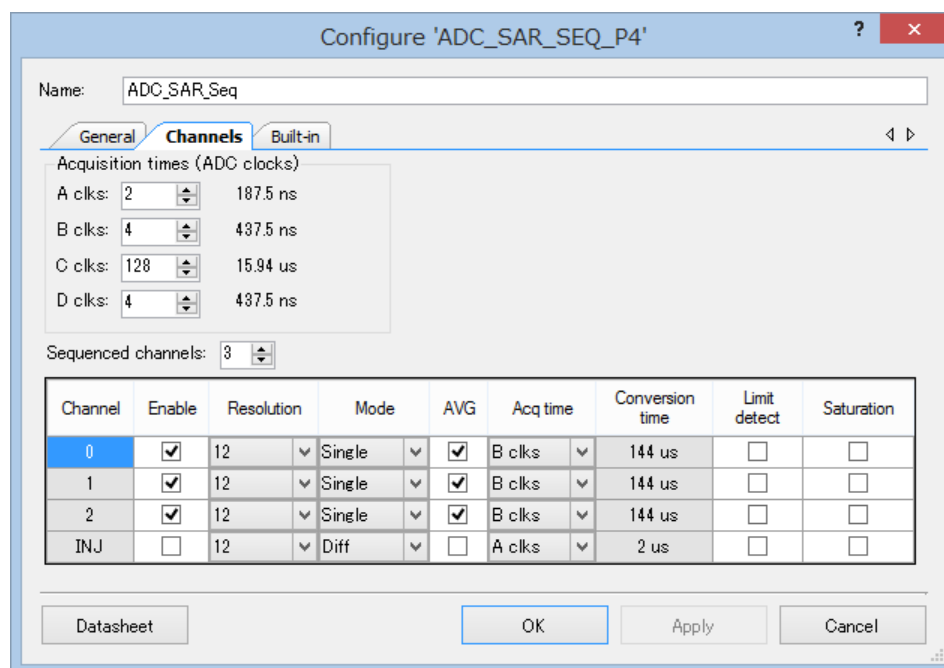
「Clock Source」は「*.cydwr」の「Clock」タブで ADC に設定した値を使うか TopDesign で外部から明示的に与えるかの選択です。

「Sample mode」はハードウェア的に ADC を起動するかどうかの選択ですが、今回はプログラムで（ソフト的に）ADC の動作を制御しているのでどちらでもかまいません。

「Result data format」の値は実際に可変抵抗を操作しながら設定値を決めました。

「Differential result format:」は差動入力時、「Single ended result format:」は片信号入力時の出力フォーマットを Signed にするか Unsigned にするかの設定ですが、今回はすべて片信号入力にしているので、「Differential result format:」の設定は無関係です。

「Single ended result format:」の方はなぜかグレーアウトして「Signed」に固定されているようです。せっかく 12bit ADC ですが、この設定では 11bit (0 - 2048) でしか変換できません。



Channel	Enable	Resolution	Mode	AVG	Acq time	Conversion time	Limit detect	Saturation
0	<input checked="" type="checkbox"/>	12	Single	<input checked="" type="checkbox"/>	B clks	144 us	<input type="checkbox"/>	<input type="checkbox"/>
1	<input checked="" type="checkbox"/>	12	Single	<input checked="" type="checkbox"/>	B clks	144 us	<input type="checkbox"/>	<input type="checkbox"/>
2	<input checked="" type="checkbox"/>	12	Single	<input checked="" type="checkbox"/>	B clks	144 us	<input type="checkbox"/>	<input type="checkbox"/>
INJ	<input type="checkbox"/>	12	Diff	<input type="checkbox"/>	A clks	2 us	<input type="checkbox"/>	<input type="checkbox"/>

「channels」タブの設定値も同様に試行錯誤の結果です。「Sequenced Channels」で入力チャネル数を設定します。

各チャネルの「Enable」にチェックを入れると実際に A/D 変換を行うようになります。「INJ」は自動的に挿入されるチャネルで、今回は使用しないので「Enable」のチェックは外しておきます。

「AVG」は入力を過去の入力値と平均化して出力するかどうかの設定です。平均するサンプル数は「General」タブの「Result data format」の「Samples averaged:」で指定します。

DEBOUNCER

物理的なスイッチは「ON/OFF」の切り替え時に値が安定しない時間があります。チャタリングと呼びます。

これを回避する方法はソフト的なものやハード的なものいろいろありますが、PSoC では「Debouncer」コンポーネントを使えばデジタル・ブロックで回避できるので使ってみました。

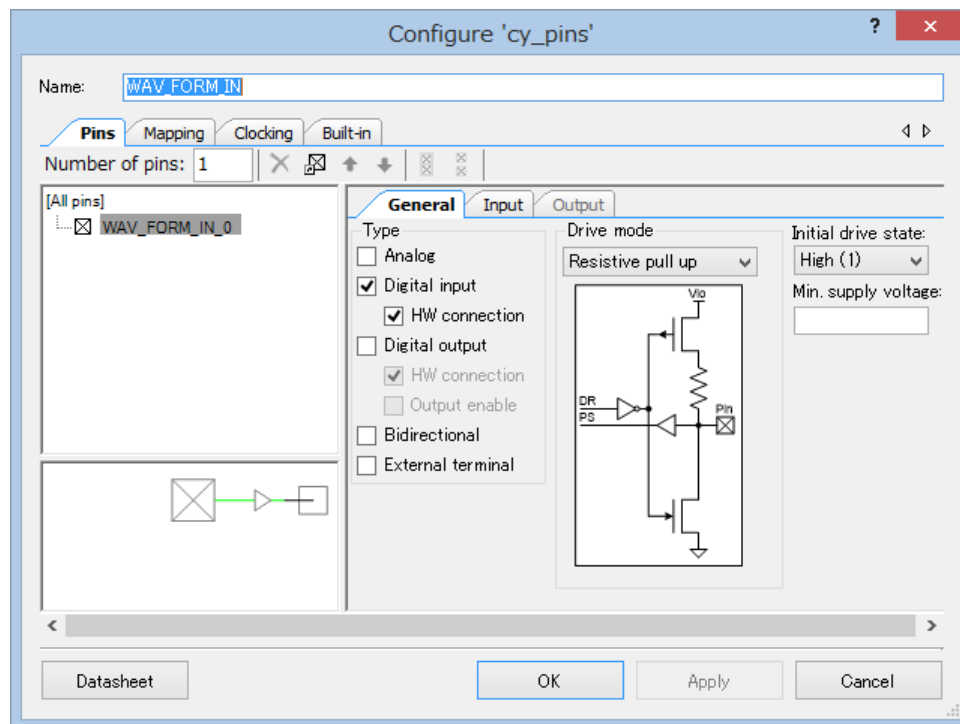
CPU（プログラム）と関係なく動作するのでプログラムで考慮する必要もなく、外付けの回路も一切不要でした。

今回 PSoC のありがたみを一番ストレートに感じられたコンポーネントです。

PIN

「WAV_FORM_IN」、「LFO_FORM_IN」はデジタル入力ピンで、「cy_pins」というコンポーネントです。

オブジェクトをダブルクリックすると「Configure」ダイアログが表示されます。



「General」タブの「Type」を設定すると、アナログ入力用、デジタル入力用、デジタル出力用などのピンの動作を切り替えられますが、普通はオブジェクト生成時に指定するのでここはあまり変更しなくても良いと思います。

「Drive mode」はプルアップ、プルダウン、オープンドレイン等の設定をします。デフォルトだと「Strong drive」になっていますが、このモードだと source、sink 両用の出力ピンとして使えます。

今回は入力デバイスとして使う「ぴゅんぴゅんコントローラー」の仕様により、プルアップに設定しました。

プルアップして使うのでボタンを押した時に L になるので「Debouncer」コンポーネントに入力する前に「Not」コンポーネントを入れて、押し下げ時に H になる様にしています。

