StampWorks

Experiments and BASIC Stamp Source Code

Version 1.2

PARALLAX INC

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call for a Return Merchandise Authorization (RMA) number, write the number on the outside of the box and send it back to Parallax. Please include your name, telephone number, shipping address, and a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax.

14-Day Money Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the product has been altered or damaged.

Copyrights and Trademarks

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

Internet Access

We maintain internet systems for your use. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

E-mail: info@parallaxinc.com
Web: www.parallaxinc.com and www.stampsinclass.com

Internet BASIC Stamp Discussion List

We maintain two e-mail discussion lists for people interested in BASIC Stamps (subscribe at www.parallaxinc.com under the technical support button). The BASIC Stamp list server includes engineers, hobbyists, and enthusiasts. The list works like this: lots of people subscribe to the list, and then all questions and answers to the list are distributed to all subscribers. It's a fun, fast, and free way to discuss BASIC Stamp issues and get answers to technical questions. This list generates about 40 messages per day.

The Stamps in Class list is for students and educators who wish to share educational ideas. To subscribe to this list go to www.stampsinclass.com and look for the E-groups list. This list generates about 5 messages per day.

Table of Contents

Table of Contents

StampWorks Preface

Dear Friends:

There are probably as many ways to learn a new subject or skill as there are students and yet, most will agree that *learning by doing* produces the longest lasting results. And, quite frankly, learning by doing is almost always the most satisfying way to learn; it involves more of the senses. That's what this text and the StampWorks kit is all about: learning to program the BASIC Stamp by actually writing programs for it. The theory sections are short and concise. You'll learn programming theory by putting it into practice. There's not a lot of hand holding here; there's work – fun work that will teach you about microcontroller programming with the Parallax BASIC Stamp.

Why take up the challenge? Why learn to write programs for the BASIC Stamp microcontroller? The answer is simple, if not obvious: microcontrollers are everywhere. They're in our television sets, our microwave ovens and our sprinkler controllers – even our cars. The fact is that most new cars today have ten or more microcontrollers managing everything from the engine, the interior climate, wheel spin (traction control), the braking system (anti-lock braking) and many other functions. In short, today's cars are safer and more comfortable due, in large part, to the use of microcontrollers.

With microcontrollers we can build "smart" circuits and devices. In the past, we would have to change wiring or components in a circuit to modify or create a new behavior. The advantage of using a microcontroller over other approaches is that changing its program can modify the behavior of our circuit or device. The advantage of using the BASIC Stamp is that writing and modifying a program is very easy and the StampWorks kit will show you just how easy it can be.

Have fun with these projects and think about how you could apply the concepts while building each one. I appreciate your feedback anytime by e-mail to jwilliams@parallaxinc.com.

StampWorks Introduction

Getting the Most from Your StampWorks Lab

This book is divided into two major sections: the StampWorks experiments and the BASIC Stamp II manual. Throughout the use of this course, you will be moving between the two sections frequently as you work with the experiments. Additional reference materials are available from download on the StampWorks page at www.parallaxinc.com, including datasheets, updates and technical details released after this publication.

Three Steps to Success with StampWorks:

1. Read Section 1 of the BASIC Stamp II manual. This section will introduce you to the BASIC Stamp II and guide you through the installation of the programming software. Another helpful resource is Robotics chapter 1 from www.stampsinclass.com.

2. Read "Prepare your StampWorks Lab for Experiments," the next section of this manual. This section walks you through the simple steps of preparing the experiment board for the projects that follow.

3. Work your way through the experiments, referring to the BASIC Stamp Manual syntax guide as needed. This is the fun part – working with the Stamp by building simple circuits and writing code.

By the time you've worked your way through all the experiments you'll be ready to develop your own Stamp projects, from the very simple to the moderately complex. The key here is to make sure you understand everything about a particular experiment before moving on to the next.

One last reminder: Have fun!

Preparing Your StampWorks Lab

Before moving into the experiments, you need to take inventory of your kit and prepare your StampWorks lab. Once this is done, you'll be able to build a wide variety of Stamp-controlled circuits with it.

The StampWorks kit includes the following items from Parallax:

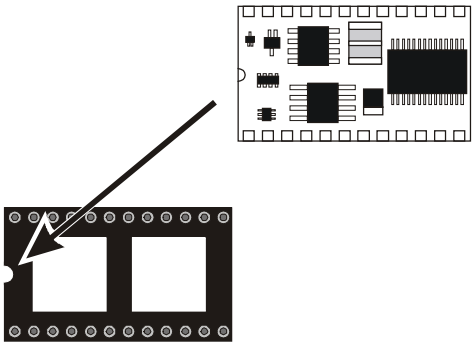
Stock Code#	Description	Quantity
28135	NX-1000 board and 2x16 LCD	1
750-00007	12V 1A wall pack power supply	1
BS2-IC	BASIC Stamp II module	1
800-00003	Serial programming cable	1
27220	StampWorks Manual	1
27000	Parallax CD-ROM	1
150-01020	1K ohm resistor, ¼ watt, 5%	4
150-01030	10K ohm resistor, ¼ watt, 5%	8
150-02210	220 ohm resistor, ¼ watt, 5%	3
150-04720	470 ohm resistor, ¼ watt, 5%	1
150-04720	4.7 k resistor, ¼ watt, 5%	2
200-01040	0.1 uF capacitor	4
201-01061	10 uF capacitor	1
201-03080	3300 uF capacitor	1
251-03230	32.768 kHz crystal	1
350-00009	Photoresistor	2
602-00009	74HC595	2
602-00010	74HC165	2
602-00015	LM358 dual op-amp	1
603-00001	MAX2719 LED display driver	1
604-00002	DS1620 digital thermometer	1
604-00005	DS1302 timekeeping chip	1
604-00009	555 timer	1
604-00020	24LC32 4K EEPROM	1
ADC0831	ADC0831 8-bit A/D converter	1
900-00001	Piezo Speaker	1
900-00005	Parallax standard servo	1
27964	12 VDC / 75 ohm stepper motor	1
451-00301	3-pin single row header	1
700-00050	22 gauge wire roll – red	1
700-00051	22 gauge wire roll – white	1
700-00052	22 gauge wire roll – black	1
28162	Digital multimeter	1
700-00065	6-piece tool set	1
700-00066	Wire cutter/stripper	1

Preparing your StampWorks Lab

To setup the StampWorks for experiments that follow, you'll need these items:

- BASIC Stamp II module
- StampWorks (INEX-1000) lab board
- 12-volt wall transformer
- Programming cable
- Red and black hookup wire
- Wire cutter/strippers

Start by removing the BASIC Stamp II module from its protective foam and carefully inserting it into the StampWorks socket. You'll notice that the BASIC Stamp II module and the StampWorks lab board socket are marked with semi-circle alignment guides. The BASIC Stamp II module should be inserted into the socket so that the alignment guides match.



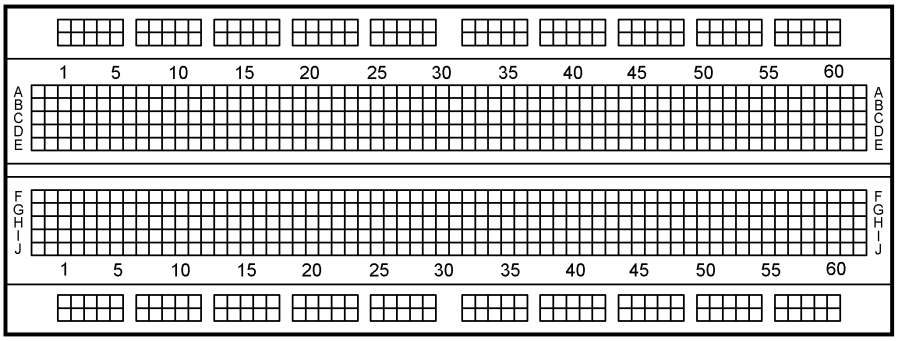
Use the programming cable to connect the StampWorks lab board to your PC. It is best to select a serial (com) port that is not already in use. If, however, you're forced to unplug another device, for example, a PDA or electronic organizer from your computer, make sure that you also disable its communication software before attempting to program your BASIC Stamp. If you haven't installed the Stamp programming software, refer to Section 1 of the Stamp II programming manual for instructions.

Ensure that the StampWorks lab board power switch is set to OFF. Connect the 2.1 mm power plug to the DC INPUT jack. Plug the 12-volt wall transformer into a suitable (120 VAC) outlet.

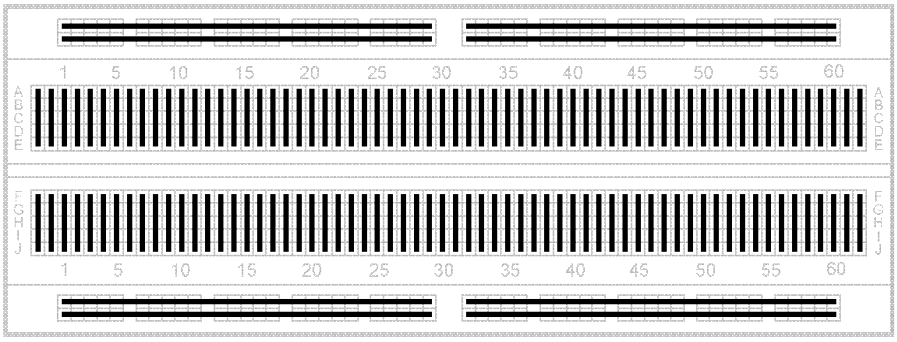
On the center portion of the breadboard is a solderless breadboard where you will build circuits that are not integral to the StampWorks lab board itself (a variety of parts are included in the StampWorks kit). It's important to understand how this breadboard works. With a little bit of preparation, it will be even easier to use with the experiments that follow.

The innermost portion of the breadboard is where we will connect our components. This section of the breadboard consists of several columns of sockets (there are numbers printed along the top for reference). For each column there are two sets of rows. The rows are labeled A through E and F through J, respectively. For any column, sockets A through E are electrically connected. The same holds true for rows F through J.

Above and below the main section of breadboard are two horizontal rows of sockets, each divided in the middle. These horizontal rows (often called "rails" or "buses") will be used to carry +5 volts (Vdd) and Ground (Vss). Our preparation of the breadboard involves connecting the rails so that they run from end-to-end, connecting the top and bottom rails together and, finally, connecting the rails to Vdd and Vss. Here's what the breadboard looks like on the outside:



If we X-Rayed the breadboard, we would see the internal connections and the breaks in the Vdd and Vss rails that need to be connected. Here's a view of the breadboard's internal connections:



Start by setting your wire stripper for 22 (0.34 mm^2) gauge. Take the spool of black wire and strip a ¼-inch (6 mm) length of insulation from the end of the wire. With your needle-nose pliers, carefully bend the bare wire 90 degrees so that it looks like this:

Now push the bare wire into the topmost (ground) rail, into the socket that is just above breadboard column 29 (this socket is just left of the middle of the breadboard, near the top). Hold the wire so that it extends to the right. Mark the insulation by lightly pinching it with the diagonal cutters at the socket above column 32. Be careful not to cut the wire.

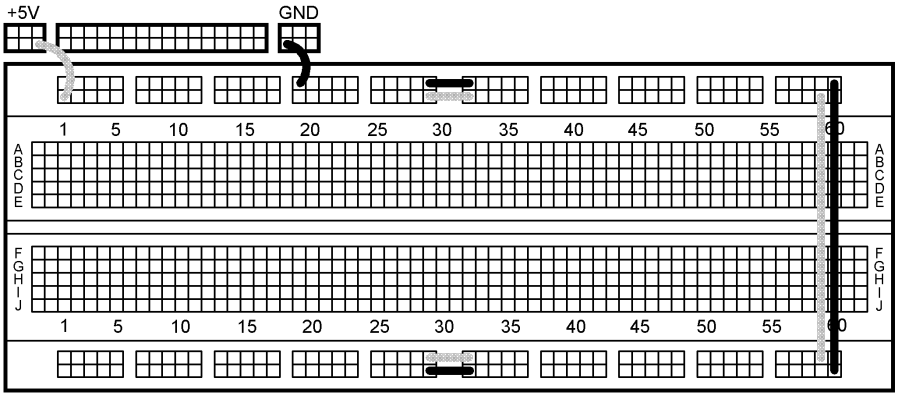
Remove the wire from the breadboard and cut it about ¼-inch (6 mm) beyond the mark you just made. With your wire strippers, remove the insulation at the mark. Now bend the second bare end 90 degrees so that the wire forms a squared "U" shape with the insulation in the middle.

If you've measured and cut carefully, this "U" shaped wire will plug comfortably into the ground rail at sockets 29 and 32. This will create a single ground rail. Repeat this process with black wire for the bottom-most rail. Then, connect the two rails together using the same process at column 60 (right-most sockets on each rail).

With the red wire, connect the top and bottom inside rail halves together. These rails will carry +5 volts, or Vdd. Connect the Vdd rails together at column 59.

Now take a 1½-inch (4 cm) section of black wire and a 1½-inch (4 cm) section of red wire and strip ¼-inch (6 mm) insulation from the ends of both. Bend each wire into a rounded "U" shape. These wires are not designed to lie flat like the other connections, making them easy to remove from the StampWorks lab board if necessary.

Carefully plug one end of the red wire into any of the terminals sockets of the +5V block (near the RESET switch) and the other end into the Vdd (+5) rail at column 1. Then, plug one end of the black wire into any of the sockets of the GND block and other end into the ground rail at column 19. BE VERY CAREFUL with these last two connections. If the Vdd and Vss rails get connected together, damage will occur when power is applied to the StampWorks lab board. When completed, your StampWorks breadboard will look like this:

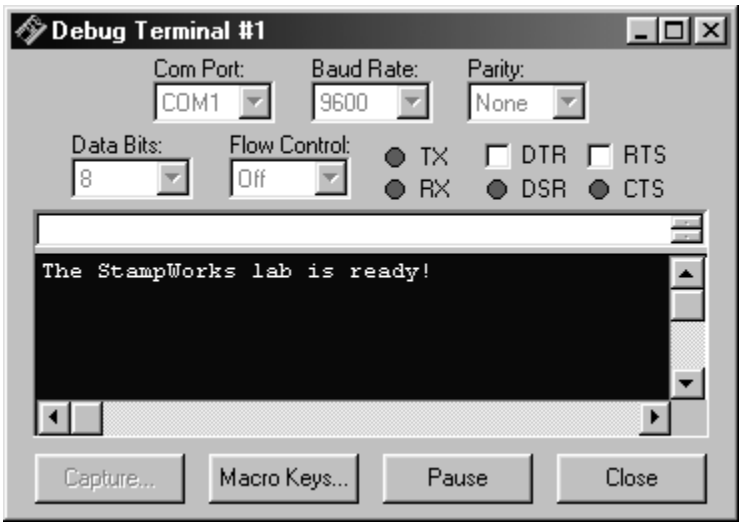


Move the StampWorks lab board power switch to ON. The green ON LED (green) should illuminate. If it doesn't, make sure that wall transformer is plugged into a live socket and that there are no wiring errors with your setup.

Start the BASIC Stamp II software editor and enter the following lines of code:

```
' {$STAMP BS2}
DEBUG "The StampWorks lab is ready!"
```

Now run the program. If all went well, the program will be downloaded to the Stamp and a **DEBUG** window will appear on screen.



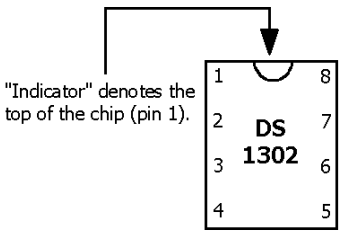
If an error occurs, check the following items:

- Is the BASIC Stamp II plugged into the NX-1000 board correctly?
- Is the StampWorks lab board power switch set to ON? Is the green ON LED lit?
- Is the programming cable connected between the PC and the StampWorks lab board?
- Have you (manually) selected the wrong PC com port?
- Is the PC com port being used by another program?

When the DEBUG window appears and tells you that the StampWorks lab is ready, it's time to talk about Stamp programming.

Connecting a Chip

There are two ways to draw a schematic. One way is considered "chip-centric" in which I/O pins appear on the chip according to their physical location. StampWorks has drawn schematics for efficiency, meaning that I/O pins are placed to make the schematic legible. I/O pins on all chips are counted according to their indicator, starting with Pin 1 and counting in a counter-clockwise direction.



StampWorks Programming Essentials

Contents of a Working Program

In Section 1 of the BASIC Stamp II manual you were introduced to the BASIC Stamp, its architecture and the concepts of variables and constants. In this section, we'll introduce the various elements of a program: linear code, branching, loops and subroutines.

The examples in this discussion use *pseudo-code* to demonstrate and describe program structure. *Italics* are used to indicate the sections of pseudo-code that require replacement with valid programming statements in order to allow the example to compile and run correctly. You need not enter any of the examples here as all of these concepts will be used in the experiments that follow.

People often think of computers and microcontrollers as "smart" devices and yet, they will do nothing without a specific set of instructions. This set of instructions is called a program. It is our job to write the program. Stamp programs are written in a programming language called PBASIC, a Parallax-specific version of the BASIC (Beginners All-purpose Symbolic Instruction Code) programming language. BASIC is very popular because of its simplicity and English-like syntax.

A working program can be as simple as a list of statements. Like this:

```
statement 1
statement 2
statement 3
END
```

This is a very simple, yet valid program structure. What you'll find, however, is that most programs do not run in a straight, linear fashion like the listing above. Program flow is often redirected with branching, looping and subroutines, with short linear sections in between. The requirements for program flow are determined by the goal of the program and the conditions under which the program is running.

Branching – Redirecting the Flow of a Program

A branching command is one that causes the flow of the program to change from its linear path. In other words, when the program encounters a branching command, it will, in almost all cases, not be running the next [linear] line of code. The program will usually go somewhere else. There are two categories of branching commands: *unconditional* and *conditional*. PBASIC has two commands, **GOTO** and **GOSUB** that cause unconditional branching.

Here's an example of an unconditional branch using **GOTO**:

```
Label:
  statement 1
  statement 2
  statement 3
  GOTO Label
```

We call this an *unconditional* branch because it always happens. **GOTO** redirects the program to another location. The location is specified as part of the **GOTO** command and is called an address. Remember that addresses start a line of code and are followed by a colon (:). You'll frequently see **GOTO** at the end of the main body of code, forcing the program statements to run again.

Conditional branching will cause the program flow to change under a specific set of circumstances. The simplest conditional branching is done with **IF-THEN** construct. The PBASIC **IF-THEN** construct is different from other flavors of BASIC. In PBASIC, **THEN** is always followed by a valid program address (other BASICs allow a variety of programming statements to follow **THEN**). If the condition statement evaluates as TRUE, the program will branch to the address specified. Otherwise, it will continue with the next line of code.

Take a look at this listing:

```
Start:
  statement 1
  statement 2
  statement 3
  IF (condition) THEN Start
```

The statements will be run and then the condition is tested. If it evaluates as TRUE, the program will branch back to the line called `Start`. If the condition evaluates as FALSE, the program will continue at the line that follows the **IF-THEN** construct.

As your requirements become more sophisticated, you'll find that you'll want your program to branch to any number of locations based on a condition. One approach is to use multiple **IF-THEN** constructs.

```
IF (condition_0) THEN Label_0
IF (condition_1) THEN Label_1
IF (condition_2) THEN Label_2
```

This approach is valid and does get used. Thankfully, PBASIC has a special command, **BRANCH**, that allows a program to jump to any number of addresses based on the value of a variable. This is very handy because the conditions we've referred to in the text are often checking the value of a control variable. **BRANCH** is a little more complicated in its setup, but very powerful in that it can replace multiple **IF-THEN** statements. **BRANCH** requires a control variable and a list of addresses

In the case of a single control variable, the previous listing can be replaced with one line of code:

```
BRANCH controlVar, [Label_0, Label_1, Label_2]
```

When **controlVar** is zero, the program will branch to **Label_0**, when **controlVar** is one the program will branch to **Label_1** and so on.

Looping – Running Code Again and Again

Looping causes sections of the program to be repeated. Looping often uses unconditional and conditional branching to create the various looping structures. Here's an example of *unconditional looping*:

```
Label:
  statement 1
  statement 2
  statement 3
  GOTO Label
```

By using **GOTO** the statements are unconditionally repeated, or looped. By using **IF-THEN**, we can add a conditional statement to the loop. The next few examples are called *conditional looping*. The loops will run under specific conditions. Conditional programming is what gives microcontrollers their "smarts."

```
Label:
  statement 1
  statement 2
  statement 3
  IF (condition) THEN Label
```

With this loop structure, statements will be run so long as the condition evaluates as TRUE. When the condition is evaluated as FALSE, the program will continue at the line following the **IF-THEN** statement. It's important to note that in the previous listing the statements will always run at least once, even if the condition is FALSE.

To prevent this from taking place, you need to test the condition before running the statements. The code can be written as follows so that the statements (1 – 3) will only run when the condition is TRUE. When the condition evaluates as FALSE, the program continues at **Label_2**.

```
Label_1:
  IF NOT (condition) THEN Label_2
  statement 1
  statement 2
  statement 3
  GOTO Label_1

Label_2:
  statement 4
```

The final example of conditional looping is the programmed loop using the **FOR-NEXT** construct.

```
FOR controlVar = startVal TO endVal STEP stepSize
  statement 1
  statement 2
  statement 3
NEXT
```

The **FOR-NEXT** construct is used to cause a section of code to execute (loop) a specific number of times. **FOR-NEXT** uses a control variable to determine the number of loops. The size of the variable will determine the upper limit of loop iterations. For example, the upper limit when using a byte-sized control variable would be 255.

The **STEP** option of **FOR-NEXT** is used when the loop needs to count increments other than one. If, for example, the loop needed to count even numbers, the code would look something like this:

```
FOR controlVar = 2 TO 20 STEP 2
  statement 1
  statement 2
  statement 3
NEXT
```

Subroutines – Reusable Code that Saves Program Space

The final programming concept we'll discuss is the subroutine. A subroutine is a section of code that can be called (run) from anywhere in the program. **GOSUB** is used to redirect the program to the subroutine code. The subroutine is terminated with the **RETURN** command. **RETURN** causes the program to jump back to the line of code that follows the calling **GOSUB** command.

```
Start:
  GOSUB MySub
  PAUSE 1000
  GOTO Start

MySub:
  statement 1
  statement 2
  statement 3
  RETURN
```

In this example, the code in the **MySub** is executed and then the program jumps back to the line **PAUSE 1000**.

The Elements of PBASIC Style

Like most versions of the BASIC programming language, PBASIC is very forgiving and the compiler enforces no particular formatting style. So long as the source code is syntactically correct, it will compile and download to the Stamp without trouble.

Why, then, would one suggest a specific style for PBASIC? Consider this: Over two million BASIC Stamps have been sold and there are nearly 2500 members of the BASIC Stamp mailing list (on Yahoo! Groups). This makes it highly likely that you'll be sharing your PBASIC code with someone, if not co-developing a BASIC Stamp-oriented project. Writing code in an organized, predictable manner will save you – and your potential colleagues – time; in analysis, in troubleshooting and especially when you return to a project after a long break.

The style guidelines presented here are just that: guidelines. They have been developed from style guidelines used by professional programmers using other high-level languages such as Java™, C/C++ and Visual Basic®. Use these guidelines as is, or modify them to suit your needs. The key is selecting a style the works well for you or your organization and sticking to it.

1. Do It Right The First Time

Many programmers, especially new ones, fall into the "I'll slug it out now and fix it later." trap. Invariably, the "fix it later" part never seems to happen and sloppy code makes its way into production projects. If you don't have time to do it right, when will you have time to do it again?

Start clean and you'll be less likely to introduce errors in your code. And if errors do pop up, clean formatting will make them easier to find and fix.

2. Be Organized and Consistent

Using a blank program template will help you organize your programs and establish a consistent presentation.

3. Use Meaningful Names

Be verbose when naming constants, variables and program labels. The compiler will allow names up to 32 characters long. Using meaningful names will reduce the number of comments and make your programs easier to read, debug and maintain.

4. Naming Constants

Begin constant names with an uppercase letter and use mixed case, using uppercase letters at the beginning of new words within the name:

```
AlarmCode       CON   25
```

5. Naming Variables

Begin variable names with a lowercase letter and use mixed case, using uppercase letters at the beginning of new words within the name. Avoid the use of internal variable names (such as B0 or W1):

```
waterLevel      VAR   Word
```

6. Naming Program Labels

Begin program labels with an uppercase letter, used mixed case, separate words within the label with an underscore character and begin new words with an uppercase letter. Labels should be preceded by at least one blank line, begin in column 1 and be terminated with a colon (except after GOTO and THEN where they appear at the end of the line and without a colon):

```
Print_String:
  READ eeAddr, char
  IF (char = 0) THEN Print_String_Exit
  DEBUG char
  eeAddr = eeAddr + 1
  GOTO Print_String

Print_String_Exit:
  RETURN
```

7. PBASIC Keywords

All PBASIC language keywords, including VAR, CON and serial/debugging format modifiers (DEC, HEX, BIN) should be uppercase:

```
Main:
  DEBUG "BASIC Stamp", CR
  END
```

8. Variable Types

Variable types should be be in mixed case and start with an uppercase letter:

```
status          VAR   Bit
counter         VAR   Nib
ovenTemp        VAR   Byte
rcValue         VAR   Word
```

9. Indent Nested Code

Nesting blocks of code improves readability and helps reduce the introduction of errors. Indenting each level with two spaces is recommended to make the code readable without taking up too much space:

```
Main:
..FOR outerLoop = 1 TO 10
....FOR innerLoop = 1 TO 10
......DEBUG DEC outerLoop, TAB, DEC innerLoop, TAB
......DEBUG DEC (outerLoop * innerLoop)
......PAUSE 100
....NEXT
..NEXT
```

Note: The dots are used to illustrate the level of nesting and are not a part of the code.

