

## プログラムを書くーその1：基本サブルーチンを作る

第2章で作った「試作 CORE」用の基本的なサブルーチンを作成してみましょう。

マイコンのプログラミングには、データシートをこまめにチェックして確認することが重要です。AVRは残念ながら英語のデータシートしか公開しておりませんが、英語のデータシートに挑戦し読んで見ることもお勧めします。

幸い、日本語に翻訳されたデータシートが公開されているので、それらを利用することも可能かと思えます。

ロボット用のプログラムを作る前に、基礎的なサブルーチンを用意しておきます。そして、ロボットの形状や目的によって、それらのサブルーチンを組み合わせて目的を持たせたプログラムにしていきたいと思えます。

### 基礎の説明

マイコンのプログラムを始める前に、次の三つのキーワードを押さえておきましょう。

- 1：I/Oポート
- 2：レジスタ
- 3：割り込み

これらを押さえておけば、色々な事ができるようになります。

特にレジスタの使い方が重要です。

レジスタの名前、たとえば「Timer/Counter Register」は略されてTCNTなどという名前でプログラムやデータシートの中で表記されています。

この略された短い名前から、本来の長い名前を思い浮かべられるようになってくると、かなりプログラミングでも楽になってきます。

### I/Oポート

マイコンは小さなコンピュータなので、その中では演算を行ったりするわけですが、マイコンでほかにも重要なI/Oポートと言うものがあります。IはInput(入力)とOはOutput(出力)の略です。

一つひとつのI/Oは入力、もしくは出力をすることが可能なわけで、今回使用している、ATmega88/168では、23個のI/Oがあります。

それぞれのI/Oには名前が付いていて、次のようなピン配列になっています。

#### ピン配列

(PCINT14/RESET) PC5	1	28	PC5 (ADC5/SCL/PCINT13)
(PCINT18/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)
(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)
(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)
(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)
(PCINT17/SXCK/IO) PD4	6	23	PC0 (ADC0/PCINT8)
VCC	7	22	GNP
GND	8	21	AREF
(PCINT8/XTAL1/TOSC1) PD8	9	20	AVCC
(PCINT17/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)
(PCINT21/OC0B/INT1) PD5	11	18	PB4 (MISO/PCINT4)
(PCINT22/OC0A/AIK0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)
(PCINT23/MIN1) PD7	13	16	PB2 (SS/OC1B/PCINT2)
(PCINT0/CK/KD/CP1) PB0	14	15	PB1 (DC1A/PCINT1)

左上にある PC6 が I/O ピンの名前です。よく見ると、例えば、PB0, PB1, PB2, PB3, PB4, PB5, PB6, PB7 というふうに、PB0~PB7 までの一つの集まりがあることが分かると思います。

AVR では、八つの I/O ピンを一つの集まりとして、I/O ポートがあります。

また、名前の隣に PD0 ピンであれば (PCINT16/RXD) とあります。これは、汎用の I/O ピンの機能のほかに、PCINT16 と RXD という二つの機能があることを示しています。

このように、一つのピンに複数の機能を持たせることはマイコンでは一般的です。

ATmega88/168 では少し変則的ですが次の表のようになっています。

ポート名	ピン	説明
PB	0, 1, 2, 3, 4, 5, 6, 7	PB6/PB7 はクロックの入力とも兼用
PC	0, 1, 2, 3, 4, 5, 6	アナログ入力などにも使われる PC4/PC5 は I2C としても兼用 PC6 はリセットとも兼用
PD	0, 1, 2, 3, 4, 5, 6, 7	タイマや SPI などの機能とも兼用

I/O ピンに対して、出力にするか、入力にするか、などを決めるのがレジスタです。

## レジスタ

マイコンの設計思想により違いがありますが、AVR の場合レジスタは特定のアドレスにマップされたメモリと言う形で存在しています。

このレジスタはマイコンの持っている機能を実現するためのスイッチの集まりともいえます。

例えば I/O ポートに対しての入出力を行うのであれば、三つのレジスタを使用します。更に、タイマなどを使う場合だと、7個ほどのレジスタを使っています。

このレジスタの設定方法が説明されているマニュアルがマイコンごとにあるのでそれを熟読しながらマイコンのプログラムをしていくと言う事になります。

## 割り込み

割り込みはとても便利な機能です。メインループでぐるぐる回っていて、I/O ポートが変化したときに割り込みを発生させたり、特定の時間がたったら割り込みを発生させたりという事が可能になります。

例えば、シリアル・ポートなどを使っていて、外部からデータが送られてきたら割り込みを発生するようにして、受信したデータを別のメモリにコピーしておく、というような事が可能になります。

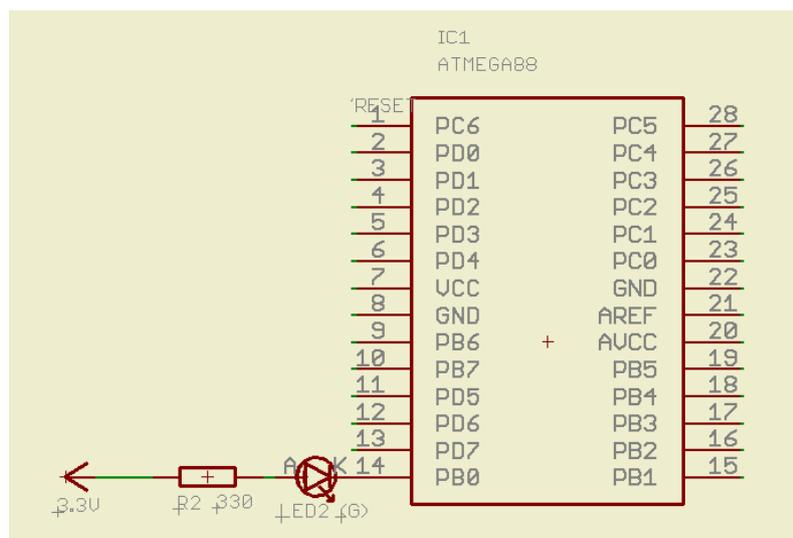
これらの設定は前に書いたレジスタをセットすることで実現できます。

## LED を点滅 (I/O ポートの出力の制御方法)

ブレッドボードの上に作成した「試作 CORE」は PB0 ピンに LED を接続しています。電氣的には PB0 を LOW にすると LED が光り、HI すると消えるような回路になっています。

