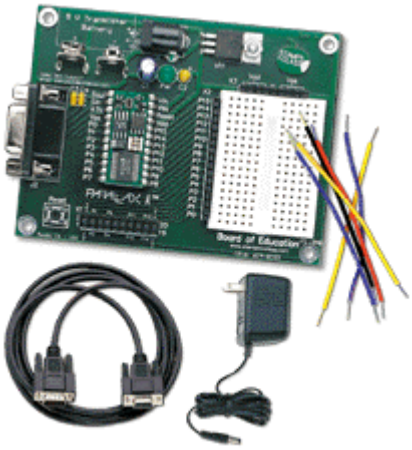
基本的な英語が分かる人なら誰でもできる

P-BASIC 言語

Version 2



Ｐ－ベーシック言語教育用キット

ボード オブ エデュケイション

(Board Of Education)

日本マイクロボット教育社

基本的な英語が分かる人なら誰でもできる
P-BASIC 言語 v2

責任の有限と保証の否認（Limits of Liability and Disclaimer of Warranty）
この本は、読者への啓蒙と教育のために意図されたものです。十分に正確を期したつもりですが、著者及び翻訳者は、エラー、脱落、いかなるアプリケイションの内容の適合性などについての責任は持てませんのでご了承ください。また、本書の情報の使用及び誤使用結果によるいかなる損害も責任は持てません。なお、本書の情報を使って装置の使用、製造又は販売等に関してそれが特許、コピーの権利又はその他の権利などを侵害していないかどうかの判断は、読者に責任があります。

著作権及び登録商標

PBASIC（ピーベーシック）はトレードマークであると共に、Parallax, Inc.のロゴでもあります。そして、BASIC Stamp は Parallax, Inc.の登録商標です。
PIC は Microchip Technology, Inc.の登録商標です。
その他の銘柄や製品名はそれぞれ持ち主の商標又は登録商標です。

基本的な英語が分かる人なら誰でもできる

P-BASIC 言語

Version 2

日本マイクロボット教育社

も　く　じ

第1章　電気回路の基本

1. 電圧･電流･抵抗の基本的関係

電圧(V)[Voltage]･電流(I)[Current]･抵抗(R)[Resistor]･電力(P)[Power]の間には次のような計算式が成り立ちます。

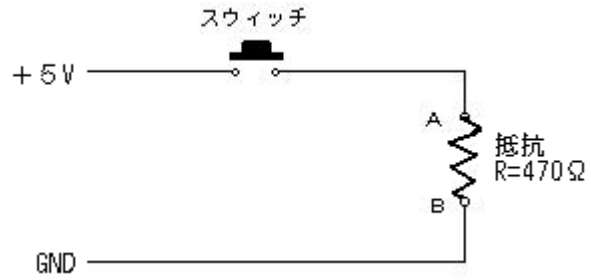
V (ボルト Volt) = I ・ R　　単位：μV、 mV、 V、 KV
I (アンペアー Ampere) = V / R　　単位：μA、 mA、 A
R (オーム Ohm) = V / I　　単位：mΩ、 Ω、 KΩ、 MΩ
P (ワット Watt) = I ・ I ・ R　= I ・ V　単位：mW、 W、 KW、 MW

電流　1 A　=　1000 mA　(ミリアンペアー)
　　1 A の 1/ 1000　= 1 mA (ミリアンペアー)
　　1mA の 1 / 1000　= 1 μA (マイクロアンペアー)

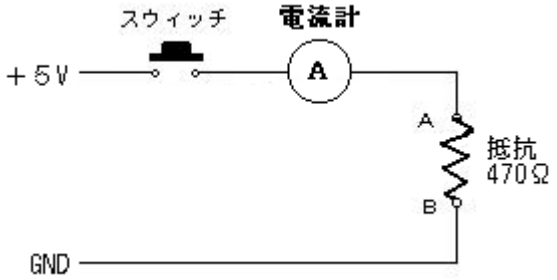
電圧　1 V の1000倍 (1 x 1000) = 1 KV (キロボルト)
　　1 V　= 1000 mV (ミリボルト)
　　1 V の 1 / 1000　= 1 mV (ミリボルト)
　　1 mV の 1/ 1000 = 1 μV (マイクロボルト)

抵抗　1Ωの1000倍 (1 x 1000) = 1 KΩ(キロオーム)
　　1 KΩの1000倍 (1000 x 1000) = 1 MΩ(メグオーム)

問1：次の回路のＡＢ間に 470Ωの抵抗をつないだ時の電流を計算で出して下さい。



実験1：上の回路に 0.1A 以上を測れる電流計（テスター等）を接続して、実際の電流を測定して、計算した値と比べてください。



考察1．問1と実験1で出した電流値が違う事があるかどうかを考えてみよう。

2. Short Circuit (ショートサーキット：短絡)

電気で言う『抵抗』とは「電気の通りにくさ」をオーム（Ω）という単位で表したものです。従って、電気の量（電流）を制限しなければならないとき、この抵抗をつないで電流を制限します。

例えば、次ページの 図 2 のような回路でスウィッチを入れたらどうなるかを考えてみましょう。

警告：電線や電池等が過熱して大変危険ですから**絶対に実験はしないで下さい。**

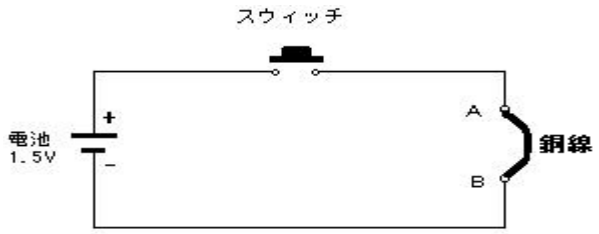


図 2

この回路の電線や銅線、そしてスウィッチ等の接触抵抗（接続点やスウィッチ等が接触する事により生ずる抵抗値）などをゼロと見れば、

抵抗値 R　　0 （Ω
電流 I ＝ V ／ R ＝ 1.5 ／ 0 ＝＞ ∞ （A）
電力 P ＝ V * I ＝ 1.5 * ∞ ＝＞ ∞ （W）

無限大の電流と言う事は現実には起こりえない事ですが、大電流には違いありません。 また、「無限大の電力」とは「無限大の熱」とも言えますから、これまた現実には難しい事です。上の警告にありますように細い電線に大電流を流すと高い熱を発生しますし、電池がすぐ無くなってしまいます。従って、このような事が起こらないようにしなければいけません。

このような回路になる事を「短絡」（Short Circuit ： ショート）と言って、家屋やビルなどの電気配線でこの短絡が起きると、ヒューズがとんで停電になったり、ひどい時はこの短絡によって起きた火花や熱などによって、まれに火災にまでなる事があります。また、もしこの短絡がコンピューターやテレビ、ラジオなどの内部で起きると、ヒューズがとんだり電源の回路が駄目になってしまう事があります。特に、電気の回路をチェックする時などネジ回しなどの金属で間違えてショートさせてしまうことが良くあります。十分に気をつけましょう。

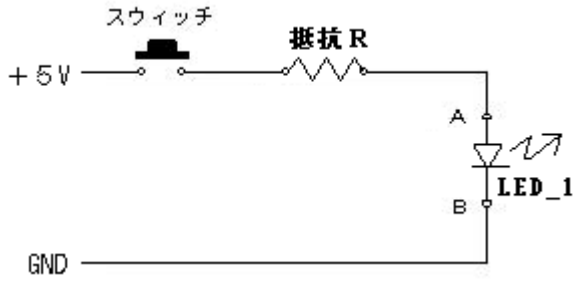
3. LED (Light Emitting Diode) (エル・イー・ディー)

ダイオード (Diode) は、アノード (プラス) とカソード (マイナス) があり、アノードからカソードの方向しか電流を通しません。LED も普通のダイオードと同じようにアノードからカソードにしか電流を通しません。電流を流した時に発光するように作った物を Light Emitting Diode (ライト・エミッティング・ダイオード)、頭文字をとって LED と呼びます。

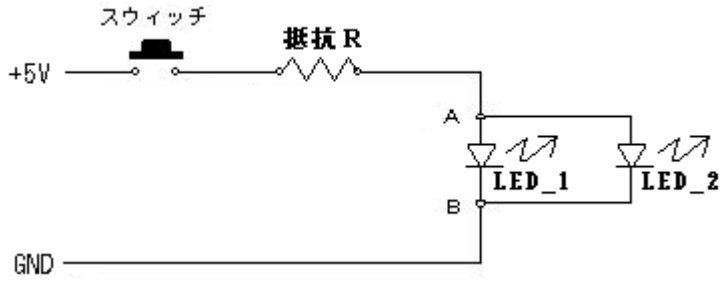
LED は、用途に合わせて赤、青、黄色など、いろいろな物が市場に出回っています。ここで使用する赤の LED は流せる最大電流が 25mA 位でそれ以上の電流を流すと焼け切れてしまいます。通常の使用は 15mA 以下で十分です。

[注意]:かなり光が強いので直接正面から LED を見てはいけません。

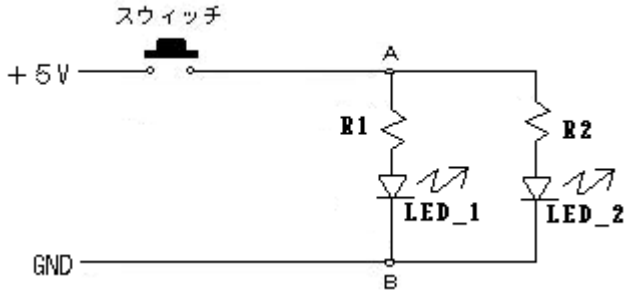
問3:次の回路の抵抗Rの値を決めてください。但し、LED の電流を約 10mA とします。



問4:次の回路の抵抗Rの値を決めてください。



問5：次の回路の抵抗R1とR2の値をそれぞれ出してください。

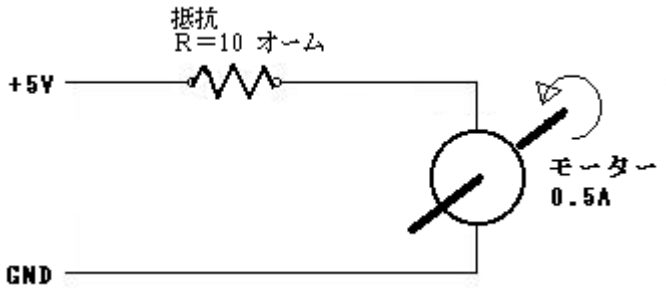


豆知識

抵抗値と電力：

電気の回路の中に使われる抵抗は、電力（消費電力）について考慮しなければなりません。流れる電流によって、1/4W、 1/2W、 1W、 5W というように設計の段階で計算をして決めなければいけません。

例えば、下図の回路では；



P = I･I･R = 0.5 x 0.5 x 10
= 2.5 (W)

従って、この抵抗の消費電力は 2.5W ですから 3W 以上、余裕を持って 5W 位の抵抗を使用する必要があります。

抵抗(Resistor)のカラーコード：

普通の抵抗には、その抵抗値を示すカラーコードの帯がついています。

このロボット製作教本で使用する抵抗は一般によく使われる 1/4 Watt（ワット）のカーボンフィルム（炭素から作られる）で±5%の誤差があるものです。その抵抗を手にとってよく見ると、色の帯の最後が金色（Gold）になっているのに気が付くでしょう。そのゴールドの帯は第 4 番目の帯で、その抵抗の誤差（Gold は±5%）を表しています。

抵抗のカラーコードは、抵抗値を表す工業規格です。各カラーの帯は数字を表していて、カラー帯の順番はその数の値を示しています。始めの 2 本のカラー帯は数字をあらわしていて、3 本目のカラー帯は乗数を表しています（或は、単に 10 なら 0 をひとつ、１００なら００を、１０００なら０００と数字に 0 を付ける、と考えても良いです）。そして、4 番目のカラー帯はそれぞれ±5、±１0、±２0%の誤差を表しています。

色 (Color)	1 番目の数字 (1st Digit)	2 番目の数字 (2nd Digit)	乗数 (Multiplier)	誤差 (Tolerance)
黒(Black)	0	0	1	
茶(Brown)	1	1	10	
赤(Red)	2	2	100	
橙(Orange)	3	3	1000	
黄(Yellow)	4	4	10000	
緑(Green)	5	5	100000	
青(Blue)	6	6	1000000	
紫(Purple)	7	7	10000000	
灰(Gray)	8	8	100000000	
白(White)	9	9	1000000000	
金(Gold)				5%
銀(Silver)				10%
無色(no color)				20%

[例]

470Ωのカラーコード

1 番目のカラー帯　　黄(Yellow)

2 番目のカラー帯　　紫(Purple)

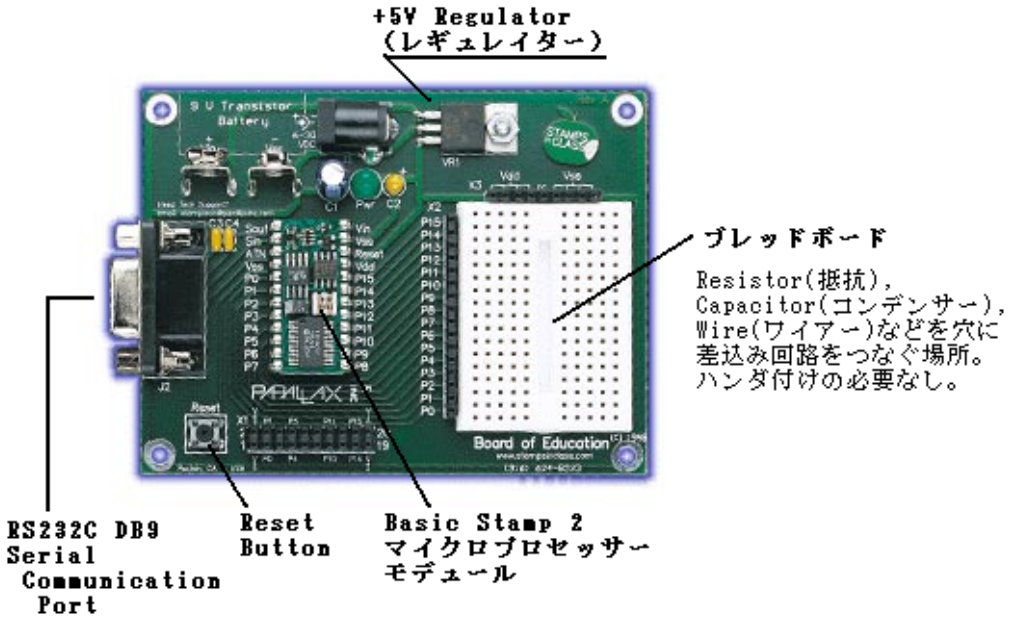
3 番目のカラー帯　　茶(Brown)

4 番目のカラー帯　　金(Gold)

黄＝4、紫＝7、茶＝10、金＝±5%　従って、470Ω ±5

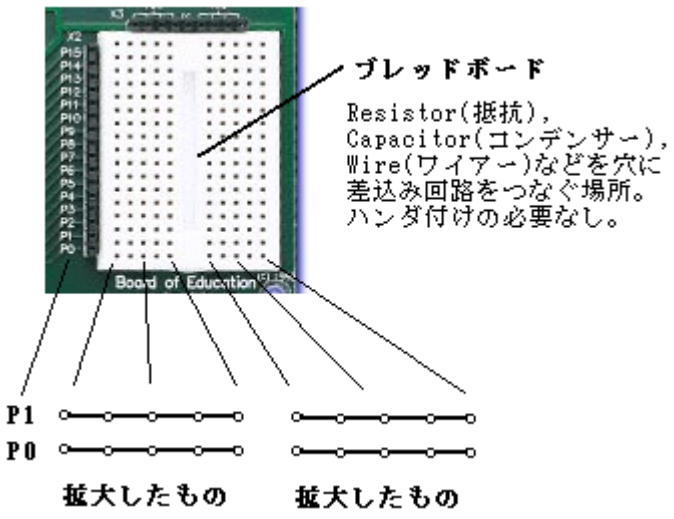
第2章　BOEプリント基板とその使用方法

(Board Of Education)

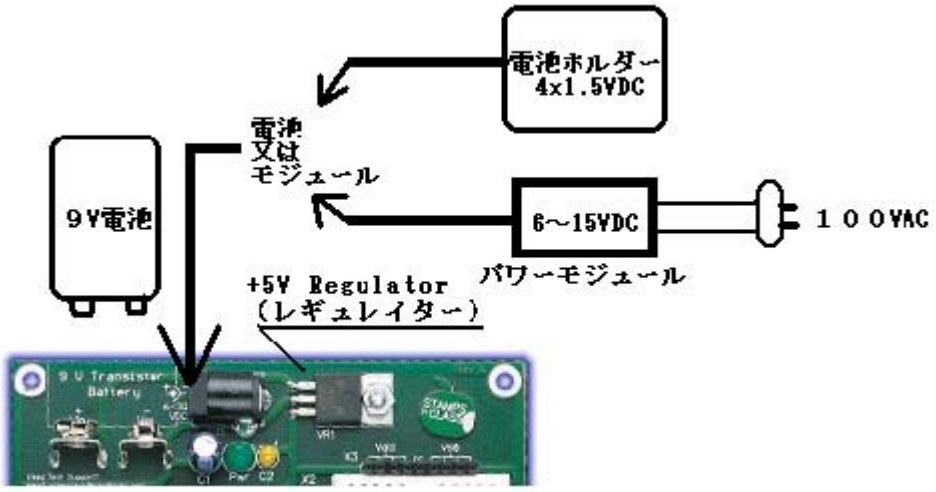


1. ブレッドボード(Bread Board)

右の図の穴に部品を差込み回路を組み立てる。差込み穴を拡大した図に示したように、左半分または右半分の横1列の5つの穴は、このブレッドボードの裏側で各々がつながっている。 P0からP15は、ブレッドボードとはつながっていないので必要に応じてWireでつながなければならない。



2. 電源 (Power Supply) について

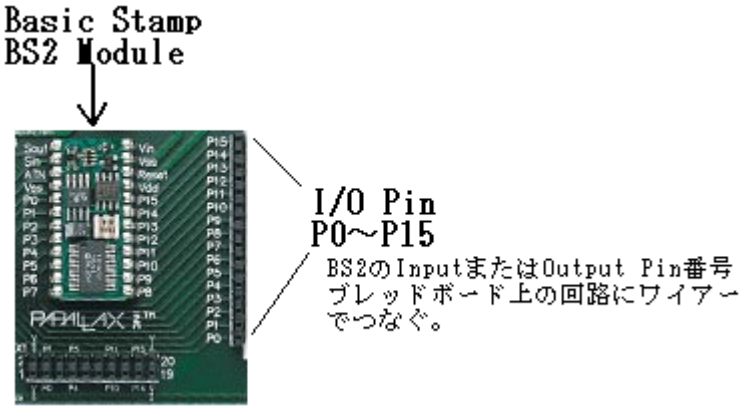


BOE(Board Of Education) への電源供給には、次の3通りの方法がある。

1. 9V の電池
2. 1．5V の電池を4本 (ボーボット用)
3. 6～15VDC のパワーモジュール (750mA 以上を推薦)

上記の中から一つを選び電源を供給する。

3. ベーシック･スタンプ BS2 モデュール



BS2 の各ピンの名称及び説明は次ページを参照してください.

