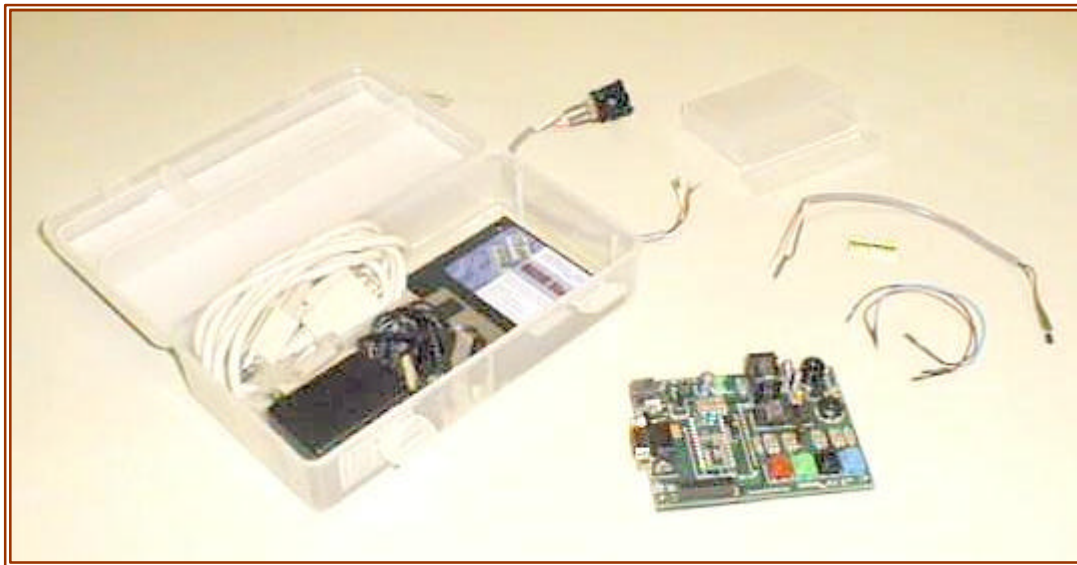


# **Application of Microcontrollers**

## **Manual**

### **Part I - Principles & The BASIC Stamp ®**

**Version 0.9p**



**Electronics Management**  
Department of Information Management Systems  
Office of Off-Campus Academic Programs  
College of Applied Sciences and Arts  
**Southern Illinois University, Carbondale**

## Forward

The Application of Microcontrollers Manual, Labs and supporting BS2 programs were developed for the Electronics Management (ELM) Off-Campus program at Southern Illinois University, Carbondale (SIUC). SIUC provides ELM courses allowing military personnel to work toward their degree at 11 bases nationwide currently. The material will also be used to supplement the on-campus ELM program in Carbondale, Illinois.

Ken Gracey at Parallax saw our early drafts and asked to post a copy on the Parallax Stamps-In-Class web site. So here it is! What does this mean to you, the user of this manual? It helps to understand the intentions of this material in using it.

At the military bases, the technical semester of 18 weeks is spanned by 3 courses in basic electronics lasting 6 weeks each. The classes meet every-other weekend. The courses are Analog Electronics, Digital Electronics and Microcontrollers. While the students are enrolled in these 3 courses, they are concurrently performing an independent study lab: Applications of Microcontrollers. This material is the basis of their course. The students receive a copy of this material, check out an ELM Stamp Kit (shown on cover) and work at home reading, testing, completing the labs and hopefully learning!

This material was written to not only teach microcontrollers, but also to reinforce and show applicability of their lecture course work in using microcontrollers. During the Analog and Digital courses the students complete Part I of this material using the BS2 and Activity board. During the last course in microcontrollers the students work on Part II of this manual (not included) dealing with the Intel 8051 using UMPS® simulation software from Virtual Micro Design.

**The Stamp Kit prepared for the students includes (and this material assumes the user has) the following:**

- **BASIC Stamp Activity Board with a BS2, cables and power supply.**
- **Three 4" Jumpers**
- **ADC 0831 added to the Activity Board (available from Parallax).**
- **A leaded 10K potentiometer for the ADC input.**
- **A leaded LM-34 temperature sensor that provides 0.01V/°F for the ADC input (available from most electronics distributors).**
- **The accompanying labs and BS2 files.**

We feel we have provided students working independently with the best possible means to learn the about basic electronic principles for microcontrollers using the BS2 and following on with the 8051 for low level programming. We also feel the material is a great addition to a traditional electronics program and we are pleased to make it available to you. We hope you enjoy the material and can benefit from our work. We encourage feedback, so please feel free to drop us Email letting us know if it has been of use to you.

Martin Heibel & William Devgenport

# Application of Microcontrollers

## Copyright Notices

Copyright 1999, Board of Trustees, Southern Illinois University. The manual and labs may be copied and distributed freely in its entirety in electronic format by individuals for educational non-profit use. Distribution of printed material is authorized for educational non-profit use. Other distribution venues, including mass electronic distribution via the Internet, require the written approval of the SIU Board of Trustees.

BASIC Stamp® is a registered trademark of Parallax, Inc. Images and drawings are reproduced by permission of Parallax, Inc.

UMPS® is a registered trademark of Virtual Micro Design. UMPS images are reproduced by permission of Virtual Micro Design.

## Disclaimer

Southern Illinois University, the manual developers, and approved distributors will not be held liable for any damages or losses incurred through the use of the manual, labs and associated materials developed at Southern Illinois University.

## Contact Information

### E-mail:

Primary developer of the manual and labs:

Martin Hebel ..... mhebel@siu.edu

Contributing developer and editor:

Will Devenport..... willd@siu.edu

Director, Off-Campus Academic Programs:

Dr. Terry Bowman..... tbowman@siu.edu

Chair, Department of Information Management Systems:

Dr. Jan Schoen Henry ..... jshenry@siu.edu

### Mailing:

Electronics Management

MC: 6614

College of Applied Sciences and Arts

Southern Illinois University, Carbondale

Carbondale, IL 62901-6614

*A product such as this is not done alone. The following people are thanked for their contributions: Ken Gracey and the gang at Parallax for their work on making this possible for our students; Philippe Techer at Virtual Micro Design for designing a great simulation package and working with us; Myke Predko for his feedback and recommendation; I. Scott MacKenzie for a concise text on the 8051; our student evaluators John Almy and Kirk Larsen for catching many mistakes and for contextual recommendations; and finally Terry Bowman and Jan Henry for budgeting the endeavor and wanting the best education for our students.*

**Key Web Sites:**

Electronics Management Home Page: ..... [www.siu.edu/~imsasa/elm](http://www.siu.edu/~imsasa/elm)

Off-Campus Programs Home Page: ..... <http://131.230.64.6/>

Parallax Incorporated Home Page: ..... [www.parallaxinc.com](http://www.parallaxinc.com)  
..... [www.stampsinclass.com](http://www.stampsinclass.com)

Virtual Micro Design Home Page (UMPS): ..... [www.vmdesign.com](http://www.vmdesign.com)

**Distributors & Additional Information:**

Digi-Key Electronics - Stamps, components ..... [www.digikey.com](http://www.digikey.com)

Jameco Electronics - Stamps, components ..... [www.jameco.com](http://www.jameco.com)

JDR Electronics - Stamps, components ..... [www.jdr.com](http://www.jdr.com)

Wirz Electronics - UMPS U.S. Sales..... [www.wirz.com](http://www.wirz.com)

Peter H. Anderson - General microcontroller information ... [www.phanderson.com](http://www.phanderson.com)

SelmaWare Solutions - Specialized interfacing software ..... [www.selmaware.com](http://www.selmaware.com)

**Texts:**

The 8051 Microcontroller, 3<sup>rd</sup> ed. 1999, Scott MacKenzie. Prentice-Hall  
ISBN: 0-13-780008-8

Handbook of Microcontrollers. 1999, Myke Predko. McGraw-Hill  
ISBN: 0-07-913716-4

Programming and Customizing the 8051 Microcontroller. 1999, Myke Predko. McGraw-Hill.  
ISBN: 0-07-134192-7

The Microcontroller Idea Book. 1994, Jan Axelson. Lakeview Research.  
ISBN: 096508190-7

## Table of Contents

<b>SECTION A: INTRODUCTION TO MICROCONTROLLERS &amp; FLOWCHARTING.....</b>	<b>A-1</b>
MICROPROCESSORS .....	A-1
THE MICROCONTROLLER.....	A-2
PROGRAMMING .....	A-2
THE BASIC STAMP II (BS2).....	A-4
FLOWCHARTING .....	A-5
<b>SECTION B: THE STAMP ACTIVITY BOARD &amp; BASIC I/O.....</b>	<b>B-1</b>
OVERVIEW .....	B-1
SIMPLE I/O COMMUNICATION.....	B-3
BOARD LEDs AND BUTTONS .....	B-5
ACTIVITY BOARD SPEAKER .....	B-6
DEBOUNCING BUTTONS.....	B-9
JUST FOR FUN!.....	B-12
<b>SECTION C: BINARY NUMBERS .....</b>	<b>C-1</b>
DECIMAL NUMBER SYSTEM .....	C-1
DIGITAL AND BINARY.....	C-1
NIBBLES, BYTES AND WORDS.....	C-2
USING DEBUG TO DISPLAY BINARY DATA.....	C-3
PBASIC2 I/O USING NIBBLES, BYTES AND WORDS.....	C-5
MSB & LSB.....	C-6
<b>SECTION D: ANALOG INPUTS AND OUTPUTS.....</b>	<b>D-1</b>
PULSE WIDTH MODULATED OUTPUT .....	D-2
ACTIVITY BOARD AOUT .....	D-4
ANALOG TO DIGITAL CONVERTERS (ADC) .....	D-6
RESISTIVE DEVICES AND RCTIME .....	D-10
SCALING INPUTS .....	D-11
<b>SECTION E: PROCESS CONTROL.....</b>	<b>E-1</b>
OVERVIEW .....	E-1
ON-OFF CONTROL .....	E-2
DIFFERENTIAL-GAP CONTROL .....	E-3
PROPORTIONAL CONTROL MODE .....	E-5
DERIVATIVE & INTEGRAL CONTROL .....	E-8
<b>SECTION F: HEXADECIMAL &amp; BS2 MEMORY .....</b>	<b>F-1</b>
HEXADECIMAL.....	F-1
BS2 MEMORY & VARIABLE STORAGE .....	F-3
BS2 MEMORY MAP AND EEPROM STORAGE .....	F-6
<b>SECTION G: LOGICAL OPERATORS AND SIGNED NUMBERS .....</b>	<b>G-1</b>
INTRODUCTION TO LOGIC .....	G-1
PBASIC2 AND LOGICAL OPERATORS .....	G-3
MASKING.....	G-5
BINARY ADDITION & SUBTRACTION .....	G-7
<b>SECTION H: DIGITAL COMMUNICATIONS.....</b>	<b>H-1</b>
INTRODUCTION .....	H-1
PARALLEL COMMUNICATIONS .....	H-1
SYNCHRONOUS SERIAL COMMUNICATIONS .....	H-4
ASYNCHRONOUS SERIAL COMMUNICATIONS .....	H-9

## Section A: Introduction to Microcontrollers & Flowcharting

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

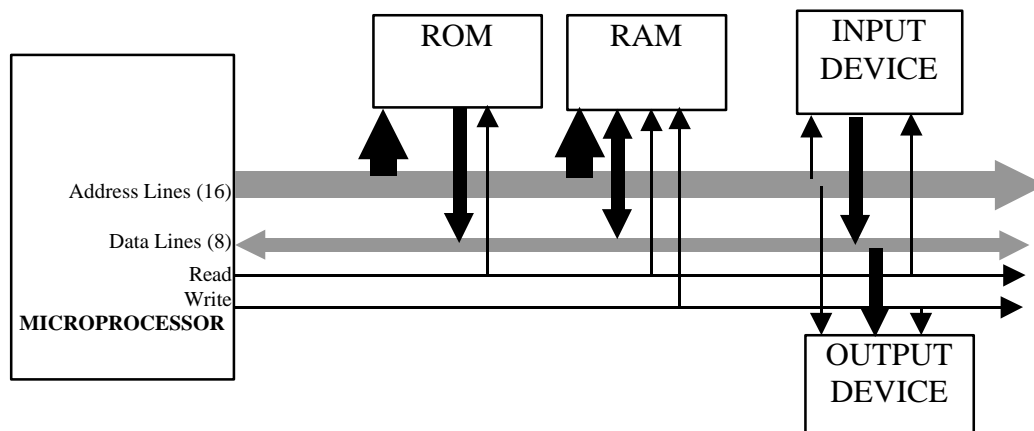
- 1) Discuss the primary differences between microcontrollers and microprocessors
- 2) Discuss the difference between processor machine languages and high level languages.
- 3) Discuss the differences between interpreted and compiled languages.
- 4) List the ROM, RAM and I/O resources available on the BASIC Stamp II.
- 5) Develop flowcharts for operations.

### Microprocessors

A microprocessor is an integrated circuit (IC) which through address lines, data lines, and control lines has the ability to:

- a) Read and execute program instructions from external Read Only Memory (ROM).
- b) Hold temporary data and programs in external Random Access Memory (RAM) by writing or reading.
- c) Perform input-from and output-to devices using these same lines.

Figure A-1 depicts simplified connections of a microprocessor to devices.



**Figure A-1: Simple Microprocessor System**

If you are familiar with the inside of typical personal computers (PC), you may be familiar with some of these components. The microprocessor may be an Intel Pentium, for RAM it may be the 32 meg in your SIMMS. Typical input and output devices, such as the mouse, keyboard, monitor and printer communicate through I/O controllers, which in turn are controlled by the microprocessor. A PC is a very complex machine because of the high level tasks that it performs and the sophisticated devices that it communicates with.

Many times electronic needs require the ability to read data from devices, and based upon the application, control output devices. Take for example a microwave oven. It requires input data such as cooking time and power setting to heat your food. It collects this data through a touch

pad. Output is the control of the microwave heating device and a digital display to provide you with information.

It is possible to use a microprocessor for this application. Electronically we would need a microprocessor bussed to ROM IC's, RAM IC's, display drivers, keypad drives, and various support components. It would take a minimum of 5 IC's to simply fulfill the requirement to read a single input, and based on the application requirements, use this data to turn on a single output.

### ***The Microcontroller***

The microcontroller is a specialized microprocessor that contains much of the circuitry and devices needed internally to collect data from inputs. It holds permanent programs in a type of ROM. It has temporary storage space for data in RAM and can control simple devices through outputs.

In our example of the microwave oven, a microcontroller has all of the essential blocks to read from a keypad, write information to the display, control the heating element and store data such as cooking time. In addition to simple ON/OFF inputs and outputs, many microcontrollers have abilities such as counting input pulses, measuring analog signals, performing pulse-width modulated output, and many more.

The microcontroller used in the BASIC Stamp II (BS2) from Parallax, Incorporated is based on the PIC16C57 from Microchip Technologies. This microcontroller has contained in it 2000 bytes of ROM memory, 72 bytes of RAM memory, and 20 I/O pins to gather data or control devices. While the ROM area to hold a program is not large, nor the RAM area to hold variables (a typical PC computer has 32 MILLION bytes of RAM for programs and variables), it may be very sufficient for control of simple systems. Of course, additional ROM, RAM, and specialized devices can be added to supplement these built in capabilities.

### ***Programming***

Microprocessors and microcontrollers work off of very specialized instructions designed for them. Each one has unique instructions to perform tasks such as reading from memory, adding numbers together and manipulating data. For example, the Intel Pentium found in IBM compatibles uses a completely different instruction set from the Motorola PowerPC used in Macintosh computers. Programs for these processors work in what is known as machine language. A task as simple as multiplying two numbers together may take hundreds of machine instructions to accomplish. These programs can be very cryptic to programmers not familiar with that processor's unique instruction set. For example code to add 3 plus 2 and store the results may look like:

```
LDA #$03
STA $1B3C
LDA #$02
ADD $1B3C
STA $1B3D
```

High level languages such as BASIC and C use instructions that are more understandable to users since they use pseudo-English to program in. In addition, a version of BASIC designed for IBM PC's may be much like the BASIC designed for Macintoshes. A line of code to add 3 plus 2 and store the result may look like:

$$\text{Sum} = 3 + 2$$

It is the job of the high level language's interpreter or compiler to take this BASIC code and make it understandable to the unique processor on which it is running. An *interpreter* decodes to machine language at run time, a *compiler* decodes the program into machine language before running it.

Let's take an example of making coffee. Simplified steps for this involve:

- Fill the maker with water.
- Insert a filter into the tray.
- Add coffee grounds into the filter.
- Put the filter tray into the maker.
- Turn on the coffee maker.
- Wait with blurry eyes.

The English language has a command to perform this task: Make Coffee. We know from experience what this means and the actions required. If this were a command in a programming language, here is how a compiled and interpreted language would utilize the command (hypothetically).

A compiled language, such as C, would read our command to "Make Coffee", and it would compile it into its individual steps and store it as Machine Code instructions. The steps of "Fill the maker with water", "Insert...", and so on would be stored in-place of "Make Coffee". When the program is executed, the processor would read "Fill the maker with water", inherently understand the instruction, and perform it. Then read "Insert a filter into tray" and perform it, and so on, until the coffee is made.

An interpreted language, such as Basic, would store a symbol or a *token* representing "Make Coffee" (in our minds it may be a dripping coffee maker, to a computer it maybe a unique number, such as "10010011"). When this program is executed, the Basic Interpreter reads the symbol that was stored. Then it must break down the symbol and inform the processor what to do. "OK, let's see, that symbol means to Make Coffee! Hey processor, fill the maker with water. Ok, what next..., processor you need to insert a filter into the tray, ok, next...."

A little simplistic? Perhaps, but that is essentially what the interpreter needs to do. In a compiled language, a simple statement is compiled, or broken down, into the many instructions understood by a processor to perform the task prior to executing. In an interpreted language, the commands are made into symbols, or tokens, which are stored and decoded (interpreted) while the program is running.

An interpreted language program can be stored in much less space, but it executes much slower and requires the presence of an interpreter to decode it.



***The BASIC Stamp II (BS2)***

The microcontroller used in the first half of this manual is the BASIC Stamp II from Parallax, Incorporated. This processor has gained world wide popularity because of its powerful but simple programming language, PBASIC2 (PBASIC is used in the original Stamp), which is based on the very popular BASIC language for many different computers. The BS2 has at its heart a PIC16C57 compatible microcontroller. Parallax added hardware and a BASIC interpreter to the PIC in developing the BS2.

Programs are edited on a host computer, using such common commands as IF-THEN, GOTO, GOSUB and so on. The code is then tokenized and transferred to the BS2 where it resides in EEPROM.

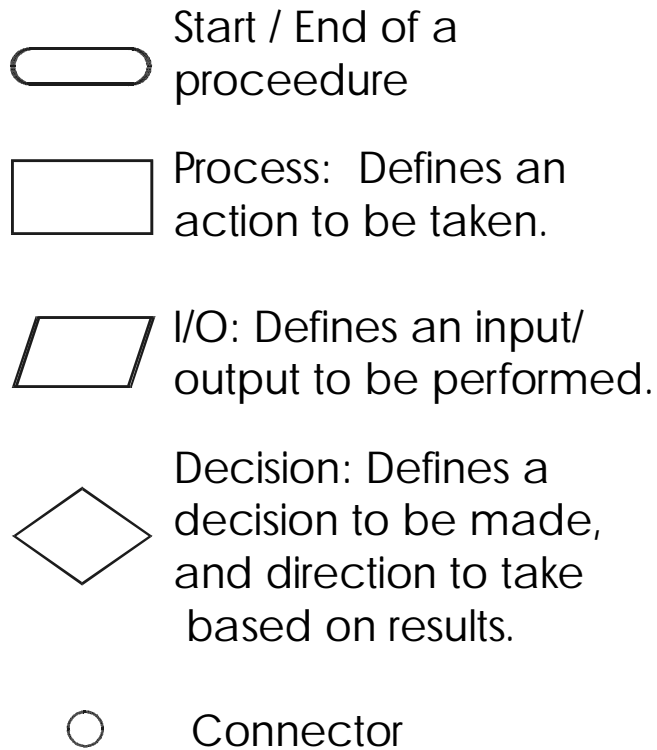
Figure H-1 of Reference A, page 207, is the schematic diagram of the BS2. The nerve center of the device is the PIC16C57 (or compatible) microcontroller. The on-chip 2K of ROM stores for the BS2's PBASIC2 Interpreter. The actual tokenized programs written by the programmer are stored in an external 2K EEPROM (Electrically Erasable Programmable Read Only Memory). Of the 20 I/O lines available on the PIC16C57, 2 are used to read and write EEPROM memory and 2 others are used for serial communications to the host PC for programming. This leaves 16 I/O lines, or pins, for the BS2 user.

Of the 72 bytes of RAM available on the PIC16C57, 32 bytes of RAM are available to the programmer with 6 of these being used as 'registers' to control the input/output pins. It is important to understand that once programmed the BS2 is completely independent of the host PC. The BS2 may be disconnected from the PC and will continue to perform its program. It may also have power removed, re-applied, and will proceed to run the stored program.

As you work through this manual you will be introduced to various PBASIC2 language commands in controlling the BS2 while applying fundamentals of electronics to microcontrollers. This manual is NOT a definitive guide on PBASIC2. Please consult Reference A for clarification on the BASIC Stamp II and PBASIC2.

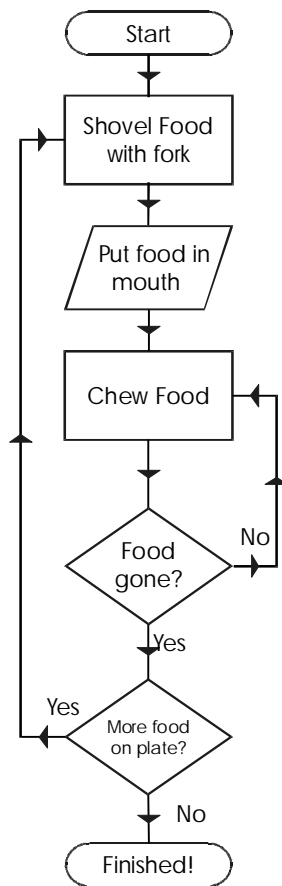
**Flowcharting**

Flowcharting is a method of symbolically representing an operation and is typically used in programming. Flowcharting is not only limited to programming. It may be applied to anything that involves operations. Let's first look at some common flowcharting symbols.



**Figure A-1: Flowcharting Symbols**

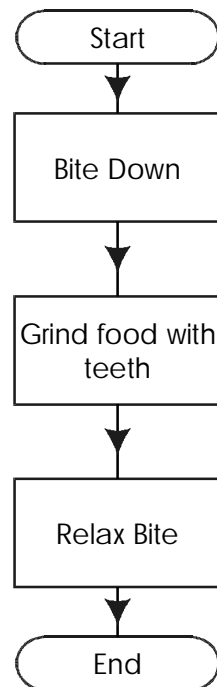
Now that we know some symbols, let's apply it to an everyday process: Eating!



**Figure A-2: Eating Process**

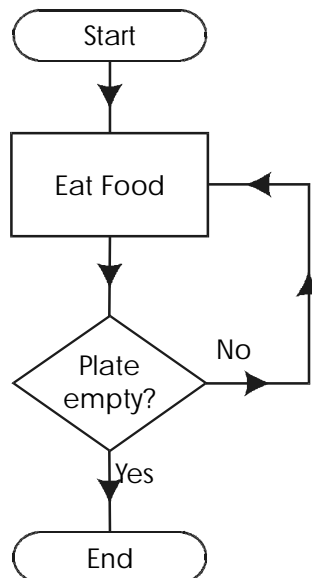
The operation that is being described by the flowchart is eating. Can you see how the symbols are used to represent the steps involved? Getting food on your fork and chewing the food are processes that have to be performed. Decisions have to be made about the process. Is the food in our mouths all chewed? Is there any food left on the plate? Depending on the answers to these questions, the process will branch in different directions whether 'Yes' it is true, or 'No' it is false. After chewing, check to see if the food in your mouth is gone. If it isn't, go back and chew more. Keep repeating until the food is finally gone, then move on to check if there is more food on the plate. If there is, go back and take another scoop. If not, we are finished eating everything on our plate and get some desert!

