# 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。

# ぴゅんぴゅんマシンとは

最近はもう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で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「ぴゅんぴゅんマシーン」の音が入っていました。

「ぴゅんぴゅんマシーン」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、とてもなさけない音がする「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「ぴゅんぴゅんマシーン」は大ヒットでした。

ぴゅんぴゅんマシーンが欲しくなってネットで調べると売っているけど、め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！！

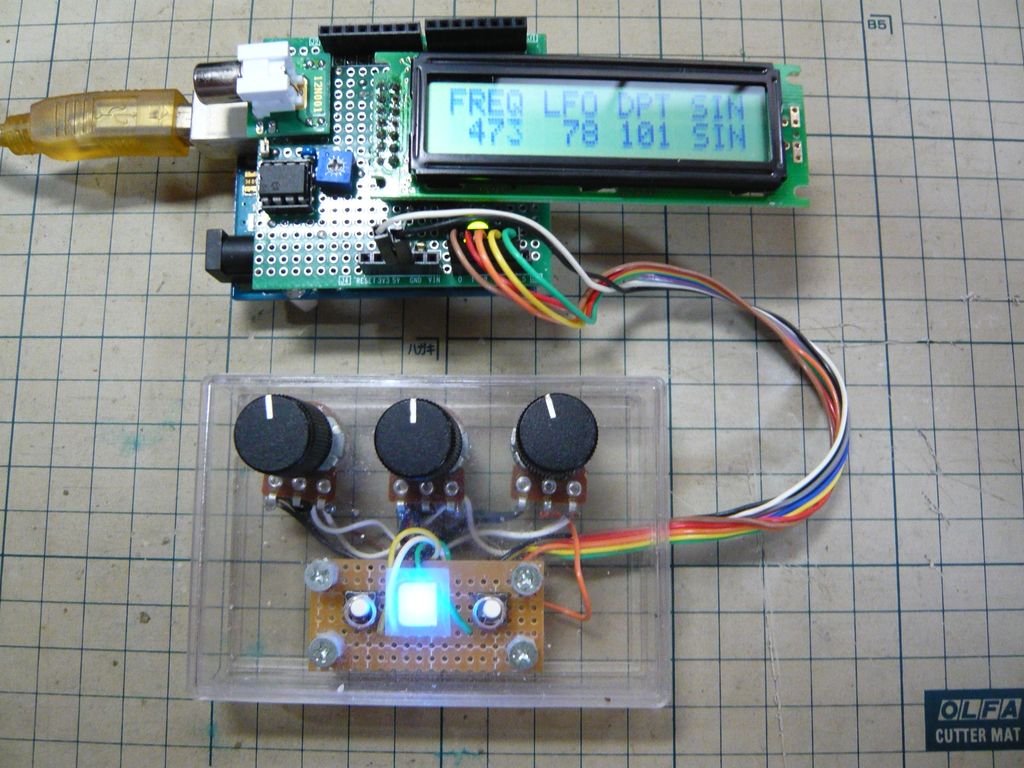
似たような音はがんばれば出せるかもしれないので、一から自分で作ってみようというのがそもそもの動機でした。