10. Be Generous With Whitespace

Whitespace (spaces and blank lines) has no effect compiler or BASIC Stamp performance, so be generous with it to make listings easier to read. As suggested in #6 above, allow at lease one blank line before program labels (two blanks lines before a subroutine label is recommended). Separate items in a parameter list with a space:

```
Main:
  BRANCH task, [Update_Motors, Scan_IR, Close_Gripper]
  GOTO Main

Update_Motors:
  PULSOUT leftMotor, leftSpeed
  PULSOUT rightMotor, rightSpeed
  PAUSE 20
  Task = (task + 1) // NumTasks
  GOTO Main
```

An exception to this guideline is with the bits parameter used with SHIFTIN and SHIFTOUT. In this case, format without spaces:

```
  SHIFTIN A2Ddata, A2Dclock, MSBPost, [result\9]
```

11. IF-THEN Conditions

Enclose IF-THEN condition statements in parenthesis:

```
Check_Temp:
  IF (indoorTemp >= setPoint) THEN AC_On
```

The StampWorks files (available for download fromwww.parallaxinc.com) include a blank programming tempalate (Blank.BS2) that will help you get started writing organized code. It's up to you to follow the rest of the guidelines above – or develop and use guidelines of your own.

Time to Experiment

Learn the Programming Concepts

What follows is a series of programming experiments that you can build and run with your StampWorks lab. The purpose of these experiments is to teach programming concepts and the use of external components with the BASIC Stamp. The experiments are focused and designed so that as you gain experience, you can combine the individual concepts to produce sophisticated programs.

Building the Projects

This section of the manual is simple but important because you will learn important programming lessons and construction techniques using your StampWorks lab. As you move through the rest of the manual, construction details will not be included (you'll be experienced by then and can make your own choices) and the discussion of the program will be less verbose, focusing specifically on special techniques or external devices connected to the BASIC Stamp.

What to do Between Projects

The circuit from one project may not be electrically compatible with another and could, in some cases, cause damage to the BASIC Stamp if the old program is run with the new circuit. For this reason, a blank program should be downloaded to the Stamp before connecting the new circuit. This will protect the Stamp by resetting the I/O lines to inputs. Here's a simple, two-line program that will clear and reset the Stamp.

```
' {$STAMP BS2}
DEBUG "Stamp clear."
```

For convenience, save this program to a file called CLEAR.BS2.

Experiment #1: Flash An LED

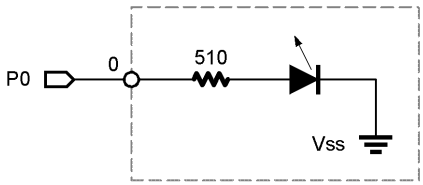
The purpose of this experiment is to flash an LED with the BASIC Stamp. Flashing LEDs are often used as alarm indicators.

New PBASIC Elements/Commands:

- CON
- HIGH
- LOW
- PAUSE
- GOTO

Building The Circuit

All StampWorks experiments use a dashed line to show parts that are already on the NX-1000 board. The LED is available on the "LED MONITOR 16 CHANNELS" part of the board.



Since the StampWorks lab board has the LEDs built in, all you have to do is connect one to the BASIC Stamp.

1. Start with a six-inch (15 cm) white wire. Strip ¼-inch (6 mm) of insulation from each end.
2. Plug one end into BASIC Stamp Port 0.
3. Plug the other end into LED Monitor Channel 0

Experiment #1: Flash an LED

```
'=========================================================================================
'
'   File..... Ex01 - Blink.BS2
'   Purpose... LED Blinker
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =======================================================================================


' ---------------------------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------------------------

' Blinks an LED connected to P0


' ---------------------------------------------------------------------------------------
' I/O Definitions
' ---------------------------------------------------------------------------------------

LEDpin          CON     0                       ' LED connected to Pin 0


' ---------------------------------------------------------------------------------------
' Constants
' ---------------------------------------------------------------------------------------

DelayTime       CON     500                     ' delay time in milliseconds


' ---------------------------------------------------------------------------------------
' Program Code
' ---------------------------------------------------------------------------------------

Main:
  HIGH LEDpin                                   ' turn LED on
  PAUSE DelayTime                               ' pause for a bit
  LOW LEDpin                                    ' turn LED off
  PAUSE DelayTime                               ' pause while off
  GOTO Main                                     ' do it again

  END
```

Behind The Scenes

Each of the Stamp's I/O pins has three bits associated with its control. A bit in the **Dirs** word determines whether the pin is an input (bit = 0) or an output (bit = 1). If the pin is configured as an output, the current state of the pin is stored in the associated bit in the **Outs** word. If the pin is configured as an input, the current pin value is taken from the associated bit in the **Ins** word.

HIGH and **LOW** actually perform two functions with one command: the selected pin is configured as an output and the value is set in the **Outs** word (1 for **HIGH**, 0 for **LOW**).

For example, this line of code:

```
HIGH 0
```

performs the same function as:

```
Dir0 = 1                                    ' make Pin 0 an output
Out0 = 1                                    ' set Pin 0 high
```

Experiment #2: Flash An LED (Version 2)

The purpose of this experiment is to flash an LED with the BASIC Stamp. The method in this experiment adds flexibility to the LED control.

New PBASIC elements/commands to know:

- VAR
- Out0 – Out15
- Dir0 – Dir15
- Byte
- Bit0 – Bit15

Building The Circuit.

Use the same circuit as in Experiment #1.

```
' =============================================================================
'
'   File..... Ex02 - Blink2.BS2
'   Purpose... LED Blinker - Version 2
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Blinks an LED connected to Pin 0.  LED on-time and off-time can be set
' independently of each other.
```

Experiment #2: Flash an LED (Version 2)

```
' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

MyLED           VAR     Out0                    ' LED connected to Pin 0


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DelayOn         CON     1000                    ' on-time time in milliseconds
DelayOff        CON     250                     ' off-time in milliseconds

On              CON     1
Off             CON     0


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  Dir0 = %1                                     ' make LED pin an output


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  MyLED = On
  PAUSE DelayOn                                 ' pause for "on" time
  MyLED = Off
  PAUSE DelayOff                                ' pause for "off" time
  GOTO Main                                     ' do it again

  END
```

Can you explain what's going on?

Since **MyLED** is a bit-sized variable, Bit0 of **cntr** will control it. It works like this: When **cntr** is odd (1, 3, 5, etc.), **Bit0** will be set (1), causing the LED to light. When **cntr** is an even number, **Bit0** will be clear (0), causing the LED to be off.

Experiment #3: Display a Counter with LEDs

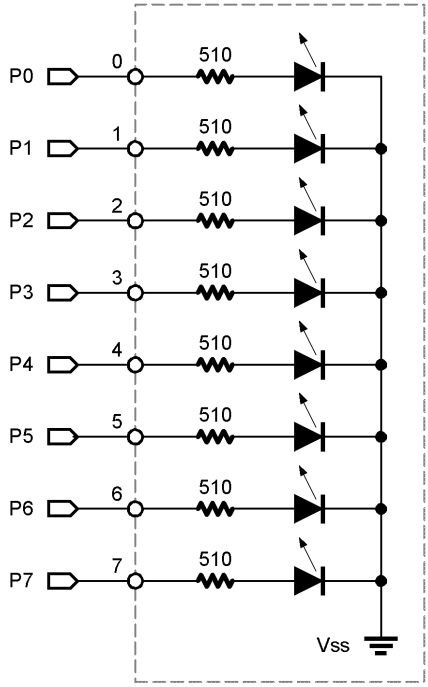
The purpose of this experiment is to display a byte-sized value with LEDs. Multiple LEDs are frequently used as complex status or value indicators.

New PBASIC elements/commands to know:

- `OutL, OutH`
- `DirL, DirH`
- `FOR-NEXT`

Building The Circuit.

These LEDs are denoted by the "LED MONITOR 16 CHANNELS" notation on the NX-1000 board.



Experiment #3: Display a Counter with LEDs

Since the StampWorks lab board has the LEDs built in, all you have to do is connect one to the BASIC Stamp.

1. Start with eight, six-inch (15 cm) white wires. Strip ¼-inch (6 mm) of insulation from the ends of each.
2. Plug one end of a wire into BASIC Stamp Port 0.
3. Plug the other end into LED Monitor Channel 0.
4. Repeat Steps 2 and 3 for LED Monitor Channels 1-7 (Stamp pins 1– 7) using more wire.

```
' =============================================================================
'
'   File..... Ex03 - LED Counter.BS2
'   Purpose... Binary Counter
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Displays a binary counter on Pins 0 - 7


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

LEDs            VAR     OutL                    ' LEDs on Pins 0 - 7


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

MinCount        CON     0                       ' counter start value
MaxCount        CON     255                     ' counter end value
DelayTime       CON     100                     ' delay time in milliseconds
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

counter         VAR     Byte


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111111                              ' make pins 0 - 7 outputs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR counter = MinCount TO MaxCount            ' loop through all count values
    LEDs = counter                              ' show count on LEDs
    PAUSE DelayTime                             ' pause before next number
  NEXT
  GOTO Main                                     ' do it again

  END
```

Behind The Scenes

As explained in Experiment #1, the state of the BASIC Stamp's output pins are stored in a memory area called **`Outs`** (**`OutL`** is the lower byte of the **`Outs`** word). Since **`OutL`** is part of the BASIC Stamp's general-purpose (RAM) memory, values can be written to and read from it. In this case, copying the value of our counter to **`OutL`** (alias for **`LEDs`**) causes the value of the counter to be displayed on the StampWorks LEDs.

Challenge

Modify the program to count backward.

Experiment #4: Science Fiction LED Display

The purpose of this experiment is to "ping-pong" across eight LEDs to create a Sci-Fi type display. Circuits like this often are used in film and television props.

New PBASIC elements/commands to know:

- << (Shift Left operator)
- >> (Shift Right operator)
- IF-THEN

Building The Circuit

Use the same circuit as in Experiment #3.

```
' =====================================================================================
'
'   File...... Ex04 - Ping Pong.BS2
'   Purpose... Ping-Pong LED Display
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =====================================================================================


' -------------------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------------------

' "Ping-Pongs" an LED (one of eight).


' -------------------------------------------------------------------------------------
' I/O Definitions
' -------------------------------------------------------------------------------------

LEDs            VAR     OutL                    ' LEDs on Pins 0 - 7
```

```
' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DelayTime       CON     100                     ' delay time in milliseconds


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111111                              ' make all pins outputs
  LEDs = %00000001                              ' start with one LED on (pin 0)


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Go_Forward:
  PAUSE DelayTime                               ' show the LED
  LEDs = LEDs << 1                              ' shift lit LED to the left
  IF (LEDs = %10000000) THEN Go_Reverse         ' test for final position
  GOTO GoForward                                ' continue in this direction

Go_Reverse:
  PAUSE DelayTime                               ' show the LED
  LEDs = LEDs >> 1                              ' shift lit LED to the right
  IF (LEDs = %00000001) THEN Go_Forward         ' test for final position
  GOTO GoReverse                                ' continue in this direction

  END
```

Behind The Scenes

This project demonstrates the ability to directly manipulate the BASIC Stamp's outputs. The program initializes the LEDs to %00000001 (LED 0 is on) then uses the shift-left operator (**<<**) to move the lit LED one position to the left. With binary numbers, shifting left by one is the same as multiplying by two. Shifting right by one (>>) is the same as dividing by two.

Both major sections of the code use **IF-THEN** to test for the limits of the display, causing the program to branch to the other section when a limit is reached.

Experiment #5: LED Graph (Dot or Bar)

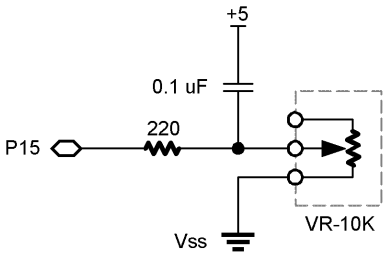
The purpose of this experiment is to create a configurable (dot or bar) LED graph. This type of graph is very common on audio equipment, specifically for VU (volume) meters. The value for the graph in the experiment will be taken from the position of a potentiometer.

New PBASIC elements/commands to know:

- `Word`
- `RCTIME`
- `*/ (Star-Slash operator)`
- `GOSUB-RETURN`
- `DCD`

Building The Circuit

Add this circuit to Experiment #4.



Experiment #5: LED Graph (Dot or Bar)

1. Using red wire (cut as required), connect the Vdd (+5) rail to socket A15.
2. Plug a 0.1 uF (104K) capacitor into sockets B14 and B15.
3. Plug a 220-ohm (RED-RED-BROWN) resistor into sockets C10 and C14.
4. Using white wire, connect socket A10 to BASIC Stamp Port 15.
5. Using white wire, connect socket E14 to the wiper of the 10K potentiometer
6. Using black wire, connect the Vss (ground) rail to the bottom terminal of the 10K potentiometer.

```
' =============================================================================
'
'   File...... Ex05 - LED Graph.BS2
'   Purpose... LED Bar Graph
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated...
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Displays a linear (bar) or dot graph using 8 LEDs


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

LEDs            VAR     OutL                    ' LED outputs
PotPin          CON     15                      ' pot wiper connects to pin 15


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DotGraf         CON     0                       ' define graph types
BarGraf         CON     1
GraphMode       CON     BarGraf                 ' define current graph mode
```

```
On              CON     1
Off             CON     0

Scale           CON     $005F                   ' scale value to make 0 .. 255

' Scale         CON     $0028                   ' scale for BS2sx
' Scale         CON     $0027                   ' sclae for BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

rawValue        VAR     Word                    ' raw value from pot
grafValue       VAR     Byte                    ' graph value
bits            VAR     Byte                    ' highest lighted bit
newBar          VAR     Byte                    ' workspace for bar graph


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111111                              ' make low pints outputs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  HIGH PotPin                                   ' discharge cap
  PAUSE 1                                       '   for 1 millisecond
  RCTIME PotPin, 1, rawValue                    ' read the Pot

  grafValue = rawValue */ Scale                 ' scale grafVal (0 - 255)

  GOSUB Show_Graph                              ' show it
  PAUSE 50
  GOTO Main                                     ' do it again

  END


' -----------------------------------------------------------------------------
' Subroutines
```

```
' -----------------------------------------------------------------------------

Show_Graph:
  IF (GraphMode = BarGraf) THEN Show_Bar        ' jump to graph mode code

Show_Dot:
  LEDs = DCD (grafValue / 32)                   ' show dot value
  RETURN

Show_Bar:
  bits = DCD (grafValue / 32)                   ' get highest bit
  newBar = 0

Build_Bar:
  IF (bits = 0) THEN Bar_Done                   ' all bar LEDs lit?
  newBar = newBar << 1                          ' no - shift left
  newBar.Bit0 = On                              ' light low end
  bits = bits >> 1                              ' mark bit lit
  GOTO Build_Bar                                ' continue

Bar_Done:
  LEDs = newBar                                 ' output new level
  RETURN
```

Behind The Scenes

After initializing the outputs, this program reads the 10K potentiometer (located on the StampWorks lab board) with **`RCTIME`**. Using **`DEBUG`** to display the raw value, it was determined that **`RCTIME`** returned values between zero (pot fully counter-clockwise) and 685 (pot turned fully clockwise). Since **`grafVal`** is a byte-sized variable, **`rawVal`** must be scaled down to fit.

To determine the scaling multiplier, divide 255 (largest possible value for **`grafVal`**) by 685 (highest value returned in **`rawVal`**). The result is 0.372.

Dealing with fractional values within PBASIC's integer math system is made possible with the **`*/`** (star-slash) operator. The parameter for **`*/`** is a 16-bit (word) variable. The upper eight bits (high byte) are multiplied as a whole value. The lower eight bits (low byte) are multiplied as a fractional value.

To determine the value of the fractional byte, multiply the desired decimal fractional value by 255 and convert to hex.

Example:

0.372 x 255 = 95 (or $5F)

Since the multiplier in the experiment is 0.372, the */ value is $005F.

The program uses the **DCD** operator to determine highest lighted bit value from **grafVal**. With eight LEDs in the graph, **grafVal** is divided by 32, forcing the result of **DCD** to output values from %00000001 (**DCD 0**) to %10000000 (**DCD 7**).

In Dot mode, this is all that is required and a single LED is lit. In Bar Mode, the lower LEDs must be filled in. This is accomplished by a loop. The control value for the loop is the variable, **bits,** which also calculated using **DCD**. In this loop, **bits** will be tested for zero to exit, so each iteration through the loop will decrement (decrease) this value.

If **bits** is greater than zero, the bar graph workspace variable, **newBar**, is shifted left and its bit 0 is set. For example, if DCD returned %1000 in **bits**, here's how **bits** and **newBar** would be affected through the loop:

bits	**newBar**
1000	0001
0100	0011
0010	0111
0001	1111
0000	(done - exit loop and display value)

The purpose for the variable, **newBar**, is to prevent the LEDs from flashing with each update. This allows the program to start with an "empty" graph and build to the current value. With this technique, the program does not have to remember the value of the previous graph.

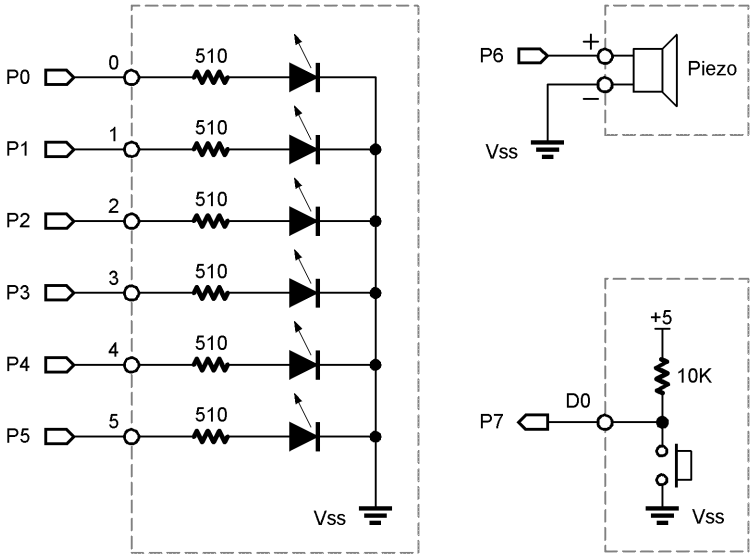
Experiment #6: A Simple Game

The purpose of this experiment is to create a simple, slot machine type game with the BASIC Stamp.

New PBASIC elements/commands to know:

- RANDOM
- & (And operator)
- FREQOUT
- BUTTON
- LOOKUP

Building The Circuit



Note: Later versions of the StampWorks lab board come with a built-in audio amplifier. Attach an 8-ohm speaker to the output of the amplifier to get the best sound from this project.

Experiment #6: A Simple Game

You may wish to substitute the piezo speaker on the StampWorks lab board with the one in the kit, which seems to have a higher volume.

1. Using white wires, connect BASIC Stamp Ports 0 – 5 to LEDs 0 – 5.
2. Using white wire, connect BASIC Stamp Port 6 to the + side of the Piezo speaker.
3. Using black wire, connect the – side of the Piezo speaker to ground.
4. Using a white wire connect BASIC Stamp Port 7 to Pushbutton D0.

```
' =============================================================================
'
'   File...... Ex06 - Las Vegas.BS2
'   Purpose... Stamp Game
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Stamp-based slot machine game that uses lights and sound.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

LEDs            VAR     OutL                    ' LED outputs
Speaker         CON     6                       ' speaker output
PlayBtn         CON     7                       ' button input to play game


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

randW           VAR     Word                    ' random number
pattern         VAR     Byte                    ' light pattern
tone            VAR     Word                    ' tone output
```

```
swData          VAR     Byte                    ' workspace variable for BUTTON
delay           VAR     Word                    ' delay while "spinning"
spin1           VAR     Byte                    ' loop counter
spin2           VAR     Byte                    ' loop counter


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %00111111                              ' make LEDs outputs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Get_Random                              ' get a random number and tone
  FREQOUT Speaker,35,tone                       ' sound the tone
  PAUSE 100
  BUTTON PlayBtn, 0, 255, 10, swData, 1, Spin   ' check for play
  GOTO Main

Spin:
  LEDs = %00111111                              ' simulate machine reset
  PAUSE 750
  LEDs = %00000000
  PAUSE 500
  delay = 75                                    ' initialize delay

  FOR spin1 = 1 TO 25                           ' spin the wheel
    GOSUB Get_Random                            ' get random number
    FREQOUT Speaker, 25, 425                    ' wheel click
    PAUSE delay                                 ' pause between clicks
    delay = delay */ $0119                      ' multiply delay by 1.1
  NEXT

  IF pattern = %00111111 THEN You_Win           ' if all lit, you win
  FREQOUT Speaker, 1000, 150                    ' otherwise, groan...
  LEDs = %00000000                              ' clear LEDs
  PAUSE 1000
  GOTO Main                                     ' do it again

You_Win:                                        ' winning lights/sound display
  FOR spin1 = 1 TO 5
```

```
    FOR spin2 = 0 TO 3
      LOOKUP spin2, [$00, $0C, $12, $21], LEDs
      LOOKUP spin2, [665, 795, 995, 1320], tone
      FREQOUT Speaker, 35, tone
      PAUSE 65
    NEXT
  NEXT

  LEDs = %00000000                              ' clear LEDs
  PAUSE 1000
  GOTO Main                                     ' do it again

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Get_Random:
  RANDOM randW                                  ' get pseudo-random number
  tone = randW & $7FF                           ' don't let tone go too high
  pattern = randW & %00111111                   ' mask out unused bits
  LEDs = pattern                                ' show the pattern
  RETURN
```

Behind The Scenes

This program demonstrates how to put more randomness into the pseudo-random nature of the **RANDOM** command. Adding a human element does it.

The program waits in a loop called **Attention**. The top of this loop calls **Get_Random** to create a pseudo-random value, a tone for the speaker and to put the new pattern on the LEDs. On returning to **Attention**, the tone is played and the button is checked for a press. The program will loop through **Attention** until you press the button.

The **BUTTON** command is used to debounce the input. Here's what gives the program its randomness: the time variations between button presses. When the button is pressed, the LEDs are lit and cleared to simulate the game resetting. Then, a **FOR-NEXT** loop is used to simulate the rolling action of a slot machine. For each roll, a "click" sound is generated and the delay between clicks is modified to simulate natural decay (slowing) of the wheel speed.

If all six LEDs are lit after the last spin, the program branches to **`You_Win`**. This routine uses **`LOOKUP`** to play a preset pattern of LEDs and tones before returning to the top of the program. If any of the LEDs is not lit, a groan will be heard from the speaker and the game will restart.

Challenge

Modify the game so that less than six LEDs have to light to for a win.

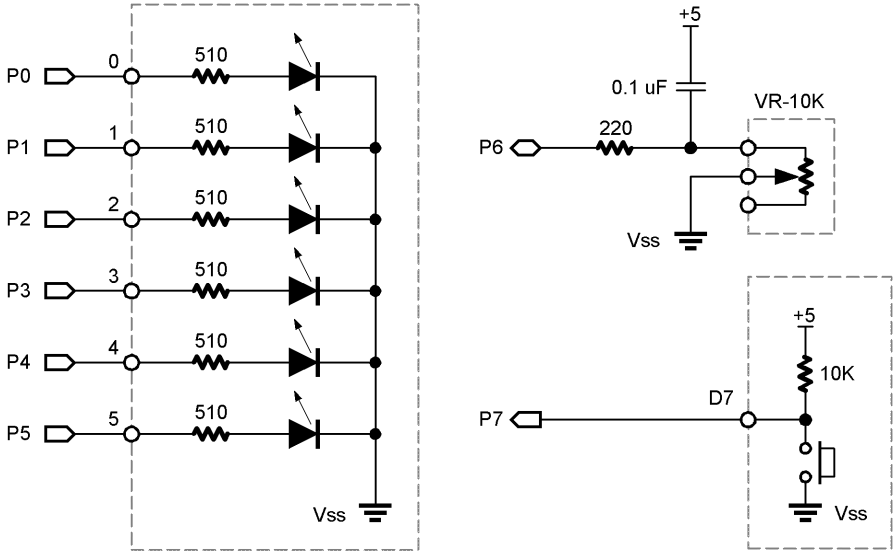
Experiment #7: A Lighting Controller

The purpose of this experiment is to create a small lighting controller, suitable for holiday trees and outdoor decorations. The outputs of this circuit will be LEDs only (To control high-voltage lighting take a look at Matt Gilliland's Microcontroller Application Cookbook).

