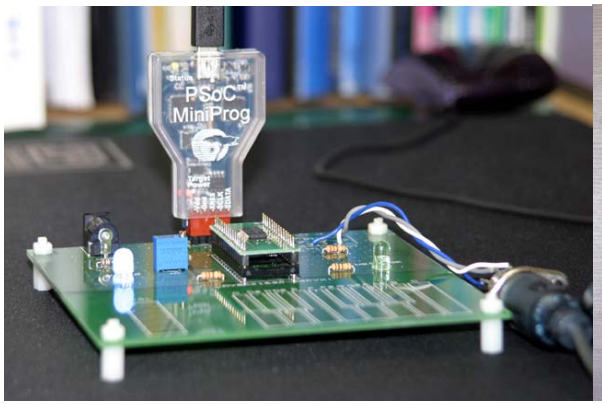
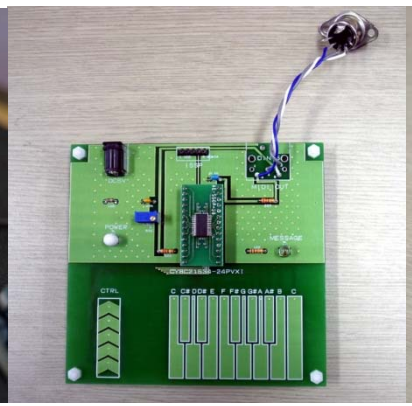
2010年度　電子情報概論

テーマ3：MIDIインタフェース

電子情報科学専攻　北川章夫



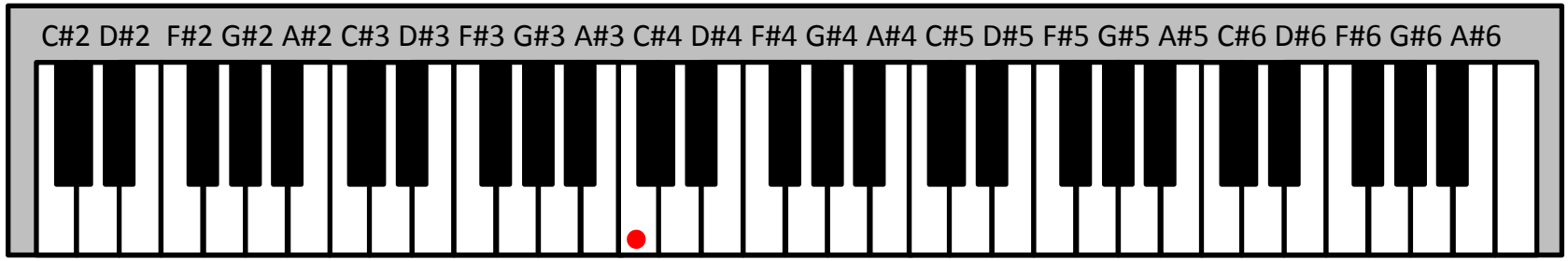



スケジュール

1. 予備知識１：音階と周波数
2. 予備知識２：MIDIハードウエアの基礎
3. 予備知識３：MIDIメッセージの基礎
4. Capsense を用いた MIDIキーボード回路
5. PSoC の開発環境とファームウエアの作成
6. 付録（I^2C Bridge によるチューニング）

[参考] 音名と階名

・音名(Note)は基準音の周波数（Pitch）に対する音階の相対表記
（基準音：A4 = 440Hz ～ 445Hzから選択）



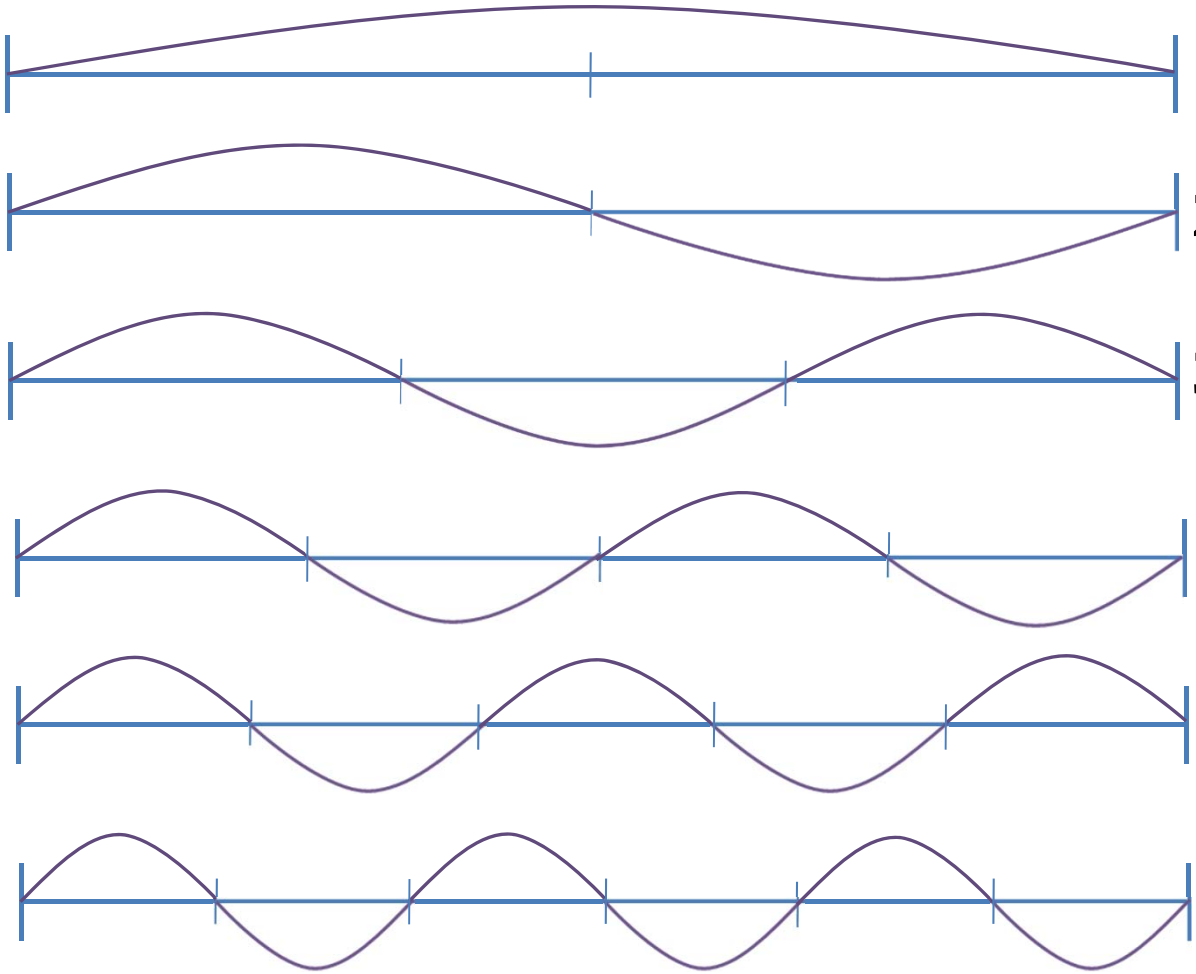
※1 数字はオクターブを表す（音名の国際表記法）。インターネット上では、CではなくAで数字を変更する表記も普及しているので注意
※2 低音楽器は意図的に他の楽器より僅かに高めのピッチにすることがある（自然に聞こえる）

