Parallax BASIC Stamp® Tutorial

Developed by:

Electronic Systems Technologies
College of Applied Sciences and Arts
Southern Illinois University Carbondale
http://www.siu.edu/~imsasa/est

Martin Hebel
mhebel@siu.edu
With support from:
Will Devenport, Mike Palic and Mike Sinno

Sponsored by:

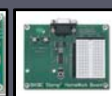
Parallax, Inc.
http://www.parallax.com/

Updated: 3/14/03 Version 1.0

Copyright Notice

Use of the Tutorial

- This tutorial is for novices in programming the BS2 from Parallax, Inc. For advanced use please refer to your BASIC Stamp Manual, help files and other sources.
- The tutorial uses the Board of Education (BOE) as the primary carrier board for the BS2, though other boards and configurations may also be used.
- The majority of the tutorial is compatible with the HomeWork board and the BASIC Stamp Activity Board except where noted.
- We welcome any constructive feedback you wish to provide. A feedback page and survey are located at: http://imsinet.casa.siu.edu/bs2_tutorial/feedback.htm If this link is no longer active, please contact Parallax at stampsinclass@parallax.com.



Parts Required

- This tutorial was written to use a minimum number of inexpensive components as possible in teaching the basic principles.
- The following are recommended:
  - A BASIC Stamp 2 and a carrier board, such as the BOE, HomeWork Board, Activity Board or NX-1000, cables and software.

Parts for Sections 4-7:
(3) 220 Ohm Resistors
(2) LEDs
(2) N.O. Momentary Pushbuttons
(2) 1K Ohm Resistors
(1) 0.1 microfarad capacitor
(1) 100K Ohm Potentiometer
(1) Piezoelectric Speaker

Additional Parts for Sections 8,9
(1) 10K Ohm Resistor
(1) 10uF Capacitor
(1) ADC0831
(1) LM34 Temperature Sensor

Contents

Home | Schematics Sec. 4-7
Links | Schematics Sec. 8

Section 1: BASIC Stamp 2 Anatomy

- Microcontrollers
- BASIC Stamp Components
- BASIC Stamp 2 Pins
- BASIC Stamp 2 Versions
- Running a Program
- Carrier and Experiment Boards
- Power Connections
- Data Connections
- Serial Data Connectors
- I/O Connections
- Component Power Connections
- Connecting Components
- Breadboard Connections
- Other Features

Microcontrollers

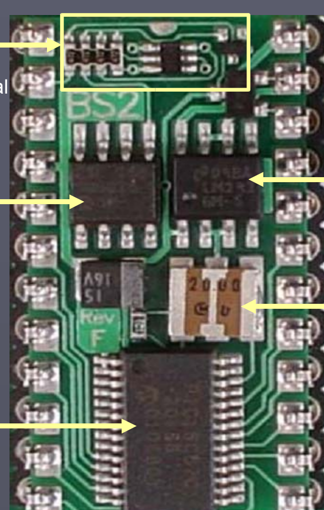
- Microcontrollers can be thought of as very small computers which may be programmed to control systems such as cell phones, microwave ovens, toys, automotive systems, etc.
- A typical household has upwards of 25 to 50 microcontrollers performing *embedded control* in numerous appliances and devices.
- The BASIC Stamps are hybrid microcontrollers which are designed to be programmed in a version of the *BASIC* programming language called *PBASIC*.
- Hardware support on the module allows fast, easy programming and use.

BASIC Stamp Module Components

Serial Signal Conditioning
Conditions voltage signals between PC serial connection (+/- 12V) and BASIC Stamp (5V)

EEPROM
Stores the tokenized PBASIC program.

Interpreter Chip
Reads the BASIC program from the EEPROM and executes the instructions.



5V Regulator
Regulates voltage to 5V with a supply of 5.5VDC to 15VDC

Resonator
Sets the speed at which instructions are processed.

BASIC Stamp 2 Pins

Pin 1: S_{OUT}

Transmits serial data during programming and using the DEBUG instruction

Pin 2: S_{IN}

Receives serial data during programming

Pin 3: ATN

Uses the serial DTR line to gain the Stamps attention for programming.

Pin 4. V_{SS}

Communications Ground (0V).

Pins 5-20:

Input/Output (I/O) pins P0 through P15

Pin 24. V_{IN}

Un-regulated input voltage (5.5-15V)

Pin 23. V_{SS}

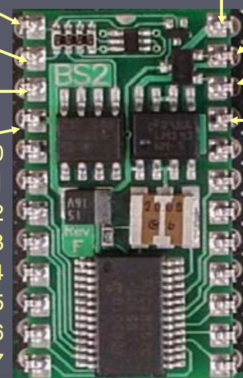
Ground (0V)

Pin 22. RES

Reset- LOW to reset

Pin 21. V_{DD}

Regulated 5V.



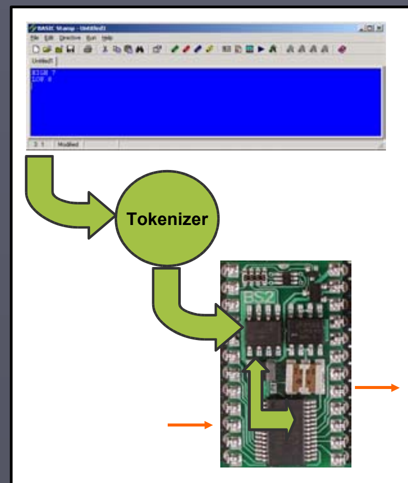
BASIC Stamp 2 Versions

- There are several versions of the BASIC Stamp. This tutorial has been written for the BASIC Stamp 2 (BS2) series of controllers.
- Each BASIC Stamp has different features, below are the most popular:

	Version	Memory	Speed	Additional Features
	BS2	2K Bytes 500 lines of code	20MHz 4000 instructions/ second	26 Bytes of RAM
	BS2 OEM	2K Bytes 500 lines of code	20MHz 4000 instructions/ second	26 Bytes of RAM Less expensive, easy to replace components.
	BS2sx	16K Bytes in 8 2K banks. 4000 lines of code	50MHz 10,000 instructions/ second	26 Bytes of RAM 63 bytes of scratchpad memory
	BS2p 24 and 40 pins versions	16K Bytes in 8 2K banks. 4000 lines of code.	20 MHz Turbo	I2C, Dallas 1- Wire, LCD, polling capabilities. 16 extra I/O on 40 pin version.

Running a Program

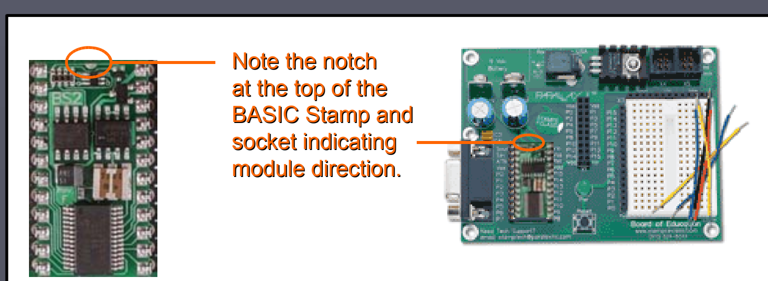
- A program is written
in the BASIC Stamp Editor
- The program is tokenized,
or converted into symbolic format.
- The tokenized program is
transmitter through the serial cable
and stored in EEPROM memory.
- The Interpreter Chip reads the
program from EEPROM and
executes the instructions reading
and controlling I/O pins. The
program will remain in EEPROM
indefinitely with or without power
applied.



Carrier and Experiment Boards

- The user may engineer their own power, communications and control circuits for the BASIC Stamp, but for beginners an assortment of carrier and experimenter boards are available for ease of development and testing.

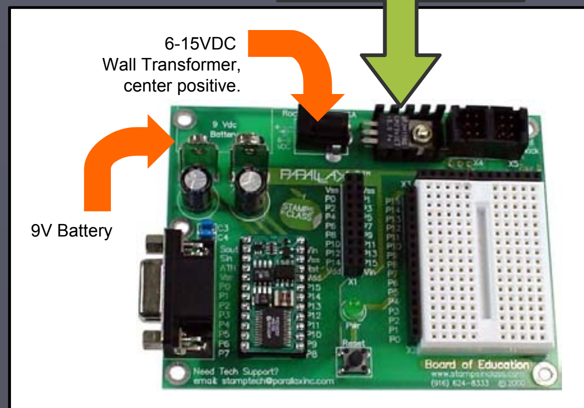
- The Board of Education (BOE) is one such board and will be the focus for this tutorial.



Power Connections

- The Board of Education may be powered from either:

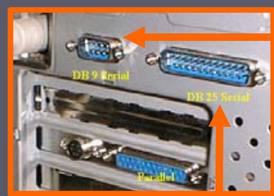
Many carrier boards, such as the BOE, have an additional 5V regulator to supplement the on-module regulator.



Data Connections

- A serial cable (modem cable) is connected between BASIC Stamp and the computer's serial communication port (COM port).
  - Serial means that data is sent or received one bit at a time.
  - The serial cable is used to download the stamp with the program written in the text editor and is sometimes used to display information from the BASIC Stamp using the DEBUG instruction.
  - Ensure that you use are using a *Straight-Through* cable (pins 2 and 3 do not cross from end-to-end) as opposed to a *Null-Modem* cable (pins 2 and 3 cross).
  - There are different connectors for different computer hardware.

Serial Data Connectors



The cable is typically connected to an available DB 9 COM port.

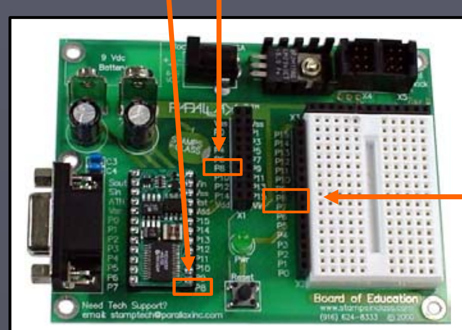
A DB 25 to DB 9 adapter may be needed on older systems

Newer systems may only have USB ports and require a USB-to-Serial Adapter.

I/O Connections

- Code, such as **HIGH 8** will be written for the BASIC Stamp. This instruction will control a device connected to P8 of the controller.

A connection to the I/O pins is also available on the 'App-Mod' header.

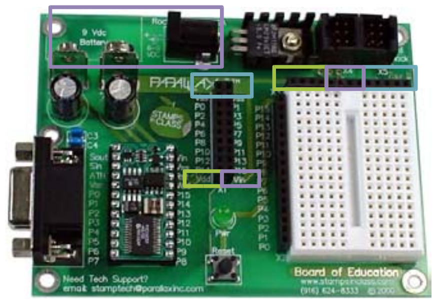


The P8 connection is available on the header next to the breadboard area.

Component Power Connections

- Power for the components are available on headers also.

+5V (Vdd) | 0V or ground (Vss) | Supply Voltage (Vin)

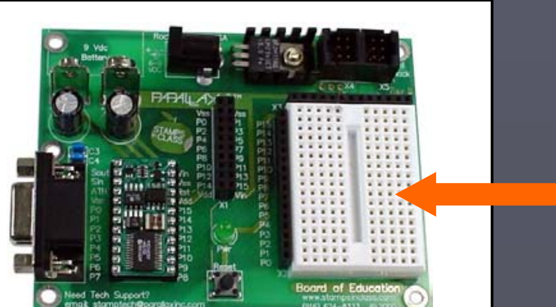


NOTE: Use of Vin may cause damaging voltages to be applied to the BASIC Stamp.

Use only under directed use!

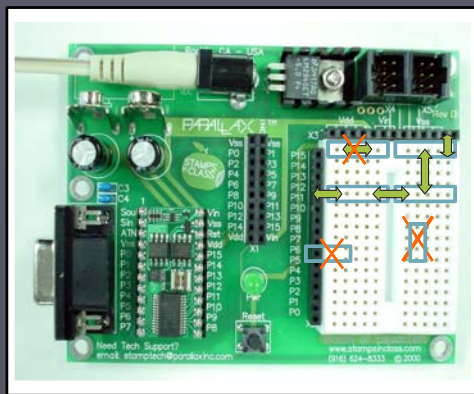
Connecting Components

- Of course, an important aspect to any BASIC Stamp project are the components that will be connected to the I/O pins of the Stamp.
- The carrier boards allow quick connections for the components.



Breadboard Connections

- Breadboard are rows of connectors used to electrically connect components and wiring.



Each row in each half of the breadboard are electrically the same point.

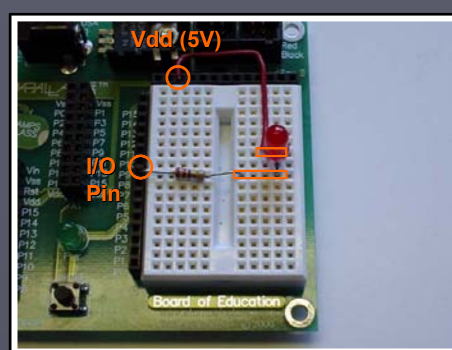
There exist no connections between the headers and the breadboards or in columns on the breadboard.

Components are connected between rows and to the headers to make electrical connections.

Components should NOT be connected on a single row or they will be shorted out of the circuit.

- This image is the Board of Education with several components connected.

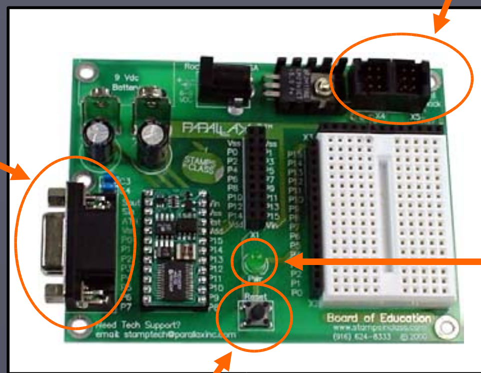
- The connections on the breadboard create a complete path for current to flow.



Other Features

Servo Header Connectors

Programming Port



Power-On LED

Reset Button to restart the program on the BASIC Stamp

Summary

- The BASIC Stamp is like a miniature computer that can be programmed to read and control Input/Output pins.
- Programs written on a PC are tokenized, serially transmitted and stored in the BASIC Stamp's EEPROM.
- The Board of Education provides a means of programming and connecting devices to the BASIC Stamp.

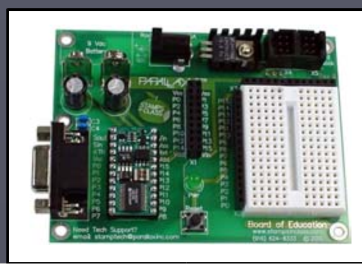
End of Section 1

Section 2: Other Carrier Boards

- Basic Stamp HomeWork Board
- BASIC Stamp Activity Board
- NX-1000
- Solder Carrier Board
- OEM module

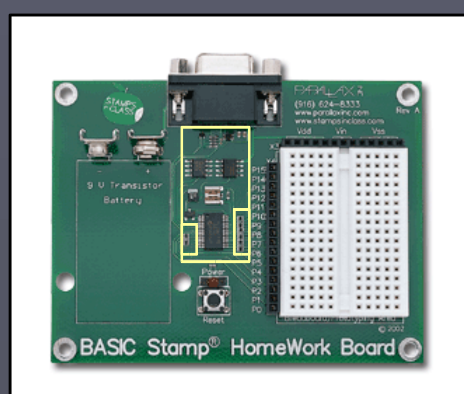
Other Programming Boards

- While this tutorial focuses on the Board of Education (BOE) carrier board, there are many others which may be used.
- All the boards have:
  - Power Connectors.
  - Communications ports.
  - P-numbered I/O connections.
  - Many have a separate 5V regulator for devices.



BASIC Stamp HomeWork Board

- The HomeWork Board is an inexpensive alternative for student projects.
  - The BASIC Stamp is integral to the board instead of a separate module.
  - All I/O have 220 ohm current limiting resistors. This means that the 220 ohm resistors used for connections in this tutorial may be omitted.



BASIC Stamp Activity Board

- The BASIC Stamp Activity Board is great board for novice users because it has commonly used devices which are pre-connected to the BASIC Stamp allowing quick program testing.

On-Board devices:
- 4 buttons
- 4 LEDs
- Speaker
- Potentiometer
- X10 power line interface
- Sockets for specific add-on ICs

- Each device is numbered, such as the blue button/LED with P7/8 (the switch is used for input, the LED is used for output).

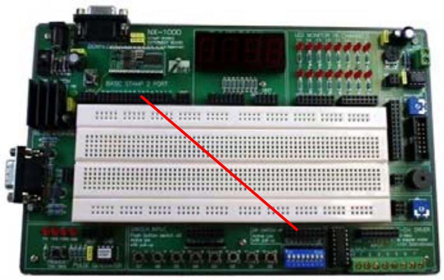
Programs in this tutorial will not work with the BS1.

- The 1st number is the I/O number if you are using a BASIC Stamp I.
- The 2nd number is the I/O number if you are using the BASIC Stamp 2 family.
- Code such as HIGH 8 will operate the blue button/LED combination with the BS2.
- Code in this tutorial is compatible with this board except where noted.

NX-1000

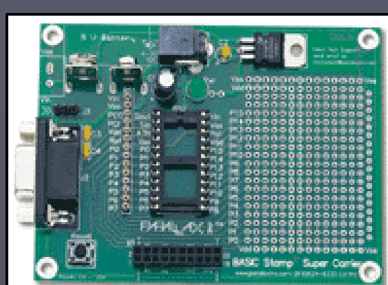
- This board is a great experimenter's board. It has a wide variety of devices which are NOT pre-connected.
- A large breadboard area accommodates many other devices.

- **Jumper wires are used between headers to connect desired devices to I/O pins.**



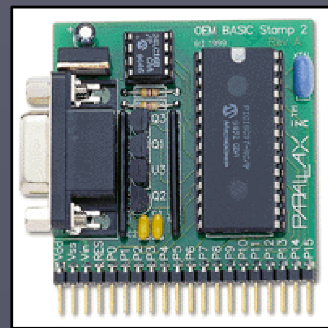
Through-Hole Solder Carrier Board

- For more permanent construction, boards with solder connections are available.



OEM module

- The BASIC Stamp 2 OEM is a discreet component version of the BS2 which may be purchased in kit form.
- The male header provides the means to 'plug-it' into your own board, or connect to other boards.



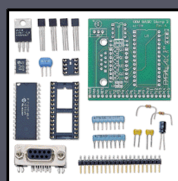
P0-P15 I/O

Reset Input (RES)

5.5 – 15V input (Vin)

Ground- 0V (Vss)

Regulated 5V (Vdd)



Power the board with EITHER:
A) 5.5-15VDC on Vin. This will also provide 5VDC regulated output on Vdd.

B) Regulated 5V Input on Vdd.

Summary

- There are a variety boards that may be used with the BASIC Stamp.
- Each has advantages and disadvantages. Choosing the best choice based on features and cost is important.

End of Section 2

Section 3: BASIC Stamp Editor

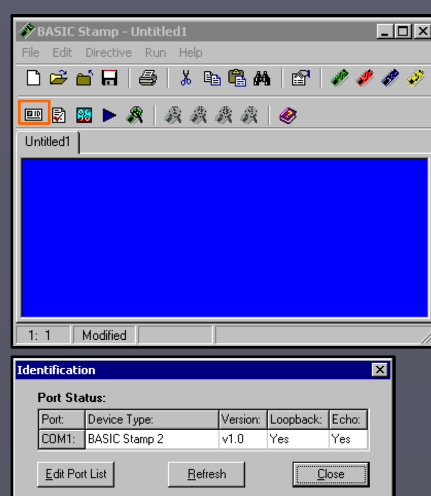
- BASIC Stamp Editor
- Identifying the BASIC Stamp
- Writing the Program
- Downloading or Running Code
- Tokenizing and Errors
- Commenting Code
- DEBUG Window
- Memory Map
- Preferences
- Help Files
- Instruction Syntax Convention

BASIC Stamp Editor

- The BASIC Stamp Editor is the application that is used to write, edit, and download the PBASIC programs for the BASIC Stamp.
- The software may be downloaded for free from Parallax. Some installations of Windows 95 and 98 may require an additional file to be installed. Please see the information on the download page for more information.
- Once installed, the Stamp Editor will be available on your desktop, and as a menu option under **Start → Program Files → Parallax Inc**

Identifying the BASIC Stamp

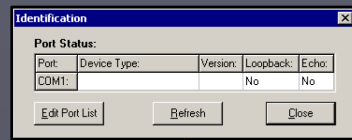
- Connect the BASIC Stamp carrier board to your computer with a serial cable.
- Power-up your BASIC Stamp carrier board.
- Use the *Identify* button to verify communications to your BASIC Stamp. 



Identification Errors

- If the ID shows:
  - No Device Type
  - No Loopback
  - No Echo

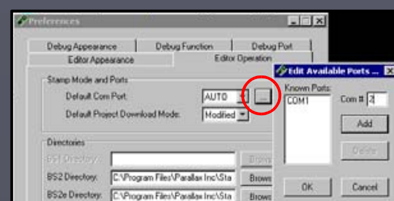
  It usually means the BASIC Stamp is not connected properly to the computer with a serial cable.



- Verify the carrier board is connected to the computer with a serial cable, full-modem variety (not null-modem).

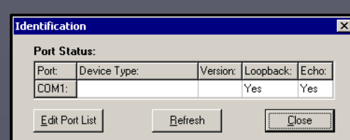
- If your computer has multiple COM ports, try another.

- If the COM port you are using is not listed, try adding it to the Stamp Editor using **Edit→Preferences**.



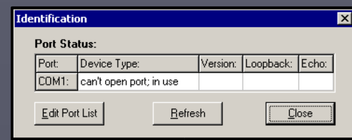
- If the ID shows:
  - No Device Type
  - Loopback - Yes
  - Echo – Yes

  It usually means the BASIC Stamp is connected, but it has no power.



- Verify the carrier board has power supplied and the power light is on (if available).

- If the COM port cannot be opened, it usually means another program has control of the port.

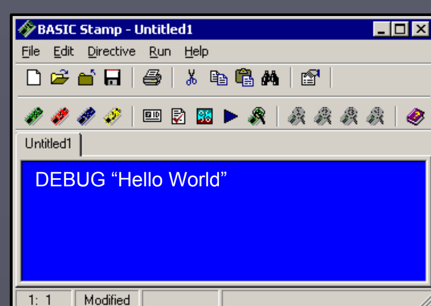


- Close any applications which may be using the port, including terminal programs, dial-up programs, Palm Pilot programs, PC Anywhere, StampPlot and other communication programs.

- If you cannot resolve the problem, if possible:
  - Test another person's operational board on your computer using their cable and yours.
  - Test your board on another computer, preferably one that had a working BASIC Stamp.
  - Contact Parallax support: support@parallax.com

Writing the Program

- BASIC Stamp programs are written in a version of BASIC called PBASIC entered into the BASIC Stamp Editor.

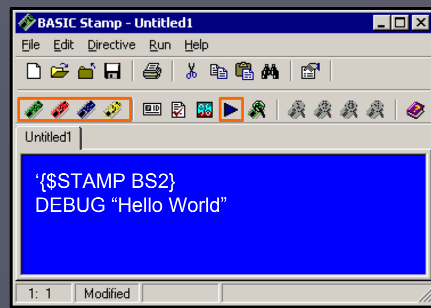


- A program typically reads inputs, processing data, and controls outputs.

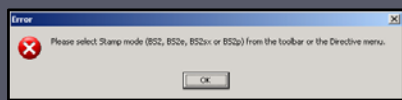
- Programs must conform to the rules of syntax so that the BASIC Stamp can understand what you are telling it.

Downloading or Running Code

- Once a program is entered, the Run button (or Ctrl-R) is used to tokenize and download the program to the BASIC Stamp.



- The Editor will request you indicate the style of BASIC Stamp you are using.

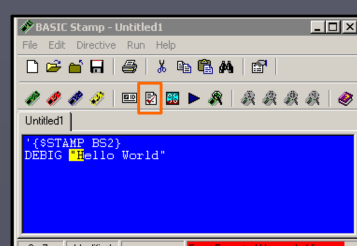


- The style may be selected from the menu, or by selecting your 'color' of your BASIC Stamp on the button bar.

- A *directive* will be added to the top of your code.

Tokenizing and Errors

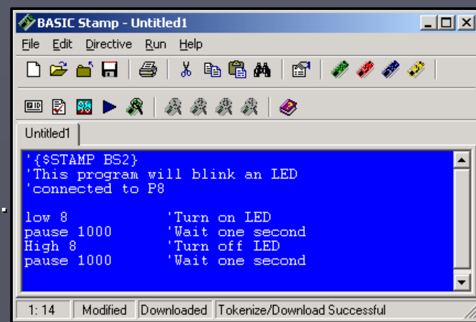
- When a program is 'Ran' the PBASIC code is converted to symbolic format called tokens. These are stored in ROM memory on your BASIC Stamp.
- In order to tokenize your program, the code must to conform to the rules of syntax for the language.
- If there are errors:
  - An error message will appear indicating a problem, the status turns red and code is highlighted.
  - Generally, the error can be found by looking before the highlighted area.
  - Read your code carefully looking for the *syntax error* or *bug*. In this example DEBUG is incorrectly spelled.



- Code may be syntax checked without downloading by using the Syntax Check button.

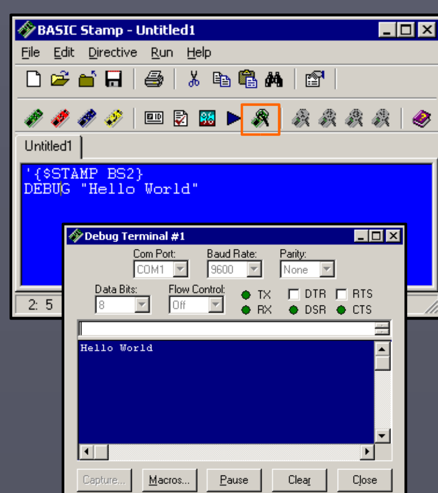
Commenting Code

- Comments, or remarks, are descriptions or explanations the programmer puts in the code to clarify what it is doing.
- Comments are signified by leading with an apostrophe.
- Comments are NOT syntax checked, nor do they increase the size of your program. So comment often and at length!



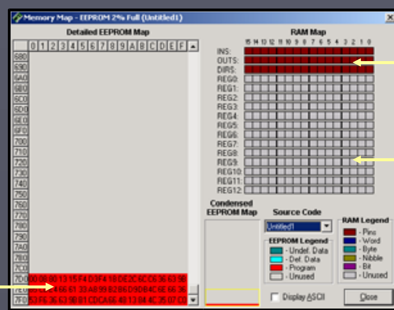
DEBUG Window

- Programs may contain a DEBUG instruction. This instruction sends serial data back to the computer on the serial cable.
- When DEBUG is present in a program, a DEBUG window will open in the Editor to view the returning data.
- The DEBUG button may be used to manually open a DEBUG window.