割り込みとポーリング

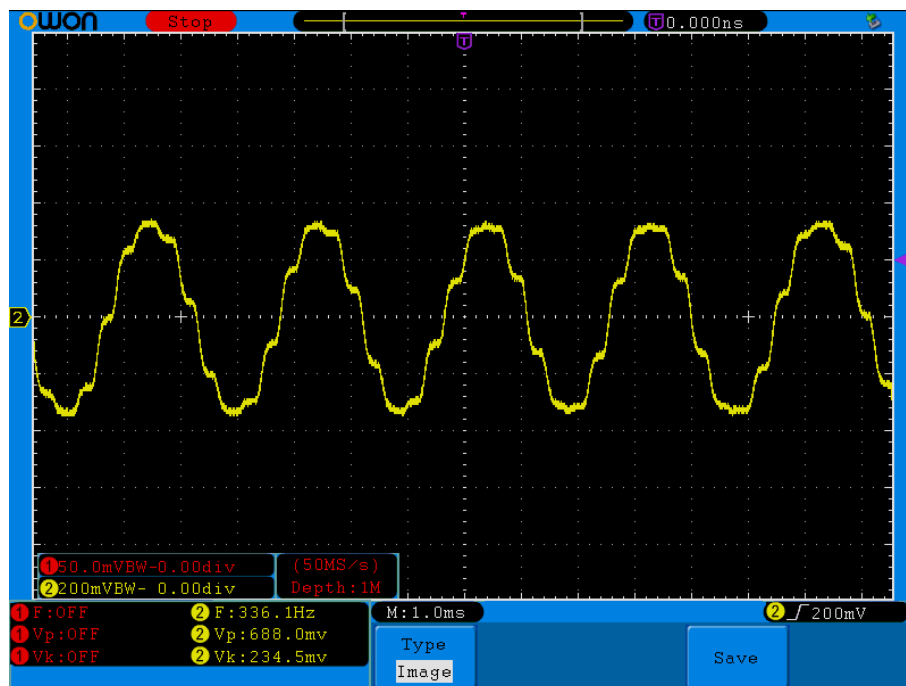
フィジカル・コンピューティングでは割り込みで処理することがほとんどだと思います。

外部のデバイスから何かアラートが上がった時にそれに対する処理をするというのが基本になると思います。

ぴゅんぴゅん 3 号も最初は全部割り込みで処理する様にプログラムしましたが、割り込みが重なると問題が発生しました。

ぴゅんぴゅん 3 号は基本的に波形生成マシンです。

そこにパラメーター設定用の ADC の割り込みがかかると波形が崩れました。



割り込みの優先順位を設定するという手もありそうですが、そもそも可変抵抗によるパラメータ設定は波形生成に比べるとそれほど優先順位が高くありません。

入力のレスポンスが悪化しますが、波形生成のタイミング（Timer 割り込み）だけ割り込み処理にして、他の処理はポーリングで処理することにした。

ポーリングとはプログラムで明示的に外部デバイスに状態を問い合わせる処理です。

こうするとデバイスの入力の受付は、波形生成のための割り込み処理より確実に後回しにされるので、結果として波形の崩れは起こらなくなりました。

6. アナログ回路を作る

LPF (Low pass filter)

PSoC の IDAC は最大 8bit です。



オシロで拡大してみると波形がガタガタです。このまま出力しても面白ノイズが出るのですが、少しはまともな波形を出力することを考えました。

このガタガタをならすためには LPF (ローパス・フィルター) を通すと改善されそうです。

波形を見ると全体的には正弦波ですが、ガタガタのところを見ると矩形波的です。

矩形波は基本波に奇数次の倍音を順次弱めながら加算していけば作れます。

数学的に言うとフーリエ級数で表現できるそうです。

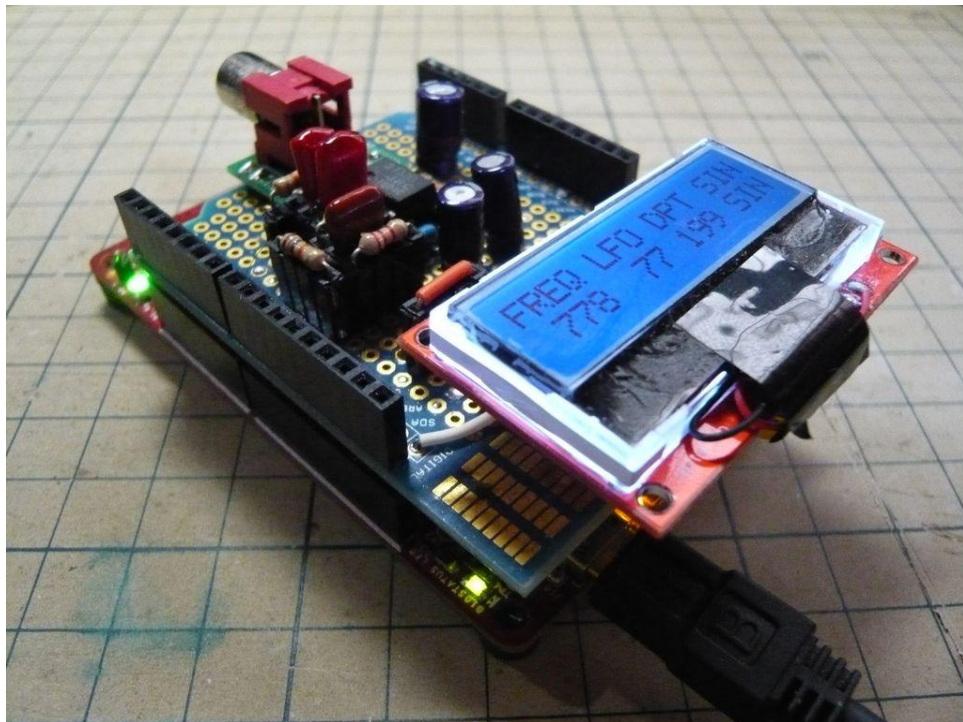
$$f(t) = \frac{\pi}{4} \left\{ \sin \omega t + \frac{1}{3} \sin 3 \omega t + \frac{1}{5} \sin 5 \omega t + \dots \right\}$$

逆に言うと基本波より上の倍音をカットすれば矩形波はならされて正弦波（基本波）に近くなるので、ガタガタは除去されます。

決め打ちした周波数の正弦波ならその周波数より上をカットしてやればいいのですが、ピュンピュン 3 号の基本波は正弦波だけではなく、周波数もツマミをいじって変化させます。

基本波に合わせてカットオフ周波数を変化させるのは大変だし、「聴いてみていい感じ」を目標にしているので単純にスパッと切れば良いというわけでもないで配分がなかなか難しい。デジタル的な歪も気持ち良ければある程度残したい。