BS2 － 各ピンとその名称

ピン番号	名 称	記 述	注 釈
1	TX	シリアルアウト	Transmit Data (DB9 Serial Cable)
2	RX	シリアルインプット	Receive Data (DB9 Serial Cable)
3	ATN	Active-high reset	ハンドシェイク(DB9 Serial Cable)
4	GND	Serial Ground	シリアルケーブルのグラウンド
5	P0	I/O pin 0	インプット/アウトプットの各ピンはソースで 20mA,シンクで 25mA まで使用できるが P0から P7の間、又は P8から P15までの間での合計使用可能電流はソースで 40mA、シンクで 50mA が最大であるので、これを超えない事が大事である。 これらの I/O ピンは原則的に TTL のレベルで使用するが、RS232 レベルの信号を受ける時は、22K Ωの抵抗を直列につながないと入力回路が燃える恐れがある。 TTL と RS232 レベル信号についての項を参照。
6	P1	I/O pin 1	
7	P2	I/O pin 2	
8	P3	I/O pin 3	
9	P4	I/O pin 4	
10	P5	I/O pin 5	
11	P6	I/O pin 6	
12	P7	I/O pin 7	
13	P8	I/O pin 8	
14	P9	I/O pin 9	
15	P10	I/O pin 10	
16	P11	I/O pin 11	
17	P12	I/O pin 12	
18	P13	I/O pin 13	
19	P14	I/O pin 14	
20	P15	I/O pin 15	
21	＋5V	+5V 電源(注1)	5V 電源入力又は出力
22	$\overline{\text{RES}}$	Active-low reset	このピンを low(ゼロ)にすると、プログラムのリセットができる。
23	GND	System Ground	システム全体のアース･グラウンド
24	PWR	Power Input(注2)	レギュレイトされた電源インプット

(注1)：直接5VDC（±10%）を供給する時に使用する。ただし、ピン24PWR にパワーをつないだ時は、5V を供給してはならない。

(注2)：ピン24に接続する電源は6～15VDC で 750mA 以上の電流を供給できる事が望ましい。

豆知識

Bit（ビット）：P0 や P5 など単体でのデータ(ビット 0、ビット 5 などともいう)。

Byte（バイト）：P0 から P7、または P8 から P15 までの 8 Bits を 1Byte いう。

ASCII code（アスキーコード）：1 Byte で表したアメリカで規格化されたコード。

Binary Number（2 進法）：0 と 1 だけの数字を使って表示する方法。

Hex Number（1 6 進法）：0 から 9 の数字と A,B,C,D,E,F の 6 文字を使う表示方法。

例：[ENTER]（改行）を 1 バイトデータで表すと；

Bit	B7	B6	B5	B4	B3	B2	B1	B0
Binary	0	0	0	0	1	1	0	1
Hex	0				D			
ASCII character	CR (Carriage Return)							

4. RS-232C Serial Port (シリアルポート)

この RS232C コネクターはコンピューターなど外部とのコミュニケイションに使う。

① TTL レベル信号と RS232C レベル信号について

TTL レベル信号は、low（ロー）のときは、0V（正しくは約 0.3V）で hi（ハイ）の時は、5V（約 4.6V）の電圧レベルで信号を作ります。
これに対し RS232C レベルの信号はおよそ－12Vから＋12Vの電圧レベルで信号を作ります。ほとんどのシリアルプリンターやモデムなどはこの RS232C 信号を使用しています。
この様に、TTL と RS232C は使用する電圧レベルが違うので TTL input に RS232C の信号を直接、接続してはいけません。

② Threshold （スレッシュホールド）

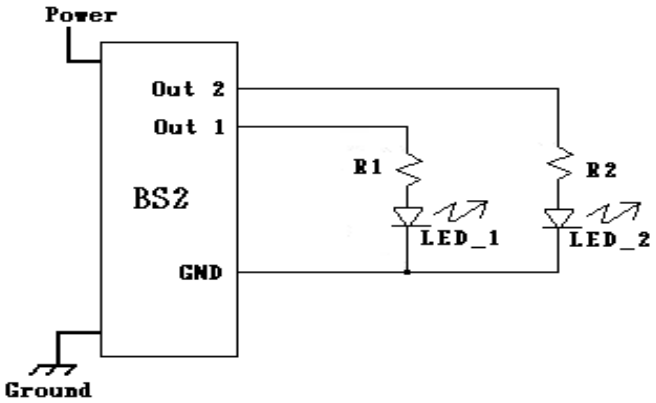
ディジタルの信号は、low か hi、あるいは 0（ゼロ）か 1（イチ）しかなく、アナログ信号のようにその半分とか、その中間等のような信号はありません。しかし、low と hi の電圧をどこかで分けなければいけません。low と hi の境目のところを、Threshold （スレッシュホールド）と言って、この BOE では、1.4V ないし 1.5V をスレッシュホールドにしています。従って、1.4V（ないし 1.5V）以下の信号は全て TTL、 RS232C の信号に関らず、low または0（ゼロ）、1.4V（ないし 1.5V）以上の信号は hi または1（イチ）になります。（注：スレッシュホールドが 1.4V か、又は 1.5V かは BS2 等の内部抵抗、接続抵抗等によって変わります。）

豆知識

Output を電流の供給源とする回路を「ソース回路 (Source Circuit)」という。
従って、下図の「ソース回路」では、Out 1 を hi にすれば LED_1 がオンになり、
Out 2 を hi にすれば LED_2 がオンになる。ここで、Out 1 と Out 2 を 2bits
のデータと考えれば、LED_1 と LED_2 は次の 4 通りの組み合わせ点滅がある。

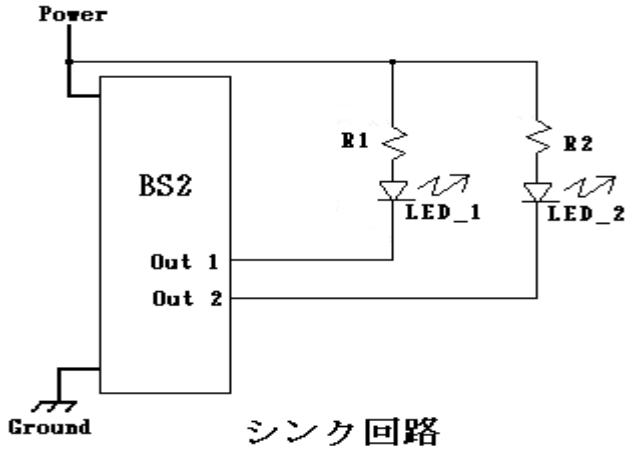
[NOTE] : hi = 1 > 1.4V , lo = 0 < 1.4V

Out 1	Out 2	LED_1	LED_2
0	0	OFF	OFF
0	1	OFF	ON
1	0	ON	OFF
1	1	ON	ON



ソース回路

前ページのソース回路に対して、電流の引き込みに Output を使用する回路を「シンク回路 (Sink Circuit)」という。



シンク回路

Out 1 と Out 2 をソース回路と同様に 2bits のデータと考えると、LED は次のような 4 通りの組み合わせ点滅になる。

Out 1	Out 2	LED_1	LED_2
0	0	ON	ON
0	1	ON	OFF
1	0	OFF	ON
1	1	OFF	OFF

[Reminder] (リマインダー:忘れないで!)

Board Of Education (BOE) の P0～P15 までの各々の Output Pin はソースで 20mA、 シンクで 25mA までで、P0～P7 (Low Byte) 又は P8～P15 (High Byte) の合計が、ソース 40mA、 シンク 50mA である。

第 3 章　プログラム用のエディター (Editor)

【 PBASIC Editor 】

1. Basic Stamp Editor

ベーシック･スタンプのプログラムを作成する為にウインドウズ版とドース版 (Windows and DOS version)をフロッピーディスク『ロボット製作』に収めてあります。

2. Windows version

この Windows version の名称は [stampw_v1_096] です。使用するコンピューターにコピーを取ると良いでしょう。この Windows version は BASIC Stamp 2 は勿論の事、BASIC Stamp 2sx と BASIC Stamp 2e のプログラム作成にも使えますので便利です。BASIC Stamp Windows は殆んど直観的に使用出来るように設計されています。

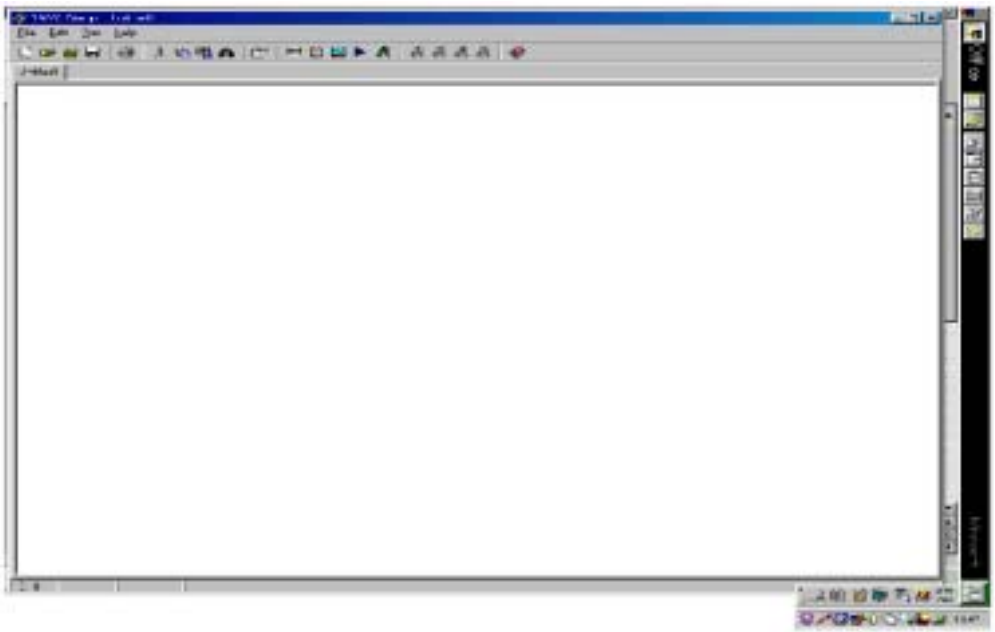
① C Drive (Hard Disc) に新しいフォルダー名(eg. Basic Stamp)を作ります。
② A Drive に『ロボット製作』のフロッピーを挿入します。
③ A Drive を開けて、[stampw_v1_096]を C Drive の新しいフォルダー名にコピー又は、ドラッグ アンド ドロップ します。
④ ショートカットアイコンを作成しておけば便利でしょう。
⑤ フロッピーディスクを安全な場所に保管しておきます。

3. Editor Window

ひとつのエディターウインドウから成り立っていて 16 種類までのソースコードファイルを変更したり見たりする事が出来ます。 エディターに載せた各ソースコードは各々のファイル名のタブでいつでも呼び出す事が出来ます。 新しいソースコードで未だ保存(セーブ)されてないファイルは "untitled#" (# は番号)と付けられます。
実際に開いているソースコードのページの状態は画面の下にあるステータスバーに表示されます。そのステータスバーにはカーサーの位置、ファイル保存、ダウンロード状態、エラーメッセージ等が含まれています。Editor window にソースコードを入力した

後、ファイルメニューの Run をクリックして Run (或は Ctrl-R)をクリックするとコードが BASIC Stamp 2 にダウンロードされプログラムがスタートします。(当然ですが BASIC Stamp 2 がシリアルケーブルで繋がれて電源が与えられている必要があります)

Cドライブにコピーをとった [stampw_v1_096] のアイコンをダブルクリックで開くとファイルメニューとツールバーがある、新しいエディターページが現れます。



ファイルメニューとツールバーを拡大した図

ファイルメニューとツールバーの一つずつを説明します。
（見やすくする為に大きく書いてあります）

① 『ファイルメニュー』

1。 File

- 1) New ： 新しいファイルを開く
- 2) Open ： 既存のファイルを開く
- 3) Save ： ファイルを保存(既にファイル名がある場合)
- 4) Save as ： ファイルに名前を付けて保存
- 5) Close ： ファイルを閉じる
- 6) Print ： プリントする

2。Edit

- 1) Undo ： やり直す
- 2) Cut ： 切り取る
- 3) Copy ： コピー
- 4) Paste ： 貼り付け
- 5) Select All ： 全て選ぶ
- 6) Find/Replace ： 検索/置換
- 7) Find Next ： 次の検索
- 8) Preferences... ： 選択 (下の Preferences の項を参照)

3。Run

- 1) Run (Ctrl+R) ： プログラムのダウンロード及び開始
- 2) Check Syntax (Ctrl+T) ： ソースコードのチェック(インストラクション に誤りがあった場合、エラーメッセージが 出る)
- 3) Memory Map (Ctrl+M) ： EEPROM、 RAM 等の使用状況を見る
- 4) Debug... > New： Debug 1、Debug 2、Debug 3、Debug 4 が ある
- 5) Identify ： Com. Port に接続した Basic Stamp の ID チェック

4。Help

1) About ： Parallax Inc.　Basic Stamp Editor Version 1.096　Copyright 2000

【 Preferences 】

(1) Editor Appearance	： タブの設定、ツールバーの表示 / 非表示選択
(2) Editor Operation	： Stamp Mode ： BS2/BS2E/BS2SX の選択
(3) Debug Appearance	： カラーとサイズの表示
(4) Debug Function	： ASCII コントロールの指定
(5) Debug Port	： Com Port、 Baud Rate 等の設定

② DEBUG WINDOW：

- この Debug Window はデータを送ったり受け取ったりするのに、ちょうどターミナルのような役をします。データの送信用と受信用を別々に大きさを調整出来る枠を用意してあります。
- Debug window は一度に別々のポートから Debug が出来るように 4 つの Debug window を用意してあります。(Basic Stamp の Network 用に使われます。)
- Debug ターミナルをプログラムのダウンロード後だけでなく、いつでも開く事が出来ます。そして、ユーザーはそれを閉じる事なく Editor Window にスイッチバックする事が出来ます。
- データの送受信の枠内でカラーとフォントサイズを指定出来ます。
- たとえデータの受信中でもポート、ボードレイト、パリテイー、データビット、フローコントロール等を設定出来ます。これは、データの受信で適切な設定が分からないような時、それを見 付けるのに役立つものです。
- Port status indicators (ポート状態の表示)は TX (Transmit)、RX (Receive)、DSR (Data Send Ready)、CTS (Clear To Send) 等の Serial Port のピン状態を示します。そして、マウスボタンクリックで DTR と RTS の状態を設定出来ます。

- 受信枠のバッファーサイズは 8192 行のラインまでスクロールバック出来ます。また、ポーズボタン(Pause Button)はデータが続けて入って来た時、一時的に止める為のものです。なお、スクリーン上ではデータが止まってもバッファーには保存されます。スクリーンは Resume (続く) をクリックするまで停止しています。
- Macro Key は Debug Terminal から外部に送信するテキスト/データを含むファイルを定義します。最高 26 までの Macro Key がひとつのマクロキーファイルの中で定義されます。
- 特別制御用文字セット(Special Control Character set) は次の種類があります。 クリアースクリーン(Clear Screen)、ホーム(Home)、ベル(Bell)、タブ(Tab)、キャリジリターン(Carriage Return)- 上の文字セットに増強したセット(Enhanced control characters)は次の通りです。

  MoveTo (x,y)、 Cursor Left、 Right、 Up and Down、 Backspace、 Line Feed、 Clear Right、 Clear Down
- なお、各々の control characters は Preference settings の中の Debug Function で無効にする事が出来ます。

③. DOS version

[コンピューターの DOS による起動とその使い方]

Windows version を使わずに DOS version で BS2 エディターを使う方法について述べます。 BS2 は勿論、BS1 のプログラミングのとき使うと良いでしょう。

1. コンピューターの電源をオフの状態のままで、付属のフロッピーディスク「ロボット製作教本」をドライブ A に入れ、コンピューターの電源を入れ、立ち上げます。

2. スクリーン上に次のようなメッセージと「A プロンプト」が表示されます。

Starting Windows 98 ….

Microsoft (R) Windows 98

(C) Copyright Microsoft Corp 1981 – 1999.

A:＼＞_　(A プロンプトと読む)

3. A プロンプトでコンピューターA ドライブの使用が可能になります。
これを、「DOS で立ち上げた」とも言います。

代表的な DOS Command を見てみましょう.

(1) DIR (ディレクトリー・Windows のフォルダーと同じようなもの)
(2) COPY *file.ext* *newfile.ext* (新しいファイル名でコピーをとる)
(3) TYPE *file.ext* (ファイルをスクリーン上に表示する。編集は不可)
(4) DEL *file.ext* (ファイルを削除する)
(5) CD *directory* (Change Directory 別のディレクトリーに行く場合)
(6) CD. . (1つ前のディレクトリーに戻る)
(7) RD *directory* (Remove Directory ディレクトリー名の削除)
(8) MKDIR *directory* (Make Directory 新しいディレクトリーを作る)
(9) C: (C Drive C ドライブに移る)
(10) A: (A Drive A ドライブに移る)

DOS コマンドの練習をしてみましょう(*斜体* はキーボードでタイプする)。
なお、タイプする文字は大文字でも小文字でも構いません.

1) A プロンプトで *DIR* とタイプして<ENTER> をうちます。

```
A:＼＞dir          <Enter>
        Volume in drive A is JMAcademy
        Volume Serial Number is ･･･14EF-3A0C
        Directory of A:＼
        ROBOT                  < DIR >   xx-xx-xx   xx:xx
        SERVO                  < DIR >   xx-xx-xx   xx:xx
        COMMAND    COM     118,174
        BS2TEST                < DIR >   xx-xx-xx   xx:xx
        STAMP2     EXE      15,123
        STAMPW~1   EXE     716,288
              3 file(s)          849,585 bytes
              3 dir(s)          211,456 bytes free
A:＼＞_
```

この様に、DIR(ectory) ディレクトリーはフロッピーディスク A の
内容をリストアップしてくれます。

2) 別のディレクトリーに行く為の CD コマンド

```
A:＼＞cd bs2test   <Enter>
A:＼＞BS2TEST＞_
```

CD は Change Directory の意味です。

ここで、BS2TEST のディレクトリーを DIR/W (横広がり) で見てみます。

```
A:＼＞BS2TEST＞_
A:＼＞BS2TEST＞dir/w    <Enter>
        Volume in drive A is JMAcademy
        Volume Serial Number is ･･･14EF-3A0C
        Directory of A:＼
        [.]  [..]  TEST_1.BS2  TEST_2.BS2  TEST_3.BS2
        TEST_4.BS2  TEST_5.BS2  ･･･････････････
        ･･･････････････････････････
             11 file(s)          7,597 bytes
             2 dir(s)         211,456 bytes free
A:＼＞BS2TEST＞_
```

3) ファイルをコピーする時は COPY コマンドを使う。

A:＼＞BS2TEST＞COPY ファイル名. 拡張子 新ファイル名. 拡張子

```
A:＼＞BS2TEST＞copy  test_1.bs2  ex_1.bs2  <Enter>
                    1 file(s) copied
A:＼＞BS2TEST＞_
```

コピーをとる前は、合計で 11 ファイルあったのでコピー後は 12 ファイルあるはずです。

DIR で調べてください。 2) で表示された BS2TEST ディレクトリーのリストにコピーした ex_1.bs2 のファイルが含まれている事を確認してください。

ex_1.bs2 は test_1.bs2 のコピーですから内容は当然同じものです。

つぎの TYPE Command で内容を見てみましょう。

4) ファイルの内容をスクリーンに表示する TYPE コマンド

```
A:＼＞BS2TEST＞type ex_1.bs2   <Enter>
‘REM test_1 Potentiometer with sound
pot     var        word
        loop:      high 14
                   pause 1
                   rctime 14, 1, pot
                   debug ? pot
        goto loop
A:＼＞BS2TEST＞_
```

このプログラムは、可変抵抗 (Potentiometer 又は Variable Resistor) の値をコンピューターのスクリーン上に表示させる PBASIC プログラムです。

5) DEL(ete) Command (削除)を使って ex_1.bs2 を削除します。

```
A:＼＞BS2TEST＞del ex_1.bs2   <Enter>
A:＼＞BS2TEST＞_
```

Directory で調べてください。合計 11 ファイルになっているはずです.