Memory Map

- The Memory Map button will open the BASIC Stamp window.
- This window shows how program (EEPROM) and variable memory (RAM) is being utilized.
- Note that the program is stored in memory from bottom-up.



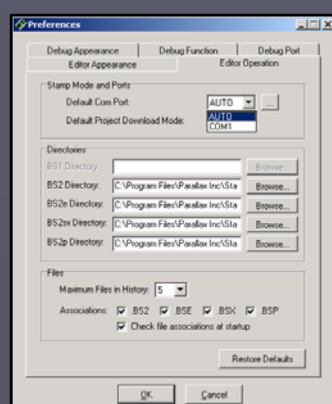
RAM Memory: I/O Control

RAM Memory: Variables

EEPROM Memory: Program space of tokenized program

Preferences

- Under the Preferences button you may:
  - Change color, font, and tab spacing for the text editor and debug screen.
  - Set the COM port on which the stamp is connected to, or be in automatic detection mode.
  - Modify the DEBUG settings.
  - You are encouraged to look through the available settings to become familiar with them.



Help Files

- There exists a help file that is very thorough at assisting you with any problems or questions you might have about instruction syntax or use while programming.

- By highlighting an instruction and pressing **F1**, the help files will open to display information on that instruction.

- Help provides a description, syntax (format) and example for each instruction.

Instruction Syntax Convention

- BASIC Stamp instructions follow a common code convention for parameters (parts) of instructions.
- Take for example the FREQOUT instructions, which may be used to generate tones from a speaker:
  FREQOUT *Pin, Period, Freq1 {, Freq2}*
  - The instruction requires that the *Pin*, *Period*, and *Freq1* is supplied and that each are separated by commas.
  - Optionally, the user MAY provide *Freq2* indicated by braces { }.
- While PBASIC is NOT case-sensitive, the common convention is to capitalize instructions, and use 1st letter upper-case for all other code.

Summary

- The BASIC Stamp Editor is an IDE (Integrated Development Environment) for:
  - Hardware identification.
  - Coding of the program.
  - Syntax (language rules) checking.
  - Memory utilization reporting.
  - Tokenizing and program transfer.
  - Integrated instruction help.

End of Section 3

Section 4: Input, Output, and Processing

- Usage Notes
- Before Changing Hardware
- Inputs, Processing, and Outputs
- Stamp I/O
- Output - Connecting an LED
  - Blinking the LED with HIGH, LOW
  - Blinking the LED with OUTPUT and OUT
- Debugging
  - DEBUG Instruction
  - DEBUG for Program Flow Information
  - Using DEBUG ? to Display Status
- Digital Inputs
  - Connecting an Active-Low Switch
  - Reading the Switch
- Controlling Outputs with Inputs
- DIRS, INS, OUTS
- Reading Analog Values with RCTime
- Frequency Output

Usage Notes

- This section is used to teach principles and construct a basic circuit.
- By the end of this section a complete circuit will be constructed consisting of 2 LEDs, 2 switches, a speaker and an RC network.
- This circuit is also used in sections 5, 6 and 7.

Before Changing Hardware

- Before you modify the hardware connected to the BASIC Stamp, it is best to download a simple program.
- This prevents the new hardware from applying voltages which may cause damage to pins configured for other hardware.
- Download the following program to the BASIC Stamp:

```
DEBUG "HELLO WORLD!"
```


Inputs, Processing, and Outputs

- Any system or program accepts input, process information, and controls outputs.

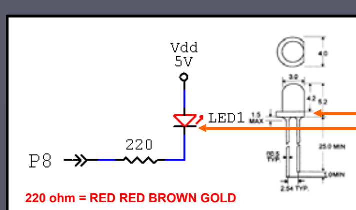
- The BASIC Stamp, and other microcontrollers, specialize in using input devices such as switches, and controlling output devices such as LEDs (Light Emitting Diodes).
- A program, written in a form of the BASIC language called *PBASIC*, is used for processing by writing code that instructs the BS2 what actions to take.

Stamp I/O

- There are 16 I/O (Input/Output) pins on the BS2 labeled P0 to P15. These are the pins through which input and output devices may be connected.
- Depending on the code that is written, each pin may act as an input to read a device, or as an output to control a device.
- We will begin by using a very common and simple output device -- the LED.

Output - Connecting an LED

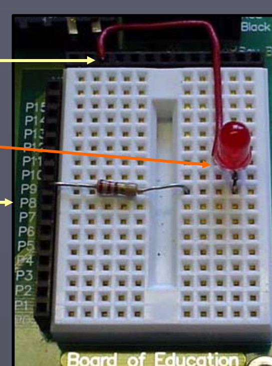
- Connect an LED to P8 as shown:



Vdd, NOT Vin.

Note cathode: the 'flat side' of LED

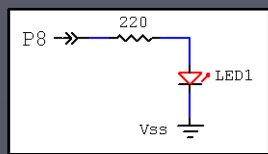
Connected on P8. Angle of shot makes it appear to be on P9.



An LED is a diode, meaning electrons can flow in only one direction, so polarity is important. The LED should have a flat side indicating the *cathode* or negative terminal. Also, the *anode* (positive terminal) generally has a longer lead than the *cathode*.

- In this configuration a LOW, or 0V, at P8 will allow current to flow through the LED to Vdd (+5V) lighting it. When P8 is HIGH (+5V), no current will flow and the LED will not light. The LED is *Active Low*.

- Another configuration that could be used is to have the LED *Active-High*. In this configuration the LED will light when the output is HIGH, or +5V. Current flows from ground or Vss (0V) to the 5V output on P8.



The 220Ω resistor will limit current flow to approximately 20mA . The output current from a BS2 pin should be limited to 20mA maximum. The maximum current for an LED is generally 30mA.

- Do NOT build this circuit, it is for information only. The circuit on the previous slide should be constructed.

Blinking the LED with HIGH, LOW

- Use the Stamp Editor to enter the following program:

```
'Prog 4A: Blink LED program

Main:
    HIGH 8        'Turn off LED
    PAUSE 1000    'Wait 1 second
    LOW 8         'Turn on LED
    PAUSE 5000    'Wait 5 seconds
GOTO Main         'Jump back to beginning
```

- Download or run the program.

- Monitor the LED. It should blink at a rate of 1 second OFF, 5 seconds ON. If not, check your configuration and code.

Code Discussion

- HIGH defines the pin to be an output and sets it to a HIGH state, digital 1 or 5V.
  - HIGH *pin 0-15*
  - HIGH 8
- LOW defines the pin to be an output and sets it to a LOW state, digital 0 or 0V.
  - LOW *pin 0-15*
  - LOW 8
- PAUSE instructs the BS2 to wait for the defined number of milliseconds (1/1000 seconds).
  - PAUSE *time in milliseconds 0-65535*
  - PAUSE 1000
- GOTO instructs the BS2 to jump to the defined label. More about this will be covered in Programming Structures.
  - GOTO *Label*

Blinking the LED with OUTPUT and OUT

- The HIGH and LOW instructions perform 2 actions:
  - Sets direction of the I/O pin to an output.
  - Sets the state of the output to be 0 or 1 (0V or 5V)
- Another means to perform the same process is to use code to set the direction, then the state.
- Enter and run this example.

```
' Prog 4B: Blink LED program using OUTPUT and OUT

OUTPUT 8          'Set P8 to be an output
Main:
    OUT8 = 1      'Turn off LED1
    PAUSE 1000    'Wait 1 second
    OUT8 = 0      'Turn on LED1
    PAUSE 5000    'Wait 5 seconds
GOTO Main         'Jump back to beginning
```

Code Discussion

- OUTPUT sets the pin to act as an output.
  - OUTPUT *pin*
  - OUTPUT 8
  - The BS2 on startup sets all I/O pins to inputs.
- OUT sets the state of the output.
  - OUTpin = *1 or 0*
  - OUT8 = 1
  - 1 sets the output HIGH (5V – Digital High or 1).
  - 0 sets the output LOW (0V – Digital Low or 0).
- Depending on program need, sometimes it is better to use the HIGH and LOW instructions, and other times to use OUTPUT and OUT.

Challenge 4A: Blink a 2nd LED

1. Connect a second *active-low* LED on P9. Solution
2. Code a program to blink only this LED using HIGH and LOW instructions. Solution
3. Code a program to blink only this LED using OUTPUT and OUT instructions. Solution

Challenge 4B: LED Cycling

- Code a program to perform the following sequence (use HIGH and LOW):
  - LED1 on P8 ON, LED2 on P9 OFF
  - Wait 2 seconds
  - LED1 on P8 ON, LED2 on P9 ON
  - Wait 1 second
  - Both LEDs OFF
  - Wait one-half second
  - Repeat

Solution

Debugging

- Debugging refers to the act of finding errors in code and correcting them. There are 2 types of errors which can be made when coding: *Syntax errors* and *Logical errors*.
  - **Syntax errors** are those that occur when the editor/compiler does not understand the code written.

    An example would be: GO TO Main
    The PBASIC tokenizer, which takes our code and puts it in a form the BS2 understands, does not have an instruction called GO TO (it has one called GOTO).

    This section of code would be flagged as having a syntax problem, which we identify and correct.

- **Logical errors** are those which have a valid syntax, but fail to perform the action we desire.

  For example, our program runs, but it seems the LED is off an abnormally long time. Looking at the code we find the bug:
  PAUSE 50000 instead of PAUSE 5000.

  The PBASIC compiler was perfectly happy with a 50 second pause, but logically it was not what we wanted to happen.

- Syntax errors are easily flagged when we try to run the program. Logical errors are more difficult because they require the programmer to analyze the code and what is occurring to determine the ‘bug’.

DEBUG Instruction

- The DEBUG instruction provides a valuable tool for the programmer.
- It provides a means of real-time feedback in debugging to:
  - Observe program execution.
  - Observe program values.
- It also allows the programmer to use a very sophisticated output device – A computer monitor.
- When a DEBUG instruction used, the Stamp Editor’s DEBUG window will open and display the data.

- When we run, or download, a program to the BS2, the program is transferred serially from Stamp Editor though a serial COM port to the BASIC Stamp.
- Using the same serial connection, the BS2 can transfer data back to the Stamp Editor to be displayed.
- Throughout this tutorial we will use DEBUG for various indications and describe the syntax used.

DEBUG for Program Flow Information

- Sometimes it is difficult to analyze a problem in the code because we have no indication where in the program the BS2 is currently at. A simple DEBUG in the code can provide feedback as to flow.
  - DEBUG "A description of code to be performed",CR
    CR is short for carriage return to move the cursor to the next line.
- DEBUG could be used to help identify the 'bug' where 50000 was typed instead of 5000.

- By placing some key DEBUG statements, we can observe the flow of the program. Of course, in most cases you may want to only place a DEBUG in the most likely areas based on observation.

```
'Prog 4C: Blink LED program with DEBUG location

OUTPUT 8          'Set P8 to output
Main:
    DEBUG " Turn Off LED",CR
    OUT8 = 1      'Turn off LED
    DEBUG " Wait 1 second",CR
    PAUSE 1000  'Wait 1 second
    DEBUG " Turn Off LED",CR
    OUT8 = 0      'Turn on LED
    DEBUG " Wait 5 seconds",CR
    PAUSE 50000           'Wait 5 seconds
    DEBUG " Go repeat program",CR
GOTO Main         'Jump back to beginning
```


Using DEBUG ? to Display Status

- Another simple use of DEBUG is to indicate the status of an output or input.
  - DEBUG ? OUTpin

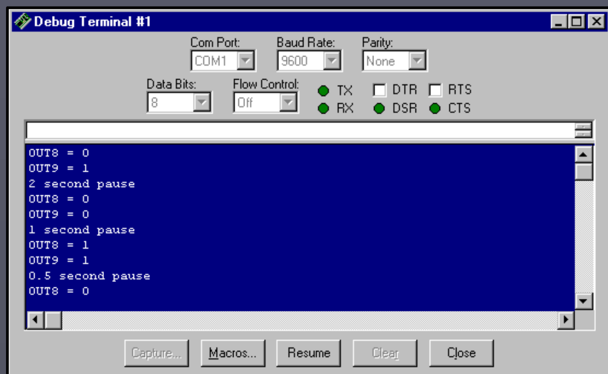
- Say for example the P8 LED was not lighting. Is it a code problem (OUT8 not going low?) or an electronics problem (LED in backwards?).
  - Using DEBUG in key spots, the status of P8 can be verified.

```
'Prog 4D: Blink LED program with DEBUG
value

Main:
    HIGH 8         'Turn off LED1
    DEBUG ? OUT8
    PAUSE 1000  'Wait 1 second
    LOW 8          'Turn on LED1
    DEBUG ? OUT8
    PAUSE 5000  'Wait 5 seconds
GOTO Main         'Jump back to beginning
```

Challenge 4C: Debugging

- Modify the code from Challenge 4B to indicate the status of P8 and P9 and describe the number of seconds of the pause.

- Example output:



Solution

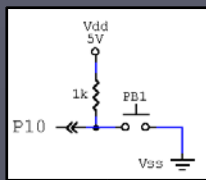
Digital Inputs

- Just as P0 – P15 on the BASIC Stamp can act as outputs to control devices, they can act as inputs to read devices, such as switches.

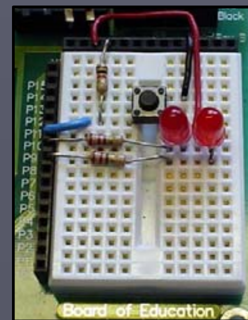
- By default, the BASIC Stamp I/O pins will act as inputs unless specifically set to be an output. In our code we specify the I/O as inputs out of good programming habits.
  - INPUT pin
  - INPUT 10

Connecting an Active-Low Switch

- Connect a push-button switch to P10 as shown:

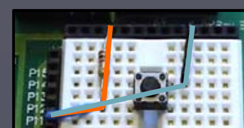


The push-buttons used in this tutorial have 4 terminals. 2 are electrically connected on one side of the button, and the other 2 on the other side. By wiring to opposing corners we ensure the proper connection independent of button rotation.



1KΩ = Brown Black Red Gold

- The push-button is a momentary normally-open (N.O.) switch. When the button IS NOT pressed (open), P10 will sense Vdd (5V, HIGH, 1) because it is *pulled-up* to Vdd.

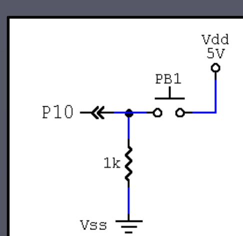


Button alone

- When PB1 IS pressed (closed), P10 will sense Vss (0V, LOW, 0) making it *Active-Low*.

Active-High Push-Button Switch

- Another configuration that could have been used is shown here. Notice that the position of the switch and resistor have been reversed.
  - When the button IS NOT pressed (open), P10 will sense Vss (0V, LOW, 0) because it is *pulled-down* to Vss.
  - When PB1 IS pressed (closed), P10 will sense Vdd (5V, HIGH, 1) making it *Active-High*.
- Do NOT build this circuit, it is for information only. The circuit on the previous slide should be constructed.



The BASIC Stamp has uncommitted inputs. That is, when an I/O pin is not connected and acting as an input, it cannot be assured to be either HIGH or LOW. Pull-up and pull-down resistors are needed to commit the input to the non-active (open) state for switches.

The 1K resistor is used to prevent a short-circuit between Vdd and Vss when the switch is closed.

Reading the Switch

- The digital value of an input can be read using the ***INpin*** instruction.
  - A 1 or 0 will be returned indicating a HIGH or LOW state on the input.
- This program uses DEBUG to display the digital value. Enter and Run the program. Note the value displayed when the push-button is in the down and up states.

```
'Prog 4E: Display the status of PB1 on P10

INPUT 10                    'Set P10 to be an input

Main:
   DEBUG ? IN10             'Display status of P10
   PAUSE 500                'Short pause
GOTO Main                   'Jump back to beginning
```


Challenge 4D: Reading a 2nd Button

1. Add a second active-low push-button switch on P11. Solution
2. Code a program to display the status of only PB2 on P11. Solution
3. Code a program to display the status of BOTH switches. Solution

Controlling Outputs with Inputs

- Now that we can control outputs and read inputs, it's time to perform a little processing and put the pieces together.
- The state of an input may be read with *INpin*.
- The state of an output may be controlled with *OUTpin*.
- Here is a program that will use the input pushbutton PB1 on P10 to control output LED1 on P8.

```
'Prog 4F: Controlling LED1 with input PB1

INPUT 10                    'Set P10 to be an input
OUTPUT 8                    'Set P8 to be an output

Main:
   OUT8 = IN10              'Set LED1 = PB1
GOTO Main                   'Jump back to beginning
```


Challenge 4E: Switch & LED Control

1. Code a program that will control LED2 on P9 with PB2 on P11.

   Solution

2. Code a program that will control:
   LED1 on P8 with PB2 on P11
   LED2 on P9 with PB1 on P10

   Solution

3. Code a program that will control:
   LED1 and LED 2 on P8 and P9 with button PB1 on P10.

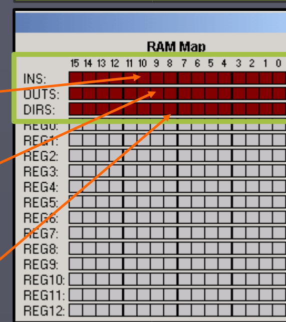
   Solution

DIRS, INS, OUTS

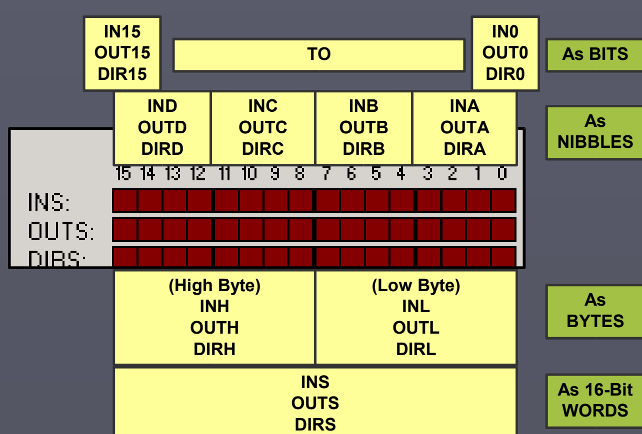
- Up to this point I/O have been set as inputs or outputs, and states set or read individually.

- Looking at the Memory Map, there are 3 16-bit registers which set the direction for the I/O, and which are read or written to.
  - **IN10** reads the value in the 10th bit (P10) of INS.
  - **OUT9 =1** sets the output state in the OUTS register for bit 9 (P9).
  - **OUTPUT 8** sets bit 8 (P8) for output in the DIRS register. This may also be written as **DIR8=1** (1=output, 0 = input).

Note: The colors of these registers are not affected by the code.



- The I/O can also be addressed as nibbles, bytes or the entire word.



- In our circuit, there are output devices on P8 and P9, and input devices on P10 and P11. P8 – P11 make up nibble C.

- The direction of the I/O can be set as a nibble with:
  DIRC = %0011 in binary. It may also be written as **DIRC = 3** in decimal, but the binary form is much easier to read for determining individual bit states.

  - This will set the DIRS nibble C for input (P11), input (P10), output (P9), output (P8).
  - Note that the bit positions are most-significant bit (MSB) to least-significant bit (LSB).

- Some various examples to illustrate the flexibility, code savings, and increased speed possibilities:
  - To read the entire lower byte (P0-P7) as inputs:
    ```
    DIRL=%00000000
    X = INL
    ```

  - To count up in binary on 8 LEDs connected on P8 to P15:
    ```
    DIRH = %11111111
    FOR X = 0 to 255
      OUTH = X
    NEXT
    ```

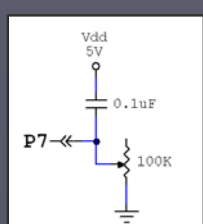
  - To set 4 outputs on P4-P7 equal to 4 inputs on P12-P15:
    ```
    DIRS = %0000000011110000
    OUTB = INDD
    ```

Reading Analog Values with RCTime

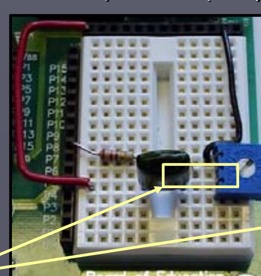
- A very simple method of bringing analog data into the BASIC Stamp is through the use of an instruction called RCTime.
- RCTime requires a capacitor and a resistor network, either of which may be variable (adjustable). Common variable resistance devices include:
  - Variable-Turn Resistors (Potentiometers)
  - Photo-Resistors
  - Temperature sensing devices such as thermistors
- Using RCTime with any of these devices can provide real-time analog data input.

Connecting the RC Network

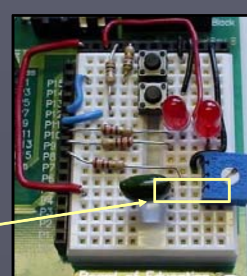
- Connect the Resistor-Capacitor (RC) network:



Resistor, capacitor and center pin of potentiometer on same row.



RC Network alone



Full Circuit with RC Network

You may also use a 1uF capacitor with a 10K ohm potentiometer.

RCTime measures the time to charge the capacitor through the resistor. The higher the resistance or capacitance, the longer the time. For a full discussion on RCTime, please see your editor help files or BASIC Stamp Manual.

RCTime Code

- Enter and run the following code.

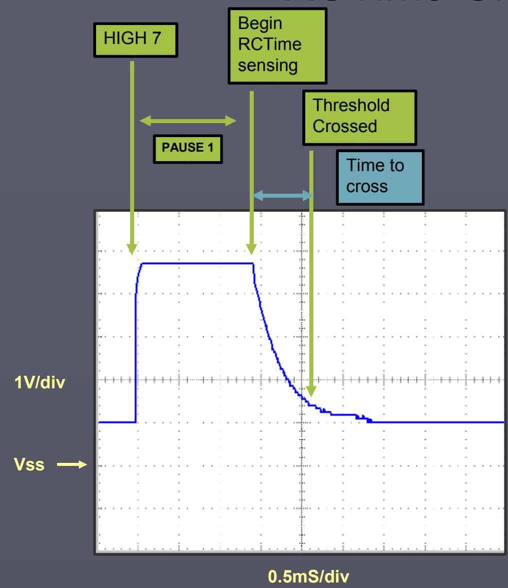
```
'Prog 4G: Monitoring RCTime

Pot VAR WORD  'Variable to hold results

Main:
   HIGH 7              'Discharge network
   PAUSE 1             'Time to fully discharge
   RCTIME 7,1,Pot      'Read charge time and store in Pot
   DEBUG ? Pot         'Display value of Pot
   PAUSE 500           'Short pause
GOTO Main              'Jump back to beginning
```

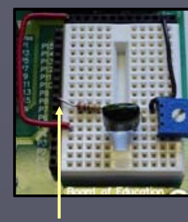
- Adjust the resistor full each direction and monitor the value. The full range should be approximately 0-6000.

RCTime Code Discussion

- **Pot VAR WORD** defines a variable named Pot. More about variables will be discussed in the next section.
- **HIGH 7** places +5V on pin 7, discharging the capacitor.
- **PAUSE 1** provides time to allow the capacitor to fully discharge.
- **RCTIME 7,1,Pot** instructs the BS2 to time on pin 7 how long it takes leave the specified state (1) and store the results into the variable Pot.
  - **RCTIME pin, state, variable**

RCTime Graph



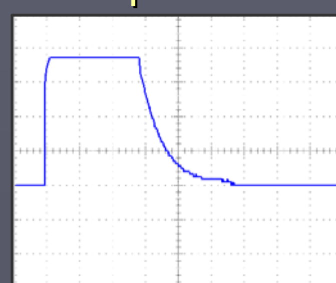


Measured at P7

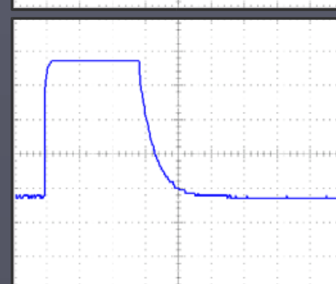
- In digital, a HIGH (1) or LOW (0) is commonly denoted by Vdd or Vss (5V and 0V), but there exists a **threshold voltage**, above which the controller senses a HIGH, and below which the controller senses a low.
- The threshold voltage for the BASIC Stamp is around 1.7V.

RCTime Graph Comparison

- With a high resistance, the current is low and the capacitor takes a relatively long time to charge.



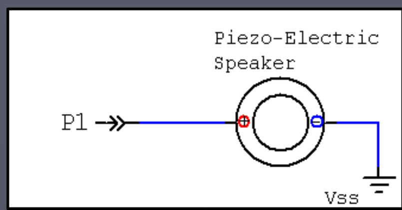
- As resistance decreases the current increases allowing the capacitor to charge more quickly.



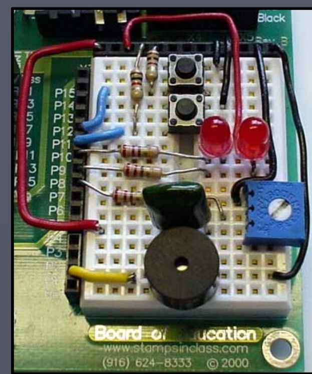
- A value proportional to the time to reach the new state is stored.

Frequency Output

- The PBASIC instruction FREQOUT can be used to easily drive a speaker for sound-effects.
- Connect the components to the circuit.



NOTE: If you are using the BASIC Stamp Activity Board, the speaker is on P11. You will need to adjust the code used in this part accordingly.



Frequency Output Code

- Enter and run the following code:

```
'Prog 4G: Simple Frequency Generation
'Activity Board -- use FREQOUT 11 for pin 11.

Main:
   FREQOUT 1, 2000, 1000    'Tone at 1000Hz for 2 seconds
   PAUSE 1000               'Short pause
GOTO Main                   'Jump back to beginning
```

If all went well you should be hearing a tone.

- The syntax of FREQOUT is:
  - **FREQOUT *pin, duration in milliseconds, frequency in Hertz***
- The allowable duration and frequency is 0 – 32767 though your speaker will only have decent tone generation between 500-4000Hz or so.

Challenge 4F: Variable Frequency Control

- Combine what was learned in using RCTIME and FREQOUT to code a program which changes the tone of the speaker in relation to the potentiometer setting.

Hint

Solution

Summary

- The BASIC Stamp can control simple output devices, such as LEDs, with instructions such as HIGH, LOW and OUT.
- The BASIC Stamp can read simple input devices, such as switches, using the IN instruction.
- Multiple I/O can be read or written to as grouping of bits.
- Simple resistive analog values may be read using the RCTime instruction.
- Output frequency on a pin can be performed with the FREQOUT instruction.