というわけで、あとから回路の定数（抵抗やコンデンサーの値）を調整できるように「ピンソケット」を使って抵抗やコンデンサーを差し替えられるようにして実装しました。



出力バッファ

IDAC の電流出力を抵抗で電圧に変換しましたが、そのままではまともに出力できないのでバッファを入れました。

例えて言うと、エレキギターの出力をそのままヘッドホンにつないでもまともに音が出ないようなものだと思います。

（オーディオ系の）アナログ回路では、バッファとは入力インピーダンスが高くて出力インピーダンスが低い回路を言います。

入力インピーダンスが高いので前段の回路が弱っちくても平気、出力インピーダンスが低いので後段の回路が弱っちくても平気という便利な回路です。

こうすると前段と後段の回路を分離して別々に考えられるようになります。

いいことばかりではなく、波形がなまるしノイズも増えるし基板の設計もしんどくなるので理屈に頼って使いすぎると痛い目にあいます。

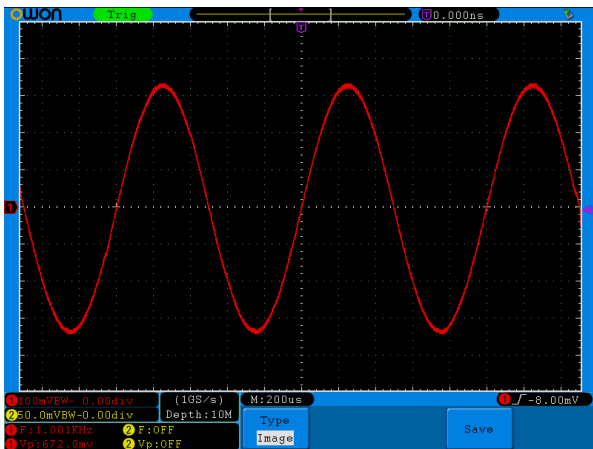
びゅんびゅん 3 号では 3 次 VCVS という回路で「バッファ」と「LPF」を一つの回路で実装しました。

3 次 LPF なので理屈としては 18dB/oct で高域が減衰され、出力のガタガタがならされます。

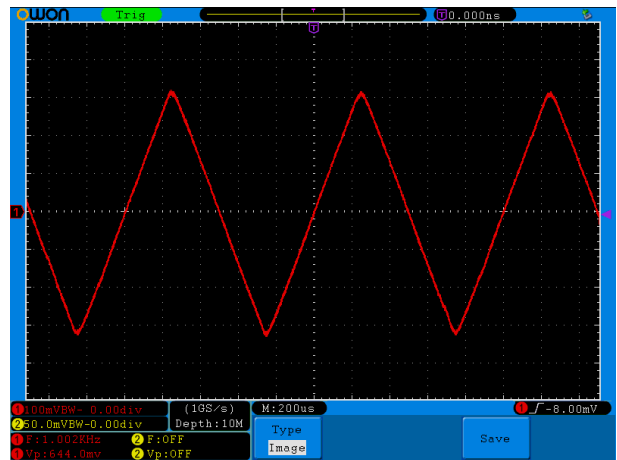
7. 出力波形

LFO をかけない状態でびゅんびゅん 3 号の出力波形をオシロで見てみました

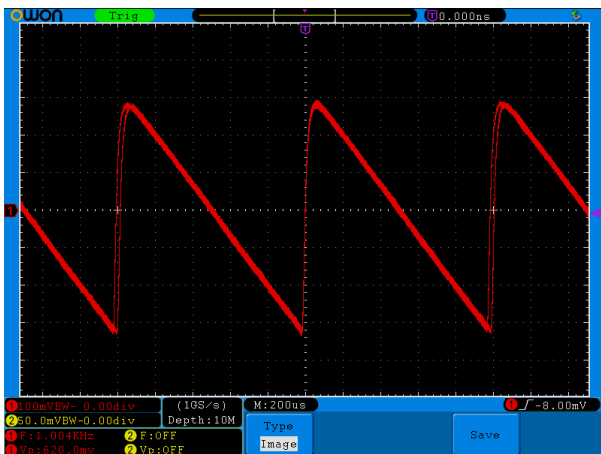
SIN (正弦波)



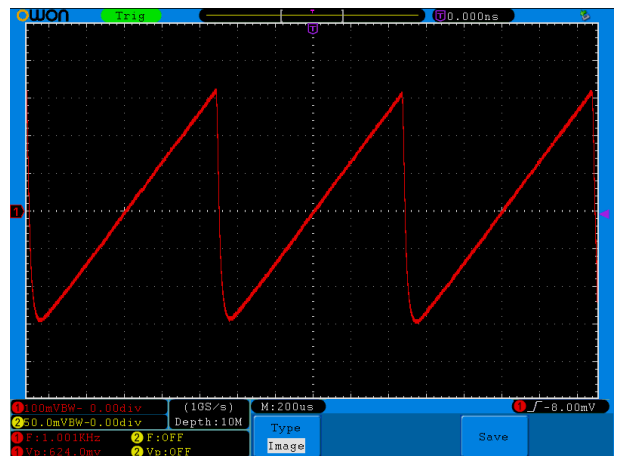
TRI (三角波)



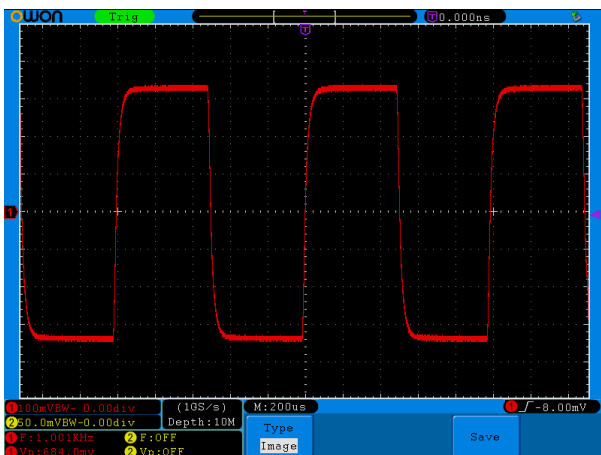
SW1 (ノコギリ波・下降)



SW2 (ノコギリ波・上昇)



SQR (矩形波)



8. 音を出して遊ぶ

これで音を出す準備はできました。

これだけでも「音」は出ます。でも、音が出ただけでは何が面白いのかわかりません。

自分で音を操るのがシンセサイザーの醍醐味です。

鍵盤を弾けなくてもツマミを回したりボタンを押しただけで「いい感じ」の音が出てくる。

これがぴゅんぴゅん3号の目標です。

ぴゅんぴゅんコントローラー



可変抵抗が3個、ボタンが2個という構成です。可変抵抗の動作は

基本波の周波数を可変 | LFOの周波数を可変 | LFOの深さを可変

という設定にしています。

ボタンは基本波とLFOの波形の切り替えに使っています。

以前、AKAIの「MPC」というPADが4×4で並んだ楽器がありました。



見た感じ簡単に「いい感じ」のグループを作れそう～！

ということで買ってはみたものの、まったく使えませんでした。

いや、使えないのは自分が悪くて、簡単には使いこなせなかったということです。普通の「楽器」の様に肉体的な鍛錬が必要なのでした。しかも、リズム系ではがんばれば使えそうだけど、飛び道具的な使い方はあんまり期待できそうにありませんでした。