・階名は主音（Key）に対する音階の相対表記

Fを主音とした階名（ヘ長調/ニ短調）：ド(F), レ(G), ミ(A), ファ(Bb), ソ(C), ラ(D), シ(E)・・・

※ 転調があるような複雑な曲では、階名が途中で切り替わって混乱するので、ミュージシャンは階名を使用しない。

持続振動（定常状態）の倍音構成



基準の周波数f_0

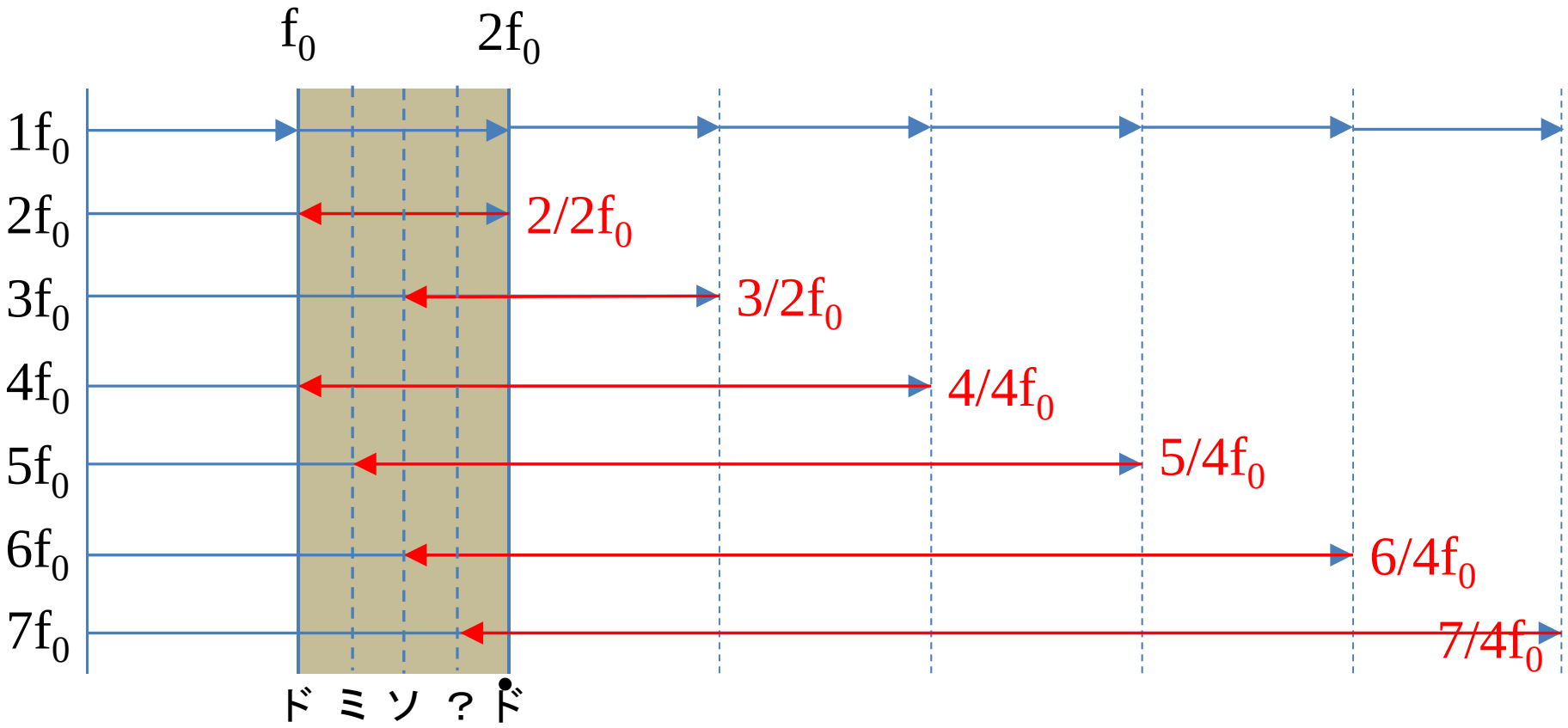
f_0の1オクターブ(8va)

f_0の2オクターブ(15ma)

注：オクターブ＝
周波数が2^N倍

※　自然界の持続振動は倍音の和によって構成されている

倍音と協和音（1オクターブ内への折り返し）



- f_0倍の音だけでメロディーを作ると音が飛びすぎて歌えないので、倍音をオクターブ下げてf_0 ～$2f_0$の間に入れて、同時に鳴らしてみる（加算する）
  - 倍音は持続振動音に含まれているので人工的に加算しても自然に聞こえる
  - 倍音をNオクターブ下げた周波数を加算しても持続振動と同様の安定感が感じられる（協和音と呼ぶ･･･安定感があるかどうかは心理的な問題であり人にもよる）

ピタゴラス音律（Pythagorean）
（ド-ソ音程を基礎とした音階）

- f_0, $(3/2)f_0$ の2個の周波数を音階として採用（多くの民族音楽も同様）
- 2音とそのオクターブだけでメロディを作るのは難しいので（お間抜けな感じになる）、さらに$(3/2)f_0$を基準として$(3/2)^2 f_0$も音階に採用
- これを繰り返してみると・・・

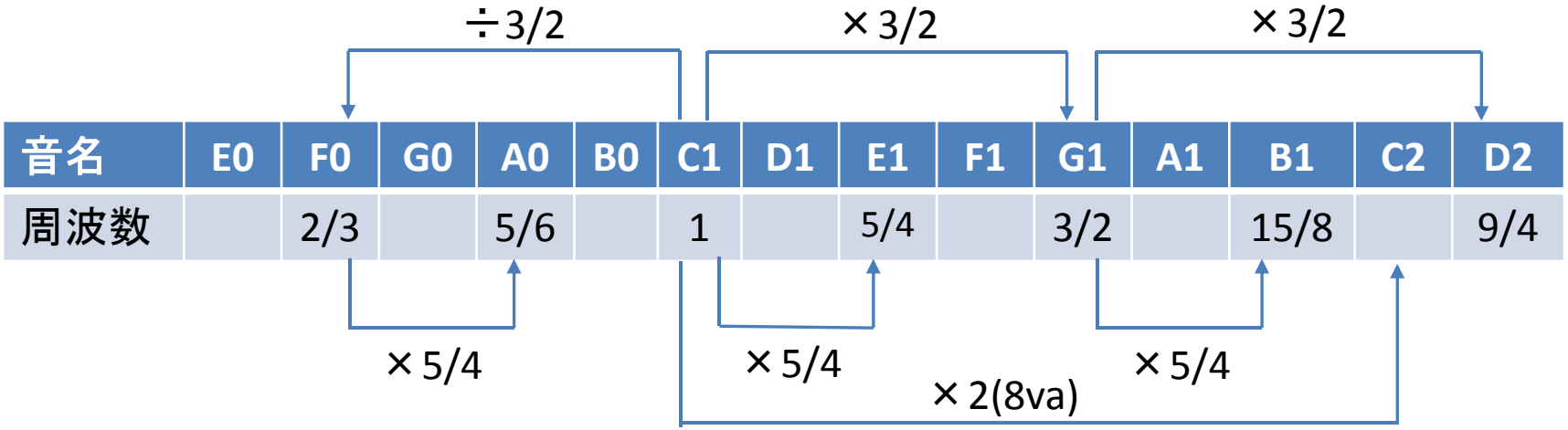
音名	C1	G1	D2	A2	E3	B3	$F4^{\#}$	$C5^{\#}$	$G5^{\#}$
指数表記 (K =3/2)	K^0	K^1	K^2	K^3	K^4	K^5	K^6	K^7	K^8
分数表記	1	3/2	9/4	27/8	81/16	243/32	729/64	2187/128	6561/256
小数表記	1	1.5	2.25	3.375	5.063	7.594	11.391	17.086	25.629

$D_6^{\#}$	$A_6^{\#}$	F_7	$\fallingdotseq C_8$
K^9	K^{10}	K^{11}	K^{12}
19683/512	59049/1024	177147/2048	531441/4096
38.443	57.665	86.498	129.746 → $\fallingdotseq 128 = 2^7$