New PBASIC elements/commands to know:

- DATA
- MIN
- // (Modulus operator)
- BRANCH

Building The Circuit.



Experiment #7: A Lighting Controller

1. Using white wires, connect BASIC Stamp Ports 0–5 to LEDs 0– 5.
2. Using red wire, connect the Vdd (+5) rail to socket A15.
3. Plug a 0.1 uF (104K) capacitor into sockets B14 and B15.
4. Plug a 220-ohm (RED-RED-BROWN) resistor into sockets C10 and C14.
5. Using white wire, connect socket A10 to BASIC Stamp Port 6.
6. Using white wire, connect socket E14 to the top terminal of the 10K potentiometer.
7. Using black wire, connect the Vss (ground) rail to the wiper (middle terminal) of the 10K potentiometer.
8. Using a white wire connect BASIC Stamp Port 7 to Pushbutton D7.

```
' =============================================================================
'
'   File..... Ex07 - Light Show.BS2
'   Purpose... Simple lighting controller
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Mini light show controller with variable speed and multiple patterns.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Select          CON     7                       ' pattern select input
PotPin          CON     6                       ' speed control Pot input
Lights          VAR     OutL                    ' light control outputs


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Scale           CON     $018A                   ' convert pot input to 0 - 1000
```

```
' Scale         CON     $00A0                   ' scale for BS2sx
' Scale         CON     $009E                   ' scale for BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

delay           VAR     Word                    ' pause time between patterns
btnVar          VAR     Byte                    ' workspace for BUTTON
mode            VAR     Byte                    ' selected mode
offset          VAR     Byte                    ' offset into light patterns
randW           VAR     Word                    ' workspace for RANDOM


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

SeqA            DATA    %000001, %000010, %000100, %001000, %010000, %100000
SeqB            DATA    %100000, %010000, %001000, %000100, %000010
                DATA    %000001, %000010, %000100, %001000, %010000
SeqC            DATA    %000000, %001100, %010010, %100001
SeqD            DATA    %100100, %010010, %001001
SeqE            DATA    %0

AMax            CON     SeqB - SeqA             ' calculate length of sequence
BMax            CON     SeqC - SeqB
CMax            CON     SeqD - SeqC
DMax            CON     SeqE - SeqD


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %00111111                              ' LED control lines are outputs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  HIGH PotPin                                   ' discharge cap
  PAUSE 1
```

```
  RCTIME PotPin, 1, delay                         ' read speed pot
  delay = (delay */ Scale) MIN 50                 ' calculate delay (50 ms ~ 1 sec)
  PAUSE delay                                     ' wait between patterns

Switch_Check:
  BUTTON Select, 0, 255, 0, btnVar, 0, Show       ' new mode?
  mode = mode + 1 // 5                            ' yes, update mode var

Show:
  BRANCH mode, [ModeA, ModeB, ModeC, ModeD, ModeE]
  GOTO Main


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

ModeA:
  offset = offset + 1 // AMax                     ' update offset (0 - 5)
  READ (SeqA + offset), Lights                    ' output new light pattern
  GOTO Main                                       ' repeat

ModeB:
  offset = offset + 1 // BMax
  READ (SeqB + offset), Lights
  GOTO Main

ModeC:
  offset = offset + 1 // CMax
  READ (SeqC + offset), Lights
  GOTO Main

ModeD:
  offset = offset + 1 // DMax
  READ (SeqD + offset), Lights
  GOTO Main

ModeE:
  RANDOM randW                                    ' get random number
  Lights = randW & %00111111                      ' light random channels
  GOTO Main
```

Behind The Scenes

Overall, this program is simpler than it first appears. The main body of the program is a loop. Timing through the main loop is controlled by the position of the potentiometer. **RCTIME** is used to read the

pot and during development the maximum pot reading was found to be 648. Multiplying the maximum pot value by 1.54 (delay */ $018A) scales the maximum value to 1000 for a one-second delay. The **MIN** operator is used in the delay scaling calculation to ensure the shortest loop-timing delay is 50 milliseconds.

The code at **Switch_Check** looks to see if button D7 is pressed. If it is, the variable, **mode**, is incremented (increased by 1). The modulus (//) operator is used to keep **mode** in the range of zero to four. This works because the modulus operator returns the remainder after a division. Since any number divided by itself will return a remainder of zero, using modulus in this manner causes **mode** to "wrap-around" from four to zero.

The final element of the main loop is called **Show**. This code uses **BRANCH** to call the code that will output the light sequence specified by **mode**. Modes A through D work similarly, retrieving light sequences from the BASIC Stamp's EEPROM (stored in **DATA** statements). Mode E outputs a random light pattern.

Take a look at the code section labeled **ModeA**. The first thing that happens is that the variable, **offset**, is updated – again using the "wrap-around" technique with the modulus operator. The value of **offset** is added to the starting position of the specified light sequence and the current light pattern is retrieved with **READ**. Notice that the **DATA** statements for each sequence are labeled (**SeqA**, **SeqB**, etc.). Internally, each of these labels is converted to a constant value that is equal to the starting address of the sequence. The length of each sequence is calculated with these constants. By using this technique, light patterns can be updated (shortened or lengthened) without having to modify the operational code called by **Show**. **ModeE** is very straightforward**,** using the **RANDOM** function to output new pattern of lights with each pass through the main loop.

Challenge

Add a new lighting sequence. What sections of the program need to be modified to make this work?

Building Circuits On Your Own

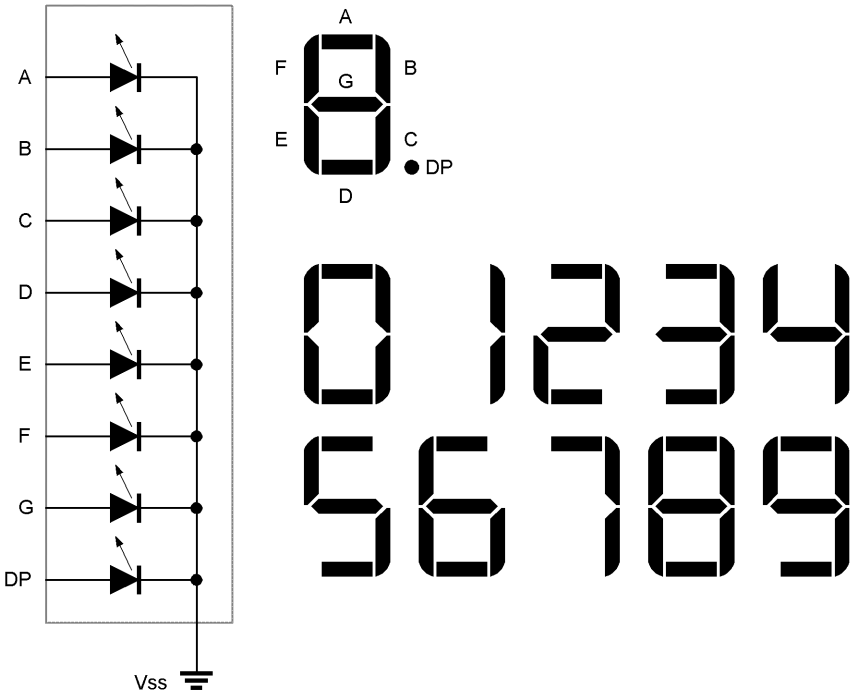
With the experience you gained in the previous section, you're ready to assemble the following circuits without specific instruction. These projects are fairly simple and you'll find them electrically similar to several of the projects that you've already built.

Proceed slowly and double-check your connections before applying power. You're well on your way to designing your own Stamp-based projects and experiments.

Let's continue with 7-segment displays....

Using 7-Segment Displays

A 7-segment display is actually seven (eight counting the decimal point) standard LEDs that have been packaged into a linear shape and arranged as a Figure-8 pattern. The LEDs in the group have a common element (anode or cathode).



By lighting specific combinations of the LEDs in the package we can create digits and even a few alpha characters (letters and symbols). Seven-segment LEDs are usually used in numeric displays.

The StampWorks lab has four, common-cathode seven-segment displays. The experiments in this section will show you how to get the most from these versatile components.

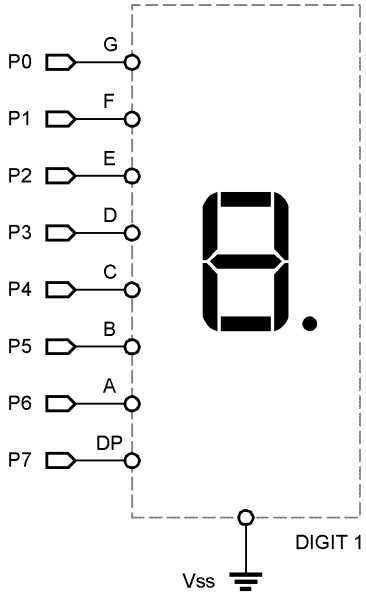
Experiment #8: A Single-Digit Counter

The purpose of this experiment is to demonstrate the use of seven-segment LED module by creating a simple decimal counter.

New PBASIC elements/commands to know:

- `Nib`

Building The Circuit.



```
' =============================================================================
'
'   File..... Ex08 - SevenSegs.BS2
'   Purpose... 7-Segment Display
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' Displays digits on a 7-segment display.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Segs            VAR     OutL                    ' 7-segment LEDs


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Blank           CON     %00000000               ' clears the display


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

counter         VAR     Nib



' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------
```

```
' Segments                .abcdefg
'                         --------
DecDig          DATA    %01111110               ' 0
                DATA    %00110000               ' 1
                DATA    %01101101               ' 2
                DATA    %01111001               ' 3
                DATA    %00110011               ' 4
                DATA    %01011011               ' 5
                DATA    %01011111               ' 6
                DATA    %01110000               ' 7
                DATA    %01111111               ' 8
                DATA    %01111011               ' 9


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111111                              ' make segments outputs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR counter = 0 TO 9                          ' count
    READ (DecDig + counter), Segs               ' put 7-seg pattern on digit
    PAUSE 1000                                  ' show for about one second
  NEXT
  GOTO Main                                     ' do it all again

  END
```

Behind The Scenes

This program is very similar to the light show program: a pattern is read from the EEPROM and output to the LEDs. In this program, sending specific patterns to the seven-segment LED creates the digits zero through nine.

Challenge

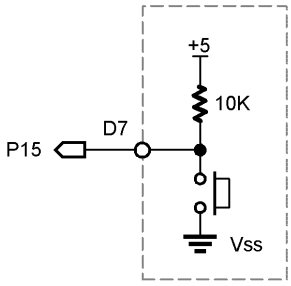
Update the program to create a single-digit HEX counter. Use the patterns below for the HEX digits.

Experiment #9: A Digital Die

The purpose of this experiment is create a digital die (one half of a pair of dice).

Building The Circuit.

Add this pushbutton to the circuit in Experiment #8.




```
' =============================================================================
'
'   File..... Ex09 - Roller.BS2
'   Purpose... Digital Die
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program combines a 7-segment display and a pushbutton input to create
' a single-digit digital die. Displays 1 to 6 when button is pressed.
```

```
' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

RollBtn         CON     15                      ' roll button on Pin 15
Segs            VAR     OutL                    ' 7-segment LEDs


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

swData          VAR     Byte                    ' data for BUTTON command
dieVal          VAR     Nib                     ' new die value
spinPos         VAR     Nib                     ' spinner position
doSpin          VAR     Nib                     ' spinner update control


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

'                         abcdefg
'                         -------
DecDig          DATA    %01111110               ' 0
                DATA    %00110000               ' 1
                DATA    %01101101               ' 2
                DATA    %01111001               ' 3
                DATA    %00110011               ' 4
                DATA    %01011011               ' 5
                DATA    %01011111               ' 6
                DATA    %01110000               ' 7
                DATA    %01111111               ' 8
                DATA    %01111011               ' 9

Bug             DATA    %01000000               ' spinning bug
                DATA    %00100000
                DATA    %00010000
                DATA    %00001000
                DATA    %00000100
                DATA    %00000010



' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------
```

```
Initialize:
  DirL = %01111111                              ' create output pins


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Get_Die                                 ' update die value
  PAUSE 5
  ' is the button pressed?
  BUTTON RollBtn, 0, 255, 10, swData, 1, Show_Die
  GOTO Main                                     ' no

Show_Die:
  READ (DecDig + dieVal), Segs                  ' show the die
  PAUSE 3000                                    ' - for 3 seconds
  GOTO Main                                     ' go again

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Get_Die:
  dieVal = (dieVal // 6) + 1                    ' limit = 1 to 6
  READ (Bug + spinPos), segs                    ' show spinner pattern
  doSpin = (doSpin + 1) // 7                    ' time to update spinner?
  IF (doSpin > 0) THEN Get_DieX                 ' only if doSpin = 0
  spinPos = spinPos + 1 // 6                    ' update spinner

Get_DieX:
  RETURN
```

Behind The Scenes

This program borrows heavily from what we've already done and should be easy for you to understand. What we've done here is added a bit of programming creativity to make a very simple program visually interesting.

There is one noteworthy point: the use of the variable, **`doSpin`**. In order to create a random value, the variable **`dieVal`** is updated rapidly until the button is pressed. This rate of change, however, is too fast to allow for a meaningful display of the rotating "bug." The variable **`doSpin`**, then, acts as a delay timer, causing the LED "bug" position to be updated every seventh pass through the **`Get_Die`** routine. This allows us to see it clearly and creates an inviting display.

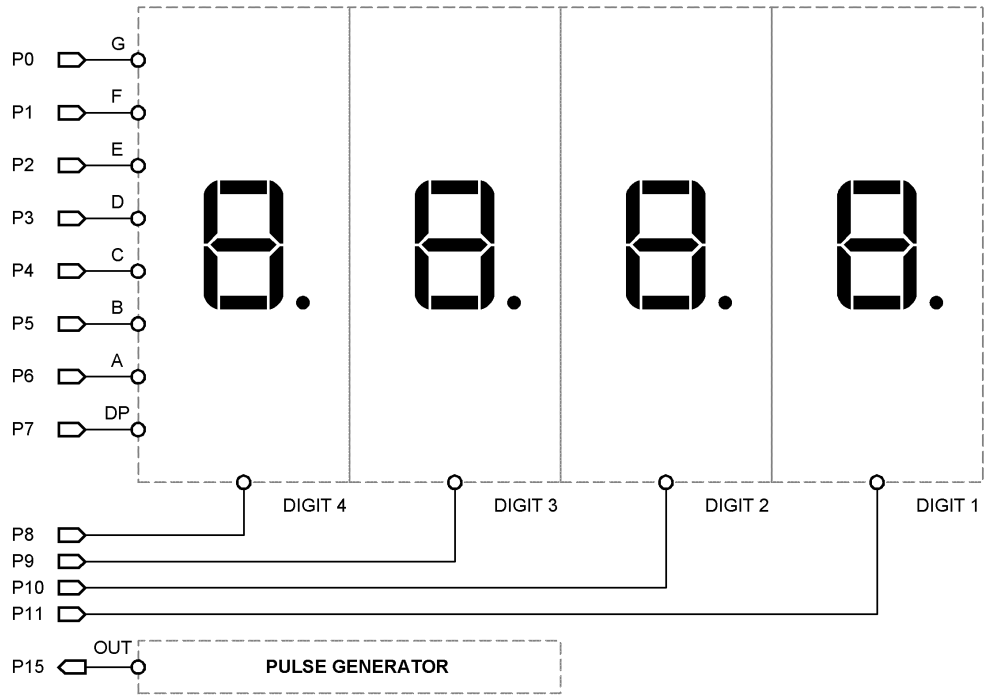
Experiment #10: LED Clock Display

The purpose of this experiment is create a simple clock display using four, seven-segment LED modules.

New PBASIC elements/commands to know:

- `OutA,OutB,OutC,OutD`
- `DirA,DirB,DirC,DirD`
- `In0 - In15`
- `DIG`

Building The Circuit



```
' =============================================================================
'
'   File..... Ex10 - Clock.BS2
'   Purpose... Simple software clock
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program monitors a 1 Hz input signal and uses it as the timebase for
' a software clock.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Segs            VAR     OutL                    ' segments
DigSel          VAR     OutC                    ' digit select
Tic             VAR     In15                    ' 1 Hz Pulse Generator input


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DecPoint        CON     %10000000               ' decimal point bit
Blank           CON     %00000000               ' all segments off

Dig0            CON     %1111                   ' digit select control
Dig1            CON     %1110
Dig2            CON     %1101
Dig3            CON     %1011
Dig4            CON     %0111

IsLow           CON     0                       ' Tic input is low
IsHigh          CON     1                       ' Tic input is high
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

secs            VAR     Word                    ' seconds
time            VAR     Word                    ' formatted time
digit           VAR     Nib                     ' current display digit


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

'                        .abcdefg
'                        --------
DecDig          DATA    %01111110               ' 0
                DATA    %00110000               ' 1
                DATA    %01101101               ' 2
                DATA    %01111001               ' 3
                DATA    %00110011               ' 4
                DATA    %01011011               ' 5
                DATA    %01011111               ' 6
                DATA    %01110000               ' 7
                DATA    %01111111               ' 8
                DATA    %01111011               ' 9


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111111                              ' make segments outputs
  DirC = %1111                                  ' make digit selects outputs
  DigSel = Dig0                                 ' all digits off


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Show_Time                               ' show current digit
  IF (Tic = IsHigh) THEN Inc_Sec                ' new second?
  GOTO Main                                     ' do it again
```

```
Inc_Sec:
  secs = (secs + 1) // 3600                    ' update seconds counter

Waiting:
  GOSUB Show_Time                              ' show current digit
  IF (Tic = IsLow) THEN Main                   ' if last tic gone, go back

  ' additional code could go here

  GOTO Waiting                                 ' do tic check again

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Show_Time:
  time = (secs / 60) * 100                     ' get minutes, put in hundreds
  time = time + (secs // 60)                   ' get seconds, put in 10s & 1s
  Segs = Blank                                 ' clear display
  ' enable digit
  LOOKUP digit, [Dig1, Dig2, Dig3, Dig4], digSel
  READ (DecDig + (time DIG digit)), Segs       ' put segment pattern in digit
  IF (digit <> 2) THEN Skip_DP
  Segs = Segs + DecPoint                       ' illuminate decimal point

Skip_DP:
  PAUSE 1                                      ' show it
  digit = (digit + 1) // 4                     ' get next digit
  RETURN
```

Behind The Scenes

The first two projects with seven-segment displays used only one digit. This project uses all four. A new problem arises; since the segment (anode) lines of the four displays are tied together, we can only show one at a time. This is accomplished by outputting the segment pattern then enabling the desired digit (by making its cathode low).

The goal of this program though, is to create a clock display, which means we want to see all four digits at the same time. While we can't actually have all four running at once, we can trick the human eye into thinking so.

The human eye has a property known as Persistence Of Vision (POV), which causes it to hold an image briefly. The brighter the image, the longer it holds in our eyes. POV is what causes us to see a bright spot in our vision after a friend snaps a flash photo. We can use POV to our advantage by rapidly cycling through each of the four digits, displaying the proper segments for that digit for a short period. If the cycle is fast enough, the POV of our eyes will cause the all four digits to appear to be lit at the same time. This process is called multiplexing.

Multiplexing is the process of sharing data lines; in this case, the segment lines to the displays are being shared. If we didn't multiplex, 28 output lines would be required to control four seven-segment displays. That's 12 more lines than are available on the BASIC Stamp.

The real work in this program happens in the subroutine called **`Show_Time`**. Its purpose is to time-format (MMSS) the seconds counter and update the current digit. Since the routine can only show one digit at a time, it must be called frequently, otherwise display strobing will occur. This program will update the display while waiting for other things to happen.

The clock display is created by moving the minutes value (secs / 60) into the thousands and hundreds columns of the variable **`time.`** The remaining seconds (secs // 60) are added to **`time`**, placing them in the tens and ones columns. Here's how the conversion math works:

Example: 754 seconds

754 / 60 = 12
12 x 100 = 1200 (time = 1200)
754 // 60 = 34
1200 + 34 = 1234 (time = 1234; 12 minutes and 34 seconds)

Now that the **`time`** display value is ready, the segments are cleared for the next update. Clearing the current segments value keeps the display sharp. If this isn't done, the old segments value will cause "ghosting" in the display. A **`LOOKUP`** table is used to enable the current digit and the segments for that digit are **`READ`** from an EEPROM **`DATA`** table.

The StampWorks display does not have the colon (:) normally found on a digital clock, so we'll enable the decimal point behind the second digit. If the current digit is not a second, the decimal point illumination is skipped. The final steps are a short delay so the digit illuminates and the current digit variable is updated.

The main loop of this program watches an incoming square-wave signal, produced by the StampWorks signal generator. When set at 1 Hz, this signal goes from LOW to HIGH once each

second. When this low-to-high transition occurs, the seconds counter is updated. The modulus operator (//) is used to keep seconds in the range of 0 to 3599 (the range of seconds in one hour).

When the seconds counter is updated, the display is refreshed and then the program waits for the incoming signal to go low, updating the display during the wait. If the program went right back to the top and the incoming signal was still high, the seconds counter would be prematurely updated, causing the clock to run fast. Once the incoming signal does go low, the program loops back to the top where it waits for the next low-to-high transition from the pulse generator.

Challenge

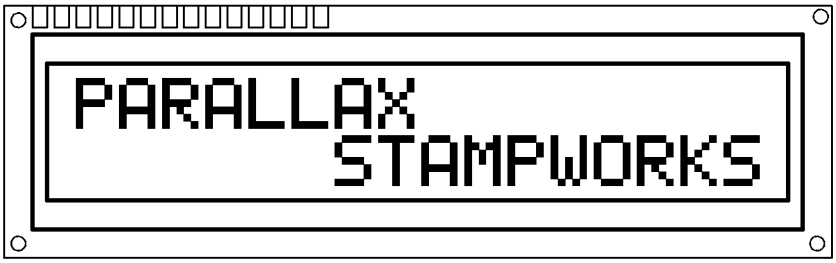
If the decimal point illumination is modified as follows, what will happen? Modify and download the program to check your answer.

```
    segs = segs + (DPoint * time.Bit0)          ' illuminate decimal point
```

Using Character LCDs

While LEDs and seven-segment displays make great output devices, there will be projects that require providing more complex information to the user. Of course, nothing beats the PC video display, but these are large, expensive and almost always impractical for microcontroller projects. Character LCD modules, on the other hand, fit the bill well. These inexpensive modules allow both text and numeric output, use very few I/O lines and require little effort from the BASIC Stamp.

Character LCD modules are available in a wide variety of configurations: one-line, two-line and four-line are very common. Screen width is also variable, but is usually 16 or 20 characters for each line.



The StampWorks LCD module (2 lines x 16 characters).
Datasheet is available for download from www.parallaxinc.com.

The StampWorks LCD module connects to the lab board by a 14-pin IDC header. The header is keyed, preventing the header from being inserted upside-down.

Initialization

The character LCD must be initialized before sending information to it. The projects in this document initialize the LCD in accordance with the specification for the Hitachi HD44780 controller. The Hitachi controller is the most popular available and many controllers are compatible with it.

Modes Of Operation

There are two essential modes of operation with character LCDs: sending a character and sending a command. When sending a character, the RS line is high and the data sent is interpreted as a character to be displayed at the current cursor position. The code sent is usually the ASCII code FOR the character. Several non-ASCII characters also are available in the LCD, as well as up to eight user-programmable custom characters.

Commands are sent to the LCD by taking the RS line low before sending the data. Several standard commands are available to manage and manipulate the LCD display.

Clear	$01	Clears the LCD and moves cursor to first position of first line
Home	$02	Moves cursor to first position of first line
Cursor Left	$10	Moves cursor to the left
Cursor Right	$14	Moves cursor to the right
Display Left	$18	Shifts entire display to the left
Display Right	$1C	Shifts entire display to the right

Connecting The LCD

The StampWorks LCD has a 14-pin IDC connector at the end of its cable. The connector is "keyed" so that it is always inserted correctly into the StampWorks lab. Simply align the connector key (small bump) with the slot in the LCD socket and press the connector into the socket until it is firmly seated.

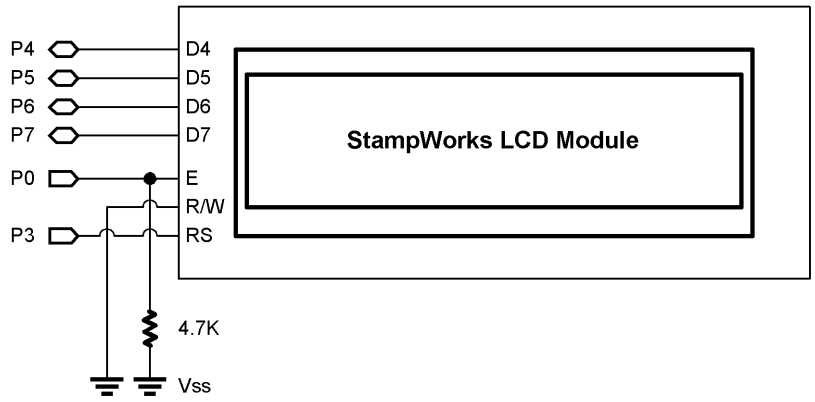
Experiment #11: A Basic LCD Demonstration

This program demonstrates character LCD fundamentals by putting the StampWorks LCD module through its paces.