接続するのは、PB0 ピンにしています。

### LED の回路



I/O ポートの入出力を決定するために使用するために、二つのレジスタを使用します。

レジスタ名	レジスタ名	説明
DDxn	Data Direction Register	入力か出力か決定する。 対応 bit が 1 の時が出力になり 0 の時入力になる。
PORTx	Port X Data Register	対応 bit に 1 を書き込むとピンが Hi になり、0 を書くと Low になる。

x の部分にポート名の B や C、D などのアルファベットが入ります。

n の部分にピン番号が入ります

LED を On/Off するためには、次のようなステップになります。

プログラム	説明
DDRB = 0b00000001;	PB0 ピンを出力にするために、bit で 1 を書き込む 0b00000001; の意味は コラム-C 言語での数値表現 を参考にしてください
PORTB = 0b00000001;	PORTB の 0bit 目 (PB0) を 1 にして LED を消す
PORTB = 0b00000000;	PORTB の 0bit 目 (PB0) を 1 にして LED をつける。

この例では、PORTB = 0b00000001; と 2 進数の値を代入しているので、0bit 目だけではなくて、他の bit も全て変えてしまいます。しかし、これでは問題があります。

そのようなときは、C 言語の bit 演算を使います。

具体的には次のように書きます

DDRB = DDRB   0b00000001;	Or する事で、DDRB の 0 ビット目を 1 にする
PORTB = PORTB   0b00000001;	Or する事で、PORTB の 0 ビット目を 1 にする
PORTB = PORTB & 0b11111110;	And する事で、DDRB の 0 ビット目を 0 にする

さらに、C 言語では更に簡単に表現する方法が用意されています。  
具体的にはまったく同じ意味の操作を次のように書くことが可能です。

DDRB  = 0b00000001;	Or する事で、DDRB の 0 ビット目を 1 にする
PORTB  = 0b00000001;	Or する事で、PORTB の 0 ビット目を 1 にする
PORTB &= 0b11111110;	And する事で、DDRB の 0 ビット目を 0 にする

しかし、実際に、このように二進数で bit を書いていくと 0 や 1 が多く並んでいて、ミスをしやすくなります。

例えば PB5 のビットであれば、シフト演算を使い次のようにして書きます。

```
DDRB |= (0x01 << 5);
```

このように 5 という数字を見ると、あ、5 bit 目である事が容易に理解できます。  
いちいち ( 0x01 << n ) と書くのも面倒ですので、

```
#define _BV(n) (0x01 << n)
```

というようにマクロを定義、さらにそれぞれのレジスタの各ビットは、#include <avr/io.h> というファイルで定義されているのでそれを使い、

DDRB  = BV( PB0 );	Or する事で、DDRB の 0 ビット目を 1 にする
PORTB  = BV( PB0 );	Or する事で、PORTB の 0 ビット目を 1 にする
PORTB &= ~BV( PB0 );	And する事で、DDRB の 0 ビット目を 0 にする

というように表現する事が AVR のプログラミングでは良く使われています。  
逆に言うと BV のマクロの内容や、PB0 の値の意味などが分からないと、

```
DDRB |= BV( PB0 );
```

と書かれたら、すぐには理解できないと思います。  
地道な作業ですが、このようなマクロがどのように定義されているか、レジスタの各 bit がどのような名前前で定義されているのは、ヘッダーファイルの中をおくと良いでしょう。

## サンプル・プログラム

今回用意したサブルーチンにするほどの事はないので、マクロとして定義しました。

```
#define LED_Init()      DDRB  |= BV( PBO )
#define LED_On()       PORTB &= ~BV( PBO)
#define LED_Off()      PORTB |= BV( PBO )
```

テストとして On/Off を続けるプログラムがリスト 1 です

```
リスト 1
(program/LED_TEST/main.c)
```

プログラムの中に

```
for (vu16 w=0; w < 50000; w++);
```

とありますが、これは、待ち時間を作るための何もしないループです。

ここで使っている vu16 とは "std\_type\_def.h" の中で定義している (コラムを参照) 型です。

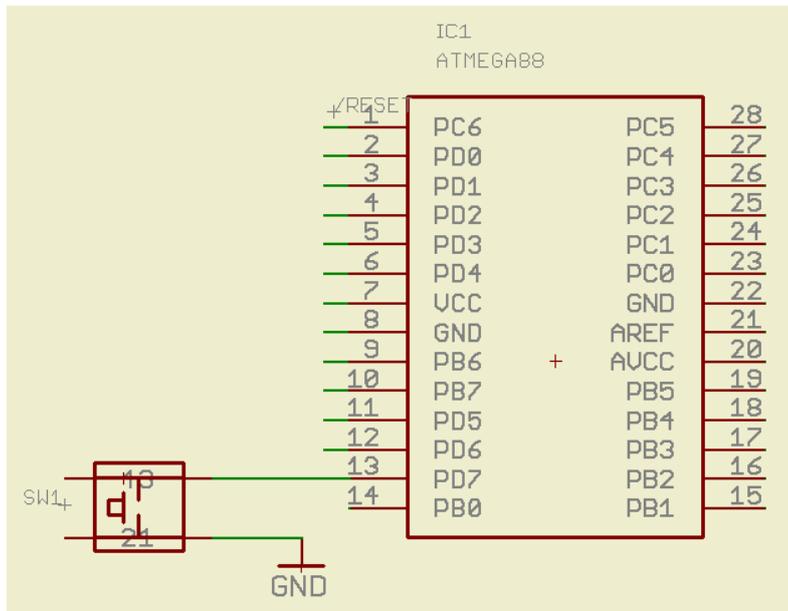
## タクト・スイッチの値を読む (I/O ポートの入力制御方法)