$(3/2)^{12} \fallingdotseq 2^7$（7オクターブ上）と近似して、$(3/2)^{0\sim11}$ の12個の音階を得る

完全純正律（Just temperament）
（ド-ミ-ソ音程を基礎とした音階）

- f_0, $(3/2)f_0$, $(5/4)f_0$ の3個の周波数を音階として採用
- さらに、$(3/2)^{-1}f_0$, $(5/4)^{-1}f_0$も採用
- まだ足りないので、上記の周波数を基準とした(3/2), (5/4)倍音も採用
- これでC1を基準として7音の音階を作ってみると・・・



3/2倍と5/4倍の関係になっている7個の音階を得る（このように周波数を選ぶと、多くの音程間に3/2倍と5/4倍の関係が生じる）

3/2, 5/4倍音を利用した音階の問題点

- 古典調律（ピタゴラス音律、完全純正律など）やその他の民族音楽の音階の周波数は等比級数になっていない
  - 音律が同じ特定の楽器同士でないと合奏できない
  - 移調や転調ができない（長調と短調ですら楽器を持ち替えなければならない）
  - 言うまでもなく、部分転調やUpper Structureを多用した現代的なコードプログレッションの理論に基づくお洒落なフレーズなどは作れない

完全純正律の例（前スライド参照）

C1(ド=f_0)を基準とするとき、3/2倍の周波数（ソ）は、(3/2)f_0、
A0(ラ=(5/6)f_0)を基準とするとき、3/2倍の周波数（ミ）は、(15/12)f_0、
となったとき倍音を感じさせる響きとなる
・・・　しかし、完全純正律ではミは、(5/4)f_0に調律されているので
A0との間に倍音を感じさせられない

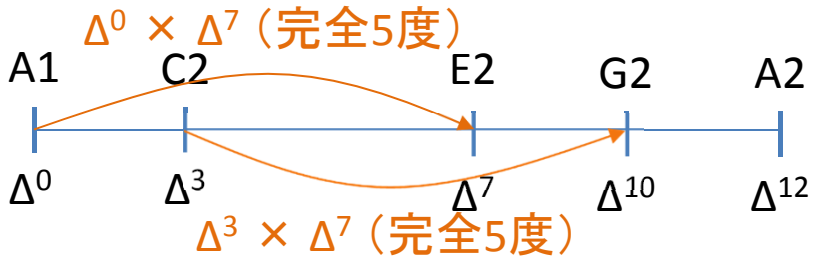
平均律（Equal temperament）

- 音階の周波数比率を一定とし、等比級数で周波数を表す
- ピタゴラス音律に合わせて、周波数を12個/1オクターブに分割する
- 12回掛けると2（1オクターブ）となる数値 Δを求めると

$$\Delta = \sqrt[12]{2} = 1.059463094\cdots$$ （ピタゴラスが嫌いな無理数だが）

参考：$\Delta_C = \sqrt[1200]{2} = 1.00057779\cdots$ の周波数比はセント(cent)と呼ばれている

音名	A1	A1#	B1	C2	C2#	D2	D2#	E2	F2	F2#	G2	G2#	A2
指数表現	Δ^0	Δ^1	Δ^2	Δ^3	Δ^4	Δ^5	Δ^6	Δ^7	Δ^8	Δ^9	Δ^{10}	Δ^{11}	Δ^{12}
小数表現	1	1.059	1.122	1.189	1.260	1.335	1.414	1.498	1.587	1.682	1.782	1.888	2



音程間の周波数比は一定になる（どの音名を基準にしてもよい）

平均律の長所と短所

- 移調や転調が自由にできる
- 一つの楽器（調律）で（近似的に）全ての楽曲に対応できる
- 平均律に調律された楽器は、ピッチを合わせれば合奏ができる
- 倍音の周波数と関係がない値なので、オクターブ以外には倍音由来の周波数を含まない（現代人は平均律に慣れているのでΔ^7, Δ^{10}などが協和音音程に聞こえる）
- 周波数比が無理数であるため、基準周波数の分周によって、オクターブ以外の周波数を作り出すことが出来ない
- 民族音楽にはなじみにくいことが多い（ような気がする）

近年のディジタル鍵盤楽器は、音階のチューニングや1タッチで各種音律が呼び出せるものが多いので、これを使うと古典音楽や民族音楽の雰囲気が出る。
但し、弦楽器では、開放弦をチューニングし直せば（所謂、変則チューニング）、微妙な音程は、指先でコントロールできるので耳さえ鍛えればOK。

音階のまとめ

音名	平均律	ピタゴラス音律	完全純正律
C2	$\Delta^0 = 1$	1	1
C2#	$\Delta^1 = 1.059$	2187/2048 = 1.068	(16/15 = 1.067)
D2	$\Delta^2 = 1.122$	9/8 = 1.125	9/8 = 1.125
D2#	$\Delta^3 = 1.189$	19683/16384 = 1.201	(6/5 = 1.2)
E2	$\Delta^4 = 1.260$	81/64 = 1.266	5/4 = 1.250
F2	$\Delta^5 = 1.335$	177147/131072 = 1.352	4/3 = 1.333
F2#	$\Delta^6 = 1.414$	729/512 = 1.424	(45/32 = 1.406)
G2	$\Delta^7 = 1.498$	3/2 = 1.5	3/2 = 1.5
G2#	$\Delta^8 = 1.587$	6561/4096 = 1.602	(8/5 = 1.6)
A2	$\Delta^9 = 1.682$	27/16 = 1.688	5/3 = 1.667
A2#	$\Delta^{10} = 1.782$	59049/32768 = 1.802	(16/9 = 1.778)
B2	$\Delta^{11} = 1.888$	243/128 = 1.898	15/8 = 1.875
C3	$\Delta^{12} = 2$	531441/262144 ≒ 2.000	2/1 = 2.000

A1 = 440～445Hz

E2 ← 誤差1%以上

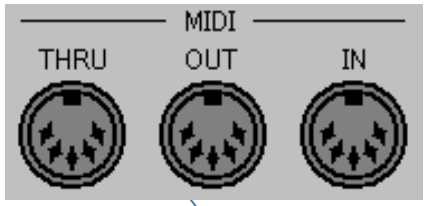
(人間の聴覚に対して、2セント = 0.1%程度の精度が必要とされている)

A2 ← 誤差1%以上

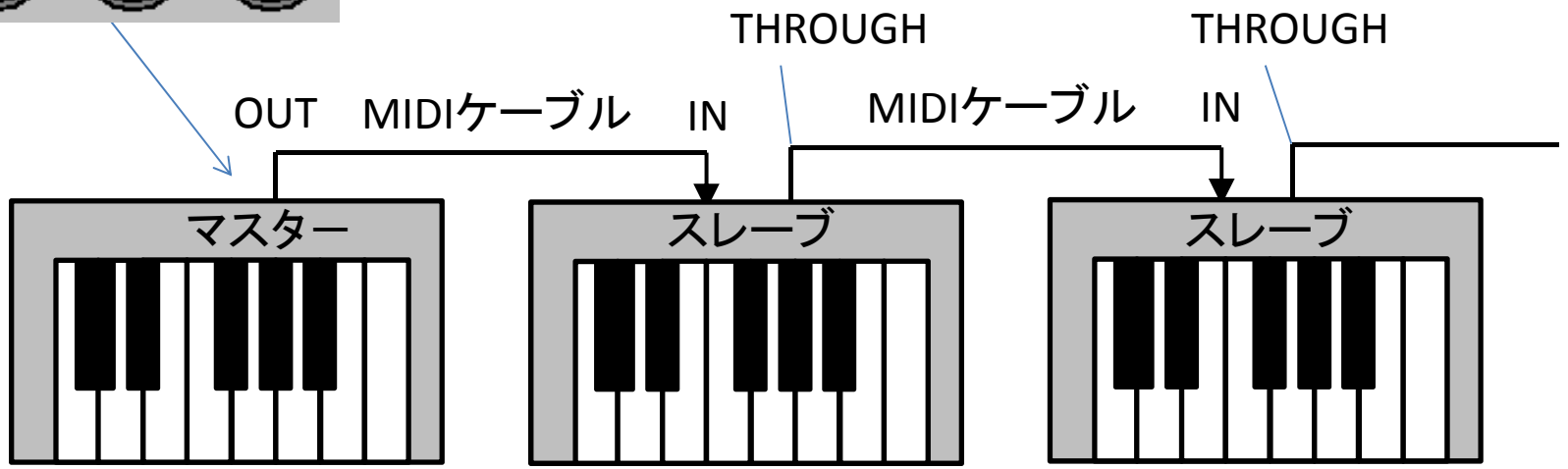
※　協和音を出すときだけ音程の調整を自動的に行う平均律楽器があれば理想的
（バイオリンややエレクトリックギターでは演奏者が不協和度をコントロールしている）