Sometimes the most difficult task in flowcharting is to decide on what level to chart. We have an operation called "Chew Food". This can be broken down further into other operations such as the following:



**Figure A-1: Chewing Process**

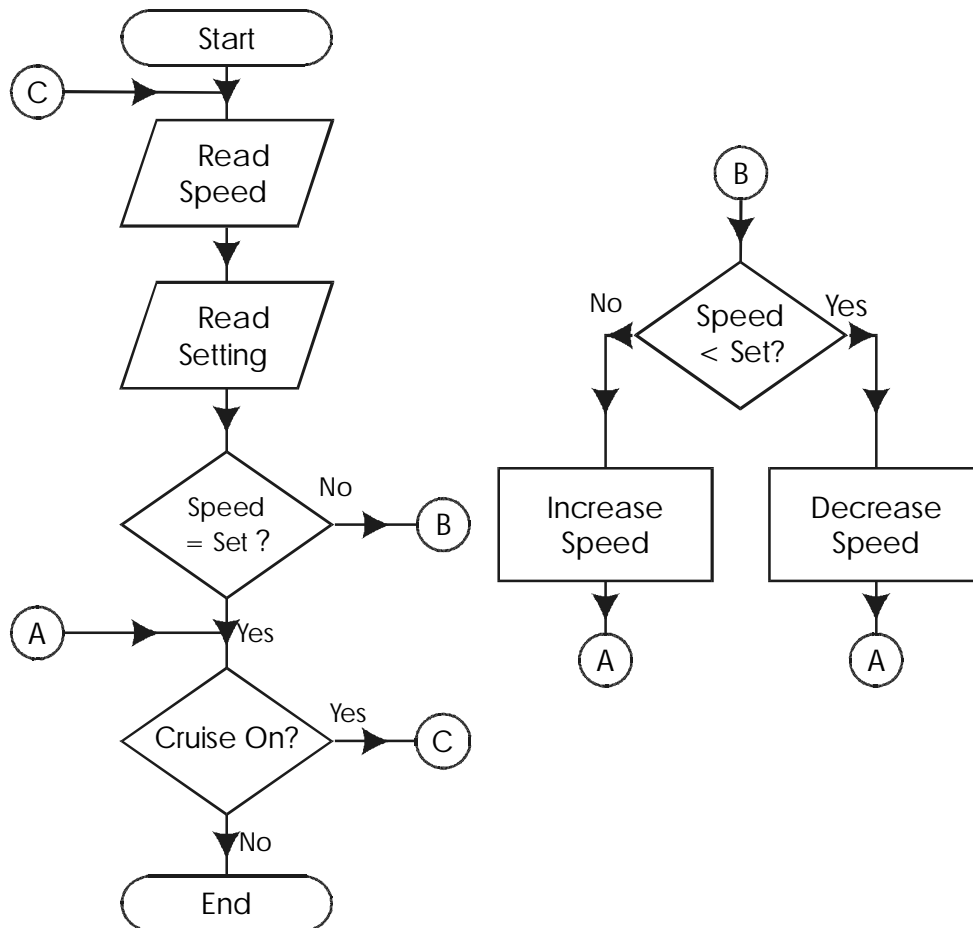
Or the eating operation may be described in a more general manner:



**Figure A-3: Simple Eating Process**

A complex process may be flowcharted at several levels. The first giving the general operation level then going deeper into specifics with other flowcharts. In general, you want the flowchart to sufficiently describe the process involved without giving step by step instructions or program code.

Flowcharts typically are read from top to bottom, with branches to the right. Circles with lettered annotations are used to denote connection points between routines. They can be used if the page is not long enough for an entire process, if you just wanted the chart neater, or if you need to branch to a sub-process.



**Figure A-2: Cruise Control Process**

Throughout the manual, selected programs will have flowcharts to describe their operation.

## **Section B: The Stamp Activity Board & BASIC I/O**

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

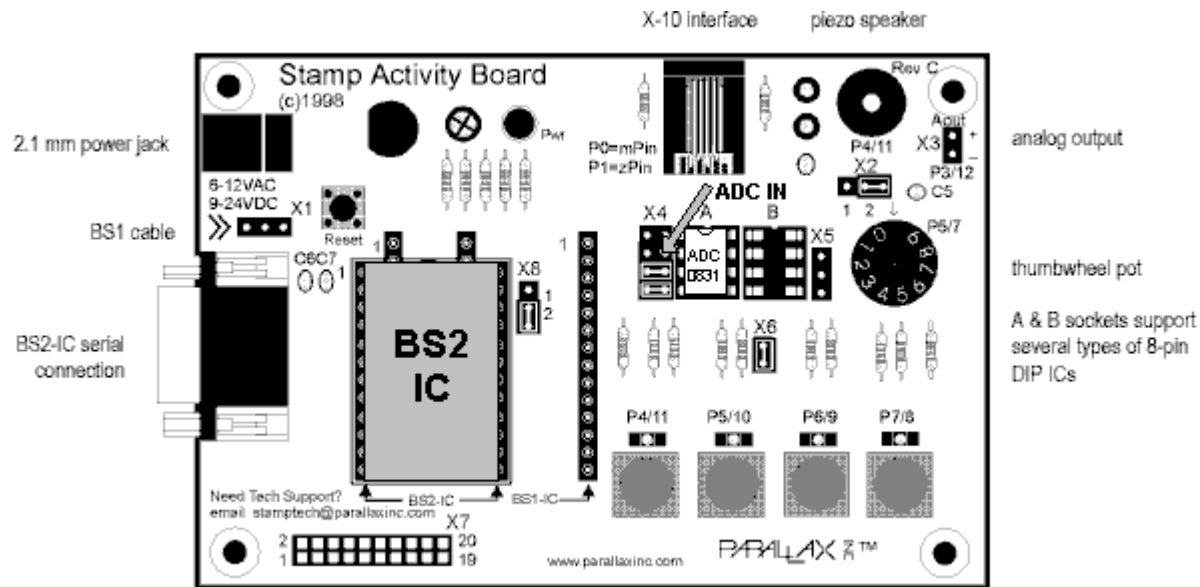
### Objectives:

- 1) List the I/O devices available on the Activity Board.
- 2) Identify the BS2 pin numbers associated with each I/O.
- 3) Discuss terminology and respective voltages associated with digital I/O.
- 4) Write PBASIC2 code to read and write to simple I/O.
- 5) Use PBASIC2 commands to control the Activity Board speaker.
- 6) Discuss the need for debouncing input devices.
- 7) Write PBASIC2 code for debouncing buttons.

### **Overview**

The Stamp Activity Board from Parallax accepts both the BS1 and BS2. The board provides the means to program Stamps and contains simple input/output devices. Once the Stamp is programmed through the serial port, it maybe disconnected from the host PC and continue to perform the program. The board contains the following:

- Host computer connectors (The DB-9 serial port will be used with the BS2).
- A Reset button to 'restart' the BS2 program.
- 4 push-button switches.
- 4 LED lamps.
- 1 potentiometer
- 1 piezo-electric speaker.
- 1 analog output.
- Additionally, this manual assumes the boards are equipped with an analog to digital converter (ADC0831).



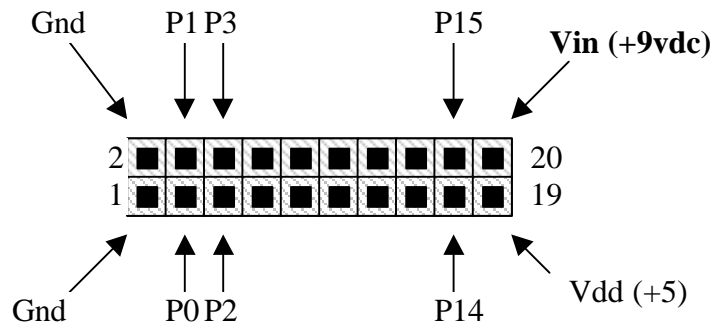
**Figure B-1: BASIC Stamp Activity Board (BSAB)**

The BS2 provides the user 16 pins for I/O devices, which are numbered P0 - P15. Since the Activity Board accommodates both the BS1 and BS2, each I/O device on the board is dual numbered representing the I/O Pin number for each style of Stamp. The second number corresponds to the BS2. For example, in Figure B-2 the speaker (SPK) is number P4/11. For the BS2 this device is connected to P11. Table B-1 provides a summary of the BS2 I/O pin numbers and corresponding Activity Board devices.

BS2 I/O	Device
P0	Pin 4 of the RJ-11 Jack (Not used for these labs)
P1	Pin 1 of the RJ-11 Jack (Not used for these labs).
P2	No Connection
P3	No Connection
P4	No Connection
P5	No Connection
P6	No Connection
P7	Potentiometer
P8	Blue Pushbutton/ LED
P9	Black Pushbutton/ LED
P10	Green Pushbutton/ LED
P11	Red Pushbutton/ LED/ Speaker
P12	Socket A, Pin 1 ( <b>A/D Enable</b> )/ X3 Analog Output
P13	Socket A, Pin 5 / Socket B Pin 3
P14	Socket A, Pin 6 ( <b>A/D Dataout</b> ) / Socket B Pin 2
P15	Socket A, Pin 7 ( <b>A/D Clock</b> ) / Socket B, Pin 1

**Table B-1: Activity Board I/O Devices**

The I/O Pin Header provides direct access to P0-P15 plus pins for +5V, supply voltage (see note) and two ground connections. Figure B-1A provides information on pin numbering.



**Figure B-1A: BSAB I/O Pin Header**

**Vin Note: Pin 20 is *unregulated* voltage from the wall transformer! Use of this voltage into ANY activity board pins or devices will likely result in *permanent damage* to the BS2 Microcontroller or the device!**

### ***Simple I/O Communication***

The BS2 is a digital device and as such uses digital inputs and outputs. A digital I/O can be HIGH or LOW. HIGHs generally correspond to the supply voltage of the system, in this case 5 volts. A LOW is normally considered to be at ground voltage, or 0 Volts. Since digital systems work in binary, often times these HIGH and LOW states are referred to in their binary form, a HIGH being a binary '1' and LOW being binary '0'. If we considered this HIGH/LOW state to be a switch action, a HIGH could be considered ON, and LOW as OFF. Often times the terms that are used for the state is interchangeable.

In summation:      HIGH = 5V = 1 = ON  
                              LOW = 0V = 0 = OFF

P0 - P15, the I/O pins of the BS2, are bi-directional. That is, they can act as an input or an output depending on need. A single program may change the direction of the pin many times based on the application. In PBASIC2 it is necessary to first set the direction of the I/O, then to set the output state or read the input state. For now we will look at controlling pins individually. In later sections we will see how to control them in groups.

Control words DIR0 - DIR15 set the direction of the corresponding I/Os P0 - P15. **When set to '0' the pin will be an input. When set to '1', the pin will be an output.** For example, to set P9 to an output:

**DIR9 = 1**



Once the direction is set, we can set the state if it is an output, or read the state if it is an input. Two more control word sets are used for these purposes. IN0 - IN15 will read P0 - P15 if set as inputs. OUT0 - OUT15 will set the state of P0 - P15 if set as outputs. Let's code a couple of programs. This first one will simply light the blue button's LED (P8).

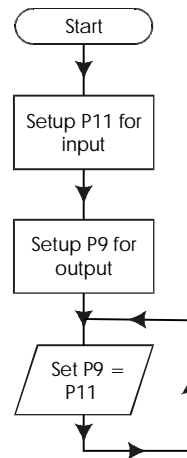
```
'PROG_B-1 (File name of program)
'Places a LOW on P8 as an output
DIR8 = 1      'Sets P8 for output
OUT8 = 0      'Sets P8 to LOW
```

**PBASIC TIP**

The ' is used to denote comments. They allow adding explanations to the program but are ignored when the program is tokenized.

This program will read one input (red button, P11) and set an output based on its condition (black button's LED).

```
'PROG_B-2
'Set P9 output to equal P11 input
'Depress Red Button while running
DIR11 = 0     'Sets P11 for input
DIR9 = 1      'Sets P9 for output
LOOP:
  OUT9 = IN11 'Set P9 = P11 state
  GOTO LOOP
```

**BASIC TIP**

LOOP: is a label. It is used to define a point in a program. GOTO will branch program execution to the defined label.

Table B-2 summarizes other PBASIC2 commands to deal with simple digital inputs and outputs.

Command word	Description
DIR0 - DIR15	Sets the direction of the I/O pin 0-15. 1 = Output. 0 = Input. e.g: DIR5=0.
IN0 - IN15	Reads the state of the pin (0-15).
OUT0 - OUT15	Sets the state of the pin (0-15) if an output.
INPUT pin	Where pin = 0 to 15. Sets the corresponding pin DIR to 0 for input.
OUTPUT pin	Where pin = 0 to 15. Sets the corresponding DIR to 1 for output.
HIGH pin	Where pin = 0 to 15. Sets the corresponding pin DIR for output and sets the pin to 1, or HIGH.
LOW pin	Where pin = 0 to 15. Sets the corresponding pin DIR for output and sets the pin to 0, or LOW.
TOGGLE	Toggles the output state of pin, 0 → 1, 1 → 0. Sets it up as an output if it was an input.

**Table B-2: Simple Pin I/O commands.**

This program will use I/O commands to toggle 2 LEDs.

```
'PROG_B-3
'Alternates P8 and P9 states
HIGH 8      'set P8 to 5V
LOW 9       'set P9 to 0V
LOOP:
  TOGGLE 8   'change state P8
  TOGGLE 9   'change state P9
  DEBUG ? OUT8, ? OUT9    'show values
  PAUSE 1000 'pause for 1 second
  GOTO LOOP  'repeat
```

#### PBASIC TIP

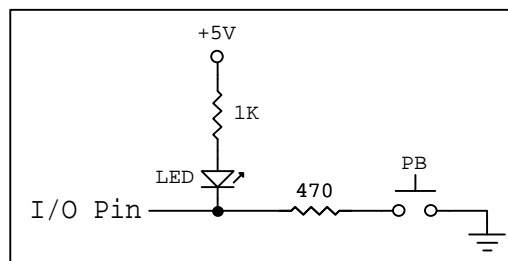
PAUSE is a delay. Its syntax is *PAUSE time*, where time is measured in milliseconds.

#### PBASIC TIP

DEBUG will send data back to the host computer to be displayed. See Ref. A for more information.

### Board LEDs and Buttons

If you have been entering and trying the examples in the previous discussion, you may have noticed that when a pin, such as P8, was set for an output and set LOW, the corresponding LED on the board lit. This is because the negative terminal (cathode) of the LED, a Light Emitting Diode, is connected to the output, and the positive terminal (anode) is connected to supply voltage (Vdd) through a resistor. The following is the schematic for the LED and button from the BS2 outputs.



#### 'TRON TIP

An LED can handle current up to around 30 mA. To prevent exceeding this, current limiting resistors are used.

An LED drops around 1.7 volts, leaving 3.3V of our original 5 volts.

$$I = V/R = 3.3/1000 = 3.3\text{mA}$$

**Figure B-2: Activity Board LED & Pushbutton Schematic**

When the I/O pin is a LOW (0V) output, current flows through the LED to positive voltage, and when HIGH, there is no current flow. Devices often are connected to be active on LOW outputs. In many cases an output can supply greater current as a LOW (current sink) than as a HIGH (current source). The BS2 can sink 25 milli-amps (.025 amps) of current as a LOW, but can source only 20 milli-amps as a HIGH.

As an input, the LED/resistor will also act to 'pull' the input up to 5V, or HIGH, when the pushbutton (PB) is not depressed since it is normally-open (N.O.). When a program reads the input from the pushbuttons, P8 - P11, it will see a HIGH or 1 when normally open, and a LOW or 0 when depressed (closed). Also, when a pushbutton is depressed, it will provide a path of current up to 5V and allow the LED to light. So as buttons are pressed on the Activity Board, the corresponding LEDs will light while the corresponding inputs will sense a LOW.

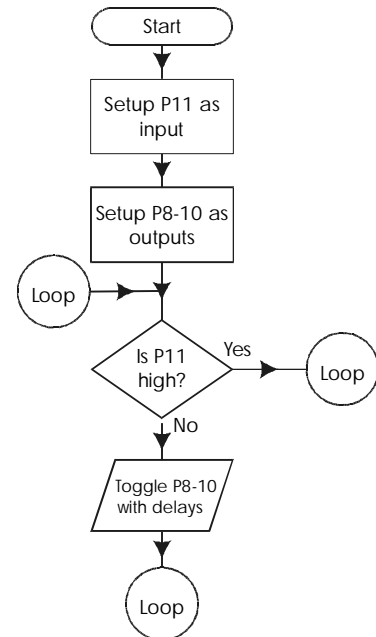
This program detects whether the red pushbutton is depressed (low), and if it is, cycle the other 3 LEDs:

```
'PROG_B-4
'Detects if RED PB is pushed
'and Cycles 3 LEDS
INPUT 11      'Set up I/O directions
OUTPUT 8
OUTPUT 9
OUTPUT 10

LOOP:
  IF IN11 = 1 then LOOP 'not pushed? Goto Loop.

  TOGGLE 8      'Toggle each output with 100 mS pauses
  PAUSE 100
  TOGGLE 9
  PAUSE 100
  TOGGLE 10
  PAUSE 100

GOTO LOOP 'Repeat
```



### Try it!

Change the program so that the outer 2 LEDs blink at the same time and alternate with the center one at 1/2 second intervals

### PBASIC TIP

*IF equality THEN label*  
IF-THEN are used for conditional branching. If the equality is true, the program will branch to the defined label. Equalities can use =, >, < and more.

### Activity Board Speaker

The speaker on the Activity Board shares I/O pin P11 with the red pushbutton. The speaker is capacitively coupled so that anytime the voltage on the pin changes from HIGH to LOW, or LOW to HIGH, the speaker will 'click'. By changing the rate, or frequency, at which this toggling takes place, different tones will be emitted by the speaker.

This program will produce a frequency of approximately 100 Hz (cycles/second). Pressing the BLUE button will end the program. Press the Activity Board 'Reset' to restart.

```
'PROG_B-5
'Sound speaker at 100 HZ (1/.010 = 100), Blue button to stop
OUTPUT 11      'set as output
INPUT 8        'set as input

LOOP:
  TOGGLE 11      'Change output state
  PAUSE 10       'pause for 10mS
  IF P8 = 1 THEN LOOP 'Loop until Blue pressed
END
```

To increase the frequency to 500 Hz, reduce PAUSE to 2 mS.

The stamp has many specialized control words for dealing with inputs and outputs. One of these is the FREQOUT command. Its syntax is:

*FREQOUT pin, duration, freq1{,freq2}*

Pin: The pin number 0-15.

Duration: The length of the tone in milliseconds

Freq1: The frequency of the tone in hertz. An optional second frequency can be used (freq2) to create a chord or a blend of frequencies. A frequency of 0 defines a rest, or no sound.

The following program will play the "CHARGE!" theme.

**'PROG\_B-6**

**'Plays the CHARGE! theme.**

**'Hit the Reset button on the activity board to replay it.**

**OUTPUT 11    'Set up direction for speaker**

**FREQOUT 11, 150, 1120**

**'Play notes**

**FREQOUT 11, 150, 1476**

**FREQOUT 11, 150, 1856**

**FREQOUT 11, 300, 2204**

**FREQOUT 11, 9, 255**

**FREQOUT 11, 200, 1856**

**FREQOUT 11, 600, 2204**

For using a combination of frequencies, the following program will play a dual tone, much like the tones used by the telephone companies (DTMF - Dual Tone Modulated Frequency).

**'PROG\_B-7**

**'Plays a dual tone (chord) of 2500 and 3000 Hz.**

**OUTPUT 11**

**FREQOUT 11, 2000, 2500, 3000**

#### **'TRON TIP**

Many commercial devices involve interfacing to telephones and utilizing the PIC's DTMF capabilities. In fact, PBASIC2 has a DTMFout instruction.

**NOTE: The RJ-11 Jack on the board is NOT a phone jack. Damage will occur if used as such.**

This program will use a loop to cycle through frequencies when the blue button is pressed.

```
'PROG_B-8
'Cycles through freq's when Blue Btn is pressed.
Freq VAR BYTE      'define a variable for freq
OUTPUT 11           'Set up I/O
INPUT 8
```

```
Loop:
IF IN8 = 1 THEN LOOP 'button not pressed, goto
Loop
```

```
FOR freq = 1 TO 60      'Define loop
    FREQOUT 11, 5, freq * 50 'Use loop variable times 50 for frequency
NEXT                    'End loop
GOTO Loop
```

### PBASIC TIP

*name VAR size.*

Where size can be either a bit (0-1), nib (0-15), byte (0-255) or a word (0-65535). See section C.

Variables can be defined to hold numeric values.

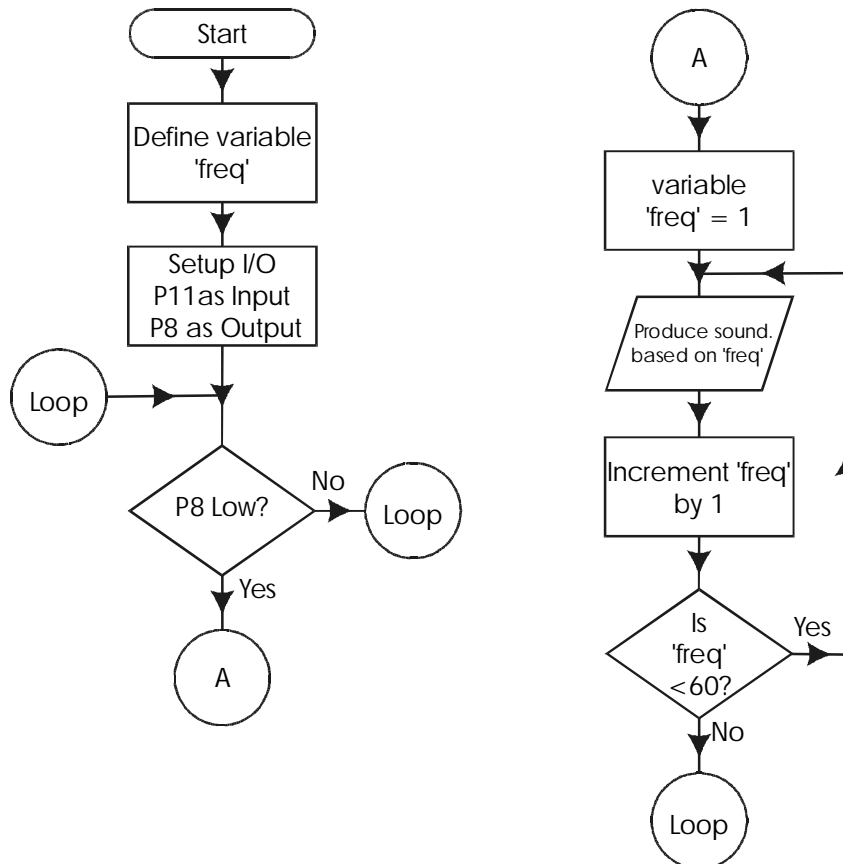
### PBASIC TIP

*FOR variable = start TO end*

*.*

*NEXT*

A FOR-NEXT loop will cause the code within the loop to be repeated. The variable will be set equal to the start value and will be incremented each iteration until it exceeds the end value causing the loop to cease.



**Debouncing Buttons**

When using inputs such as buttons, two issues can come into play as the program reads the input:

- 1) Did the button bounce when it was pressed?
- 2) What should be done if it is still closed next time I check it?

Mechanical switches and other inputs, such as optical sensors, can cause 'bounce' when actuated. With mechanical switches the contacts may make and break numerous times before they finally settle. A very fast processor may take this to mean the switch is pressed numerous times. Optical sensors are often used in 'non-contact' switching application such as conveyor counting. Products on the conveyor may vibrate and intermittently make & break the light beam as it passes between the light source and sensor.

Programmers need to take this into account. In many instances the input is read. If the switch is closed the program will pause and check it again to ensure it is still closed and stable. A flag is set to 'remember' that it was closed so next time around the action won't be repeated.

This program will check to see if the blue button is pressed and display a "\*" in the debug window for each time the button is sensed down. It uses no debouncing or checking.

```
'PROG_B-9
'Places a * in debug window whenever Blue is pressed.
'No debouncing or checking

INPUT 8
Loop:
  If IN8 = 0 THEN Action 'If button pressed, take action.
  GOTO Loop

Action:
DEBUG "*" "
GOTO Loop
```

It nearly impossible to press the button fast enough to perform the action only once. This next program is modified to take into account button bounce by simply allowing 50 milliseconds of settling time.

```
'PROG_B-10
'Places a * in debug window whenever Blue is pressed.
'Debouncing, no checking

INPUT 8
Loop:
  If IN8 = 0 THEN Action      'If button is pressed, take action
  GOTO Loop
Action:
  PAUSE 50                   'wait and see if still pressed
  IF IN8 = 1 THEN Loop
  DEBUG "*" "
GOTO Loop
```

By allowing settling time and pressing the button quickly, it is much easier to get only action once, but what if the 'box' gets stuck in front of the beam? How do we keep from counting it again? This program uses a flag to keep track of status of the input.

**'PROG\_B-11**

**'Places a \* in debug window whenever Blue is pressed.**

**'Debouncing and checking**

**INPUT 8**

**FlagON var bit 'variable to hold status bit**

**FlagOn = 0 'clear the flag**

**Loop:**

**If IN8 = 0 THEN Action 'If pressed, take action**

**FlagON = 0 'not pressed, reset flag (0)**

**GOTO Loop**

**Action:**

**PAUSE 50**

**IF (IN8 = 1) OR (FlagON = 1) THEN Loop 'If no longer pressed, OR the flag is set, skip**

**DEBUG "\*" "**

**FlagON = 1 'Once we do it, set the flag. (1)**

**GOTO Loop**

#### **PBASIC TIP**

IF-THEN statements can use Boolean expressions, such as AND & OR to qualify several expressions.

No matter how long the button is continually depressed, it will only display one "\*".

Since debouncing switches is such a common programming task, Parallax built into the PBASIC2 instruction set a command to deal with it: **BUTTON**.

*BUTTON pin, downstate, delay, rate, bytevariable, targetstate, label*

Pin: (0-15) defines the pin number of the input.

Downstate: (0 or 1) specifying which logical state occurs when a button is pressed. The Activity Board buttons are a 0 when pressed.

Delay: (0-255) determines how long to wait for the switch to settle. 0 and 255 are special cases. 0 for delay means do not debounce it or auto-repeat. 255 means to debounce, but do not repeat (see rate).

Rate: (0-255) is how fast the action should repeat if held down (similar to your computer keyboard keys).

Bytevariable: The name of a special variable of byte size that is used in debouncing and must be defined by the programmer.

Targetstate: The state of button on which to branch (0 = not pressed, 1 = pressed for activity board buttons).

Label: The program label to branch to when conditions are met.