New PBASIC elements/commands to know:

- PULSOUT
- HighNib, LowNib
- ^ (Exclusive OR operator)

Building The Circuit




```
' =============================================================================
'
'   File..... Ex11 - LCD Demo.BS2
'   Purpose... Essential LCD control
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates essential character LCD control.
'
' The connections for this program conform to the BS2p LCDIN and LCDOUT
' commands.  Use this program for the BS2, BS2e or BS2sx.  There is a separate
' program for the BS2p.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

E               CON     0                       ' LCD Enable pin  (1 = enabled)
RS              CON     3                       ' Register Select (1 = char)
LCDbus          VAR     OutB                    ' 4-bit LCD data bus


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

ClrLCD          CON     $01                     ' clear the LCD
CrsrHm          CON     $02                     ' move cursor to home position
CrsrLf          CON     $10                     ' move cursor left
CrsrRt          CON     $14                     ' move cursor right
DispLf          CON     $18                     ' shift displayed chars left
DispRt          CON     $1C                     ' shift displayed chars right
DDRam           CON     $80                     ' Display Data RAM control


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

char            VAR     Byte                    ' character sent to LCD
index           VAR     Byte                    ' loop counter


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------
```

```
Msg             DATA    "THE BASIC STAMP!", 0   ' preload EEPROM with message


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111101                              ' setup pins for LCD

LCD_Init:
  PAUSE 500                                     ' let the LCD settle
  LCDbus = %0011                                ' 8-bit mode
  PULSOUT E, 1
  PAUSE 5
  PULSOUT E, 1
  PULSOUT E, 1
  LCDbus = %0010                                ' 4-bit mode
  PULSOUT E, 1
  char = %00001100                              ' disp on, crsr off, blink off
  GOSUB LCD_Command
  char = %00000110                              ' inc crsr, no disp shift
  GOSUB LCD_Command


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  char = ClrLCD                                 ' clear the LCD
  GOSUB LCD_Command
  PAUSE 500
  index = Msg                                   ' get EE address of message

Read_Char:
  READ index, char                              ' get character from EEPROM
  IF (char = 0) THEN Msg_Done                   ' if 0, message is complete
  GOSUB LCD_Write                               ' write the character
  index = index + 1                             ' point to next character
  GOTO Read_Char                                ' go get it

Msg_Done:                                       ' the message is complete
  PAUSE 2000                                    ' wait 2 seconds
  char = CrsrHm                                 ' move the cursor home
  GOSUB LCD_Command
  char = %00001110                              ' turn the cursor on
```

```
  GOSUB LCD_Command
  PAUSE 500

  char = CrsrRt
  FOR index = 1 TO 15                          ' move the cursor accross display
    GOSUB LCD_Command
    PAUSE 150
  NEXT

  FOR index = 14 TO 0                          ' go backward by moving cursor
    char = DDRam + index                       '  to a specific address
    GOSUB LCD_Command
    PAUSE 150
  NEXT

  char = %00001101                             ' cursor off, blink on
  GOSUB LCD_Command
  PAUSE 2000

  char = %00001100                             ' blink off
  GOSUB LCD_Command

  FOR index = 1 TO 10                          ' flash display
    char = char ^ %00000100                    ' toggle display bit
    GOSUB LCD_Command
    PAUSE 250
  NEXT
  PAUSE 1000

  FOR index = 1 TO 16                          ' shift display
    char = DispRt
    GOSUB LCD_Command
    PAUSE 100
  NEXT
  PAUSE 1000

  FOR index = 1 TO 16                          ' shift display back
    char = DispLf
    GOSUB LCD_Command
    PAUSE 100
  NEXT
  PAUSE 1000
  GOTO Main                                    ' do it all over

  END
```

```
' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

LCD_Command:
  LOW RS                                        ' enter command mode

LCD_Write:
  LCDbus = char.HighNib                         ' output high nibble
  PULSOUT E, 1                                  ' strobe the Enable line
  LCDbus = char.LowNib                          ' output low nibble
  PULSOUT E, 1
  HIGH RS                                       ' return to character mode
  RETURN
```

Behind The Scenes

This is a very simple program, which demonstrates the basic functions of a character LCD. The LCD is initialized using four-bit mode in accordance with the Hitachi HD44780 controller specifications. This mode is used to minimize the number of BASIC Stamp I/O lines needed to control the LCD. While it is possible to connect to and control the LCD with eight data lines, this will not cause a noticeable improvement in program performance and will use four more I/O lines.

Experiment #11: A Basic LCD Demonstration

The basics of the initialization are appropriate for most applications:

- The display is on
- The cursor is off
- Display blinking is disabled
- The cursor is automatically incremented after each write
- The display does not shift

With the use of four data bits, two write cycles are necessary to send a byte to the LCD. The BASIC Stamps' **HighNib** and **LowNib** variable modifiers make this process exceedingly easy. Each nibble is latched into the LCD by blipping the E (enable) line with **PULSOUT**.

The demo starts by clearing the LCD and displaying a message that has been stored in a **DATA** statement. This technique of storing messages in EEPROM is very useful and makes programs easier to update. In this program, characters are written until a zero is encountered. This method lets us change the length of the string without worry about **FOR-NEXT** control settings. With the message displayed, the cursor position is returned home (first position of first line) and turned on (an underline cursor appears).

The cursor is sent back and forth across the LCD using two techniques. The first uses the cursor-right command. Moving the cursor back is accomplished by manually positioning the cursor. Manual cursor positioning is required by many LCD programs for tidy formatting of the information in the display.

With the cursor back home, it is turned off and the blink attribute is enabled. **Blink** causes the current cursor position to alternate between the character and a solid black box. This can be useful as an attention getter. Another attention-getting technique is to flash the entire display. This is accomplished by toggling the display enable bit. The Exclusive OR operator (^) simplifies bit toggling, as any bit XOR'd with a "1" will invert (1 XOR 1 = 0, 0 XOR 1 = 1).

Using the display shift commands, the entire display is shifted off-screen to the right, then back. What this demonstrates is that the display is actually a window into the LCD's memory. One method of using the additional memory is to write messages off-screen and shift to them.

Experiment #12: Creating Custom LCD Characters

This program demonstrates the creation of custom LCD characters, animation with the custom characters and initializing the LCD for multi-line mode.

Building The Circuit

Use the same circuit as in Experiment #11.

```
' =============================================================================
'
'   File..... Ex12 - LCD Characters.BS2
'   Purpose... Custom LCD Characters
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates custom character creation and animation on a
' character LCD.
'
' The connections for this program conform to the BS2p LCDIN and LCDOUT
' commands.  Use this program for the BS2, BS2e or BS2sx.  There is a separate
' program for the BS2p.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

E               CON     0                       ' LCD Enable pin  (1 = enabled)
RS              CON     3                       ' Register Select (1 = char)
LCDbus          VAR     OutB                    ' 4-bit LCD data bus
```

```
' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

ClrLCD          CON     $01                     ' clear the LCD
CrsrHm          CON     $02                     ' move cursor to home position
CrsrLf          CON     $10                     ' move cursor left
CrsrRt          CON     $14                     ' move cursor right
DispLf          CON     $18                     ' shift displayed chars left
DispRt          CON     $1C                     ' shift displayed chars right
DDRam           CON     $80                     ' Display Data RAM control
CGRam           CON     $40                     ' Custom character RAM
Line1           CON     $80                     ' DDRAM address of line 1
Line2           CON     $C0                     ' DDRAM address of line 2


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

char            VAR     Byte                    ' character sent to LCD
newChar         VAR     Byte                    ' new character for animation
index1          VAR     Byte                    ' loop counter
index2          VAR     Byte                    ' loop counter


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

Msg1            DATA    "THE BASIC STAMP "      ' preload EEPROM with messages
Msg2            DATA    " IS VERY COOL! ", 3

CC0             DATA    $0E, $1F, $1C, $18, $1C, $1F, $0E, $00  ' character 0
CC1             DATA    $0E, $1F, $1F, $18, $1F, $1F, $0E, $00  ' character 1
CC2             DATA    $0E, $1F, $1F, $1F, $1F, $1F, $0E, $00  ' character 2
Smiley          DATA    $00, $0A, $0A, $00, $11, $0E, $06, $00  ' smiley face


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111101                              ' setup pins for LCD
```

```
LCD_Init:
  PAUSE 500                                     ' let the LCD settle
  LCDbus = %0011                                ' 8-bit mode
  PULSOUT E, 1
  PAUSE 5
  PULSOUT E, 1
  PULSOUT E, 1
  LCDbus = %0010                                ' 4-bit mode
  PULSOUT E, 1
  char = %00101000                              ' multi-line mode
  GOSUB LCD_Command
  char = %00001100                              ' disp on, crsr off, blink off
  GOSUB LCD_Command
  char = %00000110                              ' inc crsr, no disp shift
  GOSUB LCD_Command

Download_Chars:                                 ' download custom chars to LCD
  char = CGRam                                  ' point to CG RAM
  GOSUB LCD_Command                             ' prepare to write CG data
  FOR index1 = CC0 TO (Smiley + 7)              ' build 4 custom chars
    READ index1, char                           ' get byte from EEPROM
    GOSUB LCD_Write                             ' put into LCD CG RAM
  NEXT


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  char = ClrLCD                                 ' clear the LCD
  GOSUB LCD_Command
  PAUSE 250

  FOR index1 = 0 TO 15                          ' get message from EEPROM
    READ (Msg1 + index1),char                   ' read a character
    GOSUB LCD_Write                             ' write it
  NEXT

  PAUSE 2000                                    ' wait 2 seconds

Animation:
  FOR index1 = 0 TO 15                          ' cover 16 characters
    READ (Msg2 + index1), newChar               ' get new char from 2nd message
    FOR index2 = 0 TO 4                         ' 5 characters in animation cycle
      char = Line2 + index1                     ' set new DDRAM address
```

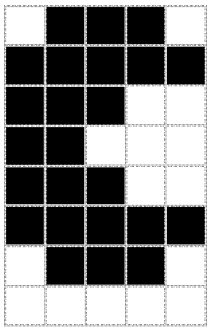
```
      GOSUB LCD_Command
      LOOKUP index2, [0, 1, 2, 1, newChar], char
      GOSUB LCD_Write                          ' write animation character
      PAUSE 50                                 ' delay between animation chars
    NEXT
  NEXT
  PAUSE 1000
  GOTO Main
                                               ' do it all over
  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

LCD_Command:
  LOW RS                                       ' enter command mode

LCD_Write:
  LCDbus = char.HighNib                        ' output high nibble
  PULSOUT E, 1                                 ' strobe the Enable line
  LCDbus = char.LowNib                         ' output low nibble
  PULSOUT E, 1
  HIGH RS                                      ' return to character mode
  RETURN
```

Behind The Scenes

In this program, the LCD is initialized for multi-line mode. This will allow both lines of the StampWorks LCD module to display information. With the display initialized, custom character data is downloaded to the LCD.

The LCD has room for eight, user-definable customer characters. The data is stored for these characters in an area called CGRAM and must be downloaded to the LCD after power-up and initialization (custom character definitions are lost when power is removed from the LCD). Each custom character requires eight bytes of data. The eighth byte is usually $00, since this is where the cursor is positioned when under the character.

The standard LCD font is five bits wide by seven bits tall. You can create custom characters that are eight bits tall, but the eighth line is generally reserved for the underline cursor. Here's an example of a custom character definition:



The shape of the character is determined by the ones and zeros in the data bytes. One in a given bit position will light a pixel; zero will extinguish it.

The bit patterns for custom characters are stored in the BASIC Stamp's EEPROM with **DATA** statements. To move the patterns into the LCD, the CGRam command is executed and the characters are written to the display. Before the characters can be used, the display must be returned to "normal" mode. The usual method is to clear the display or home the cursor.

Interestingly, the LCD retrieves the bit patterns from memory while refreshing the display. In advanced applications, the CGRam memory can be updated while the program is running to create unusual display effects.

Experiment #12: Creating Custom LCD Characters

The heart of this program is the animation loop. This code grabs a character from the second message, then, for each character in that message, displays the animation sequence at the desired character location on the second line of the LCD. A **LOOKUP** table is used to cycle the custom characters for the animation sequence. At the end of the sequence, the new character is revealed.

Challenge

Create your own custom character sequence. Update the initialization and animation code to accommodate your custom characters.

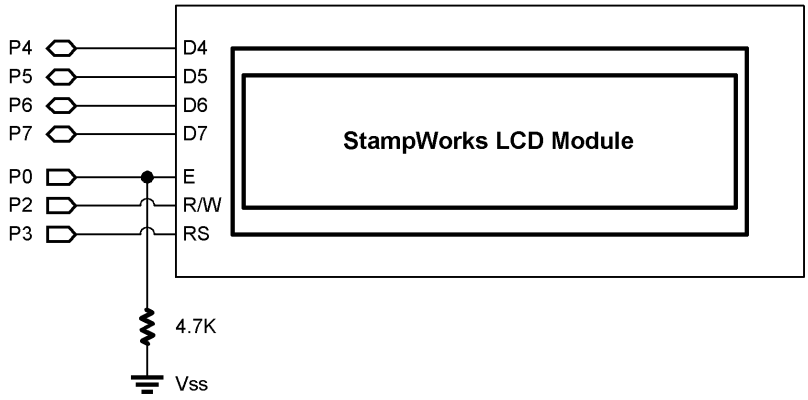
Experiment #13: Reading the LCD RAM

This program demonstrates the use of the LCD's CGRAM space as external memory.

New PBASIC elements/commands to know:

- `InA, InB, InC, InD`

Building The Circuit




```
' =============================================================================
'
'   File..... Ex13 - LCD Read.BS2
'   Purpose... Read data from LCD
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

Experiment #13: Reading the LCD RAM

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates how to read data from the LCD's display or CGRAM
' areas.
'
' The connections for this program conform to the BS2p LCDIN and LCDOUT
' commands.  Use this program for the BS2, BS2e or BS2sx.  There is a separate
' program for the BS2p.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

E               CON     0                       ' LCD Enable pin  (1 = enabled)
RW              CON     2                       ' LCD Read/Write pin (1 = write)
RS              CON     3                       ' Register Select (1 = char)
LCDdirs         VAR     DirB
LCDbusOut       VAR     OutB                    ' 4-bit LCD data bus
LCDbusIn        VAR     InB


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

ClrLCD          CON     $01                     ' clear the LCD
CrsrHm          CON     $02                     ' move cursor to home position
CrsrLf          CON     $10                     ' move cursor left
CrsrRt          CON     $14                     ' move cursor right
DispLf          CON     $18                     ' shift displayed chars left
DispRt          CON     $1C                     ' shift displayed chars right
DDRam           CON     $80                     ' Display Data RAM control
CGRam           CON     $40                     ' Custom character RAM


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

char            VAR     Byte                    ' character sent to LCD
index           VAR     Byte                    ' loop counter
rVar            VAR     Word                    ' for random number
addr            VAR     Byte                    ' address to write/read
tOut            VAR     Byte                    ' test value to write to LCD
```

```
tIn             VAR     Byte                    ' test value to read from LCD
temp            VAR     Word                    ' temp value for numeric display
width           VAR     Nib                     ' width of number to display


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111101                              ' setup pins for LCD

LCD_Init:
  PAUSE 500                                     ' let the LCD settle
  LCDbusOut = %0011                             ' 8-bit mode
  PULSOUT E, 1
  PAUSE 5
  PULSOUT E, 1
  PULSOUT E, 1
  LCDbusOut = %0010                             ' 4-bit mode
  PULSOUT E, 1
  char = %00001100                              ' disp on, crsr off, blink off
  GOSUB LCD_Command
  char = %00000110                              ' inc crsr, no disp shift
  GOSUB LCD_Command


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  char = ClrLCD                                 ' clear the LCD
  GOSUB LCD_Command

  FOR index = 0 TO 14                           ' create display
    LOOKUP index, ["ADDR=??  ???/???"], char
    GOSUB LCD_Write
  NEXT

Loop:
  RANDOM rVar                                   ' generate random number
  addr = rVar.LowByte & $3F                     ' create address (0 to 63)
  tOut = rVar.HighByte                          ' create test value (0 to 255)

  char = CGRam + addr                           ' set CGRAM pointer
  GOSUB LCD_Command
```

```
  char = tOut
  GOSUB LCD_Write                               ' move the value to CGRAM
  PAUSE 100                                     ' wait a bit, then go get it

  char = CGRam + addr                           ' set CGRAM pointer
  GOSUB LCD_Command
  GOSUB LCD_Read                                ' read value from LCD
  tIn = char

  ' display results

  char = DDRam + 5                              ' show address at position 5
  GOSUB LCD_Command
  temp = addr
  width = 2
  GOSUB Put_Val

  char = DDRam + 9                              ' show output at position 8
  GOSUB LCD_Command
  temp = tOut
  width = 3
  GOSUB Put_Val

  char = DDRam + 13                             ' show input at position 12
  GOSUB LCD_Command
  temp = tIn
  width = 3
  GOSUB Put_Val
  PAUSE 1000
  GOTO Loop                                     ' do it again

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Put_Val:
  FOR index = (width - 1) TO 0                  ' display digits left to right
    char = (temp DIG index) + 48                ' convert digit to ASCII
    GOSUB LCD_Write                             ' put digit in display
  NEXT
  RETURN
```

```
LCD_Command:
  LOW RS                                        ' enter command mode

LCD_Write:
  LCDbusOut = char.HighNib                      ' output high nibble
  PULSOUT E, 1                                  ' strobe the Enable line
  LCDbusOut = char.LowNib                       ' output low nibble
  PULSOUT E, 1
  HIGH RS                                       ' return to character mode
  RETURN


LCD_Read:
  HIGH RS                                       ' data command
  HIGH RW                                       ' read
  LCDdirs = %0000                               ' make data lines inputs
  HIGH E
  char.HighNib = LCDbusIn                       ' get high nibble
  LOW E
  HIGH E
  char.LowNib = LCDbusIn                        ' get low nibble
  LOW E
  LCDdirs = %1111                               ' return data lines to outputs
  LOW RW
  RETURN
```

Experiment #13: Reading the LCD RAM

Behind The Scenes

This program demonstrates the versatility of the BASIC Stamp's I/O lines and their ability to be reconfigured mid-program. Writing to the LCD was covered in the last two experiments. To read data back, the BASIC Stamp's I/O lines must be reconfigured as inputs. This is no problem for the BASIC Stamp. Aside from the I/O reconfiguration, reading from the LCD requires an additional control line: RW. In most programs this line can be tied low to allow writing to the LCD. For reading from the LCD the RW line is made high.

The program generates an address and data using the **RANDOM** function. The address is kept in the range of 0 to 63 by masking out the highest bits of the **LowByte** returned by the **RANDOM** function. The **HighByte** is used as the data to be written to and read back from the LCD.

The data is stored in the LCD's CGRAM area. This means -- in this program -- that the CGRAM memory cannot be used for custom characters. In programs that require less than eight custom characters the remaining bytes of CGRAM can be used as off-board memory.

Reading data from the LCD is identical to writing: the address is set and the data is retrieved. For this to take place, the LCD data lines must be reconfigured as inputs. Blipping the E (enable) line makes the data (one nibble at a time) available for the BASIC Stamp. Once again, **HighNib** and **LowNib** are used, this time to build a single byte from the two nibbles returned during the read operation.

When the retrieved data is ready, the address, output data and input data are written to the LCD for examination. As short subroutine, **Put_Val**, handles writing numerical values to the LCD. To use this routine, move the cursor to the desired location, put the value to be displayed in **temp**, the number of characters to display in **width**, then call **Put_Val**. The subroutine uses the **DIG** operator to extract a digit from **temp** and adds 48 to convert it to ASCII so that it can be displayed on the LCD.

Experiment #14: Magic 8-Ball Game

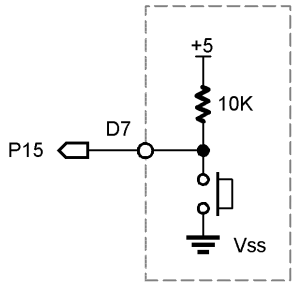
This program demonstrates the 8x10 font capability of StampWorks LCD module. The 8x10 font allows descended letters (g, j, p, q and y) to be displayed properly.

New PBASIC elements/commands to know:

- LOOKDOWN

Building The Circuit

Add this pushbutton to the circuit in Experiment #11 (remember to reconnect LCD.RW to Vss).




```
' =============================================================================
'
'   File..... Ex14 - LCD Magic 8-Ball.BS2
'   Purpose... Magic 8-Ball simulation
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program simulates a Magic 8-Ball.  Ask a question, then press the
' button to get your answer.
'
' The program also demonstrates using a 2-Line display as a single-line display
' with the 5x10 font set.  When using the 5x10 font, true descended characters
' are available but must be remapped from the LCD ROM.
'
' The connections for this program conform to the BS2p LCDIN and LCDOUT
' commands.  Use this program for the BS2, BS2e or BS2sx.  There is a separate
' program for the BS2p.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

E               CON     0                       ' LCD Enable pin  (1 = enabled)
RS              CON     3                       ' Register Select (1 = char)
LCDbus          VAR     OutB                    ' 4-bit LCD data out
AskButton       CON     15                      ' Ask button input pin


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

ClrLCD          CON     $01                     ' clear the LCD
CrsrHm          CON     $02                     ' move cursor to home position
CrsrLf          CON     $10                     ' move cursor left
CrsrRt          CON     $14                     ' move cursor right
DispLf          CON     $18                     ' shift displayed chars left
DispRt          CON     $1C                     ' shift displayed chars right
DDRam           CON     $80                     ' Display Data RAM control
CGRam           CON     $40                     ' Custom character RAM control

NumAnswers      CON     6                       ' 6 possible answers

_g              CON     $E7                     ' DDROM addresses of descenders
_j              CON     $EA
_p              CON     $F0
_q              CON     $F1
_y              CON     $F9
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

char            VAR     Byte                    ' character sent to LCD
addr            VAR     Byte                    ' message address
swData          VAR     Byte                    ' workspace for BUTTON
answer          VAR     Nib                     ' answer pointer
clock           VAR     Nib                     ' animation clock
pntr            VAR     Nib                     ' pointer to animation character


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

Prompt          DATA    "Ask a question", 0

Ans0            DATA    "Definitely YES", 0
Ans1            DATA    "Possible...", 0
Ans2            DATA    "Definitely NO", 0
Ans3            DATA    "Not likely...", 0
Ans4            DATA    "Answer uncertain", 0
Ans5            DATA    "Please ask again", 0


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11111101                              ' setup pins for LCD

LCD_Init:
  PAUSE 500                                     ' let the LCD settle
  LCDbus = %0011                                ' 8-bit mode
  PULSOUT E, 1
  PAUSE 5
  PULSOUT E, 1
  PULSOUT E, 1
  LCDbus = %0010                                ' 4-bit mode
  PULSOUT E, 1
  char = %00100100                              ' select 5x10 font
  GOSUB LCD_Command
  char = %00001100                              ' disp on, crsr off, blink off
  GOSUB LCD_Command
```

```
  char = %00000110                              ' inc crsr, no disp shift
  GOSUB LCD_Command


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  char = ClrLCD                                 ' clear the LCD
  GOSUB LCD_Command
  addr = Prompt
  GOSUB Show_Message                            ' print prompt

Rollem:
  GOSUB Shuffle                                 ' shuffle until button pressed
  PAUSE 5
  BUTTON AskButton, 0, 255, 10, swData, 1, Show_Answer
  GOTO Rollem

Show_Answer:
  ' get address of answer message
  LOOKUP answer, [Ans0, Ans1, Ans2, Ans3, Ans4, Ans5], addr

  char = ClrLCD
  GOSUB LCD_Command
  GOSUB Show_Message
  PAUSE 2000                                    ' give time to read answer
  GOTO Main                                     ' do it all over

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

LCD_Command:
  LOW RS                                        ' enter command mode

LCD_Write:
  LCDbus = char.HighNib                         ' output high nibble
  PULSOUT E,1                                   ' strobe the Enable line
  LCDbus = char.LowNib                          ' output low nibble
  PULSOUT E,1
  HIGH RS                                       ' return to character mode
  RETURN
```

```
Show_Message:
  READ addr,char                                ' read a character from EEPROM
  IF (char = 0) THEN Msg_Done                   ' if 0, message is complete
  GOSUB Translate                               ' fix letters with descenders
  GOSUB LCD_Write                               ' write the character
  addr = addr + 1                               ' point to next character
  GOTO Show_Message

Msg_Done:
  RETURN


' convert to descender font
' - does not change other characters

Translate:
  LOOKDOWN char, ["g", "j", "q", "p", "y"], char
  LOOKUP char, [_g, _j, _q, _p, _y], char
  RETURN


Shuffle:
  answer = (answer + 1) // NumAnswers            ' update answer pointer
  clock = (clock + 1) // 15                     ' update pointer clock
  IF (clock > 0) THEN Shuffle_Done              ' time to update animation?
  char = DDRam + 15                             ' yes, write at pos 15
  GOSUB LCD_Command
  LOOKUP pntr, ["-+|*"], char                   ' load animation character
  GOSUB LCD_Write                               ' write it
  pntr = (pntr + 1) // 4                        ' update animation char

Shuffle_Done:
  RETURN
```

Behind The Scenes

The standard 5x7 LCD font suffers aesthetically when it comes to descended letters, those letters with tails (g, j, p, q and y). The nature of the font map causes these letters to be "squashed" into the display. Many LCDs support a 5x10 character font and provide additional mapping for properly descended characters.