なので単純に音をアナログ的に操れるデバイスが欲しくなって、可変抵抗を 3 個とタクトスイッチを 2 個並べたぴゅんぴゅんコントローラー（1 号）を作成しました。

いきなり使ってみて簡単に音を操れる、という意味では「ぴゅんぴゅんコントローラー」は、これはこれでなかなか使いやすい入力デバイスだと思っています。

9. 波形の増やし方

ぴゅんぴゅん 3 号の波形データは<wavetable.h>で定義してあります。値は 12bit(0 - 4095)で 1 サイクルで 1024 個です。

ここに値（配列）を追加してやると扱える波形を増やせます。

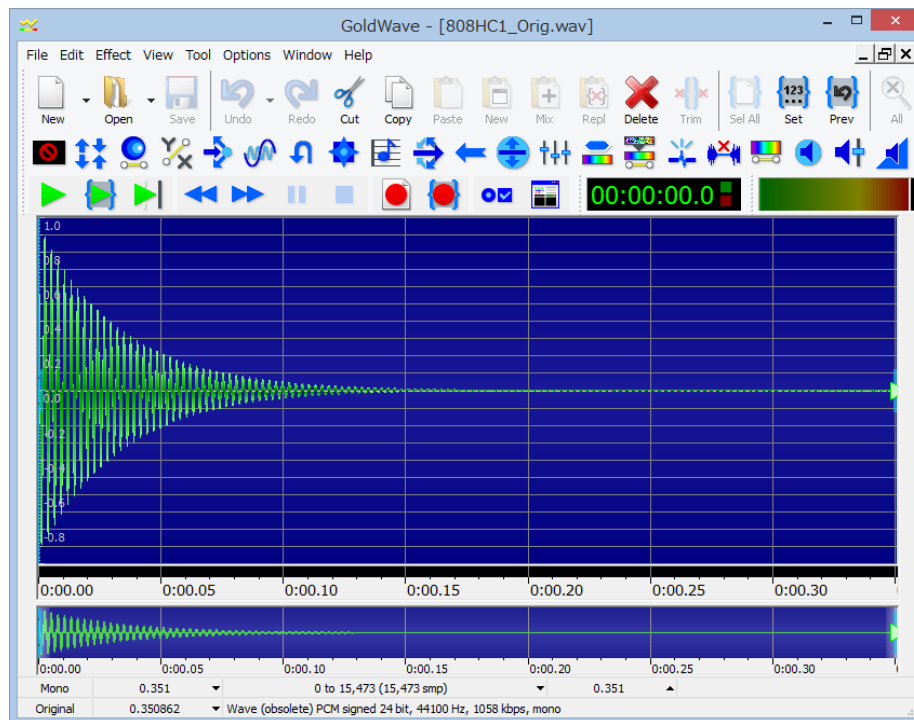
基本的な波形、「正弦波」、「三角波」、「矩形波」、「ノコギリ波」を定義してありますが、12bit で 1024 個のデータなら<wavetable.h>で定義してやればプログラムで扱えるようになります。

自分で録音したり Sampling CD から抜き出した音を使いたい場合、どうすればいいのか考えてみました。

波形編集ソフト

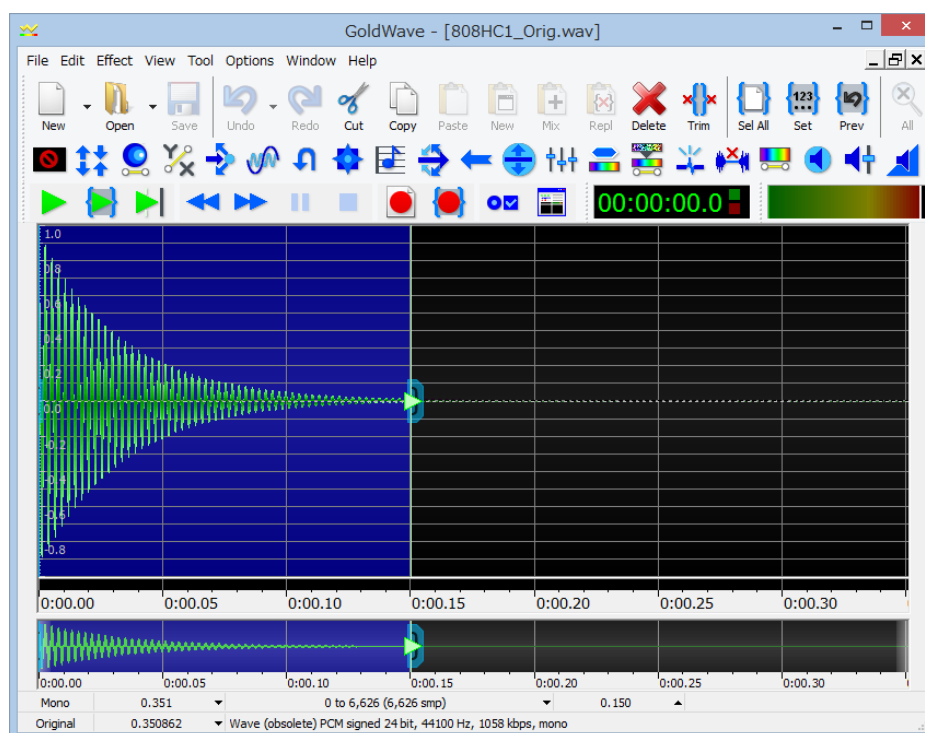
最近は波形編集に「GoldWave」というソフトを使っています。波形編集というと「Sound Forge」とか「Wavelab」が有名ですが UI のダサさに我慢できれば「GoldWave」もなかなか使えます。

TR-808 の conga の音を加工することにします。（サンプリング CD からもらいました）



ステータスバーを見ると「0 to 15,473 smp」とあるので 1024 個におさえるためには 10 分の 1 以上圧縮する必要があります。

まずは、画面の右側の波形が収束している余韻みたいなところをカットしてみます



6,626smp に切り詰められました。

ステータスバーを見ると

Wave(obsolete)PCM signed 24bit, 44100Hz, 1058kbs, mono

となっています。

このまま使うと、サンプリングレートが 44,100Hz なので、ぴゅんぴゅん 3 号の Wave 系で「1Hz」で再生するとちょうどぴったりで再生されることになります。

※ぴゅんぴゅん 3 号の基本波のサンプリングレートは 48,000Hz にしているのでぴったりではなかった (@@ ;

<main.c>の先頭あたりで#define している「SAMPLE_CLOCK」を下記の通り 44,100kHz に定義すれば 1Hz でちょうどになると思います。

```
#define SAMPLE_CLOCK (44100.0f)
```