Even though the command is called 'Button', it can be used with any type of digital input signal.

This program will produce a different tone based on the buttons, P8 - P10, depressed. Also, each one uses different debouncing techniques.

# 'PROG\_B-12

'Different tones for Blue, Black, Green with diverse BUTTON command uses.

```

Freq var word                                'set up variables
btnBLUE var byte
btnBLACK var byte
btnGREEN var byte

btnBLUE = 0                                  'initialize button bytevariables
btnBLACK = 0
btnGREEN = 0

Loop:
freq = 1500                                  'select frequency
BUTTON 8, 0, 255, 0, btnBLUE, 1, SNDOUT      'read blue button debounce, but no repeat
freq = 2000
BUTTON 9, 0, 50, 200, btnBLACK,1,SNDOUT      'read blk button, quick delay, slow repeat
freq = 3000
BUTTON 10, 0, 250,10, btnGREEN,1,SNDOUT      'read grn button, slow delay, fast repeat
GOTO Loop

SNDOUT:                                       'sounds tone based on frequency defined
FREQOUT 11, 100, freq

GOTO Loop                                    'Repeat

```

## Try it!

Change the program so that holding down ALL the buttons (except red) and releasing any one will produce the sounds.

Lets' look at those Button commands and break them down:

**BUTTON 8, 0, 255, 0, btnBLUE, 1, SNDOUT**

Button 8:	Use the Button command on P8.
0	Look for a state of 8 for action to occur.
255	Don't delay at all (do it right now!).
0	No repeat rate (do it once only).
btnBlue	This is a byte variable that is needed for the button command. It can have any name, this is just what was chosen. Define and clear it before using it!
1	Branch when the condition is true (button = 0)
SNDOUT	When all conditions are met, branch to this label.

The next button command is slightly different in that it has a of delay of 50 (short) and a repeat rate of 200 (long). The last one has a very long delay andt a very short repeat. In this manner we have very fine control of how we want our programs to react when a button (or any input signal) needs debouncing.



***Just for fun!***

Run the following program. In a dark room, wave the Activity Board back and forth while holding it firmly.

**'PROG\_B-13**

**'Running down the Highway!**

**'Run program, turn off lights, carefully wave board.**

**LOW 10**

**LOW 8**

**HIGH 9**

**Loop:**

**PAUSE 2**

**TOGGLE 9**

**PAUSE 5**

**TOGGLE 9**

**GOTO loop**

**'TRON TIP**

When still, the center LED appears always on, though it is actually blinking at a high rate. By waving it you circumvent your eye's persistence of vision to see the blinking. Try this with an alarm clock with an LED display to see the segment scanning.

## Section C: Binary Numbers

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Discuss the relationship between the decimal and binary number systems.
- 2) Discuss the need to use binary numbers.
- 3) Convert binary numbers to decimal.
- 4) List the number of bits and range for nibbles, bytes and words.
- 5) Write PBASIC2 code to use bit groups for inputs and outputs.
- 6) Write PBASIC2 code to use the debug window to show data.

### Decimal Number System

In our decimal number system we use unique symbols to represent 10 different levels or states. These symbols are 0 - 9, our numbers. But we are not limited to just representing only 10 levels because we can combine these numbers to represent even higher numbers. Once we have counted past the first ten digits (0-9) we add a new place and start over again. Looking at a number, such as 1456, we know from common knowledge that this represents one thousand, four hundred and fifty-six.

Looking at this number mathematically each place that we add represents a higher power of our base, which is 10.

Place	4 <sup>th</sup>	3 <sup>rd</sup>	2 <sup>nd</sup>	1 <sup>st</sup>
Weight	$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
Number	1	4	5	6
Result = Number x Weight	1000	400	50	6

**Table C-1: Decimal Weights**

Our final number is the sum of the results for each place:

$$1000+400+50+6 = 1456$$

### Digital and Binary

Microcontrollers, computers and other digital electronics work only in 2 states: HIGH or LOW. This limits them to a numbering system with only 2 digits. This is the binary number system. Just as in decimal, where there are 10 unique digits from 0 to 9, in binary there are 2 unique digits: 0 and 1.

We are not limited to representing only two possible values of course. After we count through our series, 0 - 1 in base 2, we add a column with the next higher weight. A binary number such as 1011 is shown broken down in Table C-2.

Place	4 <sup>th</sup>	3 <sup>rd</sup>	2 <sup>nd</sup>	1 <sup>st</sup>
Weight	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Number	1	0	1	1
Result = Number x Weight	8	0	2	1

**Table C-2: Binary Weights**

Again, our final number is the sum of the results for each place. The number converted to decimal would be:

$$8 + 0 + 2 + 1 = 11$$

To differentiate between number bases, since 1011 can have different meaning in different bases, numbers not in base 10 (decimal) are written with a subscript representing the number base:  $1011_2$  represents a number in base 2 or binary.

**PBASIC TIP**

To indicate a number is written in binary, precede it with %.  
InNib = %1011

***Nibbles, Bytes and Words*****Nibbles**

Binary digits (bits) are commonly grouped together to represent a range of decimal numbers. A single bit has 2 possible values, 0 or 1. A grouping of 4 bits is referred to as a *nibble*. A 4-bit nibble has 16 possible combinations of 1's and 0's, and can represent the decimal numbers 0 - 15. Table C-3 shows the binary and decimal number for 0 - 15.

Binary	Decimal
0000 <sub>2</sub>	0
0001 <sub>2</sub>	1
0010 <sub>2</sub>	2
0011 <sub>2</sub>	3
0100 <sub>2</sub>	4
0101 <sub>2</sub>	5
0110 <sub>2</sub>	6
0111 <sub>2</sub>	7
1000 <sub>2</sub>	8
1001 <sub>2</sub>	9
1010 <sub>2</sub>	10
1011 <sub>2</sub>	11
1100 <sub>2</sub>	12
1101 <sub>2</sub>	13
1110 <sub>2</sub>	14
1111 <sub>2</sub>	15

**Table C-3: Binary vs. Decimal Numbers**

Let's try a conversion to confirm the table. The binary number  $1101_2$  has a 1 in the 8's, 4's and 1's place and a 0 in the 2's place. Converting this binary number to decimal would be:  $8 + 4 + 1$  or 13.

Two formulas can be used to calculate the number of possible combinations in a binary number and the highest decimal count the binary number can represent based on the number of places.

$2^n$  = Number of combinations or states, where n is the number of bits.

$2^n - 1$  = The highest decimal equivalent.

Since a nibble is 4 bits the number of possible combinations of 1's and 0's would be  $2^4$  which equals 16. So a nibble has 16 states that it can represent. The highest decimal number a nibble can represent is calculated by  $2^4 - 1$  which is 15.

### Byte

The next highest grouping is that of 8 bits. This is referred to as a *byte*. This is the most common term used in representing digital values. The possible combinations in a byte is equal to  $2^8$  or 256. The highest possible count is  $2^n - 1$  or 255. As you can see, as we add bits our number size can grow rapidly. The binary weights of the bit positions in a byte are:

128 64 32 16 8 4 2 1

So a number such as  $0100110_2$  would equal  $64 + 4 + 2$  or 70.

### Word

A grouping of 16 bits is often referred to as a *word*. The possible combinations in a word is equal to  $2^{16}$  or 65536. The highest possible count is  $2^{16} - 1$  or 65535. In some literature 'word' is used to simply refer to the number of bits a microcontroller or microprocessor works with.

Depending on the processor, 8, 16 or 32 bit word groupings are common. In using the BS2 the term 'word' will imply a 16-bit binary number.

### Using Debug to Display Binary Data

The debug window has been used earlier in allowing the BS2 to send information back to the host computer. How and what DEBUG displays is controlled by the syntax, or structure, of the statement. Let's take a look at a few methods to write debug statements. See reference A for more detail.

**DEBUG ? IN5**

'The ? tells DEBUG to display variable AND it's value in decimal.  
'It also inserts a carriage return (CR) so the next statement printed is  
'on following line.

Output:

IN5 = 0

**DEBUG 65**

'Displays the ASCII character for 65, or whatever  
'number or variable follow. Does NOT add a CR.

**x var byte**

**x = 66**

**DEBUG x**

'Displays the character 'B'

Output:

AB

**DEBUG DEC 65**

'Displays the decimal 65. Does not add a CR.

**DEBUG IBIN8 66**

'Displays 66 in binary. 8 indicates to pad out to 8 bits even if 0. No CR.  
' % indicates a binary number.

Output:

64%01000001

**DEBUG "HELLO"**

'Displays the string, No CR.

Output:

HELLO

#### 'TRON TIP

ASCII is a  
standardized code  
which defines a  
character, number,  
and control codes  
for displays or  
printing.

**'Use a comma to have multiple output statements, use CR to force a carriage return  
 DEBUG "The ASCII character 67 is ", 67, CR,"And 67 in binary is ", IBIN8 67, CR**

Ouput:                      The ASCII character 67 is C  
                               And 67 in binary is %01000010

Let's write a program that will count from 0 to 15 and display the results in both binary and decimal. Since we will not exceed a count of 15, we can define this as a nibble, or 'nib'. A loop will allow us to go through the count.

```
'PROG_C-1
'Count 0 - 15 and display in decimal & binary

DataCnt var nib
FOR DataCnt = 0 to 15                      'show data and add a carriage return
    Debug DEC DataCnt, " Decimal = ", IBin4 DataCnt," Binary", CR
Next
```

The output of the program displays data for each value starting with:

0 Decimal = %0000 Binary

and ending with:

15 Decimal = %1111 Binary

We could use binary numbers to define a count as in the following program that counts from 64 to 85 denoted by binary values (note that the program was changed to use bytes versus nibbles since our numbers are > 15 and < 256):

```
'PROG_C-2
'Count 64-85 defined by binary value, show results in binary & decimal

DataCnt var byte
FOR DataCnt = %01000000 to %01010101      '(64 to 85 in binary)
    Debug IBIN8 DataCnt, " Binary = ", DEC DataCnt, " Decimal", CR
Next
```

**Try it!**

Change the program count from 1000 to 1050 in decimal!

### PBASIC2 I/O using Nibbles, Bytes and Words

Just as PBASIC2 can use commands such as IN3 to address the single input on pin P3, it also allows us to address the 16 I/O pins in groups of bits as nibbles, bytes, or the entire word. This provides the ability to read-from or write-to a group of bits simultaneously.

Table C-4 shows the words associated with each I/O (P0 - P15) and the direction command associated with each of the single bits and the bit groups.

Word Name	DIRS															
Byte Name	DIRH								DIRL							
Nibble Name	DIRD				DIRC				DIRB				DIRA			
Bit Name	DIR15	DIR14	DIR13	DIR12	DIR11	DIR10	DIR9	DIR8	DIR7	DIR6	DIR5	DIR4	DIR3	DIR2	DIR1	DIR0
Pins	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

**Table C-4: Direction Command Structures**

It is easiest to set these words using the binary values since each bit represents one I/O. For example: **DIRB = %1101** would set P7, P6 and P4 as outputs since they have 1's in the places represented by them. P5 would be set as an input since its value is 0.

The code **DIRB = 13** would have the same effect, but it is harder to visualize since we do not have 1's and 0's to relate the bit positions to.

In the same manner, we could use DIRH (H stands for high-order byte) to simultaneously set P8 - P15 or DIRL (low-order byte) for P0 - P7. The following two tables show the associated IN and OUT commands for the bit groups.

Word Name	OUTS															
Byte Name	OUTH								OUTL							
Nibble Name	OUTD				OUTC				OUTB				OUTA			
Bit Name	OUT15	OUT14	OUT13	OUT12	OUT11	OUT10	OUT9	OUT8	OUT7	OUT6	OUT5	OUT4	OUT3	OUT2	OUT1	OUT0
Pins	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

**Table C-5: Input Command Structures**

Word Name	INS															
Byte Name	INH								INL							
Nibble Name	IND				INC				INB				INA			
Bit Name	IN15	IN14	IN13	IN12	IN11	IN10	IN9	IN8	IN7	IN6	IN5	IN4	IN3	IN2	IN1	IN0
Pins	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

**Table C-6: Output Command Structures**

The following program will set P8 - P11 (our buttons/LEDs) to outputs, count from 0 - 15, and display the binary value on the LEDs.

```
'PROG_C-3
'Show binary counting on LEDs. (reversed so 0 = ON)

BinaryCount var nib          'Define nibble variable for counting
DIRC = %1111                 'Set P8-P11 nibble as output

Loop:
FOR BinaryCount = 0 TO 15    'Start Counting
  PAUSE 500                  'Pause 1/2 second
  OUTC = BinaryCount ^ %1111 'Reverse the nibble and send to output
NEXT
PAUSE 3000                   'Wait 3 seconds
GOTO Loop                    'Repeat
```

**PBASIC TIP**

Since LEDs and Pushbuttons are "LOW - ON", the ^ (XOR) operator is used to reverse the state. More about this will be covered in later sections.

The following program will read all the pushbuttons simultaneously as an input nibble, store them as a variable and display the results in binary and decimal.

```
'PROG_C-4
'Read buttons as a nibble and store
'(Reversed so Pressed = '1')

NibIn var nib                'Define a nibble variable
DIRC = %0000                 'Set P8 - P11 as inputs

Loop:
NibIn = INC ^ %1111          'Read input, and reverse it
Debug IBIN4 NibIn, " ", DEC NibIn, CR
GOTO LOOP                    'Repeat
```

**'TRON TIP**

The movement of groups of bits simultaneously is known as *parallel* transfer or communications.

**MSB & LSB**

4 decimal digits represent a decimal number such as 1450. The place to the far-left carries the most weight, one thousand. This place can be referred to as the most significant place because changes such as from 1 to 2 would represent changes of thousands of units. Where-as the far-right place can be referred to as the least significant place because a change of 1 carries little weight in the overall value of the number. If our number were 1.486 the 1's place would now be the most significant place because it would most dramatically affect the overall value.

Binary number places are often referred to by their most and least significant bits. So in a binary number such as 1010 the far-left 1 carries the most weight (8's place) and the far-right bit carries the least weight (1's place) of the number. The *most significant bit* farthest to the left is called the *MSB* of the number, and the *least significant bit* to the far right is the *LSB*.

These terms are important to understand because digital values are often passed between systems in a serial fashion. It is important to know whether the originating system is sending out the LSB or the MSB first so that it can be re-assembled correctly in the receiving system. If a device sends out data LSB first, such as 1011<sub>2</sub> sent as 1 then 1 then 0 then 1 (LSB to MSB), the receiving device must reassemble it as 1011 versus 1101.

**'TRON TIP**

The movement of groups of bits one bit at a time is known as *serial* communications.

In the following program the operator would enter one bit at a time to represent the binary number.

#### 'PROG\_C-5

'The user will press the Black button to start the data acquisition.

'Blue's LED will flash 4 times for about 1 second each.

'The user will enter their bit by pressing the Red button while the Blue LED is ON.

'The program will invert the button so 1 = pressed, 0 = not pressed.

'And re-assemble the 'nibble' MSB first.

```
NibData  var nib      'holds the incoming nibble
BitCount var nib      'Used to count the 4 bits
BitIn    var bit      'Holds incoming bit
Timer    var word     'Used to wait for input for 1 second
```

Loop:

IF IN9 = 1 THEN Loop

'Wait for Blk button

PAUSE 1000

'Wait one second

NibData = 0

'Reset Nibble variable

FOR BitCount = 1 TO 4

'Loop to get 4 bits

    NibData = NibData << 1

'Shift bits in nibble over 1

    BitIn = 0

'Set incoming bit to 0

    LOW 8

'Light Blue's LED

    FOR timer = 1 TO 800

'Loop for about one second

        IF IN11 = 1 THEN Not1 ' If at anytime red is pressed (0),

        BitIn = 1

' set the incoming bit equal to 1

    Not1:

    NEXT

    HIGH 8

'Turn off Blue's LED

    NibData = Nibdata + BitIn

'Add new bit to nibble

    DEBUG IBIN4 ? NibData

'Display new nibble in binary

    PAUSE 1000

'Wait a second

NEXT

'Go back for next bit

DEBUG ? NibData

'Display the nibble in Decimal

GOTO Loop

'Start all over

#### 'TRON TIP

In electronics terminology, the information the you give the program by pressing the button is the *data*.

Blue's LED blinking to tell you to send the data is the *clock*.

#### PBASIC TIP

<< and >> shift bits in a group by the amount specified, left or right respectively.

0110 <<1 → 1100

#### PBASIC TIP

A For-Next loop within a For-Next is known as a nested loop.

#### Try it!

Change the program to receive LSB first. Hint for one method: We added '1' to the LSB of Nibdata. What would we have to add for the MSB of NibData?

In the program we are the transmitter of the nibble, and the BS2 is the receiver. The program is expecting the MSB of our binary number first. To send it the number 12, binary 1100, we WOULD press the red button on the first 2 flashes to represent the first two 1's. We WOULD NOT press the red button on the last two flashed to represent the two 0's.



The output of the debug window would be:

```
NibData = %0001  
NibData = %0011  
NibData = %0110  
NibData = %1100  
NibData = 12
```

The last line shows our number in decimal, the number 12 as we meant to send it.

If we sent the data LSB first, as 0 then 0 then 1 then 1, the debug screen would display:

```
NibData = %0000  
NibData = %0000  
NibData = %0001  
NibData = %0011  
NibData = 3
```

The number 3 is NOT what we intended to send it!

## Section D: Analog Inputs and Outputs

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Discuss how digital values are representative of analog quantities.
- 2) Discuss how Pulse Width Modulation can be used to vary the output of a device.
- 3) Write PBASIC2 code to use the PWM command.
- 4) Discuss analog to digital conversion and quanta levels.
- 5) Program the BS2 to gather data from the Activity Board 0831ADC
- 6) Program the BS2 to read the Activity Board potentiometer.

Ours is an analog world. Very few things are black and white (or on and off). An analog range has a minimum and maximum limit and an infinite number of possible values 'in-between'. The temperature in a room may be 68, 69 or maybe 70 degrees. We can become more precise in our measuring and say it is 69.3, or 69.28, or even 69.2835 degrees. Depending on how precise we want to measure it there can be an infinite number of possible temperatures.

While digital inputs are either HIGH or LOW, ON or OFF, we can use collections of bits to represent a finite quantity of analog values. The *resolution* of the analog quantity we want to measure or control depends on the number of bits that are utilized. Take for example a dimmer switch for a light in a room. With a bulb size of 150 watts the dimmer switch can vary the intensity anywhere from fully-off to fully-on with an almost endless number of possibilities. The degree of brightness is limited only by how finely we can control the dimmer knob.

A digital equivalent may be a situation where we turn on or off a combination of bulbs to achieve the lighting level we desire. Assume that within a room light fixture, there are four lamps controlled by four switches. The wattage rating of each lamp is selected based on the binary weighting. If the smallest lamp (LSB) is 10 watts, the three other bulbs would be 20 watts, 40 watts, and 80 watts respectively. By selecting the combination of bulbs that are ON, the room light can be adjusted from totally dark (all OFF = 0 watts.) to very bright (all ON=150 watts). By controlling the ON/OFF state of the four switches, a total of 16 possible light levels can be achieved. Although this 4-bit group cannot provide a true analog between 0 and full on, its ability to resolve 16 light levels may be suitable for this application. Table D-1 lists the 16 possible switch combinations, the decimal equivalent of this binary number, and the total light output level that would result in our example.

### **'TRON TIP**

*Lumens* is the actual term used to indicate the intensity or output of a bulb. *Watts* is the power consumed, much of which is wasted as heat. Watts is used here because it is the more common term.

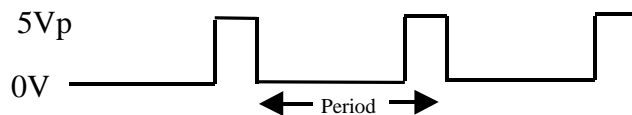
DECIMAL	MSB 8 (80 W)	4 (40 W)	2 (20W)	LSB 1 (10 W)	OUTPUT (WATTS)
0	0	0	0	0	0
1	0	0	0	1	10
2	0	0	1	0	20
3	0	0	1	1	30
4	0	1	0	0	40
5	0	1	0	1	50
6	0	1	1	0	60
7	0	1	1	1	70
8	1	0	0	0	80
9	1	0	0	1	90
10	1	0	1	0	100
11	1	0	1	1	110
12	1	1	0	0	120
13	1	1	0	1	130
14	1	1	1	0	140
15	1	1	1	1	150

**Table D-1: Lighting Digital Equivalents**

As you can see, using 4 switches representing a nibble (4 bits) we can resolve 16 different levels with step sizes of 10 watts. Adding 4 more switches and incrementing the bulb sizes even closer, say using 1, 2, 4, 8, 16, 32, 64 and 128 watt bulbs (if available!), we could reproduce finer control. Using 8 bulbs we could control the lighting from 0 watts to 256 watts in step sizes of 1 watt providing a resolution of 256, or 8 bits. This would more closely mimic the operation of our analog dimmer switch.

### ***Pulse Width Modulated Output***

One method used in having analog control from a digital system is through the use of pulse width modulation (PWM). In PWM the output is cycled between HIGH and LOW over a short period of time. The longer the output is HIGH, the greater the average output is. Take for example the waveform in Figure D-1.

**Figure D-1: PWM Waveform**

The wave is HIGH, at 5 Volts, only about 25% of the time during its period. If the voltage is averaged out over time, the average voltage would be 1.25V (0.25 x 5V). The percentage of time that a wave is HIGH over during its period (from one repetition to the next) is known as Duty Cycle. Figure C-1 has a duty cycle of 25%. The average voltage output of a PWM signal is equal to the peak voltage times the duty cycle. In this case:

$$V_{avg} = 5V \times .25 = 1.25V$$

A PWM application does not necessarily have to be 'filtered' to produce an equivalent analog voltage. By applying the HIGHS and LOWs directly to an ON/OFF device, we control the average power applied. A heater may generate 500 watts of heat when on. If a PWM wave is

used to turn the heater on and off rapidly with a 10% duty cycle, it would cause the heater to produce an apparent 50 watts of heat.

The LEDs on the activity board are either on or off, producing only 2 levels of light. Using the PBASIC command PWM we can apply rapid pulses of varying duty cycles to the LEDs producing. Our eyes do not detect the rapid pulses of light, instead they perceive the average power represented by the duty cycle. The syntax for the PWM command is:

*PWM pin, duty, cycles*

Pin: 0 - 15 corresponding to the output pin.

Duty: 0 -255, where 0 = always off and 255 = always on (255 would be a 100% duty cycle).

Cycles: 0 - 65535 specifies the approximate period of the wave in milliseconds.

**'PROG\_D-1**

**'This program will alternately dim and brighten the P8 LED**  
**LED con 8**

**Duty var byte**  
**OUTPUT LED**

**'Set LED to constant 8**

**'Define a byte variable for duty**  
**'set LED to be an output**

**Loop:**

**FOR Duty = 0 TO 255**

**'Cycle through the 255 levels**

**PWM LED, Duty, 20**

**'Apply PWM to LED for 20mS each pass**

**NEXT**

**'Slowly dims LED (high = off!!)**

**FOR Duty = 255 TO 0**

**'Cycle through backwards**

**PWM LED, Duty, 20**

**NEXT**

**GOTO Loop**

**'Repeat**

### PBASIC TIP

Con is similar to var. Instead of creating a variable, which can be changed, it creates a constant which is fixed to a value. These values are not stored in memory, but used when the program is tokenized, saving valuable memory.

### PBASIC TIP

Unlike most BASICs, PBASIC understands that a FOR-NEXT with a start value less than end value requires it to count backwards. No negative step value is required to be stated.

### Try it!

Change the program to work in the most noticeable area of change of the LED brightness only!

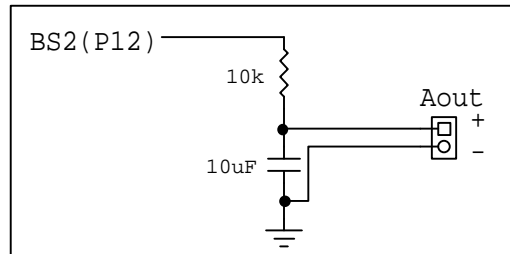
A multimeter connected between P8 and Gnd on the I/O Pin Header would display voltages from 5V to 0V as the LED brightens. Is this the ACTUAL voltage on P8? No, the voltage is really bursts of 0V and 5V that make up the PWM wave. Multimeters perform sampling and average out the voltage. Just as your eye was averaging the light it was seeing, the multimeter is acting as a low-pass filter. The high frequency pulses are being averaged out and being seen as varying DC levels.

### 'TRON TIP

The better voltmeters actually measure the RMS ( root mean square) of the changing voltage, though for a square wave, like our pulses, equates equally for RMS or average.

### Activity Board Aout

The Activity Board has a set of pins, X3, labeled Aout for analog output. This output is connected to P12 of the BS2. Aout has an RC (resistor-capacitor) network (Figure D-2) attached to it which (like the heater, our eyes, and the voltmeter) performs averaging of the pulses to produce an analog value directly proportional to the PWM applied to it.



**Figure D-2: Aout RC Network**

A capacitor acts as a storage device. As voltage rises, it will store the voltage. As voltage decreases, it will begin to discharge that voltage. The resistor in series with it from the source of the voltage will limit how quickly it can charge and discharge. Essentially, when the PWM voltage is applied to the RC network it will average out the quickly changing voltage to produce a fairly constant variable DC level between 0 and 5 volts.