【注】 : この Del コマンドで気をつけなければいけないことは、本当に削除したいファイルだけを削除しないと、警告も何も表示されなくてそのファイルは無くなってしまう事です。

上記5つのコマンドはとても役に立つものですので練習してください。

豆知識

DOS COMMAND のいろいろ(コマンドの後に<Enter>を入れる事)

	コマンドの例	説明
CD	A:＼＞CD *directory 名*	Change Directory 別のディレクトリーに行く場合
CD. .		1つ前のディレクトリーに戻る
CD＼		1番初めのディレクトリーに戻る(ルートディレクトリーと言う)
MKDIR	A:＼＞MKDIR *directory 名*	新しいディレクトリーを作る
RD	A:＼＞rd *directory 名*	Remove Directory(ディレクトリーを削除する)
REN	A:＼＞ren *old.ext new.ext*	REName ファイル名を変える時に使う
COPY	A:＼＞BS2TEST＞ *copy test_1.bs2 ex_1.bs2*	名前を変えてコピーをとる
DIR	A:＼＞dir	ディレクトリーをリストアップする
DIR/W	A:＼＞dir/w	ディレクトリーを横広がりでリストアップする
TYPE	A:＼＞type *file.ext*	*file* の内容をスクリーン上に表示する
DEL	A:＼＞del *file.ext*	*file.ext* を削除する
C:	A:＼＞c:	C Drive に変える
A:	C:＼＞a:	A Drive に変える

④ DOS version によるプログラミング

Aプロンプトでコンピュータードライブの使用が可能になったら、Directoryを調べて下さい。

STAMP2.EXEというファイルがこれから使うBS2 Editorです。このエディターで新規のファイルを作る場合は、A プロンプトで *STAMP2* とタイプをすれば新しいファイルが開きますので、そこにプログラムを書いていきます。 既存のファイルを開いて編集をしたり或はソースコード等をダウンロードしたい場合は、STAMP2.EXEがある同じディレクトリーにそのファイルがなければいけません。 別の言い方をすれば、開きたいファイルがあるディレクトリーに STAMP2 . EXE をコピーして、そのディレクトリーから *STAMP2* とタイプしてエディターを開いても良いでしょう。
(反対に、STAMP2.EXEがあるディレクトリーにファイルをコピーしても、結構です)

Stamp 2 のエディターが開くとスクリーンがブルーで、A プロンプト(A:＼＞_)が表示されます。このスクリーンが開いている間、何時でもF1のキーを押すとファイルの開き方、ダウンロードの仕方、ファイルの保存等々のファンクションキーの説明が出ます。

注意:新しく書いたプログラムは、閉じる前に必ず名前をつけて保存するのを忘れないで下さい。

⑤ PBASIC プログラムの書き方

＊PBASIC とは一般の BASIC 言語にパララックス社（Parallax）が特別な命令言語を加えたのでこのように呼びます。

1） 一般の BASIC 言語は、各々のインストラクション(命令)ごとにラインナンバーを書きますが、PBASIC では必要に応じて「ラベル名」を書きます。

(例）一般の BASIC 言語

```
10      FOR     C = 1 to 256      ‘ FOR…NEXT 命令言語
20      PRINT   C                 ‘ C の値をスクリーンに表示
30      NEXT                      ‘ C が 256 になるまで繰り返す
40      GOTO 10                   ‘ ライン番号１０に行け
```

(例） PBASIC 言語

```
loop_1:                           ‘ ラベル名
        for       b2 = 1 to 256   ‘ for….next 命令言語
        debug ?   b2              ‘ b2 の値をスクリーンに表示
        next                      ‘ b2 が 256 になるまで繰り返す
goto loop_1                       ‘ ラベル名 loop_1 に行け
```

2） プログラムの書き方は、命令言語（インストラクション）の全てを決められた英語で書かなければなりませんが、プログラムの名前、日時、製作者名、その他必要なメッセージなどは、‘ （アポストロフィ；Apostrophe）の後にアルファベットで書かなければいけません。

特に要所要所のインストラクションの後にはこのメッセージを入れる習慣をつけておくと、後日、自分のプログラムを見直したときなどにとても理解しやすくなります。(案外、後日見なおした時に、何のためにこのインストラクションを入れたのだろう？と思う事があります)。

3） ラベル名、インストラクション、メッセージ等は上の例でも分かるように、揃えると見易くなります。

（例）

```
‘プログラム名：テスト__1　平成　　年　　月　　日作成

　ラベル名1：（タブ）　（タブ）　（タブ）　　‘ メッセージ1
　（タブ）　（タブ）　命令_1　（タブ）　　‘ メッセージ2
　（タブ）　（タブ）　命令_2　（タブ）　　‘ メッセージ3
　　　　・　　・　　・　　・
　ラベル名2：（タブ）　（タブ）　（タブ）　　‘ メッセージ5
　 goto　ラベル名1　　（タブ）　（タブ）　　‘ メッセージ6
```

4） プログラムを書いたら、ファイル名をつけて保存（Save）するのを忘れないようにして下さい。 保存を忘れてエディターから出てしまうと警告も出ないで終わってしまいます。 もちろん、その場合は折角のファイルもなくなってしまいますので十分注意してください。
その時、ファイルの名前は分かり易い好きなファイル名を付ければ良いですが、その拡張子は必ず「. BS2」としなければいけません。

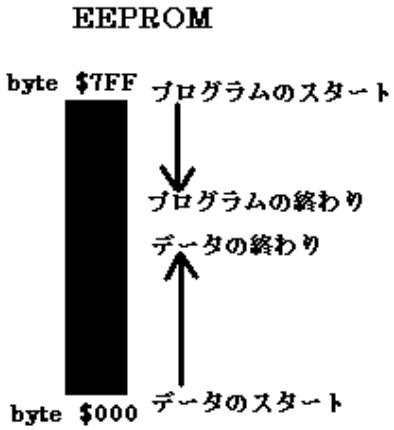
（例）

Test_1.bs2
EXPERIMENT_1.BS2

第4章 ベーシック･スタンプ 2（BS 2）について

[1] プログラムとデータメモリー

ベーシック･スタンプ 2 (以下 BS 2 と呼ぶ) は 2K バイトの EEPROM を使用し、プログラムとそれに伴うデータをそこに書き込みます。また、プログラムの為の I/O ピンの設定、臨時のデータ書き込み用に 32 バイトの RAM が用意されています。この RAM は words(ワード)、bytes(バイト)、nibbles(ニブル)、あるいは bits(ビット)等として使用できます。臨時に書き込んだ RAM データはプログラムを新しくスタートさせる毎に、全てゼロにクリアーされます。従って BS 2 は、EEPROM に書き込んだプログラム及びデータは、たとえ電源が無くなったとしても消える事はありませんが、RAM に書き込んだ臨時のデータは電源を入れた時、リセットボタンを押した時そして、コンピューターからプログラムをダウンロードした時には、全てゼロになってしまいます。



2K バイトの EEPROM は次のように使用方法が決められています。

[注]：byte $000, $7FF の $ サインは Hex number で表されている意味です。

EEPROM は Electrical Erasable Programmable Read Only Memory の頭文字をとったもので、電気的にデータを書いたり消したり出来、書き込んだプログラムやデータなどは電源がなくても、電気的に消す指令を出すまで保持しています。

上図のようにプログラムは、EEPROM の1番上のアドレス$7FF(10 進法で 2047)からア

ドレスの下方向に順番に書き、データはアドレス $000 から上方向に順番に書いていきます。データの量が少なければプログラム用のスペースが増えますが、データ量が多すぎるとプログラムスペースが少なくなり、書けなくなってしまいますので注意が必要です。

［2］ 32 バイトの RAM スペースとワードについて

Word は全て 2 バイト(16 ビット)使用可能

Word	種　　類	Read/Write(読／書)
$0	各ピンのインプット状態を読む	Read-only
$1	各ピンを出力状態にラッチする	Read/Write
$2	各ピンの入･出力の方向を決める	Read/Write
$3	一般の使用可能ワード領域	Read/Write
$4	一般の使用可能ワード領域	Read/Write
$5	一般の使用可能ワード領域	Read/Write
$6	一般の使用可能ワード領域	Read/Write
$7	一般の使用可能ワード領域	Read/Write
$8	一般の使用可能ワード領域	Read/Write
$9	一般の使用可能ワード領域	Read/Write
$A	一般の使用可能ワード領域	Read/Write
$B	一般の使用可能ワード領域	Read/Write
$C	一般の使用可能ワード領域	Read/Write
$D	一般の使用可能ワード領域	Read/Write
$E	一般の使用可能ワード領域	Read/Write
$F	一般の使用可能ワード領域	Read/Write

＊［ 初期の BASIC STAMP 1 で使われた b2 から b11 までの Byte Name はそのまま使えます。］(次頁及び BS2 プログラム例と解説を参照して下さい)

BASIC STAMP 1 (BS1) の RAM スペースとワードについて

初期のベーシック･スタンプ1 (BS1) はインプット或はアウトプット (I/O) を決めたり、変数を保管する為に 16 バイトの RAM が用意されています。BS1 のプログラムで RAM を使うときは、この表の中からワードかバイト名を選んで使います。
BS2 のプログラミングでも使えますが、BS1 と BS2 の RAM 名の混乱に気を付けてください。

ワード名	バイト名	ビット名	特記
Port	Pins	Pin0～Pin7	I/O ピン;各ビットを指定できる
	Dirs	Dir0～Dir7	I/O ピンの方向を各ビットで指定できる
W0	B0	Bit0～Bit7	各ビットを指定できる
	B1	Bit8～Bit15	各ビットを指定できる
W1	B2		
	B3		
W2	B4		
	B5		
W3	B6		
	B7		
W4	B8		
	B9		
W5	B10		
	B11		
W6	B12		GOSUB インストラクションで使われる
	B13		GOSUB インストラクションで使われる

上の表から、変数 (variable) 或は I/O ピンをバイト (Pins, Dirs, B0～B13) として使っても良いですし、16 ビットワード (Port, W0～W6) として使うことも出来ます。
また、初めの２つのデータバイト（Pin0～Pin7, Dir0～Dir7, Bit0～Bit15）は各々個別にビット単位で使うことが出来ます。ひとつのビット、例えば、ステータスフラッグ（status flag）を保管したい時など使うと便利です。
＊Reserved Word は巻末に表で示しましたので参照して下さい。

(1) Word $0

このワードは、16 の I/O ピン全部又はその1部を読むためのものです。
Word $0 は次のようなシンボリック名 (Symbolic Name) で読む事ができます。

```
INS     'INS の 16 ビット全部を読む時

        INL     'INS の hi バイト(ビット0～7までの8ビット)を読む
        INH     'INS の hi バイト(ビット8～15までの8ビット)を読む

        INA     'INL の low nibble(ビット0～3までの4ビット)を読む
        INB     'INL の hi nibble(ビット4～7までの4ビット)を読む
        INC     'INH の low nibble(ビット8～11までの4ビット)を読む
        IND     'INH の hi nibble(ビット12～15までの4ビット)を読む

        IN0     'INS の Bit 0 (I/O pin の P0)
        IN1     'INS の Bit 1 (I/O pin の P1)
                |
                |
        IN14    'INS の Bit 14 (I/O pin の P14)
        IN15    'INS の Bit 15 (I/O pin の P15)
```

(2) Word $1

Word $1 は 16 の I/O ピン全てをアウトプットに固定します (Output Latches)。
もし、あるピンがたとえ入力モードになっていても、入力信号は無視されますが出力モードに指定した場合は、信号をその出力ピンにセットします。

シンボリック名 (Symbolic Name) は次の通りです。

```
        OUTS            'すべてのピン 16 ビットを出力モードにする
```

```
OUTL        'OUTS の low byte (B0～B7 まで)
OUTH        'OUTS の high byte (B8～B15 まで)

OUTA        'OUTL の low nibble (B0～B3 まで)
OUTB        'OUTL の high nibble (B4～B7 まで)
OUTC        'OUTH の low nibble (B8～B11 まで)
OUTD        'OUTH の high nibble (B12～B15 まで)

OUT0        'OUTS の Bit 0 (I/O pin の P0)
OUT1        'OUTS の Bit 1 (I/O pin の P1)
```

```
OUT15       'OUTS の Bit 15 (I/O pin の P15)
```

(3) Word $2

Word $2 は 16 I/O の全てのピンの入･出力方向を指定します。
あるピンを入力モードにする場合は、該当する Word $2 ビットを0にクリアーしなければいけません。また、あるピンを出力モードにしたければ、該当する Word$2 ビットを1にセットします。Symbolic Name は次の通りです。

```
DIRS        '全てのピン 16 ビットを指定の入･出力にする

DIRL        'DIRS の low byte (B0～B7 まで)
DIRH        'DIRS の high byte (B8～B15 まで)

DIRA        'DIRL の low byte (B0～B3 まで)
DIRB        'DIRL の high byte (B4～B7 まで)
DIRC        'DIRH の low byte (B8～B11 まで)
```

DIRD　　'DIRH の high byte (B12～B15 まで)

DIR0　　'DIRS の Bit 0 (I/O pin の P0)

DIR15　　'DIRS の Bit 15 (I/O pin の P15)

(4) Word $3 から Word $F まで

Word $3 から Word $F までは、予め指定されたシンボル名を持っていない一般的な Word 領域です。 変数声明(Variable statement)等はこれらのメモリーを使います。

［3］ BS2 のコンパイルタイムとランタイム

(Compile-time, Run-time)

BS2用のプログラム PBASIC は、Compile-time と Run-time の2つに大きく分類する事が出来ます。 Compile-time はプログラムが走るためのコード(Executable code)を発生しませんが Alt-M を押す事によって内容を見る事が出来ます。
これに対し Run-time はコードを発生してプログラムを走らせます。

Compile-time の声明には、VAR、 CON、 DATA の3つがあります。
それらを宣言するためには以下のようにします。

1. [Compile-time]

VAR(iable) statement (変数声明)の定義

プログラムの初めに Pre-assign されたシンボル名(Word $0～Word $2)を除く全ての可変シンボル名は、使用する前に各シンボル名の定義をしなければいけません。 この事を Declare(宣言)と言います。

次にサンプルを示します。

```
'Declare the variables (変数の宣言)
cat             var     nib     'cat をニブル(nibble)変数にする
mouse           var     bit     'mouse を 1 ビット(1bit)変数にする
dog             var     byte    'dog を 1 バイト(1byte)変数にする
rhino           var     word    'rhino をワード(word=16bits)変数にする
snake           var     bit(10) 'snake を 10 個のビットアレー(10-piece array)
                                'にする
```

(注: 上記の cat, mouse, dog, rhino, snake 等のシンボル名はどんな名前でも構いません。但し、シンボル名を重複したり Pre-assign された名前等は使用出来ません。)

*これら nib, bit, byte, word 等はプログラムを Compile(機械語に編集)する時、自動的に使用されてない RAM スペースに配置されます。 また Alt-M を押す事によって RAM の内容を見る事が出来ます。 Alt-M を押すとスクリーンに 3 っの I/O words それから全ての words, bytes, nibs, bits 等が順に表示されます。 そして最後に RAM スペースが空いている場合はそれが表示されます。
従って、RAM スペースがどのくらい空いているかを調べたい時等にもこの Alt-M を利用する事が出来ます。

VAR(iable) の使い方には次のような選択もあります。

```
'define unique variables (独自の変数を定義する)

sym1    VAR     bit             'sym1 を 1 ビット 変数(1 bit variable)にする
sym2    VAR     nib             'sym2 をニブル 変数(nibble variable)にする
sym3    VAR     byte            'sym3 をバイト変数(byte variable)にする
sym4    VAR     word            'sym4 をワード変数(word variable)にする
```

bit, nib, byte, word 等の後に( )の中に数字を入れてアレーの大きさ(array size)を宣言する事も出来ます。

```
sym5    VAR     nib(10)         ' sym5 を 10 ニブルアレー(10 nibble array)にする
```

' define variables-within-variables or alias variables (変数の中の変数、 或は別名 'を定義する)

```
sym6    VAR     sym4.highbit    ' variable sym4 の highbit(modifiers を参照)を
                                ' 1 bit 変数にする
sym7    VAR     sym4.lowbit     ' variable sym4 の lowbit(modifiers を参照)を
                                ' 1 bit 変数にする
sym8    VAR     sym2            ' sym2 というシンボル名を sym8 に交替する
```

non-unique variable (bit, nib, byte, word 等の代わりに使う名前) を VAR の中で使用する場合は variable 名を付けた後にピリオドを付けその後に次に示す、モディファイアー(modifiers) を付ける事が出来ます。

Modifiers （モディファイアーズ)

*使用可能な variable modifiers のリスト

```
        lowbyte         ' word の low byte
        highbyte        ' word の high byte
        byte0           ' word の byte0 (low byte)
        byte1           ' word の byte1 (high byte)
        lownib          ' word 又は byte の low nibble
        highnib         ' word 又は byte の high nibble
        nib0            ' word 又は byte の nib0
        nib1            ' word 又は byte の nib1
        nib2            ' word 又は byte の nib2
```

```
nib3            ' word 又は byte の nib3
lowbit          ' word, byte 又は nibble の low bit
highbit         ' word, byte 又は nibble の high bit
bit0            ' word, byte 又は nibble の bit 0
bit1            ' word, byte 又は nibble の bit 1
bit2            ' word, byte 又は nibble の bit 2
bit3            ' word, byte 又は nibble の bit 3
bit4            ' word, byte 又は nibble の bit 4
bit5            ' word, byte 又は nibble の bit 5
bit6            ' word, byte 又は nibble の bit 6
bit7            ' word, byte 又は nibble の bit 7
bit8            ' word, byte 又は nibble の bit 8
bit9            ' word, byte 又は nibble の bit 9
bit10           ' word, byte 又は nibble の bit 10
bit11           ' word, byte 又は nibble の bit 11
bit12           ' word, byte 又は nibble の bit 12
bit13           ' word, byte 又は nibble の bit 13
bit14           ' word, byte 又は nibble の bit 14
bit15           ' word, byte 又は nibble の bit 15
```

[変数の宣言についてのまとめ]

変数を宣言するためには、VAR ステートメントを使う。
VAR で申告されるシンボル名は独自な名前(unique variable)か、変数の中の変数(variable-within-variable)又は、別名(alias-variable)を宣言する。

独自な名前(unique variable)を定義するには次のようにする;

```
symbol   VAR      size(array)
```

ここに:
- symbol は変数の為の独自の名前
- size は WORD, BYTE, NIB, BIT の何れかで指定する
- (array)はアレーサイズを宣言する場合に指定する (オプション)

また、変数の中の変数、又は別名を定義する場合；

```
symbol   VAR      variable.modifiers
```

ここに：　- symbol は変数の為の独自の名前
- variable は定義された変数の名前
- modifiers はオプションで変数の中の変数を定義する為に使う

コンパイラー（Compiler：機械語に編集するもの）は独自の変数の場合にはサイズによって全ての宣言をグループ分けして、RAM の未使用域に書き込みます。
Alt-M のキーを押す事によって RAM の内容を見る事が出来ます。

CON(stant) statement (定数値声明)の定義

CON 声明は VAR 声明にたいへん似ています。相違点は VAR が可変であるのに対して CON はシンボル名に定数を割り当てます。これは1度宣言しておけば何回でもプログラムの中でその数字を使う事が出来るのでとても便利です。

CON の構文は次のようにします；

```
symbol   CON      expression          'シンボル名に数字･式を割り当てる
```

ここに：　- symbol は定数のための独自なシンボル名
- expression は解決できる定数

```
例：     level    CON      10                  'このプログラムの中では level という
                                               'シンボル名は 10 の事である、と宣言
         limit    CON      10*4<<2             'limit は 160
```

［ 10*4<<2 ］の意味
10 x 4 = 40 (10 進数)　この 40 を2進数で表し2回左にシフトする。即ち、
40 (10 進数)　　= 00101000 (2 進数)
2回左にシフト = 10100000 (2 進数)　= 160 (10 進数)

CON の後の数字･式 (expression)は binary operators (バイナリー･オペレイター) を使って表す事が出来ます。(Binary Operators の項を参照してください)

2. DATA statement (データ声明)の定義

EEPROM のメモリー領域でプログラムに使われていない領域はデータを保存する為に使われます。 前に述べましたようにプログラムは EEPROM の最後の住所($7FF)から最初の住所($000)に向かって書き込まれていきます。 ("プログラムとデータメモリー" 参照)この書き込みは自動的に行なわれます。 これに対し、データはメモリーの最初の住所から終りの住所に向かって書き込まれます。 プログラムとデータの合計は 2K バイトの範囲を超える事は出来ません。 もし、この範囲を超える事があれば、 コンパイラー(Compiler)が知らせてくれます。

DATA statements はメモリーの未使用領域に書き込まれます。 初めに、DATA の書き込む場所を 0 (零) にセットします。 それからバイト毎にひとつずつ住所を進めます。

DATA statement の書き方は次の様にします。

```
table      DATA     "Here is a string..."
```

ここに : -table は独自のシンボル名
-テキスト(メッセージ等)は " " (引用符 double quotation mark)
の中に書きます

このシンボル名は data pointer (データがある場所をポイントする)である 定数値 (CON statement と同様)を割り当てます。 DATA の後のテキストは一定の語句で一字毎のバイトのリストになっています。 上記の例で、もしこの DATA statement がプログラムの最初とすれば table というシンボル名は 0 (零)という定数値を持ちます(data pointer)。そして、 Here is a string... というテキストが一字一字に分解され 1 バイトずつ EEPROM のメモリーに順番に保管されます。
Alt-M でそれらの結果を見る事が出来ます。