プログラムは 1024smp までしか認識しないので、オーバーしたサンプルは途中でブチ切れます。

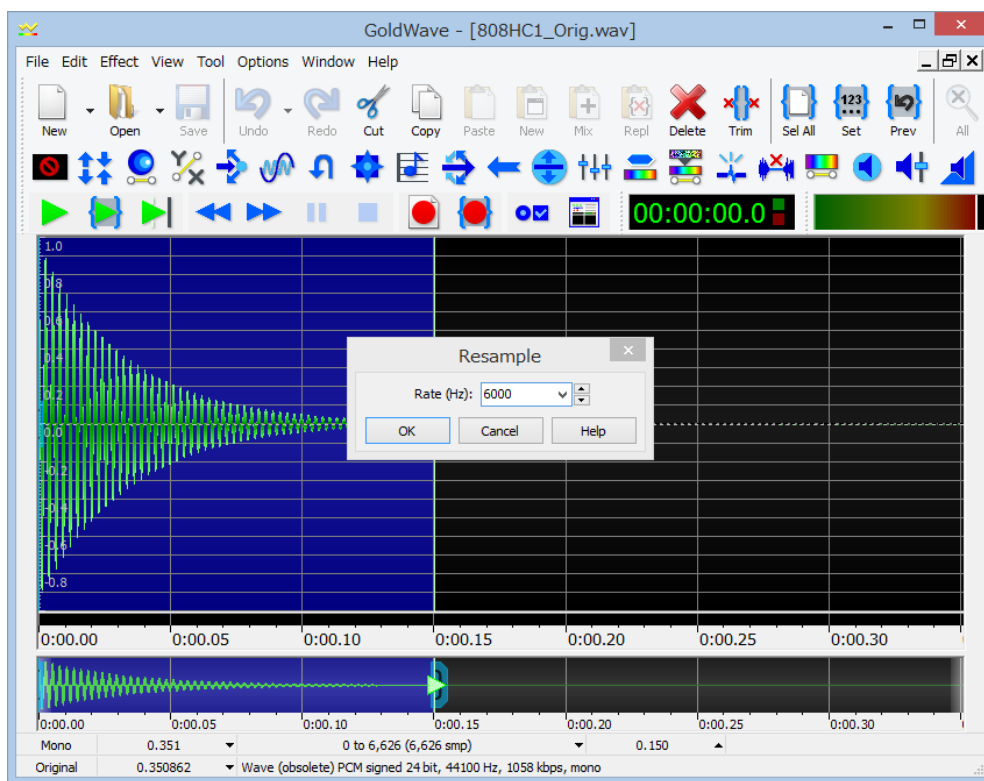
サンプリング数を 1024 にするためにサンプリングレートを下げてみます。

$$6626 / 1024 = \text{だいたい } 6.47$$

なので、サンプリングレートをだいたい 1/7 程度にすることを考えてみます。

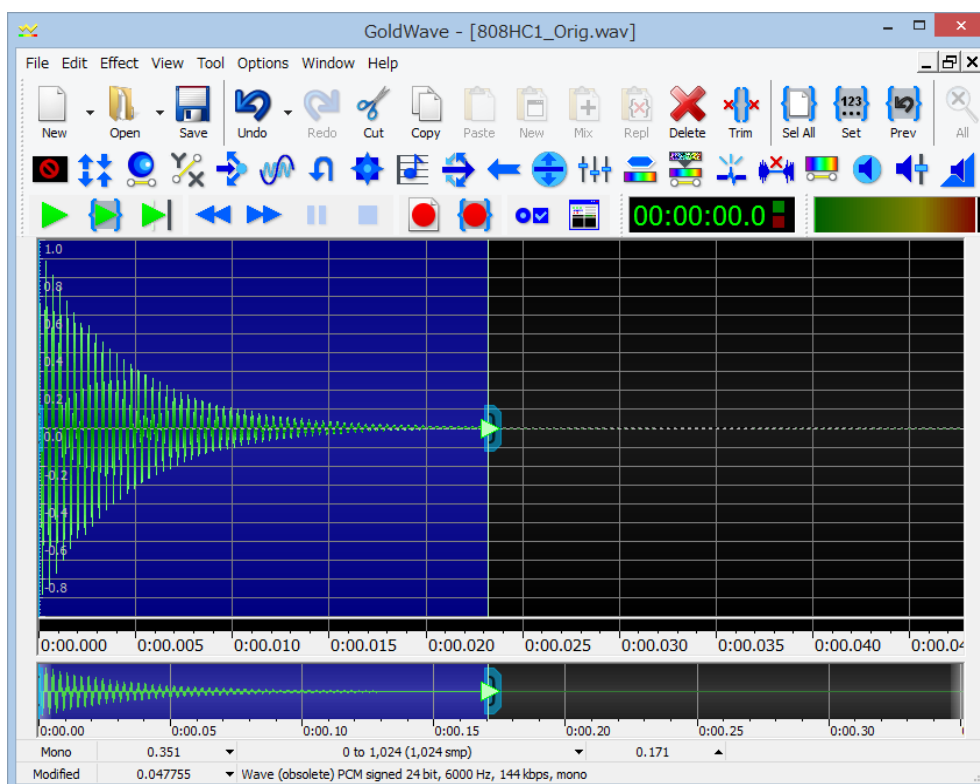
$$44,100 / 7 = 6,300$$

キリがいいところで 6,000 (Hz) にしてみます。

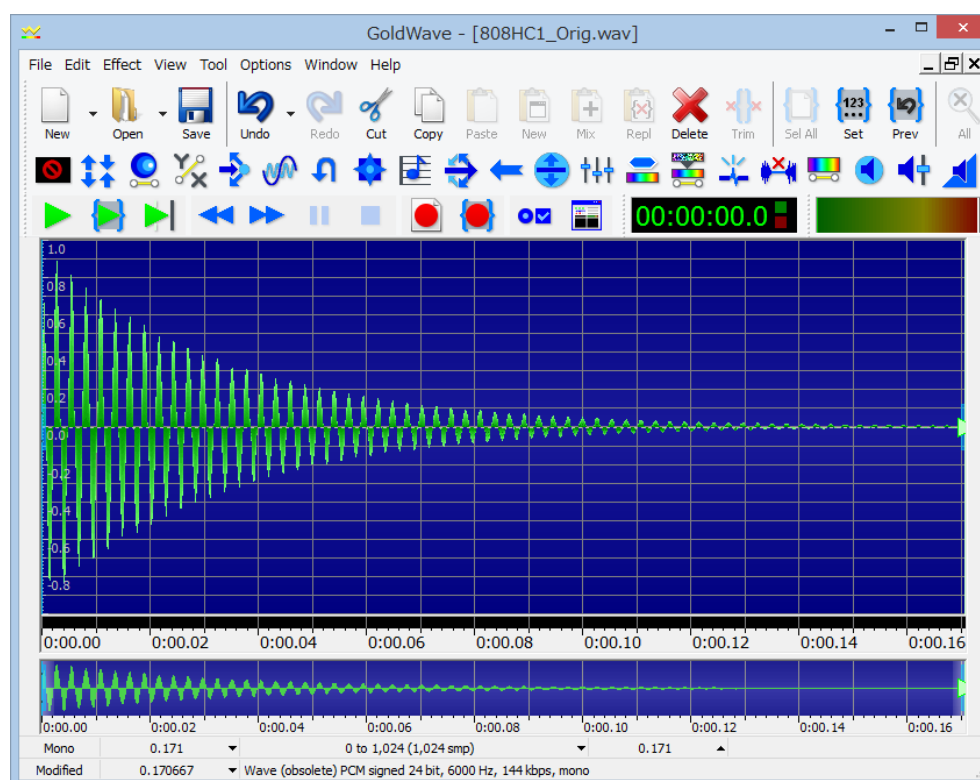


これでサンプリング・レートが変更されサンプルが間引きされるのでサンプル数を減らせます。

さらにサンプル数がちょうど 1024 になるように範囲を設定します。



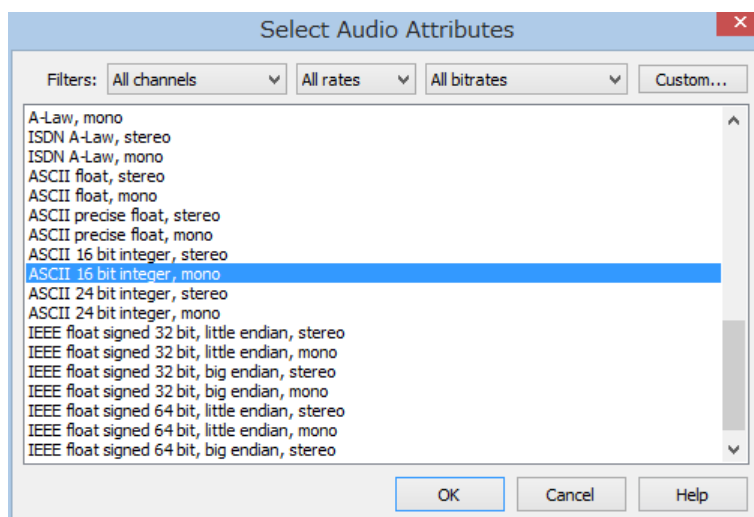
メニューバーから「Edit」 - 「Trim」を実行するとサンプル数が 1024 個の波形になります。