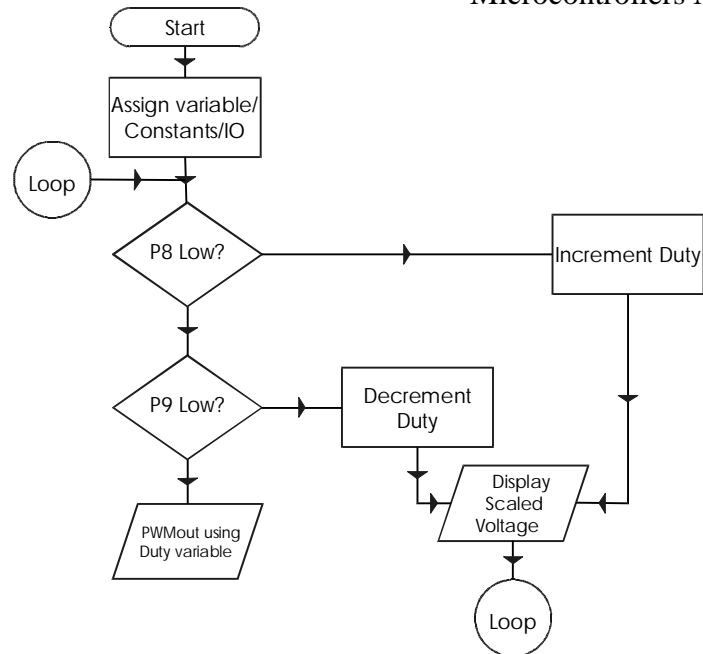
Since the PWM command uses a byte to control the duty cycle, we can resolve the voltage down to a value defined by a transfer function of:

$$\frac{\text{Range of Output}}{\text{Range of Input}}$$

Where Output is the 0 - 5V span, and Input is the 0-255 resolution of duty, so  $5V/255 = .0196$ . In other words, for each 1 bit of change in the duty, the output voltage would change by .0196 volts. Based on a given input we can calculate the output voltage with the following equation:

$$V_{out} = \text{Duty}(5V/255) = \text{Duty} * .0196$$

For example, a PBASIC duty of 150 on the PWM pulse would calculate out to an output voltage of 2.94 volts. Let's try a program to adjust the DC voltage on this output by varying the duty of the PWM.

**'PROG\_D-2**

'This program starts at a midscale duty of 128, or 2.5 volts, on Aout

'Use the Blue button to increase duty, Red to lower.

'Debug will display duty and Voltage in tenths

Aout con 12

'constant for P12

Duty var Byte

'Duty variable

volt\_x\_10 var word

'Output volt \* 10 variable

RedBtn var byte

'Red button bytevariable

BlueBtn var byte

'Blue button bytevariable

INPUT 8

'Set up directions

INPUT 11

OUTPUT Aout

Duty = 128

'Set to mid-scale

GOTO Display

'Display data

**PBASIC TIP**

PBASIC uses integer math, so there are no fractions. In using math we need to ensure we don't lose accuracy in fractions.

**Loop:**

BUTTON 8,0,0,100,Bluebtn, 1, Raise 'Blue btn - Raise Aout

BUTTON 11,0,0,0,redbtn, 1, Lower 'Red btn - Lower Aout

PWM Aout, Duty, 100 'Burst of PWM

GOTO Loop

**Raise:**

Duty = Duty + 1 max 255 'bump up duty

GOTO Display

**Lower:**

Duty = Duty - 1 min 1 'bump down duty

GOTO Display

**PBASIC TIP**

Min and Max can be used so we don't exceed maximum or minimum values (0 - 255 for a byte).

In math:

0 - 1 = 255 → rolls back to max value.

255 + 1 = 0 → rolls over to min value.

**Display:**

volt\_x\_10 = 50 \* Duty / 255 'Calc expected volt

Debug "Duty = ", DEC Duty, ", Volt x 10 = ", DEC volt\_x\_10, CR 'Display duty & volt

GOTO Loop

Running the program with a voltmeter across Aout would show the initial voltage around 2.5 volts. Pressing the Red button will increase the duty cycle, and the Blue button will lower it. In

the debug window is displayed the calculated voltage multiplied by 10 (PBASIC does not work in fractions, so we just shift our number over). So a value of 2.5 volts would appear as 25 tenths. As the duty cycle is changed, the new value is displayed.

### **Analog to Digital Converters (ADC)**

**NOTE: Do not use this manual in connecting analog inputs to the Activity Board. Please use the associated labs.**

Just as a system can use a group of digital bits to define a finite amount of varying voltage output levels, there are also means to bring analog values into a digital system. A digital thermostat uses a sensor to measure the varying temperature of a room by producing an analog voltage proportional to room temperature. It then converts that voltage into a series of 1's and 0's to be represented in a digital system.

A variety of integrated circuits are available that can perform the task of analog to digital conversion. The purpose of an analog to digital converter (ADC) is to relate a range of analog input values to a range of binary representations. As stated earlier, an analog range contains an infinite number of possible values between its minimum and maximum. The finite resolution of a binary word forces the ADC to use each of its possible combinations to represent a segment of the analog input range. For example, assume we are covering a 0-5 volt analog range with a 4-bit binary number. The 4-bit binary number can represent a range of 0-15. Dividing the 5 volt analog range up into 15 equal “compartments” results in approximately .33 volts per compartment.

These compartments are termed *quanta levels*. The binary output of the ADC would allow you to predict the input voltage with some degree of accuracy. The inherent error due to the finite nature of the binary conversion is equal to  $\pm \frac{1}{2}$  of the quanta level value. In this example, the quantization error is  $\pm .167$  volts (or  $\pm \frac{1}{2}$  1 LSB). Table D-2 lists the input voltages, the binary conversions, and decimal equivalents of the example.

INPUT VOLTAGE ( $\pm .167$ )	MSB 8	4	2	LSB 1	DECIMAL VALUE
0	0	0	0	0	0
.33	0	0	0	1	1
.67	0	0	1	0	2
1.00	0	0	1	1	3
1.33	0	1	0	0	4
1.67	0	1	0	1	5
2.00	0	1	1	0	6
2.33	0	1	1	1	7
2.67	1	0	0	0	8
3.00	1	0	0	1	9
3.33	1	0	1	0	10
3.67	1	0	1	1	11
4.00	1	1	0	0	12
4.33	1	1	0	1	13
4.67	1	1	1	0	14
5.00	1	1	1	1	15

**Table D-2: Analog Voltage to Binary**

Using only four bits to represent an analog range results in relatively poor resolution and a high degree of quantization error. Analog to digital converters that convert to 8-bit bytes yield better resolution (255) and lower quantizing error. More precise applications may require 12-bit converters that have a resolution of 4,095.

The analog to digital converter included on the Activity Board is the ADC 0831. It is an 8-bit converter and it is spanned to cover an analog input range of 0 to 5 volts. Given this span and its resolution of 255, the range is divided up into 19.6 mV quanta levels. If the input analog voltage is known, the digital output can be predicted by:

$$\text{Digital value} = \frac{V_{in}}{\text{quanta level}} .$$

For example, if the input voltage is 2.5 volts, the digital conversion will be:

$$\begin{aligned} \text{Digital value} &= 2.5 / .0196 = 127.6 \text{ rounded up} = 128_{10}. \\ &(\text{Remember the resolution is finite and } 127.6 \text{ would fall into the next quanta level, } 128). \\ \text{The ADC would be holding this value as a binary number} &= 10000000_2 \end{aligned}$$

Of course, if you knew the digital converted value and the quanta level value, you could predict the input voltage. Rearranging the formula:

$$V_{in} = \text{Digital value} * \text{quanta level}$$

For example, if the digital conversion is 87, the input voltage is approximately

$$V_{in} = 87 * .019 \text{ V} = 1.70 \text{ Volts}$$

Table D-3 lists various  $V_{in}$  values and their conversions based on the ADC 0831 when spanned 0 to 5 volts.

VIN	DECIMAL VALUE	BINARY VALUE
4.6 volts	235	11101011
3.51	179	10110011
2.49	127	01111111
1.86	95	01011111
1.274	65	01000001
.06	4	00000100

**Table D-3: 8-Bit ADC Examples**

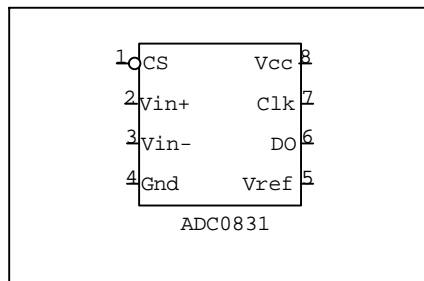
Once the analog input value is converted to a binary number it needs to be made available for the digital system to use. Two methods are available for this. The easiest is to simply use 8 pins on the ADC which through 8 data lines can transfer the information to another component, such as a microcontroller. This is known as a parallel output. While this is very fast and simple, when we



have a limited number of I/O lines to work with, such as in the BS2, it would greatly reduce the number of I/O's we can have for other devices.

The second method is that once the analog value is converted to an internal binary number, the data is shifted out in a serial fashion. The processor requests the bits of the data one at a time, and re-assembles the entire byte internally. The trade off for line reduction is complexity in transfer.

The ADC0831 included on the Activity Board is a serial 8-bit ADC. Figure D-3 shows the pins of the ADC0831 and Table D- 4 describes their purpose.



**Figure D-3: ADC 0831**

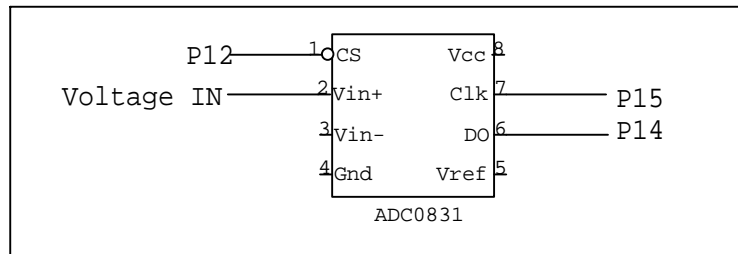
PIN	Description
CS	Chip Select: Brought LOW (0) to start a conversion.
Vin+	Analog Input, 0 - 5V.
Vin-	Lower reference voltage. Normally at ground voltage, but may be a higher. Defines low end of Analog voltage in.
Gnd	Ground power connection for IC.
Vref	Normally connected to 5V, but may be adjusted for precision critical applications.
DO	Data Out: The bits appear serially from this output to be collected by processor.
Clk	Clock Input: Processor 'clocks' or pulses this input to request each bit of data.
Vcc	+5V power connection for IC

**Table D-4: ADC0831 Pin Descriptions**

On the ADC0831, when the CS input is brought LOW, a voltage of 3.51 would be converted to the digital equivalent of 179. This value in binary would be  $10110011_2$ . The input voltage of 3.51 V would be converted and stored in the ADC0831 as 10110011. To transfer that data to the BS2, the program needs to place a series of HIGH and LOWS on the Clk input and collect the bits as they appear on the DO output pin.

The very first clock pulse to the ADC0831 will cause a HIGH to appear on the DO output regardless of the converted number, while each subsequent pulse will cause the binary number to be shifted out. This first HIGH is considered a start-bit, and some programs may check to ensure it is present for proper operation.

The data leaving the DO output comes out as Most Significant Bit (MSB) first, which refers to the left-most bit of out number 10110011. This is important to know so they are re-assembled in the correct order. As these bits are shifted out and collected by the processor they must be shifted into a variable so the original byte is reconstructed.



**Figure D-4: ADC0831 Activity Board BS2 Pin Numbers**

The following program demonstrates how to control the ADC0831 on the activity board to perform a conversion and gather the data. Figure D-4 shows the AC0831 with associated BS2 pin numbers. It assumes that a voltage source is connected to the Analog Input of the ADC0831 (see associated labs).

#### 'PROG\_D-3

'This program will pulse the 0831 Clk, acquire each bit and reassemble  
'the byte representing temperature.

CS con 12

Clk con 15

Datain var Byte

Bitcnt var Byte

Loop:

Datain = 0

LOW Clk

LOW CS

FOR bitcnt = 1 TO 8

PULSOUT Clk, 5

Datain= Datain + IN14

Datain = Datain << 1

NEXT

HIGH CS

DEBUG ? Datain

DEBUG ? Datain/5

Pause 1000

GOTO Loop

'define constants & variables

'holds incoming number

'For-next variable

'clear variable for new conversion.

'Ready the clock line.

'Select the chip.

'Start loop (Start bit is

'automatically shifted out)

'pulse the clk pin

'Collect the bit

'Shift it over

'Done, deselect the ADC

'Display it on PC

'Convert to tenths of a Volt (50/10)

'pause for a second

'repeat

#### PBASIC TIP

<< and >> specify to  
shift the bits in a variable  
left or right respectively.  
The number following  
the symbol specifies the  
number of bits to shift.

#### PBASIC TIP

*PULSOUT, pin, time*  
PULSOUT will toggle  
the specified output for  
the defined time in milli-  
seconds.

PBASIC2 makes it even easier with an instruction named SHIFTIN.

*SHIFTIN dpin, cpin, mode,[results\bits]*

Where:

Dpin: (0-15) The BS2 number of the data pin.

Cpin: (0-15) The BS2 number of clock pin.

Mode: (0 - 3)

0: MSBPRES - MSB is first bit to come in, sample before the clock is pulsed.

1: LSBPRE - LSB first bit to come in, sample before the clock is pulsed.

2: MSBPOST - MSB is first bit to come in, sample after the clock is pulsed.

3: LSBPOST - LSB first bit to come in, sample after the clock is pulsed.

Result is the name of a variable to store the data.

bits is the number of bits to collect.

The following program is included in the Activity Board program files from Parallax. It has been slightly altered to display the input voltage in tenths of a volt in the debug window.

**'PROG D-4**

**'Use SHIFTIN command to read the ADC0831 serial ADC**

<b>ADres</b>	<b>var</b>	<b>byte</b>	<b>' A/D result (8 bits)</b>
<b>ADcs</b>	<b>con</b>	<b>12</b>	<b>' A/D enable (low true)</b>
<b>ADdat</b>	<b>con</b>	<b>14</b>	<b>' A/D data line</b>
<b>ADclk</b>	<b>con</b>	<b>15</b>	<b>' A/D clock</b>
 <b>START:</b>			
<b>LOW ADcs</b>			<b>' Enable ADC</b>
	<b>SHIFTIN ADdat,ADclk,msbpost,[ADres\9]</b>		<b>' Shift in the data</b>
<b>HIGH ADcs</b>			<b>' Disable ADC</b>
<b>DEBUG "Byte =", DEC ADres</b>			<b>' Display the result</b>
<b>DEBUG " Tenths of Volt = ", DEC ADres * 50/26 ,CR</b>			<b>' Display in tenths of volts</b>
<b>PAUSE 100</b>			<b>' Wait a 0.1 Seconds</b>
<b>GOTO START</b>			<b>' Repeat forever</b>

### ***Resistive Devices and RCTIME***

The BS2 has another means to measure an analog input: RCTIME. RC time uses an RC network and measures the time to charge or discharge a capacitor. Using an output to place 5V discharges the capacitor. When RCTIME is executed, the output is switched to an input and the voltage held by the capacitor is sensed. For the BS2 a voltage above 1.5V is considered a HIGH (this is the *threshold voltage*). As the BS2 output switches to an input the capacitor begins to charge through the resistor. The larger the resistance, the longer the charge time (see Reference A for a longer discussion).

This time to discharge down to 1.5V is calculated and returned to the program as a word variable. The activity board has a potentiometer (pot) which acts as a variable resistance. By using the RCTIME function and varying the pot we can collect analog values.

*RCTIME pin, state, resultvariable*

Pin: (0-15) Is the pin number of the BS2

State: (1 or 0) State to be sensed for when discharging.

ResultVariable: The word variable to store time result (each increment = 2μS, or millionths of a second, to charge).

Let's try a program that will read the pot, display the data, and play a frequency based on it.

**'PROG\_D-5**

**'Measure RC time of the potentiometer to adjust speaker frequency.**

**RCdata var word**

**'Variable for the data**

**RCpin con 7**

**'Define the input pin**

**Spkpin con 11**

**'Define speaker pin**

**Loop:**

**HIGH RCpin**

**'Charge cap for 10 ms**

**PAUSE 10**

**RCTIME RCpin, 1, RCdata**

**'Perform discharge and measure**

**DEBUG ? RCdata**

**'Display results**

**FREQOUT Spkpin, 50, RCdata**

**'Sound speaker using data.**

**GOTO Loop**

### Try it!

Change the program to reverse the frequency in relationship to the potentiometer (CCW = lower vice higher).

Of course, a potentiometer is only one device that can be used as a variable resistance source. Other ones include thermistors that vary resistance with temperature and photoresistors that detect varying light.

## Scaling Inputs

Evaluating resistance is the fundamental principle of measurement in many analog applications. There is a relationship between the range of measurement and the change in resistance. With the RCTIME instruction, it can be seen that the count value is a function of the range being measured. To relate the count value to what is actually being measured, scaling must be employed.

As an example, let's consider the Activity Board's potentiometer using in program D-5. From its full counterclockwise position, a clockwise rotation of 270 degrees returned count values from 1 to around 5300. Assuming the potentiometer is linear, each count represents approximately .0509 degrees of movement. In this application, the scaler is .0509 degrees/count.

Scaler = Range of measure/Range of Input.

Scaler = 270/5300 = .0509

Using the scaler and rearranging the formula, the angular position of the potentiometer knob can now be assessed at any returned count value.

Angle = RCdata x .0509

Try the program again and pay attention to the position of the knob and the returned value. For our example, a count of 1790 resulted in a movement of  $90^\circ$  ( $1790 * .0509 = 90$ ).

In programming, it is often best to convert and store the count in terms of the units being measured. The programmer must understand the mathematical operations and limits of the microcontroller. In order to get the BS2 to give us the results we want:

- Decimal points cannot be used.
- Division returns truncated quotients ( $10/3 = 3$ )
- No final (if a word variable) or interim calculation can result in a value  $> 65535$ .
- Operators in formulas are performed from left to right regardless of function.

Consider how the following formula can maximize accuracy while adhering to the constraints of PBASIC2:

$$\text{Degrees} = \text{RCData}/10 * 27 / 53$$

Since RCData can go as high as 5300,  $5300 \times 270 = 1431000$ . By dividing both RCData and 270 by 10 (a total of dividing by 100), we will never exceed 65535. The divisor is reduced by 100 ( $10 \times 10$ ) so that our calculation will be correct.

Modifying the last program to show degrees:

```
'PROG_D-6
'Display potentiometer position in degrees.
RCdata var word           'Variable for the data
RCpin con 7               'Define the input pin
Spkpin con 11             'Define speaker pin

Loop:
  HIGH RCpin              'Charge cap for 10 ms
  PAUSE 10
  RCTIME RCpin, 1, RCdata
  DEBUG DEC RCdata/10 * 27 / 53," Degrees",CR  'Perform discharge and measure
  FREQOUT Spkpin, 50, RCdata                  'Scale and display results
  GOTO Loop                                   'Sound speaker using data.
```

## Section E: Process Control

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Discuss methods of process-control.
- 2) Discuss advantages and disadvantages of the different process-control methods.
- 3) Write PBASIC2 code to perform simple process-control.

### Overview

Process Control is simply the act of measuring inputs and taking action based on the inputs. The type of reaction that takes place upon evaluation of the inputs defines the *process control mode*. Five common process control modes are ON/OFF, Differential Gap, Proportional, Integral, and Derivative.

The fundamental characteristic that distinguishes these control modes are listed below.

<i>Process Control Mode</i>	<i>Evaluation</i>	<i>Action</i>
ON/OFF	Is the variable above or below a specific desired value?	Drive the output Fully ON or Fully OFF
Differential Gap	Is the variable outside of an allowable range defined by an upper and lower limit?	Output is turned fully ON and OFF to drive the measured value through the range.
Proportional	How far is the measured variable away from the desired value?	Take a degree of action relative to the magnitude of the error.
Integral	Does an error still persist?	Continue taking more forceful action for the duration that the error exists.
Derivative	How fast is the error occurring?	Take action base on the rate at which the error is occurring.

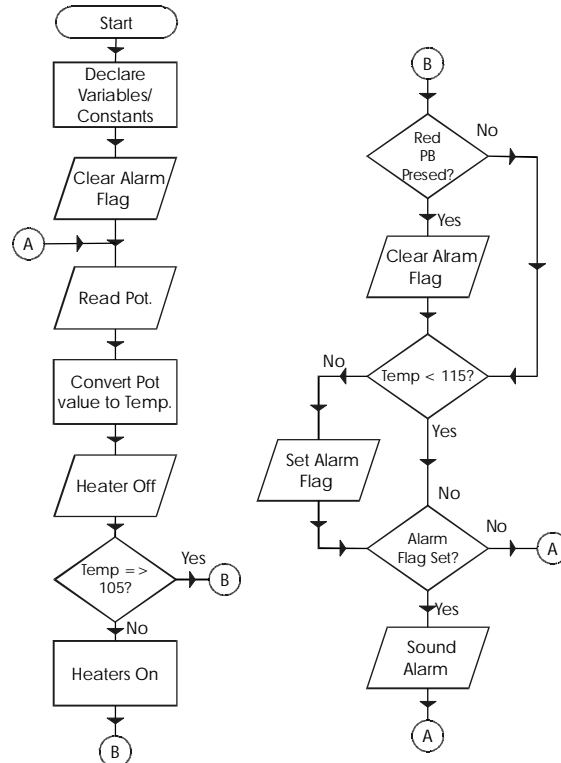
Microcontrollers have had an extraordinary impact on the area of embedded process control. Programmability, low cost, and small size makes the microcontroller a natural choice for processes requiring the simplest ON/OFF control to those that may demand sophisticated evaluation of multiple-input data and time-critical response. The wide range of application demands has spawned the development of microcontroller families with just as wide of a range of features. Faced with a process control application, selecting a controller with appropriate features and programming the proper mode of response can very interesting. When forced to use a particular microcontroller, a designer may have to rely on creativity as its features may point to its limitations. Whether selecting the right controller for the job or creatively performing a task in spite of a controllers limitations requires you to be familiar with fundamental process control modes. This section overviews the five fundamental process control modes and, given the features (and limitations) of the BS2, attempts to demonstrate some of these modes.

We will look at each of these from the standpoint of controlling an egg incubator to contrast and compare the process control modes. In this incubator scenario, we will assume that temperature must be maintained between  $100^{\circ}$  -  $110^{\circ}$ . To maintain the application the incubator must have an ability to measure temperature, evaluate this measurement relative to a desired setpoint, and initiate appropriate action to a heating element.

In order to simulate the incubator system with the Activity Board, we will assume that the potentiometer represents a temperature sensor. Furthermore, using the RCTIME command, it returns a range of 1 to 5000 counts representing measured temperatures of  $80^{\circ}$  -  $130^{\circ}$ . The program will employ scaling and referencing to whole degrees ( $100 \text{ counts} = 1^{\circ}$ ). The blue button's LED will act as the heater indicator. If it is lit, the heater is energized. Our incubator will also have an alarm setpoint of  $115^{\circ}$ , indicated by the red button's LED, which will latch on once energized until the red button is pressed.

### ON-OFF Control

ON-OFF control is the simplest of the control modes. Full output action is taken based on whether the measured value is above or below the desired value. As a result, the output action drives the measurement back toward the setpoint. As the measured value passes the setpoint the error polarity changes and the output is driven fully to the opposite direction. The process cycles back and forth past the setpoint based on this action. Let's look at our incubator. It needs to have a temperature maintained between  $100^{\circ}$  -  $110^{\circ}$ . To get a median value, we will use a setpoint of  $105^{\circ}$ . If temperature falls below  $105^{\circ}$  the heater will turn on. At or above  $105^{\circ}$  the heater will shut off. Our alarm will sound at or above  $115^{\circ}$ . The following flowchart and Program E-1 demonstrates the process.



**'PROG\_E-1****'ON-OFF control of incubator. Setpoint of 105°.****'Pot adjust temperature sensed by program.****'Blue's LED is indicates heat on, Red's indicates alarm (>115°). Press Red to reset.****RCin var word****'Variable to hold RC time****Temp var byte****'RC time is spanned to ~80-130 and stored here****AlarmFlag var Bit****'One bit to indicate alarm condition. 0 = no alarm****Pot con 7****'Define Pot I/O pin****AlarmFlag = 0****'Clear Alarm flag****Loop:****HIGH pot****'Charge pot capacitor for 10 mS****PAUSE 10****RCTIME Pot, 1, Rcin****'Measure discharge time****Temp = Rcin / 106 + 80****'Convert to temp****DEBUG CR, "Temperature is: ",DEC Temp****HIGH 8****'Turn off heaters for an instant****IF Temp => 105 THEN AlarmOff****'If at or above setpoint do not turn on heaters****LOW 8****'If below turn ON heaters****DEBUG ": Heater is ON "****'Display heaters are on****AlarmOff:****'If RED PB is pressed, reset alarm****INPUT 11****'Set up P11 for input****PAUSE 10****'Allow 10mS to stabilize****IF IN11 = 1 THEN AlarmOn****'Red not pressed? Skip next.****AlarmFlag = 0****'Red pressed, clear alarm flag.****AlarmOn****'Check to see if Alarm needs to be set****IF Temp < 115 THEN AlarmSound****'Less the alarm setpoint? Skip.****AlarmFlag = 1****'or else set the alarms flag.****AlarmSound:****'Check if we need to buzz alarm****If AlarmFlag = 0 THEN Loop****'Alarm flag not set? Skip and start over****FREQOUT 11, 50, 2000****'Or else sound speaker****DEBUG " \*\*\*\* ALARM \*\*\*\*"****'Display condition****Goto Loop**

ON/OFF control is suitable for processes that have large capacity, sluggish response, and a relatively constant level of disturbance. If our incubator was large, well insulated, and kept in a constant room environment, ON/OFF control may be acceptable. The major problem with ON/OFF control is that the output drive may cycle rapidly as the measurement hovers about the setpoint. As the incubator in our example approaches 105°, any noise riding on the measured value would be interpreted as rapid fluctuations in temperature. The microcontrollers evaluation would rapidly cycle the heating element On and Off during this period. In the incubator, rapidly switching the heating element may cause annoying RF interference. This rapid cycling could be damaging to electromechanical output elements such as motors, relays, and solenoids.