## フル・ディスクリー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。

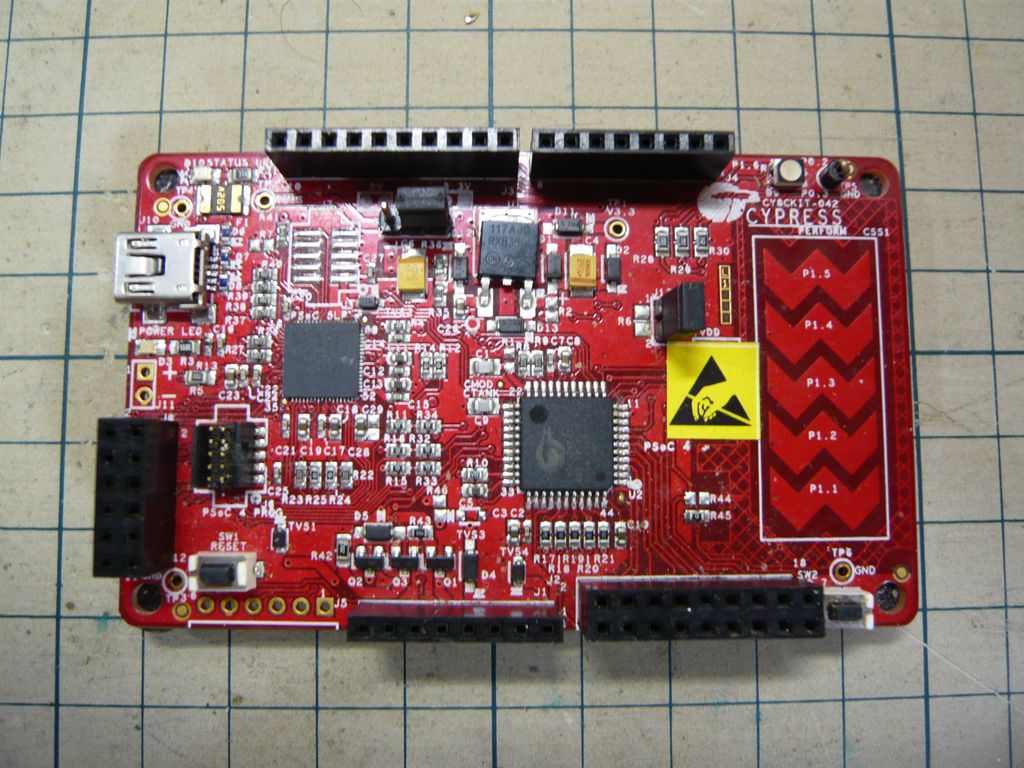
https://github.com/ryood/C86

# 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のシールドが使えて
* PSoCのカスタマイズ可能な周辺機能が使えて
* 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。

# 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。

**楽をしたいなら「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。

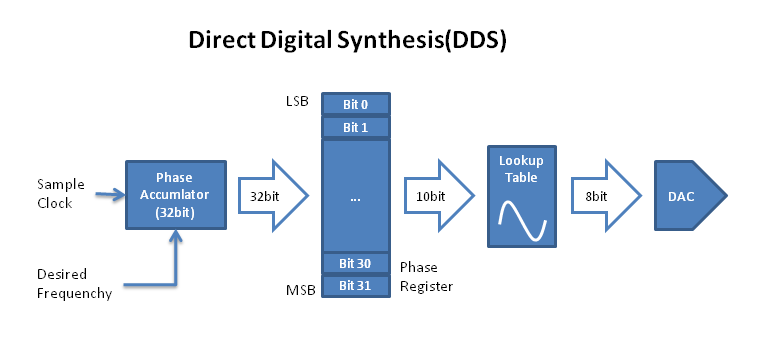
# 音を作る

## 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。



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

|  |
| --- |
| // Caluclate 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); |

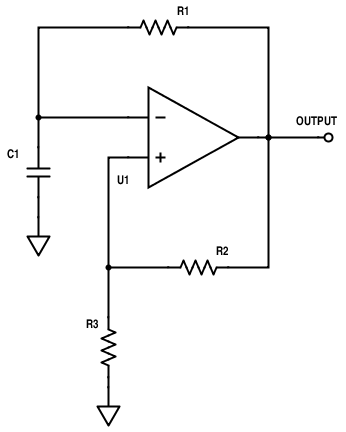
これだけです。

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 Ocillator」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、

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では？

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 = (tuningWord / 2 ^ n) \* SAMPLING\_RATE;

（f\_out: 出力周波数, n: tuningWordのビット長）

で求まります。

(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なことになります。

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

tuningWord = 1

となり、16bit演算で、CDと一緒の44.1kHzの場合で考えると、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[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に下げてやれば

f\_out = (1 / 2　^　16) \* 4410 = 0.0673[Hz]

まで下の周波数が生成できることになります。

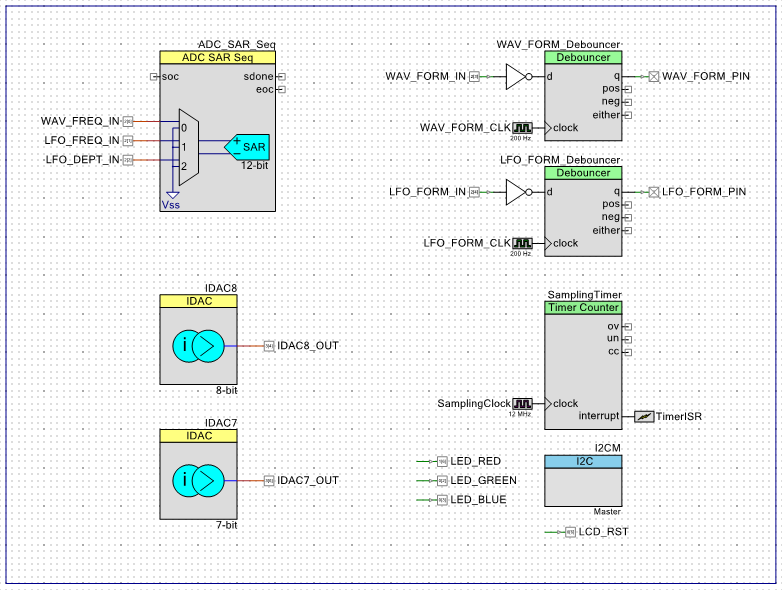
たかだか1/10ですが、「桁足らず？」で演算できなくなるのをフォローするためには2系統作るのがよさそうです。

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。

# 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(MHz) \* (250 / 65535) = だいたい45.777(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

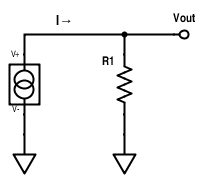
このコンポーネントは波形を生成する大事なクロックを司っているのですが、バグも音の味ということで、そのうち直したいと思います（＾ｑ＾；