End of Section 4

Section 5: Variables and Aliases

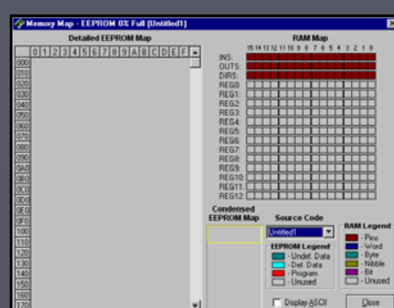
- Variables
  - RAM Memory
  - Variable Types
  - Variable Declaration
  - Variable Conventions
  - Coding with Variables
- Constants
  - Coding with Constants
- I/O Aliases
  - Coding using I/O Aliase
  - Common Circuit Declar

Variables Overview

- Variables are needed when a program requires a value to be stored.
- Variables correspond to a memory location which we can read from and write to (Random Access Memory – RAM).
- Variables allow the programmer to use descriptive words to indicate the contents of the memory location.
- Aliases may also be declared for I/O control to allow descriptive words to indicate device connections.

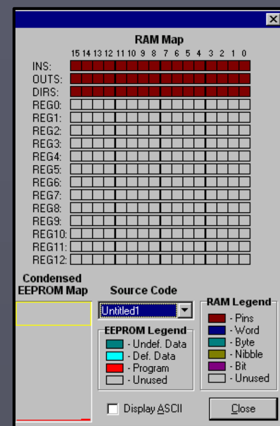
RAM Memory

- Once a program in entered, the Memory Map button on the toolbar may be clicked to view the contents of the code memory (EEPROM Map) and the variable memory (RAM Map).
- In the BS2, the code space is 2K bytes (2048 bytes) in size and fills from the bottom up.
- The RAM for variable storage is 26 bytes in size.



- INS, OUTS and DIRS are the registers (RAM locations) which hold the status of the I/O pins.
- REG0 – REG12 are 16-bit registers (word sized) used for general variable storage.
- The variable registers may hold:
  - 13 16-bit variables (Words)
  - 26 8-bit variables (Bytes)
  - 52 4-bit variables (Nibbles)
  - 208 1-bit variables (Bits)

    OR
  - Any combination of the above within memory size constraints.



Variable Types

- A variable may be declared to be a word, byte, nibble or bit. To maximize the limited memory, programmers use the smallest size that will meet the needs of the variable requirements.
- The maximum number of unique states (modulus) for each size is: **2^n where *n* is the number of bits.**
  i.e.: A byte, with 8 bits, has 2^8 unique values or 256.
- In binary, the maximum value for each size is:
  2^n-1 where *n* is the number of bits.
  i.e.: A byte, with 8 bits, can hold 0 to 2^8-1, or 0 to 255.

Bit	2^1	0 or 1
Nibble (Nib)	2^4	0 to 15
Byte	2^8	0 to 255
16-bit Word	2^{16}	0 to 65535 unsigned -32768 to +32767 signed

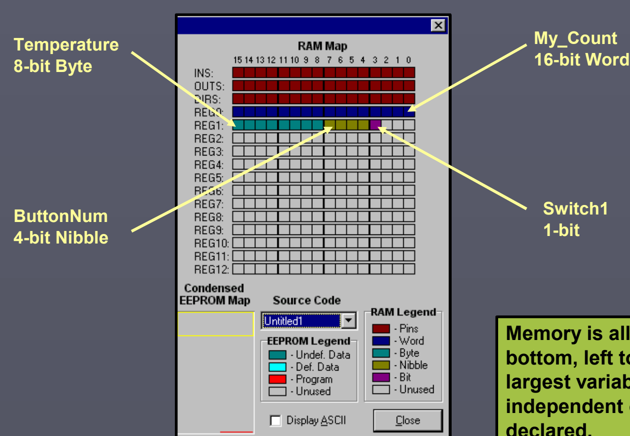
Variable Declaration

- A variable in PBASIC is declared with the syntax of:
  variableName VAR size

- For example, the following will declare a variable named *Temperature* and allocate a byte of memory for it. This 8-bit location can be used to hold values between 0 and 255.
  Temperature VAR BYTE

```
'Prog 5A: Test of variable declaration

' *************** Declare Variables
Temperature      VAR   BYTE
My_Count         VAR   WORD
Switch1          VAR   BIT
ButtonNum        VAR  NIB

Temperature = 100
```


- Open the Memory Map to see how the RAM was allocated for the various declarations:



Memory is allocated top to bottom, left to right from largest variables to smallest independent of the order declared.

- Variables can be read and modified. Enter and run the following code. Monitor the values of each as to when they overflow the limits of their size.

```
'Prog 5B: Test of variable sizes

' ********************** Declare Variables
ByteCount        VAR  BYTE
WordCount        VAR  WORD
BitCount         VAR  BIT
NibCount         VAR  NIB

Main:
  WordCount = WordCount + 1000        'Add to each variable
  ByteCount = ByteCount + 20
  NibCount = NibCount + 1
  BitCount = BitCount + 1
  DEBUG CLS                           'Clear the screen
  DEBUG ? WordCount : DEBUG ? ByteCount : DEBUG ? NibCount : DEBUG ? BitCount
  PAUSE 500
GOTO Main
```

Colons may be used to separate instructions on a single line

Variable Conventions

- Good variable naming and commenting is important and there are certain rules in naming:
  - Variables cannot contain special characters such as !, @,$ except for an underscore, _. My_Variable
  - Variables may contain numbers but cannot start with a number. Freq1
  - Variable names cannot be a PBASIC instruction.
  - Variables should have descriptive names. Generally, capitalize the 1st character of each word. CountOfPresses
  - Declare all variables at the top of your code and comment their use.
  - Size the variable appropriate to its use conserving memory whenever possible.
  - Re-use variables for common tasks such as multiple loops in the code.

```
'Sample program with variables

' ****************** Declarations *************************
' ****************** Variables
x               VAR BYTE       'General use variable
PressCount      VAR WORD       'Holds number of times button is pressed
Pot_Value       VAR WORD       'Value of Pot from RCTIME
Switch1         VAR BIT        'Value of switch 1
```

- All the variables in the above code fragment are legally named and follow good convention.
- Examples of illegal variable names:
  - My Count — Space in name
  - 1Switch — Starts with a value
  - Stop! — Invalid name character
  - Count — PBASIC instruction
- Due to space constraints, the tutorial examples may be briefer in commenting than good practice dictates.

Challenge 5A: Variable Naming

- Declare variables for the following requirements:
  - To hold the number of seconds in a minute.
  - To hold the number of dogs in a litter.
  - To hold the count of cars in a 50 car garage.
  - To hold the status of an output.
  - To hold the indoor temperature.
  - To hold the temperature of a kitchen oven.

Solution

Coding with Variables

- To assign a variable a value, an equation sets it equal to a value:
  VariableName = value
- Once assigned a value, variables may be used in place of values for any number of purposes.
- Enter and test this program by slowly and quickly adjusting the potentiometer.

```
'Prog 5C: Play tone based on amount of potentiometer movement
'Activity board users use FREQOUT 11 instead of 1
'****************** Declarations ***********************
'****************** Variables
Pot_Current         VAR   WORD          'Current value of potentiometer
Pot_Last            VAR   WORD          'Last value of potentiometer
Freq_Play           VAR   WORD          'Tone to sound speaker

Main:
            HIGH 7: PAUSE 1                        ' Read Potentiometer using RCTIME
            RCTIME 7,1,Pot_Current                 '  and store as current pot value
            Freq_Play = Pot_Current - Pot_Last     ' Determine amount of change since last reading
            FREQOUT 1,500,Freq_Play                ' Play tone based on change
            Pot_Last = Pot_Current                 ' Save current pot value for last value
GOTO Main
```


Constants

- Constants provide the ability to assign names to values that *do not change*. They allow an easier reading of code.
- Unlike variables, which are used at run-time, constants are used when the program is compiled or tokenized and *use no additional memory*.
- Common uses of constants:
  - Naming of I/O pin numbers.
  - Naming of values which will not change such as PI (Note: the BS2 operates on whole numbers only).
- Constant names follow the same rules as variables and are declared as follows:
  constantName CON value
- An example may be in Program 5C of naming the pin to which the speaker is connected:
  Speaker CON 1

Coding with Constants

- In fact, let's clean up program 5C using constants.
- By using constants the code is more readable, and if we need to change a pin connection, only the constant value needs to be updated.

```
'Prog 5D: Play tone based on amount of potentiometer movement using constants
'****************** Declarations ************************
'****************** Variables
Pot_Current          VAR  WORD          'Current value of potentiometer
Pot_Last             VAR  WORD          'Last value of potentiometer
Freq_Play            VAR  WORD          'Tone to sound speaker
'****************** Constants
Speaker              CON 1              ' Speaker pin (Activity board users use 11)
PotPin               CON 7              ' Potentiometer pin
SpeakerDur           CON 500            ' Duration to sound speaker

Main:
        HIGH PotPin : PAUSE 1                       ' Read Potentiometer using RCTIME
        RCTIME PotPin,1,Pot_Current                 '  and store as current pot value
        Freq_Play = Pot_Current - Pot_Last          ' Determine amount of chance since last reading
        FREQOUT Speaker, SpeakerDur, Freq_Play      ' Play tone based on change
        Pot_Last = Pot_Current                      ' Save current pot value for last value
GOTO Main
```


Challenge 5B: LED Constants

- Below is the challenge solution to blink 2 LEDs. Modify the code to use constant names for LED pin connections.

```
'** 4B Challenge Solution – Blink second LED **

Main:
        LOW 8           'LED1 on
        HIGH 9          'LED2 off
        PAUSE 2000      'Wait 2 seconds
        LOW 9           'LED2 ON (P9 LED stays on)
        PAUSE 1000      'Wait 1 second
        HIGH 8          'LED1 off
        HIGH 9          'LED2 off
        PAUSE 500       'Wait one-half second
GOTO Main
```

Solution

I/O Aliases

- Just as names can be assigned to RAM memory locations using the VAR instruction, VAR can be used to assign names to the status of I/O when using the **IN** and **OUT** instructions. This creates an *alias* name for the I/O.
  AliasName VAR INpin
  AliasName VAR OUTpin

- Example: PB1 VAR IN10

- This allows for cleaner code and does not use any additional memory.

Coding using I/O Aliases

- Let's modify a previous program to make it a bit more readable using I/O aliases.
- Notice that OUTPUT and INPUT could not be made more readable without first assigning constants to the pin numbers.

```
'{$STAMP BS2}
'Prog 5E: Controlling output LED1 with PB1 using I/O Variables

' *************** Declarations *************
' *************** I/O Aliases
PB1       VAR  IN10          ' Pushbutton input pin
LED1      VAR  OUT8          ' LED1 output pin

'*************** Set I/O Directions
INPUT 10             'Set P10 to be an input
OUTPUT 8             'Set P8 to be an output

Main:
   LED1 = PB1        'Set LED state = pushbutton state
GOTO Main            'Jump back to beginning
```

Challenge 5C: I/O Aliases

- Code a program that will control:
  LED2 on P9 with the PB1 on P10
  LED1 on P8 with the PB2 on P11

Solution

Common Circuit Declarations

- For the remainder of this section, a common section of declarations will apply to all the programs to minimize the amount of coding and space required.

- In some cases an LED may be controlled with HIGH and LOW, other times it may be controlled with IN and OUT. Note that these 2 uses require 2 different variables. If only LED1 VAR OUT8 were used, a line of code such as HIGH LED1 would really mean HIGH 1 or HIGH 0 since LED1 would return the *value* of OUT8.

```
' *********Section 5 Common Circuit Declarations ***********
' **************** I/O Aliases ******************
LED1        VAR     OUT8        'LED 1 pin I/O
LED2        VAR     OUT9        'LED 2 pin I/O
PB1         VAR     IN10        'Pushbutton 1 pin I/O
PB2         VAR     IN11        'Pushbutton 2 pin I/O
Pot         VAR     WORD        'Potentiometer value
' *************** Constants **********************
LED1_Pin    CON     8           ' Constant to hold pin number of LED 1
LED2_Pin    CON     9           ' Constant to hold pin number of LED 2
PB1_Pin     CON     10          ' Constant to hold pin number of pushbutton 1
PB2_Pin     CON     11          ' Constant to hold pin number of pushbutton 2
Speaker     CON     1           ' Speaker Pin ***** Activity board users set to 11 ******
Pot_Pin     CON     7           ' Input for Potentiometer RCTIME network
PB_On       CON     0           ' Constant for state of pressed switch (Active-Low)
PB_Off      CON     1           ' Constant for state of un-pressed switch
LED_On      CON     0           ' Constant for state to light an LED (Active-Low)
LED_Off     CON     1           ' Constant for state to turn off an LED
' *************** Set common I/O directions *******
OUTPUT LED1_Pin                 'Set pin for LED1 to be an output
OUTPUT LED2_Pin                 'Set pin for LED2 to be an output
INPUT PB1_Pin                   'Set pin for pushbutton 1 to be an input
INPUT PB2_Pin                   'Set pin for pushbutton 2 to be an input
' ***************  Example uses **********************
'LED2 = LED_On                  'OUT9 = 0
'LED1 = PB1                     'OUT8 = IN10
'HIGH LED1_Pin                  'HIGH 8
```


Summary

- Variables are used to hold values that change in the program.
- There are 26 bytes available for variables.
- Variables may be sized as bits, nibbles, bytes or words depending on the required size.
- Constants can be declared to hold name values that DO NOT change.
- I/O pins can be names to give a descriptive identifier to the pin's use.

End of Section 5

Section 6: Program Flow

- Introduction to Flow
  - Program Planning – Pseudo-Code & Flowcharts
  - Sequential Flow
  - Sequential Flow Example
- Branching Overview
  - Looping with GOTO
  - Looping Flow Example
- Conditionals Overview
  - IF-THEN
  - IF-THEN Example: Alarm
  - Looping with a Counter
  - Repeating Alarm
  - FOR-NEXT
  - Repeating Alarm with FOR-Loop
  - Speaker Tone with FOR-Loop
- Subroutines
  - Cleaner Coding with GOSUBS
  - Sounding Alarms with GOSUB
- Using the BRANCH Instruction
- Saving Power – END & Sleep

Introduction to Flow

- The programs in the tutorial have been relatively easy and follow a sequence of steps from top to bottom. At the end of each program, GOTO Main has been used to loop the program back to the start.
- Virtually all microcontroller programs will continually repeat since they are typically embedded in processes to be operated continually.
- Sequential flow (top to bottom), looping, unconditional branching, and conditional branching will be explored in this section.
- The newer PBASIC 2.5 implementation greatly extends control structures. Please review Appendix A after this section.

Program Planning – Pseudo-Code & Flowcharts

- Depending on your proficiency, a little planning can help a lot in developing programs.
  - Plan your device placement carefully for good layout and utilization of I/O.
  - Decide on variables needed for storage or manipulation.
  - Plan the flow of your program. Use pseudo-code and/or flowcharts to structure the code properly.

Pseudo-Code

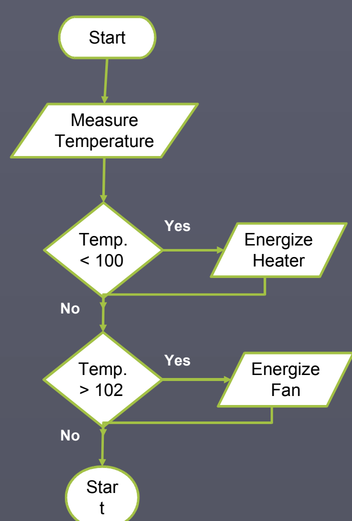
- Pseudo-Code are English statements describing what steps the program will take. They are not programmable code, but a guide in writing the code.
- For example, a program is needed to control the temperature of an incubator at 101F. Without even knowing code, a general outline can be made in pseudo-code.

- Start of program
- Measure temperature
- Temperature < 100 F?
  - Yes, Turn on heat
- Temperature > 102 F?
  - Yes, Turn on cooling fan
- Go back to start.

Flowcharts

- Flowcharts are means of developing the flow of a program visually.
- Symbols are used to indicate the type of operation.

Start/Stop	Input/Output	Process
Decision	Connector	Pre-Defined Process



Sequential Flow

- Sequential flow of code begins at the top with the first instruction, then the next in line, then the next and so on.
- When there exist logical errors in the code, one of the best means is to manually step through it by looking at each line and analyzing what it performs, then moving to the next appropriate line. At some point the programmer may see a flaw in the flow of the program.
- Sequential flow is the easiest to program and debug.

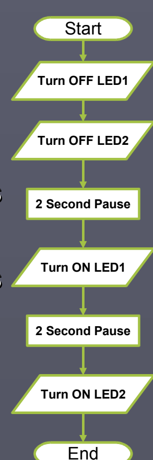
Sequential Flow Example

Pseudo-Code:

Start of program

- Turn off LED 1
- Turn off LED 2
- Pause for 2 seconds
- Light LED 1
- Pause for 2 seconds
- Light LED 2
- End of program

Flowchart:



Code:

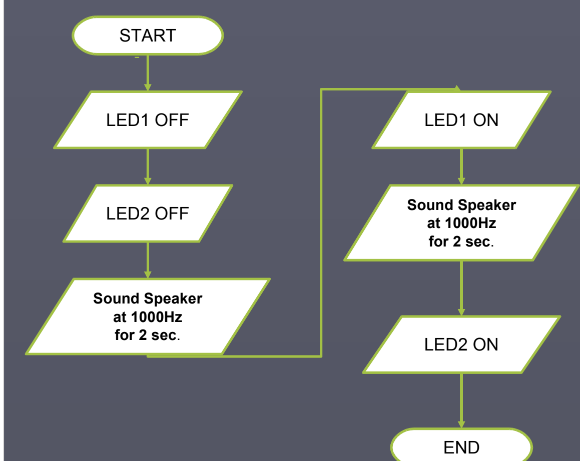
```
' <<<< INSERT COMMON
'  CIRCUIT DECLARATIONS >>>>

'Prog 6A: Example of sequential flow

' ****** Main program ************
LED1 = LED_Off     'Turn off LED 1
LED2 = LED_Off     'Turn off LED 2
PAUSE 2000         'Pause for 2 sec.
LED1 = LED_On      'Light LED 1
PAUSE 2000         'Pause for 2 sec.
LED2 = LED_On      'Light LED 2
END
```

Sequential Flow Example Discussion

- In all three, the flow of the program was from the top to the bottom with no branches, loops, or decisions. Even though most programs will contain loops and branches, sections of the program will be sequential.
- The previous program only performs the routine once, because there is no looping. It will perform the routine when the program is downloaded (ran) or any time the BS2 is reset.
- After the program is complete, the circuit speaker may click and the LED's may blink briefly.
- In many cases, the code comments may be the flowchart text or pseudo-code descriptions.

Challenge 6A : Sequential Code

- Write and test code for the following operation (use the common circuit variables and constants).



Solution

Branching Overview - GOTO

- Branching is the act of breaking out of a sequence to perform code in another location of the program.
- The simplest form of branching is to use the **GOTO** instruction: GOTO *label*
- A label is a name given to a certain location in the program. The labels follow the same naming convention that variables and constants do. They should be representative of the code to be performed.

Looping with GOTO

- Looping is the act of repeating a section of code.
- Our programs in section 4 used looping with GOTOs extensively so the programs would repeat.
- Let's modify program 6A to include looping.

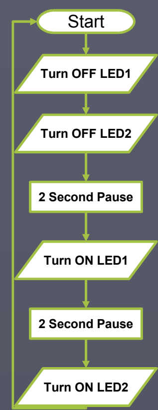
Looping Flow Example

Flowchart:

Pseudo-Code:

Start of program

- Turn off LED 1
- Turn off LED 2
- Pause for 2 seconds
- Light LED 1
- Pause for 2 seconds
- Light LED 2
- Go back to start



Code:

```
' <<<< INSERT COMMON
' CIRCUIT DECLARATIONS >>>>

'Prog 6B: Example of sequential flow
'            with looping

' ****** Main program ************
Main:
  LED1 = LED_Off   'Turn off LED 1
  LED2 = LED_Off   'Turn off LED 2
  PAUSE 2000       'Pause for 2 sec.
  LED1 = LED_On    'Light LED 1
  PAUSE 2000       'Pause for 2 sec.
  LED2 = LED_On    'Light LED 2
GOTO Main          'Repeat sequence
```


Looping Flow Discussion

- The program will be in a continual loop.
- Enter and run the program.
- Is the lighting of LED 2 noticeable? Why not? How could the program be modified to be better?


```
' <<<< INSERT COMMON
' CIRCUIT DECLARATIONS >>>>

'Prog 6B: Example of sequential flow
'            with looping

' ****** Main program ************
Main:
  LED1 = LED_Off   'Turn off LED 1

  LED2 = LED_Off   'Turn off LED 2

  PAUSE 2000       'Pause for 2 sec.

  LED1 = LED_On    'Light LED 1

  PAUSE 2000       'Pause for 2 sec.

  LED2 = LED_On    'Light LED 2

GOTO Main          'Repeat sequence
```

Conditionals Overview

- The previous example is an *unconditional branch*; the program will branch back to Main regardless of any code parameters.
- In a *conditional branch* a decision is made based on a current condition to branch or not to branch.
- As humans, we constantly make decisions based on input as to what to perform. Shower too cold? Turn up the hot. Shower too hot? Turn down the hot water.
- Microcontrollers can be programmed to act based on current conditions. Switch closed? Sound an alarm!

IF…THEN

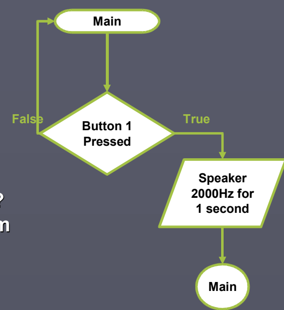
- The **IF-THEN** is the primary means of conditional branching.
  IF *condition* THEN *addressLabel*
- If the condition is evaluated to be true, execution will branch to the named address label.
- If the condition is not true, execution will continue to the next step in the program sequence.
- A condition is typically an equality:
  value1 = value2
  value1 > value2
  value1 < value2
  IN8 = 1

Compared to many versions of BASIC and other languages, the PBASIC 2.0 implementation of the IF-THEN is fairly limited. See the PBASIC 2.5 appendix for new implementations of IF-THEN.

IF-THEN Example: Alarm

- This program will sound the alarm as long as pushbutton 1 is pressed.

Flowchart



Pseudo-Code

Start:
- **Is button 1 pressed?**
- **Yes, Go sound Alarm**
- **No, Go back to start**

Alarm
- **Sound speaker**
- **Go back to start of program**

Program Code

```
' <<<< INSERT SECTION 5 COMMON
' CIRCUIT DECLARATIONS >>>>

'Prog 6C: Conditional Branching Alarm

Main:
  ' If pushbutton 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  'Sound the alarm
  FREQOUT Speaker, 1000, 2000
GOTO Main
```


IF-THEN Code Discussion

Without button 1 pressed, the condition would be false, (PB1 = 1, PB_On = 0) and the flow of our code would follow this path:

Execution begins

Condition is false, continue to next line

```
Main:
  ' If switch 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  'Sound the alarm
  FREQOUT Speaker, 1000, 2000
GOTO Main
```

Go back to start of program

And so it would repeat as long as the switch is not pressed.

With button 1 pressed, the condition would be true (PB1 = 0, PB_On = 0) and the flow of our code would follow this path:

Execution begi

```
Main:
  ' If switch 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  'Sound the alarm
  FREQOUT Speaker, 1000, 2000
GOTO Main
```

Condition is true, GOTO the address label of alarm

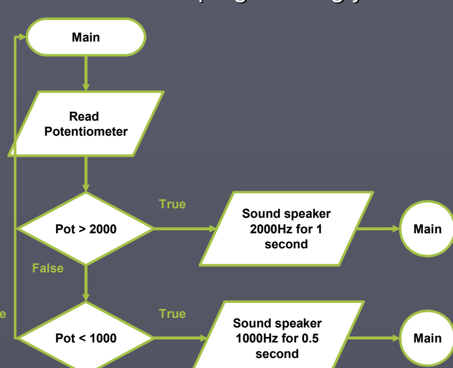
Follow the sequence

Go back to start of program

And so it would repeat as long as the button is pressed.

Challenge 6B: Potentiometer Alarm

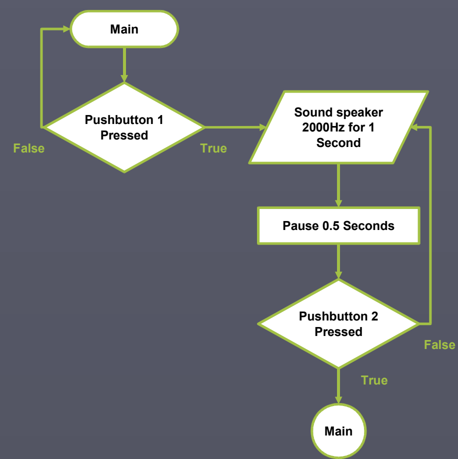
- Code a program to perform the following:
  - If the value of the potentiometer is greater than 2000, sound the speaker at 2000 Hz for 1 second.
  - If the value of the potentiometer is less than 1000, sound the speaker at 1000 Hz for 0.5 seconds.
  - Use the flowchart in programming your code.



Solution

Challenge 6C: Lock-in Alarm

- Code a program that will sound a 2000Hz, 1 second tone if PB1 is pressed. The alarm will lock-in and repeat until PB2 is pressed. Follow the following flowchart.



Solution

Looping with a Counter

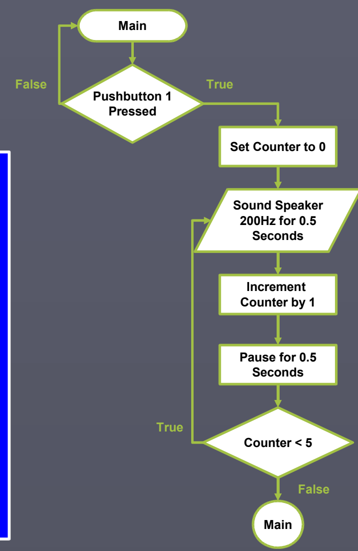
- In many circumstances a program needs to keep count of an event, such as the number of times a button is pressed. A counter may also be used to perform an action a specific number of times.

- A counter is simply a variable which is incremented or decremented each instance (typically by 1).

- Steps in using counters:
  - Declare a variable for the counter
  - Reset or initialize the counter
  - Update the counter
  - Act upon value of count

Repeating Alarm

- This example uses a counter to sound the speaker 5 times when PB1 is pressed.




```
' <<<< INSERT SECTION 5 CIRCUIT DECLARATIONS
'Prog 6D: Looping with counter Alarm
Counter VAR NIB       'Variable for counting

Main:
              ' If pushbutton 1 is pressed,
              ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  Counter = 0                          'Initialize counter
Alarm_Again:
  FREQOUT Speaker, 500, 2000  'Sound the alarm
  PAUSE 500
  Counter = Counter + 1                'Increment Counter
  IF Counter < 5 THEN Alarm_Again    'Check counter
GOTO Main
```


Repeating Alarm Major Points

A variable appropriately sized is declared.