Using the 5x10 font is straightforward; it requires a single additional command in the initialization sequence. To display properly descended characters, however, is a bit trickier since these characters

are not mapped at equal offsets to their ASCII counterparts. Thankfully, the BASIC Stamp has a couple of table-oriented commands that simplify the translation process.

After initialization, the screen is cleared and the user is prompted to think of a question. The **Show_Message** subroutine displays a message at the current cursor position. The message is stored in a **DATA** statement and passed to the subroutine by its EEPROM address. **Show_Message** reads characters from the EEPROM until it finds a zero, passing each character to the subroutine, **Translate**, which re-maps the ASCII value for descended letters. **Translate** uses a clever trick with **LOOKUP** and **LOOKDOWN**.

When a character is passed to **Translate**, it is compared to the list of known descended letters. If the character is in this list, it is converted to a value that will be used by the **LOOKUP** table to re-map the character to the descended version in the LCD font map. If the character is not in the descended list, it will pass through **Translate** unaffected.

The main loop of the program waits for you to press the button, creating a randomized answer by continuously calling the **Shuffle** subroutine. **Shuffle** updates the answer variable and creates an animated bug. The animation is created with standard characters and updated every 15 cycles through the **Shuffle** subroutine. When the button is finally pressed, the EEPROM address of the corresponding answer is loaded with **LOOKUP** and the "magic" answer is displayed.

Challenge

Create custom characters that use the 5x10 font mode. Note: 16 bytes must be used for each character, even though only ten will be displayed.

Moving Forward

The first three sections of this manual dealt specifically with output devices, because the choice of output to the user is often critical to the success of a project. By now, you should be very comfortable with LEDs, seven-segment displays and LCDs. From this point forward we will present a variety of experiments -- some simple, others complex which will round your education as a BASIC Stamp programmer and give you the confidence you need to develop your own BASIC Stamp-controlled applications.

Remember, the key to success here is to complete each experiment and to take on each challenge. Then, go further by challenging yourself. Each time you modify a program you will learn something. It's okay if your experiments don't work as expected, because you will still be learning.

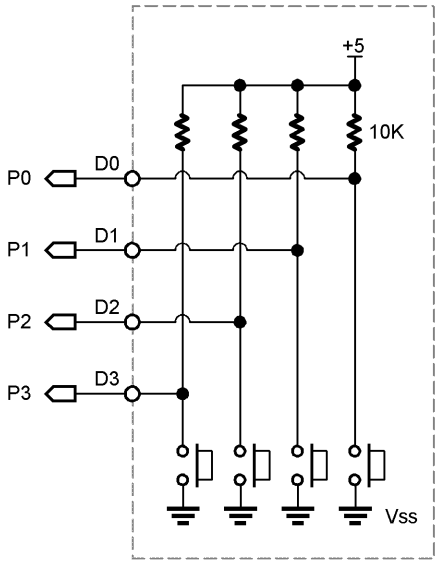
Experiment #15: Debouncing Multiple Inputs

The experiment will teach you how to debounce multiple BASIC Stamp inputs. With modification, any number of inputs from two to 16 can be debounced with this code.

New PBASIC elements/commands to know:

- ~ (1's compliment operator)
- CLS (DEBUG modifier)
- IBIN, IBIN1 – IBIN16 (DEBUG modifier)

Building The Circuit




```
' =============================================================================
'
'   File..... Ex15 - Debounce.BS2
'   Purpose... Multi-input button debouncing
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
```

Experiment #15: Debouncing Multiple Inputs

```
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' ============================================================================


' ----------------------------------------------------------------------------
' Program Description
' ----------------------------------------------------------------------------

' This program demonstrates the simultaneous debouncing of multiple inputs. The
' input subroutine is easily adjusted to handle any number of inputs.


' ----------------------------------------------------------------------------
' I/O Definitions
' ----------------------------------------------------------------------------


SwInputs        VAR     InA                     ' four inputs, pins 0 - 3


' ----------------------------------------------------------------------------
' Variables
' ----------------------------------------------------------------------------

switches        VAR     Nib                     ' debounced inputs
x               VAR     Nib                     ' loop counter


' ----------------------------------------------------------------------------
' Program Code
' ----------------------------------------------------------------------------

Main:
  GOSUB Get_Switches                            ' get debounced inputs
  DEBUG Home, "Inputs = ", IBIN4 switches       ' display in binary mode
  PAUSE 50                                      ' a little time between readings
  GOTO Main                                     ' do it again

  END

' ----------------------------------------------------------------------------
' Subroutines
' ----------------------------------------------------------------------------
```

```
Get_Switches:
  switches = %1111                              ' enable all four inputs
  FOR x = 1 TO 10
    switches = switches & ~SwInputs             ' test inputs
    PAUSE 5                                     ' delay between tests
  NEXT
  RETURN
```

Behind The Scenes

When debouncing only one input, the BASIC Stamp's **BUTTON** function works perfectly and even adds a couple of useful features (like auto-repeat). To debounce two or more inputs, we need to create a bit of code. The workhorse of this experiment is the subroutine **Get_Switches**. As presented, it will accommodate four switch inputs. It can be modified for any number of inputs from two to 16.

The purpose of **Get_Switches** is to make sure that the inputs stay on solid for 50 milliseconds with no contact bouncing. Debounced inputs will be retuned in the variable, **switches**, with a valid input represented by a 1 in the switch position.

The **Get_Switches** routine starts by assuming that all switch inputs will be valid, so all the bits of switches are set to one. Then, using a **FOR-NEXT** loop, the inputs are scanned and compared to the previous state. Since the inputs are active low (zero when pressed), the one's compliment operator (~) inverts them. The **And** operator (&) is used to update the current state. For a switch to be valid, it must remain pressed through the entire **FOR-NEXT** loop.

Here's how the debouncing technique works: When a switch is pressed, the input to the BASIC Stamp will be zero. The one's compliment operator will invert zero to one. One "Anded" with one is still one, so that switch remains valid. If the switch is not pressed, the input to the BASIC Stamp will be one (because of the 10K pull-up to Vdd). One is inverted to zero. Zero "Anded" with any number is zero and will cause the switch to remain invalid through the entire debounce cycle.

The debounce switch inputs are displayed in a **DEBUG** window with the **IBIN4** modifier so that the value of each switch input is clearly displayed.

Challenge

Modify the program to debounce and display eight switches.

Experiment #16: Counting Events

This experiment demonstrates an events-based program delay.

Building The Circuit


```
' =================================================================================
'
'   File...... Ex16 - Counter.BS2
'   Purpose... Counts external events
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =================================================================================


' ---------------------------------------------------------------------------------
' Program Description
' ---------------------------------------------------------------------------------

' Counts extenal events by wait for a low-to-high transition on the event
' input pin.


' ---------------------------------------------------------------------------------
' Revision History
' ---------------------------------------------------------------------------------


' ---------------------------------------------------------------------------------
' I/O Definitions
' ---------------------------------------------------------------------------------

EventIn         VAR     In15                    ' event input pin
```

```
' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

IsLow           CON     0
IsHigh          CON     1
Target          CON     1000                    ' target count


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

eCount          VAR     Word                    ' event count



' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Init:
  PAUSE 250                                     ' let DEBUG window open
  DEBUG CLS, "Started... ", CR
  eCount = 0                                    ' clear counter


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Wait_For_Count                          ' wait for 1000 pulses
  DEBUG "Count complete."

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Wait_For_Count:
  IF (EventIn = IsLow) THEN Wait_For_Count      ' wait for input to go high
  eCount = eCount + 1                           ' increment event count
  DEBUG Home, 10, "Count = ", DEC eCount, CR
```

```
  IF (eCount = Target) THEN Wait_Done           ' check against target

Wait_Low:
  IF (EventIn = IsHigh) THEN Wait_Low           ' wait for input to go low
  GOTO Wait_For_Count

Wait_Done:
  RETURN
```

Behind The Scenes

The purpose of the **Wait_For_Count** subroutine is to cause the program to wait for a specified number of events. In an industrial setting, for example, a packaging system we might need to run a conveyor belt until 100 boxes pass.

When the program is passed to **Wait_For_Count**, the input pin is monitored for a low-to-high transition. When the line goes high, the counter is incremented and the program waits for the line to go low. When this happens, the code loops back for the next high input. When the target count is reached, the subroutine returns to the main program. The time spent in the subroutine is determined by the rate of incoming events.

Note that the subroutine expects a clean input. A noisy input could cause spurious counts, leading to early termination of the subroutine. One method of dealing with a noisy input – when the time between expected events is known – is to add a **PAUSE** statement after the start of an event. The idea is to **PAUSE** when the event starts and end the **PAUSE** after the event with a bit of lead-time before the next event is expected. The code that follows works when the events are about a half-second in length and the time between events is two seconds:

```
Wait_For_Count:
  IF (P_in = IsLow) THEN Wait_For_Count         ' wait for high pulse
  pCount = pCount + 1                           ' increment count
  DEBUG Home, 10, "Count = ", DEC eCount, CR
  IF (pCount = Target) THEN Wait_Done           ' check against target
  PAUSE 1500                                    ' clean-up noisy input

Wait_Low:
  IF (P_in = IsHigh) THEN Wait_Low              ' wait for pulse to go low
  GOTO Wait_For_Count

Wait_Done:
  RETURN
```

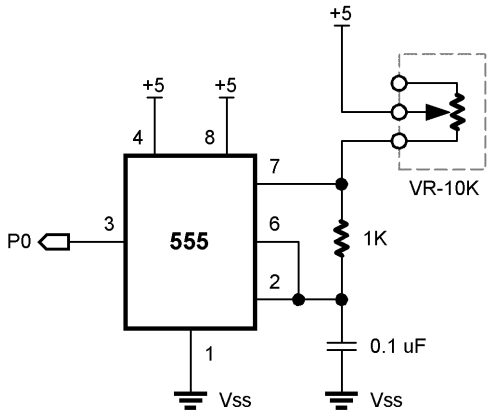
Experiment #17: Frequency Measurement

This experiment determines the frequency of an incoming pulse stream by using the BASIC Stamp's **COUNT** function.

New PBASIC elements/commands to know:

- COUNT

Building The Circuit (Note that schematic is NOT chip-centric)




```
' =============================================================================
'
'   File..... Ex17 - FreqIn1.BS2
'   Purpose... Frequency input
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

Experiment #17: Frequency Measurement

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program monitors and displays the frequency of a signal on Pin 0.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

FreqPin         CON     0                       ' frequency input pin


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

OneSec          CON     1000                    ' one second - BS2
' OneSec        CON     2500                    ' BS2sx
' OneSec        CON     3484                    ' BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

freq            VAR     Word                    ' frequency


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  COUNT FreqPin, OneSec, freq                   ' collect pulses for 1 second
  DEBUG CLS, "Frequency: ", DEC freq, " Hz"     ' display on DEBUG screen
  GOTO Main                                     ' do it again

  END
```

Behind The Scenes

In the previous experiment, several lines of code were used to count pulses on an input pin. That method works when counting to a specific number. Other programs will want to count the number of pulses that arrive during a specified time period. The BASIC Stamp's **COUNT** function is designed for this purpose.

The frequency of an oscillating signal is defined as the number of cycles per second and is expressed in Hertz. The BASIC Stamp's **COUNT** function monitors the specified pin for a given amount of time. To create a frequency meter, the specified time window is set to 1000 milliseconds (one second).

Challenge

Improve the responsiveness (make it update more frequently) of this program by changing the **COUNT** period. What other adjustment has to be made? How does this change affect the ability to measure very low frequency signals?

Experiment #18: Advanced Frequency Measurement

This experiment uses **PULSIN** to create a responsive frequency meter.

New PBASIC elements/commands to know:

- PULSIN

Building The Circuit

Use the same circuit as in Experiment #18.

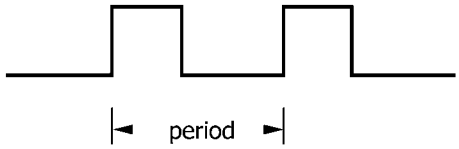
```
' =============================================================================
'
'   File..... Ex18 - FreqIn2.BS2
'   Purpose... Frequency Input
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2p}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program monitors and displays the frequency of a signal on Pin 0.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

FreqPin         CON     0                       ' frequency input pin


' -----------------------------------------------------------------------------
' Constants
```

```
' -----------------------------------------------------------------------------

Convert         CON     $0200                   ' input to uSeconds (BS2)
' Convert       CON     $00CC                   ' BS2sx
' Convert       CON     $00C0                   ' BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

pHigh           VAR     Word                    ' high pulse width
pLow            VAR     Word                    ' low pulse width
period          VAR     Word                    ' cycle time (high + low)
freq            VAR     Word                    ' frequency


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  PULSIN FreqPin, 0, pHigh                      ' get high portion of input
  PULSIN FreqPin, 1, pLow                       ' get low portion of input
  period = (pHigh + pLow) */ Convert            ' calculate cycle width in uSecs
  freq = 50000 / period * 20                    ' calculate frequency

  ' display on DEBUG screen

  DEBUG Home
  DEBUG "Period..... ", DEC period, " uS    ", CR
  DEBUG "Frequency... ", DEC freq, " Hz    "
  GOTO Main                                     ' do it again

  END
```

Behind The Scenes

In the last experiment, you learned that the frequency of a signal is defined as the number of cycles per second. You created a simple frequency meter by counting the number of pulses (cycles) in one second. This method works well, especially for low-frequency signals. There will be times, however, when project requirements will dictate a quicker response time for frequency measurement.

The frequency of a signal can be calculated from its period, or the time for one complete cycle.



By measuring the period of an incoming signal, its frequency can be calculated with the equation (where the period is expressed in seconds):

frequency = 1 / period

The BASIC Stamp's **PULSIN** function is designed to measure the width of an incoming pulse. By using **PULSIN** to measure the high and low portions of an incoming signal, its period can be calculated and the frequency can be determined. The result of **PULSIN** is expressed in units of two microseconds. Thus, the formula for calculating frequency becomes:

frequency = 500,000 / period

This creates a problem for BASIC Stamp math though, as it can only deal with 16-bit numbers (maximum value is 65,535). To fix the formula, we convert 500,000 to 50,000 x 10 and rewrite the formula like this

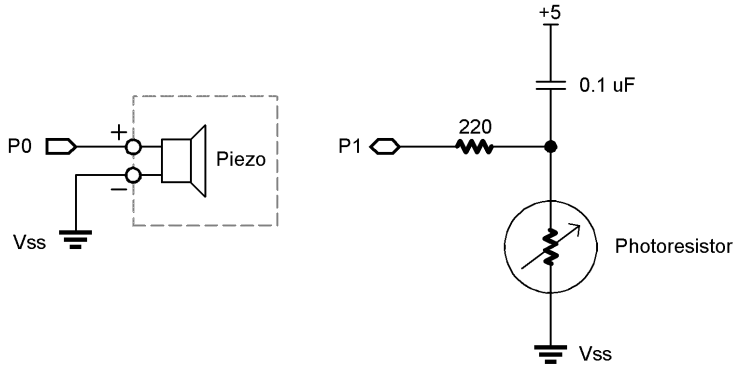
frequency = 50,000 / period * 10

Run the program and adjust the 10K pot. Notice that the **DEBUG** screen is updated without delay and that there is no "hunting" as when using **COUNT** to determine frequency.

Experiment #19
A Light-Controlled Theremin

This experiment demonstrates **FREQOUT** by creating a light-controlled Theremin (the first electronic musical instrument ever produced).

Building The Circuit



Note: Later versions of the StampWorks lab board come with a built-in audio amplifier. Attach an 8-ohm speaker to the output of the amplifier to get the best sound from this project.

```
' =============================================================================
'
'   File..... Ex19 - Theremin.BS2
'   Purpose... Simple Digital Theremin
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
```

Experiment #19: A Light-Controlled Theremin

```
' This program uses RCTIME with a photocell to create a light-controlled
' theremin.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Speaker         CON     0                       ' piezo speaker output
PitchCtrl       CON     1                       ' pitch control (RCTIME) input


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Scale           CON     $0100                   ' divider for BS2/BS2e
'Scale          CON     $0066                   ' divider for BS2sx
'Scale          CON     $0073                   ' divider for BS2p

Threshold       CON     200                     ' cutoff frequency to play


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

tone            VAR     Word                    ' frequency output


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  HIGH PitchCtrl                                ' discharge cap
  PAUSE 1                                       ' for 1 ms
  RCTIME PitchCtrl, 1, tone                     ' read the light sensor
  tone = tone */ Scale                          ' scale input

  IF (tone < Threshold) THEN Main               ' skip for ambient light
  FREQOUT Speaker, 25, tone                     ' output the tone
  GOTO Main

  END
```

Behind The Scenes

A Theremin is an interesting musical device used to create those weird, haunting sounds often heard in old horror movies. This version uses the light falling onto a photocell to create the output tone.

Since the photocell is a resistive device, **RCTIME** can be used to read its value. **FREQOUT** is used to play the note. The constant, **Threshold**, is used to control the cutoff point of the Theremin. When the photocell reading falls below this value, no sound is played. This value should be adjusted to the point where the Theremin stops playing when the photocell is not covered in ambient light.

Challenge

Add a second RC circuit using a 10K pot instead of a photocell. Use this circuit to adjust the threshold value to varying light conditions.

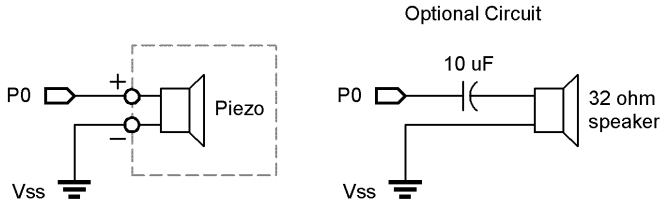
Experiment #20
Sound Effects

This experiment uses **FREQOUT** and **DTMFOUT** to create a telephone sound effects machine.

New PBASIC elements/commands to know:

- DTMFOUT

Building The Circuit



Note: Later versions of the StampWorks lab board come with a built-in audio amplifier. Attach an 8-ohm speaker to the output of the amplifier to get the best sound from this project.

```
' =====================================================================================
'
'   File...... Ex20 - Sound FX.BS2
'   Purpose... Stamp-generated sounds
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =====================================================================================


' -------------------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------------------

' This program demonstrates several realistic and interesting sound effects
' that can be generated by the BASIC Stamp using FREQOUT and DTMFOUT.  This
```

```
' program works best when played through an amplifier.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Speaker         CON     0                       ' speaker on pin 0


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

R               CON      0                      ' rest
C               CON     33                      ' ideal is 32.703
Cs              CON     35                      ' ideal is 34.648
D               CON     39                      ' ideal is 38.891
E               CON     41                      ' ideal is 41.203
F               CON     44                      ' ideal is 43.654
Fs              CON     46                      ' ideal is 46.249
G               CON     49                      ' ideal is 48.999
Gs              CON     52                      ' ideal is 51.913
A               CON     55                      ' ideal is 55.000
As              CON     58                      ' ideal is 58.270
B               CON     62                      ' ideal is 61.735

N1              CON     500                     ' whole note duration
N2              CON     N1/2                    ' half note
N3              CON     N1/3                    ' third note
N4              CON     N1/4                    ' quarter note
N8              CON     N1/8                    ' eighth note

ScaleT          CON     $0100                   ' time scale - BS2/BS2e
ScaleF          CON     $0100                   ' frequency scale - BS2/BS2e

' ScaleT        CON     $0280                   ' time scale - BS2sx
' ScaleF        CON     $0066                   ' frequency scale - BS2sx

' ScaleT        CON     $03C6                   ' time scale - BS2p
' ScaleF        CON     $0043                   ' frequency scale - BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------
```

```
x               VAR     Word                    ' loop counter
note1           VAR     Word                    ' first tone for FREQOUT
note2           VAR     Word                    ' second tone for FREQOUT
onTime          VAR     Word                    ' duration for FREQOUT
offTime         VAR     Word
oct1            VAR     Nib                     ' octave for freq1 (1 - 8)
oct2            VAR     Nib                     ' octave for freq2 (1 - 8)
eePtr           VAR     Byte                    ' EEPROM pointer
digit           VAR     Byte                    ' DTMF digit
clickDly        VAR     Word                    ' delay betweens "clicks"


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------
'
Phone1          DATA    "972-555-1212", 0       ' a stored telephone number
Phone2          DATA    "916-624-8333", 0       ' another number


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  PAUSE 250
  DEBUG CLS, "BASIC Stamp Sound FX Demo", CR, CR

Dial_Tone:
  DEBUG "Dial tone", CR
  onTime = 35 */ ScaleT
  note1 = 35 */ ScaleF
  FREQOUT Speaker, onTime, note1                ' "click"
  PAUSE 100
  onTime = 2000 */ ScaleT
  note1 = 350 */ ScaleF
  note2 = 440 */ ScaleF
  FREQOUT Speaker, onTime, note1, note2         ' combine 350 Hz & 440 Hz

Dial_Phone1:                                    ' dial phone from EE
  DEBUG "Dialing number: "
  eePtr = Phone1                                ' initialize eePtr pointer
  GOSUB Dial_Phone

Phone_Busy:
  PAUSE 1000
  DEBUG CR, " - busy...", CR
```

```
  onTime = 400 */ ScaleT
  note1 = 480 */ ScaleF
  note2 = 620 */ ScaleF
  FOR x = 1 TO 8
    FREQOUT Speaker, onTime, note1, note2        ' combine 480 Hz and 620 Hz
    PAUSE 620
  NEXT
  onTime = 35 */ ScaleT
  note1 = 35 */ ScaleF
  FREQOUT Speaker, onTime, note1                ' "click"

Dial_Phone2:
  DEBUG "Calling Parallax: "
  eePtr = Phone2
  GOSUB Dial_Phone

Phone_Rings:
  PAUSE 1000
  DEBUG CR, " - ringing"
  onTime = 2000 */ ScaleT
  note1 = 440 */ ScaleF
  note2 = 480 */ ScaleF
  FREQOUT Speaker, onTime, note1, note2         ' combine 440 Hz and 480 Hz
  PAUSE 4000
  FREQOUT Speaker, onTime, note1, note2         ' combine 440 Hz and 480 Hz
  PAUSE 2000

Camptown_Song:
  DEBUG CR, "Play a Camptown song", CR
  FOR x = 0 TO 13
    LOOKUP x, [ G, G, E, G, A, G, E, R, E, D, R, E, D, R], note1
    LOOKUP x, [ 4, 4, 4, 4, 4, 4, 4, 1, 4, 4, 1, 4, 4, 1], oct1
    LOOKUP x, [N2, N2, N2, N2, N2, N2, N2, N2, N2, N1, N2, N2, N1, N8], onTime
    GOSUB Play_1_Note
  NEXT

Howler:
  DEBUG "Howler -- watch out!!!", CR
  FOR x = 1 TO 4
    onTime = 1000 */ ScaleT
    note1 = 1400 */ ScaleF
    note2 = 2060 */ ScaleF
    FREQOUT Speaker, onTime, note1, note2       ' combine 1400 Hz and 2060 Hz
    onTime = 1000 */ ScaleT
    note1 = 2450 */ ScaleF
    note2 = 2600 */ ScaleF
    FREQOUT Speaker, onTime, note1, note2       ' combine 2450 Hz and 2600 Hz
```

```
  NEXT

Roulette_Wheel:
  DEBUG "Roulette Wheel", CR
  onTime = 5 */ ScaleT                          ' onTime for "click"
  note1 = 35 */ ScaleF                          ' frequency for "click"
  clickDly = 250                                ' starting delay between clicks
  FOR x = 1 TO 8                                ' spin up wheel
    FREQOUT Speaker, onTime, note1              ' click
    PAUSE clickDly
    clickDly = clickDly */ $00BF                ' accelerate (speed * 0.75)
  NEXT
  FOR x = 1 TO 10                               ' spin stable
    FREQOUT Speaker, onTime, note1
    PAUSE clickDly
  NEXT
  FOR x = 1 TO 20                               ' slow down
    FREQOUT Speaker, onTime, note1
    PAUSE clickDly
    clickDly = clickDly */ $010C                ' decelerate (speed * 1.05)
  NEXT
  FOR x = 1 TO 30                               ' slow down and stop
    FREQOUT Speaker, onTime, note1
    PAUSE clickDly
    clickDly = clickDly */ $0119                ' decelerate (speed * 1.10)
  NEXT

Computer_Beeps:                                 ' looks great with randmom LEDs
  DEBUG "50's Sci-Fi Computer", CR
  FOR x = 1 TO 50                               ' run about 5 seconds
    onTime = 50 */ ScaleT
    RANDOM note1                                ' create random note
    note1 = (note1 // 2500) */ ScaleF           ' don't let note go to high
    FREQOUT Speaker, onTime, note1              ' play it
    PAUSE 100                                   ' short pause between notes
  NEXT

Space_Transporter:
  DEBUG "Space Transporter", CR
  onTime = 10 */ ScaleT
  FOR x = 5 TO 5000 STEP 5                      ' frequency sweep up
    note1 = x */ ScaleF
    FREQOUT Speaker, onTime, note1, note1 */ 323
  NEXT
  FOR x = 5000 TO 5 STEP 50                     ' frequency sweep down
    note1 = x */ ScaleF
    FREQOUT Speaker, onTime, note1, note1 */ 323
```

```
  NEXT

  DEBUG CR, "Sound demo complete."
  INPUT Speaker

  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Dial_Phone:
  READ eePtr, digit                             ' read a digit
  IF (digit = 0) THEN Dial_Exit                 ' when 0, number is done
  DEBUG digit                                   ' display digit
  IF (digit < "0") THEN Next_Digit              ' don't dial non-digits
  onTime = 150 */ ScaleT
  offTime = 75 */ ScaleT
  DTMFOUT Speaker, onTime, offTime, [digit - 48]

Next_Digit:
  eePtr = eePtr + 1                             ' update eePtr pointer
  GOTO Dial_Phone                               ' get another digit

Dial_Exit:
  RETURN


Play_1_Note:
  note1 = note1 << (oct1 - 1)                   ' get frequency for note + octave
  onTime = onTime */ ScaleT
  note1 = note1 */ ScaleF
  FREQOUT Speaker, onTime, note1                ' play it
  RETURN


Play_2_Notes:
  note1 = note1 << (oct1 - 1)                   ' get frequency for note + octave
  note2 = note2 << (oct2 - 1)                   ' get frequency for note + octave
  onTime = onTime */ ScaleT
  note1 = note1 */ ScaleF
  note2 = note2 */ ScaleF
  FREQOUT Speaker, onTime, note1, note2         ' play both
  RETURN
```

Behind The Scenes

The a bit of programming creativity, the BASIC Stamp is able to create some very interesting sound effects. Since most of the sounds we hear on the telephone (other than voice) are generated with two tones, the BASIC Stamp's **FREQOUT** and **DTMFOUT** functions can be used to generate telephone sound effects.