これを 12bit で出力したいのですが、ここから先はテキスト形式の方が編集しやすいので 16bit のテキスト形式で出力させてみます。

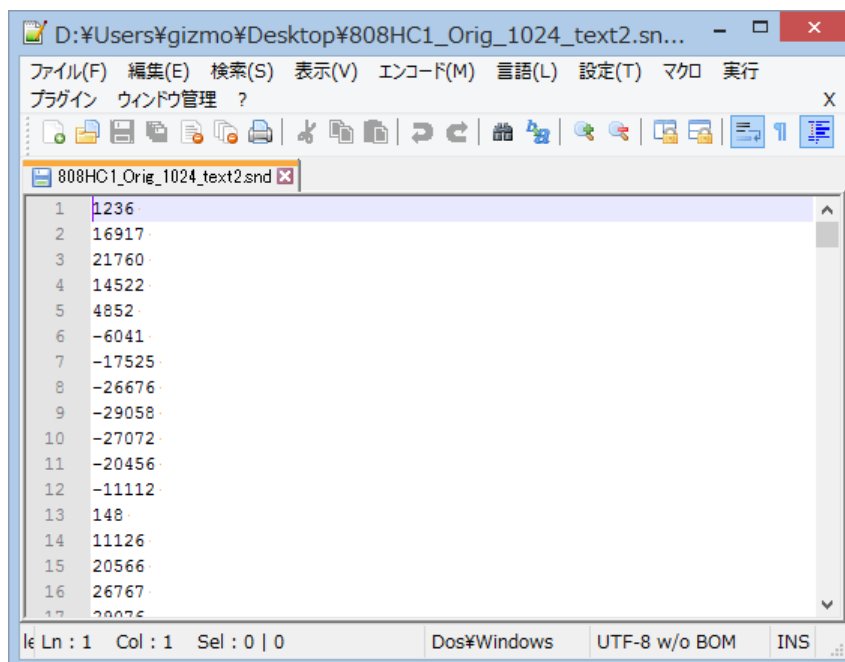
「File」 - 「Save As」で表示されるダイアログで「ファイルの種類」で「Raw(*.snd)」を選択します。

ダイアログの「Set Attribute...」ボタンを押すと「Select Audio Attributes」ダイアログが表示されます。



「ASCII 16bit integer, mono」を選択して「OK」を押します。

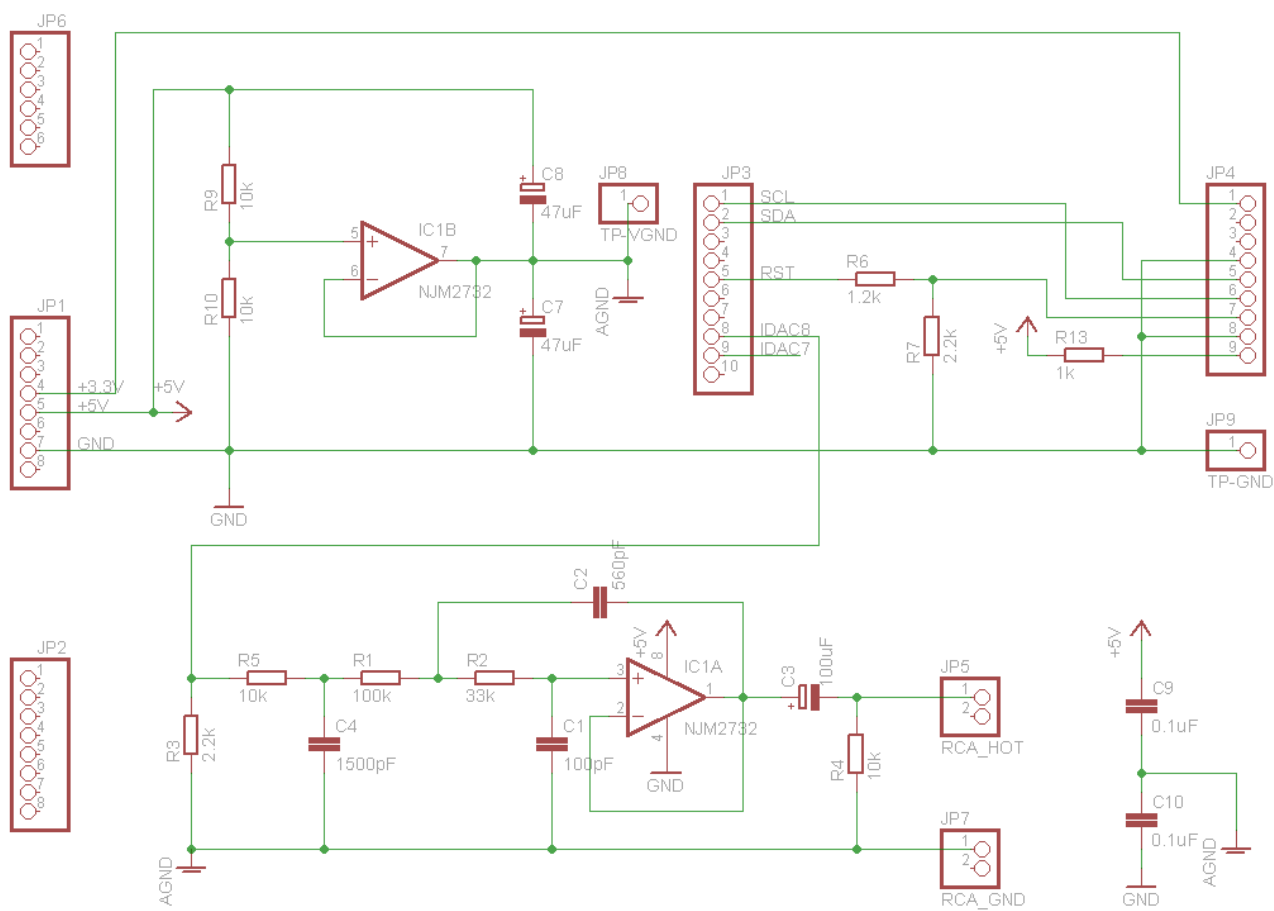
「Save Sound As」ダイアログに戻るので、「保存」ボタンを押せば 16bit、1024 個の波形がテキスト形式で保存されます。