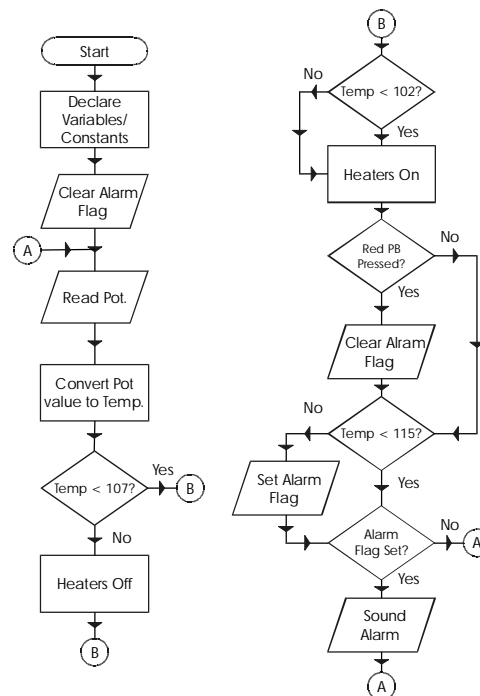
***Differential-Gap Control***

Differential-gap control is similar to ON/OFF control in that only full output action is taken. Differential gap control does not take this action based on a single desired setpoint. Instead, it



defines an upper and lower tolerable limit. When the measured value goes beyond a limit, full appropriate action is taken and maintained until the measured value is driven to the opposite limit. Full opposite action is then taken to drive the process back to the other limit. As you see, the process is cycled between the upper and lower limit. The rate of this cycling is determined by the overall speed of response of the process. If the process is large and sluggish, and will allow for range of measurement, the Differential-gap mode of control is appropriate.

To demonstrate differential gap control in our example, let's define a desired setpoint of  $105^{\circ} \pm 2^{\circ}$ . Therefore, the heater will energize when temperature drops to  $103^{\circ}$  and turn off when temperature exceeds  $107^{\circ}$ . Let's take look at our modified process.



#### 'PROG\_E-2

'Differential Gap control of incubator. ON at 102 decreasing, OFF at 108 increasing.

'Pot adjust temperature sensed by program.

'Blue's LED is indicates heat on, Red's indicates alarm (>115). Press Red to reset.

RCin var word

Temp var byte

AlarmFlag var Bit

Pot con 7

'Variable to hold RC time

'RC time is spanned to ~80-130 and stored here

'One bit to indicate alarm condition. 0 = no alarm

'Define Pot I/O pin

AlarmFlag = 0

'Clear Alarm flag

Loop:

HIGH pot: Pause 10

RCTIME Pot, 1, RCin: Temp = RCin/106 + 80 'Measure and convert

Debug CR, "Temperature is: ",DEC Temp 'Display

If Temp < 107 then Heaton

HIGH 8

'<107 ? Skip turning heat off

'or else turn heat off

Heaton: IF Temp > 102 then Indicate

'>102? Skip turning heat on

#### PBASIC TIP

Colons can be used to separate commands, instead of having one per line.

<b>LOW 8</b>	'else turn it on
<b>Indicate: If IN8 = 1 then AlarmOff</b>	'If heat is Off, skip display
<b>DEBUG ": Heater is ON "</b>	'else display it
<b>AlarmOff:Input 11:Pause 10</b>	'Check for alarm reset
<b>If IN11 = 1 then AlarmOn</b>	'If not pressed, leave it on
<b>AlarmFlag = 0</b>	'else clear it if
<b>AlarmOn: If Temp &lt; 115 then AlarmSound</b>	'Less than alarm temp? don't set
<b>AlarmFlag = 1</b>	'else set it
<b>AlarmSound: If AlarmFlag = 0 then Loop</b>	'Alarm not set? Don't sound, repeat loop
<b>Freqout 11, 50, 2000: Debug " **** ALARM ****"</b>	'else sound & display.
<b>Goto Loop</b>	

Both ON/OFF and Differential Gap control respond in a fully ON or fully OFF output action based on a comparison of measurement to a setpoint. It is the nature of these control methods for the measured variable to cycle above and below the setpoints. Time delays involved in the sensor's response to temperature change along with the delays associated with the heater warming-up and cooling-down cause overshoot. If the magnitude of the overshoot is tolerable, Differential Gap control is the method of choice due to its simplicity.

### ***Proportional Control Mode***

In proportional control, the drive to the output element is variable. The level of drive is varied proportionally to the magnitude of difference between the desired setpoint and the measured value. Unlike the continuous cycling inherent in the previous control modes, proportional control attempts to maintain a stable condition by applying the exact amount of drive required to account for the continuous system losses.

In designing our incubator system for proportional control, we need to assess what is assumed to be the average operating conditions. Lets assume that the incubator is operated indoors in a room that is held at an average 75°. Further more, given the size and insulative characteristics of the casing, we could determine that it requires a continuous 5000 BTU of heat input to maintain a 105° interior temperature. Given this assessment, it would make sense that if we incorporate a 10,000 BTU heating element it could be driven at 50% and therefore add an amount of heat exactly equal to the amount being lost. The result is a constant internal temperature equal to the setpoint.

This initial amount of drive is called the *bias* in a proportional control system. The *bias* is the amount of drive necessary to meet the average process disturbances.

If the bias is correct and the process disturbances are exactly equal to the assessed averages the measured temperature would be steady and equal to the setpoint. What occurs to the system when room temperature is set back to 65° at night or is allowed to approach 80° the heat of the day? Obviously if room temperature is not exactly 75° the bias will be either a little too low or a little too high and some degree of error will exist. Proportional control will evaluate the magnitude of this error and appropriately add to or delete from the bias in an attempt to stay near the setpoint.

#### **'Tron Tip**

The difference between the setpoint and the measured value is termed *error*.  
Error= setpoint - measured

How much the drive changes in relation to the magnitude of error is dependent on several system-related factors. Too forceful of action could set the system into cycling as we saw in ON/Off control. Too little action would allow the system to drift well away from the desired setpoint. Entire textbooks and courses are dedicated to the science of process control tuning.

For our purpose it suffices to understand that system *gain* is a function of the percentage of output changed divided by the percentage of allowable error. In our example the drive can be adjusted from 0 to 100% and the maximum allowable error is defined as 100 - 110 degrees. The gain of our system is 10% drive per 1° error. The proportional control has a set bias of 50% and is adjusted so that if actual temperature reaches 100 degrees the heater will be fully on, or 100% drive. If the temperature reached 110 degrees, the heater would be fully off, or 0% drive. For every 1 degree error, the drive would change by 10%. The equation for this would look like:

$$\%Drive = 50\% + (Error \times 10\%)$$

If the temperature was 103°F:

$$Error = 105 - 103 = 2$$

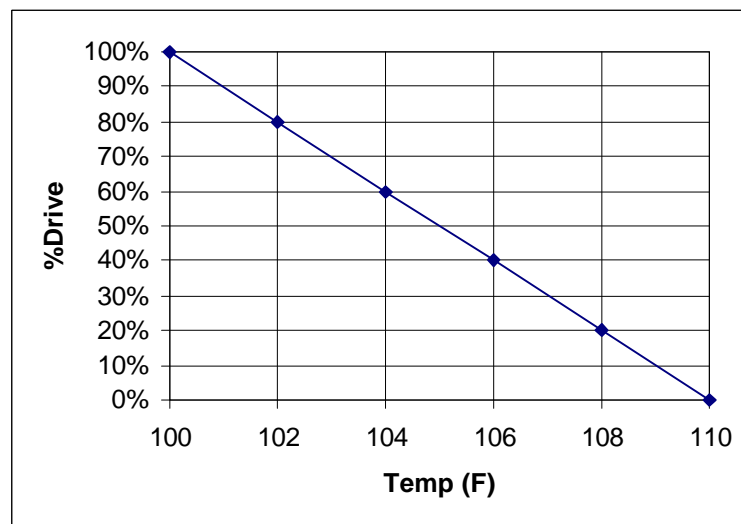
$$\%Drive = 50\% + (2 \times 10\%) = 70\%$$

If the temperature was 108°F:

$$Error = 105 - 108 = -3$$

$$\%Drive = 50\% + (-3 \times 10\%) = 20\%$$

Figure E-1 is a plot of Temperature vs. %Drive for our incubator.



**Figure E-1: Temperature vs. %Drive**

Lets look at a program that will control our simulated incubator using Proportional control. For visual data, the %drive will be shown by how long the blue button's LED is on over a 1 second period. At either end, 0% drive and 100% drive, the LED will flicker due to constraints of PBASIC. Also, without adding more code, the maximum error is limited to 5 due to PBASIC calculation limitations.

**'PROG\_E-3****'Proportional Control of Incubator, 100F = Fully OFF, 110F = Fully ON****'For visual display, the %Drive will be based on how long Blue's LED is on over 1 sec.****'Pot adjusts temperature sensed by program. Red's LED indicates alarm (>115).****'Press Red to reset alarm****Error var nib****'Stores the error, 0-5****Drive var word****'Stores %Drive, 0-100%****TimeOn var word****'Time On based on drive over 1 second.****RCin var word****'Variable to hold RC time****Temp var byte****'RC time is spanned to ~80-130 and stored here****AlarmFlag var Bit****'One bit to indicate alarm condition. 0 = no alarm****Pot con 7****'Define Pot I/O pin****AlarmFlag = 0****'Clear Alarm flag****Loop:****HIGH pot: Pause 10****'Charge cap for 10 mS****RCTIME Pot, 1, RCin: Temp = RCin/106 + 80****'Measure and convert****Debug CR, "Temperature is: ",DEC Temp****'Display****IF Temp < 105 then BelowSetpoint****'To avoid neg numbers, used 2 calc's.****Error = Temp - 105 MAX 5****'Calculate Error if below setpoint****Drive = -Error \* 10 + 50: GOTO Control****'Calculate %Drive based on Error****BelowSetpoint:****Error = 105 - Temp MAX 5****'Calculate Error if Above setpoint****Drive = Error \* 10 + 50****'Calculate %Drive Based on Error****Control:****DEBUG ": Error = ",DEC Error****'Display Error****DEBUG ": %Drive = ", DEC Drive****'Display Drive****TimeOn = Drive \* 10****'Time Heater on based on error****DEBUG ": Time On = ", DEC TimeOn****'Display Time on****LOW 8****'Turn on heater****PAUSE TimeOn****'Wait for calculated time****HIGH 8****'Turn off heater****PAUSE 1000-TimeOn****'Wait for the rest of the second****Input 11****'Check for alarm reset****If IN11 = 1 then AlarmOn****'If not pressed, leave it on****AlarmFlag = 0****'else clear it if****AlarmOn: If Temp < 115 then AlarmSound****'Less than alarm temp? don't set****AlarmFlag = 1****'else set it****AlarmSound: If AlarmFlag = 0 then Loop****'Alarm not set? Don't sound, repeat loop****Freqout 11, 50, 2000: Debug " \*\*\*\* ALARM \*\*\*\*\*" 'else sound & display.****Goto Loop**

At 105 degrees (as adjusted by the potentiometer), blue's LED is lit about 1/2 of a second out of 1 second, or 50%. As the temperature is decreased the LED is lit for a longer percentage of the 1 second. As it rises above 105, the LED is lit a shorter percentage.

We could easily change the program to use the PWM command and drive the Aout output. At 105°F, Aout should read around 2.5 volts. At 100°F, Aout would be near 5 volts. At 110°F Aout would be near 0V.

The code for the Control section of program E-3 would be as follows:

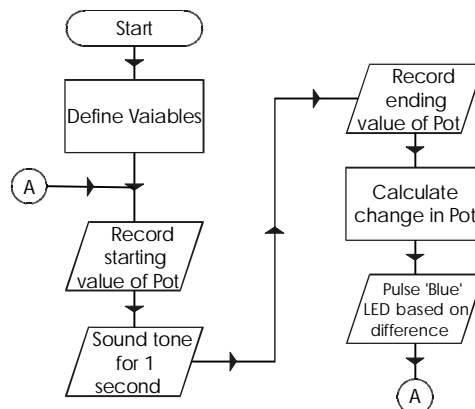
```
'Prog_E-3b (E-3 modified)
.
.
.
Control:
  DEBUG ": Error = ",DEC Error          'Display Error
  DEBUG ": %Drive = ", DEC Drive        'Display Drive
  PWM 12, 255/100 * Drive,50            'PWM = %drive of full (255) for 50mS

INPUT 11
.
.
.
```

### ***Derivative & Integral Control***

While we will not go through a lengthy program of Derivative and Integral control, they are very much worth a short discussion and example. Derivative control adjusts the drive based on the *rate of change* of the error. For our example, the faster the temperature drops, the greater the amount of drive. Or conversely, the faster it rises the more the drive will be reduced. Derivative control by itself is not useful because output drive is only determined by the rate of error signal. But when combined with Proportional control the combination can be effective at quickly compensating for a disturbance. This is known as Proportional plus Derivative control (PD control). PD control mode is effective for small capacity, rapidly changing processes where disturbances happen quickly and at high magnitudes. A rapidly decreasing temperature in our incubator would indicate that the door may have been opened. Derivative control would drive the heater hard in an effort to compensate for the anticipated further drop in temperature. As a programmer, you must develop a method to evaluate the derivative or rate of change of the error signal. This rate is then taken into account in your control evaluation.

Consider the following short program:



**'PROG\_E-4**

**'Derivative Control Example**

**'This program will Sound a short high tone (READY) followed by a long lone tone(GO).**

**'The amount the pot moves DURING the 2nd tone will determine how long**

**'blue's LED stays lit.**

**StartVal var word**

**StopVal var word**

**Change var word**

**HIGH 8**

**'Store pot beginning value**

**'Store pot ending value**

**'Holds calculated change**

**'Turn off Blue's LED**

**Loop:**

**FREQOUT 11, 50, 3000**

**PAUSE 200**

**HIGH 7: PAUSE 10: RCTIME 7, 1, StartVal**

**FREQOUT 11, 300, 1000**

**HIGH 7: Pause 10: RCTime 7, 1, StopVal**

**'Sound READY short high tone**

**'Charge RC and get pot value**

**'Sound GO measuring low long tone**

**'Get Pot's end value**

**If StopVal > StartVal THEN Switch**

**Change = StartVal - StopVal**

**Goto Pulselt**

**'Make sure we don't go negative**

**'Calculate change**

**Switch:**

**Change = StopVal - StartVal**

**'Calculate Change if other direction**

**Pulselt:**

**debug "Start ",DEC StartVal," : End",DEC StopVal, ": Change",DEC Change,cr**

**LOW 8: PAUSE Change: HIGH 8**

**PAUSE 3000**

**GOTO Loop**

**'Display data**

**'Light LED for amount of change time**

**'Take a 3 second breather**

**'Start over!**

In Program E-4, the quicker the pot is moved during the longer low tone, the longer blue's LED stays lit. See, you are taking more forceful action based on the rate of change of the measured value.

Understanding the purpose of implementing the Integral control mode takes us back to an issue addressed earlier. That is the issue of determining the correct bias or initial drive necessary to meet the process' average disturbances. In our incubator, our assessment of average disturbances was a 75° room temperature. It was determined that a 50% bias would be exactly correct for this condition. And, at a room temperature of 75° there would be no error. In our assessment, the 75 degrees reflected the average temperature over a 24-hour period. As the actual room temperature drifts to 65° in the middle of the night to 80° during the day, it may be more appropriate to slowly adjust the bias up and down accordingly to meet these gradual drifts in the process disturbance level. This is exactly what Integral control action does. The error is periodically checked and the bias “bumped” up or down accordingly until the error is driven completely away.

Proportional control would attempt to compensate for a disturbance. But, due to its fundamental nature, it could not drive the error back to zero. Consider the action in our incubator at the night time low of 65° if only proportional control is implemented. When room temperature drops to 65° the incubator cools due to the increased losses. A 50% drive would be insufficient to keep up with the new losses. In order to have a higher drive level, some error MUST exist. If the error

went to 0 (105°F), the drive would be back at 50%, but this is insufficient so temperature would decrease, giving an error, increasing drive.... See the circle?

With only proportional control, the incubator stabilizes at a drive level higher than 50% but still at a temperature below the desired setpoint. For our system, let's say it stabilizes at 60% drive. 10% of the drive value is due to error and 50% drive due the original bias level. But we really don't like operating with a continual error. Integral control, over relatively long periods of time (minutes, hours, or even days), will compensate for the drive due to error. Eventually the 60% of drive will be due to 50% of bias and 10% of Integral and 0% error. Integral mode compliments proportional action by “coasting” the measured value to being exacting equal to the desired setpoint. By controlling the rate at which integral evaluation is performed and the forcefulness of action taken, the error can be driven away quickly with little or no overshoot.

Many precision controls are a combination of Proportional, Integral and Derivative control, known as PID control.

## Section F: Hexadecimal & BS2 Memory

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Explain the reason for working in the hexadecimal number system.
- 2) Convert between binary and hexadecimal number systems.
- 3) Explain the areas of the BS2 Memory Map and their contents.
- 4) Discuss the function and use of the BS2 registers.
- 5) Discuss the use of the BS2 EEPROM.
- 6) Program in PBASIC2 to read and write to EEPROM memory locations.

### Hexadecimal

The binary number system was introduced in Section C. It is the number system utilized in digital systems to hold values because they only have 1's and 0's to work with. In programming, addressing, and viewing contents of a digital system we often need to work in this number system or one directly corresponding to it. It can become confusing to always work in binary (was that number  $10110100_2$  or  $10010100_2$ ?), and conversion to decimal does not always convey the meaning of the number clearly enough ( $1110_2$  sets certain bits we can readily see, seeing 14 just doesn't mean as much). Conversion can often be too complex for a simple digital system to perform.

Enter the hexadecimal (hex) number system. Digital systems often provide output as hex and accept input in hex. Let's look at why. Just as decimal is base 10, and binary base 2, hexadecimal is a base 16 number system. Each position to the left is a higher power of 16. And as in decimal where there are 10 unique digits (0-9), in hex we need 16 unique digits. The first 10 are from our decimal system (0-9) with the remaining 6 taken from our alphabet, A - F, to represent the decimal numbers 10 - 15. To count from 0 to 17 in hex would be as follows:

$0_{16}$   $1_{16}$   $2_{16}$   $3_{16}$   $4_{16}$   $5_{16}$   $6_{16}$   $7_{16}$   $8_{16}$   $9_{16}$   $A_{16}$   $B_{16}$   $C_{16}$   $D_{16}$   $E_{16}$   $F_{16}$   $10_{16}$   $11_{16}$

Hexadecimals are typically denoted with a subscripted 16, such as  $F_{16}$ . In programming a '\$' is commonly used to denote a hex number ( $\$FF = FF_{16}$ ). Let's look at a weighted representation of the number  $C3A9_{16}$ .

Place	$4^{th}$	$3^{rd}$	$2^{nd}$	$1^{st}$
Weight	$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
Number	$C_{16}$ (12)	$3_{16}$	$A_{16}$ (10)	$9_{16}$
Result = Number x Weight	49152	786	160	9

**Table F-1: Hexadecimal Weighting**

Converting  $C3A9_{16}$  to decimal results in a 50089 ( $49152 + 786 + 160 + 9$ ). Table F-2 is a comparison of the decimal, binary and hexadecimal numbers 0 - 15. As you can see, for a group of 4 bits (a nibble) there is a unique hexadecimal number. For a larger grouping of binary numbers, each consecutive nibble is represented by a hexadecimal number. Let's try a conversion and check our results.



$$10110100_2 = 180_{10}$$

$$0100_2 = 4_{16} \text{ (Low nibble)}$$

$$1011_2 = B_{16} \text{ (High nibble)}$$

$$B4_{16} = (11 \times 16) + (4 \times 1) = 180_{10}$$

**'TRON TIP**

If you've watched a PC boot up, you may have noticed a COM address such as 03F8. Even sophisticated digital systems, like home computers, often use hex.

So a large binary word such as  $1010011001011111_2$  would be represented in hexadecimal as:  $A65F_{16}$ . Much easier to read, isn't it? And with a little practice you will be able to readily see that the nibble represented by a number such as 6 is  $0110_2$  and know exactly which bits are ON and OFF.

Decimal	Binary	Hexadecimal
0	$0000_2$	$0_{16}$
1	$0001_2$	$1_{16}$
2	$0010_2$	$2_{16}$
3	$0011_2$	$3_{16}$
4	$0100_2$	$4_{16}$
5	$0101_2$	$5_{16}$
6	$0110_2$	$6_{16}$
7	$0111_2$	$7_{16}$
8	$1000_2$	$8_{16}$
9	$1001_2$	$9_{16}$
10	$1010_2$	$A_{16}$
11	$1011_2$	$B_{16}$
12	$1100_2$	$C_{16}$
13	$1101_2$	$D_{16}$
14	$1110_2$	$E_{16}$
15	$1111_2$	$F_{16}$

**Table F-2: Decimal-Binary-Hexadecimal Comparison**

Program C-4 accepted button presses, individually or in combinations, and displayed the binary and decimal equivalents. Let's modify it to also show the hexadecimal value.

**'PROG\_F-1**

**'Read buttons as a nibble and display as bin, hex and dec.**

**'(Reversed so Pressed = '1')**

**NibIn var nib**  
**DIRC = %0000**

**'Define a nibble variable**  
**'Set P8 - P11 as inputs**

**Loop:**

**NibIn = INC ^ %1111** **'Read input, and reverse it**  
**Debug IBIN4 NibIn, "** **", IHEX NibIn, " ", DEC NibIn, CR**  
**GOTO LOOP** **'Repeat**

**PBASIC TIP**

Just as we display decimals using DEC and binary with IBIN, the IHEX command will display numbers in hexadecimal. An optional number can be used to specify the number of places to pad (IHEX4 3F → \$003F).

## **BS2 Memory & Variable Storage**

As it was discussed in Section A, there are 32 bytes, or 16 words, of RAM available in programming the BS2. These words are known as *registers*. Three of these registers are used to address the direction, output and input of P0-P15. As we saw in Section C these words are called DIRS, OUTS and INS. This leaves 13 registers to use for variables in the PBASIC2 programs.

PBASIC2 takes care of the task of taking the variables we assign in the program and allocating them space in these 13 registers. Given code example such as the following, where a nibble and word are declared, PBASIC2 will allocate register space to hold the values that they represent.

```
NibIn var Nib
Pot var Word
```

PBASIC actually allocates the RAM by words, then bytes, then nibbles and finally bits in the order that they are declared in the program.

While PBASIC2 manages this RAM for us, we also have direct control over it. W0 - W12 are PBASIC commands to read or write directly to the registers 0 - 12. We can also address the registers as bytes using B0 - B26, where W0 is made up of B0 (high order) and B1 (low order), W1 is B2 and B3 and so on.