今度は、タクト・スイッチを使って見ましょう。

LED の時は、I/O ポートを出力として使いましたが、今回は入力として使います。

試作 CORE では、LED を PD7 に接続しています。

タクト・スイッチの回路



今回は使用するのは三つのレジスタですが、一つは LED でも使った DDRB レジスタと同じパターンで、PORT D の入出力の方向を決定する DDRD を使います。

レジスタ名	レジスタ名	説明
DDxn	Data Direction Register	入力か出力か決定する。 対応 bit が 1 の時が出力になり 0 の時入力になる。
PORTx	Port X Data Register	入力モードの時に、対応 bit に 1 を書き込むとピンがプルアップされる。
PINx	The Port X Input Pins Address	対応ポートの入力状態が入っているレジスタ (アドレス)

入力で使うのに、出力用の PORT X を使えるところが面白いところです。

対応する PORT を入力にしているのに、PORT X で対応するピンに対して、出力にする事でプルアップすることができます。

LED と同様に、1Bit の処理の為にサブルーチンを作るほどではないのでマクロで定義しました。

```
#define SW_Init() { DDRD &= ~BV( PD7 ); PORTD |= BV( PD7 ); }
#define SW_State() PIN_D & BV( PD7 )
```

## サンプル・プログラム

タクト・スイッチを押していないときは、LED はゆっくり点滅して、押している時は、早く点滅するようなサンプルを書きましたのでご覧ください。

リスト 2

(program/TACT\_SW\_TEST/main.c)

## 電池の電圧を測定する (ADC の利用)

今まで、I/O ポートはデジタルな値を取り扱っていましたが、ADC ポートを使えば電圧を測定することが可能です。ただし、測定できる範囲と分解能があります。

今回の回路では 0V ~ 3.3V の間の電圧を 10bit の解像度で測定できます。

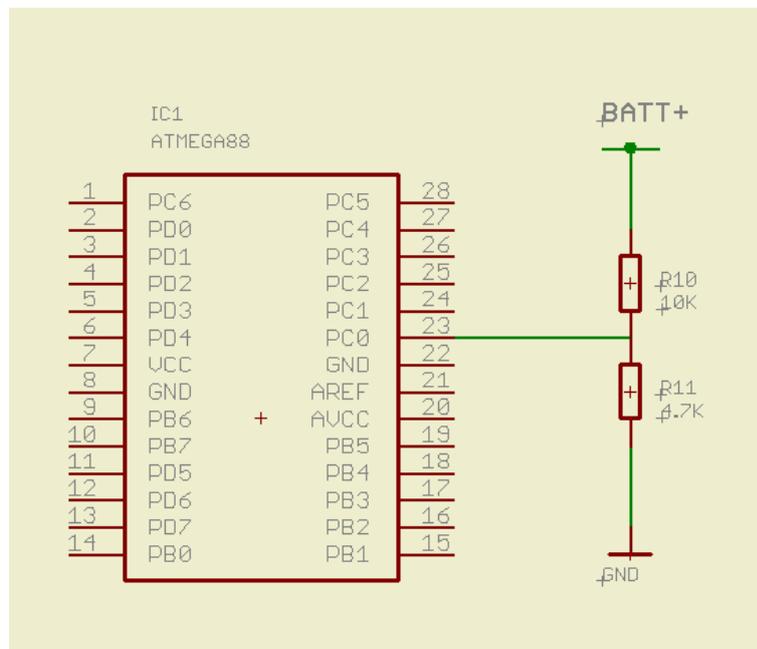
より範囲の広い電圧を測定する場合は、オペアンプなどを使用して別途アナログ回路を作り、その範囲に収まるようにする必要がありますが、今回は電池の電圧だけですので抵抗を 2 個使い、分圧して、それを ADC ポートで読むようにしたいと思います。

さて、今回使用している、AVR は、10bit (1024) の解像度を持っているので、例えば電圧範囲を 0V~3.3V とすると、それを 1024 で割った 0.003222656V 測定することができます。たとえば、値が 125 を示していたら 0.4028V となります。

今回は、電池の電圧範囲は 4.2 です。そのまま、ADC ポートに接続すると測定不能領域に入ってしまう。

そのため、今回は抵抗を 2 本使って分圧として測定するようにしたいと思います。

回路 分圧



今回は、測定範囲が 3.3V までで、入力される電圧の上限を 10V だとすると。  $3.3V/10V$  で 0.33 倍になるように抵抗値を決めればよいことになります。

抵抗器は無限に色々な抵抗値があるわけではなく、一般的に売られている抵抗器の中で、0.33 倍になるように選びます。

今回は、10K と 4.7K を選び、

R1 を 10K R2 を 4.7K にして 0.319 倍になるようにしました。

ADC を使うためには、I/O よりも少し多くのレジスタの設定が必要になります。

レジスタ名	レジスタ名	説明
ADMUX	ADC Multiplexer Selection Register	入力チャンネルの選択
ADCSRA	ADC Control and Status Register A	ADC の状態レジスタ A
ADCSRB	ADC Control and Status Register B	ADC の状態レジスタ B
ADC (ADCH/ADCL)	The ADC Data Register	データレジスタ