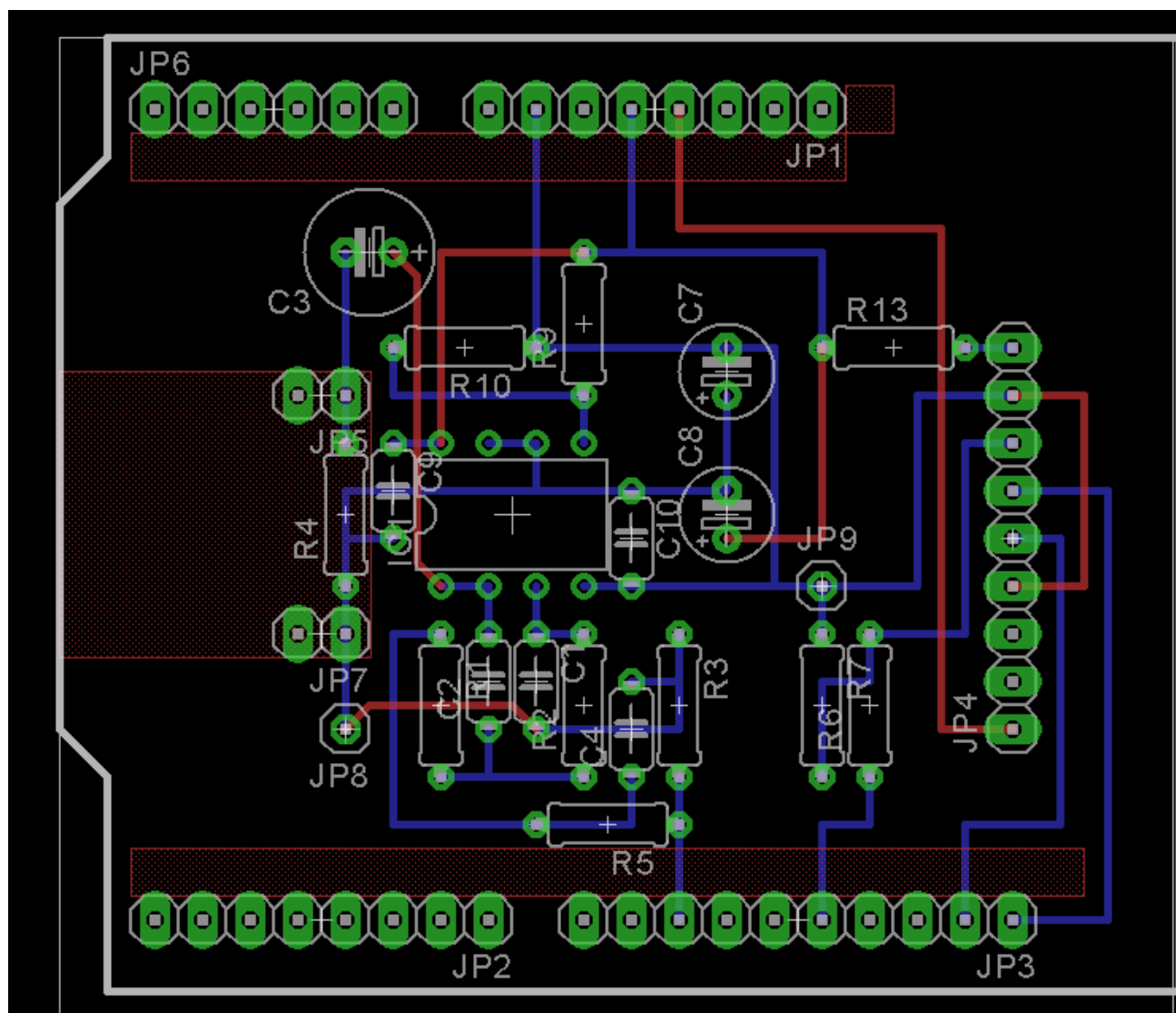


テキスト形式なので Excel とかエディタのマクロなどで編集して 12bit (0..4095 の範囲) の配列の表記に変換してやれば<wavetable.h>に埋め込めると思います。

10. 資料

回路図





BOM

品名	パーツ番号	仕様	購入店
フィルム・コンデンサ	C1	100pF	
	C2	560pF	
	C4	1500pF	
電解コンデンサ	C3	100uF	
	C7	47uF	
	C8	47uF	
積セラ	C9	0.1uF	
	C10	0.1uF	
OPAMP	IC1	NJM2732	
カーボン抵抗	R1	100k	
	R2	33k	
	R3	2.2k	
	R4	10k	
	R5	10k	
	R6	1.2k	
	R7	2.2k	
	R9	10k	
	R10	10k	
	R13	1k	
Arduino シールド用 ピンソケット	JP1	8P	
	JP2	8P	
	JP3	10P	
	JP6	6P	
ピンソケット	JP4	9P	
RCA ジャック	JP5,JP7	RCA ジャック DIP 化キット	秋月
チェック端子	JP8,JP9		
LCD		SPLC792-I2C-M	aitendo
シールド基板		Arduino 用ユニバーサル プロトシールド基板	秋月
IC ソケット		DIP 8P 300mil	

※R3、R1、R2、C4、C1、C2 はピンソケット（表外）を使って実装しました。

<main.c>

```

/* =====
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * =====
 */

#include <project.h>
#include <stdio.h>
#include <math.h>
#include "wavetable.h"

#define SAMPLE_CLOCK (48000.0f)

/* I2C slave address to communicate with */
#define I2C_LCD_ADDR (0b0111110)

/* Buffer and packet size */
#define I2C_LCD_BUFFER_SIZE (2u)
#define I2C_LCD_PACKET_SIZE (I2C_LCD_BUFFER_SIZE)

/* Command valid status */
#define I2C_LCD_TRANSFER_CMPLT (0x00u)
#define I2C_LCD_TRANSFER_ERROR (0xFFu)

/* ADC channels */
#define ADC_CH_WAV_FREQ_N (0x00u)
#define ADC_CH_LFO_FREQ_N (0x01u)
#define ADC_CH_LFO_DEPT_N (0x02u)

/* ADC limits */
#define ADC_LOW_LIMIT ((int16)0x000)
#define ADC_HIGH_LIMIT ((int16)0x7FF)

/* Wave Tables */
#define WAVE_SHAPE_N (5)
#define WAVE_TABLE_LEN (1024)

/* Wave & LFO Frequency Limit */
#define WAVE_FREQ_MAX ((double)1000.0f)
#define LFO_FREQ_MAX ((double)10.f)

/*****
 * マクロ
 *****/

/* Set LED RED color */
#define RGB_LED_ON_RED ¥
do{ ¥
    LED_RED_Write (0u); ¥
    LED_GREEN_Write(1u); ¥
}while(0)

/* Set LED GREEN color */
#define RGB_LED_ON_GREEN ¥
do{ ¥
    LED_RED_Write (1u); ¥

```

```

        LED_GREEN_Write(0u); ¥
    }while(0)

/* ADC limits trim */
#define ADC_LIMIT(x) ¥
    ((x)<ADC_LOW_LIMIT?ADC_LOW_LIMIT:((x)>=ADC_HIGH_LIMIT?ADC_HIGH_LIMIT:(x)))

/*****
* 大域変数
*****/

/* 波形パラメータ */
volatile double waveFrequency;
volatile double lfoFrequency;
volatile uint8 lfoDepth;
volatile uint8 waveShape;
volatile uint8 lfoShape;

const uint16 *waveTables[WAVE_SHAPE_N];

/* DDS 用変数 */
volatile uint32 phaseRegister;
volatile uint32 tuningWord;

volatile uint32 lfoPhaseRegister;
volatile uint32 lfoTuningWord;

/* 入力デバイス用変数 */
int16 adcResult[ADC_SAR_SEQ_TOTAL_CHANNELS_NUM];
uint8 swWavForm;
uint8 swLfoForm;
uint8 prevSwWavForm = 0u;
uint8 prevSwLfoForm = 0u;

/* 表示用 */
const char *waveShapeStr[] = {
    "SIN", "TRI", "SQR", "SW1", "SW2"
};

/*****
* LCD 制御
*
*****/

/* LCD のコントラストの設定 */
uint8 contrast = 0b100000;    // 3.0V 時 数値を上げると濃くなります。
                                // 2.7V では 0b111000 くらいにしてください。
                                // コントラストは電源電圧、温度によりかなり変化します。実際の液晶をみて調整してくだ
                                さい。

uint32 LCD_Write(uint8 *buffer)
{
    uint32 status = I2C_LCD_TRANSFER_ERROR;

    I2CM_I2CMasterWriteBuf(I2C_LCD_ADDR, buffer, I2C_LCD_PACKET_SIZE, I2CM_I2C_MODE_COMPLETE_XFER);

    while (0u == (I2CM_I2CMasterStatus() & I2CM_I2C_MSTAT_WR_CMPLT))
    {
        /* Waits until master completes write transfer */
    }

    /* Displays transfer status */
    if (0u == (I2CM_I2C_MSTAT_ERR_XFER & I2CM_I2CMasterStatus()))
    {
        RGB_LED_ON_GREEN;

        /* Check if all bytes was written */
    }
}