### 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

Vout = 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)」を選べるようになっていて、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、接続がパラレル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て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。

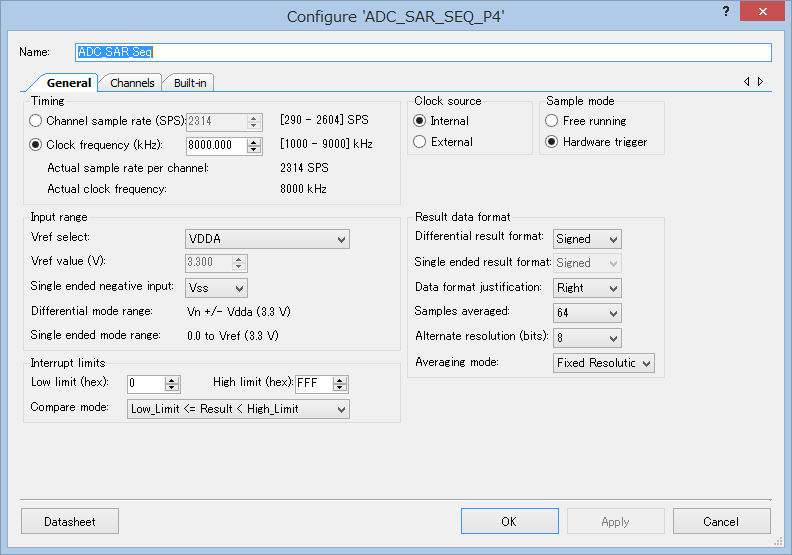
### ADC

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「Seaquecing 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。

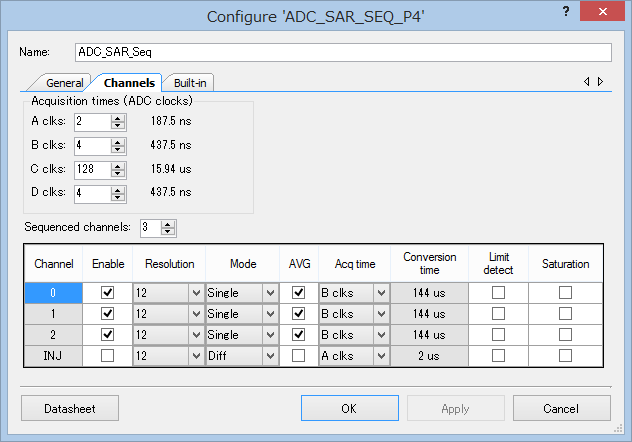
「Clock Source」は「\*.cydwr」の「Clock」タブでADCに設定した値を使うかTopDesignで外部から明示的に与えるかの選択です。

「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 fresult formt:」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 fresult formt:」の方はなぜだ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。



「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。

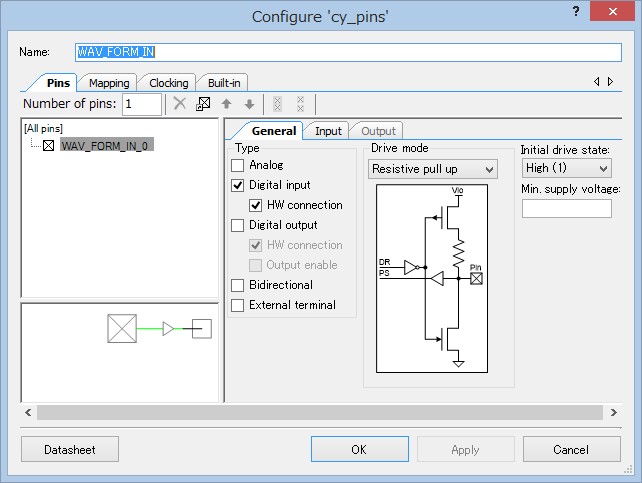
CPU（プログラム）と関係なく動作するのでプログラムで考慮する必要もなく、外付けの回路も一切不要でした。

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

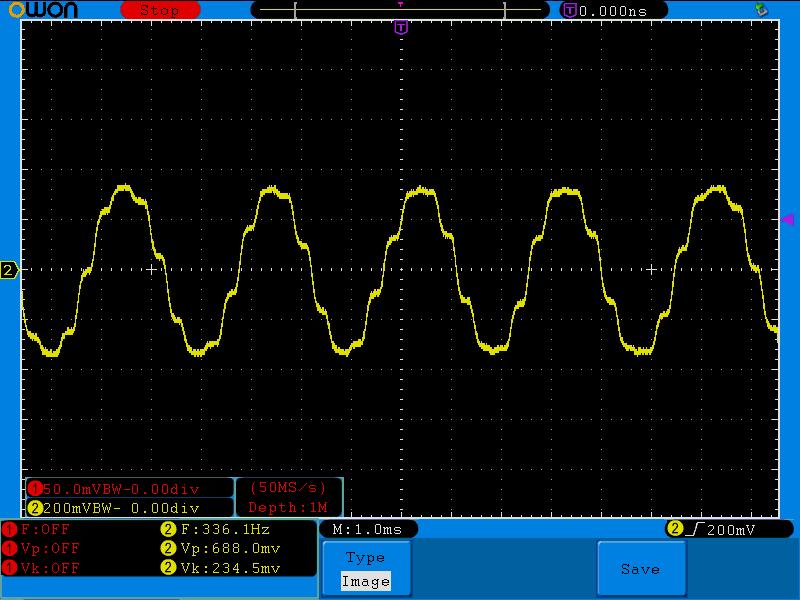
フィジカル・コンピューティングでは割込みで処理することがほとんどだと思います。

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ー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割込み）だけ割込み処理にして、他の処理はポーリングで処理することにしました。