まず、ATmega88/168 には、五つの ADC チャンネルがあるので、どのチャンネルで AD の入力とするかのチャンネルを選択する必要があります。それを行うのが、ADMUX レジスタです。

そして、ADC の開始やサンプリング速度などを決定し、さらに現在の状態を見る事ができるのが ADCRSA と ADCRSB レジスタです。

最後に、変換された値が入っているのが ADC レジスタです。マイコンの内部では ADCH/ADCL と二つのレジスタを合わせて 16bit の値として ADC として値を読み出すことが可能です。

### サンプル・プログラム

入力電圧が 4V 以下になったときに、LED を点灯するようにしたプログラムをご覧ください。

#### リスト 3

(program/ADC\_TEST/main.c)

このサンプルで、ADC\_Get( u08 ch ) の中で書いている ADCRSA レジスタへの操作です。

まず、ADC 変換を始めるために ADEN と ADSC の二つの bit を 1 にする事で ADC を開始します。そして、同じレジスタの ADSC が 0 になるまで待ち、0 になったら、ADC の値を読むという手順が必要です。

## センサの値を読み込む

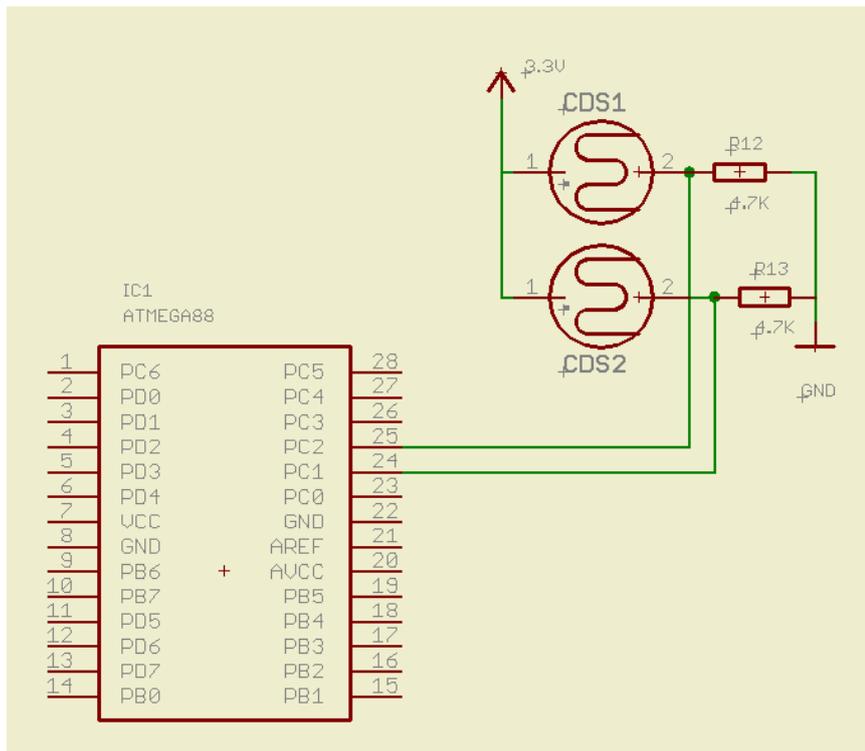
ロボットとして、外界の状態を把握し、それにより動きを変えたいくなります。

今回は、シンプルに CdS を使い明るさの変化を取得したいと思います。

回路は、電池の電圧を測定している分圧とほぼ同じ方法で値を取得しています。回路的に ADC1 と ADC2 のチャンネルにセンサを接続しています。

プログラムも、同じ方法で値を取得しています。

回路 CdS 回路



CdS も電池の電圧を測定するのと同様、分圧しています。どの程度の分圧にするかの抵抗値を決定する必要がありますが、今回は実験した結果、4.7K $\Omega$ 程度が適当ではないかと思います。

測定値になりますが、自然光の下で約 2V 程度ですので数値的には 620 程度になります。しかし、この値は測定する環境により変化します。

そのため、センサで取得した明るさを絶対値として、使用すると思ったとおり動かない事があります。そこで、プログラムがスタートしたら、ある時間、明るさを測定し、その値を保持して、それを基準に比較するようにしてみました。

## サンプル・プログラム

プログラムは、明るさに応じて LED の点滅速度を変えてみるようにしました。

プログラム

/program/CDS\_TEST/main.c

## ブザーの制御

ブザーは圧電素子（ピエゾ素子）を使った物を使用します。（秋月のP-1251）

この素子に、適当な信号を与えることで音を出すことができます。音とは空気の振動なわけですが、どの程度の振動を作るかと言う事になります。音というと、音階を思い浮かべますが、（コラム「音階について」を参考）例えば、「ラ」の440Hzの音を出すことを考えて見ましょう。「ラ」の音を出すためには1秒間に、440回圧電素子を、On/Offすれば、OKです。

1秒を440で割れば0.00227272秒ごとにOn/Offにするわけですが、正確に時間を測定する必要が出てきます。今回の回路には、9.2Mhzのクロックが接続してあるので、そのクロックをカウントすれば時間を測定することができます。このような処理にうってつけなのは、タイマ機能です。

ATmega88/168には三つのタイマが付いています。このタイマは後ほど説明しますPWMで制御するサーボ・モータにも使用しています。

マイコンを使用して何かを作るときに、限られた機能をどのように割り振るかを考えるところが一つ面白いところでもあります。音を出す部分ではTimer2を使うことにしました。

さて、タイマは色々種類のタイマがあり、設定するためのレジスタも増え8個のレジスタを使います。そのうちの三つはカウンタの本体と、比較するための値を入れているレジスタで、設定するために使用するのは五つとなります。