なお、data pointer は@サインを付けて新しいポインター値を表示する事によって何時でも指示が可能です。

例 ：

```
list    DATA    @$100,"some data"          'pointer 100(Hex)=256(Dec)
```

DATA には定義されたもの(defined DATA)と定義しない未定のもの(undefined DATA)の割当てについて、いくつかの使い方があります。

既定のデータ(defined DATA) はコンパイルする時 (compile time) に既に分かっているので宣言されます。 これに対して、 未定のデータ(undefined DATA) は、データの為のスペースだけを単に割り当てます。 undefined DATA はプログラムが走っている時 (run-time) にデータを取り込みます。

定義されたデータと定義されてないデータの例を次に示します。

defined data (定義されたデータ)

```
fee     DATA    0,1,2,3,4,5,6,7,8,9   'バイトを定義
fie     DATA    word 1000             '$03 と$E8 の 2 バイトを定義
                                      '(次ページの word 1000 参照)
foe     DATA    0 (256)               '初期値を 0 にして 256 バイト'を定義
```

[word 1000] について

ここにある 1000 は 10 進法で表されていて、2 進法と 16 進法で表すと ；

1000(Dec) = 0000 0011 1110 1000 (Bin) = 03E8 (Hex)

DATA はバイトサイズなので、 $E8 と$03 の 2 バイトになります。

undefined data (定義されてないデータ)

```
fum     DATA    (1024)          '未定のデータ (undefined data) 用に
                                ' 1 K バイトを予約
abc     DATA    word (16)'未定のデータ (undefined data) 用に
                                ' 16 ワードを予約
```

[DATA statement のまとめ]

データは EEPROM のプログラム領域に重ならない範囲でバイト使用を定義します。

- DATA は現在のデータポインターの定数値で割り当てたところのシンボル名で表す事が出来ます
- DATA はバイトサイズであるが ' word ' は保管場所を 2 バイトにワードを分けて使う事が出来ます
- @サインはデータポインターを変更する時に使います
- 定義されたデータは数字も文字も一緒に全て書き出します
- 定義されたデータは バイトかワードの (array) を使う事が出来ます
- 定義しないデータは先に値いが入っていない (array) を予約出来ます

DATA は DATA シンボルを付託したものを含む事も出来ます。

例 :

```
        t1      DATA        "Here's table 1...",0
        t2      DATA        "Here's table 2...",0
        t3      DATA        "Here's table 3...",0
        t4      DATA        "Here's table 4...",0

        start   DATA        word t1, word t2, word t3, word t4
```

3. [Run-time]

Run-time (プログラム実行)の表現

プログラム実行の表現は、 定数、 変数、 オペレイター(Binary Operators を参照) そして( )かっこ等を含む事が出来ます。そして、それらは16 ビットを使って分解させられます。

Constants (定数、一定のもの) は次のように幾つかの形で表す事が出来ます

```
label           ' label は CON statement を使って定数を定義します
$BA1F           ' Hex ( 16 進数 )
%111001111      ' Binary ( 2 進数 )
99              ' Decimal ( 10 進数 )
"A"             ' ASCII ( アメリカの工業規格での表現法)
```

[NOTE] Double quotation mark (引用符)の中の文字列はコンパイラーによってひとつずつの文字に分けられます。
例:

```
"DOG" は "D", "O", "G"
"String" は "S", "t", "r", "i", "n", "g"
```

次に、 unary (ユーナリー)、 binary (2 進法)、 conditional (条件付) 等の表現法を幾つかの例と共に示します。

[4] Unary Operator (ユーナリーオペレイター)

ABS	サイン付の 16 ビット値の絶対値(Absolute)
SQR	サイン無しの 16 ビット値の平方根(Square root)
DCD	4 ビット値(0, 1, ... 14, 15)の 2 の n 乗(1, 2, 4, ...16384, 32768)

NCD	16 ビット値のプライオリテイーエンコーダ(Priority encoder) (=>32768, =>16384,....=1 : 15, 14,.....1;0 ->$FFFF)
-	ニゲイション(negation)16 ビット値の 2 の余数(Two's complement)
~	ビットワイズ NOT(bitwise NOT)16 ビット値の 1 の余数 (One's complement) (チルドとも読みます)
SIN	8 ビット値の正弦(サイン)結果は+127～127 の範囲内です
COS	8 ビット値の余弦(コサイン)結果は+127～127 の範囲内です

(1) A B S (Absolute value ： 絶対値)

ある数の絶対値は、0 (零)からの差をプラスの数として表したものです。
次のプログラムで確かめて下さい。

```
w1 = -99                    ' Put -99 (two's complement format) into w1
        debug sdec ? w1     ' Display it on the screen as a signed #

w1 = ABS w1                 ' Now take its absolute value
        debug sdec ? w1     ' Display it on the screen as a signed #
```

(2) S Q R (Square root ： 平方根)

この SQR は 16 ビットで表された符号のない数を計算します。(実数でなければいけません)　また、BS2 は整数のみの計算しかしませんので、例えば、SQR100=10 ですが SQR99=9 (実際は 9.94987...)と小数点以下切捨てで計算されます。
次のインストラクションで確かめて下さい。

```
        debug SQR 100       ' Display square root of 100 (10)
        debug SQR 99        ' Display of square root of 99 (9 due to
                            ' truncation)
```

(3) D C D (ディコーダー)

2 の n 乗の型を 16 ビット中のビット番号を 1 にセットする事によって表します。

例

```
w1 = DCD 12              ' Set bit 12
debug bin ? w1           ' Display result (%0001000000000000)
```

(4) N C D (Priority encoder ： プライオリテイー　エンコーダー)

NCD は 16 ビットの値の中の最も高いビットの 1 (2 の n 乗の位置)を見つけ、そのビットの位置に 1 を加えたビット位置を返します。 もし、 16 ビットの中に 1 を含んでなければ NCD は 0 を返します。 NCD はある値より大きいか等しいかを早く見つける時に有効です。

例

```
w1 = %1101               ' Highest bit set is bit 3
debug ? NCD w1           ' Show the NCD of w1 (4)
```

(5)　- (Negates ： ニゲイツ)

Negates は 2's コンプリメントで表した 16 ビットの値です。

例

```
w1 = -99                 ' Put -99 (two's complement format) into w1
debug sdec ? w1          ' Display it on the screen as a signed #

w1 = ABS w1              ' Now take its absolute value
debug sdec ? w1          ' Display it on the screen as a signed #
```

(6) ~ (Tilde ： チルド)

ある数をビットで表した反対の数で 0 ならば 1 、 1 ならば 0 にします。
この操作は " bitwise NOT " (ビットワイズ ノット) として知られています。

例

```
b1 = %11110001                  ' Store bits in byte b1
     debug bin ? b1             ' Display in binary (%11110001)
b1 = ~ b1                       ' Complement b1
     debug bin ? b1             ' Display in binary (%00001110)
```

(7) S I N (Sine ： サイン)

与えられた角度のサイン(Sine) を求めることは、 その角度の端から円の中心線に垂線をおろしたその垂線の斜辺に対する割合を求めるものです(三角形の高さ)。

一般に使われている円が 0 から 359 度に分けるのに対して BS2 SIN operator は 0 から 255 までの数を使って分けます。 そしてこの円を半分にして 0 から 180 度までを 128unit として表し、 0 から-180 度までをマイナスの 128unit として表します。

この unit の事を brad (ブラッド) 或は binary radian と呼ぶ事があります。
なお、1 brad は 1.406 ° に相当します。
brads から角度に変換する計算は次のようにします。

(brads) x 180 ÷128 = (角度)

例

brads = 32
32 x 180 ÷128 = 45 °
brads = 96
96 x 180 ÷128 = 135 °

角度を brads に変換する場合は

(角度) x 128 ÷180 = (brads)

例

角度 = 45 °
45 x 128 ÷180 = 32 (brads)
角度 = -45 °
(-45) x 128 ÷180 = -32 (brads) [* 2's complement]

[Demo Program]

```
degr    var     w1                      ' Define variables
sine    var     w2

        for degr = 0 to 359 step 45  ' Use degrees
        sine = SIN (degr * 128 / 180)       ' Convert to brads、 do SIN

        debug "Angle: ",DEC degr,tab,"Sine: ",SDEC sine,cr      ' Display
        next
```

(8) C O S (Cosine ： コサイン)

SIN operator が垂線の斜辺に対する割合を求めるのに対して、 COS operator は底辺の斜面に対する割合を求める、という事が相違するだけであとは全く同じです。

デモ用のプログラムは SIN を COS に代えて試して下さい。

［5］ Binary (two‐argument) Operators

16種類の Binary Operators(バイナリー･オペレイター)を次に説明します。

+	加算
-	減算
/	除算
//	除算で余りを返す
*	乗算
**	乗算の結果の high 16 ビットを返す
*/	乗算の結果の middle 16 ビットを返す
MIN	最低値
MAX	最高値
DIG	10 進数を返す
<<	左にシフト
>>	右にシフト
REV	ビットの並びを逆さにする
&	Bitwise AND (ビットの AND 操作)
¦	Bitwise OR (ビットの OR 操作)
^	Bitwise XOR (ビットの XOR 操作)

(1) + (Addition ： 加法)

加算のオペレイターは変数又は定数の加算をして 16 ビットで結果を返します。範囲は整数の 0 から 65535 までで、それより大きい場合はキャリービット(carry bit) が無くなってしまいます。 もし、サイン付の 16 ビット数を加えた場合でも結果はサイン付の値で正しい答えを出します。

例えば、 (-1575)+976=(-599)　これをプログラムで確かめてみましょう。

```
w1 = -1575
w2 = 976

        w1 = w1 + w2            ' Add the numbers
        debug sdec ? w1         ' Show the result (-599)
```

(2)　- (Subtraction ： 減法)

```
w1 = 1000
w2 = 1999

        w1 = w1 - w2            ' Subtract the numbers
        debug sdec ? w1         ' Show the result (-999)
```

(3) /　(Division ： 除法)

0 から 65535 までのサイン無しの整数だけで除算を行ないます。 サイン付の数字を使うと正しい答えが出ませんので気を付けて下さい。

```
w1 = 1000
w2 = 5

        w1 = w1 / w2            ' Divide w1 by w2
        debug dec ? w1          ' Show the result (200)
```

もし、計算式の中に一つだけの負の数がある場合は絶対値で計算してその結果をニゲイト(negate)します。 その場合は、 ビット 15 はサインビットになりますので数字としては使えません。 従って、 全ての数は(-32767) から (+32767)の間でなければなりません。

例

```
sign      var       bit              ' Bit to hold the sign

w1 = 100
w2 = -3200

sign = w1.bit15 ^ w2.bit15           ' Sign = (w1 sign) XOR (w2 sign)
          w2 = abs w2 / abs w1       ' Divide absolute values

if sign = 0 then skip0               ' Negate result if one of the
          w2 = -w2                   ' arguments was negative

skip0:
          debug sdec ? w2            ' Show the result (-32)
```

(4) //　Double Slash (ダブルスラッシュ)

Return remainder ： 剰余(割り算の余り)を返すオペレイターです。
例えば、 1000/6 = (約) 166.667 です。 整数の計算ですから小数点以下は切捨てになり答えは 166 と出ます。 166 はおよその数で 166 * 6 = 996 で 1000 には 4 だけ不足、すなわち 4 が余り(remainder) という事になります。　この場合は // (double slash) は 4 を返します。
もし、 1000/5 のように余りが 0 の場合は 0 を返します。

例

```
w1 = 1000
w2 = 6

          w1 = w1 // w2              ' Get remainder of w1 / w2
          debug dec ? w1             ' Show the result (4)
```

(5) * (Multiplication ： 乗法)

整数の乗算でサイン無しの場合は 0 から 65535 までの計算を行ないます。 もし、乗算の結果が 65535 より大きい場合は 16 ビットを超えたものが失われます。 また、サイン付の乗算の場合は(-32767) から (+32767)の間で正しい結果が得られます。

例

```
w1 = 1000
w2 = -19
        w1 = w1 * w2            ' Multiply w1 by w2
        debug sdec ? w1         ' Show the result (-19000)
```

(6) ** (Double star ： ダブルスター)

もし、 16 ビット同士の乗算をすれば、 結果は 32 ビットの大きさになる可能性があります。 しかし、普通の乗算では 16 ビットの大きさしか出来ませんので、上位 16 ビットの計算を出来るようにしたのが、このダブルスターです。

例えば、 65,000 * 65,000 = 4,225,000,000 ($FDE8 * $FDE8 = $FBD46240)

この例では、 普通の乗算は下部 16 ビットですから答えは $6240 です。 そこでダブルスターを使うと、 上部 16 ビットですから答えは$FBD4 となります。

次のプログラムで確かめて下さい。

```
w1 = $FDE8

        w2 = w1 ** w1           ' Multiply $FDE8 by $FDE8
        debug hex ? w2          ' Return high 16 bits
```

(7) */ (Star Slash ： スター　スラッシュ)

変数又は定数の乗算において、 32 ビットの結果の中の中間 16 ビットを返すためのオペレイターです。 これは、 小数点を含んだ数字の乗算に威力を発揮します。
整数の乗算は上部のバイト(上位 16 ビット、 0 から 255 或は$FF の数)で行い、 小数点以下の数字は下部のバイト(下位 16 ビット、 0 から 255 までの 1/256 ずつの数)で行ないます。

例えば、 ある数に 1.5 を掛ける事をやってみましょう。 小数点より上の数はそのまま 1 を上部のバイトと考えればいいでしょう。 小数点以下の数字、 0.5 は 128/256=0.5 ですから下部のバイトを 128 と考えます。 これを 32 バイトの Hex ナンバーで表すと上部バイトと下部バイトがはっきりと分かれているので分かりやすいと思います。 1.5 を Hex で表すと$0180 となります。 次のプログラムで確かめて下さい。

例

```
        w1 = 100

        w1 = w1 */ $0180                ' Multiply by 1.5 [1 + (128/256)]
             debug ? w1                 ' Show result (150)
```

この */ インストラクションの計算は整数部分を上位のバイトに置き、 小数点以下の数字を 256 倍してそれを Hex ナンバーで下位バイトにすればよいでしょう。
例えば、円周率 π=3.14159 の場合は 3 は Hex でも$03、 そして 0.14159 * 256 = 36 = $24 ですから スタースラッシュ */ は$0324 となります。 決して、完璧な計算とは言えませんがエラーは約 0.1%ですからそれほど問題はないと思います。

(8) MIN (Minimum ： ミニマム (最低値))

16 ビットの正の整数で最低値を決めます。 使い方は次のようにします。

value MIN limit

ここに： - value は MIN インストラクションを実行する値
- limit は使用可能な value の最低値

この事は次のように言い換える事が出来ます。

もし、value が limit より小さければ value = limit にし、もし value が limit に等しいかそれより大きい時は value はそのままにする。

例

```
for w1 = 100 to 0 step -10          ' value w1 を 100 から 0 に 10 づつ
                                    ' 減らす
debug ? w1 MIN 50                   ' Show w1、 but use MIN to clamp at
                                    ' 50 ( 50 未満は無視 )
next
```

(9) MAX (Maximum ： 最高値)

16 ビットの正の整数で最高値を決めます。 使い方は次のようにします。

value MAX limit

ここに： - value は MAX インストラクションを実行する値
- limit は使用可能な value の最高値

この事は次のように言い換える事が出来ます。

もし、value が limit より大きければ value = limit にし、もし value が limit に等しいかそれより小さい時は value はそのままにする。

例

```
        for w1 = 0 to 100 step 10   ' value w1 を 0 から 100 に 10 づつ増やす
        debug ? w1 MAX 50           ' Show w1、 but use MAX to clamp at 50 ( 50
                                    ' を超える数は無視 )
        next
```

(10) D I G (Digits ： 数字)

DIG で指定された場所の正の整数で表された数字を返します。 指定場所とは 0 (一番右の数字) から 4 (一番左の数字)の 5 つ(0 から 65535 の数字)の事です。

例

```
w1 = 9742
        debug ? w1 DIG 2            ' Digit 2 (右から 3 番目) = 7 を表示
for b0 = 0 to 4                     ' b0 は 1 バイトのメモリー
        debug ? w1 DIG b0           ' Show digits 0 through 4 of 9742
next
```

(11) < < (Shift left ： 左にシフト)

例 : shift left

```
        25 << 3             ' 25 を左に 3 回シフトするという事は 25 に 2 を
                            ' 3 回掛けるのと同義、 即ち、25*2*2*2 = 200(Dec)

        25 (Dec)        = 00011001 (Bin)    ' 25 を 2 進数で表す
        25 x 2          = 00110010          ' 1 回左にシフトする
        25 x 2 x 2      = 01100100          ' 2 回目のシフト
        25 x 2 x 2 x 2  = 11001000          ' 3 回目のシフトで答えは 200 (Dec)
```

(12) > > (Shift right ： 右にシフト)

例 ： shift right

```
112>>4              ' 10 進数の 112 を右に 4 回シフトするという事は
                    ' 112 を 2 で 4 回割るのと同義、 又は 112 ÷16 = 7

112 (Dec)                   = 01110000      ' 112 を 2 進数で表す
112 ÷ 2                     = 00111000      ' 1 回右にシフト
112 ÷ 2 ÷ 2                 = 00011100      ' 2 回目のシフト
112 ÷ 2 ÷ 2 ÷ 2             = 00001110      ' 3 回目のシフト
112 ÷ 2 ÷ 2 ÷ 2 ÷ 2 = 00000111      ' 4 回目のシフトで答えは
                                            ' 7(Dec)
```

(13) R E V (Reverse ： 裏返し)

このオペレイターは二進数を鏡で見たように裏返して返します。
例えば、 %10101101 の REV4 は %1011 とビット 0 からビット 3 まで(1101)を裏返します。

例

```
debug bin ? %11001011 REV 4         ' Mirror 1st 4 bits (%1101)
```

(14) & (Bitwise AND ： ビットワイズ　アンド)

2 つの数の各ビットの AND をとり結果を返します

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

例

debug bin ? %00001111 & %10101101 ' AND の結果を表示 (%00001101)

(15) ¦ (Bitwise OR ： ビットワイズ　オア)

2 つの数の各ビットの OR をとり結果を返します

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

例

debug bin ? %00001111 ¦ %10101001 ' OR の結果を表示 (%10101111)

(16) ^ (Bitwise XOR ： ビットワイズ エックスオア)

2 つの数の各ビットの XOR をとり結果を返します

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

例

debug bin ? %00001111 ^ %10101001 ' XOR の結果を表示 (%10100110)

[6] Conditional operators

条件を調べる Conditional 表現 (IF) の使用には、 特別な unary operator と幾つかの binary operators があります。 Conditional operators は最も高い優先順位を持っています (highest priority)。

NOT	最も高い優先順位の unary
AND, OR, XOR	最も高い優先順位の binaries

[注] : 上記 4 つのオペレイターは算術的には unary、 binary operator の

~

&、 |、 ^

と同じですが、適用の方法に相違があります。

優先順位は低いですが、条件を調べる binary operators (普通のオペレイターよりは優先順位は高い)を次に示します。

```
<       ' Less than ( 以下 )
<=      ' Less than or equal to ( 以下又は等しい )
=       ' Equal to ( 等しい )
=>      ' Equal to or greater than ( 等しい又は以上 )
>       ' Greater than ( 以上 )
<>      ' Not equal ( 等しくない )
```

これらのオペレイターは、 その条件に合致してなければ (false フォルスと言う) 0 (零)を返し、 合致していれば (true トゥルーと言う) $FFFF を返します。

例 :

```
IF x<1 or not y>3 and z=0   then loopback
                    'もし x<1 か又は y が 3 より大きくなく、 なお且つ
                    ' z=0 なら ラベル名 loopback に行け
```

第5章 ベーシック・スタンプ 2の命令言語

ベーシック言語は、私達が日常使っている言葉にとても近いので、基本的な英語が分かる人なら誰でも使えるプログラミング言語です。特に、ベーシック・スタンプ用に開発された PBASIC （ピーベーシック）命令言語は、IF．．．THEN．．．，GOTO ．．．，FOR..．．．NEXT などの、いわゆる、代表的なベーシック言語を使うかたわら、PULSOUT, BUTTON,SERIN, SEROUT 等々、とても使い易く改良された命令言語が含まれ、初心者にも大変取り組み易くなっています。

基本的に PBASIC の命令言語（Instruction：インストラクション）は、次の 3 種類に大きく分ける事ができます。

1．INPUT（入力）
2．DECISION／PROCESSING（決定／処理）
3．OUTPUT（出力）

PBASIC の代表的な INSTRUCTION SET

INPUT 入力	DECISION/PROCESSING 決定／処理	OUTPUT 出力
SERIN….	IF….THEN….	PWM….
INPUT…	BRANCH….	FREQOUT….
BUTTON….	GOTO….	SHIFTOUT….
PULSIN….	GOSUB….	SEROUT….
	FOR….NEXT	OUTPUT….
	PAUSE….	LOW….
	SLEEP….	HIGH….
		DEBUG….