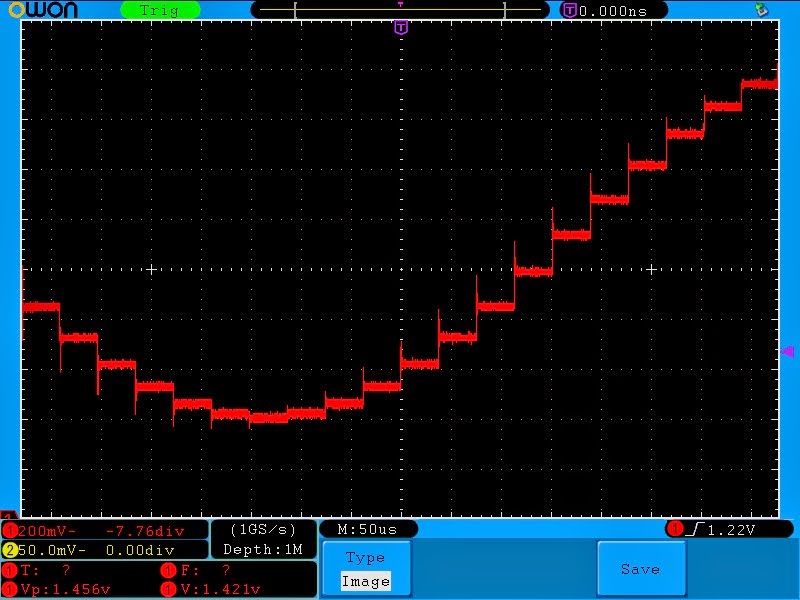
ポーリングとはプログラムで明示的に外部デバイスに状態を問い合わせる処理です。

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。

# アナログ回路を作る

## LPF (Low pass filter)

PSoCのIDACは最大8bitです。



オシロで拡大して見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。

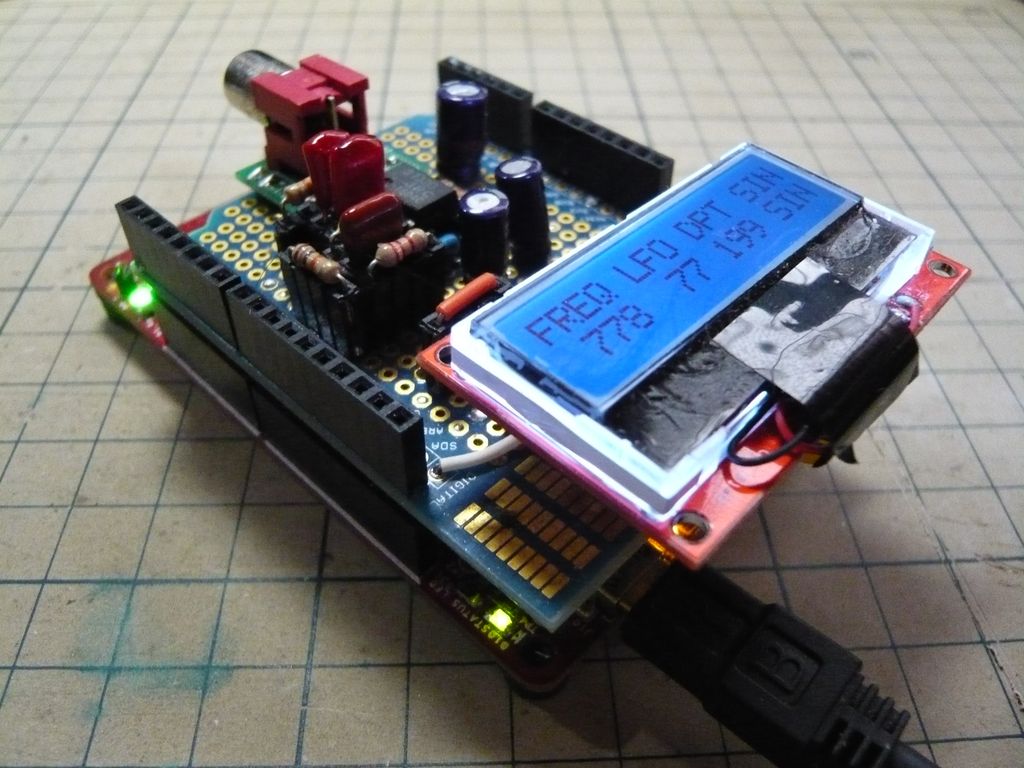
数学的に言うとフーリエ級数で表現できるそうです。

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の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。