レジスタ名	レジスタ名	説明
TCNTn	Timer/Counter Register	カウントアップされている値が入っているレジスタ
OCRnA	Output Compare Register A	カウンタと値を比較するための値を入れているレジスタ
OCRnB	Output Compare Register B	カウンタと値を比較するための値をいれているレジスタ
TCCRnA	Timer/Counter2 Control Register A	カウンタの設定を行うためのレジスタ A
TCCRnB	Timer/Counter2 Control Register B	カウンタの設定を行うためのレジスタ B
ASSR	Timer/Counter2 Asynchronous Status Register	同期制御を行うためのレジスタ
TIFR2	Timer/Counter2 Interrupt Flag Register	割り込みフラグをコツレジスタ
GTCCR	General Timer/Counter Control Register	同期してカウンタのリセットなどを行うためのレジスタ

## カウンタの作動原理

タイマには多くの機能がありますが、その中で一番単純な作動パターンを説明します。

まず、カウンタは、マイコンのハードウェアで実装されていて、プログラムが作動中に自動的にカウントアップする TCNT レジスタを持っています。

そして、この TCNT レジスタは Timer 2 の場合 8bit の幅があるので、0 からカウントアップしていき 255 まで進むと、また、0 に戻るという性質があります。(設定で 0-255-0 というようにすることも可能、また Timer1 は 16bit で Timer2 は 8bit である)

カウントアップするタイミングは、マイコンのクロックと同じ速度か、それとも、クロックが 8 アップして一つ上げるか、更にクロックが 3 2 回で一つだけ上げるかという具合にクロックと連動しています。

このように、クロック n 回でカウンタを一つ上げるような設定をプリスケールと呼び、TCCR2B の中で指定できるようになっています。

自動的にカウントするだけではたいした事ができませんが、たとえば、255 から 0 になる時に割り込みを発生させたり、I/O を操作する事が可能です。

さらに、比較レジスタ (OCRn) に値を入れて、カウンタが同じ値になったら、割り込みを発生させたり、I/O を操作する事が可能になります。

比較レジスタにちょうど良い値を入れれば、0.1 秒おきに On/Off したりすることが可能になります。たとえば、今回のように音を出すような場合、1 秒間に 440 回 I/O を On/Off したいと思うような場合はうってつけです。タイマを使うと設定だけしてしまえば、あとはプログラムは別の処理をしても勝手にマイコンのハードウェアが自動的に I/O を On/Off してくれます。

この自動的に、I/O を On/Off するという機能を使うと音を出すことが容易に実現可能です。また、オクターブを上げるには、それを倍にすればよいわけです。そこで、注目したのが、Timer2 のプリスケールの表です。

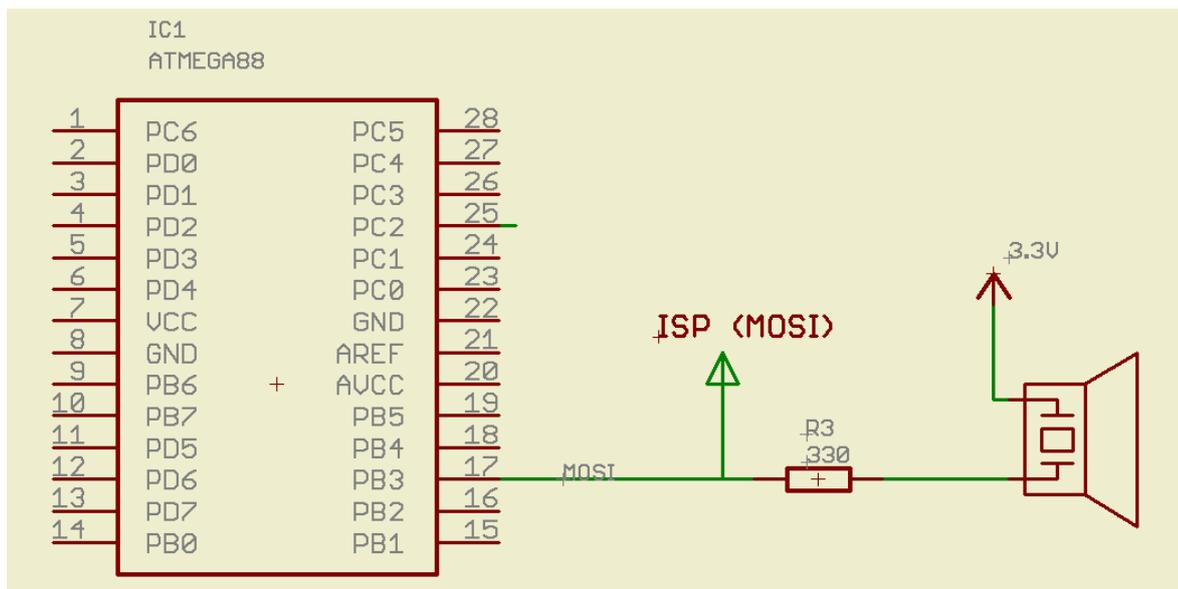
CS22	CS21	CS20	意味
0	0	0	タイマ停止
0	0	1	プリスケールなし
0	1	0	8 分周
0	1	1	3 2 分周
1	0	0	6 4 分周
1	0	1	1 2 8 分周
1	1	0	2 5 6 分周
1	1	1	1 0 2 4 分周

この中に、プリスケールが、3 2 ・ 6 4 ・ 1 2 8 ・ 2 5 6 分周の部分が、2 倍ずつ増えていく事がわかります。これを使えば、オクターブ上や下の音を出す時はプリスケールの値を変化させるだけで可能になります。

3 2, 6 4, 1 2 8, 2 5 6 ですので 4 オクターブの 4 オクターブ分の音域をカバーできるルーチンが作れます。

早速回路を書いてみました。タイマ 2 の出力は OCA2 の PB3 ピンに出ますが、この PB3 はプログラムの書き込みの MOSI ピンと同じピンですので、このまま圧電素子を接続するとプログラムが書き込めなくなりますので、抵抗を 1 本入れてみました。