上の表は、代表的なインストラクションですが、これだけでもかなり複雑なプログラムを組む事が可能です。

【 プログラムを書くにあたっての注意事項 】

プログラムの書き始めには、何のプログラムなのかが分かるように、タイトルとかメッセージ、日付等を書きます。これは、Remark(リマーク)と言って、どこに書いてもよいですが必ず Apostrophe(アポストロフィ：(’)の意味)の符号を始めに付けてからアルファベット(英語)で書きます。また、長さの制限はありませんが、行が変わったら必ず Apostrophe で書き始めなければいけません。この Remark は、各インストラクションの後ろにも書くように習慣をつけると、あとでプログラムを見直した時などにとても役に立ちます。

各インストラクションの説明と使用例

[1]BRANCHING ブランチング

メインプログラムから出て別のプログラムに行ったり、比較してその結果で指定のラベルにジャンプさせたりする**時に使用します。ラベル名は分かり易い名前が良いでしょう。そしてラベル名の後ろにはコロン(:)を付けます。**

(1) IF....THEN (イフ ゼン)

もし. . . なら. . . に行け

使い方

```
IF condition THEN addressLabel

        condition       ; 仮定の状態を指示する
        addressLabel    ; 仮定した状態が正しい時に行くラベル名
例                      ;解説
IF  x=1 THEN  repeat    ;「もし x=1 ならラベル名 repeat に行け」という
                        ;命令で x=1 でなければ次のインストラクション
                        ;に進む
        ....            ;次のインストラクション
repeat:                 ;ラベル名 (コロンを付けなければいけません)
        ....            ;インストラクション
```

プログラム例

```
' Define Variables and Constants

speed    con      50                ' constant value speed=50
counter  var      word              ' counter is variable

loop:                                ' ラベル名 " loop "
         counter = counter - speed   ' 右辺の counter - 50 を計算してその答えを左
                                     '  辺の counter に入れる
         if counter = 0 then finish  ' もし counter=0 ならラベル名 finish に行け
                                     ' (もし counter=0 でなければ次のインストラクシ
                                     'ョンに行く)
goto loop                            ' ラベル loop に行け
finish:  end                         ' プログラム終了
```

(2) BRANCH　(ブランチ)

多くの IF...THEN が続く場合に便利

使い方

```
BRANCH   offset,[address0,address1,..addressN]

         offset          ;[ ]の中のどちらのアドレスに行くかを指定する
         address         ;どこのアドレスに行くかのラベル名
```

プログラム例　次の様な場合に BRANCH を使うと便利です

```
direction   var     word     ' direction の宣言、 サイズはワード(2 bytes)

if direction = 0 then case_0  'direction=0 ならラベル名 case_0 に行け
if direction = 1 then case_1  'direction=1 ならラベル名 case_1 に行け
if direction = 2 then case_2  'direction=2 ならラベル名 case_2 に行け
```

```
case_0:  .... ( インストラクション 0 )
case_1:  .... ( インストラクション 1 )
case_2:  .... ( インストラクション 2 )
```

上記の 3 つの if statements を BRANCH を使って書くと次の様になります。

```
direction   var     word            ' direction は変数でサイズはワード(2 bytes)

BRANCH direction, ( case_0, case_1, case_2)
                                    'direction の値が offset ですから direction=0 な
                                    'ら case_0、direction=1 なら case_1、
                                    ' direction=2 なら case_2 に行きます
case_0:  .... ( インストラクション 0 )
case_1:  .... ( インストラクション 1 )
case_2:  .... ( インストラクション 2 )
```

(3) GOTO (ゴートュー)

。。。に行け

使い方

```
GOTO    addressLabel
addressLabel        ; どこに行くかのラベル名
```

プログラム例

```
loop:                       ' ラベル名
        x = x + 1
goto loop           ' 無条件でラベル名 loop に行く
```

(4) GOSUB (ゴーサブ)

指定のサブルーチンに行け

使い方

```
GOSUB   addressLabel
addressLabel                ; どこに行くかのラベル名
```

プログラム例

```
note    var     word            ' note ワード宣言(2 バイト)
tone    var     word            ' tone ワード宣言(2 バイト)

        loop:                   ' ラベル名
                note = 1000     ' note は 1000 ( 10 進数)
                tone =  520     ' tone は  520 ( 10 進数)
                gosub   test    ' ラベル名 test (サブルーチン) に行け

                note =  500     ' note は  500 ( 10 進数)
                tone =  650     ' tone は  650 ( 10 進数)
                gosub   test    ' ラベル名 test (サブルーチン) に行け

        goto  loop              ' ラベル名 loop に行け
test :                          ' ラベル名 (サブルーチン)
        freqout  15, note, tone ' 音をピン 15 から note の周波数で tone
                                ' の音色を出せ
        pause 100               ' 100 msec (0.1 秒)休止
return                          ' リターン [ gosub の命令を発した場所+1 (次
                                ' のインストラクション)に帰れという命令 ]
```

(5) RETURN (リターン)

サブルーチンから来た場所の次の命令に帰れ

使い方

RETURN

サブルーチンから戻る

プログラム例　GOSUB の項を参照して下さい

[2] LOOPING ルーピング

指定の条件を満たすまで、グルグル続ける

(1) FOR....NEXT (フォー ネクスト)

。。。の条件に合ったら、ループから出る

使い方

```
FOR variable = start TO end {STEP stepVal}
        ･･･( インストラクション)
NEXT
variable ;カウンターとしてのビット、ニブル、バイト、ワード等
start    ; 変数の始めの数
end      ; 変数の最後の数
stepVal  ; オプション、 For...Next ループのステップの状態指示
```

プログラム例

```
Hz      var     word                    ' Hz のワード宣言

        FOR   Hz = 1 TO 4000 STEP 100   ' Hz は 1 から 100 毎に 4000 まで繰り
                                        ' 返す
        freqout  15, 70, Hz, 4000-Hz    ' ピン 15 から 70msec の間で Hz の周
                                        ' 波数を出しその後 4000-Hz の周
                                        ' 波数を出す
        NEXT                            ' 次の Hz で freqout を行なう
```

また、FOR…NEXT は、for…next の内部に別の for…next を作る事が出来ます。これをネスティド(NESTED)と言います。

FOR….NEXT の NESTED (ネスティド例)：

```
x       var     byte
```

```
y       var     byte

        for     x = 1 to 5
        debug ? x                   'x を表示
        for     y = 1 to 5
                debug ? y           'y を表示
        next
next
```

上のプログラムは、x=1 を表示後、y=1,y=2,・・y=5 と表示し x=2 と表示した後、y の表示を繰り返すものです。

[3] NUMERICS ニューメリックス

数

(1) LOOKUP (ルックアップ)

インデックスで指定した値を探し変数として結果を保管

使い方

LOOKUP index,[value0,value1,..valueN],resultVariable

index ; リストから取り出す値の項目番号

value0, value1.. ; サイズ 16 ビット以内での変数又は定数リスト

resultVariable ; インデックスで指定された value をここに保管

もし、 index = 0 なら value0 を resultVariable (宣言された変数)に保管、

もし、 index = 1 なら value1 を resultVariable に保管します。

・・・・

もし、 index = N なら valueN を resultVariable に保管します。

プログラム例

4 つのモーターコイルのついたステッピングモーターをコントロールし、DEBUG でその動きを PC 上に表示するプログラムを作成します。

ステッピング モーター(Stepping Motor 又は Stepper Motor) の各コイルには 1 本ずつ、計 4 本の I/O ラインを使い次の表のような信号を加えます。

Step #	Binary	Decimal
0	1010	10
1	1001	9
2	0101	5
3	0110	6

ステッピングモーターにステップ#0 から#3 までの信号を与え、これを繰り返す事によって、モーターは回ります。

(実際にはステッピングモーターは接続せず、スクリーン上で動きを見てください。)

```
'REM Driving a stepping motor with on-screen simulation

dirs = %00001111                    ' I/O ピンの出入力方向を決めます
                                    ' 0 は入力、1 は出力を指定します
Rotate:
        for b2 = 0 to 3             'b2 is pre-defined byte variable

        LOOKUP b2,(10,9,5,6),b3     'b3 is pre-defined byte variable

        pins=b3                     ' ステップのパターンを pins に保管
        LOOKUP b2,("¦ ／―＼"),b3 ' オフセット(0~3)を"picture"用に変換

        debug cls,#%pins,"   ",#@b3 ' "モーター"のアニメイション表示
                                    '  注:double quotes の間は2スペース
        next
goto Rotate                         ' 繰り返す
```

(2) LOOKDOWN (ルックダウン)

指定した値とマッチしたものを探し出す

使い方

LOOKDOWN value,{comparisonOp,}[value0,value1,.valueN],resultVariable

value ； リストの値と比較するための変数又は定数

comparisonOp ； オプションだが、次の中のひとつを使用しても良い

=	等しい (default : comparisonOp を書かない場合)
<>	等しくない
>	それ以上
<	それ以下
>=	等しいかそれ以上
<=	等しいかそれ以下

valeu0,value1... ； サイズ 16 ビットまでの変数又は定数の値

resultVariable ；マッチしたものが見付かった場合にそのインデックス番号を保管する

解説

comparisonOp は value と value0 を比較します。 もしその比較が True なら 0 (零) を resultVariable の中に書き、 False なら次の比較(value と value1)をします。 その比較の結果が True なら 1 を resultVariable に書き込みます。

この比較のプロセスを True が生じるか全部の比較が終るまで続けます。

もし、 該当するものが無い場合は、 resultVariable には何も書かれません。

(3) RANDOM (ランダム)

乱数を発生させる

使い方

RANDOM wordvariable

wordvariable ； ワードで表された乱数を保管する

[4] DIGITAL I/O ディジタル アイオー

ディジタル入出力

(1) OUTPUT (アウトプット)

ピンをアウトプット(出力)にセットするがラッチはしない

使い方

```
OUTPUT pin
pin                 ; 出力として使用するピン番号 0 ~ 15
```

プログラム例

```
output   0          ' ピン 0 を出力にセットする
output   15         ' ピン 15 を出力にセットする
```

(2) LOW (ロー)

ピンを出力にセットし出力を low(0) にラッチする

使い方

```
LOW   pin
pin                 ; ローにしたい出力ピンの番号 0 ~ 15
```

(3) HIGH (ハイ)

ピンを出力にセットし出力を high(1) にラッチする

使い方

```
HIGH   pin
pin                 ; ハイにしたい出力ピンの番号 0 ~ 15
```

(4) TOGGLE (タグル)

ピンを出力にセットして出力が 1 なら 0 に、0 なら 1 に反転

使い方

```
TOGGLE pin
pin                 ;0 から 15 までの出力ピンの状態を反転させる
```

(5) PULSOUT （パルスアウト）

指定されたパルスを出力する

使い方

```
PULSOUT pin,period
pin             ; パルス出力するピン番号を指定
period          ; 単位 2 マイクロ秒でのパルス幅指定( 0 ~
                  65535 )
```

プログラム例

```
PULSOUT 2, 1000         ' ピン 2 から 2x1000=2000 マイクロ秒=2 ミリ秒
                        ' のパルスを出す
```

(6) INPUT （インプット）

ピンをインプット（入力）にセットするがラッチはしない

使い方

```
INPUT pin
pin             ; 入力として使用するピン番号 0 ~ 15
```

プログラム例

```
input   0               ' ピン 0 を入力にセットする
input   15              ' ピン 15 を入力にセットする
```

(7) PULSIN （パルスイン）

入力するパルスを計る

使い方

```
PULSIN pin,state,resultVariable
pin ; パルス入力するピン番号
state ; パルスを立ち上がり(1)から計るか立ち下がり(0)から計る
        かを設定
resultVariable ; パルス幅の値を保管する
```

解説

ピン番号を指定するとそのピンは入力モードになります。 state で決められた所(立ち上がりか立ち下がり)から 2 マイクロ秒の分解度で入力したパルスを測定します。 その結果は resultVariable に保管されます。

最大 65535 x 2 (マイクロ秒) か 131 m 秒でそれ以上の場合は resultVariable には 0 (零) が書き込まれます。

プログラム例

```
width   var     word            ' width のワード宣言
        pulsin  4, 1, width     ' ピン 4 のパルスの立ち上がりから立ち下がる
                                ' まで(パルス幅) を測り width に保管する
        ･････                   ' 次のインストラクション
```

(8) REVERSE (リバース)

もしピンが出力にセットされていたら入力に、入力にセットされていたら出力にセットする

使い方

REVERSE pin

pin ; 変数か定数(0 から 15) で指定したピンの入出力を反転

プログラム例

```
dir3 = 0        ' ピン 3 を入力 ( input ) にセットする
reverse 3       ' ピン 3 を出力 ( output ) にセットする
･････           '(インストラクション)
reverse 3       'ピン 3 を入力 ( input ) にセットする
```

(9) BUTTON (ボタン)

プッシュボタン等のスイッチを検知する

使い方

BUTTON pin,downstate,delay,rate,bytevariable,targetstate,address

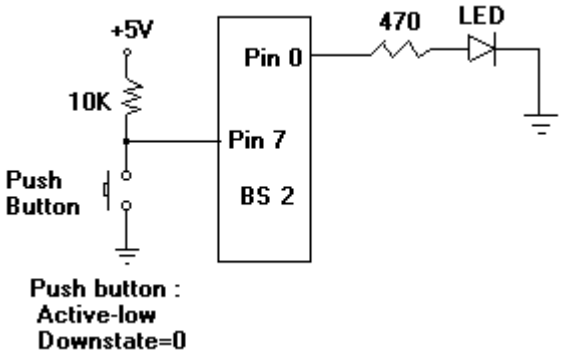
pin ; I/O pin number

downstate ; プッシュボタンが押された時の状態 (1か0)

delay ； どの位プッシュボタンが押された状態が必要かを示す
rate ； ボタンルーテインのサイクルを示す
bytevariable ； プッシュボタンの状態を書き込む
targetstate ； プッシュボタンがどちらの状態（1か0）の時ブランチするかを指定する
address ； ブランチする所のラベル名

プログラム例

次の回路でプッシュボタンを押すごとに LED が点滅するプログラムを作る。




```
' Toggles the state of an LED when the push button is pressed

b2 = 0                                  ' cleared bytevariable
        loop:
                BUTTON 7, 0, 200, 100, b2, 0, skip_1
                                        ' 7=input pin 7,
                                        ' 0=pushbutton を押した時 0 になる,
                                        ' 200=delay time,
                                        ' 100=auto repeat rate,
                                        ' b2=workspace 1 byte variable,
                                        ' 0=ぼたんが押されてない状 態の時ブ
                                        '  ランチに行く( 1 =  押されている状態) 、
                                        ' skip_1=ブランチのアドレス、
```

```
        toggle 0          ' ピン 0 の信号を反転する
skip_1:                   ' アドレス名(ラベル名)
goto loop                 ' loop に行け
```

(10) SHIFTIN (シフトイン)

クロックに同期させた信号を入力する

使い方

SHIFTIN dpin,cpin,mode,[result{＼bits}{,result{＼bits}...}]

dpin ; 同期シリアル機器のデータ出力に接続する入力ピンの指定

cpin ; 同期シリアル機器のクロック入力に接続する出力ピンの指定

mode ; 予め決められた 4 種類のデータビットの構成を決める 0 から 3 までの値 (下の MODE 詳細を参照)

result ; 入力したデータを保管する variable 名

bits ; ビット数を指定 (オプションで指定がなければ自動的に 8 ビット)

[MODE の詳細]

0 はクロックパルスの立ち上がりで MSB(Most Significant Bit)が最初

1 はクロックパルスの立ち上がりで LSB(Least Significant Bit)が最初

2 はクロックパルスの立ち下がりで MSB(Most Significant Bit)が最初

3 はクロックパルスの立ち下がりで LSB(Least Significant Bit)が最初

0 ~ 3 の値いは次のシンボル名を使っても良いです。

値	シンボル名
0	MSBPRE
1	LSBPRE
2	MSBPOST
3	LSBPOST

プログラム例

```
SHIFTIN 0, 1, msbpre, [b1]          ' ピン 0=データ入力, ピン 1=クロック アウト,
                                    ' クロックの立ち上がりで MSB から入力 b1
                                    ' に書き込む

debug "Pre-clock ： ", bin 8 b1, cr
                                    ' スクリーンに[Pre-clock ： (バイナリー値)]を
                                    ' 表示し, キャリジリターン( cr )を送る
```

(11) SHIFTOUT （シフトアウト）

クロックに同期させた信号を出力する

使い方

SHIFTOUT dpin,cpin,mode,[data{＼bits}{,data{＼bits}...}]

dpin ； データ出力ピンの指定

cpin ； クロック出力ピンの指定

mode ； 予め決められた 2 種類のデータビットの構成を決める 0 と 1 の値 (下の MODE 詳細を参照)

data ； 出力するデータ(データは毎秒約 16 kbits で出力します)

bits ； ビット数を指定 (オプションで指定がなければ自動的に 8 ビット)

[MODE の詳細]

0 は LSB(Least Significant Bit)が最初

1 は MSB(Most Significant Bit)が最初

0 と 1 の値いは次のシンボル名を使っても良いです。

値	シンボル名
0	LSBFIRST
1	MSBFIRST

(12) COUNT (カウント)

ミリ秒でサイクルを数える

使い方

COUNT pin, period, variable
pin ; I/O pin 番号
period ; カウントする時間をミリ秒で指定する
variable ; 通常ワードとしてカウントした数を保管する

[解説]

period は variable 又は コンスタント ナンバー(1 から 65535 まで) で指定します。測定は 4 マイクロ秒のパルス幅で行なう為、 1 サイクルの入力パルス(0 から 1 又は 1 から 0 の変化) 8 マイクロ秒が最小で最高 125 KHz まで測定出来ます。

プログラム例

プッシュボタンをピン 7 につないで、 1 秒間に何回押せるかを測定する

```
cycles          var     word
loop:
        debug cls, "How many times can you press the button in 1 seconds ?", cr
                'cls はスクリーンをクリアーしてカーサーをホームポジションに
                '置く。 メッセージの後の cr はキャリジリターン
        pause 1000: debug "Ready, Set ...."; pause 500 : debug "GO !", cr
                '1000 msec (ミリ秒) 休止後、スクリーンに表示
        count 7, 1000, cycles
                'ピン7で1秒間カウントする
        debug cr, "Your score is :  ", DEC cycles, cr
                'cycles を10進法で表示する
        pause 3000
                '3000 msec = 3 秒  休止
        debug "Press button if you like to try again. ", cr
                'cr (Carriage Return)  改行
```

```
hold:   if IN7 = 1 then hold' プッシュボタンが押されるまでこれを繰り返す
                                    ' IN7=0 なら次のインストラクションに行く
        goto loop
```

[5] SERIAL I/O シリアル

直列入出力

(1) SERIN (シリアルイン))