ぴゅんぴゅん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。

# 出力波形

LFOをかけない状態でぴゅんぴゅん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） |
| C:\Users\gizmo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20150429_004552.png | C:\Users\gizmo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20150429_004855.png |
|  |  |
| SW1（ノコギリ波・下降） | SW2（ノコギリ波・上昇） |
| C:\Users\gizmo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20150429_005029.png | C:\Users\gizmo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20150429_005056.png |
|  |  |
| SQR（矩形波） |  |
| C:\Users\gizmo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20150429_004948.png |  |

# 音を出して遊ぶ

これで音を出す準備はできました。

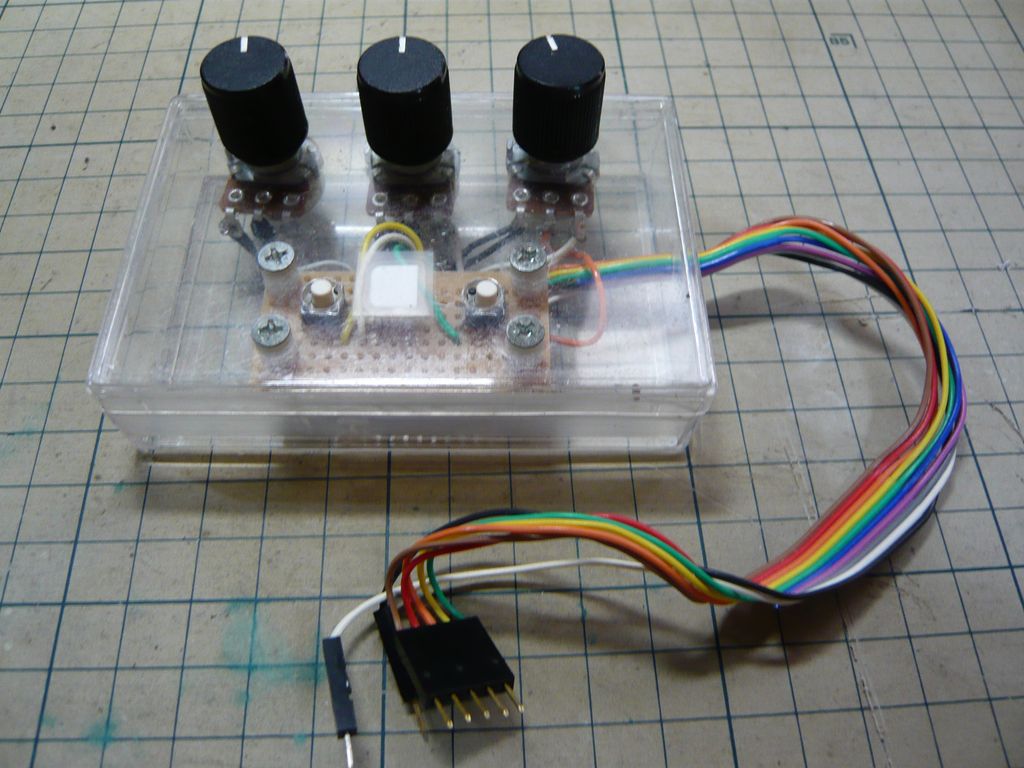
これだけでも「音」は出ます。でも、音が出ただけでは何が面白いのかわかりません。

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」のグルーブを作れそう～！

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。

# 波形の増やし方

ぴゅんぴゅん3号の波形データーは<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。

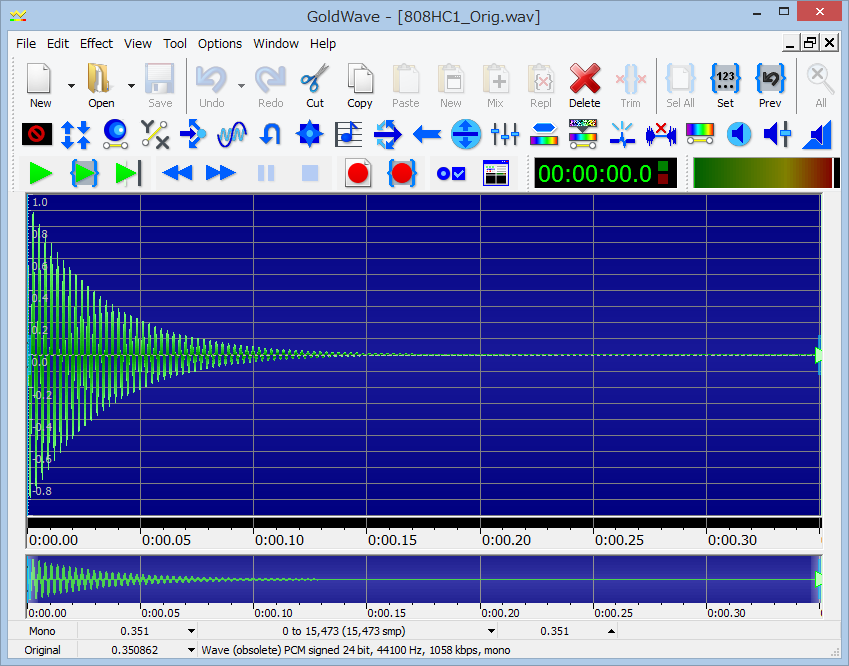
基本的な波形、「正弦波」、「三角波」、「矩形波」、「ノコギリ波」を定義してありますが、12bitで1024個のデータなら<wavetable.h>で定義してやればプログラムで扱えるようになります。