Counter is reset to 0 Upon entering routine.

Counter is updated within loop.

Counter is checked. If not at full count, loop back AFTER the reset point.

```
' <<<< INSERT SECTION 5 CIRCUIT DECLARATIONS
'Prog 6D: Looping with counter Alarm
Counter VAR NIB                     'Variable for counting

Main:
  ' If pushbutton 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  Counter = 0                       'Initialize counter
Alarm_Again:
  FREQOUT Speaker, 500, 2000        'Sound the alarm
  PAUSE 500
  Counter = Counter + 1             'Increment Counter
  IF Counter < 5 THEN Alarm_Again   'Check counter
GOTO Main
```

Insert **DEBUG ? Counter** after the update to view the count

FOR-NEXT

- Since looping routines using counters is so prevalent, a specialized loop called the FOR-Loop or FOR-NEXT can be used in many instances.

- The general structure of a FOR-Loop is:

```
FOR variable = start_value TO end_value
   'Block of code to be repeated
NEXT
```

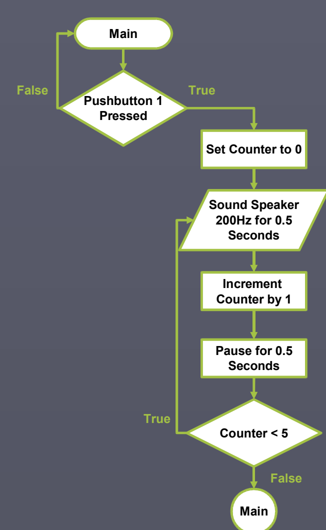
- The above example will increment or update by +1 each repetition starting from the start_value to the end_value. An optional STEP value may be used to update by other increments:

```
FOR variable = start_value TO end_value STEP value
   'Block of code to be repeated
NEXT
```


Repeating Alarm with FOR-Loop

- This example uses a FOR-Loop counter to sound the speaker 5 times when PB1 is pressed.

```
' <<<< INSERT SECTION 5 COMMON
 CIRCUIT DECLARATIONS >>>>>>

'Prog 6E: Looping with Counter Alarm using FOR-Loop
Counter VAR NIB      'Variable for counting

Main:
 ' If pushbutton 1 is pressed, then go sound alarm
 IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
 FOR Counter = 0 to 4
  FREQOUT Speaker, 500, 2000 'Sound the alarm
  PAUSE 500
 NEXT
GOTO Main
```



Note that the flowchart has not changed, only the method to code it.

Repeating Alarm with FOR-Loop Major Points

A variable appropriately sized is declared.

Counter is set to start value (0) upon start of FOR-Loop

Counter is updated by 1. If the count has not exceeded the end value (4), repeat sequence.

```
' <<<< INSERT SECTION 5 COMMON CIRCUIT DECLARATIONS
'Prog 6E: Looping with Counter Alarm using FOR-Loop
Counter VAR NIB                  'Variable for counting

Main:
  ' If pushbutton 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  FOR Counter = 0 to 4
    FREQOUT Speaker, 500, 2000 'Sound the alarm
    PAUSE 500
  NEXT
GOTO Main
```

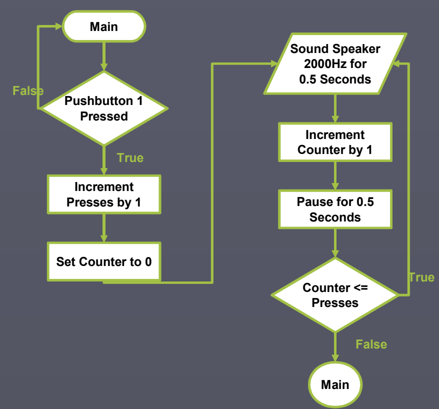

Speaker Tone with FOR-Loop

- The counter value in the FOR-Loop is often used within the code. Enter and run this example. Analyze it to determine how it works.

```
' <<<< INSERT SECTION 5 COMMON CIRCUIT DECLARATIONS >>>>
'Prog. 6F: Using FOR-Loop for frequency generation

Freq VAR WORD

FOR Freq = 500 to 4000 Step 100
  FREQOUT Speaker, 50, Freq
NEXT

END
```

Challenge 6D: Count to Presses with FOR-Loop

- Code a program that will keep track of the total number of times PB1 was pressed, and sound that number of tones (pressed once→ one beep, pressed twice → two beeps, etc) up to 15. A flow chart is provided to guide you. Use a FOR-Loop and a variable to keep track of total times pressed.



Solution

Subroutine Overview

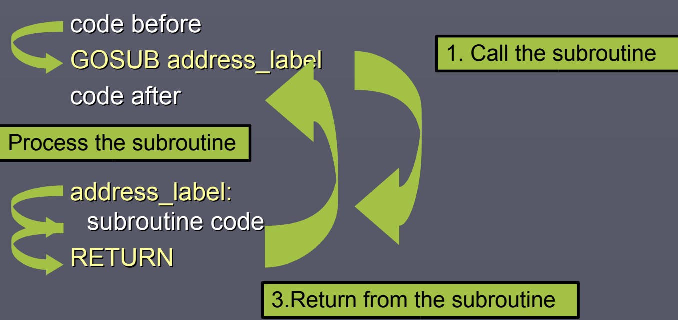
- So far in the examples a GOTO has been used to branch to another routine. When the code in the routine is complete, a GOTO was used to go to a specific label, namely Main.

- Often times a routine is called by multiple locations in a program, but instead of always branching back to a specific location, we need to the code to return to where it branched from.

- A GOSUB is a special type of branching instruction to call a routine. When the routine is complete, execution returns to the next statement after a call.

- The routine called by a GOSUB is known as a *subroutine* (GO SUBroutine).

- The basic structure and flow of a GOSUB call is:



Every GOSUB *must* end with a RETURN. The return points are maintained in what is called a *stack*. GOSUBs add to the stack, RETURNs take from the stack. GOSUBs without returns cause the stack to overflow and cause problems. Have you ever had your computer crash due to a 'Stack Overflow'?

Cleaner Coding with GOSUBS

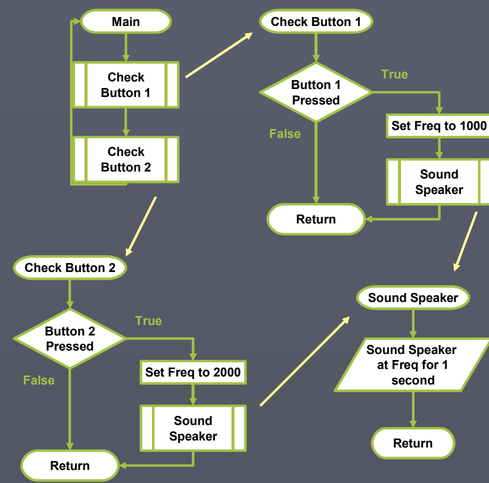
- A good program should be easy to read. Often the main loop only consists of GOSUB calls. The subroutines perform various actions required by the program. By reading through the main loop, the basic operation of the program can often be determined.

```
Main:
  GOSUB ReadInputDevice
  GOSUB ControlOutputDevice
GOTO Main

ReadInputDevice:
  'code
RETURN

ControlOutputDevice:
  'code
RETURN
```

Another good practice is to write subroutines that fit entirely on a single screen (25 lines) for easier reading and debugging.

Sounding Alarms with GOSUB

- This program uses GOSUBs to sound the speaker if either pushbutton is pressed.
- The 'Pre-Defined Process' block is used to denote a subroutine call, which is written as separate routine.



- Enter and run.
- Since the conditional IF-THEN calls a routine directly the code jumps over the GOSUB if the button is NOT pressed.
- With structured code it is easier to read and modify individual routines and add routines.

```
' <<<< INSERT SECTION 5 COMMON CIRCUIT DECLARATIONS
'Prog 6G: Sounding alarm with GOSUBs

Freq          VAR Word

Main:
  GOSUB Check_PB1                  'Call subroutine for PB1
  GOSUB Check_PB2                  'Call subroutine for PB2
GOTO Main

Check_PB1:             ' *** Sound alarm if PB1 pressed
  If PB1 = PB_Off THEN Check1_Done          ' If NOT pressed, jump
  Freq = 1000                               ' Set Tone
  GOSUB Sound_Speaker                       ' over GOSUB
Check1_Done:
RETURN

Check_PB2:             ' *** Sound alarm if PB2 pressed
  If PB2 = PB_Off THEN Check2_Done          ' If NOT pressed, jump
  Freq = 2000                               ' Set Tone
  GOSUB Sound_Speaker                       ' over GOSUB
Check2_Done:
RETURN

Sound_Speaker:         ' *** Sound speaker for alarm
  FREQOUT Speaker, 1000,Freq  'Sound speaker
RETURN
```

Flow with no buttons pressed.

```
Main:
  GOSUB Check_PB1                    'Call subroutine for PB1
  GOSUB Check_PB2                    'Call subroutine for PB2
GOTO Main

Check_PB1:            ' *** Sound alarm if PB1 pressed
  If PB1 = PB_Off THEN Check1_Done   'If NOT pressed, jump
  GOSUB Sound_Speaker                ' over GOSUB
Check1_Done:
RETURN

Check_PB2:            ' *** Sound alarm if PB2 pressed
  If PB2 = PB_Off THEN Check2_Done   'If NOT pressed, jump
  GOSUB Sound_Speaker                ' over GOSUB
Check2_Done:
RETURN

Sound_Speaker:        ' *** Sound speaker for alarm
  FREQOUT Speaker, 1000,2000         'Sound speaker
RETURN
```


Flow with PB1 pressed

```
Main:
  GOSUB Check_PB1                    'Call subroutine for PB1
  GOSUB Check_PB2                    'Call subroutine for PB2
GOTO Main

Check_PB1:            ' *** Sound alarm if PB1 pressed
  If PB1 = PB_Off THEN Check1_Done   'If NOT pressed, jump
  GOSUB Sound_Speaker                ' over GOSUB
Check1_Done:
RETURN

Check_PB2:            ' *** Sound alarm if PB2 pressed
  If PB2 = PB_Off THEN Check2_Done   'If NOT pressed, jump
  GOSUB Sound_Speaker                ' over GOSUB
Check2_Done:
RETURN

Sound_Speaker:        ' *** Sound speaker for alarm
  FREQOUT Speaker, 1000,2000         'Sound speaker
RETURN
```

Challenge 6E: Adding a Subroutine

- Add an alarm-operational indicator to Program 6G.
  - Code a subroutine to light LED1 for 0.25 seconds. Call the subroutine from your main loop.
  - Draw the modified main loop and new routine in flowchart form.
  - NOTE: The controller may operate so quickly you may not discern the LED blinking. Add a one second pause in the main loop to help see the blink.

Solution

Using the BRANCH Instruction

- It is common for programs to need to take a different action based on the value of some variable. Using IF-THEN's a sample program snippet may be:

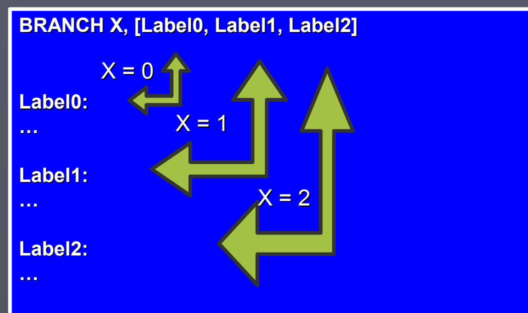
```
IF X=0 THEN Routine0
IF X=1 THEN Routine1
IF X=2 THEN Routine2
… and so on
```

- The BRANCH instruction may be used for much simpler (and memory-saving) way:

```
BRANCH X,[Routine0,Routine1,Routine2]
```

- The structure of BRANCH is:
  BRANCH offset,[label0,label1,label2…]

- Based on the value of the offset (0 to 255), execution will branch to the defined label.



- The following program will use PB1 to increment a variable, Press_Count, to a maximum value of 3 and branch to a routine based on X.

```
' <<<< COMMON CIRCUIT DECLARATIONS
'Prog 6H: Using BRANCH for display and sounds
Press_Count          VAR       NIB
X                    VAR       BYTE

Main:
  GOSUB Butn_UP_Count                     'Go check if PB1 pressed
  GOSUB Run_Routine                       'Go run routine for sounds/display
GOTO Main

Butn_Up_Count:
  IF PB1 = PB_OFF THEN End_Up_Count       'If button not pressed, end routine
  FREQOUT Speaker,500,4000                'Beep speaker to indicate button is down
  Press_Count = Press_Count + 1           'Add one to count
  IF Press_Count <4 THEN End_Up_Count     'If < 4, ok, otherwise reset
  Press_Count = 0                         'Reset count to 0
END_Up_Count:
  Return                                  'Return from routine
```

Code continues on next screen

```
Run_Routine:
  BRANCH Press_Count, [Blink, Wink, Cycle, Quiet]          'Branch based on offset
Return

Blink:                                  'Routine blinks LEDs and plays one tone
  TOGGLE LED1_Pin : TOGGLE LED2_Pin
  FREQOUT Speaker, 100,500
  PAUSE 250
Return

Wink:                                   'Routine winks each LED and plays 2 tones
  LED1 = LED_On : LED2 = LED_Off
  FREQOUT Speaker, 500,4000
  LED1 = LED_Off : LED2 = LED_On
  FREQOUT Speaker, 500,2000
Return

Cycle:                                  'Blinks LED and cycles frequency up
  FOR X = 1 to 20
    TOGGLE LED1_PIN
    TOGGLE LED2_PIN
    FREQOUT Speaker, X * 10, 200 * X
  NEXT
  PAUSE 500
Return

Quiet:                                  'Peaceful silence
  LED1 = LED_Off : LED2 = LED_Off
  PAUSE 500
Return
```

In Program 6H, as PB1 is pressed, the LEDs and speaker will provide 4 unique behaviors.

Challenge 6F: Adding a Branch Routine

- Add a 5th routine unique routine to Program 6H.

Solution

Saving Power – END & SLEEP

- We may not think much about the power used by the BS2 when running off a wall outlet, but when running on batteries it doesn't take long to deplete them.
- If our program doesn't need to operate continually, we can END it saving power.
- Also, our program may not need to be performing tasks continually. There may be times it may 'sleep' saving power.

END

- When the BS2 is running and performing instructions, it uses 8mA of current.
- By ending a program (using END), the BS2 goes into a low power mode and stops all instruction processing, consuming only 40uA of current. That's 200 times less current. **Note: This current draw does NOT take into account any loads being driven, such as LEDs.**
- The BS2 will not wake again until power is cycled or it is reset.

Sleep

- Sleep allows the BASIC Stamp to go into a low power mode for a set period of time.
  Sleep Period
  - **Period** is the amount of seconds to sleep, rounded up to the nearest 2.3 seconds.

- When 'Sleeping' the BASIC Stamp will wake momentarily every 2.3 seconds. During this time, all I/O will be switched back to inputs momentarily. This may have negative effects in some systems.

- In many cases, PAUSE can be replaced with SLEEP to conserve power.

Summary

- Sequential flow performs instructions from top of code to the bottom.
- Looping using GOTO can be used to branch to a new location identified in the program.
- An IF-THEN can be used to perform branches in a program based on a defined condition.
- The FOR-NEXT loop is a specialized loop for counting.
- GOSUB are used to branch AND return from routine.
- The BRANCH instruction can be used to branch to one of several locations based on a parameter.
- END and SLEEP can be used to conserve battery life.

End of Section 6

Section 7: Math and Data Operations

- Math Overview
- DEBUG Modifiers
- Basic Math Operations
  - Integer Math and Variable Sizes
  - Precedence of Operations
  - Maximum Workspace Limit
  - Signed Math Operations
- Boolean Operations and Math
  - Boolean Evaluations
  - Bitwise Boolean Operators
- LOOKUP Table
- Writing and Reading EEPROM
- DATA Statement

Math Overview

- The BASIC Stamp can perform many math operations. Some important limitations are:
  - The BASIC Stamp operates in *integer math*, meaning it does not calculate with decimal places.
  - In order to store negative values, WORD sized variables are required. Not all math operations support negative values.
  - The largest value of intermediate math operations is 65,535.
  - Math operations in a line of code are performed from left to right, not based on operator precedence, though parenthesis may be used for precedence.
  - There are ways around many math limitation discussed in this section, but they are beyond the scope of this tutorial.

DEBUG Modifiers

- So far this tutorial has been using DEBUG ? to display the contents of I/O or variables.
- The DEBUG instruction is quite flexible in the way it can display and format data by using modifiers.
- Data may be displayed as:
  - ASCII characters (No modifier).
  - Decimal values (DEC).
  - Hexadecimal values (HEX).
  - Binary values (BIN).

DEBUG Modifies Examples

Modifier Code	Output	Explanation
DEBUG 65	A	ASCII Value
DEBUG DEC 65	65	Decimal Value
DEBUG SDEC -65	-65	Signed Decimal
DEBUG BIN 65	1000001	Binary
DEBUG IBIN 65	%1000001	Indicated Binary - %
DEBUG IBIN8 65	%01000001	Indicated Binary with 8 places
DEBUG IHEX2 65	$41	Indicated Hexadecimal -$ with 2 places

DEBUG Formatting

- Strings of text may be displayed by enclosing in Double-Quotes.
  DEBUG "Hello"

- Use predefined constants for formatting lines and the screen:
  CR – Carriage Return to move to next line.
  HOME – Cursor to home position.
  CLS – Clear Screen

- Separate multiple values with commas.
  DEBUG CLS,"The value is ", DEC 65, CR

DEBUG Values Example

```
'Prog 7A: Displaying values from Potentiometer
'Adjust potentiometer to view values.

Pot VAR WORD
Main:
        HIGH 7 : PAUSE 10          'Prepare capacitor
        RCTIME 7,1,Pot             'Read RC Time
        Pot = Pot / 20             'Scale
                                   'Display Header
        DEBUG CLS, "ASCII  VALUE  BINARY  HEXADECIMAL", CR
                                   'Display Values
        DEBUG Pot,"      ", DEC3 Pot, "   ", IBIN8 Pot, "   ", IHEX2  Pot
        PAUSE 1000
GOTO Main
```


Basic Math Operations

- The BASIC Stamp can perform many math operations, such as:
  - + Add
  - - Subtract
  - * Multiple
  - / Divide
- Operations can be used in assignment to a variable:
  X = Y * 2
- Operations may also be used in instructions such as DEBUG:
  DEBUG DEC Y * 2

Integer Math and Variable Sizes

- The BASIC Stamp works in Integer Math, that is it does not compute with decimal places.
  DEBUG DEC 100 / 3
  Result: 33
- When assigning data to variables, ensure the variable is large enough to hold the result.
  X VAR BYTE
  X = 100
  X = X * 3
  DEBUG DEC X
  Result: 44.   Why?

Precedence of Operations

- Math operations are performed from left to right of the equation and NOT based on precedence of operators.
  DEBUG DEC 10 + 5 * 2
  Result: 30
- Parenthesis may be used to set precedence in calculations.
  DEBUG DEC 10 + (5 * 2)
  Result: 20

Maximum Workspace Limit

- The maximum value for any intermediate operation is 65,535. Care should be taken when performing complex calculations to ensure this value is not exceeded:
  DEBUG DEC 5000 * 100 / 500
  Result: 82 → 5000 * 100 exceeded limit.

- Grouping will not help in this case:
  DEBUG DEC 5000 * (100 / 500)
  Result: 0 → (100/500) is 0.2, or integer 0.

- Write the equation to prevent overflow or underflow without losing too much accuracy:
  DEBUG DEC 5000 / 500 * 100
  Result: 1000

Signed Math Operations

- The BASIC Stamp can work with negative values:
  - Word sized variables must be used to hold results for a range of –32,768 to +32,767.
  - Operations on negative values is limited, and generally should only be used with +, - and * when working with signed values.
  - Use the DEBUG SDEC (signed decimal) modifier to view signed values.
    X VAR WORD
    X = 100 * -20
    DEBUG SDEC X,CR
    Result: -2000

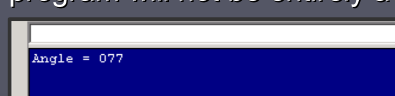
Some Other Math Functions

- PBASIC has a variety of other math functions. The following is a partial list.

ABS	Returns the absolute value	DEBUG DEC ABS -50 Result: 50
SIN, COS	Returns trigonometric value in binary radians from -128 to 128 over 0 to 360 degrees	DEBUG DEC SIN 180 Result: -122
SQR	Returns the square root integer value.	DEBUG DEC SQR 100 Result: 10
MIN, MAX	Returns the value limited to the specified minimum or maximum.	X = 80 DEBUG DEC X MIN 100 Result: 100
//	Modulus -- Returns the remainder. What is left after all the possible whole quantities are taken out?	DEBUG DEC 40 // 6 Result: 4 (40-36)

Challenge 7A: Scaling the Potentiometer

- Scale the input data from the potentiometer to display its position in degrees from 0 at the minimum position to 300 (or what you feel is appropriate for the movement of your potentiometer) at the maximum position.
  - Show the value in the DEBUG window as a decimal value with 3 places (i.e: 090).
  - Always display the data on the 1st line of the DEBUG window.
  - Hint: To scale the data, multiply the value by the new maximum and divide by the old maximum, but be careful of the math constraints!
  - Due to the non-linearity of the RCTIME results, your program will not be entirely accurate.



Solution

Boolean Operations and Math

- The BASIC Stamp can perform Boolean operations in 2 ways:
  - Evaluation of expressions
  - Bit manipulation
- States:
  - A state can either be considered TRUE or FALSE, or the bit states of 1 and 0 respectively.
  - If an input returns a 1, it would be considered to be TRUE.

Summary of Boolean Operation

T = TRUE (or 1)   F = FALSE (or 0)

NOT	Inverts the state	F = NOT T T = NOT F
AND	All must be true for the result to be true (need this AND that).	F = F AND F F = F AND T F = T AND F T = T AND T
OR	Any must be true for the result to be true (need this OR that)	F = F OR F T = F OR T T = T OR F T = T OR 1
XOR	Exclusive OR: Either, but not both, must be true for the result to be true	F = F XOR F T = F XOR T T = T XOR F F = T XOR T

Boolean Evaluations

- AND, OR and XOR may be used in IF-THEN statements to evaluate multiple conditions.

- In an IF-THEN, a value is true if it is greater than 0.
  - IF PB1 THEN Routine ' True if button is not pressed (Active-Low)
  - IF Pot THEN Routine ' True if Pot > 0

- Comparisons are evaluated to be 1 or True when the condition is met.
  - IF (PB1=PB_On) THEN Routine ' True if button is pressed ' (0 = 0 for Active Low)
  - IF (POT > 1000) THEN Routine ' True if potentiometer > 1000

- Using Boolean math, the results of individual evaluations are combined using the logic rules for an overall result.

- The following program will sound the speaker when BOTH pushbuttons are pressed (PB1 AND PB2 must be pressed).

```
'Program 7A: Boolean Evaluations
'*** Insert Common Circuit Declarations ***
Main:
  'Sound Alarm if both buttons are pressed
  IF (PB1=PB_On) AND (PB2=PB_On) THEN Alarm
GOTO Main

Alarm:
  FREQOUT Speaker,100,2000
GOTO Main
```

- **If neither button is pressed:**
  (PB1=PB_On) AND (PB2=PB_On)
  1 = 0 1 = 0
  False False
  False AND False = False
- **If only PB1 button is pressed:**
  (PB1=PB_On) AND (PB2=PB_On)
  0 = 0 1 = 0
  True False
  True AND False = False
- **If both buttons are pressed:**
  (PB1=PB_On) AND (PB2=PB_On)
  0 = 0 0 = 0
  True True
  True AND True = True

Challenge 7B:Boolean Evaluations

Use AND and OR evaluations:

2. Modify Program 7A to sound the alarm when either button is pressed. Solution
3. Modify Program 7A to sound the alarm if the potentiometer is greater than 500 or PB1 is pressed. Solution
4. Modify Program 7A to sound the alarm if the potentiometer is greater than 500 but only if PB2 is not pressed. Solution

Bitwise Boolean Operators

- While AND, OR, NOT and XOR may be used to evaluate expressions, there exist the Bitwise Boolean operators which may be used to evaluate or modify groups of bits.

- The bitwise operators also perform AND, OR, NOT and XOR, but on one or more bits in a nibble, byte or word length value in binary.

- Each bit in one expression is logically evaluated with the same bit position in a second expression.

- Take for example: %1010 | %1110 (| = OR)
  Each bit in the first nibble is OR'd with the same bit position in the second nibble (% indicates a binary value).

  %1010
  %1110
  %1110

- Where either column has a 1, the result has a 1.

- The following are the bitwise operators:
  & = AND
  | = OR (Typically Shift \ Key to get a bar - |)
  ^ = XOR
  ~ = NOT

- What would be the result of %1111 & %0100 ?
  Click for answer.

%0100

Inverting an Input

- One common use is to invert the state of a Active-Low input:
  IF IN8 = 0 THEN ….

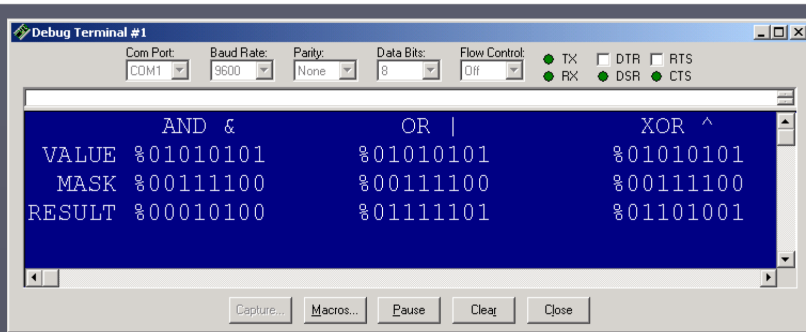
- It just doesn't seem 'natural' in programming sometimes to be active-low. When a button is pressed, it is more natural to want a HIGH than a LOW (active-low buttons and LEDs are common because of electrical properties of many devices).

- By using a bitwise NOT to invert the data:
  IF ~IN8 = 1 THEN ….

Masking BITs

- Bit masking is used to force a single bit, or bits, in a byte to a certain state using the Boolean bit-wise operators.
- For example, given any byte value, bit position 3 (starting with 0) may be force on with: ByteVal = ByteVal | %00001000.
- Program 7B will count from 0 to 255 in binary, and use the various operators for masking. Note the effect of the mask for each.


```
'Prog. 7B – Byte Masking
X VAR BYTE
MASK  CON %00111100
DEBUG CLS

Main:
FOR X = 0 TO 255
  DEBUG HOME, "        AND &         OR |          XOR ^",CR
  DEBUG " VALUE ",IBIN8 X,"     ",IBIN8 X,"       ",IBIN8 X,CR
  DEBUG "  MASK ",IBIN8 MASK,"     ",IBIN8 MASK,"       ",IBIN8 MASK,CR
  DEBUG "RESULT ",IBIN8 X & MASK,"     ",IBIN8 X | MASK,"       ",IBIN8 X^MASK,CR
  PAUSE 1000
NEXT
GOTO Main
```



- Use & 0 to force a bit LOW, | 1 to force a bit HIGH, and ^ 1 to complement (toggle) a bit.
- The BASIC Stamp works in bits naturally, and can actually change any bit state using a Bit modifier:
  ByteVal.BIT3 = 1.
- Masking is used more heavily in microprocessor based systems which are not bit-oriented.