Musical Instrument Digital Interface

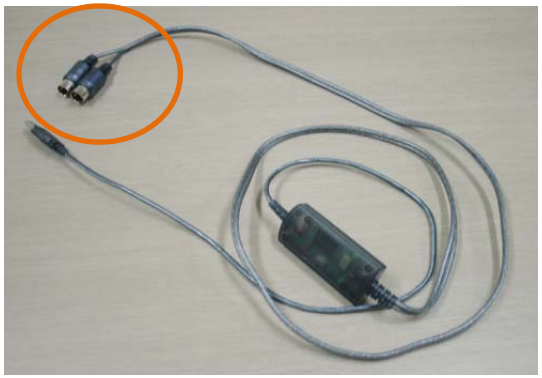
MIDIインタフェースのハードウエア



規格としては、IN, OUT, THRU(THROUGH)があるが、最近は、IN, THROUGH/OUT切り替えの2ポート型の機器が多い



DIN5コネクタ

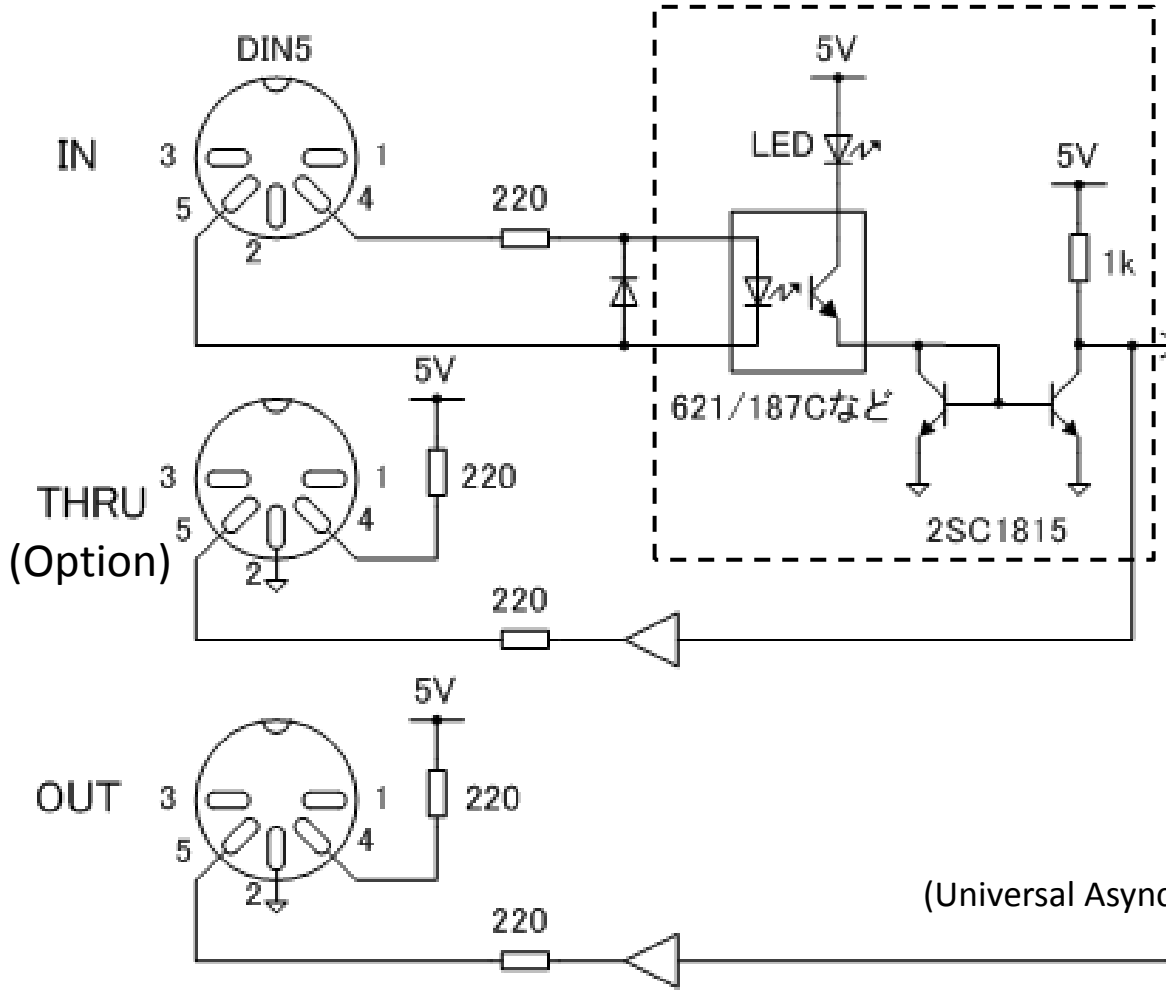


PC接続用のMIDI-USB変換ケーブル

MIDI側は丸ピンDIN5

ソフトウエアから見ると、シリアル通信と同じだが、ハードウエアは送信と受信が分離している

MIDIインタフェースの回路例



通常はCMOSレベルのドライバ付きのフォトカプラを使用

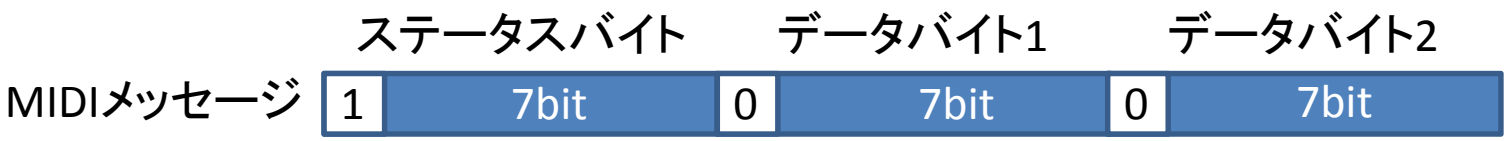
8051, HITAC6301, NEC7810などの仕様を引きずっている

(Universal Asynchronous Receiver Transmitter)

DIN5コネクタはメス型を前面から見たとき

MIDIの制御方式

- MIDIデータ通信の方式
  - 機器の制御情報（MIDIメッセージ）をマスタが送り、必要なメッセージのみをスレーブが実行する。マスタはスレーブの状態（エラーなど）は関知しない。
- クロック
  - 転送速度：32.25kbps（MIDI転送レート）
  - Serialクロック：32.26k * 8 = 250kHz
- MIDIメッセージのフォーマット
  - ステータスバイト（命令）（80H～FFH）とデータバイト（数値）（00H～7FH）の列により構成される
    - つまり、MSBをフラグとして使用している
  - データバイトの個数はステータスバイトの種類によって異なる