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

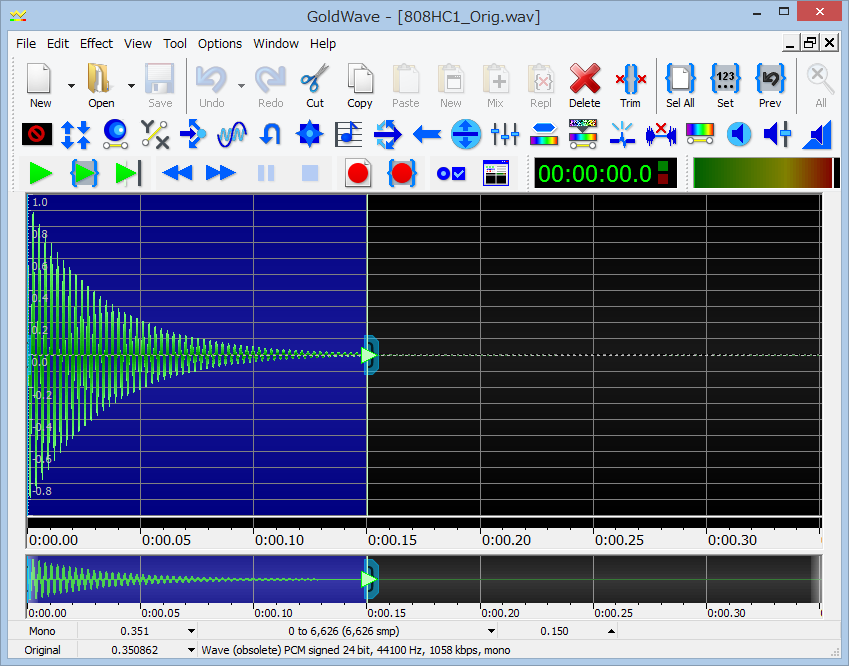
最近は波形編集に「GoldWave」というソフトを使っています。波形編集というと「Sound Forge」とか「Wavelab」が有名ですがUIのダサさに我慢できれば「GoldWave」もなかなか使えます。

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(oblolete)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。