LOOKUP Table

- A Lookup table is similar to branching in that a table is indexed, but in this case a value is stored in the variable.
  LOOKUP index,[value0,value2,…],variable

- For example, given the following:

  0

  LOOKUP I,[85,123,210,15],R

  If I = 0, 85 would be stored in R
  If I = 1, 123 would be stored in R
  and so on….

Playing a Tune with Lookup

- This program uses a table to look up 29 notes used to play a song. Notice how constants are used to define the frequency for each note.
- Can you recognize the song?

```
'Prog. 7C – Playing a tune with LOOKUP
I         VAR BYTE                    ' Counter for position in tune.
Freq      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], Freq
  FREQOUT 1,350,Freq                  ' *** Use 11 instead of 1 for Activity Board
NEXT
```


Challenge 7D: Playing Charge! Using Lookup

- The following program plays the Charge! Tune.

```
Speaker CON 1     '11 for Activity board
FREQOUT Speaker, 150, 1120
FREQOUT Speaker, 150, 1476
FREQOUT Speaker, 150, 1856
FREQOUT Speaker, 300, 2204
FREQOUT Speaker, 9,  255
FREQOUT Speaker, 200,1856
FREQOUT Speaker, 600, 2204
```

- Write a new program that uses 2 lookup tables to get the note's frequency AND duration and plays the tune (don't worry about using note constants).

There also exists an instruction called LOOKDOWN.
Use your help files to investigate the use of this instruction!

Solution

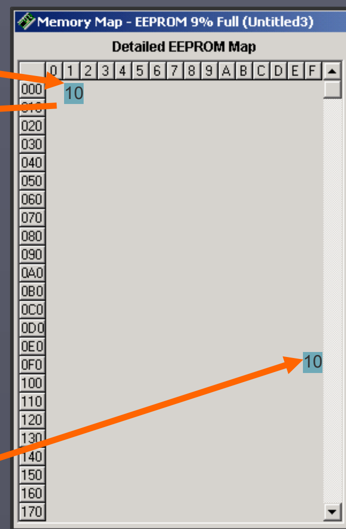
Writing and Reading EEPROM

- Data is typically stored using variables in RAM memory. RAM is volatile memory, in that when power is lost, the contents of RAM is destroyed.
- EEPROM memory is persistent in that it will maintain its contents in the event of a power failure. In fact, this is where your BS2 program is stored.
- WRITE and READ are instructions that allow the programmer to store and recall data in this non-volatile memory.

- WRITE allows the storage of a byte value into a memory address:
  WRITE address, byte value
- READ reads the contents of a memory address and stores it into the specified variable
  READ address, variable
- EEPROM has a finite number of WRITE cycles (> 1000). Continual, long term-use of WRITE is not recommended.

```
'Store 10 in address 1
WRITE 1,10
```

```
'Read contents of address 1
'and store in variable X
READ 1, X
```



- There are 2048 memory locations in the BS2.
- Program memory is stored from bottom of memory up. Writing over program space will corrupt your program.
- Memory address are shown in Hexadecimal. Hex may be used in memory addresses:

```
'Store 10 in address $0FF
WRITE $0FF,10
```


Testing READ and WRITE

- Enter and run the Program 7D on the next slide. program.
- It will display the current contents of the 1st 10 memory locations, write new values to those address, and re-read.
- Power down and back up the BASIC Stamp. Note that the values have not changed when power was removed.
- The Memory Map window of the Editor does NOT show changes to memory caused by WRITE.

```
'Program 7D: Reading and Writing to EEPROM
Addr      VAR      BYTE
Value     VAR      BYTE

DEBUG CR,"READING MEMORY ",CR :
GOSUB ReadMemory

PAUSE 2000 : DEBUG CR,"WRITING VALUES TO EEPROM ",CR
GOSUB WriteMemory

PAUSE 2000 :  DEBUG CR,"READING MEMORY AGAIN ",CR
GOSUB ReadMemory
END

ReadMemory:  FOR Addr = 0 TO 9              ' Cycle through 10  addresses
                 READ Addr, Value           ' Read value from EEPROM address, store in Value
                 DEBUG DEC Value," "        ' Display contents of Value
             NEXT
             RETURN

WriteMemory:  FOR Addr = 0 TO 9             ' Cycle through 10 addresses
                 Value = Addr               ' Set value equal to address to get a unique value
                 WRITE Addr, Value          ' Store contents of Value into EEPROM address
              NEXT
              RETURN
```


Storing a tune in EEPROM

- Program 7E puts READ and WRITE to use by storing notes (frequency values) in EEPROM memory to be played back as a 'tune'. Once the program is running:
  - Adjust the potentiometer to a desired frequency.
  - Press PB1 to store the note. LED1 will blink.
  - Store as many notes as you desire (up to 255).
  - Press PB2 to play your tune.
  - Power down and up, press PB2 to play your tune again.
  - You may test your tune, then store more notes.
  - When storing a note, the next address is filled with a 0 to mark the end of tune.
  - Since the address is defined as a byte variable, the maximum number of notes is 256 and well clear of the program storage area.

```
' Prog 7E: Stores notes for creating a tune.
' INSERT COMMON CIRCUIT DECLARATIONS
Addr        VAR BYTE     'EEPROM Address
Freq        VAR WORD     'Frequency
LED1 = LED_Off : LED2 = LED_Off
Main:
  GOSUB FindTone
  IF PB1=PB_ON THEN StoreNote
  IF PB2=PB_ON THEN PlayTune
GOTO Main

FindTone: 'Play note based on pot. Position
  HIGH Pot_PIN : PAUSE 1
  RCTIME Pot_Pin,1, Freq
  FREQOUT Speaker, 500, Freq
  RETURN

StoreNote:   ' Store freq. In EEPROM.
             ' Divide by 25 to condense to byte
  WRITE Addr, Freq / 25 MIN 1 MAX 255
  Addr = Addr + 1      ' Next address
  WRITE Addr, 0        ' Store 0 to mark end
  LED1 = LED_On        ' Blink LED1
  PAUSE 500 :   LED1 = LED_Off
GOTO Main
```

```
PlayTune:
  LED2 = LED_On      'Turn on LED2
  PAUSE 1000
  Addr = 0           ' Start of EEPROM
TuneLoop:
  READ Addr, Freq    'Read EEPROM
  IF Freq = 0 THEN EndTune       ' 0 = end of tune
  FREQOUT Speaker, 500,Freq * 25   'Play note
  Addr = Addr + 1    ' Increment and do next note
GOTO TuneLoop

EndTune:
  PAUSE 2000
  LED2 = LED_Off     'LED2 off
  GOTO Main
```


DATA Statement

- The DATA statement may used to predefine the contents of EEPROM memory locations. The general format of a DATA Statement is:
  Label DATA value, value, value
  or
  Label DATA "String"
  The use of Labels is optional:
  DATA "String"

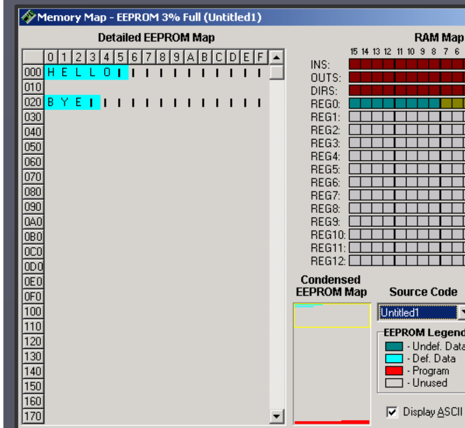
- The values specified are stored starting at the 'top' (address 0) of the BASIC Stamp's EEPROM unless a starting address is provided:
  Label DATA @address,"String"

- The starting EEPROM address may then be accessed by use of the READ instruction:
  READ 0,variable
  READ Label, variable

- Label points to the STARTING address. Subsequent addresses can be accessed by adding to the starting address:
  READ Label+1, variable

```
' Prog. 7F: Using DATA to store
X         VAR       NIB
Value     VAR       BYTE
  ' Store string starting at address 0
  ' or next available location
MSG1      DATA      "HELLO",CR

  ' Store string starting at address Hex $20
MSG2      DATA      @$020,66,89,69,13

  ' Loop through and read/display
  ' data in MSG1
FOR X = 0 to 5
          READ MSG1+X, Value
          DEBUG Value
NEXT
  ' Loop through and read/display
  ' data in MSG1
FOR X = 0 to 3
          READ MSG2+X, Value
          DEBUG Value
NEXT
```

Open the Memory Map to see where the data is stored. Note unused bytes in the row are filled with binary 0 (null).



Check 'Display ASCII' to view data as ASCII Characters.

Using DATA for a Phonebook Dialer

- The code on the following screen uses DATA Statements to create a phone book program.
- The user selects a name using PB1, then dials that person's number using PB2.
- The DTMFOUT instruction generates tones used in dialing a telephone. **NOTE: This is for simulation only. Refer to your BASIC Stamp Manual for instructions on interfacing the BS2 to the telephone system!**

```
'Prog. 7G: DATA Phonebook Dialer
' INSERT COMMON CIRCUIT DECLARATIONS
DATA    @$00,"BILL"
DATA    @$10,"555-1234"
DATA    @$20,"PARALLAX"
DATA    @$30,"1-888-512-1024"
DATA    @$40,"JIM"
DATA    @$50,"555-4567"
DATA    @$60,0
Name    VAR     BYTE

X       VAR     BYTE
C       VAR     BYTE
Name = $00

Main:
  DEBUG CLS,"Press PB1 to select name.",CR
  DEBUG "Press PB2 to dial number.",CR
  FOR X = 0 TO 15
    READ Name+X,C
    IF C = 0 THEN EndName
    DEBUG C
  NEXT
EndName:
  PAUSE 1000
  IF PB1=PB_ON THEN NextName
  IF PB2=PB_ON THEN DialNumber
GOTO Main
```

```
NextName:
  Name = Name + $20
  READ Name,C
  IF C <> 0 THEN Main
  Name = $00
GOTO Main


DialNumber:
  DEBUG CR
  FOR X = 0 to 15
    READ Name+$10+X,C
    IF C = 0 THEN EndDIAL
    DTMFOUT Speaker,[C]
    DEBUG C
   NEXT
EndDial:
   PAUSE 1000
GOTO Main
```


Challenge 7E: Analyze and Modify Phonebook

- 1) Analyze program 7G and answer the following questions:
  - How does the program identify the end of the name or number?
  - How does the program move to the next name in the listing?
  - How does the program access the correct number to dial?

Solution

- 2) Add an entry for your own name and number and test.

Solution

Summary

- Math operations are limited to integer math with intermediate operations of 65,535.
- Boolean operators, such as AND, OR can be used logically combine multiple evaluations.
- Boolean bitwise operators can be used in modifying individual bits in a group.
- READ, WRITE and DATA instructions can be used to store and read information from non-volatile EEPROM memory.

Section 8: Data Communications & Control

Data Communications Overview

- Simple devices such as switches, LEDs, and buzzers are pretty simple to control or read since the data is very simple – On or Off.

- More sophisticated devices require the transfer of larger amounts of data. These devices include Analog to Digital converters, real time clocks, numerous other devices and, of course, other controllers.

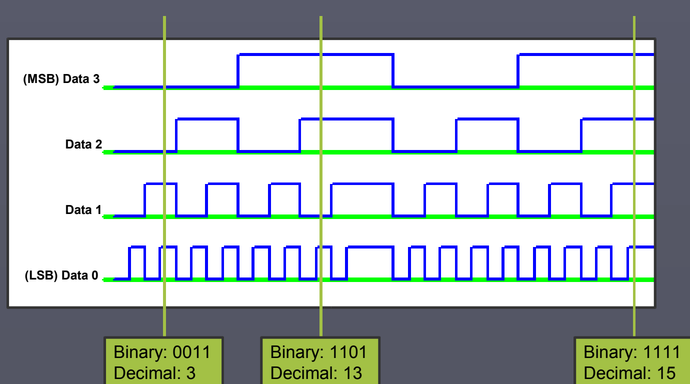
- There are 2 primary means to transfer data between devices:
  - **Parallel:** All bits are transferred simultaneously. Printers using the Centronics printer port and internal computer data transfers are examples of parallel communication.

  - **Serial:** Bits are transferred one at a time. Serial transfer examples include your mouse, USB ports, TV remote controls, and programming of your BASIC Stamp through the serial port.

Parallel Communications

- In parallel communications as many data lines are required as needed to transfer the entire 'chunk' of data.
  - Transferring a nibble? 4 data lines are required.
  - Transferring a byte? 8 data lines are required.
  - Transferring a 16-bit word? 16 data lines are required.
- Additional lines may be needed for common grounds, synchronization and control.

- The image below shows the parallel transfer of a nibble. At any point the individual bits comprise a value.



BASIC Stamp Parallel Communications

- In this example, parallel communications will be performed to send data from one BASIC Stamp to another.
- In this section, we will be modifying the circuit, and we can use unpopulated boards for this example.
- Following this, the board will be re-populated with 1 LED (P8), and the buzzer.

- Connect lines between 2 BASIC Stamps as follows:

Stamp #1 – Transmitter	Stamp #2 – Receiver
P0	P4
P1	P5
P2	P6
P3	P7
Vss	Vss

- Program each BASIC Stamp with the appropriate program. Monitor the the DEBUG window from the receiver. Since only 4 bits are used, only 16 unique combinations may be transferred.

```
' Program 8A-T: Nibble Parallel Transmitter
X   VAR   NIB           'Variable for counting
DIRA = %1111            'Set nibble A as outputs
                        '(P0 – P3)

Main:
 FOR X = 0 to 15        ' Count 0 to15
   OUTA = X             ' Place data on Nibble A
   PAUSE 500
 NEXT
GOTO Main
```

```
' Program 8A-R: Nibble Parallel Receiver
Y   VAR   NIB
DIRB = %0000            ' Set nibble B as inputs
                        ' (P4 - P7)

Main:
 Y  = INB               ' Read nibble B
 DEBUG ? Y              ' Display incoming data
GOTO Main
```


- When program 8A is ran, and the receiver is monitored, a long stream of 0's, then 1's, then 2's, etc, is seen. This is because there are no means in the example to inform the receiver when it it getting new data.

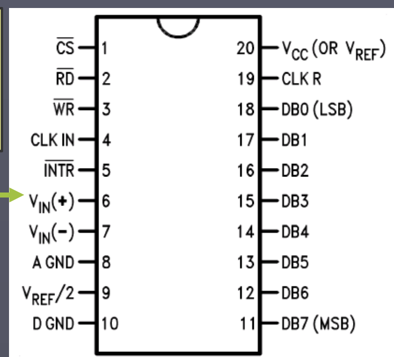
- A control line, or lines, could be used to control flow and have the devices communicate when new data is ready and when it has been read.

- Many devices transfer data in parallel formats. An example is the ADC0801 Analog to Digital converter which converts an analog voltage to a Parallel byte.

The ADC0801 has a number of control pins for communication control. Please see an ADC0801 data sheet for more information.

Link to datasheet

Analog Voltage In



Digital Data Out

Parallel Transfer Summary

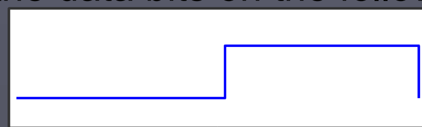
- Parallel data transfer is very simple and very fast. Data is simply placed-on or read-from a 'bus'. Microcomputers use parallel data transfers between the microprocessors and all the internal devices such as memory, sound cards, hard drives, CD-ROMS, etc.

- A major disadvantage is the large number of lines (and thus I/O) required. This method does not loan itself to communications over any appreciable distance.

Serial Communications

- Serial communications sends a 'chunk' of data a single bit at a time. →01101010→ This alleviates the need for numerous data lines, but leads to other problems.

- Identify the data bits on the following trace:



- Is the data 01? 000111? 00001111? Without more information, we really can't say for sure… and neither can a controller.

- In serial communications there are 2 major categories:

  - **Synchronous:** Transmitter and receiver are locked and synchronized in the transfer of data.

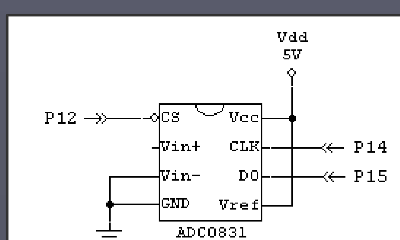
  - **Asynchronous:** Transmitter and receiver are not locked, but in agreement of the transmission timing.

Synchronous Communications - SHIFTIN

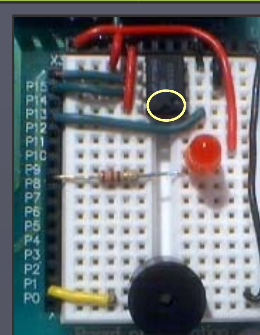
- In synchronous communications often a separate clock line is used to lock the transmitter and receiver together.
- One or the other sends clock pulses indicating individual bit transfers.
- We will work with a device that transfers data to the BASIC Stamp using synchronous communications. The ADC0831 serial analog to digital (A/D) converter. Link to data sheet

ADC0831 8-Bit Serial A/D

- Connect the ADC0831 as follows:



Note that the IC is placed with top-down (notch at bottom) for easier wiring.



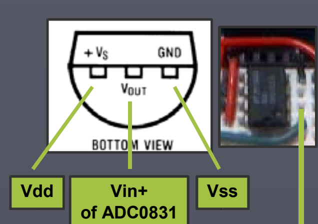
The Activity Board can accept an ADC0831. Note that slightly different I/O are used in the code.

Vin+

- The ADC 0831 converts an analog voltage, typically 0-5V, to an 8-bit digital value.
- Pin descriptions:
  - **CS:** Chip Select – Enables operation. Active low to enable.
  - **Vin+:** Analog voltage input to be converted.
  - **Vin-:** Negative reference (ground for this circuit).
  - **CLK:** Clock pulses are applied here to shift out each bit.
  - **D0:** Data Output - Data appears at this output following each clock pulse.
  - **Vref:** Upper reference voltage (+5V for this circuit).

Connecting a Temperature Sensor

- Now that an A/D is connected, we need some source of voltage to convert. A source could be a voltage divider, a potentiometer, or many varieties of sensors.
- One such sensor is the LM34, which converts temperature to a voltage where the output is 0.01V per degree Fahrenheit. At 100F, the output will be 1.0 Volt. Connect the LM34 as indicated.




Leaded LM34 from Parallax

After connecting, if the LM34 becomes very warm, disconnect and reverse the power leads. (It is common for people to get these backwards).

Operational temperature for a common LM34 style is +32F to +300F.

Reading the ADC0831 with SHIFTIN

```
'Prog. 8B: Use SHIFTIN command to read the ADC0831 serial ADC

ADres    VAR     BYTE              ' A/D result (8 bits)
ADcs     CON     12                ' A/D enable (low true)
ADdat    CON     15                ' A/D data line  *** Activity board use 14
ADclk    CON     14                ' A/D clock      *** Activity board use 15

Main:
        LOW ADcs                                  ' Enable ADC
        SHIFTIN ADdat,ADclk,msbpost,[ADres\9]     ' Shift in the data
        HIGH ADcs                                 ' Disable ADC
        DEBUG CLS,"Dec: ", DEC ADres              ' Display the result in decimal
        DEBUG "    Bin: ", IBIN8 ADres            ' Display the result in binary
                ' Display in hundredths of volts
                ' Current Value * New Maximum (500) / Old Maximum (255)  or 50/26
        DEBUG "    Hundredths of Volt = ", DEC ADres * 50/26 ,CR
        PAUSE 500                                 ' Wait a 0.5 Seconds
GOTO Main
```

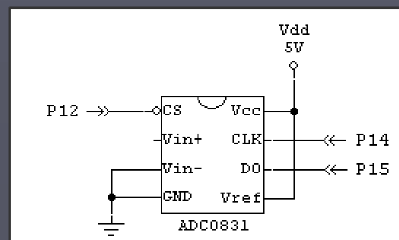

- A general form of SHIFTIN is:
  SHIFTIN *Dpin, Cpin, Mode, [Variable {\Bits}]*
  - **Dpin** defines the pin what will contain the data.
  - **Cpin** defines the pin that will be used for clocking.
  - **Mode** defines the format the data will take in relation to the clock (more on this soon).
  - **Variable** is the variable to hold the data being clocked-in.
  - **Bits** defines the number of bits to be shifted – thus the number of clock pulses to send.

- The example code is:

```
LOW ADcs                                  ' Enable ADC
SHIFTIN ADdat,ADclk,msbpost,[ADres\9]     ' Shift in the data
HIGH ADcs                                 ' Disable ADC
```

- When the SHIFTIN instruction is performed, the BASIC Stamp will send clock pulses on ADclk (P14), accept the data on ADdat (P15), and shift the results into variable ADres for 9 bits. The 1st bit is unused, providing 8 bits of data.

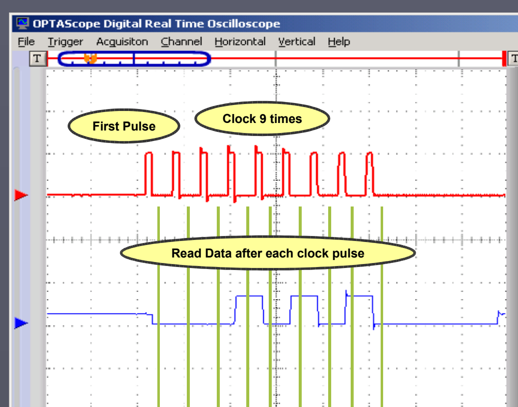
- ADcs (P12) is set low to enable the IC.



O-Scope Capture of SHIFTIN

P14 – ADclk: Clock line

P15 – ADdat: Data line



ADres = 0 0 0 1 0 1 0 1 0

MSB LSB

1st bit is not valid data

= decimal 42
* 5V / 255 to scale
= 0.82V

At 100uS/Divisions, the data rate is approximately 17,000 bits per second or 17 Kbps

- Notice that first a clock pulse was sent, then the data was read on the data line from MSB to LSB. This is why the setting of MSBPOST was used in the SHIFTIN mode. MSB is first data, read after pulse (post).
- The other options for the mode are:
  MSBPRE
  LSBPOST
  LSBPRE

- Program 8B displays the temperature in hundredths of volts.
  ADres * 50 / 26
  Since 1V = 100F this calculation can also be used to calculate the temperature from the byte value.
- With a total maximum range of 0 – 500F, and an 8-Bit A/D which has a maximum value of 255, the resolution of our circuit is:
  500 / 255 or 1.96F (2F in integer math)
- The LM34 cannot measure 500F, but the A/D is scaled for 0-5V so 500 is used in the calculations.

SHIFTOUT

- SHIFTOUT works similarly to SHIFTIN, but is used to send data to a device, such as an external serial EEPROM.
  SHIFTOUT *Dpin, Cpin, Mode, [OutputData {\Bits}]*
- Note that in both instructions, the BASIC Stamp has control of the clock line.

Synchronous Communications Summary

- Using SHIFTIN and SHIFTOUT for synchronous communications, separate clock and data lines are required. The clock is used to signal the position of each bit.
- Since a separate line is used for bit position signaling, relatively high data rates can be achieved.
- This mode of communication still requires at least 2 lines – Data and Clock.

Challenge 8A: Temperature Alarm

- Program the BASIC Stamp to monitor the temperature and alarm the buzzer at 2000Hz for 1 second if a temperature exceeds 100.

Solution

Asynchronous Communications

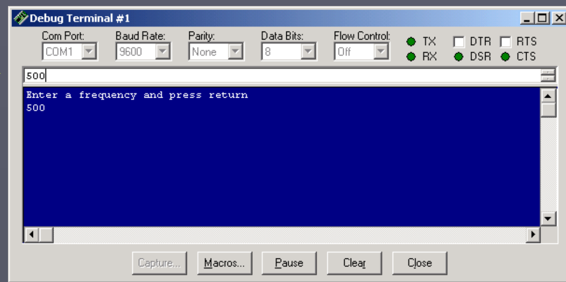
- Using Asynchronous communications, a single line may be used. The position of each bit in the data stream is based upon an agreement in timing between the transmitter and receiver.
- This is a very popular form of serial communications between devices, such as the serial port on your computer. When you program the BS2 or use DEBUG, the BASIC Stamp is using RS-232 Asynchronous communication.
- Another means is using the SEROUT and SERIN instructions.

SERIN Instruction

- A general form of the SERIN instruction is:
  SEROUT *Tpin Baudmode, Timeout, Label,[Variable]*
- Where:
  - Tpin is the pin to transmit out from. 16 may be used to transmit from the programming port.
  - Baudmode is a value which defines characteristics of the data transmission, such as baud rate.
  - Timeout is the length of time to wait for data before continuing.
  - Label defines where to branch to if a timeout occurs.
  - Variable holds the incoming data.
- Let's test a program that uses SERIN to control the frequency of the buzzer.

Controlling the Buzzer's Tone

```
'Prog. 8C: Use SERIN instruction to control buzzer's tone
Rpin      CON      16                  ' From programming port
BMode     CON      84                  ' BAUD mode -- Use 240 for BS2SX, BS2P
MaxTime   CON      3000       ' Timeout Value – 3 seconds
Freq      VAR      WORD                ' Hold incoming data

Main:
  DEBUG CLS,"Enter a frequency and press return  ",CR       ' Request Freq
  SERIN RPin, BMode, MaxTime, Timeout, [DEC freq]           ' Await serial data
  DEBUG "Playing tone at ", DEC Freq, "Hz.",CR              ' Notify user
  FREQOUT 1,1000,Freq                                       ' Play tone
  GOTO Main
Timeout:
  DEBUG "Timeout!", CR                                ' Notify user of timeout
  PAUSE 500                                           ' Short wait
GOTO Main
```

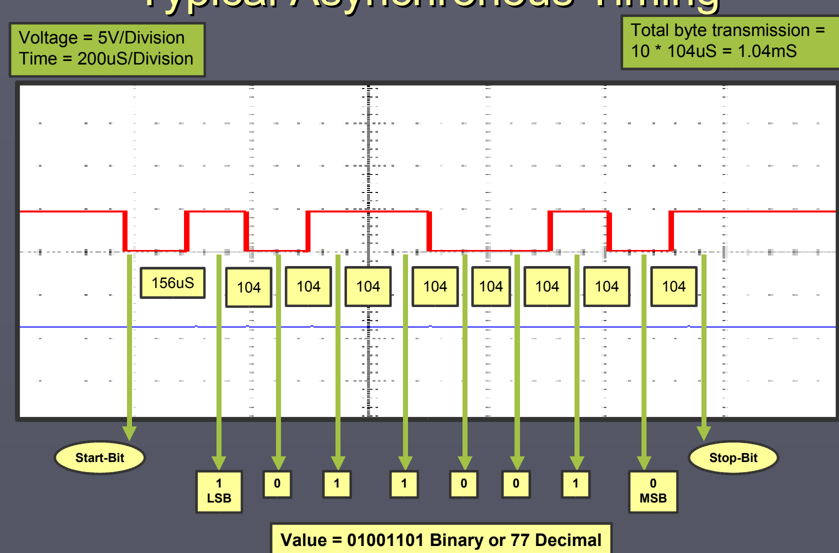


- When the program runs, enter a value in the text box and hit Enter.
- Notice what occurs when 3 seconds elapses without entering data.