Let's look at a program that will declare various variables, assign them, and then we'll look at the PBASIC2 memory map.

```
'PROG_F_2
'Declares and sets variables for memory map testing.
'Consists of 2 words, 2 bytes and 4 nibbles and 3 bits
```

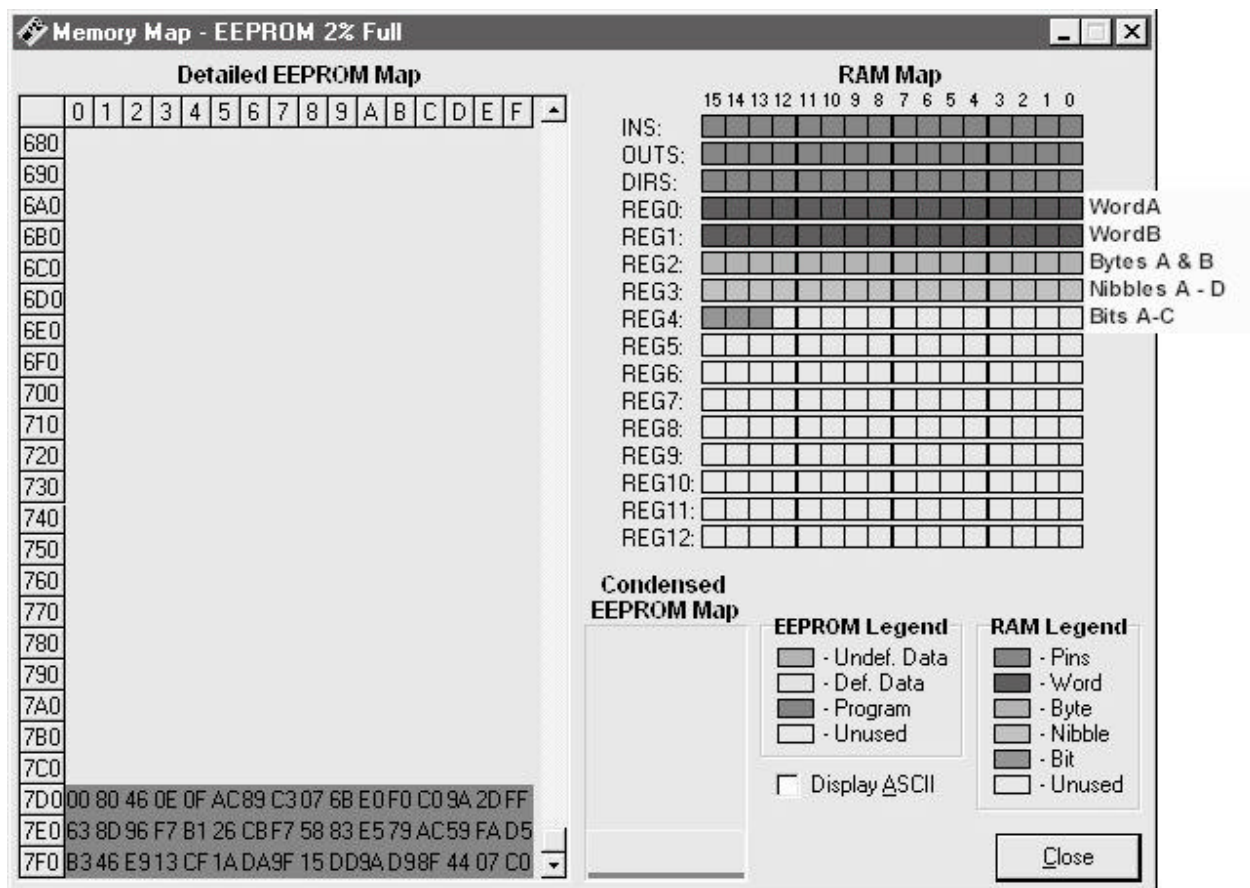
```
WordA var word      'Declare variables
ByteA var byte
NibbleA var nib
ByteB var byte
NibbleB var nib
BitA var bit
NibbleC var nib
NibbleD var nib
WordB var word
BitB var bit
BitC var bit
```

```
WordA = $0123      'Set values in variables.
WordB = $4567
ByteA = $89
ByteB = $AB
NibbleA = $C
NibbleB = $D
NibbleC = $E
NibbleD = $F
BitA = 1
BitB = 0
BitC = 1
```

The Memory Map, Figure F-1, can be displayed by hitting the memory map button icon (13<sup>th</sup> on button bar). It displays the following:

- A detailed map of the 2K EEPROM on the left is where our tokenized program is stored (the full 2K is too large to show at once).
- A condensed map of the 2K EEPROM in middle bottom.
- A RAM map showing utilization of RAM on right.
- Legends for each.

Figure F-1 is the memory map for Program F-2 is in gray scale and little hard to read. Labels have been added to clarify. Note that our 2 word variables have been defined first in REG 0 and 1, followed by REG 2 holding our 2 bytes, REG3 holds our 4 nibbles, and the left 3 positions of REG4 holds the 3 bits. DIRS, INS, and OUTS control P0 - P15 and are not used for variable space.



**Figure F-1 : Memory Map for PROG\_F-1**

Let's modify the program to directly read REG3 (W3) that holds our 4 nibbles containing values  $C_{16}$  -  $F_{16}$ . (to save space we will use colons to separate the commands).

```
'PROG_F_2b
'Declares and sets variables for memory map testing.
'Consists of 2 words, 2 bytes and 4 nibbles and 3 bits

'Declare variables
WordA var word: ByteA var byte: NibbleA var nib: ByteB var byte
NibbleB var nib: BitA var bit: NibbleC var nib: NibbleD var nib: WordB var word
BitB var bit: BitC var bit

'Set values in variables.
WordA = $0123: WordB = $4567: ByteA = $89: ByteB = $AB: NibbleA = $C
NibbleB = $D: NibbleC = $E: NibbleD = $F: BitA = 1: BitB = 0: BitC = 1

'Read the data directly as a word
DEBUG IHEX ? W3
```

Running this program produces the result:

**W3 = \$FEDC**

Why not \$CDEF? Well, NibbleA was assigned the first location in memory for nibbles, and NibbleD the last. This is backwards from the normal convention of highest order to the left, thus our word is assembled backwards.

We could also write directly to these locations. Let's modify our program to initialize the variables as usual, display WordA, change the low-order byte of it ( $B_1$ ) to  $FF_{16}$  and read it again.

```
'PROG_F_2c
'Declares and sets variables for memory map testing.
'Consists of 2 words, 2 bytes and 4 nibbles.
'Declare variables
WordA var word: ByteA var byte: NibbleA var nib: ByteB var byte
NibbleB var nib: NibbleC var nib: NibbleD var nib: WordB var word

'Set values in variables.
WordA = $0123: WordB = $4567: ByteA = $89: ByteB = $AB
NibbleA = $C: NibbleB = $D: NibbleC = $E: NibbleD = $F

'Display WordA, change it, Display again.
DEBUG IHEX4 ? WordA
B0 = $FF
DEBUG IHEX4 ? WordA
'Display contents of IHEX4
'Directly change the low order byte in WordA
'Display again
```

The program returns the following results:

**WordA = \$0123**

**WordA = \$01FF**

PBASIC2 provides even greater flexibility. We can use a portion of a variable by selecting a section of it, such as the following examples:

.BIT5 -- 6th bit (of BIT0 - BIT7) of a word or byte.  
 .NIB2 -- 3<sup>rd</sup> nibble (of NIB0- NIB3) of a word.

See Reference A for a full list of *variable modifiers* (pg. 222 in version 1.9).

From any of our programs add the following at the end:

**DEBUG IHEX ? WordB.NIB2**

It would return the following:

WordB.NIB2 = \$5

This makes sense because WordA was set to  $4567_{16}$ , making the 3<sup>rd</sup> nibble equal to  $5_{16}$ .

While these bit manipulations may not seem very important now, they can be very powerful tools to a seasoned programmer where every bit of code and storage is important

### ***BS2 Memory Map and EEPROM Storage***

As we mentioned in Section A, the BS2 utilizes a 2K EEPROM to store the tokenized PBASIC2 programs. Look again at Figure F-1. The left side displays the 2K of EEPROM where the PBASIC2 programs are stored. If one could scroll up, it would be seen that the numbers running down the rows range from 000 to 7F0. This is hexadecimal of course. The numbers across the top of the columns range from 0 to F. The memory is shown arranged in groups of 16 bytes per row. Looking at the very last row of bytes, the leftmost byte containing  $B3_{16}$  would be in memory location  $7F0_{16}$ . The rightmost byte is  $7FF_{16}$  and contains  $C0_{16}$ . Converting  $7FF_{16}$  to decimal we get 2047, which makes sense because it is a 2K EEPROM with memory locations from 0 - 2047.

When we run a program from our PBASIC2 programming environment of stampw.exe, the program entered is tokenized and stored in this EEPROM memory. Using the memory map, the user can see the tokenized program in hexadecimal. Notice that it writes programs at the END of the memory. When the BS2 runs programs it reads these tokens from memory, interprets them into the native machine language of the microcontroller (PIC16C57 in this case) and carries out the instructions.

Remember that this memory is non-volatile. When we disconnect the BS2 from the power supply the program in memory remains intact (our data in RAM will be destroyed though). We can write and read data directly from this non-volatile memory using WRITE and READ commands! Let's say we were collecting data from a process and we would be crushed if the vital data was destroyed because the power went out for a second. If we store our data in EEPROM this would not be a problem. Also, having a possible 2048 bytes of EEPROM storage is a heck of a lot more than our 26 bytes of RAM. Our only caution is to not WRITE over our program!! The syntax for WRITE is:

*WRITE location, byte*

Where:

Location: the EEPROM memory location, 0 - 2047.

Byte: The byte value to be stored.

READ is:

*READ location, variable*

Where:

Location: the EEPROM memory location, 0 -2047.

Variable: The variable into which the byte in memory is stored.

Let's run a program that will store 'musical' notes and play them back. Here are some of the features of it:

- The Activity Board POT will adjust the frequency of the note.
- Press the green button to save the note, this button will repeat.
- Press the black button to play the song.
- Press the blue button to reset the tune recorded to the beginning.
- Once a tune is stored, upon loss and return of power, the song will still be in memory.
- The note is based on a pot setting, from 0 - ~5000, a word. To save it we have to store it as bytes. We use the .HIGHBYTE and .LOWBYTE to break it down.
- On playback, we have to know where the end is. We'll use a FF<sub>16</sub> to mark the end since this HIGHBYTE of FF (FF00<sub>16</sub>) and 65280 is well out of range of the possible maximum.
- To keep track of our tune in memory, byte variables will be used. Since each note takes 2 bytes, this limits the song to 127 notes.

#### **'PROG\_F-3**

**'This program will save a song in EEPROM memory. The note is based on POT position**

**'Grn saves note, black plays song, Blue resets song.**

**Location var byte      'Record memory location, max notes is 128 (max byte/2 for words).**

**PLocation var byte    'Play memory location variable**

**Note var word          'Holds note defined by RCTIME of POT**

**Check0 var byte        'Used to check if end of tune on play**

**'Define bytevariables for buttons**

**Grnbv var byte: Bluebv var byte: Blkbv var byte**

**'Assign pin numbers**

**Pot con 7: Spk con 11: GrnBtn con 10: BlkBtn con 9: BlueBtn con 8:**

**'Reset Byte Variables.**

**Grnbv = 0: Bluebv = 0: Blkbv =0**

**Location = 0**

**DEBUG "READY FOR NEW NOTES! - Turn POT to find a note.",CR**

**DEBUG "Press:",CR,"GREEN to store note",CR,"BLACK to play tune",CR,"BLUE to restart",CR**

**Loop:**

**HIGH Pot: PAUSE 200:**

**'Charge up cap**

**RCTIME Pot, 1, Note**

**'Measure it**

**FREQOUT spk, 200, Note**

**'Play it**

**BUTTON GrnBtn, 0, 1, 5, Grnbv, 1, StoreIt**

**'Debounce with repeat, store note**

**BUTTON BlkBtn, 0, 255, 0, Blkbv, 1, PlayIt**

**'Debounce, no repeat, play tune**

**BUTTON BlueBtn, 0, 255, 0, Bluebv, 1, ClearIt**

**'Debounce, no repeat, clear tune**

**Goto Loop**

```

StoreIt:
  WRITE Location, Note.HIGHBYTE           'Save the high byte of our note word in
                                           ' location
  WRITE Location + 1, Note.LOWBYTE        'Save the low byte of our note word in
                                           'location +1
  DEBUG "STORED note ", DEC location/2 + 1, " of 127 in ", IHEX3 Location, CR
  Location = Location + 2 MAX 254         'Increment Location counter by 2
  WRITE Location, $FF                     'Write a hex FF to indicate end of song
Goto Loop

PlayIt:
  DEBUG "***** PLAYING! *****", CR
  Pause 500: PLocation = 0                'Pause for effect, reset play location counter
Play_loop:
  READ PLocation, Check0                  'Read to ensure not end of song
  IF Check0 = $FF THEN EndPlay            'If it is, then end playing
  READ PLocation, Note.HIGHBYTE           'Read high byte and put in note word
  READ PLocation + 1, Note.LOWBYTE        'Read low byte and put in note word
  FREQOUT spk, 200, Note                  'play note for 200 ms
  PLocation = PLocation + 2               'increment play location
GOTO Play_loop                            'Play next note

EndPlay
  DEBUG "READY TO CONTINUE RECORDING Tune!", CR
  PAUSE 1000: GOTO Loop

ClearIt:
  DEBUG "TUNE CLEARED! ", CR
  Location = 0: WRITE Location, $FF        'Reset record location back to 0, mark as end
                                           'of tune.
GOTO Loop

```

Program F-3 should meet all of it's programming requirements. Once we store a tune in memory, even though power is lost and restored, the stored tune is still available! While the tune was limited to 127 notes, it could have been written to hold many more notes by using a word variable, but a limit would have been needed to ensure it did not overwrite the program in the EEPROM!

Our program uses a fair portion of our available program memory ( $5F8_{16} - 7FF_{16}$ ). Also note that even after recording a tune, there is no trace of it in the EEPROM memory map. The map does not read stored values, just what the stamp program WILL place into memory when downloaded.

**Data**

There is a command that we can use to write data directly to the EEPROM memory when the program is tokenized and downloaded. This is the DATA command.

dataname DATA values  
Or  
dataname Data @location, values

Where:

Dataname: Name of string to store.

Values: Data to be stored, in string or number format (DATA "HELLO" or DATA 72,69,76,76,79).

@location: Defines memory location to begin storing data (DATA @10,"HELLO"). If this is not specified, the data will begin to be stored at 000<sub>16</sub>. Consecutive DATA statements without location declarations will continue from where the last one finished. PBASIC2 will not allow you to place 2 data values in the same location.

This code will return an error since the location of "BYE" is told to overwrite part of the location of "HELLO".

**First DATA @\$10, "HELLO!"**  
**Second DATA @\$13, "BYE!"**

To retrieve the data, use the READ command, where the memory location to read is defined by the *dataname*. Lets look at a program that will store a tune in EEPROM and play it back. It will also show the name of the tune from data. Let's store two tunes, one being the CHARGE! and the other being a series of beeps. To make it simple, we will take our word-sized notes from Program B-6 (plays CHARGE!) and divide them by 10 and store the values as bytes. We will also divide the duration by 3 to be able to store these values as bytes.

This is the CHARGE theme:

<b>FREQOUT 11, 150, 1120</b>	<b>'Play notes</b>
<b>FREQOUT 11, 150, 1476</b>	
<b>FREQOUT 11, 150, 1856</b>	
<b>FREQOUT 11, 300,2204</b>	
<b>FREQOUT 11, 9, 255</b>	
<b>FREQOUT 11, 200,1856</b>	
<b>FREQOUT 11, 600, 2204</b>	

Dividing out our new durations and notes gives us:

50, 112  
50, 148  
50, 186  
100, 220  
3,26  
67,186  
200,220

Let's also store the name, and display it. The same routine will be used to play the two tunes, but by pointing to different memory location we can select the tune to play. Initially the variable



Pointer will point at the start of the song name. Once we loop through and display the name, the base will be incremented by  $10_{16}$  to point at the start of the song.

#### 'PROG\_F-4

'Saves 2 set songs in EEPROM memory, displays them and plays them

'By setting up the base memory location, we can play multiple tunes.

'Use Red and Green buttons to play.

```
ChargeName DATA "CHARGE!" 'Store name and notes at set locations
ChargeNotes DATA @$10, 50, 112, 50, 148, 50, 186, 100, 220, 3,26 ,67,186, 200,220
BeepName DATA @$30,"BEEPS!!" 'Store name and notes at set locations
BeepNotes DATA @$40, 100, 150, 90, 180, 80, 200, 70, 220, 60, 240
x var byte 'General Counting variable
Pointer var byte 'Holds base location for memory
Freq var byte 'Declare for frequency of note
duration var byte 'Declare for Duration of note
char var byte 'Declare to hold name character
```

```
INPUT 11: INPUT 10 'Set inputs on buttons
LOOP:
Pointer = ChargeName 'Set base memory location to CHARGE ($0) name
IF IN11 = 0 THEN PLAY 'Is Red pressed? Play song.
Pointer = BeepName 'Set base memory location to BEEPs ($30) name
IF IN10 = 0 THEN PLAY 'Is Green pressed? Play song.
GOTO LOOP:
```

```
Play: 'Display song name
debug CR
FOR x = 0 to 6
    READ Pointer + x, char 'Read a character
    DEBUG char 'Display it
NEXT
```

```
Pointer = Pointer + $10 'Increment base to notes location
FOR x = 0 to 15 Step 2 'Play the song
    READ Pointer + x,Duration 'Get duration from memory
    READ Pointer + x + 1, Freq 'Get Freq from memory
    FREQOUT 11, Duration * 3, Freq * 10 'Multiply out and play
NEXT
GOTO Loop
```

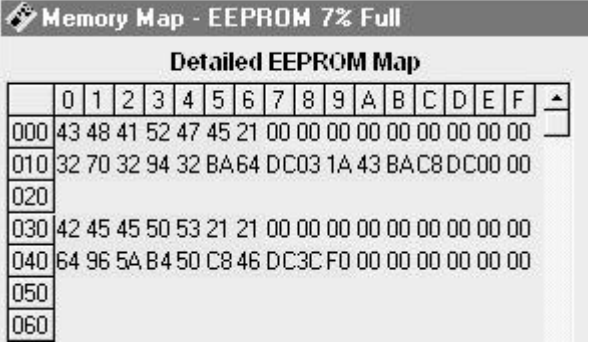
#### Try it!

Add a tune of your own to the program!

Many times it is necessary to store a number of text strings. If we wrote a debug statement every time we wanted to display a string (or send it to another device), it would require large amount of program memory. Since PBASIC2 has no feature to store text string in variables, this is a possible alternative.

Figure F-2 is the EEPROM area of the memory map. Note that there is data now at the beginning of the map from the DATA statements. Unless defined otherwise with @, data is written starting at  $000_{16}$ .

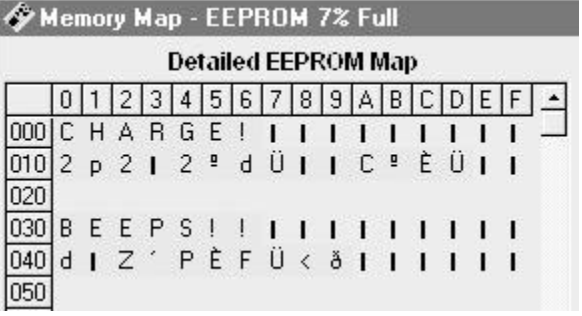
Compare the locations that data is stored to the definitions given in our data @ statements. To make it easier, Figure F-3 is the same memory map with the "Display ASCII" checkbox checked. Instead of showing the hexadecimal values, it shows the values in ASCII character code.



**Detailed EEPROM Map**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000	43	48	41	52	47	45	21	00	00	00	00	00	00	00	00	00	
010	32	70	32	94	32	BA	64	DC	03	1A	43	BA	C8	DC	00	00	
020																	
030	42	45	45	50	53	21	21	00	00	00	00	00	00	00	00	00	
040	64	96	5A	B4	50	C8	46	DC	3C	F0	00	00	00	00	00	00	
050																	
060																	

**Figure F-2: Memory Map Showing Data in Hexadecimal**



**Detailed EEPROM Map**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000	C	H	A	R	G	E	!										
010	2	p	2		2	"	d	Ü			C	"	È	Ü			
020																	
030	B	E	E	P	S	!	!										
040	d		Z	'	P	È	F	Ü	<	ø							
050																	

**Figure F-3: Memory Map Showing Data in ASCII**

Often times looking at memory in various forms of Hex, ASCII, and decimal (if possible) will reveal some indication to what is stored there and what it may be used for. From Figure F-2 convert the hexadecimal numbers in memory locations 010<sub>16</sub> - 01D<sub>16</sub> to decimal. Look familiar?



## Section G: Logical Operators and Signed Numbers

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Discuss how logic is used to make true/false decisions.
- 2) Write the truth tables for ANDs, ORs, NOTs and XORs.
- 3) Write PBASIC2 code utilizing logic operators.
- 4) Apply masks to perform bit operations.
- 5) Write PBASIC2 code to mask individual bits in binary numbers.
- 7) Convert numbers to 2's compliment signed numbers.
- 8) Subtract binary numbers using 2's compliment.

### Introduction to Logic

Logic is simply evaluating conditions as being true or false and making a decision or producing an output based on the evaluation. We do it every day. When stopped by a traffic signal, in most cases we will proceed *when the light turns green AND the coast is clear*. Our decision to proceed is based on both of these statements being true. In another instance at the traffic light we may wish to make a right hand turn. To proceed there are three conditions we might have to check for:

- The light is green.
- Right hand turns are allowed.
- Coast is clear.

Of course not all of these have to be true for us to make the turn. We would have *permission* to turn if either *the light is green OR right turns are allowed*. We still have to check for cars, so we need *permission AND the fact no cars are coming*. Let's throw in one more fact: How do we know we have permission to turn right on red? Well, the absence of a sign saying "No turn on red" is the common method. By most state laws there is permission to turn right on red unless posted! So we are not really checking if we have permission, we check to see if permission is rescinded by a posted sign. Logically we would say *if NOT a posted sign*.

Let's rewrite our rules and assign letters to them:

- The light is green (A).
- "No turn on red" sign posted (B).
- Coast is clear (C).

Finally we can make a statement that we have permission:

- We may proceed (D).

The permission D is a function of A B and C. So what is our statement? When the light is green (A) OR there is NOT a sign posted (B) AND no cars are coming (C) may we proceed (D). Let's represent it by just symbols:

$$D = [A \text{ OR } \text{NOT}(B)] \text{ AND } C$$

Why the brackets around A OR NOT(B) ? Either of these conditions has to be true, light green OR NOT a "no turn on red" sign, BEFORE we evaluate that the coast is clear (C). Just as in math we can group terms to ensure which functions takes priority.

When we say 'AND' we imply that both conditions have to be true before a statement is true. If either one is false, the statement will be false. 'OR' implies that either one or the other has to be true otherwise it will be false. True or False. Two states. In binary there are also only 2 states, 1 or 0. We can equate a true condition to a binary 1 and a false condition to a binary 0. Let's look a logical equation using binary:

$$C = 1 \text{ AND } 0$$

What would C be equal to? Well, as AND implies both terms have to be true for the AND equation to be true. Since in this equation there is a 0 (false), the result must be 0(false). There exists in digital systems four basic logic operators: AND, OR, NOT, XOR (Exclusive-OR). Each one evaluates expressions and returns values according to their function.

#### 'TRON TIP

These logic functions have equivalent electronic gates that are the building blocks of ALL digital circuits.

### AND

Both (or all) terms must be true before the expression is true.

The following is the *truth table* for an AND. A and B represent two terms, and the last column is the result of using the function (AND in this instance) to evaluate them.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

**Table G-1: 'AND' Truth Table**

Note that the only combination that will return a 1 (true) for C is when both A and B are 1 (true).

### OR

Either (or any) term must be true before the expression is true.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

**Table G-2: 'OR' Truth Table**

Note that any, or all, terms can be 1 to return a 1.

**NOT**

NOT simply makes a true statement false or a false statement true. This is known as *complementing* or *inverting*.

A	NOT A
0	1
1	0

**Table G-3: 'NOT' Truth Table****XOR**

XOR is similar to the OR, but if BOTH terms are true, the expression will be false. EITHER term must be true, BUT NOT BOTH, for the expression to be true.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

**Table G-4: 'XOR' Truth Table**

It is important to note something here. Look at the cases for the XOR when B is 1.

- When A is 0 and B is 1, the result is 1.
- When A is 1 and B is 1, the result is 0.

With B equal to 1, A is complemented. This is another way to perform a NOT function, and in fact, PBASIC2 does not have a NOT operator.

**PBASIC2 and Logical Operators**

PBASIC2 uses symbols to represent the logical functions:

AND: &

OR: |

XOR: ^

Lets try a short program to test each function.

```
DEBUG "1 AND 0 = ",DEC 1 & 0, CR
DEBUG "1 OR 0 = ", DEC 1 | 0, CR
DEBUG "1 XOR 1 = ",DEC 1 ^ 0,CR
```

The result in the debug screen would be:

```
1 AND 0 = 0
1 OR 0 = 1
1 XOR 1 = 0
```

Notice that the XOR can be used to complement or invert a value.

When using an IF-THEN statement, the IF expression is evaluated as being true or false. If true, the THEN is performed (limited to a jump in PBASIC2). An expression such as `IF 100 > 45` would be true since 100 IS greater than 45. Logical operators can also be used here, let's look an example.

#### **'PROG\_G-1**

**'This program will logically evaluate the BLUE and BLACK pushbuttons,  
'and produce a tone if the expression is true. To make buttons pressed = true,  
'the XOR function will be used to invert the input since pressed = 0.**

```
INPUT 8      'Set up Blue PB for input
INPUT 9      'Set up Black PB for input
A var bit    'Decare bit variables
B var bit
```

#### **Loop:**

```
A = IN8 ^ 1  'Read Blue and compliment
B = IN9 ^ 1  'Read Black and compliment
IF A & B THEN IsTrue 'Evaluate and sound tone if true
Goto Loop    'Repeat
```

```
IsTrue      'Sound tone
Freqout 11, 100, 2500
GOTO Loop
```

#### **Try it!**

Change the program so that the green and blue and black buttons  
have to be depressed!

Running this the speaker will sound when both BLUE AND BLACK are pressed. For the IF statement we could have written:

```
IF A & B = 1 THEN IsTrue
```

But it wasn't necessary, since the `A AND B` would return a 1 (true) when both are pressed making the IF true. If we want to produce an output when the result was false would could have done this two ways:

```
IF A & B = 0 THEN IsTrue 'Tone when A AND B is NOT true by checking = 0!
```

Or

```
IF A & B ^ 1 THEN IsTrue 'Tone when A AND B is NOT true by inverting a false (0)!
```

We can also evaluate more than two expressions. Let's try our example of being stopped at a traffic light. Here are the input definitions:

- The light is green (Blue PB).
- "No turn on red" sign posted (Black PB).
- Coast is clear (Green PB).