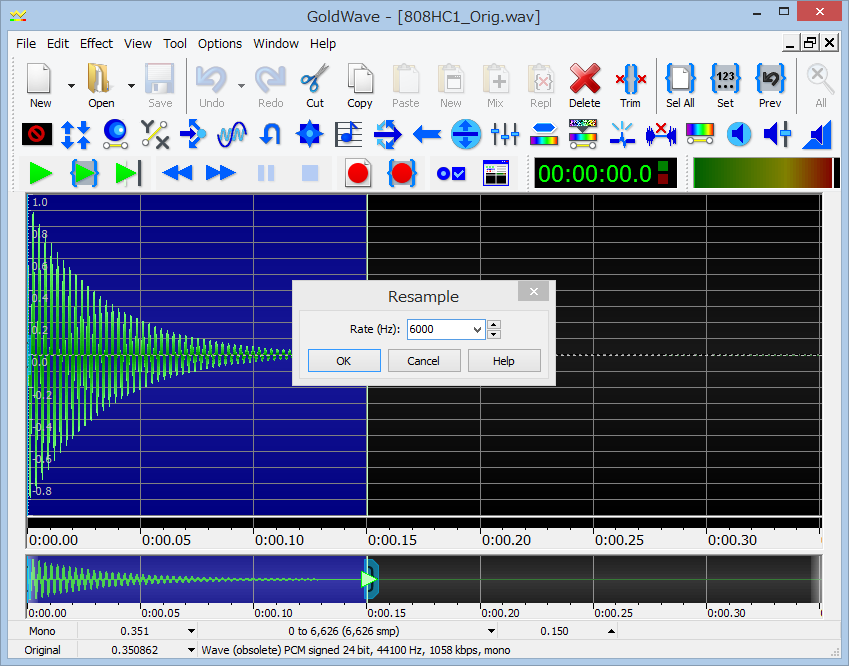
サンプリング数を1024にするためにサンプリングレートを下げてみます。

6626 / 1024 = 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。

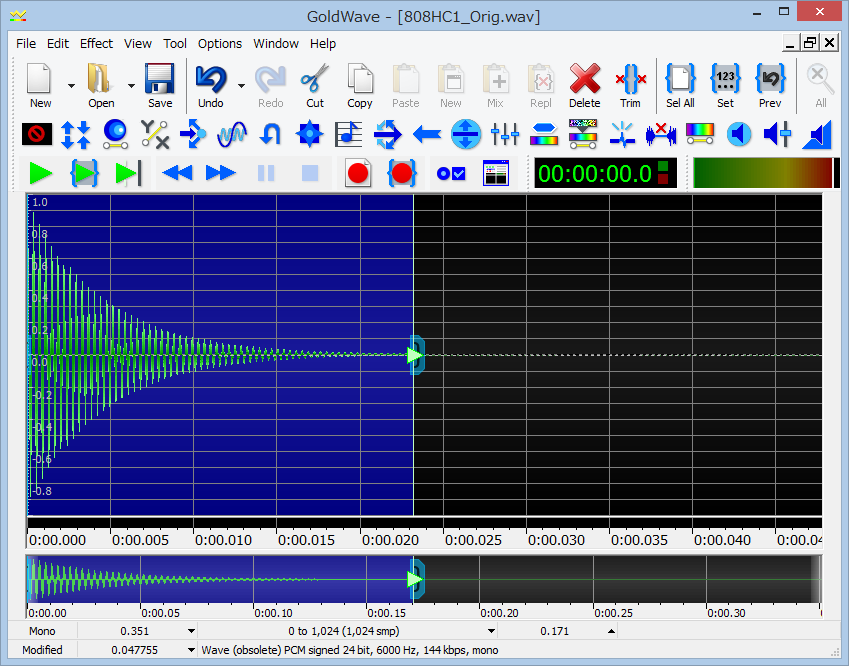
44,100 / 7 = 6,300

キリがいいところで6,000（Hz）にしてみます。

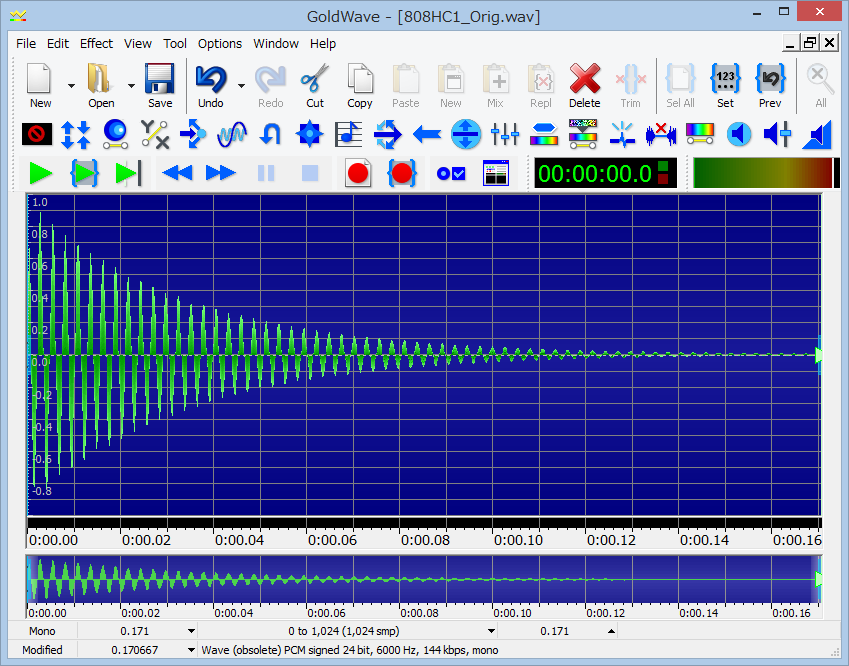


これでサンプリング・レートが変更されサンプルが間引きされるのでサンプル数を減らせます。

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。



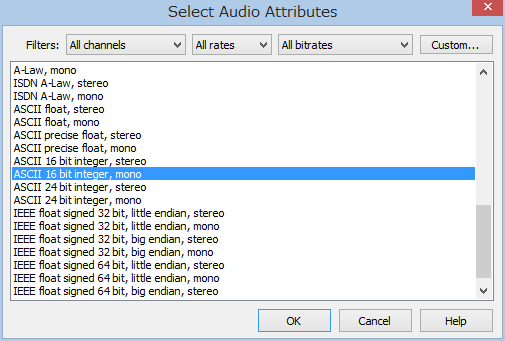
メニューバーから「Edit」-「Trim」を実行するとサンプル数が1024個の波形になります。



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。

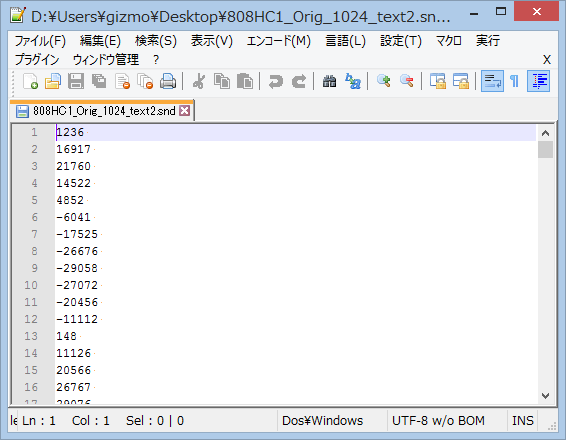
「File」-「Save As」で表示されるダイアログで「ファイルの種類」で「Raw(\*.snd)」を選択します。