参考：JIS　X 6054-1, X 6054-2
JMSC　MIDI 1.0規格Ver.4.1

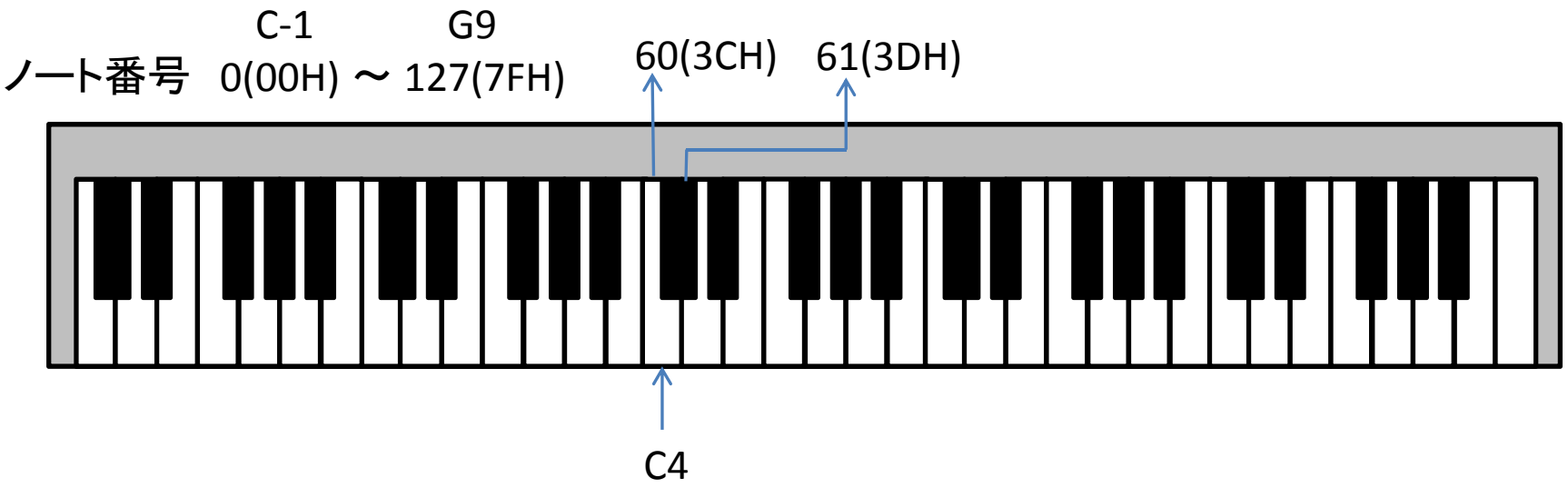
ステータスバイト

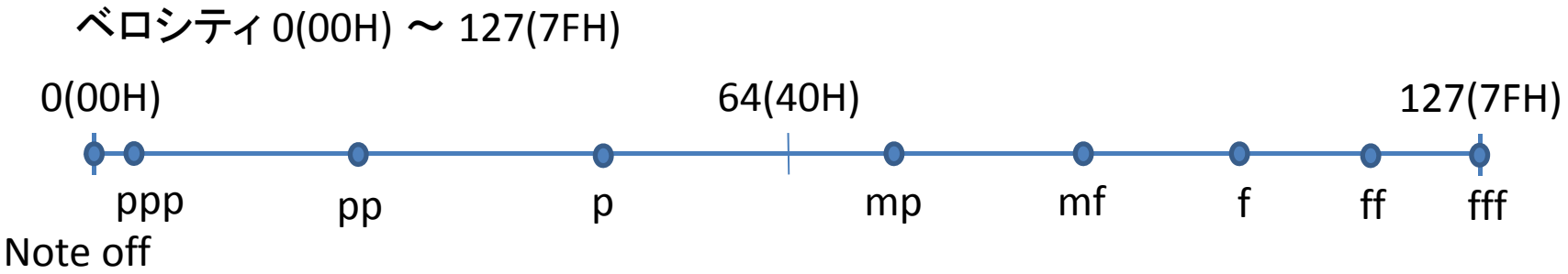
- MIDIチャネル n
  - 各機器に予めMIDIチャネル（n=1~16）を設定しておき、マスターはステータスバイトの中でMIDIチャネルを指定して機器を選択する
  - MIDIチャネルは、ステータスバイト中の下位4bit（0H～FH）で指定する
    - n = 0HはMIDIチャネル1と呼ばれるので注意
    - n = 9H(MIDIチャネル10)は、打楽器（ドラム/パーカッション）に割り当てることが業界の暗黙の了解となっている(GM音源規格)
    - 16チャネル以上必要な場合は、ポートを増やす（MIDI規格の対応なし）

メッセージの例（この他に非常に多くの種類がある。詳しくはMIDIバイブルI・II参照）

メッセージ名	メッセージデータ (HEX)	備考
Note On	9n note(1bite) velocity(1Byte)	発音を開始する
Note Off	8n note(1bite) velocity(1Byte) 9n note(1Byte) 00H	発音を停止する (リリースを開始する)
Pitch Bend Change	En LSB(1Byte) MSB(1Byte)	音程を連続的に変化させる
Program Change	Cn program_No(1Byte)	音色を切り替える

データバイト（Note#, Velocity）





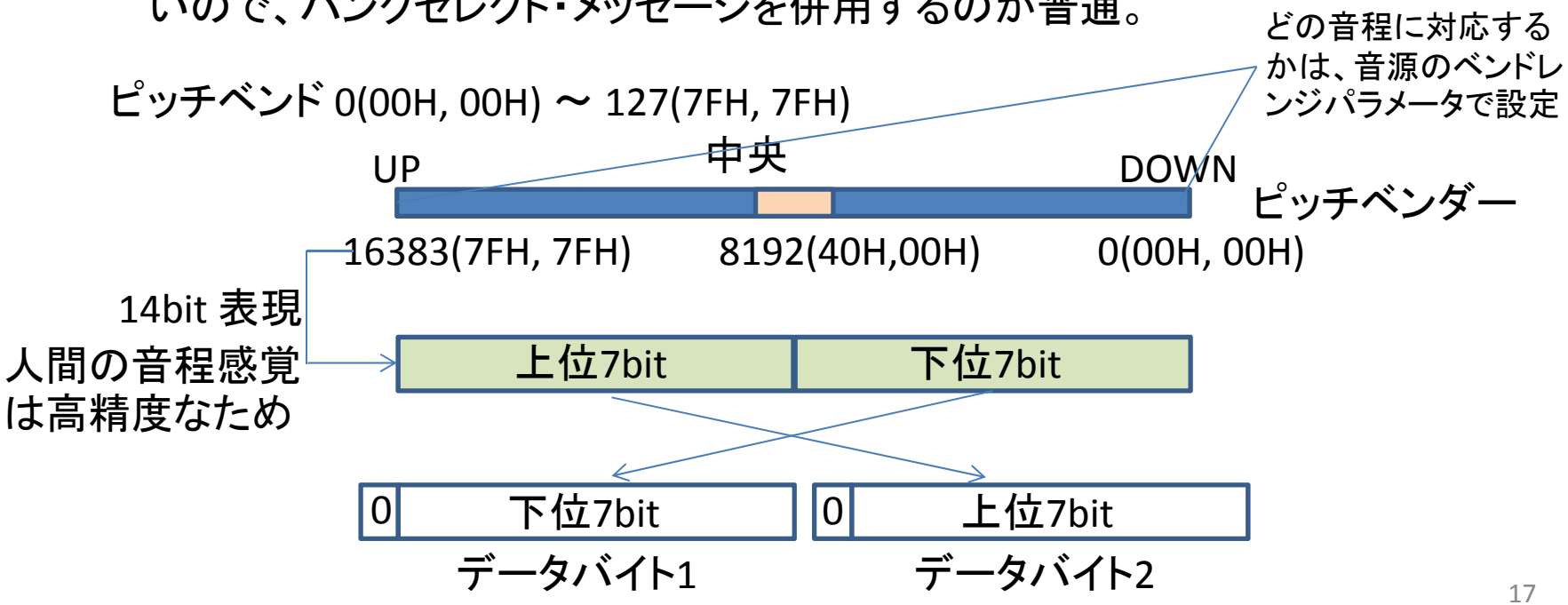
注：この図は大まかな目安。MIDI規格では強さ記号は定量化されていない。
音の強さは、単なる音量ではなく、エンベロープやスペクトラムの変化に関係しているため演奏者の感覚に合わせて音色毎にチューニングが必要。