DTMFOUT is actually a specialized version of **FREQOUT**. Its purpose is to play the dual-tones required to dial a telephone. Instead of passing a tone (or tones), the digit(s) to be dialed are passed as parameters. In actual dialing applications, the DTMF on-time and off-time can be specified to deal with telephone line quality.

This program also presents the BASIC Stamp's basic musical ability by playing a simple song. Constants for note frequency (in the first octave) and note timing simplify the operational code. The **Play_1_Note** subroutine adjusts note frequency for the specified octave. The musical quality can suffer a bit in the higher octaves because of rounding errors. Using the ideal values shown, the constants table can be expanded to create accurate musical notes. Keep in mind that each octave doubles the frequency of a note.

Octave 2 = Octave 1 * 2
Octave 3 = Octave 2 * 2
Octave 4 = Octave 3 * 2

And so on…

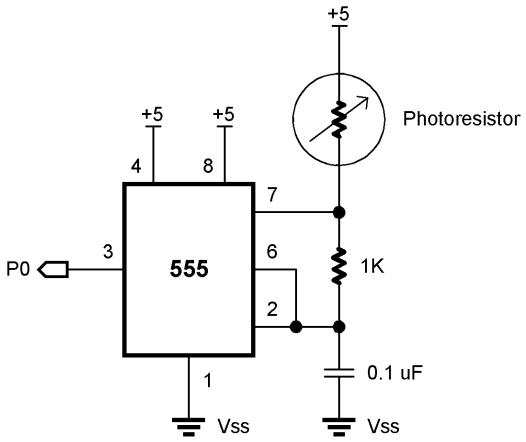
Challenge

Convert (a portion of) your favorite song to play on the BASIC Stamp.

Experiment #21
Analog Input with PULSIN

The experiment reads a resistive component using **PULSIN** and a free-running oscillator.

Building The Circuit (Note that schematic is NOT chip-centric)




```
' =============================================================================
'
'   File..... Ex21 - AnalogIn.BS2
'   Purpose... Analog input using PULSIN
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program "reads" an analog value by using that component to control the
```

```
' output frequency of a 555-based oscillator.  PULSIN is used to measure the
' high portion of the signal as it is controlled by the varialbe resistance.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

PulseInput      CON     0


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

P75             CON     $00C0                   ' 0.75
P50             CON     $0080                   ' 0.50
P25             CON     $0040                   ' 0.25


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

rValue          VAR     Word                    ' raw value
sValue          VAR     Word                    ' smoothed value


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  PULSIN PulseInput, 1, rValue                  ' get high portion of input
  sValue = (rValue */ P25) + (sValue */ P75)

  DEBUG Home
  DEBUG "Raw value... ", DEC rValue, "     ", CR
  DEBUG "Filtered.... ", DEC sValue, "     "

  GOTO Main                                     ' do it again
```

Behind The Scenes

In this experiment, the 555 is configured as an oscillator. Analyzing the output, the width of the low portion of the output is controlled by the resistance of the photocell. By measuring the low portion of

the 555's output signal with **PULSIN**, the BASIC Stamp is able to determine the relative value of the photocell.

Once the raw value is available, adding a portion of the raw value with a portion of the last filtered value digitally filters it. The ratio of raw-to-filtered readings in this equation will determine the responsiveness of the filter. The larger the raw portion, the faster the filter.

Challenge

Create a final output value that is scaled so that its range is between zero and 1000.

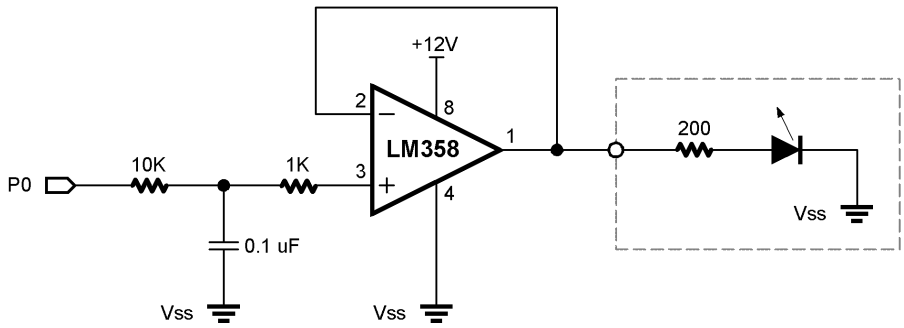
Experiment #22: Analog Output with PWM

This program shows how create a variable voltage output with PWM.

New PBASIC elements/commands to know:

- PWM

Building The Circuit



Note that this circuit requires 12V. The only place you can get 12V on the StampWorks lab board is from the +V screw terminal at the high-current driver location.

```
' =============================================================================
'
'   File..... Ex22 - Throb.BS2
'   Purpose... Output a variable voltage with PWM
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

Experiment #22: Analog Output with PWM

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates how the PWM command can be used with an opamp
' buffer to create a variable voltage output.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

D2Aout          CON     0                       ' analog out pin


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

OnTime          CON     10                      ' 10 milliseconds, BS2
'OnTime         CON     25                      ' BS2sx
'OnTime         CON     15                      ' BS2p


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

level           VAR     Byte                    ' analog level


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR level = 0 TO 255                          ' increase voltage to LED
    PWM D2Aout, level, OnTime
  NEXT

  PAUSE 250

  FOR level = 255 TO 0                          ' decrease voltage to LED
    PWM D2Aout, level, OnTime
  NEXT

  GOTO Main                                       ' do it again
```

Behind The Scenes

While most BASIC Stamp applications will deal with digital signals, some will require analog output; a variable voltage between zero and some maximum voltage. The BASIC Stamp's **PWM** function is designed to generate analog voltages when combined with an R/C filter. The **PWM** function outputs a series of pulses which have a programmable on-time to off-time ratio (duty cycle). The greater the duty cycle, the greater voltage output. A duty cycle of 255 will charge the capacitor to five volts.

In this experiment, one half of the LM358 is used to provide a buffered voltage to the LED. The op-amp buffer prevents the capacitor from discharging too quickly under load. The LED brightness and dims because the changing voltage through its series resistor changes the current through the LED. Notice that the LED seems to snap on and get brighter, then dim to a level and snap off. This happens when the output of the LM358 crosses the forward voltage threshold (the minimum voltage for the LED to light) of the LED (about 1.8 volts).

Using the digital multimeter, monitor Pin 1 of the LM358.

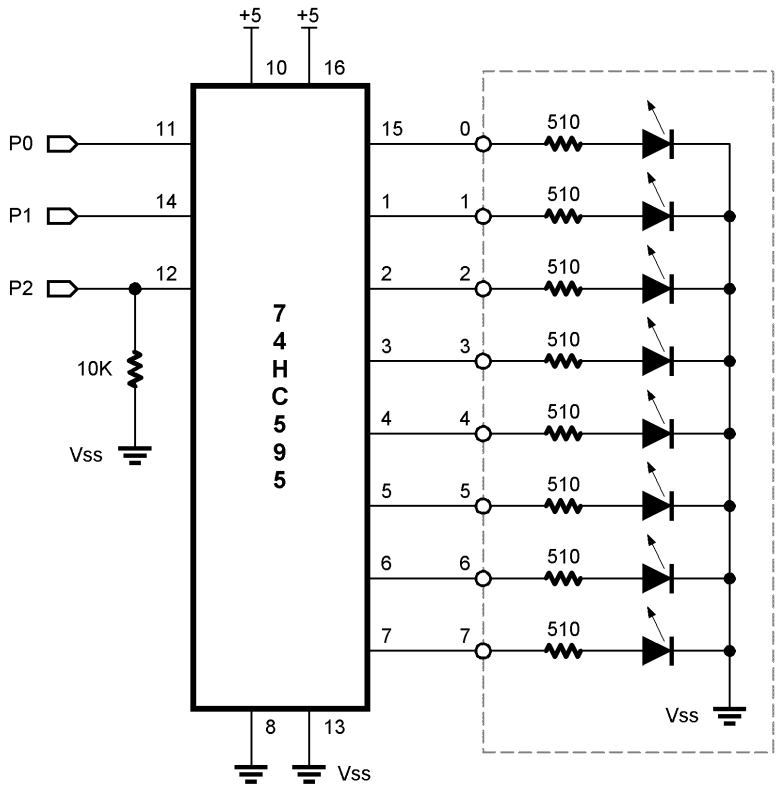
Experiment #23: Expanding Outputs

This experiment demonstrates the expansion of BASIC Stamp outputs with a simple shift register. Three lines are used to control eight LEDs with a 74x595 shift register.

New PBASIC elements/commands to know:

- SHIFTOUT

Building The Circuit (Note that schematic is NOT chip-centric)



Experiment #23b: Expanded Outputs

```
' =============================================================================
'
'   File..... Ex23 - 74HC595.BS2
'   Purpose... Expanded outputs with 74HC595
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demostrates a simple method of turning three Stamp lines into
' eight outputs with a 74HC595 shift register.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Clock           CON     0                       ' shift clock (74HC595.11)
DataOut         CON     1                       ' serial data out (74HC595.14)
Latch           CON     2                       ' output latch (74HC595.12)


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DelayTime       CON     100


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

pattern         VAR     Byte                    ' output pattern
```

```
' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  LOW Latch                                     ' make output and keep low
  pattern = %00000001


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Go_Forward:
  GOSUB Out_595
  PAUSE DelayTime                               ' put pattern on 74x595
  pattern = pattern << 1                        ' shift pattern to the left
  IF (pattern = %10000000) THEN Go_Reverse      ' test for final position
  GOTO Go_Forward                               ' continue in this direction

Go_Reverse:
  GOSUB Out_595
  PAUSE DelayTime
  pattern = pattern >> 1
  IF (pattern = %00000001) THEN Go_Forward
  GOTO Go_Reverse


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Out_595:
  SHIFTOUT DataOut, Clock, MSBFirst, [pattern]  ' send pattern to 74x595
  PULSOUT Latch, 5                              ' latch outputs
  RETURN
```

Behind The Scenes

The BASIC Stamp is extraordinarily flexible in its ability to redefine the direction (input or output) of its I/O pins, yet very few applications require this flexibility. For the most part, microcontroller applications will define pins as either inputs or outputs at initialization and the definitions will remain unchanged through the program.

Experiment #23b: Expanded Outputs

We can use the fact that outputs are outputs and conserve valuable BASIC Stamp I/O lines at the same time by using a simple component called a serial-in, parallel-out shift register. In this experiment, the 74x595 is used. With just three BASIC Stamp lines, this program is able to control eight LEDs through the 74x595.

The 74x595 converts a synchronous serial data stream to eight parallel outputs. Synchronous serial data actually has two components: the serial data and a serial clock. The BASIC Stamp's **SHIFTOUT** command handles the details of the data and clock lines and writes data to a synchronous device, in this case, the 74x595. With the 74x595, the data must be latched to the outputs after the shift process. Latching is accomplished by briefly pulsing the Latch control line. This prevents the outputs from "rippling" as new data is being shifted in.

Being serial devices, shift registers can be cascaded. By cascading, the BASIC Stamp is able to control dozens of 74x595 outputs with the same three control lines. To connect cascaded 74x595s, the clock and latch lines are all tied together and the SQ output from one stage connects to the serial input of the next stage.

Experiment #23b: Expanding Outputs

This experiment demonstrates further expansion of BASIC Stamp outputs by cascading two 75x595 shift registers.

(Schematic on the next page)

Behind The Scenes

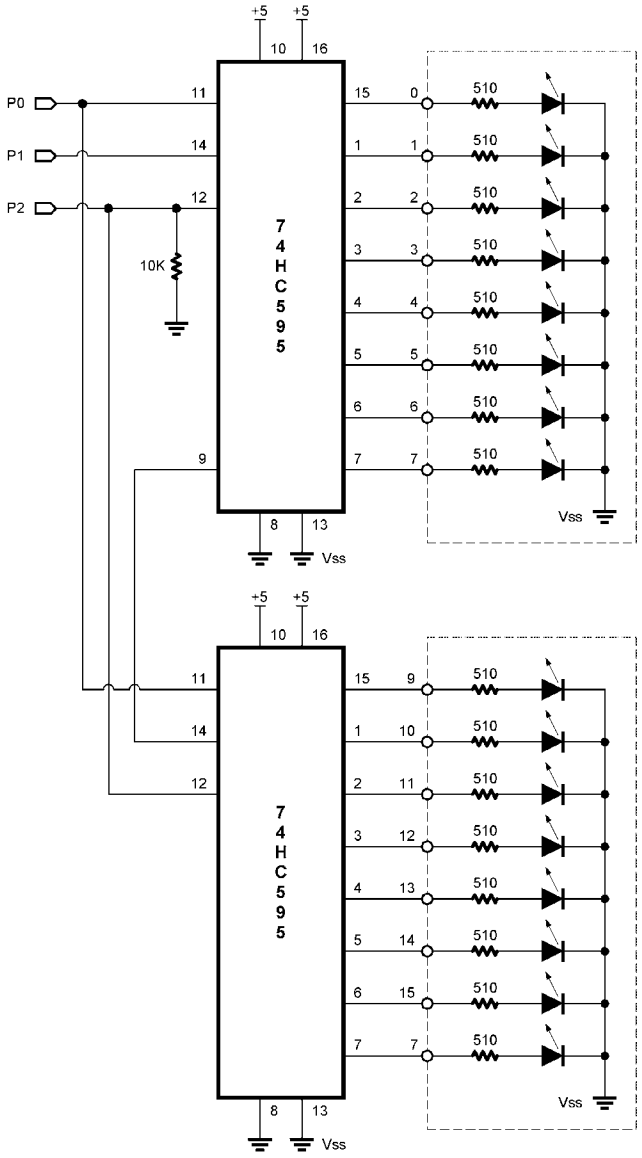
The 75x595 has a Serial Output pin (9) that allows the cascading of multiple devices for more outputs. In this configuration, the Clock and Latch pins are shared to keep all devices synchronized.

When cascading multiple shift registers, you must send the data for the device that is furthest down the chain first. Subsequent SHIFTOUT sequences will "push" the data through each register until the data is loaded into the correct device. Applying the latch pulse at that point causes the new data in all shift registers to appear at the outputs.

The demo program illustrates this point by independently displaying a binary counter and a ping-pong visual display using two 75x595 shift registers and eight LEDs for each. Note that the counter display is controlled by the 75x595 that is furthest from the BASIC Stamp, so its data is shifted out first.

Experiment #23b: Expanded Outputs

Building The Circuit (Note that schematic is NOT chip-centric)



```
' =============================================================================
'
'   File..... Ex23b - 74HC595 x 2.BS2
'   Purpose... Expanded outputs with 74HC595
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demostrates a simple method of turning three Stamp lines into
' 16 outputs with two 74HC595 shift registers.  The data lines into the second
' 74HC595 is fed by the SQh output (pin 9) of the first.  The clock and latch
' pins of the second 74HC595 are connected to the same pins on the first.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

DataOut         CON     0                       ' serial data out (74HC595.14)
Clock           CON     1                       ' shift clock (74HC595.11)
Latch           CON     2                       ' output latch (74HC595.12)


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DelayTime       CON     100


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

pattern         VAR     Byte                    ' output pattern
counter         VAR     Byte
```

Experiment #23b: Expanded Outputs

```
' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  LOW Latch                                     ' make output and keep low
  pattern = %00000001


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Go_Forward:
  counter = counter + 1                         ' update counter
  GOSUB Out_595
  PAUSE DelayTime                               ' put pattern on 74x595
  pattern = pattern << 1                        ' shift pattern to the left
  IF (pattern = %10000000) THEN Go_Reverse      ' test for final position
  GOTO Go_Forward                               ' continue in this direction

Go_Reverse:
  counter = counter + 1
  GOSUB Out_595
  PAUSE DelayTime
  pattern = pattern >> 1
  IF (pattern = %00000001) THEN Go_Forward
  GOTO Go_Reverse


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Out_595:
  SHIFTOUT DataOut, Clock, MSBFirst, [counter]  ' send counter to 2nd 74HC595
  SHIFTOUT DataOut, Clock, MSBFirst, [pattern]  ' send pattern to 1st 74HC595
  PULSOUT Latch, 5                              ' latch outputs
  RETURN
```

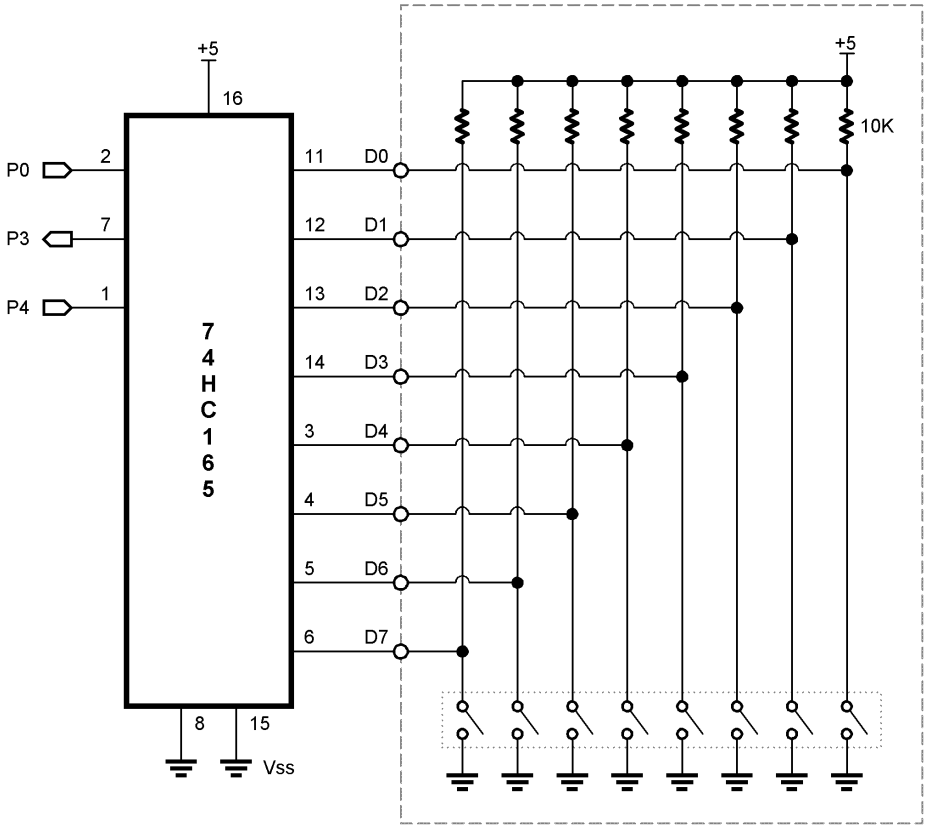
Experiment #24: Expanding Inputs

This experiment demonstrates the expansion of BASIC Stamp inputs with a simple shift register. Three lines are used to read an eight-position DIP-switch.

New PBASIC elements/commands to know:

- SHIFTIN

Building The Circuit (Note that schematic is NOT chip-centric)



Experiment #24b: Expanded Inputs

```
' =============================================================================
'
'   File..... Ex24 - 74HC165.BS2
'   Purpose... Input expansion with 74HC165
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program shows how to read eight inputs with just three Stamp pins using
' a 74HC165 shift register.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Clock           CON     0                       ' shift clock (74x165.2)
DataIn          CON     3                       ' shift data (74x165.7)
Load            CON     4                       ' input load (74x165.1)


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

switches        VAR     Byte                    ' inputs switches


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  HIGH Load                                     ' make output; initialize to 1
```

```
' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Read_165                                ' read 8-pos dip switch
  DEBUG Home, "Switches = ", BIN8 switches      ' display binary mode
  PAUSE 100
  GOTO Main                                     ' do it again


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Read_165:
  PULSOUT Load, 5                               ' grab the switch inputs
  SHIFTIN DataIn, Clock, MSBPre, [switches]     ' shift them in
  RETURN
```

Behind The Scenes

The experiment demonstrates **SHIFTIN**, the complimentary function to **SHIFTOUT**. In this case, three BASIC Stamp I/O lines are used to read the state of eight input switches. To read the data from the 74x165, the parallel inputs are latched by briefly pulsing the Load line, then using **SHIFTIN** to move the data into the BASIC Stamp.

Note that the DIP-switches are pulled-up to Vdd, so setting them to "ON" creates a logic low input to the shift register. By using the Q\ (inverted) output from the 74x165, the data arrives at the BASIC Stamp with Bit 1 indicating a switch is on.

Experiment #24b: Expanding Inputs

This experiment demonstrates further expansion of BASIC Stamp inputs by cascading two shift registers.

(Schematic on next page)

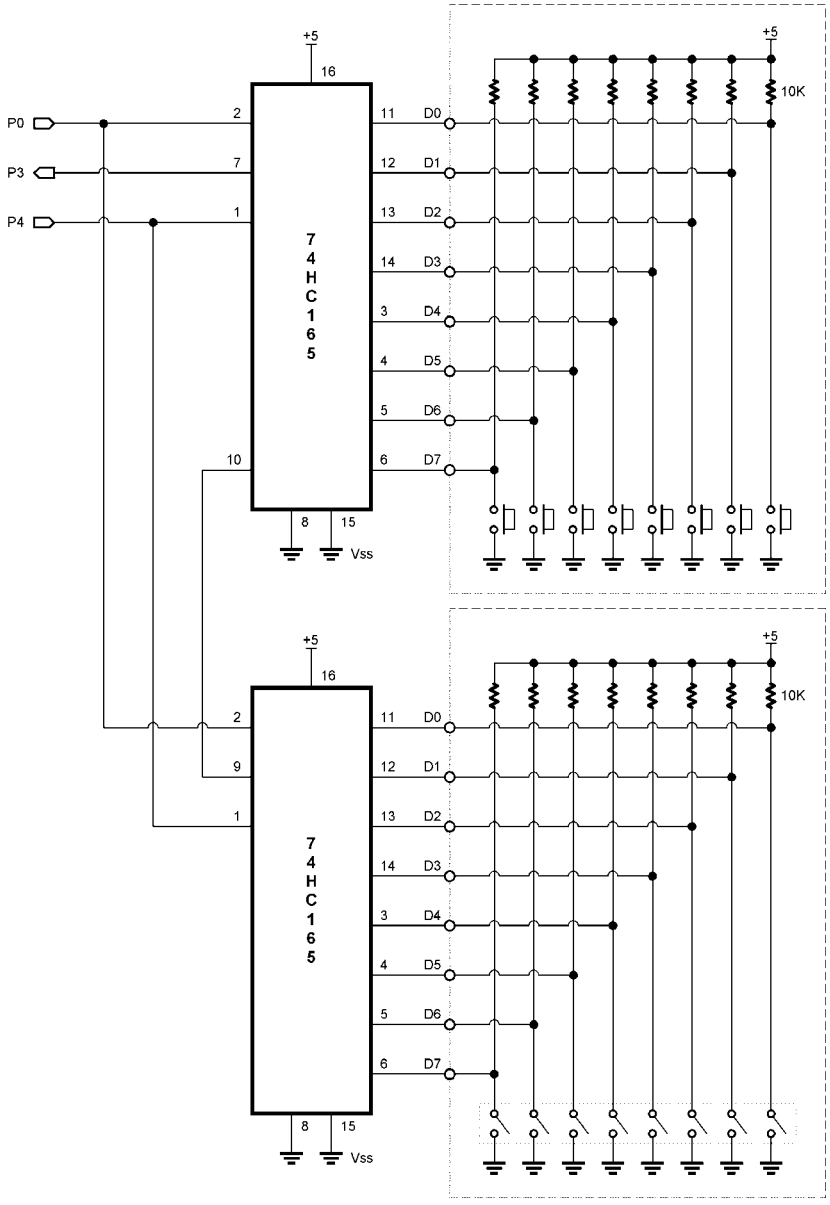
Behind The Scenes

This program is very similar to 23b in that the Serial Output (pin 9) from one shift register is fed into the Serial input (pin 10) of the next device up the chain. Note that the non-inverted output is used on the second 74x165 because the inverted output of the deice connected directly to the BASIC Stamp will take care of the inversion.