## 回路 ブザー



この状態でプログラムを書き込むと、圧電素子から音が出てきて、プログラムが書き込まれたこともわかるのでちょっと便利な副作用です。

### サンプル・プログラム

タクト・スイッチを押すと、音階が上がっていくプログラムを作ってみましたのでご覧ください

### リスト 3

(program/SOUND\_TEST/main.c)



## シンプルなシリアル通信

コマンド・サーボを制御する前に、シンプルにシリアル通信を行うテスト・プログラムを紹介しましょう。

今回シリアル通信関連も色々な設定がありますので、使用するレジスタは5個ほどあります。基本的には、通信速度の設定、通信する bit 数やストップビットなどの指定となります。

レジスタ名	レジスタ名	説明
UDRn	USART I/O Data Register n	送受信を行うためのデータを入れるレジスタです。
UCSRnA	USART Control and Status Register n A	通信中の状態などを表すレジスタです
UCSRnB	USART Control and Status Register n B	通信の割り込みなどの設定を行う
UCSRnC	USART Control and Status Register n C	パリティなどの設定を行うためのレジスタ
UBRRnL/H	USART Baud Rate Registers	通信速度 BPS を決めるためのレジスタ

割り込みを使わないシンプルな方法で、サンプル・プログラムを書きましたので、参考にしてみてください。

### サンプル

(program/UART\_01\_TEST/main.c)

このサンプルでは、通信速度 115.2K で 8 ビット、ストップビット 1、ノンパリティで通信して、アルファベットの小文字で 'a' ~ 'z' までを表示を繰り返すプログラムです。

また、もしも、入力がされたら、その文字 + 1 にしたものを表示し続けます。

ポイントは送信する場合では、まず送ることができるか、UCSR0A レジスタの UDRE0 をチェックする必要があります。また、受信する場合は、データが有るか確認してから受信します。割り込みを使わない場合は、非常に簡単に実装することができます。

## コマンド・サーボの制御

コマンド式のサーボは、今までの PWM の時間的幅で回転角度指定を指示する方法とは違い、ホスト（今回はマイコン）とサーボの間でシリアル通信をしてデータを送る方式になっています。

サーボ・モータには一つひとつ ID 番号が付いているので、一つの信号線に複数のサーボを接続する事ができます。

また、PWM サーボでは角度を指定するわけではなく、角度から PWM の幅を計算して、そのパルスが発生させる必要がありましたが、コマンド方式のサーボでは角度指定するだけです。処理が簡単になります。さらに、回転させるための時間も指定できるので制御が楽です。

データを送るための電氣的規格には色々な種類がありますが、コマンド式のサーボでは色々な方法がありますが、その前に、半二重と全二重について説明しておきます。

## 半二重と全二重

通信には送信と受信を行うわけですが、同時に送信と受信が行うことができるのが全二重、同時に行えないのが半二重です。

電車の線路で言えば、複線は全二重で、単線は半二重です。また、電話は全二重ですが、糸電話は半二重です。全二重を行うためには、送信と受信、線が 2 本必要になりますが、半二重の場合は 1 本しか必要ありません。

コマンド・サーボ方式では配線量を減らすために、半二重を採用しているものが殆どです。

## 電氣的な接続は？

さて、送信と受信を行うために配線を行いますが、その配線された線の中をどのような電気信号が流れるか、また、送るタイミングの合わせ方をどうするのかなども、ホスト側とサーボ側で合わせておく必要があります。