```



```

        if(I2CM_I2CMasterGetWriteBufSize() == I2C_LCD_BUFFER_SIZE)
        {
            status = I2C_LCD_TRANSFER_CMPLT;

            // 1命令ごとに余裕を見て 50us ウェイトします。
            CyDelayUs(50);

        }
    }
    else
    {
        RGB_LED_ON_RED;
    }

    (void) I2CM_I2CMasterClearStatus();

    return (status);
}

// コマンドを送信します。HD44780 でいう RS=0 に相当
void LCD_Cmd(uint8 cmd)
{
    uint8 buffer[I2C_LCD_BUFFER_SIZE];
    buffer[0] = 0b00000000;
    buffer[1] = cmd;
    (void) LCD_Write(buffer);
}

// データを送信します。HD44780 でいう RS=1 に相当
void LCD_Data(uint8 data)
{
    uint8 buffer[I2C_LCD_BUFFER_SIZE];
    buffer[0] = 0b01000000;
    buffer[1] = data;
    (void) LCD_Write(buffer);
}

void LCD_Init()
{
    CyDelay(40);
    LCD_Cmd(0b00111000); // function set
    LCD_Cmd(0b00111001); // function set
    LCD_Cmd(0b00010100); // interval osc
    LCD_Cmd(0b01110000 | (contrast & 0xF)); // contrast Low
    LCD_Cmd(0b01011100 | ((contrast >> 4) & 0x3)); // contrast High/icon/power
    LCD_Cmd(0b01101100); // follower control
    CyDelay(300);

    LCD_Cmd(0b00111000); // function set
    LCD_Cmd(0b00001100); // Display On
}

void LCD_Clear()
{
    LCD_Cmd(0b00000001); // Clear Display
    CyDelay(2); // Clear Display は追加ウェイトが必要
}

void LCD_SetPos(uint32 x, uint32 y)
{
    LCD_Cmd(0b10000000 | (x + y * 0x40));
}

// (主に) 文字列を連続送信します。
void LCD_Puts(char8 *s)
{
    while(*s) {
        LCD_Data((uint8)*s++);
    }
}

```

```

}

/*=====
* 入力処理
*
*=====*/
// ADC
void pollingADC()
{
    ADC_SAR_Seq_StartConvert();
    while (ADC_SAR_Seq_IsEndConversion(ADC_SAR_Seq_RETURN_STATUS) == 0u) {
        // 変換終了を待つ
        ;
    }
    adcResult[ADC_CH_WAV_FREQ_N] = ADC_LIMIT(ADC_SAR_Seq_GetResult16(ADC_CH_WAV_FREQ_N));
    adcResult[ADC_CH_LFO_FREQ_N] = ADC_LIMIT(ADC_SAR_Seq_GetResult16(ADC_CH_LFO_FREQ_N));
    adcResult[ADC_CH_LFO_DEPT_N] = ADC_LIMIT(ADC_SAR_Seq_GetResult16(ADC_CH_LFO_DEPT_N));

    waveFrequency = WAVE_FREQ_MAX * adcResult[ADC_CH_WAV_FREQ_N] / 2048;
    lfoFrequency = LFO_FREQ_MAX * adcResult[ADC_CH_LFO_FREQ_N] / 2048;
    lfoDepth = (uint8)(adcResult[ADC_CH_LFO_DEPT_N] / 8);
}

// Switches
void pollingSW()
{
    swWavForm = WAV_FORM_PIN_Read();
    if (swWavForm && !prevSwWavForm) {
        waveShape++;
        if (waveShape >= WAVE_SHAPE_N)
            waveShape = 0;
    }

    swLfoForm = LFO_FORM_PIN_Read();
    if (swLfoForm && !prevSwLfoForm) {
        lfoShape++;
        if (lfoShape >= WAVE_SHAPE_N)
            lfoShape = 0;
    }

    prevSwWavForm = swWavForm;
    prevSwLfoForm = swLfoForm;
}

/*=====
* 波形生成
*
*=====*/
CY_ISR(TimerISR_Handler)
{
    uint16 index;
    int32 waveValue, lfoValue;

    // Caluculate LFO Value
    //
    lfoPhaseRegister += lfoTuningWord;

    // 32bit の phaseRegister をテーブルの 10bit(1024 個)に丸める
    index = lfoPhaseRegister >> 22;

    // lookupTable(11bit + 1bit) * (lfoDepth(8bit) -> 20bit) : 31bit + 1bit
    lfoValue = ((int32)(*(waveTables[lfoShape] + index)) - 2048) * ((int32)lfoDepth << 12);

    // tuningWord(32bit) * lfoValue(31bit + 1bit) : (63bit + 1bit) -> 31bit + 1bit
    lfoValue = (int32)((int64)tuningWord * lfoValue >> 31);

    // Caluculate Wave Value
    //

```

```

    phaseRegister += tuningWord + lfoValue;

    // 32bit の phaseRegister をテーブルの 10bit(1024 個)に丸める
    index = phaseRegister >> 22;
    waveValue = *(waveTables[waveShape] + index);

    //DACSetVoltage(waveValue);
    IDAC8_SetValue(waveValue >> 4);
    IDAC7_SetValue(lfoValue >> 5);

    SamplingTimer_ClearInterrupt(SamplingTimer_INTR_MASK_TC);
}

/*=====
 * メインルーチン
 *
 *=====*/
int main()
{
    char lcdLine[16 + 1];

    // 変数を初期化
    waveFrequency = 1000.0f;
    tuningWord = waveFrequency * pow(2.0, 32) / SAMPLE_CLOCK;
    phaseRegister = 0;

    lfoFrequency = 1.0f;
    lfoTuningWord = lfoFrequency * pow(2.0, 32) / SAMPLE_CLOCK;
    lfoPhaseRegister = 0;

    lfoDepth = 255;
    waveShape = 0;
    lfoShape = 0;

    waveTables[0] = waveTableSine;
    waveTables[1] = waveTableTriangle;
    waveTables[2] = waveTableSquare;
    waveTables[3] = waveTableSawtoothDown;
    waveTables[4] = waveTableSawtoothUp;

    // コンポーネントを初期化
    SamplingTimer_Start();
    TimerISR_StartEx(TimerISR_Handler);

    /* Init and start sequencing SAR ADC */
    ADC_SAR_Seq_Start();
    ADC_SAR_Seq_StartConvert();

    /* Init I2C LCD */
    I2CM_Start();

    CyGlobalIntEnable;

    // LCD を RESET
    CyDelay(500);
    LCD_RST_Write(0u);
    CyDelay(1);
    LCD_RST_Write(1u);
    CyDelay(10);

    LCD_Init();
    LCD_Clear();

    LCD_Puts("PyunPyun");

    LCD_SetPos(1, 1);
    LCD_Puts("Machine #3");

```

```

CyDelay(1000);

/* Start Wave Output */
IDAC8_Start();
IDAC7_Start();

for(;;)
{
    pollingADC();
    pollingSW();

    tuningWord = waveFrequency * pow(2.0, 32) / SAMPLE_CLOCK;
    lfoTuningWord = lfoFrequency * pow(2.0, 32) / SAMPLE_CLOCK;

    sprintf(lcdLine, "FREQ LFO DPT %s", waveShapeStr[waveShape]);
    LCD_SetPos(0, 0);
    LCD_Puts(lcdLine);

    sprintf(
        lcdLine, "%4d%4d%4d %s",
        (int)waveFrequency,
        (int)(lfoFrequency * 10),
        lfoDepth,
        waveShapeStr[lfoShape]
    );
    LCD_SetPos(0, 1);
    LCD_Puts(lcdLine);

    //CyDelay(100);
}
}

/* [] END OF FILE */

```

波形設定の例

Name	WAVE	LFO	DPT	Wave Shape	LFO Shape
QQ Car	500	8	128	SQR	SQR
Ame Pato	645	24	165	SQR	SW1
Pacman	505	23	156	TRI	TRI
UFO	612	81	165	SW1	SW1
Psy Trance	157	25	255	SIN	SW1
Robotork	手動	77	89	SQR	SQR

このパラメーターで実際に音出したものを「¥Wave¥」フォルダに収録しています。

おわりに

ぴゅんぴゅん3号はまだまだ細かなバグがあるので、取り込む波形やアナログ回路の作りこみ次第でいろいろと変な音が出てきます。

これをヒントに、シンセの自作を楽しんでください (^ q ^ /

ぴゅんぴゅん 3号

2015 年 8 月 16 日 初版発行

著者 ryood

発行所 PNPMS (ぴゅんぴゅんマシン製作所)

<http://dad8893.blogspot.jp/>

Google+: ryood