データバイト（Program#, Pitch Bend）

プログラム番号 0(00H) ～ 127(7FH)

音色(Timbre)を切り替える。音色とプログラム番号の関係は、ユーザが予め音源に設定するかメーカ規格（GS（ローランド）、XG（ヤマハ）、GM（メーカ共通））によって異なる。

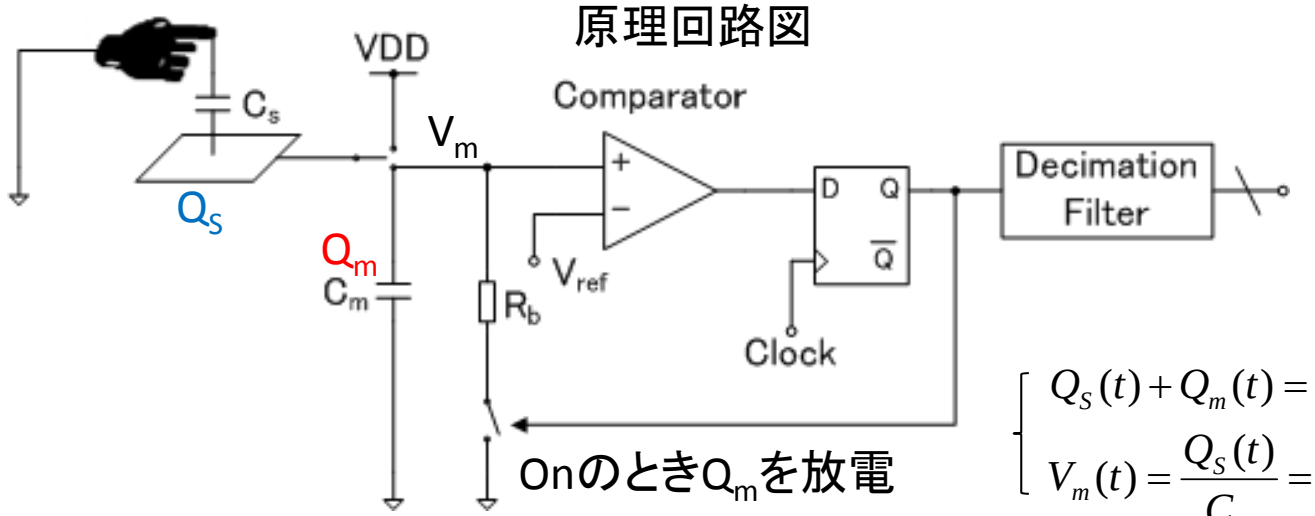
プログラム・チェンジ・メッセージだけでは、音色が128種類しか選べないので、バンクセレクト・メッセージを併用するのが普通。



PSoC CapSenseの動作原理

PSoC CY8C24***, CY8C21*** シリーズではCSD(CapSense Sigma-Delta)方式の高精度タッチセンサ・モジュールが使用できる。（PSoC FirstTouchはCY8C21434を搭載）

原理回路図



$$\begin{cases} Q_S(t)+Q_m(t)=C_S VDD+Q_m(t-T_S) \\ V_m(t)=\dfrac{Q_S(t)}{C_S}=\dfrac{Q_m(t)}{C_m} \end{cases}$$

Z変換

$$\begin{cases} Q_S(z)+Q_m(z)=C_S VDD+z^{-1}Q_m(z) \\ V_m(z)=\dfrac{Q_S(z)}{C_S}=\dfrac{Q_m(z)}{C_m} \end{cases}$$

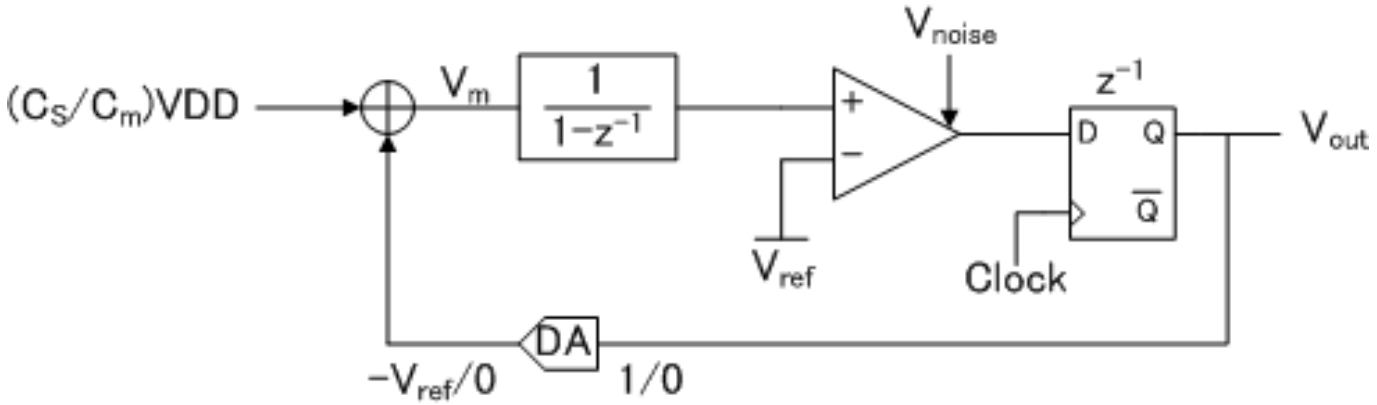
整理

(C_S/C_m)VDDを入力してV_mを出力する積分器として動作

$$V_m(z)=\frac{C_S}{C_m}\frac{VDD}{1-z^{-1}+\dfrac{C_S}{C_m}}\cong VDD\frac{C_S}{C_m}\frac{1}{1-z^{-1}}$$

(Euler Transformation)

CapSenseの変換精度



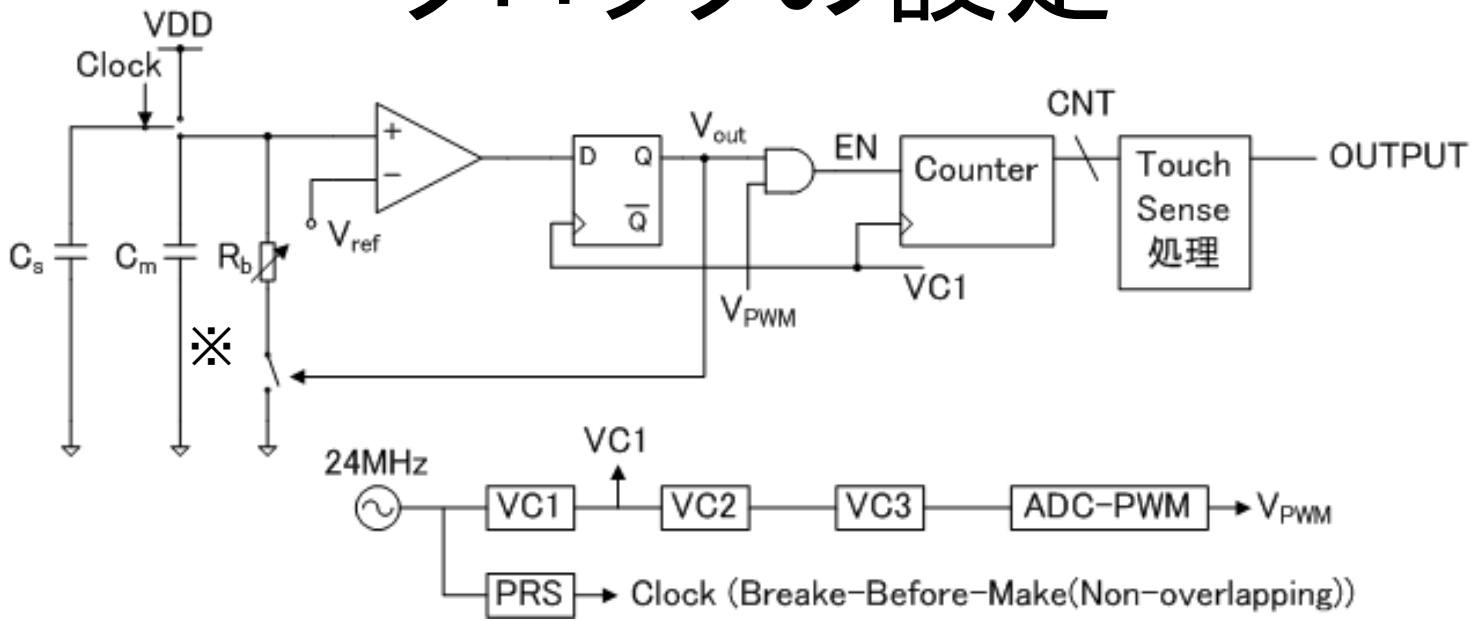
$$V_{out}(z) = \frac{z^{-1}}{1-z^{-1}}\{\frac{C_S}{C_m}VDD - V_{out}(z)\} + V_{noise}$$