- When data is entered in the textbox, it is transmitted from the serial port to the BASIC Stamp which accepts it with the SERIN instruction.
- As each character is entered it is seen in the Debug window output because the programming port *echoes* data back.
- The SERIN instruction uses **[DEC Freq]** to accept a string of characters for a value. If **[Freq]** were only used, only one character of data would be accepted.
- The BASIC Stamp does NOT buffer data. The SERIN instruction must be awaiting data for it to be processed.

- When the transmitter sends data, it begins by sending a *start-bit*, then the data bits (LSB to MSB) at set intervals, and finally a *stop-bit* to complete the frame of data.
- Transmission speeds are described by a BAUD rate. A common BAUD rate is 9600. This correlates to 9600 bits per second for RS-232. The inverse of this value (1/9600) is 104uS which is width of each bit, or bit interval.
- The receiver will sense the start-bit:
  - The first bit will be collected at the 1.5x the interval to be at the center of the first data bit (1.5 x 104uS = 156uS).
  - Each successive bit will be collected at the transmitted interval.

Typical Asynchronous Timing



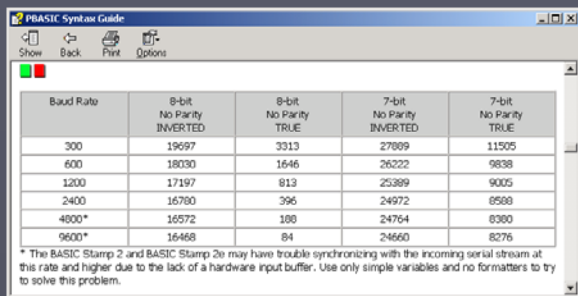
RS-232 Standards

- While serial communication may be performed using any of the standard I/O, they are not fully compliant with the RS-232 standard.
- RS-232 defines a **logic level 0 as +3 to +25V**, and a logic **level 1 as –3V to –25V**. This is an inverted signal with non-TTL voltage levels.
- The programming port has circuitry to invert the signal to make it more RS-232 compliant.
- The BS2 can send data as inverted or non-inverted (true). This example used non-inverted since the programming port inverts it with hardware.

- Other major factors in defining the transmission are:
  - Baud Rate – Speed at which the data is transmitted.
  - Number of data bits – Typically 8.
  - Number of Stop bits – Typically 1.
  - Whether Parity is used. Parity is an additional bit sent to check the data frame for errors. Even (E), Odd (O) or None (N) are common choices. Typically error checking is performed in other ways and the parity bit is not used.
  - A short hand method of summarizing the transmission mode is:
    Baud Bits-Parity-stop bits
    9600 8-N-1

- The mode used in transmitting or receiving are defined with a unique number.

- The help files summarize common values. Note that since different BS2 styles operate at different speeds, it is important to ensure you are using the correct table.



Baud Rate	8-bit No Parity INVERTED	8-bit No Parity TRUE	7-bit No Parity INVERTED	7-bit No Parity TRUE
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800*	16572	188	24764	8380
9600*	16468	84	24660	8276

* The BASIC Stamp 2 and BASIC Stamp 2e may have trouble synchronizing with the incoming serial stream at this rate and higher due to the lack of a hardware input buffer. Use only simple variables and no formatters to try to solve this problem.

SEROUT

- Just as SERIN can be used to accept data, SEROUT can be used to transmit data.
  SEROUT *Tpin, Baudmode, [OutputData]*

- In our example, DEBUG was used to send data to the computer. DEBUG is simply a specialized SEROUT which is defined for P16, 9600 8-N-1.

- DEBUG also automatically opens the DEBUG window. Using SEROUT we must manually open and configure the DEBUG Window. 

Challenge 8B: Using SEROUT

1) Program 8C used DEBUG to send data to the PC.
Replace all DEBUG instructions with SEROUT instructions.
Sample: DEBUG "The value is: ", DEC X,CR
Would become:
SEROUT TPIN, BMode, ["The value is: ", DEC X,CR]
P16 is used for both transmitting and receiving with the programming port. Test your program with the DEBUG window.

Solution

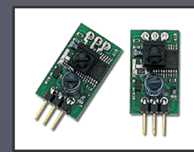
2) Change the BAUD rate to 1200 BAUD, 8-N-1, True for transmission and reception. Test.

Solution

Other SEROUT Uses

- Since SEROUT only requires a single line for data transfers, it is a very popular choice for communications.
- Data may be easily transferred between BASIC Stamps electrically:

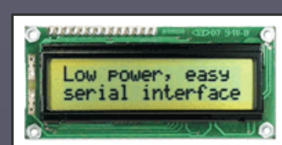
- Optically, such as using infrared.
  LINK



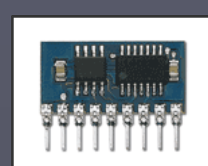
- Using RF Transmitters/Receivers
  Link

- For Modems
  Link

- For Serial LCD Modules
  Link



- With Motor Controls
  Link



- And many other devices including digital voltmeters and GPS units.

Asynchronous Summary

- Asynchronous communications are very popular because of the ability to transfer data with only a single line, making RF, optical and other forms possible easily.
- The data transfer is based on agreed timings and other characteristics.
- It is good for moderate data transfer speeds.

Pulse Width Data

- Sometimes data may be represented by the width of a byte instead of bits within a stream.
- Some devices operate on sending or receiving a pulse width.
- The BASIC Stamp can generate a pulse using PULSOUT and capture a pulse using PULSIN.

Using PULSOUT

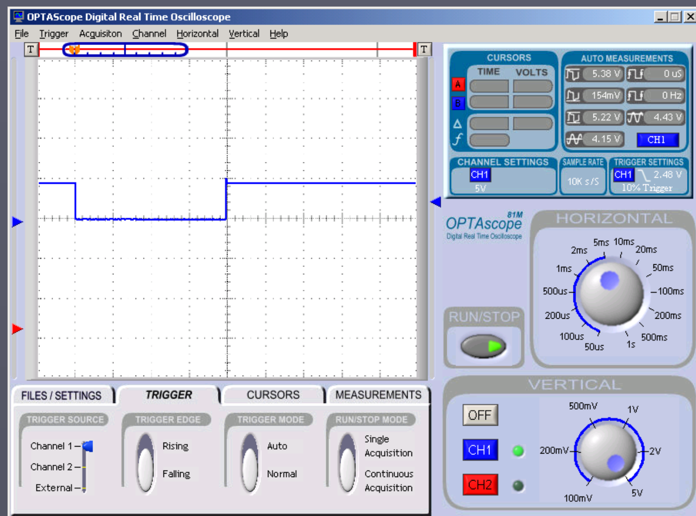
- PULSOUT sends a pulse for the defined period.
PULSOUT pin, period
- The pulse will be the opposite state of the current state of the output.
- The period is a value between 1 and 65535. The length of the period is dependent on the style of BS2:
  - BS2 and BS2E: 2uS
  - BS2SX: 0.8uS
  - BS2P: 0.75uS

- The following is a simple program to light the LED using PULSOUT over a range of periods.

```
'Prog. 8D – Lighting an LED with PULSOUT
X VAR WORD
HIGH 8                          'Start the LED in OFF state

Main:
  FOR X = 1 to 65000 STEP 1000
  PULSOUT 8, X                  ' Pulse LED for period of X
  DEBUG ? X                     ' Display value of X
  PAUSE 500                     ' Short Pause
NEXT
GOTO Main
```

- How would the operation of this circuit be different if it began with LOW 8?

O-Scope Capture of PULSOUT

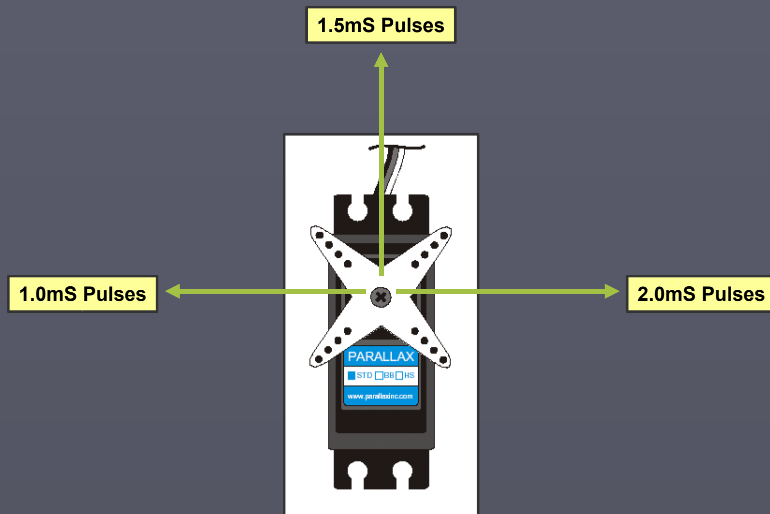


With the BS2, a period of 1 is 2uS. If this pulse lasts 20mS (5mS/div), what was the period?

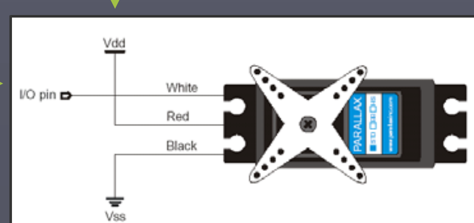
10,000

Positioning a Servo with PULSOUT

- A very common use of PULSOUT is in motion control of a servo.
- A non-modified servo operates by moving the rotor to an absolute position defined by the length of a pulse.
- A pulse width from 1mS to 2mS define a position for the servo between 0 and 180 degrees.



- The servos is powered from a high-current Vdd source (+5), though it can handle up to 6V for greater force. The on-chip BS2 regulator does NOT supply sufficient current to drive the servo. If your board has a separate regulator, you will be able to drive the servo.
- The pulse stream is directly from a BS2 I/O.



- The BOE has headers for direct servo connections from P12-P15.

Lead Color Code:
White
Red
Black

- The power connection to the servo header is Vin. If you are powering your BOE from a source greater than 6V the servo will be damaged. 4AA batteries are the recommended source of power.

Controlling Servo Position

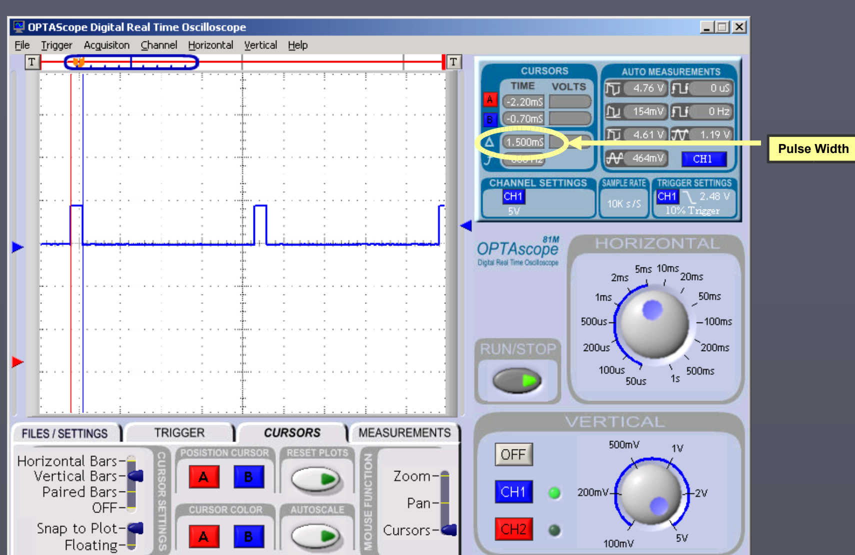
- With the BS2 and BS2 the period over which the servo is controllable for 1mS – 2mS is 500 to 1000. For the BS2SX and BS2P the period is 1250-2500.

- Program 8E uses serial communications with the DEBUG window to allow you to enter the angle to move the servo.

- Notice that a pulse train is sent by looping the PULSOUT to provide the servo time to move to the position. A pause of at least 20mS is required between pulses.

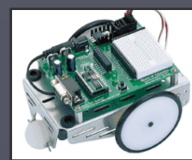
```
'Prog. 8D: Controlling a servo using PULSOUT
RpinCON                 16                      ' From programming port
BMode       CON         84                      ' BAUD mode -- Use 240 for BS2SX, BS2P
MaxTime     CON         3000                    ' Timeout Value – 3 seconds
Servo       CON         13                      ' Servo I/O
Angle       VAR         WORD                    ' Hold incoming data
Period      VAR         WORD                    ' Hold conversion to period
X           VAR         NIB                     ' Counting variable

LOW SERVO                                       ' Start I/O out low for HIGH pulses

Main:
  DEBUG CLS,"Enter an angle (0-180) and press return  ",CR    ' Request angle
  SERIN RPin, BMode, MaxTime, Timeout, [DEC Angle]            ' Await serial data
  Period = Angle * 28 / 10 + 500                              ' Convert to persiod (BS2/E)
'  Period = Angle * 7 + 1250                                  ' Convert to period (BS2SX/P)
  DEBUG ? Period                                              ' Display period
  FOR X = 0 to 15                                             ' Send pulse train of 16 pulses
   PULSOUT Servo, Period
   Pause 20                                                   ' Pause between pulses
 NEXT
Pause 1000
GOTO Main

Timeout:
  DEBUG "Timeout!", CR                                        ' Notify user of timeout
  PAUSE 500                                                   ' Short wait
GOTO Main
```


Pulse Train Capture



Modified Servos

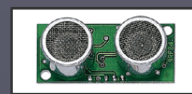
- The standard servo has motion only over a limited range (0-180 degrees). Internal to the servos is a feedback system and mechanical stops.
- In modified version used by the BOE-Bot robot, the feedback network and stops are removed for full, continuous motion as use as a wheel motor. The center value (750) is a dead stop, above and below this values will turn clockwise or counter clockwise at different speeds.



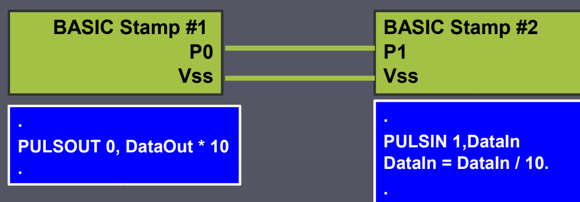
PULSIN

- The BASIC Stamp also support the measuring of a pulse using PULSIN.
  PULSIN *Pin, State, Variable*
  - Pin is the pin on which the pulse will be measured.
  - State is the desired pulse state to measure. 1 for a HIGH pulse, 0 for a LOW Pulse.
  - Variable is where the period of the pulse is stored.

- Several sensors report their readings by way of a pulse proportional to their reading.
  - Ultra Sonic Range Finder LINK



- PULSOUT and PULSIN can also be used for effective communications between BASIC Stamps, though 'Stretching" the pulse slightly is recommended for accuracy.



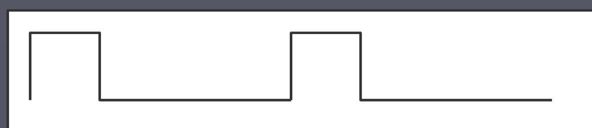
Pulse Width Modulation

- Pulse Width Modulation is used for device control at varying levels.

- Instead of having an output ON or OFF, PWM pulses the output to effectively control the percentage of time the output is on.

- This output may be used to drive DC loads at a variable rate or voltage.

- PWM works by controlling the HIGH to LOW ratio in a period of time.
- In this wave form, it is high 25% of the time and low 75% of the time. The average power applied to a load would be 25% of maximum. The average voltage would be 25% of the maximum. Vave = %Duty * Vmax = 25% * 5V = 1.25V



- Alternately, a 75% duty cycle would be high 75% of the time. 75% * 5V = 3.75V

- When controlling the BASIC Stamp, Duty is expressed as a value from 0 (0%) to 255 (100%).
- The BASIC Stamp PWM waves produced will not be as 'clean' as discussed, but will produce the desired effects.

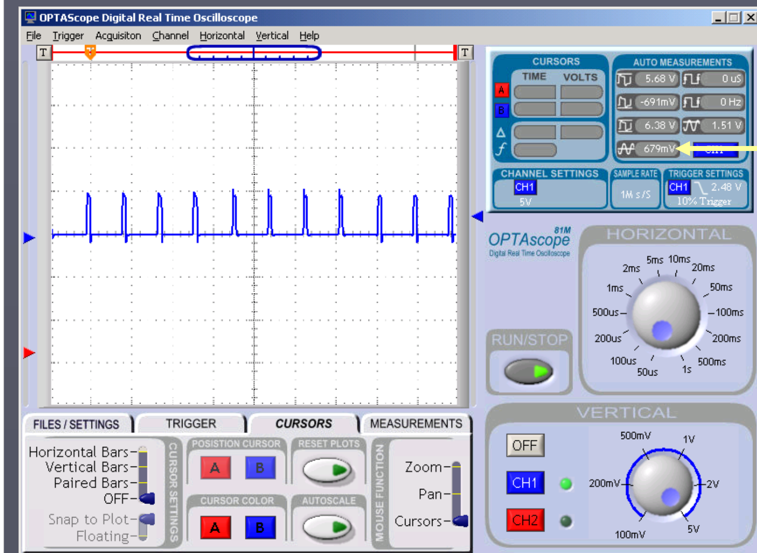
PWM Instruction

- **PWM *Pin, Duty, Cycles***
  - **Pin** is the pin to use for output.
  - **Duty** is the value from 0 to 255 which defines the amount of time high.
  - **Cycles** is the number of repetitions to drive the output, 0-255.
- Program 8E will once again use the DEBUG interface to allow entering a PWM value for testing.
- An LED does not have good linearity for brightness, but a high duty will cause the LED to light dimly, and a low duty will light it more brightly. Remember, the LED is connected active low.

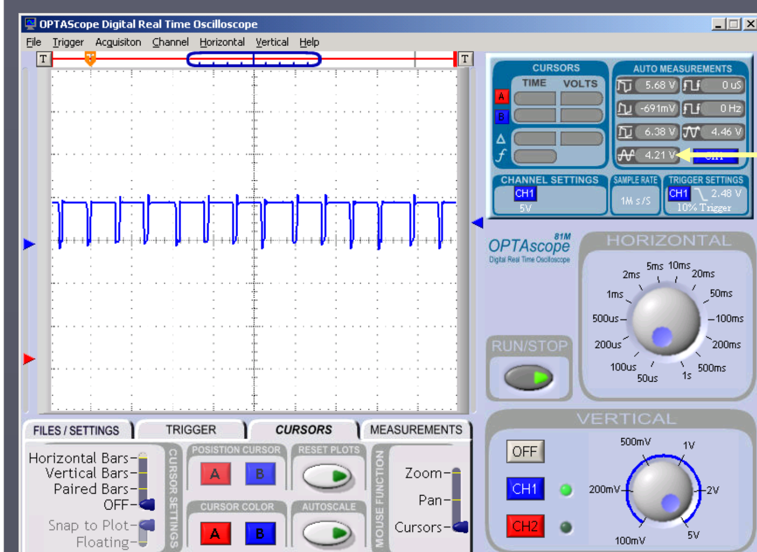
Driving an LED with PWM

```
'Prog. 8E: Controlling an LED with PWM
Rpin      CON    16          ' From programming port
BMode     CON    84          ' BAUD mode -- Use 240 for BS2SX, BS2P
MaxTime   CON    3000        ' Timeout Value – 3 seconds
Duty      VAR    Byte        ' Hold incoming data for duty
X         VAR    NIB         ' Counting variable

Main:
  DEBUG CLS,"Enter a PWM duty (0-255) and press return  ",CR     ' Request duty
  SERIN RPin, BMode, MaxTime, Timeout, [DEC Duty]          ' Await serial data
    FOR X = 0 to 5                                         ' PWM LED 5 times
    PWM 8,Duty, 255
    NEXT
Pause 1000
GOTO Main

Timeout:
  DEBUG "Timeout!", CR              ' Notify user of timeout
  PAUSE 500                         ' Short wait
GOTO Main
```

PWM Waveforms



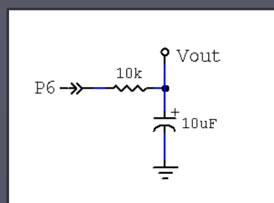
PWM duty of 25 (out of 255). Note average voltage of 0.68V.

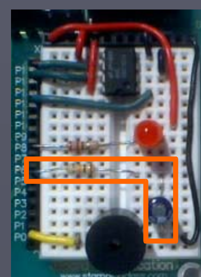


PWM duty of 225 (out of 255). Note average voltage of 4.21V

Filtering PWM

- By adding a low-pass filter to the circuit, the PWM may be converted to an analog voltage, though without buffering will be able to drive very few devices without degrading the signal.





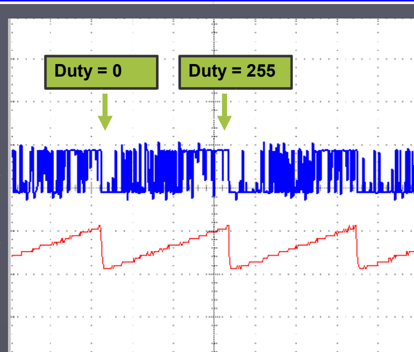
The capacitor is an Electrolytic style and is polarity sensitive. Be sure to connect (–) to Vss

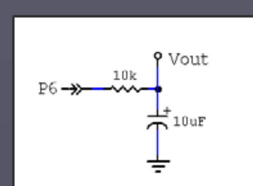
10K ohm is Brown-Black-Orange


```
'Prog 8F: Cycle through PWM for Analog output
X VAR BYTE
Main:
    FOR X = 0 TO 255
       PWM 6,X,10               '***Activity board use P12
    NEXT
GOTO Main
```

PWM at I/O Output

Filtered PWM at Vout





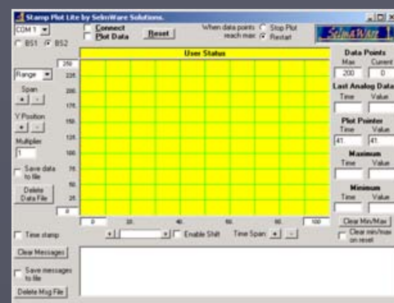
End of Section 8

Section 9: Data Acquisition

- The DEBUG window is a great simple method of monitoring data, but there are several software and hardware tools which can be of benefit also for monitor your system.
- This section will take a brief look at StampPlot Lite, StampPlot Pro/Standard, StampDAQ, and the OPTAScope.

StampPlot ™ Lite

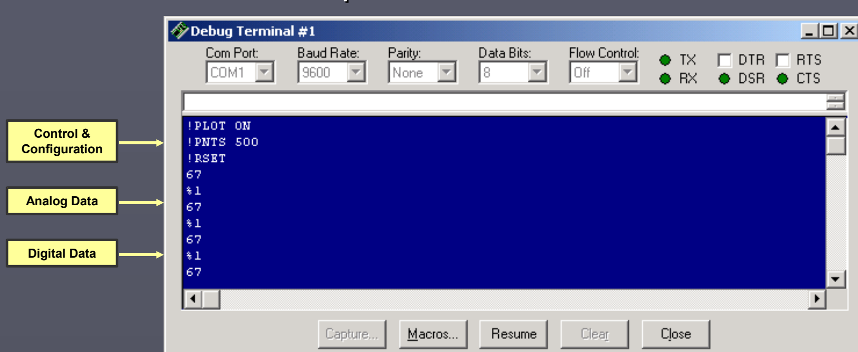
- StampPlot Lite is a digital strip chart recorded for the BASIC Stamp and is freely distributed. Link
- StampPlot Lite accepts serial data and takes the place of your DEBUG window for monitoring.

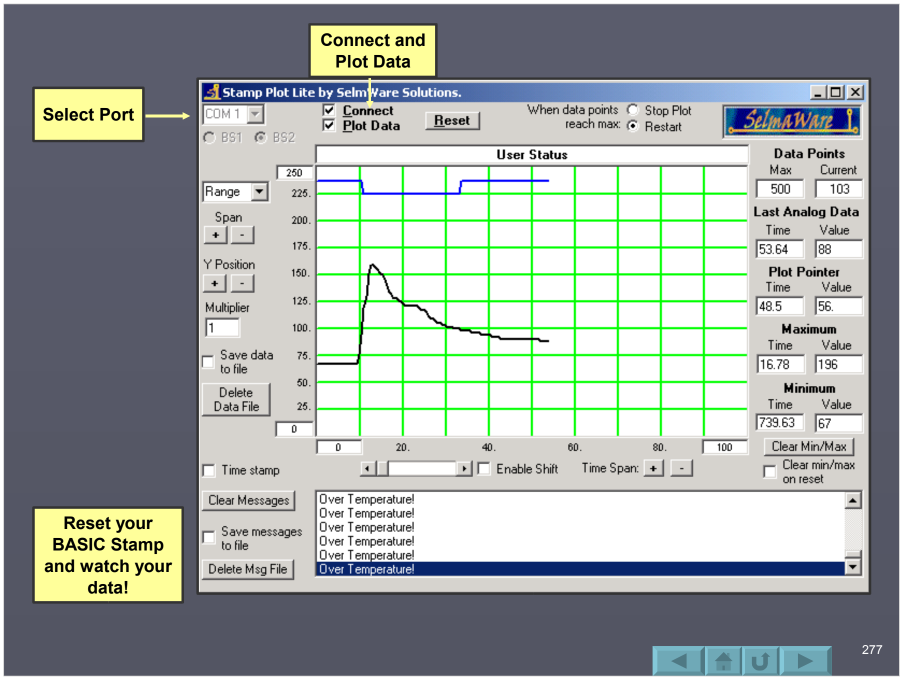


- Some basic rules for using StampPlot:
  - To plot an analog value, DEBUG a decimal value.
    DEBUG DEC X,CR
    50
  - To plot a digital value, send up to 8 bits starting with %.
    DEBUG IBIN4 INA,CR
    %1011
  - To control or configure the plot, send a plot instruction beginning with !. DEBUG "!SPAN 0,500",CR
  - Data strings not meeting the above will be sent to the message box. DEBUG "Hello World!",CR
  - All strings sent MUST end with a carriage return (CR).
- Lite can also send data collected to text files for importing into other programs.

```
'Prog. 9A: Monitoring with StampPlot LIte
Temp        VAR Word                        ' Holds converted temperature
ADres       VAR         BYTE                ' A/D result (8 bits)
ADcs        CON         12                  ' A/D enable (low true)
ADdat       CON         15                  ' A/D data line  *** Activity board use 14
ADclk       CON         14                  ' A/D clock      *** Activity board use 15
LED         CON         8
OUTPUT LED                                  ' Set LED as output

PAUSE 1000                                  'Short pause for comms
DEBUG "!PNTS 500",CR                        'Set number of data points
DEBUG "!RSET",CR                            'Reset the plot

Main:
    LOW ADcs                                ' Enable ADC
    SHIFTIN ADdat,ADclk,msbpost,[ADres\9]   ' Shift in the data
    HIGH ADcs                               ' Disable ADC
    Temp = ADres * 50/26                    ' Calculate temperature
    HIGH LED                                ' Turn off alarm LED
    IF Temp < 100 THEN NoAlarm              ' Is above alarm setpoint?
     DEBUG "Over Temperature!",CR           ' Add message
     LOW 8                                  ' Light Alarm LED
NoAlarm:
    DEBUG DEC Temp, CR                      ' Sent temperature
    DEBUG IBIN1 OUT8,CR                     ' Send alarm bit as binary
    PAUSE 500                               ' Wait a 0.5 Seconds
GOTO Main
```


- When the DEBUG Windows open, it's a good idea to make sure your data is properly formatted.
- Then close the DEBUG window so StampPlot can access that COM port.