取り扱う電圧の幅、時間的なタイミング、プロトコルのような細かな点を合わせておかなければいけません。

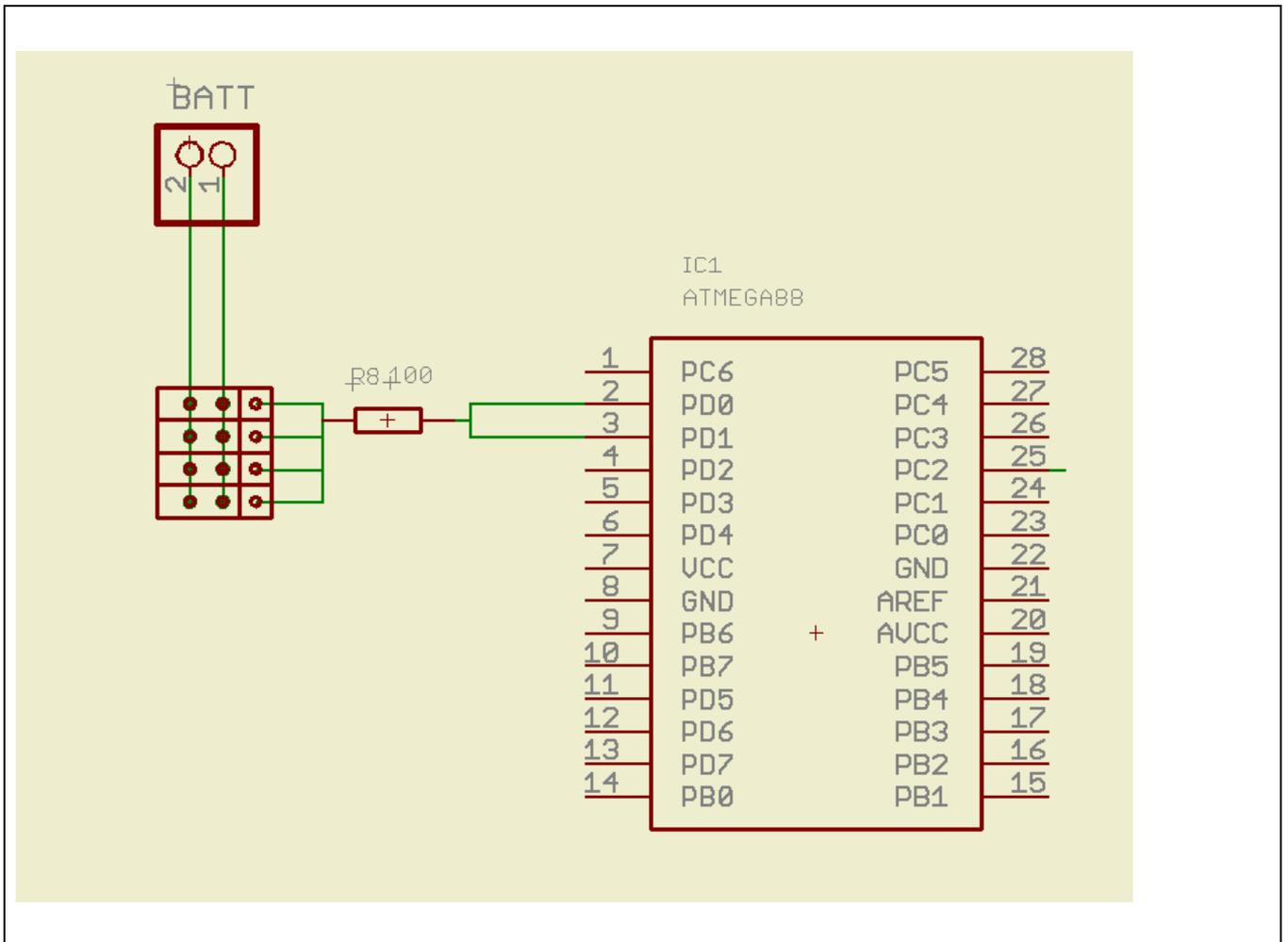
コマンド式のサーボではよく採用されているのが、RS485 という規格です。

RS485 はノイズに強く、距離も伸ばせると言われる差動方式というものを使っています。そのための IC もあります。

また、RS485 のように、何メートルも先ではなくて良いので、特別なことをしないでデータを送る場合はマイコンから出ている信号をそのまま送る方法もあり、TTL 方式と表記されている場合もあります。

## 回路図

今回使用する、TTL 方式のサーボ・モータですので、UART の TX と RX を直接信号線に接続します。  
念のため信号線には保護用の抵抗を入れておきました。



またサーボは四つほど接続できるように考えてみました。

## コマンド・サーボ RS304 用のプログラムを書いてみる

早速ですが、手元にある RS304 のテスト・プログラムを書いてみました。  
なお、RS304 の資料はメーカーのサイトの下記の URL にデータがあります。

[http://www.rc.futaba.co.jp/hobby/04\\_robot/robot3.html](http://www.rc.futaba.co.jp/hobby/04_robot/robot3.html)

さて、テスト・プログラムでは、まず動くことを確認するためと言う意味で書いたプログラムが下記の通りです。具体的な作動内容はプログラムのソースコードを見ていただくとして、簡単に説明しておきましょう。

ホスト側からは、決められたデータを送信することでコマンドを送ります。

サーボは初期設定の通信速度は下記のようになっていますので、プログラムもそれに合わせて設定しておきます。

### 通信速度

項目	値
ビット/秒	115.2kbps
データビット	8bit
パリティ	なし
ストップビット	1 bit
フロー制御	なし

双葉のサーボの場合、サーボの中にある設定がメモリマップのようになっており、それに対しての書き込みを行うような方式を取っています。

### メモリマップ

サーボもどのメモリに値を書くのかや、何処のメモリを読むのかなどというようなコマンドが、パケットとなって送られています。

メモリマップには大きくわけで、Read Only の場所と、Read/Write できる場所と分かれています。

アドレス	領域
0 ~ 3	変更不可領域
4 ~ 29	ROM 領域 (書き込みオプションで書き換え可能)
30 ~ 59	RAM 領域

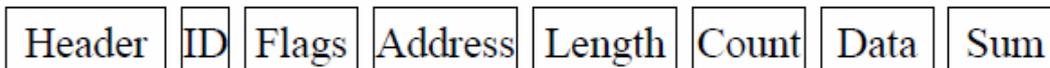
さらに詳しいメモリマップはサーボ・モータの説明書を参考にしてください。

## パケット

コマンドのパケットにはショート、ロング、リターンの3種類ありますがここでは、よく使うショートパケットの説明をします。

ショートパケットには八つの項目が次のような構成になっています。

### パケット構成



それぞれの構成は次のようになっています。

パケットの要素	説明
Header	0xFAAF の 2Byte から構成されています
ID	対象とするサーボの ID 番号を指定します。 番号は 1 ~ 127 となり、ID が 255 の場合は全サーボへの共通コマンドとなります
Flag	Bit 単位での意味があります。 書き込むのか、読むのかなどの指定をおこないます 詳しくはマニュアルを参考にしてください
Address	アクセスするメモリマップのアドレス
Length	データブロックの長さを指定
Count	ブロックの数、ショートパケットでは 1 に固定
Data	書き込むデータ長
Sum	ID~Data の各 1 バイトを XOR した値となります。

この書式でパケットを送ることでサーボ・モータに命令を送ります。

目的位置の指定は、RAM 領域の 0x1E と 0x1F のアドレスに書き込みます。具体的には次のようになります。

ID が 1 のサーボを 90.0 度に回転させる。

90 度は、16 進数で 0x384 になりますので、パケットは次のようになります。

パケット	内容	説明
Header	0xFA, 0xAF	固定
ID	1	サーボの番号
Flag	0	RAM 領域で特別に
Address	0x1E	目標位置のメモリマップアドレス
Len	2	2 Byte を書き込む