$$V_{out} = z^{-1}\frac{C_S}{C_m}VDD + \underbrace{(1-z^{-1})}V_{noise}$$

1次DSMとして動作(-30log(OSR)の量子化誤差)

容量C_Sを入力してディジタル変換したと考える

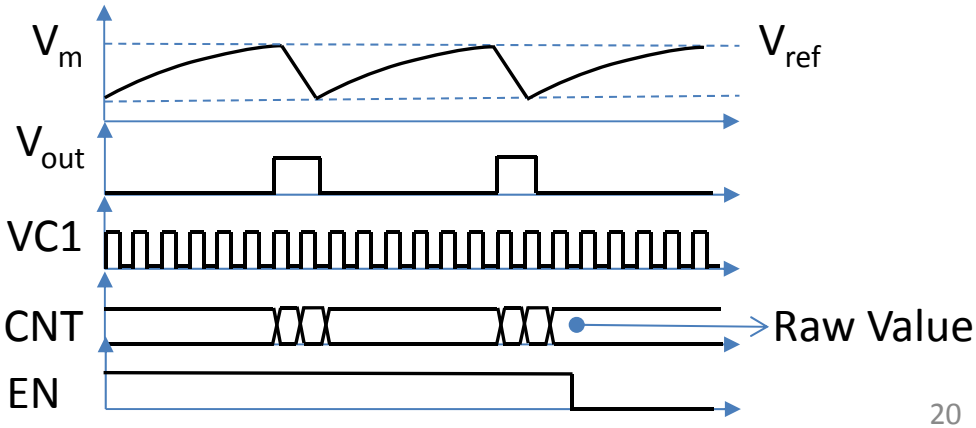
クロックの設定



VC1~VC3: Divider, PWM Pulse width modulator, PRS: Pseudo-Random Sequence Generator

V_{PWM}: カウンタのゲート時間
VC1: サンプリング・クロック
Clock: スイッチトキャパシタ積分時定数の調整

※ Csが小さい場合は、Rbの値を大きくしてカウント値を大きくできる



出力データとパラメータ

出力変数

変数	内容
waSnsResult[Sensor#]	現在のカウンタ出力（Raw Count）
waSnsBaseline[Sensor#]	ドリフト成分 (Baseline)
waSnsDiff[Sensor#]	Raw Count – Baseline

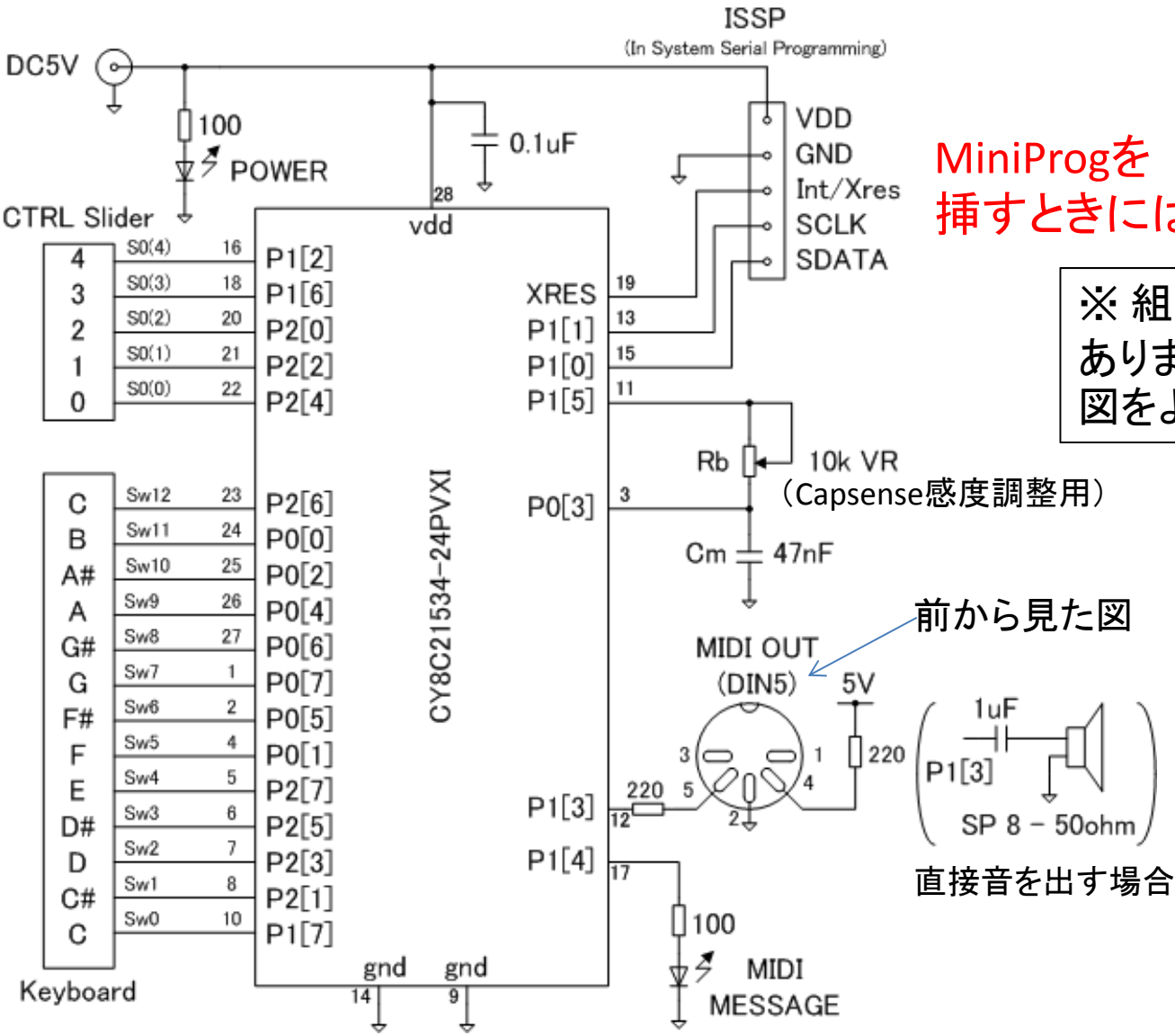
判定用パラメータ

パラメータ	内容
Finger Threshold	waSnsDiffがこの値を超えたら、センサをONとしてbaSnsMaskに1を入力（タッチの完了を判断）
Noise Threshold	waSnsBaselineとwaSnsResultの差がこの値を超えたらDiffを計算開始（タッチの開始を判断）