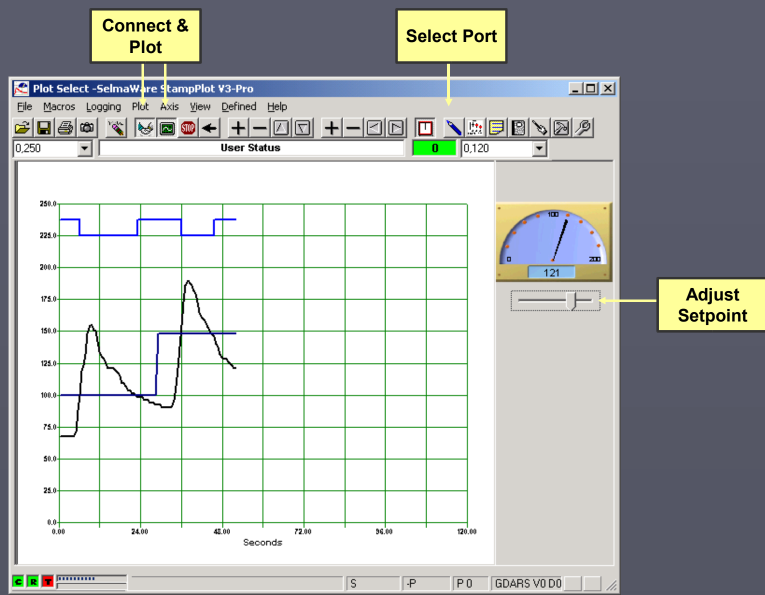
StampPlot ™ Standard/Pro

- StampPlot Standard and Pro are compatible with Lite, but adds many features including Multiple analog channels and Stamp controlled Graphical User Interface (GUI) construction. Link
- Multiple analog values can be plotted by separating with a comma.
  DEBUG DEC X, ",", DEC Y,CR
  50,100
- Plot Control Objects can be created and read with code.

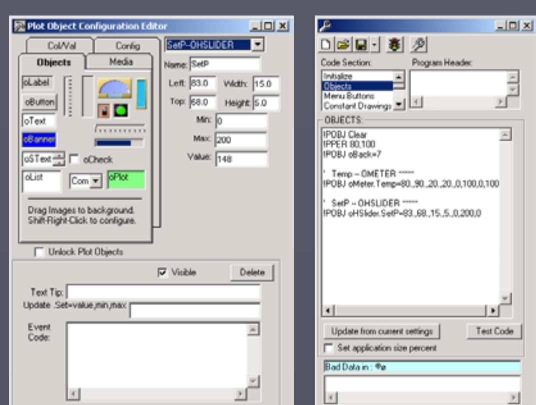
```
'Prog. 8C: Plot and control with StampPlot Standard/Pro
Temp        VAR Word                              ' Converted temperature
SetPoint    VAR Byte                              ' Setpoint value
ADres       VAR        BYTE                       ' A/D result (8 bits)
ADcs        CON        12                         ' A/D enable (low true)
ADdat       CON        15                         ' A/D data line  *** Activity board use 14
ADclk       CON        14                         ' A/D clock       *** Activity board use 15
LED         CON        8
OUTPUT LED                                        ' Set LED as output

PAUSE 1000
DEBUG CR,"!POBJ Clear",CR                         ' Clear all plot control objects
DEBUG "!PPER 80,100",CR                           ' Set plot size

                                                  'Create a meter called Temp
DEBUG "!POBJ oMeter.Temp=80.,90.,20.,20.,0,200,0,200",CR
                                                  ' Create a slider called SetP
DEBUG "!POBJ oHSlider.SetP=83.,68.,15.,5.,0,200,100",CR

DEBUG "!PNTS 500",CR                              ' Number of data points
DEBUG "!RSET",CR                                  ' Reset Plot

SetPoint = 100                                    ' Set initial setpoint
```

Continued on next slide


```
Main:
            GOSUB Read_Temp
            GOSUB Get_SetPoint
            GOSUB Update_Plot
            PAUSE 500
GOTO Main

Read_Temp:
            LOW ADcs                              ' Enable ADC
            SHIFTIN ADdat,ADclk,msbpost,[ADres\9]   ' Shift in the data
            HIGH ADcs                             ' Disable ADC
            Temp = ADres * 50/26                  ' Convert to temperature
            HIGH LED                              ' Alarm LED off
            IF Temp < SetPoint THEN NoAlarm       ' Check if below setpoint
            LOW 8                                 ' Enable alarm if not below
NoAlarm:
Return

Get_SetPoint:
   DEBUG "!READ (Setp)",CR                        ' Read setpoint from plot
   SERIN 16,84,500, Timeout,[DEC SetPoint]        ' Accept returning data
Timeout:
Return

Update_Plot:
   DEBUG DEC Temp, ",", DEC SetPoint, CR          ' Plot temp & setpoint
   DEBUG IBIN1 OUT8,CR                            ' plot alarm bit
   DEBUG "!POBJ Temp = ", DEC Temp,CR             ' update meter reading
Return
```



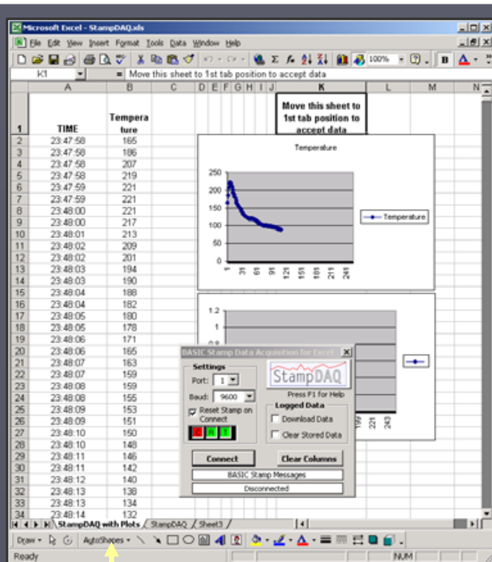
- StampPlot Standard is free for use by home & educational BASIC Stamp users.
- The Pro license adds the ability to perform drag and drop building of interfaces.



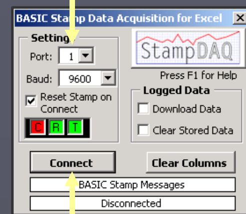
StampDAQ ™

- StampDAQ is a macro for Excel® (2000 or higher) that may be used to bring data directly into a spread sheet for analysis. Link
- Up to 10 values may be accepted.
  DEBUG "DATA,TIME,", DEC val1,",", DEC val2,CR
  DATA,TIME,50,100
  TIME is replaced by current time by StampDAQ


```
'Prog. 9C: Sending data to StampDAQ
Temp        VAR Word
SetPoint    VAR Byte
ADres       VAR     BYTE              ' A/D result (8 bits)
ADcs        CON     12                ' A/D enable (low true)
ADdat       CON     15                ' A/D data line  *** Activity board use 14
ADclk       CON     14                ' A/D clock      *** Activity board use 15
LED         CON 8

PAUSE 1000
DEBUG CR,"CLEARDATA",CR                       'Clear columns
DEBUG "LABEL,TIME,Temperature,,,,,,,",CR      'Label columns

Main:
            LOW ADcs                                  ' Enable ADC
            SHIFTIN ADdat,ADclk,msbpost,[ADres\9]     ' Shift in the data
            HIGH ADcs                                 ' Disable ADC
            Temp = ADres * 50/26                      ' Calculate Temp
            DEBUG "DATA,TIME,", DEC Temp, CR          ' Send data
            PAUSE 500
GOTO Main
```



Select COM port Settings

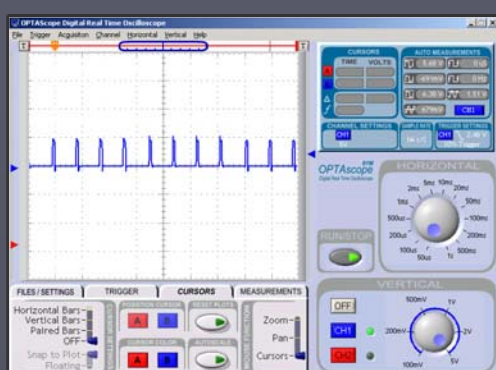


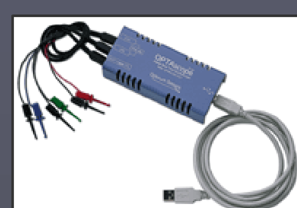
Connect and reset the BASIC Stamp

Select a Worksheet by dragging to 1st position

OPTAScope ®

- Aside from using serial data, the OPTAScope is a great little inexpensive 2–Channel USB O-Scope.
- The waveform captures in this tutorial were produced with it.





LINK

End of Section 9

Appendix A: PBASIC 2.5 Updates

- Introduction
- Version 2.5 Directive
- Compatibility with Version 2.0
- I/O Aliases Using PIN
- IF…THEN…ELSE
- SELECT…CASE
- DO…LOOP
- EXIT
- ON…GOTO
- ON…GOSUB
- DEBUGIN
- Coding on Multiple Lines

Introduction

- Using the exact same BASIC Stamp, Parallax has extended the PBASIC language to incorporate new functionality and control structures.
- The vast majority of the additions previously could have been formed using IF-THEN statements, but the additions allow for cleaner, more-structured coding.

Version 2.5 Directive

- Just as the program must contain a directive to define the version of the BASIC Stamp being programmed, a new directive is added to define the version of the tokenizer to use.

Compatibility with Version 2.0

- The new tokenizer (2.5) is fully compatible with the previous version 2.0 code with one exception:
  - Version 2.0 did NOT require a colon following a label identifier, though, by convention, one was normally used.
  - Version 2.5 DOES require a colon following a label identifier.

```
'{$STAMP BS2}
'{$PBASIC 2.5}

Main:
' Code goes here
Goto Main
```


I/O Aliases Using PIN

- In Version 2.0, both CON and VAR were used to define aliases for I/O to allow different instructions to access the I/O:

```
LED1 VAR OUT8
LED1_Pin CON8
HIGH LED1_Pin       ' HIGH 8
LED1=0              ' OUT8 = 0
```

- The PIN type definition creates an alias that may be used in either fashion:

```
LED1 PIN 8
HIGH LED1           ' HIGH 8
LED1 = 0            ' OUT8 = 0
```

- This can greatly simplify programming.

IF…THEN…ELSE

- The previous implementation of the IF statement was:
  IF *condition* THEN *addressLabel*
  If the condition is TRUE, execution would branch to the defined label.

```
'{$STAMP BS2}
'Prog App A-1: Traditional IF
' Blink LED fast if switch is pressed, Blink LED slow if not pressed
Main:
IF IN10=0 Then Fast            'Branch if pressed (TRUE)
        LOW 8                  'Blink slow if not pressed
        PAUSE 1000
        HIGH 8:  PAUSE 1000
Fast:
        LOW 8                  'Blink Fast
        PAUSE 200
        HIGH 8: PAUSE 200
Done:
        GOTO Main
```

Perform if condition FALSE

Perform if condition TRUE

- The new implementation allows the use of non-jumps as the THEN statement:
  IF *condition* THEN *statements(s)*
  IF IN10=0 THEN LOW 8: PAUSE 200: HIGH 8

- A statement block can be performed using an IF-ENDIF block.

IF IN10=0 **THEN**
    LOW 8
    PAUSE 200
    HIGH 8
ENDIF

- An ELSE may be used to define statements to be performed if the statements are FALSE:

  IF *condition* THEN

  'Condition is true statements

  ELSE

  'Condition is FALSE statements

  ENDIF


```
'{$STAMP BS2}
'{$PBASIC 2.5}
'Prog App A-2: Enhanced IF-THEN-ELSE
'Blink LED fast if switch is pressed, Blink LED slow if not pressed
Main:
IF IN10=0 THEN
          LOW 8                        'Blink fast if pressed (TRUE)
          PAUSE 200
          HIGH 8:  PAUSE 200
ELSE
          LOW 8                        'Blink slow if NOT pressed (false)
          PAUSE 1000
          HIGH 8: PAUSE 1000
ENDIF
GOTO Main
```

Perform if condition TRUE

Perform if condition FALSE

SELECT…CASE

- SELECT-CASE may be used to select an action based on the current status of a value.
- SELECT-CASE may be used to replace multiple IF-THEN statements.
- The structure is:

SELECT expression:
  CASE condition
    statement(s) to perform if condition is true
  CASE condition
    statement(s) to perform if condition is true
  CASE ELSE
    statement(s) to perform if no other conditions are met
ENDSELECT

- The condition for an expression may be:
  - A single value:
    CASE 10
  - A relational operator <,>,<=,<>,>=,=:
    CASE <=5000
  - Multiple conditions:
    CASE <10, >100
  - A range:
    40 TO 100

```
' Prog App. A-2: Using Case-Select
'{$PBASIC 2.5}
POT VAR WORD : LED1 PIN 8 : LED2 PIN 9
SPEAKER PIN 1     'Use 11 with activity board
Main:
 HIGH 7: PAUSE 1 : RCTIME 7,1,POT            ' Read potentiometer
 DEBUG ? POT                                 ' Display value
 SELECT POT                                  ' Begin SELECT based on POT value
   CASE < 1000                               ' Perform if POT < 1000
         HIGH LED2                                 '  LED2 Off
         LOW LED1                                  '  LED1
         FREQOUT Speaker,250,1000                  '  Sound low tone
   CASE 1000 to 3000                         ' Perform if POT is 1000 to 3000
         HIGH LED2                                 '  Both LEDs off
         HIGH LED1
   CASE 3000 TO 6000                         ' Perform if POT is 3000 to 6000
         LOW LED2                                  '  LED2 on
         HIGH LED1                                 '  LED1 off
         FREQOUT Speaker,250,2000                  '  Sound high tone
   CASE ELSE                                 ' Perform if no other case matches
         DEBUG "OUT OF RANGE",CR                   '  Display message
         PAUSE 250
 ENDSELECT                                   ' End of select block
GOTO Main                                    ' Repeat
```


DO…LOOP

- The DO-LOOP is a structured means of repeating a section of code, with or without conditionals.
- In it's simplest form, the DO-LOOP is used to repeat code continually, much as GOTO Main has been used.

```
DO
    statements(s)
LOOP
```

- The statements between DO and LOOP will repeat 'forever'.

- A condition may be used to determine if the statements will be performed prior to the statements (pretest).
  DO WHILE (*condition*)
      Statement(s)
  LOOP

- If the condition is determined to be false, execution will branch to after the loop.

- A condition may be used to determine if the statements will be performed again *after* the statements are performed once (posttest).
  DO
      Statement(s)
  LOOP WHILE (*condition*)

- If the condition is determined to be true *after passing through once*, execution will branch to the top of the loop.

Code is performed only if condition is true, then repeats while the condition is true.

Code is performed once, then repeats while the condition is true.

```
'{$PBASIC 2.5}
'Prog App A-4: Test of DO-LOOP
LED1 PIN 8
LED2 PIN 9
SW1 PIN 10
SW2 PIN 11

DO                          ' Start of main loop
 DO WHILE (SW1=0)           ' Start of loop, perform
                            ' if SW1 is pressed
  TOGGLE LED1               ' Blink LED1
  PAUSE 100
 LOOP                       ' End of loop

 DO                         ' Start of loop
  TOGGLE LED2               ' blink LED2
  PAUSE 100
 LOOP WHILE (SW2=0)         ' End of loop,
                            ' REPEAT if switch is pressed
 PAUSE 500                  ' Pause for 1/2 second
LOOP                        ' End of main loop
```


EXIT

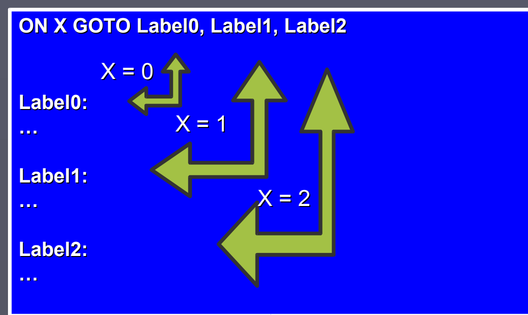
- Exit may be used to gracefully exit a FOR-LOOP or a DO-LOOP based on a condition.

```
DO
    statement1
    statement2
    IF condition THEN EXIT
LOOP
statement3
```

- If the condition in the IF-THEN is TRUE, the DO-LOOP will terminate and statement3 will be performed.

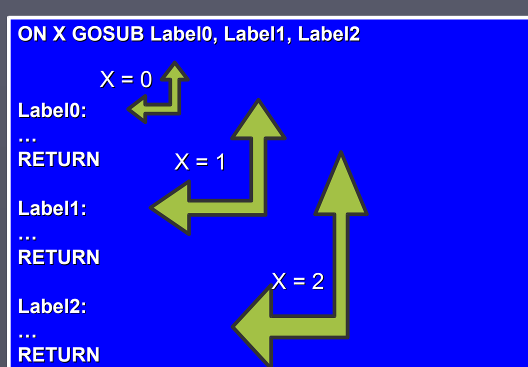
ON…GOTO

- ON-GOTO is functionally equivalent to the BRANCH statement in that an index value is used to branch to one of several labels.
  ON *offset* GOTO *label0, label1, label2…*



ON…GOSUB

- ON-GOSUB combines the clean-coding of the ON-GOTO (or BRANCH) with the power of a GOSUB.
  ON *index* GOSUB *label0, label1, label2…*

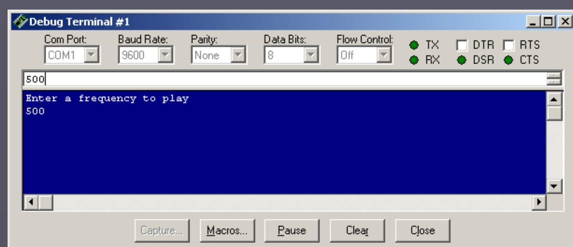


```
'{$PBASIC 2.5}
'Prog App A-5: Test of ON-GOSUB
LED1 PIN 8 : LED2 PIN 9 : SW1 PIN 10: SW2 PIN 11 : Speaker PIN 1
Press_Count VAR NIB : X VAR BYTE

DO
  IF SW1=0 THEN Press_Count = Press_Count + 1          ' If SW1 pressed, add one
  IF Press_Count = 4 THEN Press_Count = 0              ' If 4, reset to 0
  ON  Press_Count GOSUB Blink, Wink, Cycle, Quiet      'GOSUB based on Press_Count
  PAUSE 500
LOOP

Blink:                                   'Routine blinks LEDs and plays one tone
  TOGGLE LED1 : TOGGLE LED2 :  FREQOUT Speaker, 100,500 :  PAUSE 250
RETURN

Wink:                                    'Routine winks each LED and plays 2 tones
  LED1 = 0 : LED2 = 1 : FREQOUT Speaker, 500,4000 :  LED1 = 1: LED2 = 0 :  FREQOUT Speaker, 500,2000
RETURN

Cycle:                                   'Blinks LED and cycles frequency up
  FOR X = 1 to 20
    TOGGLE LED2 : TOGGLE LED2 :  FREQOUT Speaker, X * 10, 200 * X
  NEXT
RETURN

Quiet:                                   'Peaceful silence
  LED1 = 1 : LED2 = 1
RETURN
```


DEBUGIN

- Just as DEBUG implements a SEROUT to send text and data at 9600 BAUD back to the PC using the programming port, DEBUGIN accepts serial data from the PC on the programming port of the BASIC Stamp. This was previously done using the SERIN instruction.

```
'{$PBASIC 2.5}
'Prog App A-6: Test of DEBUGIN to play a tone
Speaker PIN 1
Freq VAR WORD

DO
  DEBUG CLS,"Enter a frequency to play",CR      ' Request a frequency
  DEBUGIN DEC Freq                              ' Accept returning data
  FREQOUT Speaker, 1000, Freq                   ' Sound tone
LOOP
```



Coding on Multiple Lines

- Sometimes a line of code becomes so long it is difficult to view on the screen on in a printout.
- Version 2.5 allows breaking a line after a comma (not in quotes), for instructions that may have many parameters, such as ON…GOSUB:

```
ON x GOSUB label0, label2, label3,
              label4, label5
```

Summary

- PBASIC 2.5 adds a great variety of codes to make coding easier and more readable.
- Structures such as IF…THEN…ELSE, ON…GOSUB, and SELECT…CASE allow programmers to structure the code better.
- Simple enhancements, such as PIN, remove complexity from programming.

End Appendix A

Appendix B: Number Systems

- Introduction to Number Systems
- Decimal
- Binary
- Binary to Decimal
- Bit Groupings
- Hexadecimal
- Hexadecimal to Decimal
- Hexadecimal to Binary
- Binary Coded Decimal (BCD)
- Conversion Table
- Conversion Calculators
- ASCII Codes

Introduction to Number Systems

- While we live in a world where the decimal number is predominant in our lives, computers and digital systems are much happier working in another number system – Binary.

- This section will discuss various number systems you will commonly encounter when working with microcontrollers.

Decimal

- It helps to first take a fresh look at a number system we are familiar with. Decimal.
- Decimal is a Base-10 number system (Deci meaning 10). We use Base-10 because that is the number of units on the first counting device used…. Fingers!
- In a Base-10 number system there are 10 unique symbols – 0 through 9.

- Any position in a value can only contain one of these symbols, such as 1354. There are 4 places, each with one digit.
- Each place to the left of the decimal point signifies a higher power of 10.

10^3	10^2	10^1	10^0
= 1000	= 100	= 10	= 1
______	______	______	______

- A number, such as 1354, is each place value multiplied weight of that position.

1	3	5	4
x1000	x100	x10	x 1
1000	300	50	4

Thus 1354 is 1000 + 300 + 50 + 4.

Binary

- In digital systems, there only only states such as ON/OFF, TRUE/FALSE, HIGH/LOW, or any manner of other terms used… including only the digits of 0 and 1.
- Having only 2 states or unique values creates a binary number system. In binary, we count and work with values in the same manner as decimal.

- Just as in decimal, each place is a higher power, but not of 10, but 2 since binary is a 2-based system.

2^3 = 8	2^2 = 4	2^1 = 2	2^0 = 1

Binary to Decimal

- By taking the value,1001, and multiplying each digit by the weight of the position, we can convert a binary value to decimal.

1	0	0	1
x8	x4	x2	x 1
8	0	0	1

Thus %1001 in binary is
8 + 0 + 0 + 1 = 9 Decimal

- If we see a number of 1001, is that decimal or binary, or some other number system?
- When various number systems are used some means is used to denote the system.
- Common ways to denote binary are:
  1001_2
  %1001 (Format used in programming the BASIC Stamp)
  1001b
  0b1001
- Decimal is typically not specially notated, but may be written as: 1001_{10}.

Bit Groupings

- Often Bits (Binary Digits) are grouped to form specially sized combinations.

  Nibble – 4 Bits

  Byte – 8 Bits

  Word – 16 Bits

- Word is actually used to refer to a pre-defined number of bits of any size (16-bit word, 24-Bit word, 32-Bit word, etc). In programming the BASIC Stamp, 16-bits will be a WORD.

- In a nibble with 4-bits, the range of values is %0000 (Decimal 0) to %1111 (Decimal 15: 8+4+2+1). Note there are 16 values: 0 to 15.

- In a byte with 8-bits, the range of values is %00000000 (Decimal 0) to %11111111 (Decimal 255: 128+64+32+16+8+4+2+1) Note there are 256 values: 0 to 255.

- An equation to find the maximum count for any number of bits is: 2^n-1 where n = number of bits. $2^8-1=255$.

Hexadecimal

- Digital systems work in binary because of their nature of having only 2 states, but as humans we have a difficulty dealing with numbers such as % 01111101. It is long and difficult to read.

- Hexadecimal is good middle between decimal and binary. It allows for easier use, $7C, and relates directly to binary.

- Hexadecimal is a base-16 number system. It is denoted by $7C (BASIC Stamp), 7Ch, 0x7C or $7C_{16}$.

- Each place is a higher power of 16.

2^3 = 4096 ______	16^2 = 256 ______	16^1 = 16 ______	16^0 = 1 ______

- But since it is base-16, 16 unique digits are needed . The first 10 are carried over from decimal, 0-9. The last 6 borrow from the alphabet, A-F, where:
  $A = 10 $B = 11 $C = 12
  $D = 13 $E = 14 $F = 15

Hexadecimal to Decimal

- By taking the value,$7C, and multiplying each digit by the weight of the position, we can convert a hexadecimal value to decimal.

7	C (12)
x16	x 1
112	12

Thus $7C in Hexadecimal is
112 + 12 = 124 Decimal.

Hexadecimal to Binary

- Because 16 (hex) is a whole power of 2 (binary), there is a direct correlation between a hexadecimal value and a binary value.
- Each hex value corresponds to a unique binary nibble, since a nibble has 16 unique states.

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

- Because each nibble is a hexadecimal value, it is simple to convert binary to hexadecimal and vice-versa.

0010 1110

2 E

- In programming or computer use many times values are represented in hexadecimal for a good human to computer interface number system.
  DIRS = $00F0
  The COM1 address is $03F8
  The MAC address is: $0C12CEF69B01

Binary Coded Decimal (BCD)

- BCD is used by many devices to have a direct correlation between a binary nibble and a decimal value.