シリアル (直列) データ入力

使い方

SERIN rpin{＼fpin},baudmode,{plabel,}{timeout,tlabel,}[inputData]

rpin ; シリアルデータをどのピンに入力するかを変数又は 0 から 15 までの定数で指定する

fpin ; オプション byte-by-byte handshaking の為のフローコントロールの為のピンを変数又は定数で指定する

baudmode ; 16 ビットで表された変数又は定数で指定したタイミングと配列を決める([Data の速さと Baudmode 値の関係]を参照)

plabel ; オプション、 parity error の時にジャンプする行き先ラベル名

timeout ; 変数又は 0 から 65535 までの定数で表したミリ秒の、 データを受け付ける時間を指定する

tlabel ; オプション、 timeout と一緒に使いデータが指定の時間内に来なかった場合のジャンプ先のラベル名

inputData ; データを受け取った時にそのデータをどうするかを決める変数又は修正するもの(modifier)等を指定する事が出来る

この SERIN は RS232C などの非同期のデータ(Asynchronous data)を受ける時に使います。 ごく基本的なものだけでのプログラム例を示します。

プログラム例

入力にピン 1 を使い、 9600 bps, 8 bits, no parity, inverted の RS232C のデータ

を受けて、 serData に保管します。

```
serData    var      byte              ' serData のバイト宣言

           serin 1, 16468, [serData]   ' baud rate 19200 bps, 8 bits no parity
                                       ' 次の[Data の速さと Baudmode 値の関係 ]を
                                       ' 参照
```

[Data の速さと Baudmode 値の関係]

Data Speed	Direct Connection (Inverted)		Through Line Driver (Noninverted)	
Baud Rate	8 bits no parity	7 bits even parity	8 bits no parity	7 bits even parity
300	19697	27889	3313	11505
600	18030	26222	1646	9838
1200	17197	25389	813	9005
2400	16780	24972	396	8588
4800	16572	24764	188	8380
9600	16468	24660	84	8276
19200	16416	24608	32	8224
38400	16390	24582	6	8198

(2) SEROUT (シリアルアウト)

シリアル (直列) データ出力

使い方

SEROUT tpin,baudmode,{pace,}[outputData]

SEROUT tpin＼fpin,baudmode,{timeout,tlabel,}[outputData]

tpin ; シリアルデータをどのピンから出力するかを指定する

fpin ; オプション byte-by-byte handshaking の為のフローコントロール
の為のピンを変数又は定数で指定する

baudmode ； 16 ビットで表された変数又は定数で指定したタイミングと
配列を決める([Data の速さと Baudmode 値の関係]を参照)
timeout ； 変数又は 0 から 65535 までの定数で表したミリ秒で、fpin の
フローコントロールを待つ時間を指定する
tlabel ； オプション、 timeout と一緒に使いデータが指定の時間内に来な
かった場合のジャンプ先のラベル名
pace ； オプション、 送信の時のバイトとバイトの間隔をミリ秒で表す
outputData ； どの様な構成で送信するかを指定する

プログラム例

```
temp    var     byte
temp = 25

        serout 1, 16468, ["Temperature is ", DEC temp, "degrees C."]
                        ' Temperature is 25 degrees C の文字列をピン 1 から
                        ' 9600 bps のボードレイトで送り出します
```

[6] ANALOG I/O　アナログ アイオー

アナログ出入力

(1) PWM （ピーダブルエム）

指定したピンからアナログ信号(パルス幅変調)を送り出す

使い方

PWM pin,duty,cycles
pin ； 出力ピンを指定する
duty ； 0 から 5 v の電圧を 0 から 255 までに分け指定する
[計算式：(duty/255)*5v　例：duty=100　(100/255)*5=1.96v(出
力電圧)]
cycles ； ミリ秒で表した PMW 出力サイクル

【 解説 】: Pulse Width Modulation (PWM)

もし、アウトプットピンをハイ(1)にすればそのピンは約 5V の電圧になります。 また、もしそのピンをロー(0)にすれば約 0V になります。 もしも、そのハイとローをものすごく速くスイッチングしたらどうなるでしょうか? ハイで 5V に近づくとローにスイッチして、今度はロー(0)に近づく前にまたハイにする、 これを速くスイッチングするわけです。ある速さ以上でこれをやると 0 と 5V の中間、、2.5V 位でとまったように見えるでしょう。

これが PWM のアイデアです。 ディジタルの 1(約 5V)と 0(約 0V)を適当な速さで出力してアナログの電圧を造るわけです。

PWM の 1 と 0 を出す割合を duty cycle (デューティーサイクル)と言います。 この duty cycle が高ければ出力電圧が高い、というように直接アナログ電圧をコントロールします。 BS2 (Basic Stamp 2)の duty cycle は 0 から 255 の間で設定出来ます。

Duty は PWM のインストラクションで割合を指定します。 その割合の決め方は次の様にします。

計算式は : 出力電圧 = (duty / 255) * 5V

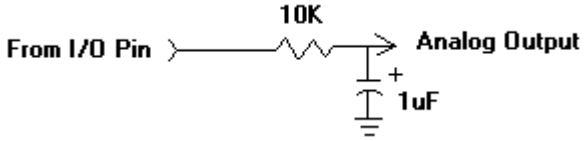
例 :

duty を 100 として上式で計算すると
(100 / 255) * 5 = 1.96 V
となります。

もし、2V の電圧を出力したければ
duty = (255 / 5) * 2 = 102
従って、 PWM のインストラクションの duty に 102 と入れれば良いわけです。

ここで充分に気を付けなければいけない事は、 上の例で示したように BS2 がこの PWM のインストラクション(命令)を実行している間は、指定通り電圧を出力しますがインストラクションが終って次のインストラクションに移った途端 0 か 1 (0V か 5V) で止まってしまいます。 そしてまた、出力はアナログ電圧と言っていますが、 この出力電圧を顕微鏡的に見れば、 0 と 1 のパルスのスイッチングである、 と言う事です。

これでは困りますので、 次の図に示す様な回路を作ります。 BS2 の出力はあくまでもディジタルの信号ですからその信号を抵抗を通して Capacitor (キャパシター或はコンデンサー)にチャージしてその電圧を溜めようという事です。 この様にすればPWM のインストラクションが終った後でも電圧はその Capacitor にチャージされていますから BS2 が別の事をやっていても構わないわけです。



PWM の回路図

(Capacitor の単位は通常 μF(マイクロファラッド)ですが uF と書く事もあります)

上の回路でどの位の間電圧を保つかは、 これに繋ぐ外部の回路がどの位の電流を使うかという事と Capacitor の内部漏電率によります。 電圧を一定に保つにはPWM インストラクションを適宜に繰り返して Capacitor に電圧を補充する必要があります。 Capacitor から電圧が放電するのにある時間がかかるのと同じ様に充電するのにもある時間が必要です。 それを指定するのが PWM インストラクションの cycles です。 この cycles は約 1 msec の周期になっています。 ですから、 もし 5 msec 充電したければ cycles を 5 と指定すれば良いわけです。 Capacitor に充電する時間は次の様に計算する事が出来ます。

Charge time = 4 * R * C (seconds ： 秒)

例えば、 上図の回路で 10 K Ω (10000 Ω) の抵抗と 1 μF(マイクロファラッド)=(0.000001 ファラッド)の Capacitor ですから

Charge time = 4 * 10000 * 0.000001 = 0.04 (秒) = 40 msec

従って、 各サイクルは約1 msec ですからPWM cycles を40 と指定すれば電圧は一定に保たれる事になります。出力ピンを 0 として 1.96 V の電圧を出力するインストラクションは次の様になります。

```
' PWM instruction for 1.96 V

        PWM 0, 100, 40              'ピン 0 から Capacitor に 1.96 V を充電する
```

PWM パルスを出した後 BS2 のピンはインプットモードでそのピンの出力用回路は切り離されます。 もし切り離されなくアウトプットのままに置かれるとすれば、PWM インストラクションで確立された電圧ではないものが Capacitor に充電される事になってしまいます。

上の PWM インストラクションで電圧を Capacitor に供給しても、 外部回路(負荷)がその充電したものを使ってしまえば無くなってしまいます。 従って、 絶えず Capacitor に充電をしていないと外部回路が働きません。 どの位の頻度で PWM インストラクションを繰り返さなければならないかは外部回路がどのくらい電流を使うか、そして尚且つ、どのくらいの電圧安定度が必要かによります。
外部回路によっては、 PWM のアウトプットに op-amp (オペ-アンプ)などのバッファー(Buffer)回路が必要でしょう。

前述の回路を実際に BS2 のピン 0 につないで次のようなプログラムを走らせ電圧計(Digital multimeter)で測定してみましょう。

```
[ pause 1000    ' 1000 msec = 1 秒 休止 ]  pause インストラクションの数字を変えて電
```

圧がどのように変化するかを実験して下さい。

```
' 実験用プログラム
again:
        PWM 0, 100, 40    ' 出力ピン 0 から duty=100/255 で 40 サイクル
        pause 1000        ' 1000 msec = 1 秒間の休止
goto again                ' 繰り返す
```

(2) RCTIME （アールシータイム）

外部回路のＲＣのチャージ／ディスチャージの時間を計る

使い方

RCTIME pin,state,resultVariable

pin ； 測定する入力ピンを指定する

state ； 測定終了状態を示す(0 又は 1)

resultVariable ； 測定された値を保管する

【 解説 】

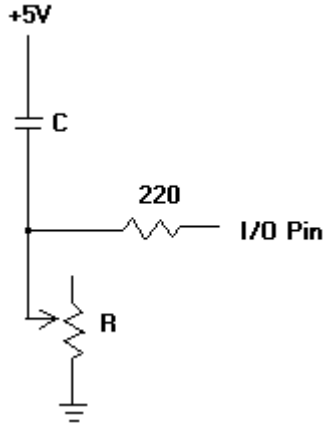
RCtime のインストラクションは抵抗とキャパシターの回路へのチャージ又はディスチャージの時間を計るためにあります。 抵抗やキャパシターを使ったセンサー(サーミスターや湿度センサー等)の値を計ったり、可変抵抗値を計ったりする事が出来ます。また、RCtime は 131 msec 以下の短い時間を計るストップウォッチの様な働きもします。

RCtime のインストラクションが実行されると 2 μsec 毎にカウンターの数字が増えていきます。 そして指定したピンが指定した状態(0 か 1)でなくなった時に、そのカウンターは止まります。 もしインストラクション実行時にピンが指定の状態でなければ RCtime は resultVariable に 1 を返します。それはインストラクションがその状態を知るのに 1 タイミングサイクル(1 timing cycle)を必要とするからです。
また、 もしピンが指定した状態を 65535 タイミングサイクル、即ち

2 μsec 毎ですから　　65535*0.000002=0.13107(sec)=131 (msec)

を超えるまで変わらなければ、 RCtime は 0 を resultVariable に返します。

RCtime インストラクションで使用される RC 回路は次のようにします。



注：220 Ωの抵抗は I/O ピンがグランドとの間でショート状態(R のスライダーの位置による)になるのでその保護の為にあります。 例えば、 R (可変抵抗)を零の位置に置き BS2 の I/O ピンをハイにした場合、 220 Ωの抵抗がなければ直接グラウンドに短絡して BS2 のアウトプットピンが燃える可能性があります。

回路図 RCtime 1

しかし 220Ωを直列に繋ぐことによって最大でも

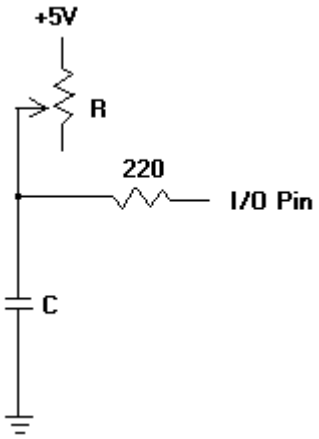
5 (V) / 220 (Ω) = 0.023 (A) = 23 mA

約 23 mA の電流しか流れません。 (実際には、 内部抵抗等がありますからもっと少ない電流になります。)

BS2 の論理的スレッシュホールドは約 1.5 V です。回路図 RCtime 1 において I/O ピンをハイ(1)にすると C と R と 220 Ωの接続点は 5 V (正確には 5 V より少し低い)になります。 この事は C がディスチャージした状態と全く同じです。

この様な状態から I/O ピンで電圧の変化(チャージの様子)を見ると 5 V から 1.5 V まで下がってきます(スパン[広がり]は 3.5 V)。 その時点で RCtime が止まります。 (指定したピンが指定した状態(0 か 1)でなくなった場合、 そのカウンターは止まります。)

次の回路で同じ事を考えてみましょう。



回路図 RCtime 2

回路図 RCtime 2でI/Oピンをロー(0)にすると C とR と220 Ωの接続点は0 V になります。 そして前と同じ様に RCtime のカウントが止まるまで、 電圧の変化を見てみますと 0 から 1.5 V までで、 ピンが指定した状態(0 か 1)でなくなる為、 RCtime は止まります。 この場合のスパンは 1.5 V だけということになります。

回路図 RCtime 1 のスパンは 3.5V、 回路図 RCtime 2 のスパンは 1.5V という事です。 前述しました様にRCtime のインストラクションは、 ピンが指定した状態(0 か 1)でなくなるまで 2 μsec のパルスでカウントする事です。

従って、 上の二つの回路のスパンをこのパルスでカウントすれば、 スパンが大きい程パルス数が多いのは当然です。 このような事を分解度(Resolution)が高い、と言います。 精度が高い事です。 従って、 上の二つの回路で RCtime を測定するなら、回路図 RCtime 1 のほうが精度の高い測定が出来るという事になります。

[プログラム例 1]
ピン 7 を使って回路図 RCtime 1 を接続し、RCtime を計ってみます。

```
result    var       word            ' result のワード宣言

          high 7                    ' ピン 7 をハイ(1)にする( C をディスチャージ)
          pause 1                   ' 1 msec 休止(C のディスチャージ時間)

          RCTIME    7, 1, result    ' RC のチャージ時間を計り結果を result に保
                                    '  管
          debug ? result            ' result をスクリーンに表示
```

上のプログラム例 1 で result にはどの様な数字が入るのかを考えてみましょう。

RCtime の値は RCtime constant (RCtime 定数)或は、τ(タウ)と呼ばれる値を使って計算する事が出来ます。 τは与えられた RC 回路の全体の電圧変化の 63% がチャージ或はディスチャージするのに必要な時間を示します。 τの値は一般に R の値と C の値を掛け合わせたものです。
(なお、ギリシャ文字 τの代わりに英語の t を使っても構いません。)

$$\tau = R \times C$$

一般に RC 回路においてある電圧からある電圧に変化するのに必要な時間は次のように計算する事が出来ます。

$$time = -\tau [\ln (V_{final} / V_{initial})]$$

ここに: ln は自然対数 (natural logarithm)
τは RCtime constant
Vfinal は指定した状態が変わる時の電圧
Vinitial は電圧の初期値

[例]

10 K の抵抗と 0.1 μF のキャパシターとして回路図 1 を考えると;

τ= (10 x 1000) x (0.1 / 1000000) = 1 / 1000 (sec) = 1 msec
従って、 RC time constant は 0.001 秒 又は 1 ミリ秒

ゆえに、 この RC 回路が 5 V から 1.5 V までにかかる時間は;

time = 0.001 x 1.204 = 0.001204 (sec)

RCtime のユニットは 2 μsec ですからそれで割ると;

0.001204 / 0.000002 = 602 (units)

(注: 上式の単位計算をすると [sec] / [sec] = 1 となり単位はないので
[units]で呼びます)

従って、 上の例では result に 602 という数字が入ります。

この例題で示されたように、 RC を回路図 RCtime 1 のようにつなげば、 Vinitial と Vfinal の電圧は変わらないのですから、 次のような略式計算が成り立ちます。
(上の計算では 602 units ですが計算しやすいように 600 units として扱います)

RCtime units = 600 x R (KΩ) x C (μF)

また、 RCtime インストラクションを使う前にチャージ/ディスチャージの時間をどの位必要かの目安は次の計算で出せます。

Charge time = 4 x R1 x C

ここに : R1 は I/O ピンからの抵抗値

従って、 回路図 1 においての Charge time は;

Charge time = 4 x 220 x 0.000001 = 88 / 1000000 (sec) = 88 (μsec)

ゆえに、 RCtime インストラクションを実行する前、ピン 7 をハイにした後 pause を 1 msec 準備すれば充分と言えます。 (前頁のプログラム例 1 を参照)
デモ用のプログラムを次に示します。

抵抗 R = 10 KΩPotentiometer (可変抵抗)、 C = 0.1 μF、 ピン 7 を使い回路図 RCtime 1 で result の値をスクリーン上に表示します。 可変抵抗の値を変化させてその値がどのように変わるかを調べて下さい。

【 Demo Program 】

```
result    var       word            ' Word variable to hold result

again:
          high 7                    ' Discharge the cap
          pause 1                   ' for 1 ms

          RCTIME 7,1,result         ' Measure RC charge time
          debug cls,dec result      ' Show value on screen

goto again
```

[7] SOUND　サウンド
音

(1) FREQOUT （フリックアウト）

0から32767ヘルツの正弦波を発生する

使い方

FREQOUT pin, duration, freq1{,freq2}
pin ； 使用する I/O pin 番号
duration ； ミリ秒で表したトーン幅
freq1 ； ヘルツで表した始めのトーン
freq2 ； オプション、 ヘルツで表した 2 番目のトーン

[解説]

Freqout は速い PWM を使って 1 種類或は 2 種類の正弦波を発生します。回路は Freqout のように PWM をフィルターにかけスピーカー或はオーディオアンプを通して音(トーン)を出します。

1 種類のトーンを出すインストラクションは次のようにします。

```
FREQOUT 2,1000,2500
```

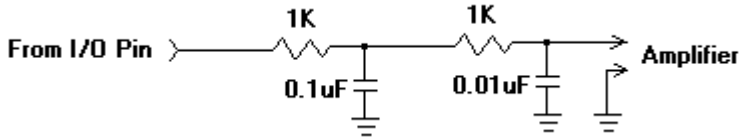
このインストラクションはピン 2 から 1 秒間(1000 msec)、 2500 ヘルツのトーンを出します。 もし、 2 種類のトーンを出したければ次のようにします。

```
FREQOUT 2,1000,2500,3000
```

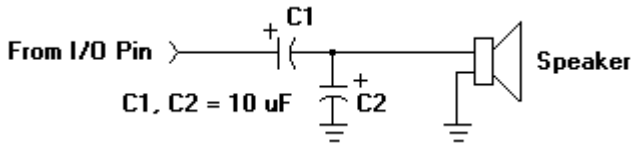
これは、ちょうど和音かベルのような音が出ます。 なお、音を出さない(休符)時は、周波数の値を 0 にします。

[周波数に関して]

回路図 Freqout 1と2 は正弦波を発生させる PWM の周波数を取り出す回路です。 Freqout は 0 から 32767 ヘルツの大変広い範囲の周波数で働きます。 必要に応じてキャパシターの値を換えるか Active filter を設計して使って下さい。



回路図 Freqout 1 [オーディオアンプに接続する時の回路]



回路図 Freqout 2 [スピーカーを使う場合]

【注意】:
1. 40Ω以上のインピーダンスを使用
2. 8Ωのスピーカーを使用する場合は 33Ωの抵抗を直列に入れる
3. スピーカーの変わりに Piezo Speaker を使う場合は C1 は必要ありません
4. C2 は高周波雑音を減らす為で、無くても構いません

Freqout インストラクションを使って "メリーさんの羊 (Mary had a little lamb) "という曲を Lookup Table を使ってプログラムしてみます。
(注: トーンの調子はキャパシター等の値によって少し違いがありますので、必要に応じて数字を適当に変えて正しいトーンを見つけて下さい)

なお、正弦波(Sine Wave)をミックスした効果を出す為に、第1の周波数に対して8ヘルツ少ない第 2 の周波数を混ぜてみます。 正弦波をミックスすると、その合計と差が発生します。その差の周波数は(このデモプログラムでは 8 ヘルツ)一つ一つの音の振動音(ビブラート(vibrato)或はクイーバ(quiver)と言います)になります。

Freqout インストラクションの freq は 16 ビットの中のビット 15(MSB)を無視します。従って、 最高発振周波数は 32767 ヘルツとなります。

なお、 休符の為に freq 1 を 0 にした時、 freq 2 = freq 1 - 8 >32767 となり雑音としてスピーカーから聞こえる可能性があります。 そこで、最高の値(MAX : binary operator の項を参照)を指定してそれを超えた場合は 0 を返すようにします。 0 が返れば、 真に休符(無音)になります。

```
' [ Demo Program ] : Mary had a little lamb (メリーさんの羊)

i       var     byte            ' Counter for position in tune.
f       var     word            ' Frequency of note for Freqout.

C       con     523             ' C note.
D       con     587             ' D note
E       con     659             ' E note
G       con     784             ' G note
R       con     0               ' Silent pause (rest).

        for i = 0 to 28         ' Play the 29 notes of the Lookup table.

lookup i,[E,D,C,D,E,E,E,R,D,D,D,R,E,G,G,R,E,D,C,D,E,E,E,E,D,D,E,D,C],f

        FREQOUT 0,350,f,(f-8) max 32768
        next

        stop
```

(2) DTMFOUT (ディーティーエムエフアウト)

DTMFの電話用トーンを発生する

(Dual Tone Multi Frequency)

使い方

DTMFout pin, {ontime,offtime,}{,tone....}

pin ; 使用する I/O pin 番号

ontime ; オプション、 ミリ秒で表したトーン幅

offtime ; オプション、 トーンの後のミリ秒で表したオフ幅

tone ; 0 から 9 番号 0 から 9 までを表す

10 * の印 (Star Mark)

11 # の印 (Pound Mark)

12 から 15 それぞれ A から D まで

[解説] DTMFOUT pin,{ontime,offtime,}[tone,tone,....]