- Raw Count < Noise Thresholdのとき Diff = 0
- Raw Count >= Noise Thresholdのとき Diff = Raw Count - Baseline
- Diff >= Finger Threshold のときタッチ状態と判断し baSnsMask[8]変数またはbIsSensorActive(Sensor#), bIsAnySensorActibe()関数に出力

実際には非常に多くのパラメータで判定しているので、詳細はCSDのDatasheetを参照

MIDIキーボードの回路図



MiniProgを
挿すときには向きに注意

※ 組み立ての説明はありませんので回路図をよく見てね。

LEDは好きな色を付けてください。ただし、負荷抵抗を計算し直すこと。

直接音を出す場合

PSoC開発環境の準備

インストール順序

1. PSoC Programmer（書き込み用）
2. PSoC Designer（開発用）
3. CY3240 USB-I^2C Bridge Software
   （PCで変数をモニタできる：デバッグ/チューニング用に使用）

ダウンロード：　http://www.cypress.com/

ソフトウエアのダウンロード > キーワード検索で探してインストール

CY3240-I^2C Bridge Softwareは、USB-I^2C Bridge 本体を持っていないと使えない。
なくても本実習では問題ない。UARTで代用することも可能（ポートが余っていれば）。

PSoC CSD(Capsense)の設定

1. Device = CY8C21534-24PVXI でプロジェクト作成
2. CSD (without clock prescaler)を配置
3. CSD Wizardの設定（workspaceでCSD_1を右クリック）

Global Settings of CSD wizard	
Buttons	13
Sliders	1
Modulator Capacitor	P0[3]
Feedback Resistor	P1[5] ※

Sensors Settingタブは、回路図を見て、センサをPinと接続しておく。

※ USB-I²C Bridgeを使用するときは、PSoCその他によるP1[0], P1[1] (ISSPポート)の使用を避け、USB-I²C BridgeとISSPを兼用にする。

PSoC MIDIキーボードの設定

Global Resource	
Power Setting	5.0V/24MHz
CPU Clock	SysClk/8
VC1	4
VC2	8
VC3	32

CSD_1 properties	
Resolution	10
Sensors Autoreset	Disabled
Debounce	1
Reference	ASE11

LED_1 properties	
Port	Port_1
Pin	Port_1_4
Drive	Active High

TX8_1 properties	
Clock	VC2
Output	Raw_0_Output_3
TX Interrupt Mode	TXRegEmpty（任意）
ClockSync	Sync to SysClk
Data Clock Out	None

TX8出力 – RO0[3] – GlobalOutOdd_3 – Port_1_3 P1[3] に接続しておく。

IMO	VC1=6	VC2=16	VC3=32
24MHz	4MHz	250kHz	7.8kHz

TX8のクロックは、31.25kbps * 8 = 250kHz

CSDは任意のVC1, VC2, VC3を設定できないので（Datasheet参照）、起動後にファームウエア上で、VC1, VC2のレジスタを変更する（別紙：ソース1参照）

PSoC音源のピッチの計算

以下は、おまけ(音階計算の演習問題)。
せっかく音階を学んだので、スピーカを繋いでPSoCで音を出してみる。
（内部クロックの周波数精度は数%程度なので実用にはならない音源だが）

Capsenseの設定に合わせて

SysClk = 24MHz
VC1 = 4
VC2 = 8
のときVC2の周波数は、
f_{ref} = 24MHz/4/8 = 750kHz

平均律では

A4 = 440Hz のとき
C5 = $440(\sqrt[12]{2})^3$ = 523.251Hz
C8 = C5 *8 = 4.18601kHz
（C4を出したいが、PWM8(8bit)では出せない）

f_{ref}を基準周波数として、PWM8により分周して音階の周波数を作り出す
（PWM8では周波数精度が低すぎだが、ディジタルブロックが1個しか残っていない）

fref/f(C8) = 179.168 ⇒ PWM8の周期を179(クロック)に設定する（誤差 -0.094%）
(A4 = 440.4Hz ピッチに相当）

PSoC音源の音階周波数の計算

音名	平均律(kHz)	純正律(kHz)	PWM8の周期	誤差(%)
C8	4.18601	4.18601	179(179)	-0.094
C#8	4.43492	4.46508	169(168)	-0.067
D8	4.69864	4.70926	159(159)	-0.39
D#8	4.97803	5.02321	151(149)	0.22
E8	5.27404	5.23251	142(143)	-0.15
F8	5.58765	5.58135	134(134)	0.22
F#8	5.91991	5.88658	127(127)	0.24
G8	6.27193	6.27902	120(119)	0.35
G#8	6.64488	6.69762	113(112)	0.12
A8	7.04000	6.97668	107(108)	0.44
A#8	7.45862	7.44180	101(101)	0.44
B8	7.90213	7.84877	95(96)	0.094
C9	8.37202	8.37202	90(90)	0.46

PSoC音源の設定

Global Resource	
Power Setting	5.0V/24MHz
CPU Clock	SysClk/8
VC1	4
VC2	8
VC3	32

CSD_1 properties	
Resolution	10
Sensors Autoreset	Disabled
Debounce	1
Reference	ASE11

LED_1 properties	
Port	Port_1
Pin	Port_1_4
Drive	Active High

PWM8_1 properties	
Clock	VC2
Enable	High
CompareOut	Row_0_Output_3
TerminalCountOut	None
CompareType	Less Than Or Equal
Interrupt Type	Terminal Count（任意）
ClockSync	Sync to SysClk
Invert Enable	Normal

PWM8出力 – RO0[3] – GlobalOutOdd_3 – Port_1_3 P1[3] に接続しておく。

PWM8のPeriod, PulseWidthは、ファームウエアで設定（別紙：ソース2参照）

USB-I2C Bridgeによる変数モニタ

Capsenseのパラメータは多いので、カウント値を見ながら調整できると便利なので、USB-I2C Bridge を使ってみる。（TX8SWを使ってもよいがRS232C変換や通信用ポートの追加が必要なので少し面倒くさい）

EzI2Csを配置（ブロックは消費しない）

EzI2Cs_1 properties	
Slave Addr	0（任意）
Address Type	Static (Dynamic:ファームで指定)
ROM Registers	Disable
I2C Clock	400k Fast
I2C Pin	P[1]0-P[1]1

Global Settings of CSD wizard	
Buttons	8
Sliders	0
Modulator Capacitor	P0[3]
Feedback Resistor	P1[5] ※

センサを減らしておかないとメモリが足りない

バッファとBridge Control Panelの概要

ファームウエア

```
struct I2C_Regs {
    WORD RawCount;
    WORD Baseline;
    WORD Diff;
    BYTE Active;
} sensData;
```

バッファ用構造体の宣言（グローバル）

```
EzI2Cs_1_SetRamBuffer(sizeof(sensData), 0, (BYTE *) &sensData);
EzI2Cs_1_Start();
```

初期化

```
sensData.RawCount = CSD_1_waSnsResult[0];
sensData.Baseline = CSD_1_waSnsBaseline[0];
sensData.Diff = CSD_1_waSnsDiff[0];
sensData.Active = CSD_1_waSnsResult[0];
```

センサ番号

バッファへの代入

アクティブなセンサのbit mask（#0 ~ 8）

スイッチ=0, スライド=1以降

Bridge Control Panel（CY3240に付属のソフト）

1. メニュー：chart > Variable Settings で読み出す変数のサイズを設定
2. 電源を供給
3. コマンド入力(ex.)：r 00 @1RawCount @0RawCount
   (00はアドレス、Byte列をbig-endian で読み出すこと)