- BCD is simply a subset of hexadecimal where A (%1010) through F (%1111) are invalid. It is denoted by 95_{BCD}.

%10010101

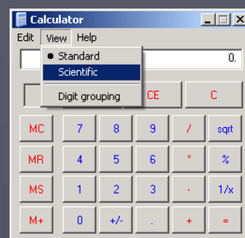
95 hexadecimal or BCD

Conversion Table

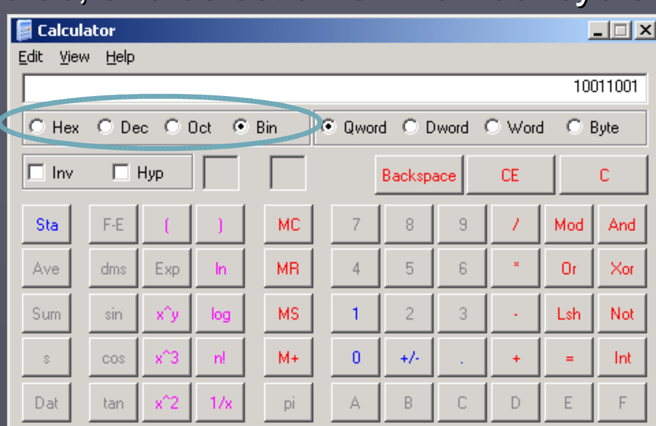
Binary	Hex	BCD	Decimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	8
1001	9	9	9
1010	A	Invalid	10
1011	B	Invalid	11
0100	C	Invalid	12
1101	D	Invalid	13
1110	E	Invalid	14
1111	F	Invalid	15
00100110	26	26	38
11111111	FF	Invalid	255

Conversion Calculators

- Many scientific calculators can convert between various number systems. The Microsoft Windows® calculator is one example. It must first be placed in scientific mode.



- Next, select the number system, enter a value, and select a new number system.



Oct is Octal, a Base-8 number system, 0 to 7, where each octal value represents 3 bits.

ASCII Codes

- A byte doesn't always represent a value. In many cases the value represents a code for a use, such as representing alpha-numeric characters. ASCII is one example.
- ASCII is a 7-bit code where each value represents a unique character or control function for the transmission of data, such as a text message to terminal. With 7-bits, there are 128 unique codes or characters that may be represented. This is a standard and strictly adhered too.
- Extended ASCII is an 8-bit code providing 256 unique characters or codes. Different systems use the upper 128 values as they desire.

ASCII Chart (first 128 characters)

Dec	Hex	Char	Name / Function	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	Null	32	20	space	64	40	@	96	60	`
1	01	SOH	Start Of Heading	33	21	!	65	41	A	97	61	a
2	02	STX	Start Of Text	34	22	"	66	42	B	98	62	b
3	03	ETX	End Of Text	35	23	#	67	43	C	99	63	c
4	04	EOT	End Of Transmit	36	24	$	68	44	D	100	64	d
5	05	ENQ	Enquiry	37	25	%	69	45	E	101	65	e
6	06	ACK	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	BEL	Bell	39	27	'	71	47	G	103	67	g
8	08	BS	Backspace	40	28	(	72	48	H	104	68	h
9	09	HT	Horizontal Tab	41	29	)	73	49	I	105	69	i
10	0A	LF	Line Feed	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	Vertical Tab	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	Form Feed	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	Carriage Return	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	Shift Out	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	Shift In	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	Data Line Escape	48	30	0	80	50	P	112	70	p
17	11	DC1	Device Control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	Device Control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	Device Control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	Device Control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	Non Acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	Synchronous Idle	54	36	6	86	56	V	118	76	v
23	17	ETB	End Transmit Block	55	37	7	87	57	W	119	77	w
24	18	CAN	Cancel	56	38	8	88	58	X	120	78	x
25	19	EM	End Of Medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	File Separator	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	Group Separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	Record Separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	Unit Separator	63	3F	?	95	5F	_	127	7F	delete

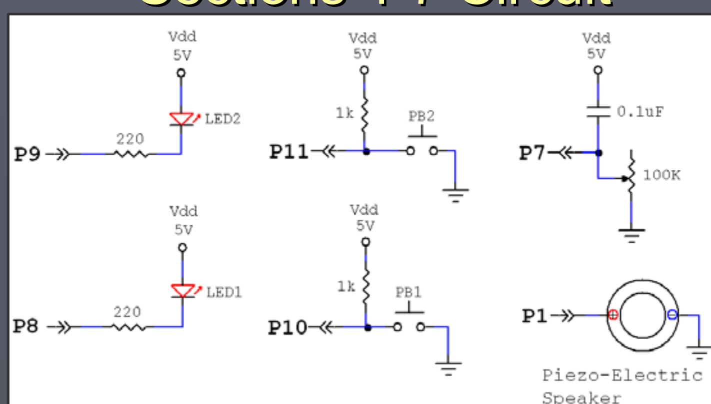
From Page 337 of the BASIC Stamp Manual Version 2

Summary

- As programmers, it is important to be able to relate controllers in other number systems, such as binary, hexadecimal and BCD.
- It is also important to understand the ASCII table and use in representing control and alphanumeric character representations.

End of Appendix B

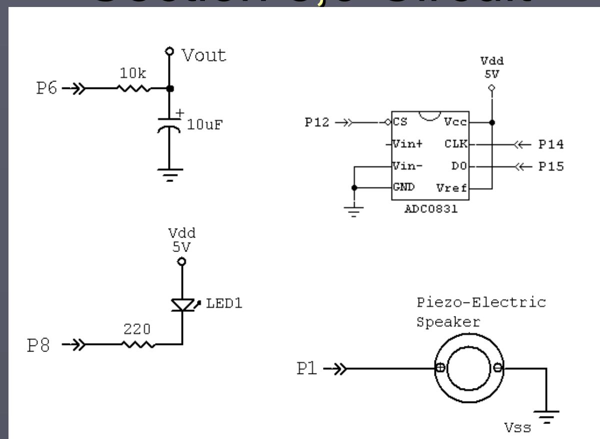
Sections 4-7 Circuit



Parts for sections 4-7:
(3) 220 Ohm Resistors
(2) Red LEDs
(2) N.O. Momentary Pushbuttons
(2) 1K Ohm Resistors
(1) 0.1 microfarad capacitor
(1) 100K Ohm Potentiometer
(1) Piezoelectric Speaker

These circuits are constructed in section 4.

Section 8,9 Circuit



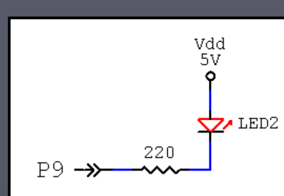
Additional Parts for Sections 8,9
(1) 10K Ohm Resistors
(1) 10uF Capacitor
(1) ADC0831
(1) LM34 Temperature Sensor

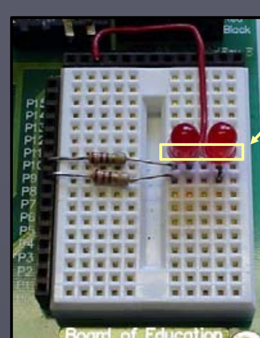
Common Circuit Declarations

```
' *********Section 5 Common Circuit Declarations ***********
' **************** I/O Aliases ******************
LED1        VAR     OUT8     'LED 1 pin I/O
LED2        VAR     OUT9     'LED 2 pin I/O
PB1         VAR     IN10     'Pushbutton 1 pin I/O
PB2         VAR     IN11     'Pushbutton 2 pin I/O
Pot         VAR     WORD     'Potentiometer value
' *************** Constants ***********************
LED1_Pin    CON     8        ' Constant to hold pin number of LED 1
LED2_Pin    CON     9        ' Constant to hold pin number of LED 2
PB1_Pin     CON     10       ' Constant to hold pin number of pushbutton 1
PB2_Pin     CON     11       ' Constant to hold pin number of pushbutton 2
Speaker     CON     1        ' Speaker Pin ***** Activity board users set to 11 ******
Pot_Pin     CON     7        ' Input for Potentiometer RCTIME network
PB_On       CON     0        ' Constant for state of pressed switch (Active-Low)
PB_Off      CON     1        ' Constant for state of un-pressed switch
LED_On      CON     0        ' Constant for state to light an LED (Active-Low)
LED_Off     CON     1        ' Constant for state to turn off an LED
' *************** Set common I/O directions *******
OUTPUT LED1_Pin              'Set pin for LED1 to be an output
OUTPUT LED2_Pin              'Set pin for LED2 to be an output
INPUT PB1_Pin                'Set pin for pushbutton 1 to be an input
INPUT PB2_Pin                'Set pin for pushbutton 2 to be an input
' ***************  Example uses **********************
'LED2 = LED_On               'OUT9 = 0
'LED1 = PB1                  'OUT8 = IN10
'HIGH LED1_Pin               'HIGH 8
```

Links

Challenge 4A-1: Solution





Anodes (to Vdd) on same row

Challenge 4A-2: Solution

```
'** 4A Challenge Solution – Blink second LED **

Main:
    HIGH 9              'Turn off LED2
    PAUSE 1000          'Wait 1 second
    LOW 9               'Turn on LED2
    PAUSE 5000          'Wait 5 seconds
GOTO Main               'Jump back to beginning
```


Challenge 4A-3: Solution

```
'** 4A Challenge Solution – Blink second LED **

OUTPUT 9                'Set P9 to output
Main:
    OUT9 = 1            'Turn off LED2
    PAUSE 1000          'Wait 1 second
    OUT9 = 0            'Turn on LED2
    PAUSE 5000          'Wait 5 seconds
GOTO Main               'Jump back to beginning
```

Challenge 4B: Solution

```
'** 4B Challenge Solution – Blink both LEDs **

Main:
        LOW 8           'LED1
        HIGH 9          'LED2
        PAUSE 2000      'Wait 2 seconds
        LOW 9           'LED2 ON (P9 LED stays on)
        PAUSE 1000      'Wait 1 second
        HIGH 8          'LED1 off
        HIGH 9          'LED1 off
        PAUSE 500       'Wait one-half second
GOTO Main
```

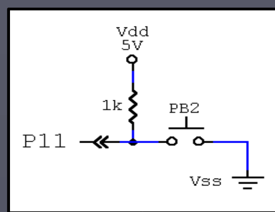
Did you waste code memory by turning on an already on P8 LED?

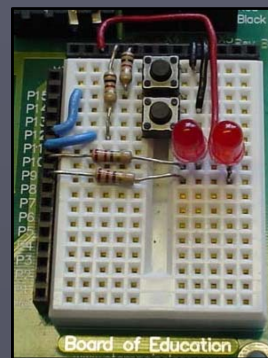
Challenge 4C: Solution

```
'{$STAMP BS2}
'** 4C Challenge Solution Example **

Main:
        LOW 8           'LED1
        HIGH 9          'LED2
        DEBUG ? OUT8
        DEBUG ? OUT9
        DEBUG "2 second pause",CR
        PAUSE 2000      'Wait 2 seconds
        LOW 9           'LED2 ON (LED P11 stays on)
        DEBUG ? OUT8
        DEBUG ? OUT9
        DEBUG "1 second pause",CR
        PAUSE 1000      'Wait 1 second
        HIGH 8          'LED1 off
        HIGH 9          'LED2 off
        DEBUG ? OUT8
        DEBUG ? OUT9
        DEBUG "0.5 second pause",CR
        PAUSE 500       'Wait one-half second
GOTO Main
```

Challenge 4D-1: Solution

Add a second pushbutton





Challenge 4D-2: Solution

```
'Challenge 4D: Reading PB2 on P11

INPUT 11                'Set P11 to be an input

Main:
   DEBUG ? IN11         'Display status of P11
   PAUSE 500            'Short pause
GOTO Main               'Jump back to beginning
```

Challenge 4D-3: Solution

```
'Challenge 4D: Reading PBs on P10 and P11

INPUT 10                'Set P10 to be an input
INPUT 11                'Set P11 to be an input

Main:
   DEBUG ? IN10         'Display status of P10
   DEBUG ? IN11         'Display status of P11
   PAUSE 500            'Short pause
GOTO Main               'Jump back to beginning
```


Challenge 4E-1: Solution

```
'Challenge: Controlling LED2 with PB2

INPUT 11                'Set P11 to be an input
OUTPUT 9                'Set P9 to be an output

Main:
   OUT9 = IN11          'Set LED2 = PB2
GOTO Main               'Jump back to beginning
```

Challenge 4E-2: Solution

```
'Challenge: Control LED1 with input PB2
'           Control LED2 with input PB1

OUTPUT 8                'Set P8 to be an output
OUTPUT 9                'Set P9 to be an output
INPUT 10                'Set P10 to be an input
INPUT 11                'Set P11 to be an input

Main:
  OUT8 = IN11           'Set LED1 = PB2
  OUT9 = IN10           'Set LED2 = PB1
GOTO Main               'Jump back to beginning
```


Challenge 4E-3: Solution

```
'Challenge: Controlling LED1 and LED2 with PB1

OUTPUT 8                'Set P8 to be an output
OUTPUT 9                'Set P9 to be an output
INPUT 10                'Set P10 to be an output

Main:
  OUT8 = IN10           'Set LED1 = PB1
  OUT9 = IN10           'Set LED2 = PB1
GOTO Main               'Jump back to beginning
```

Challenge 4F: Hint

Use the variable value of **pot** in the frequency portion of the FREQOUT instruction.

For a better tracking, reduces duration to 250 (1/4 second) and remove the PAUSE before looping back.

Debugging out values slows down loop time.

Challenge 4F: Solution

```
'{$STAMP BS2}
'Challenge 4F Solution  - your exact code may vary
'Activity Board users should have FREQOUT 11…

Pot VAR WORD                ' Variable to hold results

Main:
   HIGH 7                   ' Discharge network
   PAUSE 10                 ' Time to fully discharge
   RCTIME 7,1,Pot           ' Read charge time and store in Pot
   FREQOUT 1, 250, Pot * 10 ' Sound tone for 1/4 second at frequency of
                            ' Pot * 10 for better range
GOTO Main                   ' Jump back to beginning
```

TRY:
The FREQOUT instruction can play 2 frequencies at once for more interesting tones: FREQOUT pin, duration, freq1, freq2

Modify the code for this:
FREQOUT 1,250,Pot,1000

Challenge 5A: Solution

Example Solutions: Your names will vary but size should not.

To hold the number of seconds in a minute.
SecsInMin VAR BYTE

To hold the number of dogs in a litter.
DogsInLitter VAR NIB

To hold the count of cars in a 50 car garage.
Cars VAR BYTE

To hold the status of an output.
PinOut VAR BIT

To hold the indoor temperature.
TempInDoor VAR BYTE

To hold the temperature of a kitchen oven.
Oven_Temp VAR WORD

Challenge 5B: Solution

```
'{$STAMP BS2}
'** 5B Challenge Solution – Blink second LED with Constants **
'********** Declarations *********
'********** Constants
LED1  CON 8                 'LED 1 pin number
LED2  CON 9                 'LED 2 pin number

Main:
        LOW LED1            'LED P8 on
        HIGH LED2           'LED P9 off
        PAUSE 2000          'Wait 2 seconds
        LOW LED1            'LED P8 ON (LED P9 stays on)
        PAUSE 1000          'Wait 1 second
        HIGH LED1           'LED P8 off
        HIGH LED2           'LED P9 off
        PAUSE 500           'Wait one-half second
GOTO Main
```

Challenge 5C: Solution

```
'Challenge 5C: Control output P8 with input P11
'                    Control output P9 with input P10

' **************** I/O Variables ******************
LED1      VAR      OUT8      'LED 1 pin I/O
LED2      VAR      OUT9      'LED 2 pin I/O
PB1       VAR      IN10      'PB1 pin I/O
PB2       VAR      IN11      'PB2 pin I/O

' ************* Set I/O direction ***************
OUTPUT 8                     'Set P8 to be an output
OUTPUT 8                     'Set P9 to be an output
INPUT 10                     'Set P10 to be an input
INPUT 11                     'Set P11 to be an input

' ************ Main program ******************
Main:
   LED1 = PB2        'Set P8 output = P11 input
   LED2 = PB1        'Set P9 output = P10 input
GOTO Main            'Loop back
```


Challenge 6A: Solution

```
' ************** Program Specific Declarations **************
'Challenge 6A: Sequential Flow
'*** Insert Common Circuit Declarations ***

' ****** Main program ************
LED1 = LED_Off                    'Turn off LED 1
LED2 = LED_Off                    'Turn off LED 2
FREQOUT Speaker,2000,1000         'Sound speaker 1000Hz, 2 sec.
LED1 = LED_On                     'Light LED 1
FREQOUT Speaker,3000,2000         'Sound speaker 3000Hz, 2 sec.
LED2 = LED_On                     'Light LED 2
END
```

Note: If using the Activity Board, the LED on P11 will light when the speaker sounds

Challenge 6B: Solution

```
'Challenge 6B: Dual Alarms
'*** Insert Common Circuit Declarations ***

Main:
  HIGH Pot_Pin
  PAUSE 10
  RCTime Pot_Pin,1,Pot
  IF Pot > 2000 THEN Alarm1          ' Pot > 2000? Sound alarm1
  IF Pot < 1000 THEN Alarm2          ' Pot < 1000? Sound alarm2
GOTO Main

Alarm1:
  FREQOUT Speaker, 1000, 2000        'Sound the alarm
GOTO Main

Alarm2:
  FREQOUT Speaker, 500, 1000         'Sound the alarm
GOTO Main
```

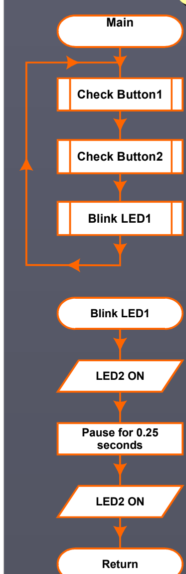

Challenge 6C: Solutions

```
' Challenge 6C: Lock In Alarm
'*** Insert Common Circuit Declarations ***

Main:
  ' If pushbutton 1 is pressed,
  ' then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  FREQOUT Speaker, 1000, 2000        'Sound the alarm
  PAUSE 500
  IF PB2 = PB_On THEN Main           'Check if PB2 is pressed to clear alarm
GOTO Alarm
```

Challenge 6D: Solution

```
'Challenge 6D: Sounding alarm to number of button presses.
'*** Insert Common Circuit Declarations ***

Counter VAR NIB                    'Variable for counting
Presses VAR NIB                    'Variable to hold number of presses

Main:
  ' If pushbutton 1 is pressed, then go sound alarm
  IF PB1 = PB_On THEN Alarm
GOTO Main

Alarm:
  Presses = Presses + 1            'Add 1 to number of presses
  FOR Counter = 1 to Presses       'Count to number of presses
    FREQOUT Speaker, 500, 2000     'Sound the alarm
    PAUSE 500
  NEXT
GOTO Main
```


Challenge 6E: Solution




```
'Challenge 6E: Addition of subroutine to blink LED1
'*** Insert Common Circuit Declarations ***

Main:
  GOSUB Check_PB1                  'Call subroutine for PB1
  GOSUB Check_PB2                  'Call subroutine for PB2
  GOSUB Blink_LED1                 'Call subroutine to blink LED1
  PAUSE 1000                 'Pause to allow blink to show
GOTO Main

' Check_PB1, Check_PB2, Sound_Speaker not shown to save space

Blink LED1:
  LED1 = LED_ON
  PAUSE 250
  LED1 = LED_OFF
RETURN
```

Challenge 6F: Solution

```
' Modifications to program 6H for solution
.
.
.
IF Press_Count < 5 THEN End_Up_Count          'If < 5, ok, otherwise reset
.
.
.
BRANCH Press_Count, [Blink, Wink, Cycle, Quiet, New1]    'Branch based on offset
.
.
.
New1:
        ' Your control code
Return
```


Challenge 7A Solution

```
'Challenge 7A: Scaling Potentiometer
'*** Insert Common Circuit Declarations ***

Degrees         VAR     WORD
Max_Degree      CON     300
Max_Value       CON     6000

Main:
  HIGH Pot_Pin                  'Read Potentiometer
  RCTime Pot_Pin,1,Pot

          'Scale Pot data by multiplying by new maximum and
          'dividing by the old maximum.  Not that both maximums were
          'divided by 100 to prevent overflowing the 65,535 limit on math
  Degrees = Pot * (Max_Degree/100) / (Max_Value/100)

          'Display in home position the value with 3 places.
  DEBUG HOME, "Angle = ", DEC3 Degrees ,CR
GOTO Main
```

Challenge 7B-1 Solution

```
'Challenge 7B-1: Boolean Evaluations
'*** Insert Common Circuit Declarations ***
Main:
  'Sound Alarm if EITHER buttons is pressed
  IF (PB1=PB_On) OR (PB2=PB_On) THEN Alarm
GOTO Main

Alarm:
  FREQOUT Speaker,100,2000
GOTO Main
```


Challenge 7B-2 Solution

```
'Challenge 7B-2: Boolean Evaluations
'*** Insert Common Circuit Declarations ***
Main:
  HIGH Pot_pin                'Read potentiometer
  PAUSE 1
  RCTIME Pot_pin,1,Pot

  'Sound Alarm if Pot > 500 or PB1 pressed
  IF (PB1=PB_On) OR (POT > 500) THEN Alarm
GOTO Main

Alarm:
  FREQOUT Speaker,100,2000
GOTO Main
```

Challenge 7B-3 Solution

```
'Challenge 7B-3: Boolean Evaluations
'*** Insert Common Circuit Declarations ***
Main:
  HIGH Pot_pin                 'Read potentiometer
  PAUSE 1
  RCTIME Pot_pin,1,Pot

  'Sound Alarm if Pot > 500 but PB2 is not pressed
  IF (PB2=PB_Off) AND (POT > 500) THEN Alarm
GOTO Main

Alarm:
  FREQOUT Speaker,100,2000
GOTO Main
```


Challenge 7D Solution

```
'Challenge 7D - Playing Charge with LOOKUP Tables.
Speaker  CON 1

I        VAR      NIB              'Table Index
F        VAR      WORD             'Frequency
D        VAR      WORD             'Duration

FOR I = 0 to 7
         ' Read duration from the table
         LOOKUP I,[150,150,150,300,9,200,600],D
         ' Read frequency from the table
         LOOKUP I,[1120,1476,1856,2204,255,1856,2204],F
         ' Play the note
         FREQOUT Speaker,D,F
Next
```

Challenge 7E-1 Solution

- How does the program identify the end of the name or number?
A byte of 0 is checked in the loops, and the listings, to identify the end.

- How does the program move to the next name in the listing?
Names are stored at $20 (20 Hex) increments ($00, $20,$40 ..). The NextName routine adds $20 to the Name variable and checks to see if the 1st character is a byte of 0, if it is, it goes back to $00 and starts over.

- How does the program access the correct number to dial?
The numbers are $10 above the name. By adding $10 to the current Name variable, the beginning of the number is addressed.

Challenge 7E-2 Solution

```
DATA   @$00,"BILL"
DATA   @$10,"555-1234"
DATA   @$20,"PARALLAX"
DATA   @$30,"1-888-512-1024"
DATA   @$40,"JIM"
DATA   @$50,"555-4567"
DATA   @$60,"MARTIN"
DATA   @$70,"555-9876"
DATA   @$80,00
```

Challenge 8A-1 Solution

```
'Challenge 8A: Alarming Temperature detector

ADres     VAR     BYTE          ' A/D result (8 bits)
ADcs      CON     12            ' A/D enable (low true)
ADdat     CON     15            ' A/D data line  *** Activity board use 14
ADclk     CON     14            ' A/D clock      *** Activity board use 15
Speaker   CON     1             ' Activity Board use 11
Temp      VAR     WORD          ' Data converted to temperature

Main:
          PAUSE 500                                   'Short Pause
          LOW ADcs                                    ' Enable ADC
          SHIFTIN ADdat,ADclk,msbpost,[ADres\9]       ' Shift in the data
          HIGH ADcs                                   ' Disable ADC
          Temp = ADres * 50/26                        ' Convert to temperature
          DEBUG CLS, "Temperature  = ", DEC Temp      ' Display temperature
          IF Temp <= 100 THEN Main                    ' If <= 100, back to main
          FREQOUT Speaker, 1000,2000                  ' If > 100, alarm
GOTO Main
```


Challenge 8B-1 Solution

```
'Solution 8B-1: Use SEROUT to send data to PC
Rpin      CON     16            ' From programming port
TPin      CON     16            ' To programming port

BMode     CON     84            ' BAUD mode -- Use 240 for BS2SX, BS2P
MaxTime   CON     3000          ' Timeout Value – 3 seconds
Freq      VAR     WORD          ' Hold incoming data

Main:
  SEROUT TPIN, BMode, [ CLS,"Enter a frequency and press return  ",CR]
  SERIN RPin, BMode, MaxTime, Timeout, [DEC freq]
  SEROUT TPIN, BMode, [ "Playing tone at ", DEC Freq, "Hz.",CR]
  FREQOUT 1,1000,Freq
  GOTO Main
Timeout:
  SEROUT TPIN, BMode, [ "Timeout!", CR]
  PAUSE 500
GOTO Main
```

Challenge 8B-2 Solution

```
'Solution 8B-2: Communicate at 1200 8-N-1
Rpin      CON     16          ' From programming port
TPin      CON     16          ' To programming port

BMode     CON     813         ' BAUD mode – Use 2063  for BS2SX, BS2P
MaxTime   CON     3000        ' Timeout Value – 3 seconds
Freq      VAR     WORD        ' Hold incoming data

Main:
  SEROUT TPin, BMode, [ CLS,"Enter a frequency and press return  ",CR]
  SERIN RPin, BMode, MaxTime, Timeout, [DEC freq]
  SEROUT TPin, BMode, [ "Playing tone at ", DEC Freq, "Hz.",CR]
  FREQOUT 1,1000,Freq
  GOTO Main
Timeout:
  SEROUT TPIN, BMode, [ "Timeout!", CR]
  PAUSE 500
GOTO Main
```

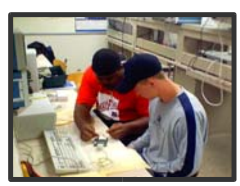
As you test this, observe the speed at which the text appears in the DEBUG Window.

Links

PARALLAX

Go

Southern Illinois University Carbondale
Electronic Systems Technologies



Go

Yahoo!® Discussion Groups

BASIC Stamps Group
Great place to post questions & get answers from thousands of BASIC Stamp users including Parallax staff.

Go

Parallax Educator's Group
A dedicated group for educators using Parallax products in the classroom. Approval required to join – see home page.

Go

Educational Curriculum

- What is a Microcontroller
- Basic Analog and Digital
- Robotics!
- Earth Measurements
- Industrial Control
- Advanced Robotics

Go

Software and other Downloads Go

More Links from Parallax Go