Pin は 0 から 15 までのピン番号を指定します。このピンはトーンを出している間は、 臨時に output mode になり、 終ったらたとえこのインストラクションの前で output mode にしてあったとしても、 それを無視して input mode にセットします。

Ontime はオプションでトーンの幅を 0 から 65535 ミリ秒で指定出来ます。 もし、この指定がなければ DTMFout は 200 ms に自動的に指定します。 (default 200 ms)

Offtime はオプションでトーンとトーンの間の幅、 或は無音の長さを 0 から 65535 ミリ秒で指定出来ます。 もし、 この指定がなければ DTMFout は 50 ms に自動的に指定します。 (default 50 ms)

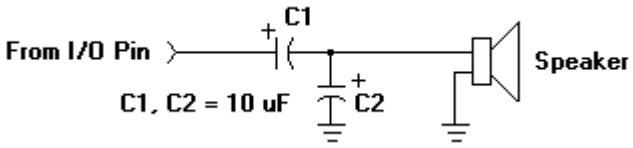
Tone は 0 から 15 までの変数或は定数で送り出すべく DTMF のトーンを指定します。トーンの 0 から 11 までは電話のタッチトーン キーパッドに合わされています。12 から 15 まで (Fourth column tones)は電話のテスト用機器やハムラジオ

(ham-radio) 等に利用されます。

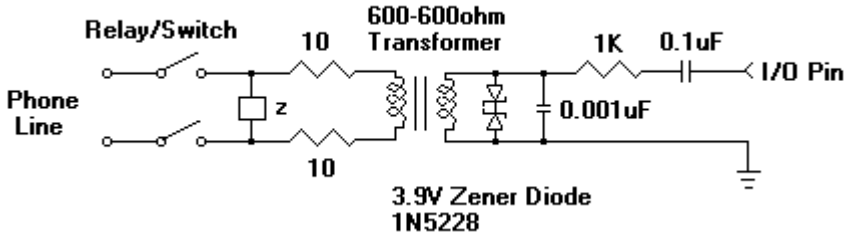
0 - 9	Digits 0 through 9
10	Star (*)
11	Pound (#)
12-15	Fourth column tones A through D (A から D に対応)

[説明]

DTMF のトーンはリモートコントロール機器とか電話のダイアル等に使われます。BS2 はそれらのトーンを DTMFout インストラクションでディジタル化して送る事が出来ます。 DTMFout 1 の回路でそのトーンを聞く事が出来ます。 また、 もし DTMFout 2 の回路を電話線に接続すればダイアルが出来ます。(注 : その国、地域、 自治体によって法的に接続が禁止、 或は制限がある場合がありますので、実際に接続する場合は電話局等に問合せて、 法的な問題及び使用電圧、 インピーダンス等を確認して下さい。)



回路図 DTMFout 1



回路図 DTMFout 2

DTMFout 2 の回路で電話番号 624-8333 にダイアルする場合は次のようにします。

```
DTMFOUT 0,[6,2,4,8,3,3,3]              ' Call 624-8333
```

もし、何かの理由(電話線のノイズ或はラジオリンク)でダイアルの間隔をゆっくりしたい場合はオプションの ontime と offtime の値を指定してやります。

```
DTMFOUT 0,500,100,[6,2,4,8,3,3,3]   ' ゆっくり 624-8333 をダイアル
```

[Demo Program]

このデモプログラムは記憶している電話番号をダイアルするものです。
DTMF の数字は 1 ニブル(nibble = 4 bits)に収まります。 従って、 次のプログラムの中の DTMF 数字(電話番号)は 3 つの EEPROM データテーブルのそれぞれのバイトに入れておきます。 なお、 電話番号の最後にあるニブル $F はダイアルの数字には無いものですから電話番号がここで終り、 という記号(Terminator ： ターミネイター)として使っています。

```
' Auto Dialer for 3 Telephone Numbers (3件の自動ダイアル)

EEloc     var     byte              ' 電話番号を保存した EEPROM address
EEbyte    var     byte              ' 2つの DTMF 用の番号を含んだ Byte
DTdigit   var     EEbyte.highnib    ' ダイアルするべく番号
phone     var     nib               ' Pick a phone #.
hiLo      var     bit               ' Upper と Lower nibble を選ぶ為のビット

Scott     data    $45,$94,$80,$2F           ' Phone: 459-4802
Chip      data    $19,$16,$62,$48,$33,$3F   ' Phone: 1-916-624-8333
Info      data    $15,$20,$55,$51,$21,$2F   ' Phone: 1-520-555-1212

for phone = 0 to 2                                  ' Dial each phone #.
        lookup    phone,[Scott,Chip,Info],EEloc     ' EEPROM にある番号の住
                                                    ' 所'を得て EEloc に書込む
```

```
dial:
        read    EEloc,EEbyte                    ' EEloc で示された住所の
                                                ' EEPROM 内容を読み取る.
        for hiLo = 0 to 1                       ' Dial upper and lower digits.
                if DTdigit = $F then done       ' $F はターミネイター
                DTMFout 0,[DTdigit]             ' Dial digit.
                EEbyte = EEbyte << 4            ' 4 bits 左にシフト
        next
                EEloc = EEloc+1                 ' 次の番号
goto dial                                       ' $F が来る迄続ける
done:                                           ' ひとつの電話番号の終り
                pause 2000                      ' Wait 2 seconds.
next                                            ' 次の電話番号
        stop                                    ' end of Program
```

[8] EEPROM ACCESS（イーイープロムアクセス）

イーイープロムの使用

(1) DATA （データ）

プログラムをダウンロードする前にデータを保管する

〔 EEPROM の(3) WRITE の項を参照して下さい 〕

(2) READ （リード）

指定した場所に EEPROM のデータをバイトで読み取る

使い方

READ location,variable

location ； 変数又は定数(0 から 2047) で指定した読み取る
eeprom の住所

variable ； eeprom から読んだバイトで表された値を保管する

[解説] READ location,variable

-Location は 0 から 2047 までの住所がありそこからデータを読む事が出来ます。
-Variable は EEPROM から読み取ったデータ(0 から 255 までの 1 バイト)を保管します。

プログラムはEEPROM に住所2047 から下方向に書き込んで行き、 データは住所 0 から上方向に書き込みます。 Read インストラクション は EEPROM のどの住所からでも 1 バイトのデータを読み出します。 EEPROM に書かれたデータは、たとえ電源がなくなっても失う事はありません。

[Demo Program]

このデモプログラムは EEPROM に文字列(Data Strings)を書き込んでそれを読むものです。 なお、 この EEPROM データはコンパイル タイム(Compile Time : プログラムを走らせる為に ALT-R を押した直ぐ後) に BS2 にダウンロードされて EEPROM に書かれ、別のデータがその上に書かれるまで 変わりません。

```
' ASCII の文字を EEPROM に書き込みます。 最後に 0 を付けてメッセージの終り
' 記号にします。

Message     data    "BS2 EEPROM Storage!",0
strAddr     var     word
char        var     byte
            strAddr = Message           ' Message の最初の住所
stringOut:
            READ StrAddr,char           ' EEPROM から最初の文字を読み char に入
                                        ' れる
            if char <> 0 then cont      ' char が 0 でなければ cont に行け
            Stop                        ' char が 0 なら終りの記号なので止まる
```

```
cont:
        debug char              ' char の内容 (メッセージ) を表示
        strAddr = strAddr+1     ' 次の文字の住所をポイントする。

goto stringOut                  ' ラベル名 stringOut に行け( 次の文字を読
                                ' む)
```

(3) WRITE (ライト)

EEPROM にバイトで書き込む

使い方

WRITE address,byte

address ; 変数又は定数で指定した EEPROM のアドレス(0 から 2047)

byte ; EEPROM に書き込むべくバイトで表したデータ

[解説] WRITE address,byte

EEPROM にデータを書き込むのは RAM に書き込むのとは幾つかの相違点があります。

(1) EEPROM に書き込むのは variable に値を保存するより時間がかかります。色々な要素があるので一概には言えませんが、 数ミリ秒もかかる事があります。 これに対し RAM へのそれは殆んど瞬間に出来ます。

(2) EEPROM への書き込み回数は有限で、凡そ 10,000,000 回位です。 書いたデータを読む事は無限です。 従って、もしプログラムが絶えず EEPROM にデータを書き込むようにしてある場合は気を付ける必要があります。 例えば、毎秒ひとつ何かを書けば、1 日では 86,400 回で 10,000,000 回は約 116 日で超えてしまいます。

(3) EEPROM の基本的な役目はプログラムを保存することです。 データはその余っ

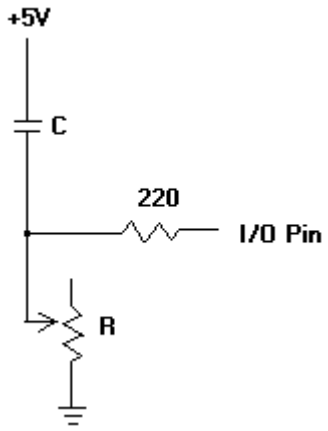
た場所に保存する事です。 もし、データの量が多過ぎるとプログラムの保存場所がなくなりそのプログラムはクラッシュ(Crash ： 破壊、 破滅)する事になります。 これを防ぐ為にはプログラム メモリーを時々チェックする必要があります。
また、次のような Data directive (データ ディレクテイブ) を使う事も出来ます。

name DATA (n)

Directive は name というアドレスで始まる EEPROM の n バイトを割り当てます。そして name + (n - 1) とアドレスを伸ばしていきます。 もしアドレスの範囲に対して Writes に制限を付ければ、スペースの問題はあまりないでしょう。 もしプログラムが大きくなってアドレスの重複が起きれば、エラーメッセージが出てプログラムのダウンロードは拒否されます。

[Demo Program]

このプログラムはデータの変化を記録する為の基本的な枠組です。 データを集めて保存し、 後でそれらのデータを使うような事に応用出来ます。 集めるデータ (sample data とここでは呼びます) を RCtime の項で使った回路をピン 7 に繋ぎます。



R = 10k Potentiometer (可変抵抗)
C = 0.1 µF capacitor

プログラムを走らせ可変抵抗値を変えてデータを集めます。 プログラムは 1 秒毎にそのデータを読み EEPROM に書き込んでから、 それを EEPROM から読み出します。

もし、これに似た WRITE インストラクションを使ったプログラムを組む場合は、データに使う EEPROM のスペースに気を付けて下さい。 定数を使って EEPROM のスペースを初めから終りまで監視すると良いでしょう。

(特記 ： このデモプログラムは EEPROM のアドレス用に不必要な程の大きなスペースを取っています。 EEPROM のアドレスを 0 から始めて 10 samples だけですから、ニブルで足りるはずですが、データ sample を簡単に増やせる様に ワード にしてあります。)

```
result      var     word            ' Word variable for RCtime result
EEaddr    var     word            ' Address of EEPROM storage locations
samples    con     10              ' Number of samples to get
log         data    (samples)       ' Set aside EEPROM for samples
endLog     con     log+samples-1   ' End of allocated EEPROM

for EEaddr = log to endLog          ' Store each sample in EEPROM

          high 7：pause 1           ' キャパシターにチャージ
          rctime 7,1,result         ' 抵抗を測る
          result = result*42/100    ' 1バイトに収まるようにする(0-255)

          debug "Storing ", dec result,tab," at ", dec EEaddr,cr
                                    ' スクリーンに表示
                    WRITE EEaddr,result     ' EEPROM に書き込む
          pause 1000                ' Wait a second
next                                ' 全てのサンプルを読む
          pause 2000：debug cls     ' Wait 2 seconds, then clear screen.

for EEaddr = log to endLog          ' 各々のサンプルを引き出す
                    read EEaddr,result      ' Read back a byte
          debug "Reading ", dec result,tab," at ", dec EEaddr,cr
                                    ' スクリーンに表示
next                                ' 全てのサンプルを引き出す
          stop
```

[9] TIME タイム

遅延などに使う時間

(1) PAUSE (ポーズ)

0から65536ミリ秒までの静止が可能

使い方

PAUSE milliseconds

milliseconds ; ミリ秒で表した時間で一時的に動作を止める

プログラム例

ピン 0 を 100 msec ローにして、それから 100 msec ハイにする事を繰り返します。
(LED を抵抗と共につなげば、点滅します。)

```
flash:
        low  7
        PAUSE 100
        high  7
        PAUSE 100

goto flash
```

[10] POWER CONTROL パワーコントロール

ベーシック･スタンプの電力制御

(1) NAP (ナップ)

18ミリ秒から2304ミリ秒までの短時間スリープ用

NAP period

period ; 0 から 7 までの数字で表された短時間スリープ
(18+period x 18 ms) (次頁を参照)

[解説]

NAP period は短時間スリープ(休止)用に使用します。 この期間は外部回路がない場合で 50 μA の電流しか使いませんので最低消費量です。
Period は変数か定数で NAP の期間(Duration)を指定します。 この期間は次のように計算され、 次頁の表に示すように 0 から 7 までの数字で指定出来るようになっています。

Duration = (2^period) * 18 ms

上の式の読み方は
(2 の period 乗 掛ける 18 milliseconds)
英語での読み方は
(2 raised to the power period, times 18 milliseconds)

Period	2^period	NAP duration
0	1	18 ms
1	2	36 ms
2	4	72 ms
3	8	144 ms
4	16	288 ms
5	32	576 ms
6	64	1152 ms (1.152 sec)
7	128	2304 ms (2.304 sec)

(2) SLEEP (スリープ)

2.3 ～ 65535秒までのスリープができる

SLEEP seconds
seconds ; 1 から 65535 までの変数又は定数で表したスリープ

[解説]

スリープは BS2 自身をオフにして指定された時間後にオンにするものです。
このスリープの長さは、 約 2.3 秒から最高約 18 時間まで指定出来ます。 スリープの間は外部回路がなければ、 約 50 μA と最低消費量です。 スリープ インストラクションの分解度(resolution)は 2.304 秒です。 これは、 指定されたスリープ時間を超え尚且つ最も近いように 2.304 の整数倍して出す、 と言うことです。
例えば、 Sleep 1 は 2.3 秒のスリープ、 Sleep 10 は 2.304 x 5 = 11.52 秒 という事になります。

(3) END （エンド）

リセットするかコンピューターを接続するまでスリープする

END

BS2 のマイクロプロセッサーをオフの最小消費電力の状態にする

[解説]

End は BS2 を無活動の状態にします。 外部回路がなければ 約 50 μA の最小消費電流になります。 リセットボタンを押すか電源をオフにした後、オンにすると活動し始めます。

[11] PROGRAM DEBUGGING ディバッギング

プログラムディバッギング

DEBUG (ディバッグ)

コンピューター上でプログラムの動作を見る

DEBUG outputData{,outputData...}

outputData ； テキスト文字,変数,定数等々を見る

[解説]　DEBUG outputData{,outputData...}

　　変数やメッセージ等を PC スクリーン上に表示します。 OutputData は次の中のひとつかそれ以上を指定出来ます。

text strings (メッセージ等の文字列)
variables (変数)
constants (定数)
expressions (数式 など)
formatting modifiers (特別なモディファイアー)
control characters (コントロール記号)

Debug はプログラムを走らせている時、 PC スクリーン上にメッセージ等を表示させプログラムをチェックしたりするのにとても役にたちます。 Debug はインストラクションの効果(正しいかどうか)を即座にフィードバック(feedback)してくれます。

例 1

```
DEBUG "Hello World!"                    ' Test message.
```

とプログラムを書き、 ALT-R を押しプログラムを BS2 にダウンロードすると、 PC スクリーン上に Debug 用のウインドウが開き、 Hello World! と表示されます。 なお、コンピューターのキーボードのスペースバーを除くキーのどれかを押すと、 Debug 用のウインドウは消えます。 しかしプログラムは止まってはいません。

例 2

```
x       var     byte:  x = 65
        DEBUG dec x                     ' Show decimal value of x
```

x = 65 ですから Debug window は 65 と表示されます。 この数字は十進数ですがヘックス(hexadecimal) や二進数 (binary) でも表示出来ます。 １０５頁のテーブル(Table Debug Modifiers) を参照して下さい。

例 3

プログラム中で幾つかの Debug インストラクションを用意して何種類かの変数等を見たい時、どの変数が何かを知りたい事があります。

```
x           var         byte：  x = 65

            DEBUG dec ? x                   ' Show decimal value of x with label "x ="
```

この Debug は x = 65 と表示します。

例 4

Debug は次のような expressions も表示します。

```
x           var         byte：  x = 65

            DEBUG dec ? 2*(x-1)         ' Show decimal result with "2*(x-1) = "
```

注：Debug window は 2*(x-1) = 128 と表示します。もし? のマークがなければ、
単に 128 と表示されます。

例 5

もし数字の構成を指定しないで Debug の表示をさせると、 その値の ASCII 文字が表示されます。

```
x           var         byte：  x = 65
            DEBUG x                                 ' Show x as ASCII.
```

数字の 65 は ASCII コードでは大文字の A にあたりますので Debug window には A が表示されます。

例 6

Debug は一つの事を表示させるだけでなくカンマで区切れば幾つかを表示させる事が出来ます。

```
x       var     byte: x = 65
        DEBUG "The ASCII code for A is: ", dec x
                                        ' Show phrase, x.
```

例 7

色々な Debug instruction でスクリーン上に表示させると、スクリーンが見にくくなって来ます。

そこで次のようなシンボルを Debug instruction に入れたりすると効果的です。

Symbol (シンボル)	Value (数値)	Effect (効果)
CLS	0	Debug screen をクリアー
HOME	1	カーサーをホームポジション(左上)に置く
BELL	7	PC のスピーカーで ビー と音を出す
BKSP	8	スペースを一つ戻る
TAB	9	タブ (8 text column)
CR	13	次のラインの始めに戻る (carriage return)

例 8

CR を入れた Debug とそれを入れないものとを次の Debug で確かめて下さい。

```
        Debug:
                Debug "A carriage return",CR
                Debug "starts a new line"
        goto Debug
```

Table Debug Modifiers

Modifier	Effect	Notes
ASC?	Displays "variablename = 'character'" + carriage return; where character is an ASCII character.	1
DEC {1..5}	Decimal text, optionally fixed for 1 to 5 digits	
SDEC {1..5}	Signed decimal text, optionally fixed for 1 to 5 digits	1, 2
HEX {1..4}	Hexadecimal text, optionally fixed for 1 to 4 digits	1
SHEX {1..4}	Signed hex text, optionally fixed for 1 to 4 digits	1, 2
IHEX {1..4}	Indicated hex text ($ prefix; e.g., $7A3), optionally fixed for 1 to 4 digits	1
ISHEX {1..4}	Indicated signed hex text, optionally fixed for 1 to4 digits	1, 2
BIN {1..16}	Binary text, optionally fixed for 1 to 16 digits	1
SBIN {1..16}	Signed binary text, optionally fixed for 1 to 16 digits	1, 2
IBIN {1..16}	Indicated binary text (% prefix; e.g., %10101100), optionally fixed for 1 to 16 digits	1, 2
ISBIN {1..16}	Indicated signed binary text, optionally fixed for 1 to16 digits	1, 2
STR bytearray	Display an ASCII string from bytearray until byte = 0.	
STR bytearray＼n	Display an ASCII string consisting of n bytes from bytearray.	
REP byte＼n	Display an ASCII string consisting of byte repeated n times (e.g., REP "X"＼10 sends XXXXXXXXXX).	