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。

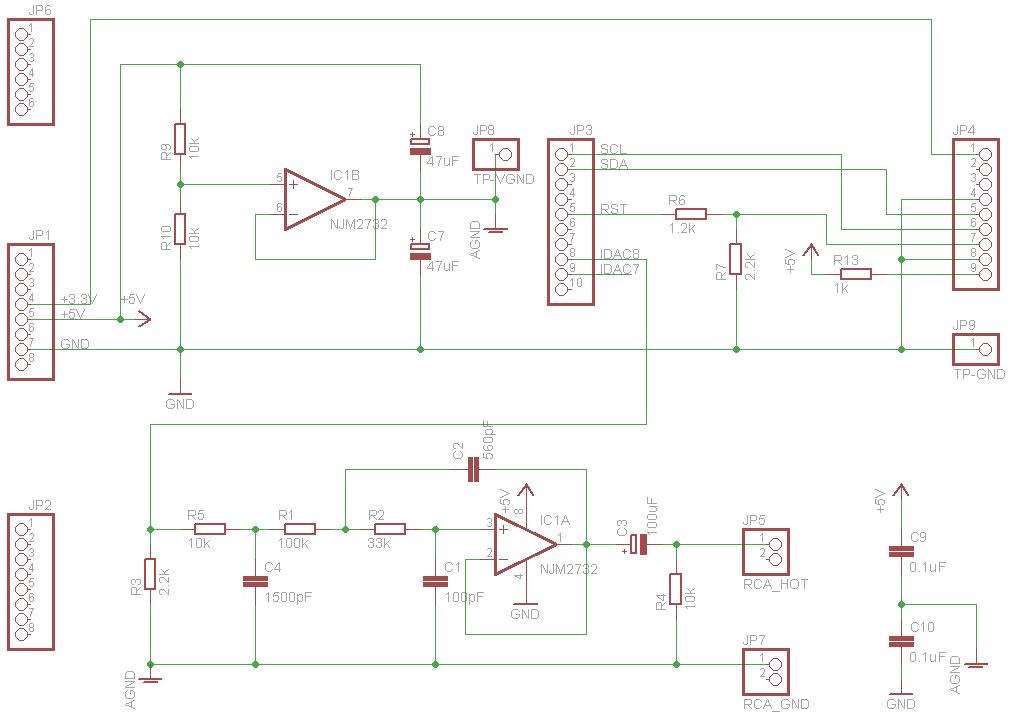
「Save Sound As」ダイアログに戻るので、「保存」ボタンを押せば16bit、1024個の波形がテキスト形式で保存されます。



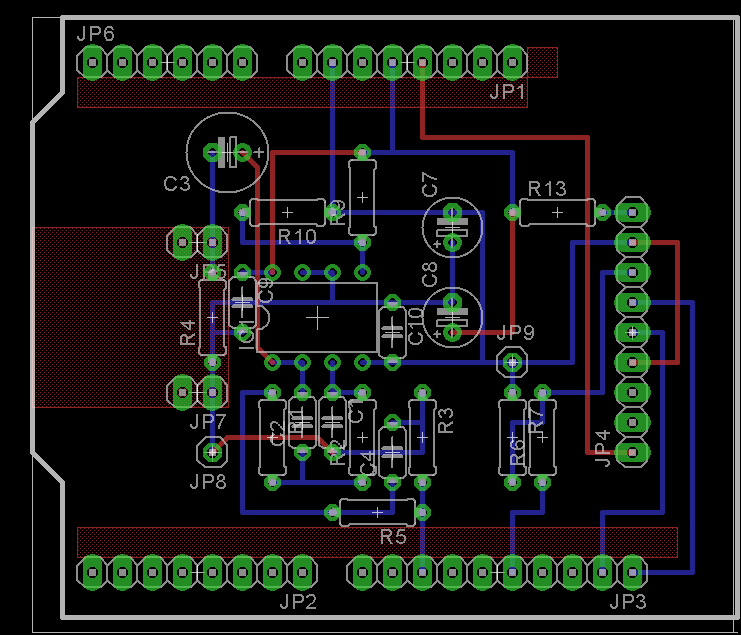
テキスト形式なのであとはExcelとかエディタのマクロなどで編集して12bit（0..4095の範囲）の配列の表記に変換してやれば<wavetable.h>に埋め込めると思います。

# 資料

## 回路図



## 基板図



## 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。

## ソースコード

<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  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* 波形パラメーター \*/  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)); // contast 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;    // Caluclate 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);    // Caluclate 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] = waveTableSqure;  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　PNPNMS（ ぴゅんぴゅんマシン製作所）  
<http://dad8893.blogspot.jp/>

Google+: ryood