**'PROG\_G-2**

'This program will logically evaluate the BLUE and BLACK and RED pushbuttons.  
 'To drive through a traffic signal, The light must be green (BLUE PB pressed),  
 'OR there can not be "no turn on red sign (BLACK PB NOT Pressed),  
 'AND the coast must be clear (GREEN PB is pressed).  
 'Buttons are inverted to make pressed = 1.

```

INPUT 8      'Set up Blue PB for input
INPUT 9      'Set up Black PB for input
INPUT 10     'Set UP Green PB for input
GreenLight var bit      'The light is green = 1
NoTurnSignPresent var bit 'There is a sign = 1
CoastClear var bit      'Coast is clear = 1
NoTurnSignInverted var Bit 'Compliment of sign, No sign = 1
  
```

**Loop:**

```

GreenLight = IN8 ^ 1      'Read Blue and compliment
NoTurnSignPresent = IN9 ^ 1 'Read Black and compliment
NoTurnSignInverted = NoTurnSignPresent ^ 1 'Compliment sign
CoastClear = IN10 ^ 1     'Read Green and compliment
  
```

```

IF GreenLight | NoTurnSignInverted & CoastClear THEN CanGo 'Evaluate and sound
                                                             'tone if true
  
```

```

Goto Loop      'Repeat
  
```

```

CanGo      'Sound tone
Freqout 11, 100, 2500
GOTO Loop
  
```

The Go tone will sound under two conditions:

- The Green button is pressed (coast is clear) AND the Black PB is NOT pressed (There is no "No turn on red" sign).
- The Green button is pressed (coast is clear) AND the Blue PB is pressed (Green light).

Notice that we did not use parenthesis for OR-ing the 'green light' and 'turn on red sign' terms. We wrote the equation from left to right to ensure that it was done prior to AND-ing with the 'coast is clear'.

***Masking***

We are not limited to performing these operations on single bits. When working with numbers greater than 1, the logical operator will be performed in a bit-wise fashion for the entire number. This means that each bit position in the first number will be logically compared with the same bit position in the second number. Take for example the following expression:

$$1011_2 \text{ AND } 010_2$$

This can be re-written as:

$$\begin{array}{r} \text{AND } 1011_2 \\ \quad 0010_2 \\ \quad 0010_2 \end{array}$$



Notice that the second term (010<sub>2</sub>) is padded with zero's to have the same number of bits. The AND operator was applied to each sets of bits to get a result of 0010<sub>2</sub>. Only the 2<sup>nd</sup> bit position from the right contains 1's in both terms. This produces a 1 for that position in the result. PBASIC2 code to perform and display this would be:

**DEBUG IBIN4 %1011 & %010 'AND two numbers, pad results to 4 binary digits**

We can also logically evaluate decimal numbers, but it will still be performed in a binary or bit-wise fashion. Take the following example of 85 OR 15:

DEBUG "85 OR 15", DEC 85 | 15

It would return:

85 OR 15 = 95

Does this make sense? Lets do the same but displaying the numbers in binary:

**DEBUG IBIN8 85, CR**  
**DEBUG IBIN8 15, CR**  
**DEBUG "\_\_\_\_\_", CR**  
**DEBUG IBIN8 85 | 15, CR**

It returns the following:

%01010101

%00001111

---

%01011111

01011111<sub>2</sub> converted to decimal is 95. Notice that for each bit column, the OR operation was evaluated so that any column that had a 1 in either term returned a 1 in the result.

Decimal is useful at times though. Remember each bit position is a higher power of 2 (starting with 2<sup>0</sup>), so if we wanted to turn on bit 5 of a byte variable named ByteIn then we could use the following code:

**ByteIn = ByteIn | 16**

**'16 is 2 to the forth or the 5<sup>th</sup> binary place.**

The ability to control bits in a binary expression, either individually or as a group, is known as *masking*. Using a logical operator and a second expression we can force single bits ON or OFF or invert them.

Here are some common reasons to mask bits and how they are performed:

#### 'TRON TIP

Microcontrollers and microprocessors always operate on bytes or words. When bit operations are required they use internal masking.

Reason	Function	Example Mask
To force a bit ON	OR with a 1 in bit position	0110 <sub>2</sub> OR 1000 <sub>2</sub> = 1100 <sub>2</sub>
To force a bit OFF	AND with a 0 in bit position	1111 <sub>2</sub> AND 0011 <sub>2</sub> = 0011 <sub>2</sub>
To toggle (invert) a bit	XOR with a 1 in bit position	1010 <sub>2</sub> XOR 1100 <sub>2</sub> = 0110 <sub>2</sub>

Let's try an example. We'll span the potentiometer so it has a range of 0 -15, and compliment the number so we can drive our active LOW LEDs to display the number. We'll force the LED on the black button to stay on no matter what.

**'PROG\_G-3**

**'Span Pot for 0 -15, and display on Activity Board LEDs, use a mask to compliment.**

**'Mask so Black's LED is always ON.**

**Potword var word**

**'Holds Pot result word**

**Pot16 var nib**

**'Holds Pot spanned 0-15**

**NibOut var nib**

**'Holds final result**

**MaskResult var Nib**

**'Hold black masked result**

**Pot con 7**

**'Constant for POT pin**

**CompMask con %1111**

**'Mask to compliment results for low active LEDs**

**BlkMask con %0010**

**'Mask to keep black LED on**

**DIRC = %1111**

**'Set each bit in nibble C to outputs (8-11)**

**Loop:**

**HIGH POT: Pause 10**

**'Charge Cap**

**RCTIME Pot, 1, Potword**

**'Measure RC time for pot**

**Pot16 = potword / 10 \* 16 /500 MAX 15**

**'Scale it down for 0-15 span**

**MaskResult = Pot16 | BlkMask**

**'Mask to keep black bit on**

**NibOut = MaskResult ^ CompMask**

**'Mask to compliment output**

**DEBUG 0, CR**

**'Display math, 0 clears display**

**DEBUG IBIN4 POT16," POT",CR**

**DEBUG IBIN4 BlkMask, " BLACK BUTTON MASK (OR)", CR**

**DEBUG "\_\_\_\_\_", CR**

**DEBUG IBIN4 MaskResult," RESULT TO KEEP BLACK ON", CR**

**DEBUG IBIN4 CompMask," COMPLIMENT MASK (XOR)", CR**

**DEBUG "\_\_\_\_\_", CR**

**DEBUG IBIN4 NibOut, " RESULT COMPLIMENTED FOR LEDs",CR**

**PAUSE 500**

**OUTC = NibOut**

**'Set the LEDs**

**GOTO Loop**

<b>Try it!</b>
----------------

Change the program so that the green's LED never comes on!
--

### ***Binary Addition & Subtraction***

To this point we have worked mainly with positive numbers in our Stamp programs, but PBASIC2 can work with negative numbers as indicated by the use of subtraction. Actually Computers can ONLY add. To perform a subtraction the processor needs to manipulate the subtrahend (the number being subtracted) and add the 2 together.

Addition in binary is performed just as our addition in decimal. We add the numbers, and if it exceeds our range of single digits (0-9), we carry to the next column and add it in there. In binary it works the same, but having only 2 digits, 0 and 1, we have a carry a lot sooner! Let's look at some sample additions:

$$\begin{array}{r}
 0 \\
 +0 \\
 \hline
 0
 \end{array}
 \qquad
 \begin{array}{r}
 0 \\
 +1 \\
 \hline
 1
 \end{array}
 \qquad
 \begin{array}{r}
 1 \\
 +0 \\
 \hline
 1
 \end{array}
 \qquad
 \begin{array}{r}
 1 \text{ (carry)} \\
 1 \\
 +1 \\
 \hline
 10
 \end{array}$$

Easy, huh? Try this: think of the numbers being added as terms for a truth table, then look at the results in the first column for each. What logical function is being performed? XOR! And in fact digital systems use XOR's when performing addition!

To add  $1001_2$  and  $0011_2$  (convert to decimal and check the results!):

$$\begin{array}{r}
 11 \text{ (carries)} \\
 1001_2 \\
 +0011_2 \\
 \hline
 1100_2
 \end{array}$$

Subtraction works with signed numbers. We subtract our numbers by simply placing a (-) between the numbers. But since computers don't inherently understand how to subtract, they need to add signed numbers (we do also,  $4-2$  is really  $4 + (-2)$ ). The form that the signed number takes is called the *2's compliment*. Complimenting all the bits in a number, then adding a 1 to the result makes a 2's compliment number.

Let's try an example by performing  $10 - 7$ , which is  $10 + (-7)$ . First we need them in binary:

$$10 = 1010_2$$

$$7 = 0111_2$$

Since 7 is negative, we need to make it signed getting the 2's compliment:

First compliment it:  $0111_2 \rightarrow 1000$

Then add 1 to it:

$$\begin{array}{r}
 1000_2 \\
 +1_2 \\
 \hline
 1001_2
 \end{array}$$

-- 2's compliment of  $0111_2$

Why isn't this just +9? There are unseen 1's all the way to most significant bit not shown. You'll see in some examples coming up. Now that we have the 2's compliment we can add our numbers together, and disregard any overflow carry if there is one:

$$\begin{array}{r}
 1010_2 \text{ (10)} \\
 +1001_2 \text{ (-7 in 2's compliment signed number)} \\
 \hline
 10011_2
 \end{array}$$

Disregarding the final carry we get:  $0011_2$  or 3. It works!

PBASIC2 works math in 2's compliment signed numbers. When working in signed numbers, if the LEFT-MOST bit is 1, it indicates that it is negative. Note from our example of converting 7 to 2's compliment. We were left with a 1 in the left most position. PBASIC2 works math using

words or 16 bit numbers. Using just positive numbers we have a range of 0 to 65535. When working with signed bits we lose the leftmost bit to indicate sign and end up with a range of -32768 to +32767.

Let's try a little sample code and analyze the results:

```
DEBUG IBIN -7      'Result: %1111111111111001
```

Can you see our 2's compliment of 7 at the end? (1001<sub>2</sub>)

Another:

```
DEBUG IBIN -32768  'Result: %1000000000000000
```

Let's try some math. If we are at 0, and subtract 1, what would we get? In decimal we get -1. In binary we would roll back from 16 0's to all 1's.

```
DEBUG IBIN 0 - 1   'Result: %1111111111111111
```

Of course, we can show the results in decimal. Let's try the following:

```
DEBUG IBIN -5, " ", DEC -5
```

We would get back:

```
%1111111111111011 65531
```

Now, we know the first is in 2's compliment form. So when we tell PBASIC to give the decimal value we get 65531, which is correct for the bit pattern we have. But that isn't really what we wanted to see for -5. We can use SBIN and SDEC (signed binary, signed decimal) to show results as signed numbers.

```
DEBUG SBIN -5, " ", SDEC -5
```

We would get back:

```
-%101 -5
```

Makes more sense, doesn't it? Just realize that there is no negative sign internal to digital systems. When we perform math like 10 - 50 our results may not be what we expect, -40. Just displaying the decimal value would produce 65496. But the processor knows this is a signed number, so when we add 100, we would get 60.

```
DEBUG SDEC 10 - 50 + 100      'Results: 60
```

NOTE: If we store the results of 10 - 50 in a byte, our signed number will be lost because the upper 8 bits would be stripped off!

<b>Temp var byte</b>	<b>'Declare byte variable</b>
<b>Temp = 10 - 50</b>	<b>'Subtract and store in a byte</b>
<b>DEBUG DEC Temp + 100</b>	<b>'Add 100 and display</b>

Our results: 316

It just doesn't make ANY sense now because by forcing our results into a byte the signed number is ruined.

Let's code one program to demonstrate negative numbers. This program will use the potentiometer and scale it so that it measures from approximately - 250 to +250 and display the results in 2's compliment and as signed numbers.

**'PROG\_G-4**

**'Scale the POT for approx. -2500 to +2500 and display results**

**POTIn var word**

**'Declare to hold Pot Results (0 to ~ 5300)**

**POTScaled var word**

**'Declare to hold scaled down (-250 to ~ +250)**

**POTpin con 7**

**'Constant for Pin number of Pot**

**Loop:**

**HIGH POTpin:Pause 10**

**'Charge RC Pot**

**RCTIME POTpin, 1, PotIn**

**'Measure Pot**

**POTScaled = PotIn / 10 - 250**

**'Scale input then display**

**DEBUG "2's compliment = ", IBIN PotScaled," Binary Signed = ",SBIN PotScaled**

**DEBUG " Decimal Signed = ", SDEC PotScaled, CR**

**PAUSE 250**

**GOTO Loop**

## Section H: Digital Communications

### Reference:

A. BASIC Stamp Manual, Version 1.9. 1998. Parallax, Inc.

### Objectives:

- 1) Discuss advantages of parallel and serial communications.
- 2) Discuss the need for address lines in digital systems.
- 3) Discuss the need for synchronizing serial data.
- 4) Discuss how asynchronous data is decoded.
- 5) Write PBASIC2 code to communicate between BS2s.

**NOTE:** The programs in this section demonstrate communication methods between 2 BASIC stamps. One or two personal computers may be used for testing these programs. If using one computer, program the transmitting Activity Board first then the receiving board. The receiver is probably the best choice to view the debug window from.

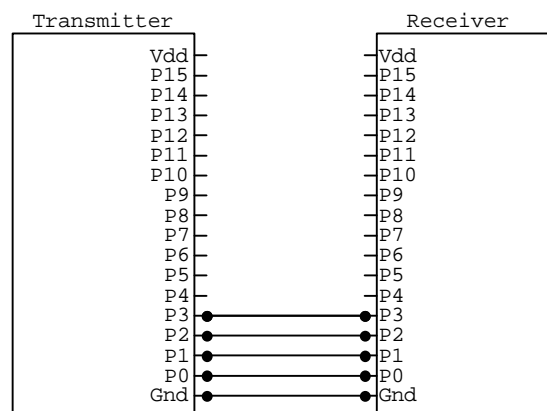
### Introduction

Digital communications are abundant anywhere digital systems are involved. On a personal computer digital communications are involved in using the mouse to move a pointer, in sending a document to the printer, and the downloading a web page on the Internet. Communications are also involved internal to the computer when the hard-drive is accessed, when characters typed are stored in RAM and displayed on the screen, and in numerous other processes.

Digital communications involves the transfer of data from one device to another, typically as a byte, but it may be a bit, nibble, or a word of 16-bits or more. Two main methods of data transport are utilized: parallel and serial. In this section we will explore both methods, their problems and solutions and their benefits and limitations. Data transport from one Activity Board acting as the transmitter to another acting as the receiver will be utilized to illustrate the communication methods.

### Parallel Communications

Parallel communications is the process of moving an entire byte (or whatever word length is involved) simultaneously. Figure H-1 is a simplified Activity Board showing the I/O pin header. Figure H-1 shows a nibble on I/O pins P0-P3 being transmitted from one stamp (transmitter) to



**Figure H-1: Simple Parallel Communications**

another (receiver). A ground connection (Gnd) is made between the two to provide a common reference voltage point.

The transmitter with data of 1010 simply places the bits on the output where the receiver can read them as inputs. The following set of programs, H-1T and H-1R, shows the code for both the transmitter (T) and receiver (R). The data for the transmitter is collected from the pushbuttons, P8-P11 (nibble C of the BS2) communicated across the data lines on P0-P3 (Nibble A of the BS2) where the data is read and placed on the LEDs of the receiver (Nibble C). As the buttons on the transmitter are pressed, the corresponding LEDs on the receiver light.

```
'PROG_H-1T
'Parallel nibble transfer transmitter
'Transfers the buttons nibble to nibble A (P0-P3) continuously

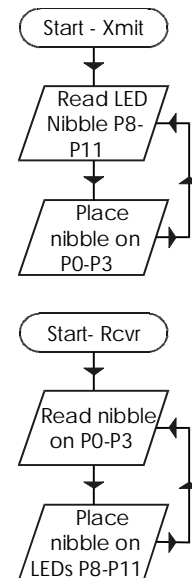
DIRC = %0000      'Set buttons as inputs
DIRA = %1111      'Set P0-P3 as outputs

Loop:
  DEBUG "Data out = ",IBIN4 INC,CR 'Display the nibble from the button
  OUTA = INC        'Sets nibble A = nibble C (transmits buttons)
  GOTO Loop
*****

'PROG_H-1R
'Parallel nibble transfer receiver
'Reads nibble A (P0-P3) and sets LEDs equal to it

DIRA = %0000      'Set up P0-P3 as inputs
DIRC = %1111      'Set up button LEDs as outputs

Loop
  DEBUG "Data in = ",IBIN4 INA, CR 'Display data
  OUTC = INA        'Set LEDs to nibble A (Receives and displays)
  GOTO Loop
```



This is fairly simple and straightforward. As the data changes on the transmitter, the receiver instantly is updated with the new data. Unfortunately this isn't desired in most cases. The processor may have other events going on that use those same data lines, or *data bus*, and the data would only be read by the receiver when the processor determined it was time for it. Take a look back our very first figure back in Section A of simple microprocessor connections. The RAM, ROM and I/O devices all share a common data bus (but each device would only be updated with the data when the process had data ready for it alone). Address lines are used to instruct the devices when to accept the data on the data bus. Only then will the device *latch* in the bits (or in the case of output devices, place data on the bus).

Figure H-2 once again shows the two Activity Boards connected, but this time with the addition of a clock line on P6. It is used in Program H-2T and H-2R to synchronize the data so that the receiver latches in the data only when clocked by the transmitter. The transmitter program sets P6 HIGH (clock) to instruct the receiver to read the data. A beep accompanies this pulse for user indication. As the buttons on the transmitter are pressed, no change will occur until the clock line goes high and the transmitter beeps. P6 is acting as an address line or clock line.

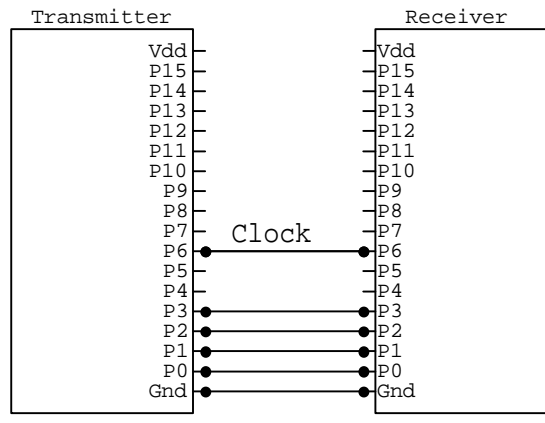


Figure H-2: Parallel Communications with Synchronizing Clock Line

```

'PROG_H-2T
'Parallel nibble transfer transmitter
'Transfers the buttons nibble to nibble A (P0-P3) continuously

x var byte          'For-next variable
Clock con 6         'Constant for transmit data clock

DIRA = %1111        'Set P0-P3 as outputs
LOW Clock           'Set clock low

loop:
  FOR x = 1 TO 10    'display non-transmitted data (for visual effects)
    Pause 200
    DEBUG "Data = ",IBIN4 INC,CR 'Display the nibble from the button
  NEXT

  DIRC = %0000       'Ready to transmit!
  OUTA =inc          'Set buttons as inputs
  HIGH Clock         'Sets nibble A = nibble C (ready Data)
  FREQOUT 11, 100, 2500 'Set data clock HIGH for transmit
  DEBUG "Data = ",IBIN4 INC, " (XMIT)",CR 'Sound Tone
  LOW Clock          'Display the nibble from the button
  GOTO loop          'End of data clock

*****

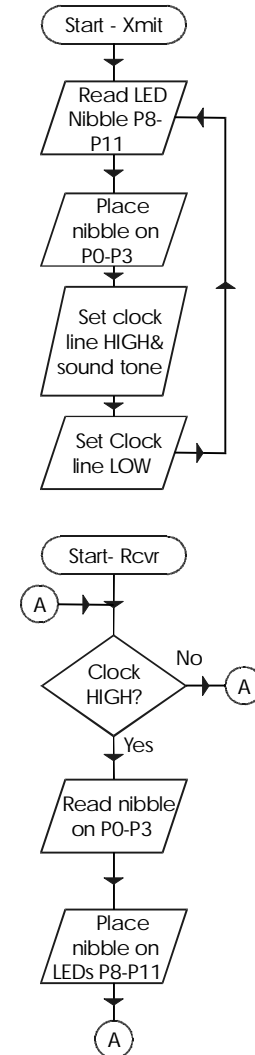
'PROG_H-2R
'Parallel nibble transfer receiver with data clock
'Reads nibble A (P0-P3) when P6 is HIGH and sets LEDs equal to it
btnClock var byte   'Variable for button function
Clock con 6         'Constant to define clock line
DIRA = %0000        'Set up P0-P3 as inputs
DIRC = %1111        'Set up button LEDs as outputs
btnClock = 0        'Clear button variable

Loop:               'Wait until P6 is high, debounce, no repeat
  Button Clock, 1, 255, 0, btnClock, 1, Accept
  Goto Loop

Accept:
  OUTC = INA         'Set LEDs to nibble A (Receives and displays)
  DEBUG "Data in = ",IBIN4 INA, CR 'Display data

GOTO LOOP

```





What if we wanted to use the same data bus, but send data to multiple Activity Boards? With a second receiver we could use P7 as its clock line. While changing the data on the bus, we can direct whether the 1st receiver or the 2nd one reads it by selecting address line P6 or P7 HIGH. Having a single data bus shared by multiple devices is one example of *multiplexing*.

Parallel communications are very fast and simple. An entire group of bits can be transferred simultaneously at very high speeds to multiple devices. Unfortunately we cannot run multiple lines to every device we want to send data to. Modems only receive data from one line, our telephone. It would be very costly to run 8 lines or more lines into our homes to have a parallel transfer of information. The same limitations are involved in fiber optics and other wireless communications. Sometimes we do not wish to use multiple lines for simple data transfer, such as from our ADC0831 digital to analog converter or from the mouse on your computer. In these cases serial communication is utilized.

### **Synchronous Serial Communications**

Virtually all digital systems process and store information in parallel internally. But when the need arises the system may rely on serial communications to transfer the data from one device to another. It may be due to physical limitations of the medium (fiber optics), space constraints of smaller sized ICs and cables (like our mouse) or the limited resources of the processor (BS2 and limited data lines).

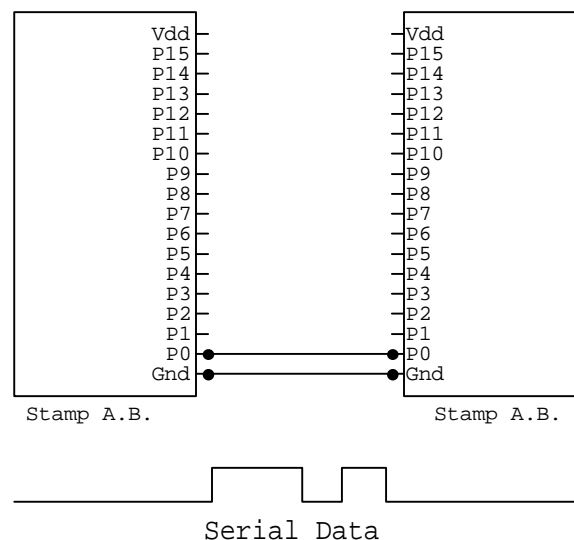
Serial communications require 3 basic steps:

- 1) Convert the parallel data to serial data.
- 2) Transfer the data
- 3) Convert the serial data back to parallel data.

#### **'TRON TIP**

In electronics, an IC to perform this function as both a transmitter and receiver is a UART (Universal Asynchronous Transmitter-Receiver).

Figure H-3 shows the connections for a simple serial connection and a sample stream of data being transferred.



**Figure H-3: Serial Data Communication with No Synchronization**

Looking at the serial data in Figure H-3 and knowing it was being transferred MSB first, we may be able to reconstruct the data by noting how long a pulse lasts. It appears to have a single HIGH arriving at the receiver first followed by single LOW and then 2 HIGHS, giving us a nibble of  $1101_2$ . We might base that on 2 pulses, a high and a low and assuming those were individual bits, meaning the long HIGH would be a 2 HIGH bits together, but this only an educated guess on our part. The data stream may very well be 4 HIGHS followed by 4 LOWS followed by 8 HIGHS giving us data of  $1111111100001111_2$ . It may also be that the LOWs on either side of the 'pulse' are also important data that are just continuous LOWs. Without more information of where the individual bits are, we couldn't say for sure.

Digital systems have the same problem. Without further information reading a stream of data is meaningless because there is nothing to break up the HIGHS and LOWs into individual bits. Program H-3T and H-3R uses the connections of Figure H-3 to attempt transmitting a nibble serially. Running the programs yields erratic results when comparing the transmitter buttons pressed and the receiver LEDs lighting because there exists no synchronization between the two.