NOTES:

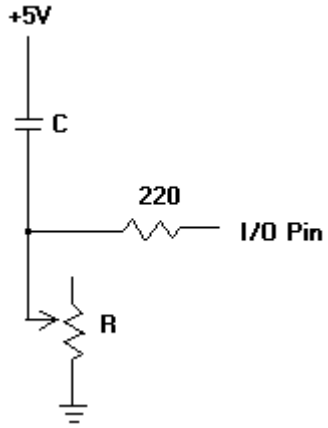
(1) DEC4 のような数字を固定する modifiers は0 (零) で不足を補います;
例. DEC4 65 は 0065 として扱う。もし､数字が指定した数より大きい場合は
上位の桁を落とします;例. DEC4 56422 は 6422 を送り出します。

(2) サインのついた modifiers は 2's complement のルールで働きます。
値は word variable のサイズより小さくてはいけません。

『 BS 2 プログラム例 』

1. Test_1

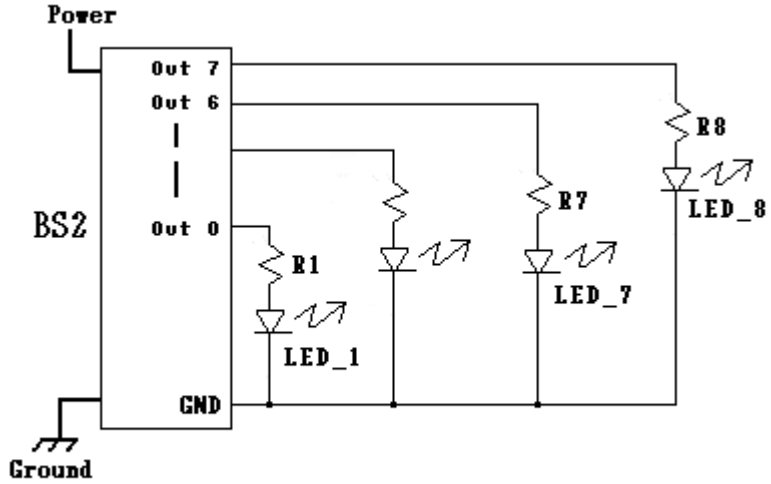
可変抵抗(Potentiometer)をピン 14 に接続してその変化をスクリーン上で見る。
回路は次のようにします。なお、C＝0.1 μF、R＝10KΩ可変抵抗です。




```
'REM test_1 Potentiometer measurement  ( REM は remark の意味)

pot     var     word            'pot のワード宣言

        loop:   high 14         'ピン 14
                pause 1         '1 msec 休止
                rctime 14,1,pot 'データをワード pot にセーブ
                debug ?pot      'スクリーンに pot のデータを表示

        goto loop               'ラベル名ループに行け
```

2. Test_2

次の回路で 8 個の LED の中の一つだけが ON になり、それが順番に移っていくプログラムです。R1 から R8 までの抵抗値は LED に約 10mA 位の電流を流すように、計算してください。

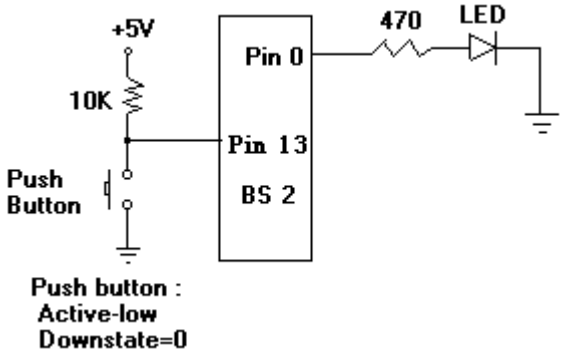


ソース回路

```
'REM test_2 Program for a LED ON at a time

loop:
        for b2 = 0 to 7
                high b2             'output = 1 ( LED ON )
                pause 100           '100 msec 休止
                low b2              'output = 0 ( LED OFF )
                pause 100
        next                        '次の b2 ( LED )
goto loop
```

3. Test_3

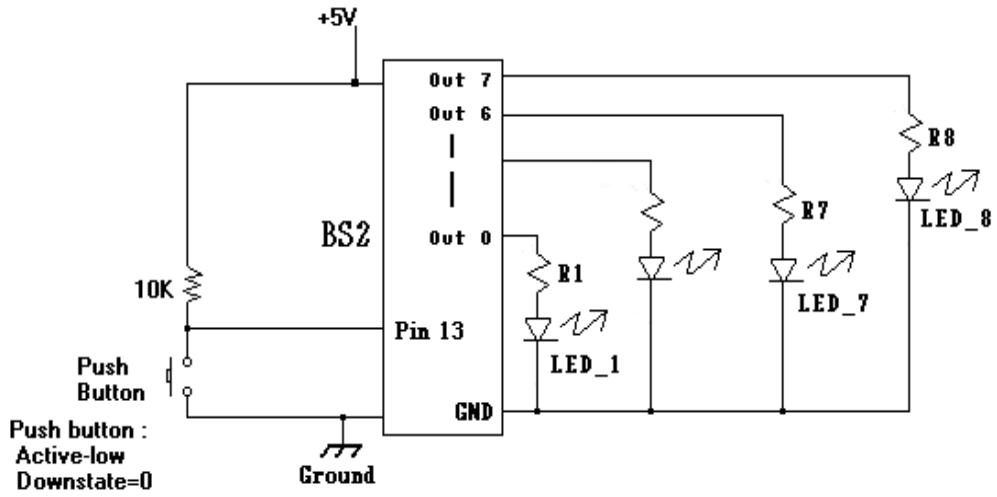
プッシュボタンをピン 13 に接続、ピン 0 に抵抗を通して LED を接続してインストラクション 『button』の働きをみる。




```
'REM Test_3   An LED toggles by push button

b2 = 0                      'button インストラクションで使用する前にクリアーする

loop:
        button 13,0,200,100,b2,0,skip_1
                            '命令言語 BUTTON の項を参照して下さい

        toggle 0            'ピン 0 の出力を 1 だったら 0 に、0 だったら 1 にする

skip_1:
        goto loop           'ラベル名 loop に行け
```

4. Test_4

Test_2 の 8-LED 回路と Test_3 のプッシュボタンを組み合わせたプログラム。
プッシュボタンを押すと LED が一つだけ点灯して順番に移っていきます。




```
'REM Test_4 Program to start LED when a push button is pushed

b2 = 0                              'クリアー (Pre-defined byte variable)

        loop:   button 13,0,200,100,b2,0,loop
                                    'ピン 13 にスイッチボタン接続

lamp:   for b3 = 0 to 7             '0 から 7 までのアウトプットピン
                high b3             'output を 1 にする(LED ON)
                pause 100           '100 msec 休止
                low b3              'output を 0 にする(LED OFF)
                next                '次の LED

        goto lamp                   '8 番目がオフになったらまた 0 から始める
```

5. Test_5

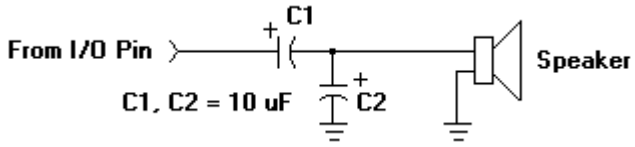
十進数と二進数の関係を並べてスクリーンに表示させる。

```
'REM Test_5   Decimal and Binary Number

loop:   for b2 = 0 to 255           '十進数の 0 から 255 まで
        pause 5                     '5 msec 休止(表示の速さを調整)

        debug "Decimal= ",dec b2," Binary= ",bin b2,cr
                                    'cr は carriage return の意味

        next                        '次の十進数
        sleep 1                     '255 が終ったら約 1 秒停止

        end                         'プログラム終了、リセットするかプログラムをダ
                                    'ウンロードするまで止まったままになる
```

(これが終ったら「 Binary と Hexadecimal 」の関係を表示させるプログラムを作って下さい。)

6. Test_6

ピン 15 に小さいスピーカーを接続して音を出す。
*音階が合わない場合は、数字を変えて下さい。
スピーカーのインピーダンスは40 Ωを使って下さい。8 Ωの場合は33 Ωの抵抗を直列につないで下さい。




```
'REM Test_6   Do, Re, Mi   Notes
loop:
do:       freqout 15,500,400          'output sound from pin 15
          pause 100
re:       freqout 15,500,450
          pause 100
mi:       freqout 15,500,500
          pause 100
fa:       freqout 15,500,520
          pause 100
so:       freqout 15,500,580
          pause 100
la:       freqout 15,500,650
          pause 100
ci:       freqout 15,500,725
          pause 100
do_h:     freqout 15,500,760
          pause 100
          sleep 10                    'about 10 second sleep
goto loop
```

7. Test_7

Test_6 で作った音階を使って『たきび』(Bonfire) のメロディーを作る。
ピン 13 に接続したプッシュボタンを押すとスタートします。(Test_3 のように
プッシュボタンをつないで下さい。)
＊ページを節約する為に 2 段組にしてありますのでご了承ください。

```
'REM Test_7     Melody of TAKIBI (Bonfire)
b2 = 0                                  'clear
start:     button 13,0,200,100,b2,0,start          'スタートボタンが押されるのを待つ

freqout 15,500,580          'so
pause 100
freqout 15,500,650          'la
pause 100
freqout 15,500,580          'so
pause 100
freqout 15,500,500          'mi
pause 100

freqout 15,500,580          'so
pause 100
freqout 15,500,650          'la
pause 100
freqout 15,500,580          'so
pause 100
freqout 15,500,500          'mi
pause 100

freqout 15,500,400          'do
pause 100
freqout 15,500,450          're
pause 100
freqout 15,500,500          'mi
pause 100
freqout 15,500,500          'mi
pause 100
freqout 15,1000,450         're
pause 1000

freqout 15,500,500          'mi
pause 100
freqout 15,500,580          'so
pause 100
freqout 15,500,580          'so
pause 100
freqout 15,500,580          'so
pause 100

freqout 15,500,650          'la
pause 100
freqout 15,500,760          'do_h
pause 100
```

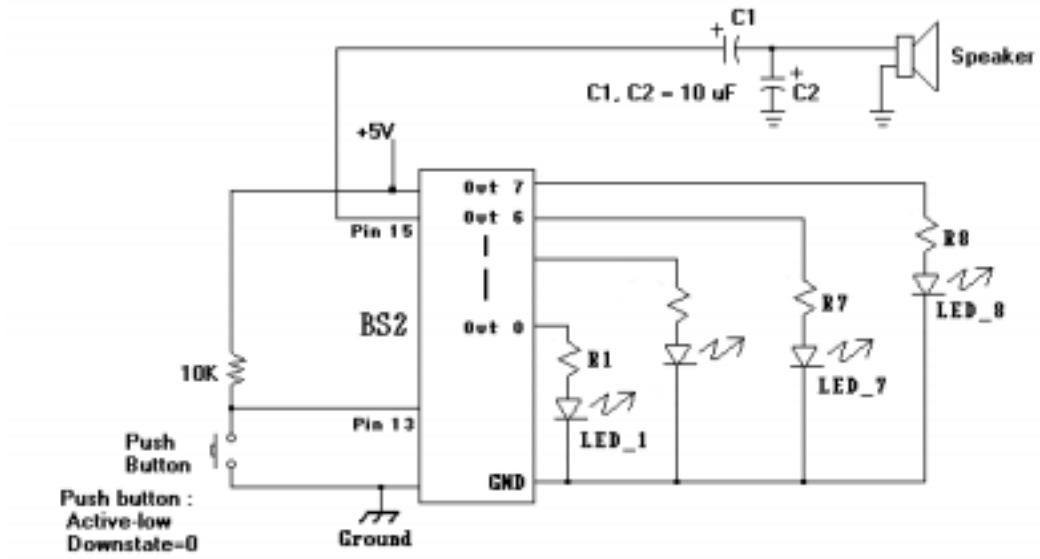
```
freqout 15,500,760      'do_h
pause 100
freqout 15,500,760      'do_h
pause 100

freqout 15,500,580      'so
pause 100
freqout 15,500,650      'la
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,500,450      're
pause 100
freqout 15,1000,400     'do

pause 1000

freqout 15,1500,450     're
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,500,520      'fa
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,1000,450     're
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,500,580      'so
pause 100
freqout 15,500,580      'so
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,1000,580     'so
pause 1000

freqout 15,500,760      'do_h
pause 100
freqout 15,500,760      'do_h
pause 100
freqout 15,500,760      'do_h
pause 100
freqout 15,500,650      'la
pause 100

freqout 15,500,580      'so
pause 100
freqout 15,500,580      'so
pause 100
freqout 15,500,760      'do_h
pause 100
freqout 15,500,760      'do_h
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,500,500      'mi
pause 100
freqout 15,500,450      're
pause 100
freqout 15,500,450      're
```

```
pause 100
freqout 15,1000,400          'do
pause 1000
```

```
goto     start
         'プッシュボタンに戻る
```

8.　Test_8

Test_6 のド、レ、ミ、ノートに手を加え音階一つ一つに LED を接続し、音階の出力に "gosub routine" を取り入れ、プッシュボタンで開始するようにしたプログラム。




```
'REM Test_8    Do, Re, Mi, Notes with LED lamps

note       var        word
b2 =0

start:     button 13,0,200,100,b2,0,start
```

```
b2 = 0
note = 400          'do
gosub sound

b2 = 1
note = 450          're
gosub sound

b2 = 2
note = 500          'mi
gosub sound

b2 = 3
note = 520          'fa
gosub sound

b2 = 4
note = 580          'so
gosub sound

b2 = 5
note = 650          'la
gosub sound

b2 = 6
note = 720          'ci
gosub sound

b2 = 7
note = 760          'do_h
gosub sound
goto start          'プッシュボタンに戻る

sound:              'サブルーチン
high b2             'LED オン
freqout 15,500,note
                    'pin 15 スピーカー
low b2              'LED オフ
pause 100

return              'サブルーチンに来
                    'た場所+1 に帰る
```

9. Test_9 （Test_8 の回路をそのまま使用します。）

『うさぎ うさぎ』の曲を音階と LED を同調させたプログラム。（Gosub routine は使いません） *なお、例によって 2 段組で書きます。

```
'REM Test_9     Usagi Usagi with LED lamp

b2 = 0                              'LED 用
b3 = 0                              'プッシュボタン用

switch:
        button 13,0,100,100,b3,0,switch
b2 = 4                              pause 250
high b2                             low b2
freqout 15,500,520          'fa     b2 = 6
pause 500                           high b2
low b2                              freqout 15,500,650          'la
                                    pause 250
b2 = 4                              low b2
high b2                             b2 = 7
freqout 15,500,520          'fa     high b2
pause 250                           freqout 15,500,725          'ci
low b2                              pause 500
b2 = 6                              low b2
high b2                             pause 250
freqout 15,500,650          'la
pause 250                           b2 = 4
low b2                              high b2
b2 = 7                              freqout 15,500,520          'fa
high b2                             pause 250
freqout 15,500,725          'ci     low b2
```

```
b2 = 4
high b2
freqout 15,500,520          'fa
pause 250
low b2
b2 = 4
high b2
freqout 15,500,520          'fa
pause 250
low b2
b2 = 6
high b2
freqout 15,500,650          'la
pause 250
low b2
b2 = 7
high b2
freqout 15,500,725          'ci
pause 250
low b2
b2 = 6
high b2
freqout 15,500,650          'la
pause 250
low b2
b2 = 7
high b2
freqout 15,500,725          'ci
pause 500
low b2
pause 250

b2 = 6
high b2
freqout 15,500,650          'la
pause 250
low b2
b2 = 7
high b2
freqout 15,500,725          'ci
pause 250
low b2
b2 = 8
high b2
freqout 15,500,760          'do_h
pause 250
low b2
b2 = 8
high b2
freqout 15,500,760          'do_h
pause 250
low b2
b2 = 7
high b2
freqout 15,500,725          'ci
pause 250
low b2
b2 = 6
high b2
freqout 15,500,650          'la
```

```
pause 100
low b2
b2 = 6
high b2
freqout 15,500,650          'la
pause 100
low b2
b2 = 4
high b2
freqout 15,500,520          'fa
pause 500
low b2
b2 = 3
high b2
freqout 15,500,500          'mi
pause 250
low b2

b2 = 6
high b2
freqout 15,500,650          'la
pause 500
low b2
b2 = 4
high b2
freqout 15,500,520          'fa
pause 500
low b2
b2 = 3
high b2
freqout 15,500,500          'mi
pause 500
low b2
b2 = 4
high b2
freqout 15,500,520          'fa
pause 100
low b2
b2 = 3
high b2
freqout 15,500,500          'mi
pause 100
low b2
b2 = 2
high b2
freqout 15,500,450          're
pause 500
low b2
b2 = 3
high b2
freqout 15,500,500          'mi
pause 1000
low b2

sleep 1

goto switch


プログラム例の終り
```

巻末 Reserved Words (Pre-defined Words)

この表は、BASIC Stamp 2（BS2）のあらかじめリザーブされた言葉（Reserved Words 又は Pre-Defined Words）です。これらのワードを、例えば、ラベル名などに使う事は出来ません。（使った場合は、エラーとなります。）

Reserved Words

BS2		
ABS	HOME	OUTL
AND	IHEX	OUTPUT
ASC	IHEX1..IHEX4	OUTS
B0..B25	IF	PAUSE
BELL	IN0..IN15	RCTIME
BKSP	INA	REV
BIN	INB	PULSIN
BIN1..BIN4	INC	PULSOUT
BIT	IND	PWM
BIT0..BIT15	INH	RANDOM
BRANCH	INL	READ
BRIGHT	INPUT	REP
BUTTON	INS	REVERSE
BYTE	ISBIN	SBIN
CLS	ISBIN1..ISBIN16	SBIN1..SBIN16
CON	ISHEX	SDEC
COS	ISHEX1..ISHEX4	SDEC1..SDEC5
COUNT	LIGHTSON	SERIN
CR	LOOKDOWN	SEROUT
DATA	LOOKUP	SHEX
DCD	LOW	SHEX1..SHEX4
DEBUG	LOWBIT	SHIFTIN
DEC	LOWNIB	SHIFTOUT
DEC1..DEC5	LSBFIRST	SIN
DIG	LSBPOST	SKIP
DIV	LSBPRE	SLEEP
DIR0..DIR15	MAX	STEP
DIRA	MIN	STOP
DIRB	MSBFIRST	STR
DIRC	MSBPOST	SQR
DIRD	MSBPRE	TAB
DIRH	NAP	THEN
DIRL	NCD	TO
DIRS	NEXT	TOGGLE
DTMFOUT	NIB	UNITOFF
END	NIB0..NIB3	UNITON
FOR	NOT	UNITSOFF
FREQOUT	NUM	VAR
GOSUB	OR	W0..W12
GOTO	OUT0..OUT15	WAIT
HEX	OUTA	WAITSTR
HEX1..HEX4	OUTB	WORD
HIGH	OUTC	WRITE
HIGHBIT	OUTD	XOR
HIGHNIB	OUTH	XOUT

索　引

Acknowledgment

本書【Ｐ－ベーシック言語】は、米国の Parallax Inc. から許可
を頂き関本清志が、翻訳及び編集をしたものです。翻訳に際し、Parallax Inc.
の Mr. Ken Gracey から多大な協力を戴きました事をここに感謝致します。

Ｐ－ベーシック言語

平成 14 年 10 月　初版発行
平成 16 年 12 月　改定（v2）

翻訳者　関本清志
発行者　関本清志
発行所　日本マイクロボット教育社

印刷所
製本所

お薦めのもの（詳細・購入はウェブページ www.microbot-ed.com でどうぞ）

『通信講座ロボット製作入門』

ロボットの基本から製作までを学べます。

『通信講座トッドラー製作』（ロボット製作中級講座）

ロボット製作入門が終わった方、P－BASIC の基本知識がある方用です。

ロボット入門用『ボーボット』

中級者用 2 足歩行『トッドラー』

日本マイクロボット教育社