In the program the **Read_165** subroutine has been updated to accommodate the second 74x165. The first **SHIFTIN** loads the data from the "buttons" shift register into the BASIC Stamp and transfers the contents from the "switches" shift register into the "buttons" shift register. The second **SHIFTIN** loads the "switches" data into the BASIC Stamp.

Experiment #24b: Expanded Inputs

Building The Circuit (Note that schematic is NOT chip-centric)



```
' =============================================================================
'
'   File..... Ex24b - 74HC165 x 2.BS2
'   Purpose... Input expansion with 74HC165
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program shows how to read 16 inputs with just three Stamp pins using
' two 74HC165 shift registers.  The serial output (pin 9) from one 74HC165
' is fed into the serial input (pin 10) of the second.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

Clock           CON     0                       ' shift clock (74x165.2)
DataIn          CON     3                       ' shift data (74x165.7)
Load            CON     4                       ' input load (74x165.1)


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

switches        VAR     Byte                    ' inputs switches
buttons         VAR     Byte                    ' push button inputs


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  HIGH Load                                     ' make output; initialize to 1
```

Experiment #24b: Expanded Inputs

```
' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Read_165                                ' read switches and buttons
  DEBUG Home
  DEBUG "Buttons = ", BIN8 buttons, CR          ' display binary mode
  DEBUG "Swithces =  ", BIN8 switches
  PAUSE 100
  GOTO Main                                     ' do it again


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Read_165:
  PULSOUT Load, 5                               ' latch inputs
  SHIFTIN DataIn, Clock, MSBPre, [buttons]      ' get buttons
  SHIFTIN DataIn, Clock, MSBPre, [switches]     ' get switches
  RETURN
```

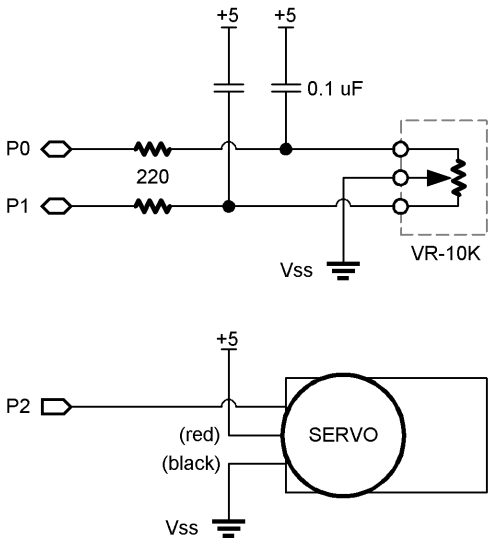
Experiment #25: Hobby Servo Control

This experiment demonstrates the control of a standard hobby servo. Hobby servos frequently are used in amateur robotics.

New PBASIC elements/commands to know:

- SDEC, SDEC1 – SDEC16 (DEBUG modifier)

Building The Circuit




```
' =============================================================================
'
'   File..... Ex25 - Servo.BS2
'   Purpose... Hobby Servo Control
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
```

```
'
' ===============================================================================


' -------------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------------

' This program shows how to control a standard servo with the BASIC Stamp.


' -------------------------------------------------------------------------------
' I/O Definitions
' -------------------------------------------------------------------------------

PotCW           CON     0                       ' clockwise pot input
PotCCW          CON     1                       ' counter-clockwise pot input
Servo           CON     2                       ' servo control pin


' -------------------------------------------------------------------------------
' Constants
' -------------------------------------------------------------------------------

Scale           CON     $0068                   ' scale RCTIME to 0 - 250, BS2
' Scale         CON     $002C                   ' BS2sx
' Scale         CON     $002A                   ' BS2p


' -------------------------------------------------------------------------------
' Variables
' -------------------------------------------------------------------------------

rcRt            VAR     Word                    ' rc reading - right
rcLf            VAR     Word                    ' rc reading - left
diff            VAR     Word                    ' difference between readings
sPos            VAR     Word                    ' servo position


' -------------------------------------------------------------------------------
' Program Code
' -------------------------------------------------------------------------------

Main:
  HIGH PotCW                                    ' discharge caps
  HIGH PotCCW
  PAUSE 1
```

```
RCTIME PotCW, 1, rcRt                          ' read clockwise
RCTIME PotCCW, 1, rcLf                         ' read counter-clockwise

rcRt = (rcRt */ Scale) MAX 250                 ' scale RCTIME to 0-250
rcLf = (rcLf */ Scale) MAX 250
sPos = rcRt - rcLf                             ' calculate position (-250 to 250)

PULSOUT Servo, (750 + sPos)                    ' move the servo
PAUSE 20

GOTO Main
```

Behind The Scenes

Hobby servos are specialized electromechanical devices used most frequently to position the control surfaces of model aircraft. The position of the servo output shaft is determined by the width of an incoming control pulse. The control pulse is typically between one and two milliseconds wide. The servo will center when the control signal is 1.5 milliseconds. In order to maintain its position, the servo must constantly be updated. The typical update frequency is about 50 times per second.
The BASIC Stamp's **PULSOUT** command is ideal command for controlling hobby servos. In this experiment, two **RCTIME** circuits are constructed around the 10K potentiometer. This circuit and the project code can be used to determine the relative position of the potentiometer. The readings from each side of the potentiometer are scaled between 0 and 250 with the ***/** and **MAX** operators. By subtracting one side from the other, a servo position value between –250 and +250 is returned.

This value is added to the centering position of 750. Remember that **PULSOUT** works in two-microsecond units, so a **PULSOUT** value of 750 will create a pulse that is 1.5 milliseconds wide, causing the servo to center. When the servo position is –250, the **PULSOUT** value is 500, creating a 1.0-millisecond pulse. At an **sPos** value of +250, the **PULSOUT** value is 1000, creating a 2.0 millisecond control pulse.

This code demonstrates that the BASIC Stamp does, indeed, work with negative numbers. You can see the value of **sPos** by inserting this line after the calculation:

```
DEBUG Home, "Position: ", SDEC sPos, "   "
```

Negative numbers are stored in two's compliment format. The **SDEC** (signed decimal) modifier prints standard decimal with the appropriate sign.

Experiment #25: Hobby Servo Control

Challenge

Replace the potentiometer with two photocells and update the code to cause the servo to center at the brightest light source.

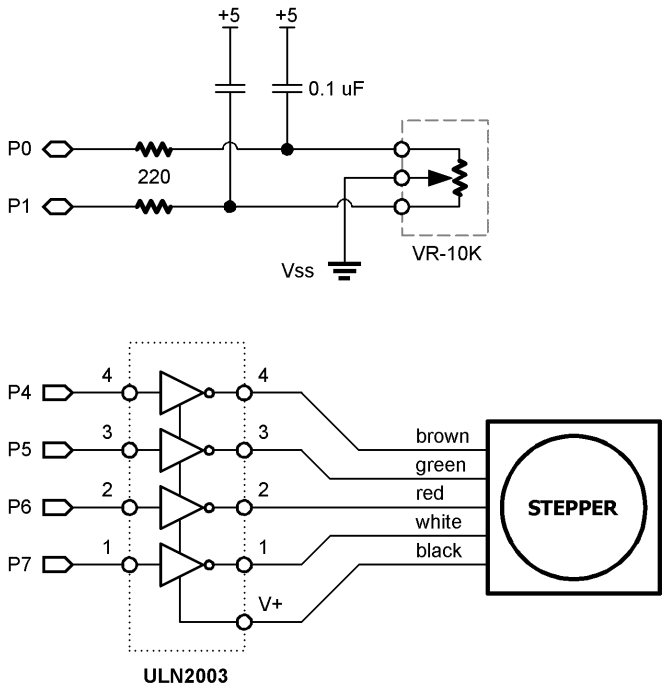
Experiment #26: Stepper Motor Control

This experiment demonstrates the control of a small 12-volt unipolar stepper motor. Stepper motors are used as precision positioning devices in robotics and industrial control applications.

New PBASIC elements/commands to know:

- ABS

Building The Circuit



```
' =============================================================================
'
'   File..... Ex26 - Stepper.BS2
'   Purpose... Stepper Motor Control
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates unipolar stepper motor control.  The pot allows the
' program to control speed and direction of the motor.


' -----------------------------------------------------------------------------
' Revision History
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

PotCW           CON     0                       ' clockwise pot input
PotCCW          CON     1                       ' counter-clockwise pot input
Coils           VAR     OutB                    ' output to stepper coils


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Scale           CON     $0100                   ' scale for BS2 (1.0)
' Scale         CON     $0080                   ' scale for BS2sx, BS2p (0.5)


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------
```

```
speed           VAR     Word                    ' delay between steps
x               VAR     Byte                    ' loop counter
sAddr           VAR     Byte                    ' EE address of step data
rcRt            VAR     Word                    ' rc reading - right
rcLf            VAR     Word                    ' rc reading - left
diff            VAR     Word                    ' difference between readings


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------
'                         __
'                        ABAB
'                       -----
Step1           DATA    %1100                   ' A on   B on   A\ off  B\ off
Step2           DATA    %0110                   ' A off  B on   A\ on   B\ off
Step3           DATA    %0011                   ' A off  B off  A\ on   B\ on
Step4           DATA    %1001                   ' A on   B off  A\ off  B\ on


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirB = %1111                                  ' make stepper pins outputs
  speed = 5                                     ' set starting speed


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR x = 1 TO 100                              ' 1 rev forward
    GOSUB Step_Fwd
  NEXT
  PAUSE 200

  FOR x = 1 TO 100                              ' 1 rev back
    GOSUB Step_Rev
  NEXT
  PAUSE 200

Step_Demo:
  HIGH PotCW                                    ' discharge caps
```

```
  HIGH PotCCW
  PAUSE 1
  RCTIME PotCW, 1, rcRt                         ' read clockwise
  RCTIME PotCCW, 1, rcLf                        ' read counter-clockwise

  rcRt = (rcRt */ Scale) MAX 600                ' set speed limits
  rcLf = (rcLf */ Scale) MAX 600
  diff = ABS(rcRt - rcLf)                       ' get difference between readings

  IF (diff < 25) THEN Step_Demo                 ' allow dead band
  IF (rcLf > rcRt) THEN Step_CCW

Step_CW:
  speed = 60 - (rcRt / 10)                      ' calculate speed
  GOSUB Step_Fwd                                ' do a step
  GOTO Step_Demo

Step_CCW:
  speed = 60 - (rcLf / 10)
  GOSUB Step_Rev
  GOTO Step_Demo


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Step_Fwd:
  sAddr = sAddr + 1 // 4                        ' point to next step
  READ (Step1 + sAddr), Coils                   ' output step data
  PAUSE speed                                   ' pause between steps
  RETURN


Step_Rev:
  sAddr = sAddr + 3 // 4                        ' point to previous step
  READ (Step1 + sAddr), Coils
  PAUSE speed
  RETURN
```

Behind The Scenes

Stepper motors differ from standard DC motors in that they do not spin freely when power is applied. For a stepper motor to rotate, the power source must be continuously pulsed in specific patterns. The step sequence (pattern) determines the direction of the stepper's rotation. The time between sequence steps determines the rotational speed. Each step causes the stepper motor to rotate a fixed angular increment. The stepper motor supplied with the StampWorks kit rotates 3.6 degrees per step. This means that one full rotation (360 degrees) of the stepper requires 100 steps.

The step sequences for the motor are stored in **`DATA`** statements. The **`StepFwd`** subroutine will read the next sequence from the table to be applied to the coils. The **`StepRev`** subroutine is identical except that it will read the previous step. Note the trick with the modulus (//) operator used in **`StepRev`**. By adding the maximum value of the sequence to the current value and then applying the modulus operator, the sequence goes in reverse. Here's the math:

```
0 + 3 // 4 = 3
3 + 3 // 4 = 2
2 + 3 // 4 = 1
1 + 3 // 4 = 0
```

This experiment reads both sides of the 10K potentiometer to determine its relative position. The differential value between the two readings is kept positive by using the **`ABS`** function. The position is used to determine the rotational direction and the strength of the position is used to determine the rotational speed. Remember, the shorter the delay between steps, the faster the stepper will rotate. A dead-band check is used to cause the motor to stop rotating when the **`RCTIME`** readings are nearly equal.

Challenge

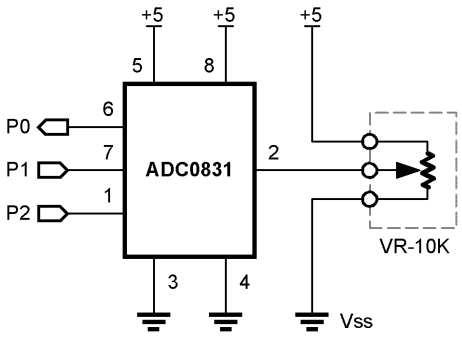
Rewrite the program to run the motor in 200 half steps. Here's the step sequence:

```
Step1 = %1000
Step2 = %1100
Step3 = %0100
Step4 = %0110
Step5 = %0010
Step6 = %0011
Step7 = %0001
Step8 = %1001
```

Experiment #27: Voltage Measurement

This experiment demonstrates the use of an analog-to-digital converter to read a variable voltage input.

Building The Circuit (Note that schematic is NOT chip-centric)




```
' =============================================================================
'
'   File..... Ex27 - ADC0831.BS2
'   Purpose... Analog to Digital conversion
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program deomstrates reading a variable voltage with an ADC0831 analog-
' to-digital convertor chip.
```

```
' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

A2Ddata         CON     0                       ' A/D data line
A2Dclock        CON     1                       ' A/D clock
A2Dcs           CON     2                       ' A/D chip select (low true)


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

result          VAR     Byte                    ' result of conversion
mVolts          VAR     Word                    ' convert to millivolts


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  HIGH A2Dcs


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Read_0831
  mVolts = result */ $139C                      ' x 19.6 (mv / unit)

  DEBUG Home
  DEBUG "ADC..... ", DEC result, "   ", CR
  DEBUG "volts... ", DEC mVolts DIG 3, ".", DEC3 mVolts

  PAUSE 100                                     ' delay between readings
  GOTO Main                                     ' do it again
```

```
' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Read_0831:
  LOW A2Dcs
  SHIFTIN A2Ddata, A2Dclock, MSBPost, [result\9]
  HIGH A2Dcs
  RETURN
```

Behind The Scenes

Previous projects have used **RCTIME** to read resistive components. This is a form of analog input, but isn't voltage measurement. For that, the BASIC Stamp needs help from an external device. The simplest way to measure a variable voltage is with an analog-to-digital converter.

In this experiment, the National Semiconductor ADC0831 is used to convert a voltage (0 – 5) to a synchronous serial signal that can be read by the BASIC Stamp with **SHIFTIN**. The nature of the ADC0831 requires nine bits to shift in the result. This is no problem for the BASIC Stamp as the **SHIFTIN** function allows the number of shifted bits to be specified.

The eight-bit result will be from zero (zero volts) to 255 (five volts). Dividing five (volts) by 255, we find that each bit in the result is equal to 19.6 millivolts. For display purposes, the result is converted to millivolts by multiplying by 19.6 (result */ $139C). A neat trick with **DEBUG** is used to display the variable, **mVolts**. The "**DIG 3**" operation prints the whole volts and the **DEC3** modifier prints the fractional volts.

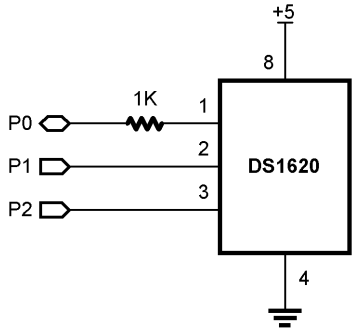
Challenge

Connect the output of Experiment 22 (Pin 1 of the LM358) to the input of the ADC0831. Write a program to create a voltage using **PWM** and read it back with the ADC0831.

Experiment #28: Temperature Measurement

This experiment demonstrates the use of a digital temperature sensor. Temperature measurement is a necessary component of environmental control applications (heating and air conditioning).

Building The Circuit (Note that schematic is NOT chip-centric)




```
' =============================================================================
'
'   File..... Ex28 - DS1620.BS2
'   Purpose... Temperature measurement
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program measures temperature using the Dallas Semiconductor DS1620
' temperature sensor.
```

```
' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

DQ              CON     0                       ' DS1620.1 (data I/O)
Clock           CON     1                       ' DS1620.2
Reset           CON     2                       ' DS1620.3


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

RdTmp           CON     $AA                     ' read temperature
WrHi            CON     $01                     ' write TH (high temp)
WrLo            CON     $02                     ' write TL (low temp)
RdHi            CON     $A1                     ' read TH
RdLo            CON     $A2                     ' read TL
StartC          CON     $EE                     ' start conversion
StopC           CON     $22                     ' stop conversion
WrCfg           CON     $0C                     ' write config register
RdCfg           CON     $AC                     ' read config register


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

tempIn          VAR     Word                    ' raw temperature
sign            VAR     tempIn.Bit8             ' 1 = negative temperature
tSign           VAR     Bit
tempC           VAR     Word                    ' Celsius
tempF           VAR     Word                    ' Fahrenheit


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  HIGH Reset                                    ' alert the DS1620
  SHIFTOUT DQ, Clock, LSBFirst, [WrCfg, %10]    ' use with CPU; free-run
  LOW Reset
  PAUSE 10
  HIGH Reset
  SHIFTOUT DQ, Clock, LSBFirst, [StartC]               ' start conversions
  LOW Reset
```

```
' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Get_Temperature                          ' read the DS1620

  DEBUG Home
  DEBUG "DS1620", CR
  DEBUG "------", CR
  DEBUG SDEC tempC, " C    ", CR
  DEBUG SDEC tempF, " F    ", CR

  PAUSE 1000                                    ' pause between readings
  GOTO Main


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Get_Temperature:
  HIGH Reset                                    ' alert the DS1620
  SHIFTOUT DQ, Clock, LSBFIRST, [RdTmp]         ' give command to read temp
  SHIFTIN DQ, Clock, LSBPRE, [tempIn\9]         ' read it in
  LOW Reset                                     ' release the DS1620

  tSign = sign                                  ' save sign bit
  tempIn = tempIn / 2                           ' round to whole degrees
  IF (tSign = 0) THEN No_Neg1
  tempIn = tempIn | $FF00                       ' extend sign bits for negative

No_Neg1:
  tempC = tempIn                                ' save Celsius value
  tempIn = tempIn */ $01CC                      ' multiply by 1.8
  IF (tSign = 0) THEN No_Neg2                   ' if negative, extend sign bits
  tempIn = tempIn | $FF00

No_Neg2:
  tempIn = tempIn + 32                          ' finish C -> F conversion
  tempF = tempIn                                ' save Fahrenheit value
  RETURN
```

Experiment #28: Temperature Measurement

Behind The Scenes

The largest organ of the human body is the skin and it is most readily affected by temperature. Little wonder then that so much effort is put into environmental control systems (heating and air conditioning).

This experiment uses the Dallas Semiconductor DS1620 digital thermometer/thermostat chip. This chip measures temperature and makes it available to the BASIC Stamp through a synchronous serial interface. The DS1620 is an intelligent device and, once programmed, is capable of stand-alone operation using the T(com), T(hi) and T(lo) outputs.

The DS1620 requires initialization before use. In active applications like this, the DS1620 is configured for free running with a CPU. After the configuration data is sent to the DS1620, a delay of 10 milliseconds is required so that the configuration can be written to the DS1620's internal EEPROM. After the delay, the DS1620 is instructed to start continuous conversions. This will ensure a current temperature reading when the BASIC Stamp requests it.

To retrieve the current temperature, the Read Temperature ($AA) command byte is sent to the DS1620. Then the latest conversion value is read back. The data returned is nine bits wide. Bit8 indicates the sign of the temperature. If negative (sign bit is 1), the other eight bits hold the two's compliment value of the temperature. Whether negative or positive, each bit of the temperature is equal to 0.5 degrees Celsius.

The Celsius temperature is converted to whole degrees by dividing by two. If negative, the upper-byte bits are set to 1 so that the value will print properly with SDEC (signed numbers in the BASIC Stamp must be 16 bits in length). The temperature is converted to Fahrenheit using the standard formula:

$$F = (C * 1.8) + 32$$

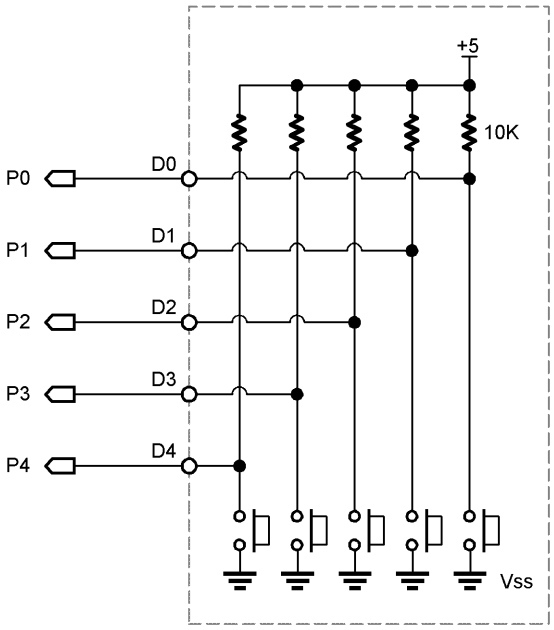
Challenge

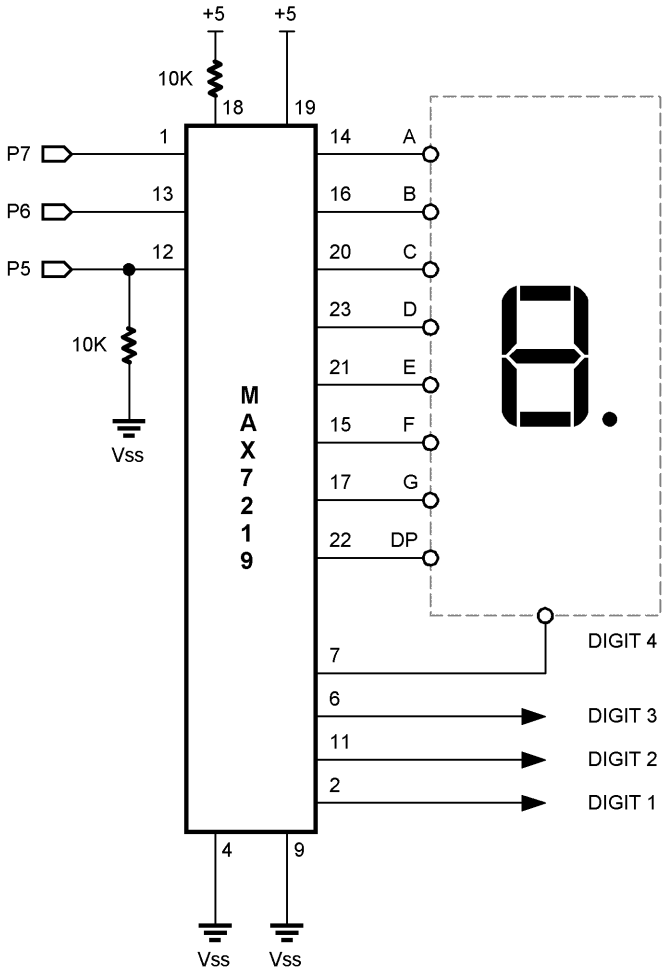
Rewrite the program to write the temperature values to the StampWorks LCD module.

Experiment #29: Advanced 7-Segment Multiplexing

This experiment demonstrates the use of seven-segment displays with an external multiplexing controller. Multi-digit seven-segment displays are frequently used on vending machines to display the amount of money entered.