Count	1	シヨートの時は 1
Data	0x84, 0x03	データ (90 度は 900 となり、Hex で表すと 0x384 となり、送るときは下位 byte の 0x84 から送り、次に上位バイトの 0x03 を送ります。
Sum	0x9B	sum を計算した結果

これらの値をサーボに送ることでサーボは回転してくれます。

そして、実際に実装しているのがサンプル・プログラムの中の SV\_RS303\_AngleSet というサブルーチンです。

また、サーボ・モータのトルクを、On/Off するための SV\_RS303\_OnOff も書きましたのでご覧ください。

```
¥Program¥SERVO (CMD)_01_TEST¥main. c
```

### ちょっとしたコツ

今回使用する、RS304MD は半二重の通信ですので、送信と受信を同じラインで TTL 方式です。

回路的には、受信 (RX) の PD0 と送信 (TX) の PD1 が接続されています。そのため、送信したデータを自分自身でも受信することになってしまいます。

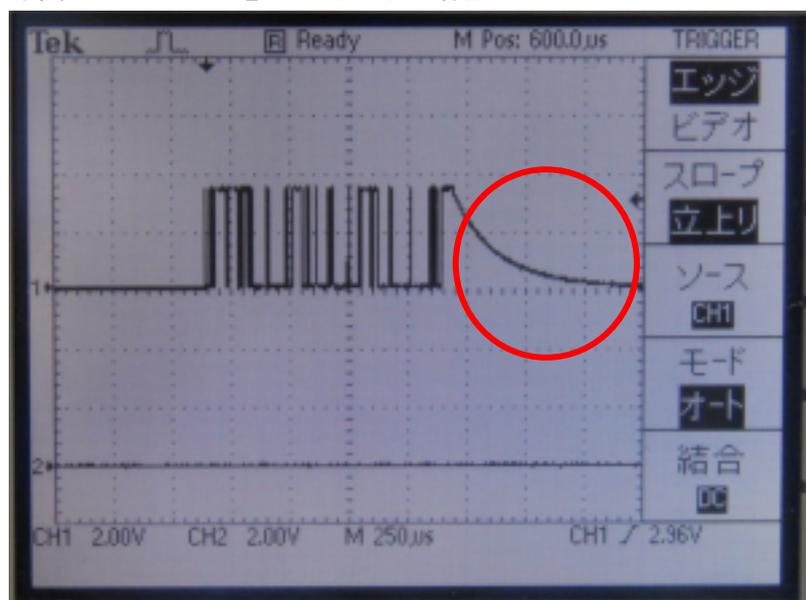
ソフトウェア的に自分が送ったデータを無視するようにするプログラムを作るのも一つの方法ですが、無視するプログラム自体も無駄とも言えます。

そこで、受信をするタイミングを自分でデータを送っている間は無視する方法を、マイコンの設定を変更してしまい、受信をしないようにしてしまう方法を使っています。

また、単に送受信のラインの片側を有効・無効にするだけではなく、初期値で TX 側をプルアップしておくことで、通信線を安定化させてたのが次のプログラムです。

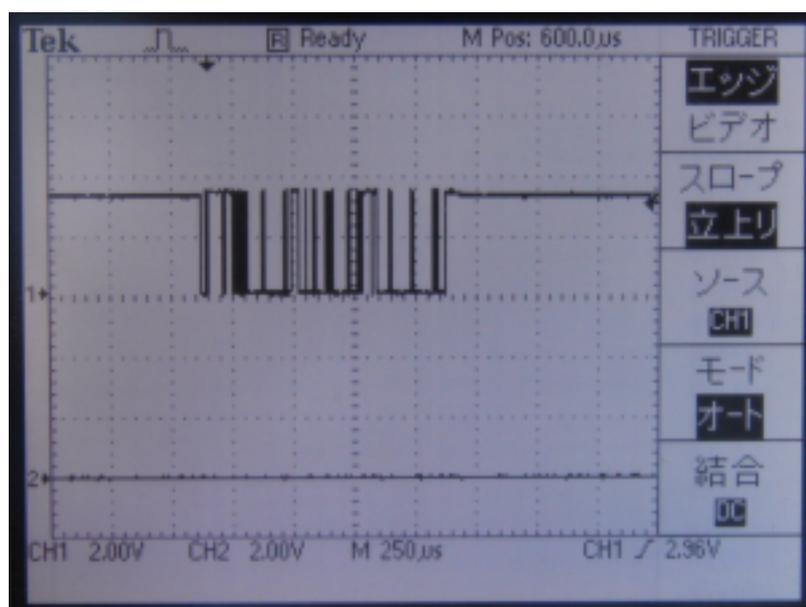
```
¥Program¥SERVO (CMD)_02_TEST¥main. c
```

写真 プルアップを入れていない場合



このように、プルアップを入れていない場合は、信号線がGND側に落ちていく様子がわかります。この写真ではGND側に落ちていますが、場合に電圧がふらつく事になり、誤作動の原因となります。

写真 プルアップを入れている場合



このように、電子回路自体を変更しなくてもソフトでプルアップができるので便利です。

### 汎用性を向上させる

さて、サーボ関連のコマンドを整理し、サーボの内部メモリに対しての読み書きを行うルーチンを作りました。

具体的にはサブルーチンの `SV_U08_Get(u08 id, u08 adr, u08 *data)` をご覧ください。違いは、今までパケットのFlagを今までは0ばかりでしたが、データを読み込む場合は0x0fを指定して、データを送り、送り終わった後、データを読むと言うものです。

ただし、このルーチンはタイムアウト処理を入れていませんので、何らかの問題でデータが送られてこなかった場合はループを抜けなくなってしまいますので注意が必要です。

```
¥Program¥SERVO (CMD)_03_TEST¥main.c
```

サンプル3では、サーボの角度を読みながら、100度以上になったら、LEDを点灯するようにしています。