#### "PROG\_H-3T

'Transmits a nibble serially from buttons to P0

'No synchronizing or timing

```

NibData var nib      'Nibble to hold button data
BitOut var bit       'Bit from nibble to be transmitted
x var nib            'For-Next variable
Output 0             'Set up P0 for output

Loop:
Pause 1000           'Pause 1 second
DIRC = %0000         'Set up P8-P11 as inputs (buttons)
NibData = INC        'Read buttons & Display
FREQOUT 11,200, 2000
Debug CR, "Nibble = ",IBIN4 NibData, CR
For x = 1 to 4       'Sends out nibble MSB first
    BitOut = NibData.Bit3 'Get MSB bit
    DEBUG ? BitOut      'Display it
    OUT0 = BitOut       'Send it
    NibData = NibData << 1 'Shift nibble bits left
Next

```

Next

Goto Loop

\*\*\*\*\*

#### 'PROG\_H-3R

'Receives a nibble serially from P0 and places on LEDs

'No synchronizing or timing

```

NibData var nib      'Nibble to hold incoming nibble
BitIn var Bit        'Bit to hold incoming Bit
X var nib            'For-Next variable
Input 0              'Set P0 as input

DIRC = %1111         'Set up Button LEDs as outputs
Loop:
For X = 1 to 4       'Read for bits and assemble nibble MSB first
    BitIn = IN0      'Read bit
    DEBUG ? BitIn    'Display
    NibData = NibData << 1 'Shift nibble
    Nibdata = NibData + BitIn 'Add bit to nibble at LSB position
Next

```

Next

OUTC = NibData 'Set buttons LEDs

Debug "Nibble in = ",IBIN4 NibData, CR, CR

Goto Loop

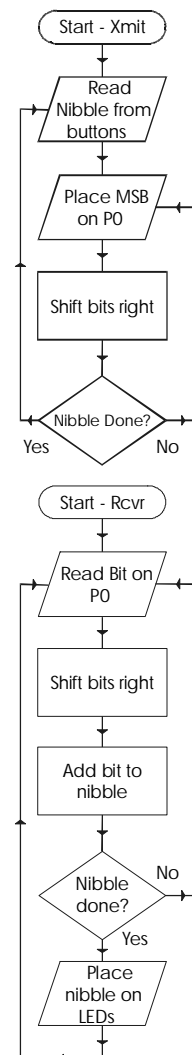
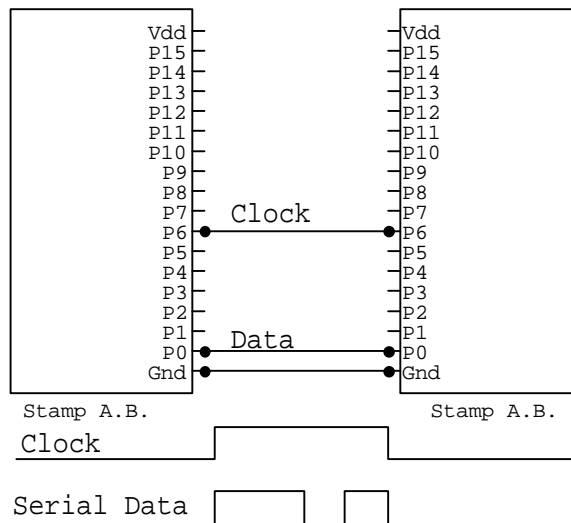


Figure H-4 and Programs H-4T and H-4R attempt to resolve the problem by adding a clock line between the two systems to synchronize the transmitter and receiver. While the data is transmitting, the clock line (P6) is HIGH so the receiver knows that data is present to be collected.



**Figure H-4: Serial Data Communications with Word (4-bit) Synchronization**

**'PROG\_H-4T**

**'Transmits a nibble serially from buttons to P0**

**'Start synchronization on P6, one second intervals**

**NibData var nib**

**'Nibble to hold button data**

**BitOut var bit**

**'Bit from nibble to be transmitted**

**x var nib**

**'For-Next variable**

**Output 0**

**'Set up P0 for output**

**DIRC = %0000**

**'Set up P8-P11 as inputs (buttons)**

**Loop:**

**Pause 1000**

**'Pause 1 second**

**Freqout 11,200, 2500**

**NibData = INC**

**'Read buttons & Display**

**Debug CR, "Nibble out = ",IBIN4 NibData, CR**

**HIGH 6**

**'Set synch line high**

**For x = 1 to 4**

**'Sends out nibble MSB first**

**BitOut = NibData.Bit3**

**'Get MSB bit**

**DEBUG ? BitOut**

**'Display it**

**OUT0 = BitOut**

**'Send it**

**NibData = NibData << 1**

**'Shift nibble bits left**

**Next**

**Low 6**

**'Clear sync line**

**Goto Loop**

\*\*\*\*\*

**'PROG\_H-4R**

**'Receives a nibble serially from P0 and places on LEDs**

**'Start Synchronization Only**

**NibData var nib**

**'Nibble to hold incoming nibble**

**BitIn var Bit**

**'Bit to hold incoming Bit**

**X var nib**

**'For-Next variable**

**btnClock var byte**

**'Button function variable**

**Clock con 6**

**Input 0**

**'Set P0 as input**

```

DIRC = %1111      'Set up Button LEDs as outputs
Input 6            'Set up P6 as input
btnClock = 0       'Clear button variable

Loop:              'Wait until sync pulse starts
  Button Clock, 1, 255, 0, btnClock, 1, Accept
Goto Loop

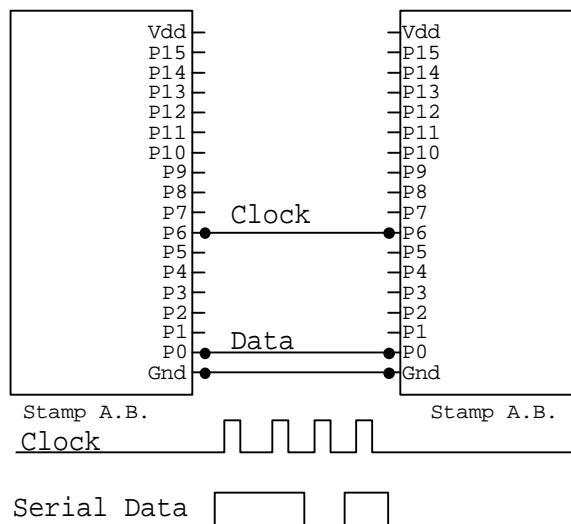
Accept
For X = 1 to 4     'Read for bits and assemble nibble MSB first
  BitIn = IN0      'Read bit
  DEBUG ? BitIn    'Display
  NibData = NibData << 1 'Shift nibble
  Nibdata = NibData + BitIn 'Add bit to nibble at LSB position
Next

OUTC = NibData     'Set buttons LEDs
Debug "Nibble in = ",IBIN4 NibData, CR, CR

Goto Loop

```

This program also performs with limited success. Even though the receiver is signaled when to collect the serial data, the rate at which it reads the four bits may not coincide with the rate at which the data is being transmitted again providing erratic results. The receiver must be clocked for each individual bit being transmitted as in the following figure and programs.



**Figure H-5: Serial Data Communications with Bit Synchronization**

```

'PROG_H-5T
'Transmits a nibble serially from buttons to P0
'Sends clock sync line for each bit

NibData var nib      'Nibble to hold button data
BitOut var bit       'Bit from nibble to be transmitted
x var nib            'For-Next variable

Output 0             'Set up P0 for output
LOW 6                'Set clock line low

Debug 0,"Press Blue Button to start",CR

StartLoop:           'Flash Blue's LED and wait for start
LOW 8: Pause 200: HIGH 8:PAUSE 200: INPUT 8
If IN8 = 1 then Startloop

```

```

DIRC = %0000          'Set up P8-P11 as inputs (buttons)
Loop:
Pause 1500             'Pause 1 second
NibData = INC          'Read buttons & Display
Debug CR, "Nibble out = ",IBIN4 NibData, CR
Freqout 11,200, 2500   'Sound tone

For x = 1 to 4          'Sends out nibble MSB first
  Pause 50
  BitOut = NibData.Bit3 'Get MSB bit
  DEBUG IBIN4 NibData, " ",? BitOut      'Display it
  OUT0 = BitOut          'Send it
  HIGH 6                 'Set clock line high
  Pause 50               'pause between bits
  LOW 6                  'Set clock line low
  NibData = NibData << 1 'Shift nibble bits left
Next

GOTO Loop
*****

'PROG_H-5R
'Receives a nibble serially from P0 and places on LEDs
'Individual Bit Sync on P6

NibData var nib        'Nibble to hold incoming nibble
BitIn var Bit          'Bit to hold incoming Bit
X var nib              'For-Next variable
btnClock var byte      'Button function variable
Clock con 6            'Define clock line as P6

Input 0                'Set P0 as input
DIRC = %1111           'Set up Button LEDs as outputs
Input 6                'Set up P6 as input
btnClock = 0           'Clear button variable

Loop:                  'Wait until sync pulse starts
DEBUG "Waiting for Data",CR
NibData = 0
For X = 1 to 4          'Read for bits and assemble nibble MSB first
  BitLoop
  Button Clock, 1, 255, 0, btnClock, 0, BitLoop      '***Wait for clock pulse
  BitIn = IN0                                         'Read bit

  NibData = NibData << 1      'Shift nibble
  Nibdata = NibData + BitIn   'Add bit to nibble at LSB position
  DEBUG "BitIn = ",IBIN BitIn, " NibData = ",IBIN4 NibData,CR
Next

OUTC = NibData          'Set buttons LEDs
Debug "Nibble in = ",IBIN4 NibData, CR, CR

GOTO Loop

```

(When running, RESET the transmitter then the receiver. Press the flashing blue button on transmitter for initial synchronizing of systems)

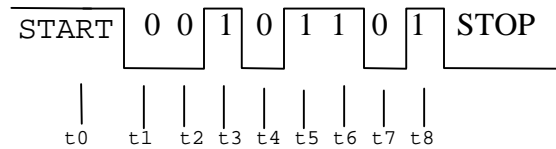
For Programs H-5T and H-5R as buttons are pressed on the transmitter and it beeps to indicate a transmission, the corresponding LEDs on the receiver light. Success!! With a little more programming, the same data was transmitted as the parallel version with only 3 lines instead of 6, but at a much slower rate!

This method would work fine if we had 3 wires to run between two systems. But in the instances of your modem or wireless mediums such as fiber optic, radio waves, or even the remote control for your television we need another method to identify the individual bits.

## Asynchronous Serial Communications

In asynchronous communications the transmitter/receiver are not locked together with a common clock. One method of asynchronous communications involves the receiver being told when to start collecting the bits and knowing how fast the transmitter is sending the data (or how long a single bit lasts).

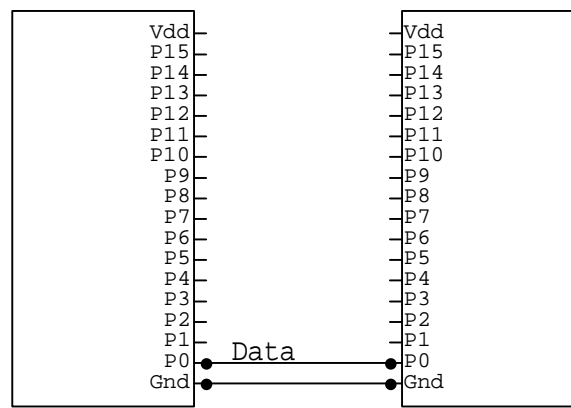
This is the basis for the RS-232 communications for your computer when it is communicating with any standard serial device, such as the mouse, a modem, or even the BS2! Let's take a look at an asynchronous byte being transmitted in Figure H-6.



**Figure H-6: Asynchronous Timing Diagram**

Lets assume the each bit is being transmitted for 50 milliseconds (50 mS or .050 seconds). The start bit is used to signal the receiver that a data stream is arriving at  $t_0$ . If the receiver is in agreement with the transmitter it will be able to predict the time for where each bit is in the stream. With a bit length of 50mS, from the time of the start bit the receiver will wait 75 mS before reading the first bit ( $t_1$ ). This ensures it reads the bit in the approximate center. From that point it will read each consecutive bit every 50 mS to approximate their center times ( $t_2$ - $t_8$ ).

Programs H-6T and H-6R with connections of Figure H-6b perform this very act. The transmitter begins with a start bit, sends the 8 data bits in a byte (the lower nibble contains our button data) and a stop bit at 50 mS (BitTime) intervals. The receiver waits until it detects the start bit, waits 75 mS ( $\text{BitTime} * 3 / 2$ ) prior to reading the first bit (MSB) then proceeds to read the consecutive bits every 50 mS (BitTime).



**Figure H-6b: Asynchronous Communications**

```

'PROG_H-6T
'Sends a bit Asynchronously on P0

NibData var nib           'Holds nibble to send
BitOut var bit            'Hold outgoing bit
BitTime var word          'length of bits in mSec

DIRC = %0000              'Set up buttons as inputs
LOW 0                     'Output P0 to LOW
BitTime = 50              'Set length of bits in mSec

Loop:
  Pause 2000               'Deley between transmissions
  NibData = INC            'Read buttons
  Freqout 11, 250, 2500    'Sound tone
  DEBUG CR, IBIN4 ? NibData 'Display
  Debug "Start Bit",CR
  HIGH 0                  'Send Start bit (HIGH)
  Pause BitTime
  BitOut = NibData.Bit3: Gosub SendBit 'get each bit from nibble, Go send
  BitOut = NibData.Bit2: Gosub SendBit
  BitOut = NibData.Bit1: Gosub SendBit
  BitOut = NibData.Bit0: Gosub SendBit

  DEBUG "Stop Bit",CR      'Send Stop bit (LOW)
  LOW 0: Pause BitTime
  Goto Loop

SendBit:
  OUT0 = BitOut            'Set output to bit
  DEBUG ? BITOUT           'display
  Pause BitTime            'Time of bit equal to BitTime
  Return

```

\*\*\*\*\*

```

'PROG_H-6R
'Receives a nibble Asynchronously on P0
BitTime var Word          'Holds expected length of bit in mS
NibIn var nib             'Incoming Nibble
BitIn var bit             'Incoming bit

Input 0                   'Set up P0 for input
DIRC = %1111              'Set button Leds as outputs
BitTime = 50              'Set Length of bits in mS

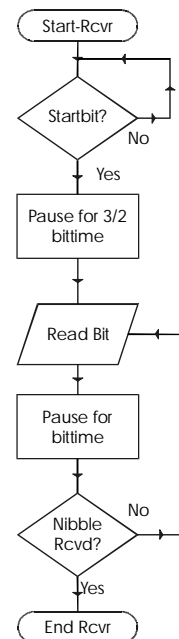
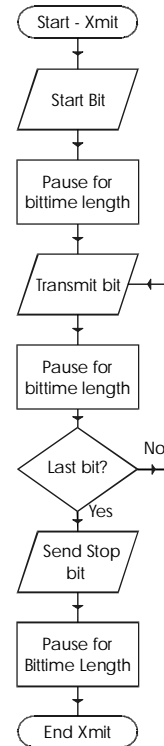
Loop:
  Debug CR,"Waiting for Start bit",CR
Waiting:
  IF IN0 = 0 THEN Waiting  'Wait for Start bit

  PAUSE BitTime * 3 / 2    'Wait until middle of MSB bit

  Gosub GetBit: NibIn.Bit3 = BitIn 'Go get the bit, Put in Nibble
  Gosub GetBit: NibIn.Bit2 = BitIn
  Gosub GetBit: NibIn.Bit1 = BitIn
  Gosub GetBit: NibIn.Bit0 = BitIn
  Debug "Nibble in = ", IBIN4 NibIn,CR,CR
  OUTC = NibIn             'Place nibble on LEDs
  GOTO Loop

GetBit:
  BitIn = IN0              'Read Bit
  DEBUG ? BitIn            'Display incoming bit
  PAUSE BitTime            'Wait until middle of next bit
  RETURN

```



But what happens if the BitTime of the receiver was not set the same as the transmitter? By changing BitTime on the receiver to 25 mS the receiver would be reading data twice as fast as

the transmitter was sending it. The receiver would have collected all 8 bits by the time the actual data reached the halfway point!

The term used to define how fast the data is transmitted is called the *Baud rate*. It refers to the number of data groups able to be transmitted per second. Lets take our example. There are 10 bits in our group, 8 data bits plus a start and stop bit. The total time to transmit this group would be  $10 \times 50 \text{ mS}$  or  $500 \text{ mS}$  (0.5 seconds). The inverse of this would define our baud rate, so  $1/0.5 = 2 \text{ baud}$ .

Let's look at a more standard baud rate: 9600 baud. To transmit our 10-bit group at 9600 baud, it would take  $1/9600$  or .000104 seconds! Therefore each bit would last one-tenth of this time or .0000104 seconds, or .01 mS, or  $104 \mu\text{Sec}$  (microseconds, millionths of a second).

We can't program PBASIC2 to have pauses short enough to achieve a baud rate of 9600. Fortunately, we don't have to because PBASIC2 has 2 commands, SEROUT and SERIN that perform asynchronous serial data transfers. The full use of these commands is extensive and powerful. We will look at the most basic usage (refer to Reference A for more detail).

SEROUT *tpin, baudmode, outputData*

Where:

tpin: The pin to transmit on (0-15).

baudmode: A 16-bit word defining characteristics of the data transmission, such as baud rate.

outputData: The byte to be transmitted.

SERIN *rpin, baudmode, timeout, tlabel [inputData]*

Where:

rpin: The pin to receive on (0-15).

baudmode: A 16-bit word defining characteristics of the data transmission, such as baud rate.

timeout: Specifies number of milliseconds to wait for data before generating a timeout.

tlabel: Label for a branch when a timeout occurs.

inputData: A variable where the received byte is stored.

Let's look at baudmode a bit closer. There are various ways we can transmit the data. It can be 7 or 8 bits, have a parity bit used to verify correct data, one or two stop bits, and inverted or non-inverted are just some choices. All of these have to be in agreement between the receiving and transmitting devices besides just the baud rate. If you've ever configured a modem you may have run across some of these settings (open the stamp2.exe preferences window and click the Debug Port tab to see an example).

The BS2 allows us to set whether it is inverted, whether it contains 8 bits or 7 bits plus a parity bit, and the length in time of each bit. All of this information is contained in the 16-bit word called baudmode. Bits 0 -12 of this word control the time length of each bit in

#### **'TRON TIP**

The RS-232 port on computers uses inverted bits where a HIGH is approximately -12 volts and LOW is +12 volts.

**DO NOT attempt serial communications to the computer** except through the programming port of the Activity Board!



microseconds ( $\mu\text{S}$ ) and bits 13 and 14 control the word length and inversion respectively (bit 15 is only used by SEROUT to define characteristics of the output pins).

When defining the time of the bit, the BS2 adds  $20\mu\text{S}$  to whatever number is given. For a baud rate of 9600 we calculated a bit time of  $104\mu\text{S}$ . Subtracting the  $20\mu\text{S}$  from this we arrive at  $84\mu\text{S}$ , or 84 for our baudmode number as far as the bit time is concerned. Using a non-inverted transmission (bit 14 = 0) and 8-bits (bit 13 = 0) our final baudmode number would be 84.

The timeout time and timeout label are optional, but typically necessary. Without a timeout, the program would wait indefinitely until serial data arrives. If there are other processes needing regular attention, the timeout allows a means of exiting the SERIN if data doesn't arrive with a set length of time. But any data that does arrive when the BS2 is not listening for it will be lost.

Program H-7T and H-7R uses SEROUT and SERIN to transmit our button bits from the transmitter to the receiver. The programs calculate the baudmode number (called ByteTime) from a given baud rate (9600 in this case). This allows us to change the baud rate without having to do the math if we desired 2400 baud instead of 9600 (the BS2 allows baud rates from 300 to 38400).

```
'PROG_H-7T
'Asynchronous byte transmitter of P0 using SEROUT command
ByteOut var byte           'Holds outgoing byte
Baud var word              'Holds Baud / 100
ByteTime var word          'Holds microsecond length of single bits - 20
Baud = 9600 / 100
ByteTime = 10000/Baud - 20  'Calculate time length of each bit

DEBUG "Baud Rate = ", DEC Baud * 100
Loop
  Pause 3000                '3 sec delay between transmission

  DIRC = %0000              'Set up buttons as inputs
  Byteout = %11111111       'Clear out ByteOut
  Byteout.LOWNIB = INC       'Set lower nibble of byteout to button nibble
  FREQOUT 11, 200, 2000     'Sound transmit tone
  DEBUG IBIN8 ? ByteOut,CR   'Display
  SEROUT 0, ByteTime, [ByteOut] 'Transmit byteout at 9600 baud, non-inverted, 8 bit
  GOTO Loop

*****

'Prog_H-7R
'Asynchronous byte receiver on P0 using SERIN command
ByteIn var byte            'Incoming Byte
Baud var word              'Holds Baud /100
ByteTime var word          'Holds microsecond length of single bits - 20
NibIn var nib              'Lower nibble of ByteIn

Baud = 9600 / 100
ByteTime = 10000/Baud - 20 'Calculate time length of each bit

DIRC = %1111              'Set up LEDs for outputs
DEBUG "Baud Rate = ", DEC Baud * 100
Loop:
  DEBUG CR, "Waiting for Start Bit",cr 'Display waiting message
  SERIN 0, ByteTime, 2000, Timeout, [ByteIn] 'Wait until a byte comes in, or timeout, store data in byteln
                                          '(expecting 9600 baud, inverted, 8 bits)
  NibIn = ByteIn.lownib     'Set NibIn to lower nibble of byteln
  OUTC = NibIn             'Set LEDs
  DEBUG "LEDs are lower nibble of incoming byte ", IBIN8 ByteIn,CR
  GOTO Loop

Timeout:
  Debug "Timeout",CR
  GOTO Loop
```

In our examples between Activity Boards, asynchronous communications required 2 lines: data and ground. Systems which use light (such as television remote controls and fiber optic systems), or radio frequencies do not require a 'ground' line since electron current is not used in the medium.

**'TRON TIP**

In all of our examples data went from the transmitter to the receiver only. There was no communications in the other direction. This is a *simplex* communication system.

If the receiver also acted as a transmitter sending data back it would be *duplex* communications.

## Final Word

We hope you have found this manual and lab set helpful in your study of microcontrollers. Southern Illinois University's Electronics Management (ELM) program has recently incorporated the BASIC Stamp into its curriculum. This manual was originally created for use in ELM 343 Microcontroller Applications Laboratory. The manual and the labs have also served to enhance the program's Digital, Microprocessor, and Industrial courses. Applications for microcontrollers can be found in all areas of the electronics industry. It is important for today's electronics technologist to have a good understanding of microcontroller technology.

We thank SIU for allowing us to share this curriculum with other users of the BASIC Stamp. Pictures from the SIU ELM program and a brief program description are appended. If you would like any further information, visit us at the web sites below.

Southern Illinois University – [www.siu.edu](http://www.siu.edu)

Electronics Management -- [www.siu.edu/~imsasa/elm](http://www.siu.edu/~imsasa/elm)

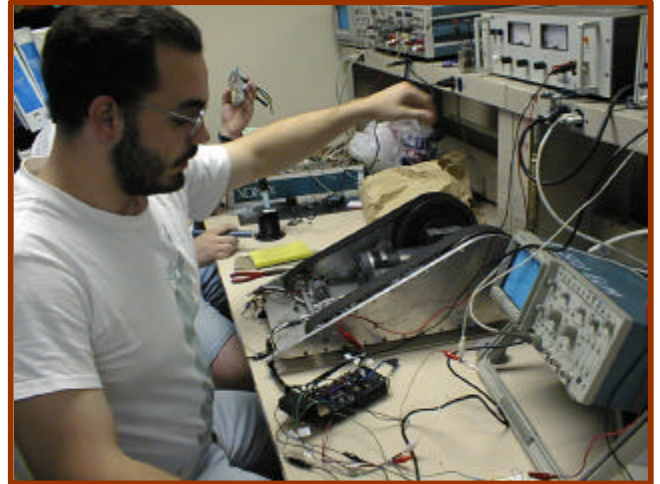
Kim and Dan are working with the BASIC Stamps in the new ELM computer lab.



You can log-in and control our dollhouse over the Net. Check it out on the "Student Project page on our WEB site.



Monty working to interface the robot with a PLC in Industrial Electronics



Steve is working on the motor drive for “Annihilator” the ELM SUMO Competition robot -- It's Microcontroller based.

The Distributed Control Group project brings management skill and a sound electronics background together. Check it out in the Student Projects area of the ELM web site.



**Electronics Management Bachelor of Science Degree  
and optional Electronics Technology Specialization  
Southern Illinois University at Carbondale  
College of Applied Science and Arts**

*The pervasive nature of electronics has resulted in the need for highly trained technologists in all sectors of business and industry.*

The Bachelor of Science in Electronics Management (ELM) is a career-oriented program designed to provide the electronics technician with the tools necessary for advancement to supervisory and managerial positions in the field of electronics. ELM graduates consistently report that the advanced technical training and managerial skills emphasized in this degree have opened many career opportunities in public, private, and military sectors of the electronics industry.

**Baccalaureate Degree Program in Electronics Management**

*The Baccalaureate Degree Program in Electronics Management is designed to provide the basic and advanced technical and managerial skills necessary to launch a successful electronics career.*

The ELM degree is a four-year 120-credit hour program designed to provide training in fundamental electronic theory and advanced managerial and technical skills. Students may enter the program at the freshmen level or build upon previous technical training received in military schools, technical institutes, and community colleges. Credit for post secondary course work, military training, and work experience will be evaluated on an individual basis. The program is ideally suited for students who possess an Associate of Applied of Science degree in Electronics Technology. The AAS degree may be coupled with SIUC 's Capstone Option which allows students to receive a baccalaureate degree in Electronic Management with the completion of an additional 60 hours of approved course work.

**Curriculum**

*The curriculum offers the flexibility to best serve the student's background and career goals.*

The ELM degree is designed as a full four-year 120-credit hour curriculum. The students must meet the 41-hour University Core Requirements and a 79-hours ELM Requirement. The first two years of the program emphasize fundamental electronics theory and technical skills. The second two years build upon these fundamental skills by providing advanced technical and managerial training. Student career objectives are enhanced by enrolling in electives courses designed to promote a working knowledge of different technical career fields.

**Electronics Technology Specialization**

*The Electronics Technology Specialization allows advanced technical course work aimed at particular sectors of the industry.*

The students in Electronics Management may choose the Electronics Technology Specialization. This specialization allows the students to select a technical curriculum focused on specific career fields within the electronics industry. Programs of study may be structured to focus on data communications technology, industrial electronics technology, biomedical equipment technology, communication technology, or microcomputer technology. The specialization requires a minimum 3 credit hour on-the-job internship. The ELM graduates can expect to compete for supervisory, managerial, or higher technical position.