Building The Circuit (Note that schematic is NOT chip-centric)




+5
+5
10K
18
19
P7
1
P6
13
P5
12
10K
Vss
MAX7219
14
A
16
B
20
C
23
D
21
E
15
F
17
G
22
DP
DIGIT 4
7
6
DIGIT 3
11
DIGIT 2
2
DIGIT 1
4
9
Vss
Vss

```
' =============================================================================
'
'   File..... Ex29 - Change Counter.BS2
'   Purpose... Controlling 7-segment displays with MAX7219
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program is a coin counter -- it will count pennies, nickels, dimes and
' quarters using pushbutton inputs.  The "bank" is displayed on four 7-segment
' LED displays that are controlled with a MAX7219.


' -----------------------------------------------------------------------------
' Revision History
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

DataPin         CON     7                       ' data pin (MAX7219.1)
Clock           CON     6                       ' clock pin (MAX7219.13)
Load            CON     5                       ' load pin (MAX7219.12)
Coins           VAR     InL                     ' coin count inputs


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Decode          CON     $09                     ' bcd decode register
Brite           CON     $0A                     ' intensity register
Scan            CON     $0B                     ' scan limit register
ShutDn          CON     $0C                     ' shutdown register (1 = on)
Test            CON     $0F                     ' display test mode
```

```
DecPnt          CON     %10000000
Blank           CON     %1111                   ' blank a digit

Yes             CON     1
No              CON     0


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

money           VAR     Word                    ' current money count
deposit         VAR     Byte                    ' coins deposited
penny           VAR     deposit.Bit0            ' bit values of deposit
nickel          VAR     deposit.Bit1
dime            VAR     deposit.Bit2
quarter         VAR     deposit.Bit3
dollar          VAR     deposit.Bit4
digit           VAR     Nib                     ' display digit
d7219           VAR     Byte                    ' data for MAX7219
index           VAR     Nib                     ' loop counter
idxOdd          VAR     index.Bit0              ' is index odd? (1 = yes)


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

' Segments               .abcdefg
'                        --------
Full            DATA    %01000111                       ' F
                DATA    %00111110                       ' U
                DATA    %00001110                       ' L
                DATA    %00001110                       ' L


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %11100000                          ' data, clock and load as outs
                                            '   coins as inputs

  FOR index = 0 TO 7
    LOOKUP index, [Scan, 3, Brite, 5, Decode, $0F, ShutDn, 1], d7219
    SHIFTOUT DataPin, Clock, MSBFirst, [d7219]
```

```
    IF (idxOdd = No) THEN No_Load
    PULSOUT Load, 5                             ' load parameter

No_Load:
  NEXT

  GOSUB Show_The_Money


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  GOSUB Get_Coins
  IF (deposit = 0) THEN Main                    ' wait for coins

  money = money + (penny * 1)                   ' add coins
  money = money + (nickel * 5)
  money = money + (dime * 10)
  money = money + (quarter * 25)
  money = money + (dollar * 100)

  GOSUB Show_The_Money                          ' update the display
  PAUSE 100
  GOTO Main


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Get_Coins:
  deposit = %00011111                           ' enable all coin inputs
  FOR index = 1 TO 10
    deposit = deposit & ~Coins                  ' test inputs
    PAUSE 5                                     ' delay between tests
  NEXT
  RETURN


Show_The_Money:
  IF (money >= 9999) THEN Show_Full
  FOR index = 4 TO 1
    d7219 = Blank
    IF ((index = 4) AND (money < 1000)) THEN Put_Digit
    d7219 = money DIG (index - 1)
```

```
    IF (index <> 3) THEN Put_Digit
    d7219 = d7219 | DecPnt                      ' decimal point on DIGIT 3

Put_Digit:
    SHIFTOUT DataPin, Clock, MSBFirst, [index, d7219]
    PULSOUT Load, 5
  NEXT
  RETURN


Show_Full:
  ' turn BCD decoding off
  SHIFTOUT DataPin, Clock, MSBFirst, [Decode, 0]
  PULSOUT Load, 5
  FOR index = 4 TO 1
    READ (4 - index + Full), d7219              ' read and send letter
    SHIFTOUT DataPin, Clock, MSBFirst, [index, d7219]
    PULSOUT Load, 5
  NEXT

  END
```

Behind The Scenes

Multiplexing multiple seven-segment displays requires a lot of effort that consumes most of the computational resources of the BASIC Stamp. Enter the MAXIM MAX7219 LED display driver. Using just three of the BASIC Stamp's I/O lines, the MAX7219 can be used to control up to eight, seven-segment displays or 64 discrete LEDs (four times the number of I/O pins available on the BASIC Stamp).

The MAX7219 connects to the LED displays in a straightforward way; pins SEG A through SEG G and SEG DP connect to segments A through G and the decimal point of all of the common-cathode displays. Pins DIGIT 0 through DIGIT 7 connect to the individual cathodes of each of the displays. If you use less than eight digits, omit the highest digit numbers. For example, this experiment uses four digits, numbered 0 through 3, not 4 through 7.

The MAX7219 has a scan-limit feature than limits display scanning to digits 0 through n, where *n* is the highest digit number. This feature ensures that the chip doesn't waste time and duty cycles (brightness) trying to scan digits that aren't there.

When the MAX7219 is used with seven-segment displays, it can be configured to automatically convert binary-coded decimal (BCD) values into appropriate patterns of segments. This makes the display of decimal numbers simple. The BCD decoding feature can be disabled to display custom patterns. This experiment does both.

From a software standpoint, driving the MAX7219 requires the controller to:

Shift 16 data bits out to the device, MSB first.
Pulse the Load line to transfer the data.

Each 16-bit data package consists of a register address followed by data to store to that register. For example, the 16-bit value $0407 (hex) writes a "7" to the fourth digit of the display. If BCD decoding is turned on for that digit, the numeral "7" will appear on that digit of the display. If decoding is not turned on, three LEDs will light, corresponding to segments G, F, and E.

In this experiment, the MAX7219 is initialized to:

Scan = 3 (Display digits 0 – 3)
Brightness = 5
Decode = $0F (BCD decode digits 0 – 3)
Shutdown = 1 (normal operation)

Initialization of the MAX7219 is handled by a loop. Each pass through the loop reads a register address or data value from a **LOOKUP** table. After each data value is shifted out, the address and data are latched into the MAX7219 by pulsing the Load line.

Most of the work takes place in the subroutine called **Show_The_Money**. When the money count is less than 9999, the value will be displayed on the seven-segment digits, otherwise the display will read "FULL." The routine scans through each digit of money and sends the digit position and value (from the **DIG** operator) to the MAX7219. Since the display shows dollars and cents, the decimal point on the third digit is enabled. When the position and digit have been shifted out, the display is updated by pulsing the Load line. To keep the display neat, the leading zero is blanked when the money value is less than 1000.

When the value of money reaches 9999, the display will change to "FULL." This is accomplished by disabling the BCD decoding of the MAX7219 and sending custom letter patterns to the MAX7219. These patterns are stored in DATA statements.

The main loop of the program is simple: it scans the switch inputs with **`Get_Coins`** and updates the money count for each switch pressed. This particular code is an excellent example of using variable aliases for readability.

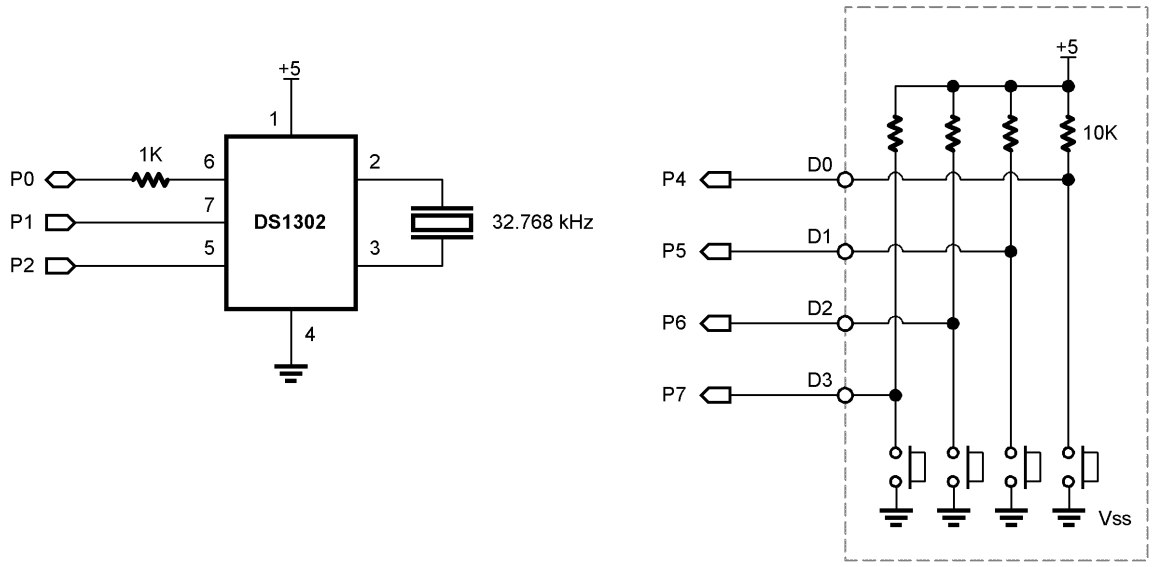
Challenge

Modify the code in experiment 27 to display the input voltage on the seven-segment displays.

Experiment #30: Using a Real-Time Clock

This experiment demonstrates the BASIC Stamp's time-keeping functions through the use of an external real-time clock (RTC). RTC time capability is crucial to time-of-day applications and applications that require the measurement of elapsed time.

Building The Circuit (Note that schematic is NOT chip-centric)




```
' =============================================================================
'
'   File..... Ex30 - DS1302.BS2
'   Purpose... RTC Control
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================
```

```
' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates the control and use of an external real-time clcok
' chip, the DS1302 from Dallas Semiconductor.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

DataIO          CON     0                       ' DS1302.6
Clock           CON     1                       ' DS1302.7
CS1302          CON     2                       ' DS1302.5
BtnsIn          VAR     InB                     ' button input


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

WrSecs          CON     $80                     ' write seconds
RdSecs          CON     $81                     ' read seconds
WrMins          CON     $82                     ' write minutes
RdMins          CON     $83                     ' read minutes
WrHrs           CON     $84                     ' write hours
RdHrs           CON     $85                     ' read hours
CWPr            CON     $8E                     ' write protect register
WPr1            CON     $80                     ' set write protect
WPr0            CON     $00                     ' clear write protect
WrBurst         CON     $BE                     ' write burst of data
RdBurst         CON     $BF                     ' read burst of data
WrRam           CON     $C0                     ' RAM address control
RdRam           CON     $C1

Yes             CON     1
No              CON     0

Hr24            CON     0
Hr12            CON     1

ClockMode       CON     Hr12                    ' use AM/PM mode
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

index           VAR     Byte                    ' loop counter
reg             VAR     Byte                    ' DS1302 address to read/write
ioByte          VAR     Byte                    ' data to/from DS1302

secs            VAR     Byte                    ' seconds
secs01          VAR     secs.LowNib
secs10          VAR     secs.HighNib
mins            VAR     Byte                    ' minutes
mins01          VAR     mins.LowNib
mins10          VAR     mins.HighNib
hrs             VAR     Byte                    ' hours
hrs01           VAR     hrs.LowNib
hrs10           VAR     hrs.HighNib
day             VAR     Byte                    ' day

ampm            VAR     hrs.Bit5                ' 0 = AM, 1 = PM
tMode           VAR     hrs.Bit7                ' 0 = 24, 1 = 12

rawTime         VAR     Word                    ' raw storage of time values
work            VAR     Byte                    ' work variable for display output
oldSecs         VAR     Byte                    ' previous seconds value
apChar          VAR     Byte                    ' "A" or "P"

btns            VAR     Nib                     ' button inputs
btnMin          VAR     btns.Bit0               ' update minutes
btnHrs          VAR     btns.Bit1               ' update hours
btnDay          VAR     btns.Bit2               ' update day
btnBack         VAR     btns.Bit3               ' go backward


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

Su              DATA    "Sunday", 0
Mo              DATA    "Monday", 0
Tu              DATA    "Tuesday", 0
We              DATA    "Wednesday", 0
Th              DATA    "Thursday", 0
Fr              DATA    "Friday", 0
Sa              DATA    "Saturday", 0
```

```
' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirL = %00000111                              ' switches are ins, others outs

  reg = CWPr                                    ' clear write protect register
  ioByte = WPr0
  GOSUB RTC_Out

  oldSecs = $99                                 ' set the display flag
  hrs = $06                                     ' preset time to 6:00 AM
  GOSUB Set_Time


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main1:
  GOSUB Get_Time                                ' read the DS1302
  IF (secs = oldSecs) THEN Check_Buttons        ' time for update?

Main2:
  GOSUB Show_Time                               ' yes, show it
  oldSecs = secs                                ' mark it

Check_Buttons:
  GOSUB Get_Buttons
  IF (btns = 0) THEN Do_Some_Task               ' let Stamp do other work
  IF (btnBack = Yes) THEN Go_Back               ' back button pressed?

Go_Forward:
  rawTime = rawTime + btnMin                    ' add one minute
  rawTime = rawTime + (btnHrs * 60)             ' add one hour
  day = (day + btnDay) // 7                     ' next day
  GOTO Update_Clock

Go_Back:
  IF (btns <= %1000) THEN Do_Some_Task          ' no update button pressed
  rawTime = rawTime + (btnMin * 1439)           ' subtract one minute
  rawTime = rawTime + (btnHrs * 1380)           ' subtract one hour
  day = (day + (btnDay * 6)) // 7               ' previous day

Update_Clock:                                   ' send updated value to DS1302
  rawTime = rawTime // 1440                     ' clean-up time mods
```

```
  GOSUB Set_Raw_Time                          ' set the clock with rawTime
  GOTO Main2

Do_Some_Task:                                 ' work when not setting clock

  ' other code here

  GOTO Main1

' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Show_Time:
  DEBUG Home
  LOOKUP day,[Su,Mo,Tu,We,Th,Fr,Sa],work      ' get address of day string

Get_Day_Char:
  READ work, ioByte                           ' grab a character
  IF (ioByte = 0) THEN Check_Clock_Mode       ' if 0, string is complete
  DEBUG ioByte                                ' print the character
  work = work + 1                             ' point to next
  GOTO Get_Day_Char

Check_Clock_Mode:
  DEBUG "     ", CR                           ' clear day name debris
  IF (ClockMode = Hr24) THEN Show24

Show12:
  DEBUG DEC2 12 - (24 - (hrs10 * 10 + hrs01) // 12)
  DEBUG ":", HEX2 mins, ":", HEX2 secs
  apChar = "A"                                ' assume AM
  IF (hrs < $12) THEN Show_AMPM               ' check time
  apChar = "P"                                ' hrs was >= $12

Show_AMPM:
  DEBUG " ", apChar, "M"                      ' print AM or PM
  GOTO Show_Time_Done

Show24:
  DEBUG HEX2 hrs, ":", HEX2 mins, ":", HEX2 secs

Show_Time_Done:
  RETURN


Get_Buttons:
```

```
  btns = %1111                                  ' enable all button inputs
  FOR index = 1 TO 10
    btns = btns & ~BtnsIn                       ' test inputs
    PAUSE 5                                           ' delay between tests
  NEXT
  PAUSE 200                                     ' slow held button(s)
  RETURN


RTC_Out:                                        ' send ioByte to reg in DS1302
  HIGH CS1302
  SHIFTOUT DataIO, Clock, LSBFirst, [reg, ioByte]
  LOW CS1302
  RETURN


RTC_In:                                         ' read ioByte from reg in DS1302
  HIGH CS1302
  SHIFTOUT DataIO, Clock, LSBFirst, [reg]
  SHIFTIN DataIO, Clock, LSBPre, [ioByte]
  LOW CS1302
  RETURN


Set_Raw_Time:                                   ' convert rawTime to BCD
  hrs10 = rawTime / 600
  hrs01 = (rawTime // 600) / 60
  mins10 = (rawTime // 60) / 10
  mins01 = rawTime // 10


Set_Time:                                       ' write data with burst mode
  HIGH CS1302
  SHIFTOUT DataIO, Clock, LSBFirst, [WrBurst]
  SHIFTOUT DataIO, Clock, LSBFirst, [secs, mins, hrs, 0, 0, day, 0, 0]
  LOW CS1302
  RETURN


Get_Time:                                       ' read data with burst mode
  HIGH CS1302
  SHIFTOUT DataIO, Clock, LSBFirst, [RdBurst]
  SHIFTIN DataIO, Clock, LSBPre, [secs, mins, hrs, day, day, day]
  LOW CS1302
  rawTime = ((hrs10 & %11) * 600) + (hrs01 * 60)
  rawTime = rawTime + (mins10 * 10) + mins01
  RETURN
```

Behind The Scenes

While it is possible to implement rudimentary timekeeping functions in code with **PAUSE**, problems arise when BASIC Stamp needs to handle other activities. This is especially true when an application needs to handle time, day and date. The cleanest solution is an external real-time clock. In this experiment, we'll use the Dallas Semiconductor DS1302. Like the DS1620, the DS1302 requires only three lines to communicate with the BASIC Stamp. Since these two devices are compatible with each other, the clock and data lines to can be shared giving the BASIC Stamp real-time clock and temperature measurement using only four I/O lines.

Once programmed the DS1302 runs by itself and accurately keeps track of seconds, minutes, hours (with an AM/PM indicator, if running in 12-hour mode), date of month, month, day of week and year with leap year compensation valid up to the year 2100. As a bonus, the DS1302 contains 31 bytes of RAM that we can use as we please. And for projects that use main's power, the DS1302 also contains a trickle-charging circuit that can charge a back-up battery.

The DS1302 is a register-based device, that is, each element of the time and date is stored in its own register (memory address). For convenience, two modes of reading and writing are available: register and burst. With register access, individual elements can be written or read. With burst access, all of the registers can be set at once and any number (starting with seconds) can be read back.

In order to keep our interface with the DS1302 simple, this experiment uses it in the 24-hour mode. In this mode, we don't have to fuss with the DS1302 AM/PM indicator bit. For a 12-hour display, we'll deduce AM/PM mathematically. In the code, time is handled as a single, word-sized variable (**`rawTime`**) that represents the number of minutes past midnight. This will make calculating durations and comparing alarm times with the current time very straightforward.

Another compelling reason to use a raw time format is that the DS1302 stores its registers in BCD (binary coded decimal). BCD is a method of storing a value between zero and 99 in a byte-sized variable. The ones digit occupies the lower nibble, the tens digit the upper. Neither nibble of a BCD byte is allowed to have a value greater than nine. Thankfully, the BASIC Stamp allows nibble-sized variables and, more importantly, it allows variables to be aliased.

This experiment demonstrates the DS1302 basics by setting the clock, then polling it for updates. Conversion to and from the DS1320 BCD register format is handled by the subroutines that set and retrieve information in burst mode.

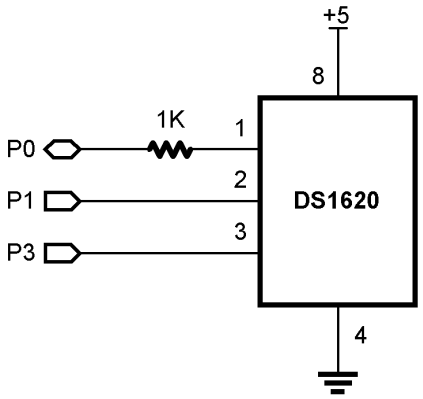
Four pushbuttons are used to set the day, hours and minutes of the clock. Normally, the buttons cause each element to increment. By holding the fourth button, each element will roll back. When no

button is pressed, the program passes to a routine called **Do_Some_Task**. This is where you would put additional code (reading a DS1620, for example).

Program output is sent to a **DEBUG** window. The **Show_Time** subroutine handles printing the day and time in the format specified by **ClockMode**.

Challenge (Advanced)

Add a DS1620 using the connections shown below. Write a program that tracks current, minimum and maximum temperature and will display (use **DEBUG**) the time and date on which the minimum and maximum temperature was measured.



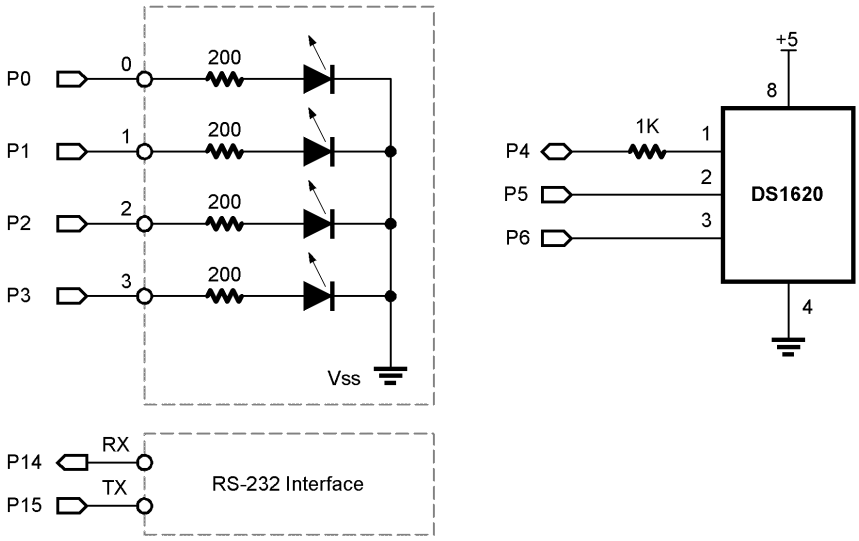
Experiment #31: Serial Communications

This experiment demonstrates the BASIC Stamp's ability to communicate with other computers through any of its I/O pins. It also demonstrates the ability to store information in the BASIC Stamp's EEPROM space.

New PBASIC elements/commands to know:

- `SERIN`
- `SEROUT`
- `WAIT (SERIN modifier)`
- `HEX (SERIN/SEROUT modifier)`
- `BIN (SERIN/SEROUT modifier)`
- `WRITE`

Building The Circuit (Note that schematic is NOT chip-centric)



```
' =============================================================================
'
'   File..... Ex31 - PollStamp.BS2
'   Purpose... Serial Communications
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------

' This program demonstrates serial communications through Stamp I/O pins.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------

LEDs            VAR     OutA                    ' LED outputs
DQ              CON     4                       ' DS1620.1 (through 1K resistor)
Clock           CON     5                       ' DS1620.2
Reset           CON     6                       ' DS1620.3

RxD             CON     14                      ' serial input - to INEX RxD
TxD             CON     15                      ' serial output - to INEX TxD


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

Baud96          CON     84                      ' 9600-8-N-1, BS2/BS2e
' Baud96        CON     240                     ' BS2sx/BS2p

CMenu           CON     $FF                     ' show command menu
CID             CON     $F0                     ' get string ID
CSet            CON     $F1                     ' set string ID
CTmp            CON     $A0                     ' get DS1620 - display raw count
CTmpC           CON     $A1                     ' get DS1620 - display in C
CTmpF           CON     $A2                     ' get DS1620 - display in F
```

```
CStat           CON     $B0                     ' get digital output status
CLEDs           CON     $B1                     ' set LED outputs

RTmp            CON     $AA                     ' read temperature
WTHi            CON     $01                     ' write TH (high temp register)
WTLo            CON     $02                     ' write TL (low temp register)
RTHi            CON     $A1                     ' read TH
RTLo            CON     $A2                     ' read TL
StartC          CON     $EE                     ' start conversion
StopC           CON     $22                     ' stop conversion
WrCfg           CON     $0C                     ' write configuration register
RdCfg           CON     $AC                     ' read configuration register


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

cmd             VAR     Byte                    ' command from PC/terminal
eeAddr          VAR     Byte                    ' EE address pointer
eeData          VAR     Byte                    ' EE data
param           VAR     Word                    ' parameter from PC
char            VAR     param.LowByte           ' character from terminal
tmpIn           VAR     Word                    ' raw data from DS1620
halfBit         VAR     tmpIn.Bit0              ' 0.5 degree C indicator
sign            VAR     tmpIn.Bit8              ' 1 = negative temperature
tempC           VAR     Word                    ' degrees C in tenths
tempF           VAR     Word                    ' degrees F in tenths
potVal          VAR     Word                    ' reading from BSAC pot
buttons         VAR     Nib                     ' input buttons


' -----------------------------------------------------------------------------
' EEPROM Data
' -----------------------------------------------------------------------------

ID              DATA "StampWorks 1.2", CR       ' CR-terminated string


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
  DirA = %1111                                  ' LED pins are outputs

  HIGH Reset                                    ' alert the DS1620
```

```
  SHIFTOUT DQ, Clock, LSBFirst, [WrCfg, %10]    ' use with CPU; free-run
  LOW Reset
  PAUSE 10
  HIGH Reset
  SHIFTOUT DQ, Clock, LSBFirst, [StartC]        ' start conversions
  LOW Reset

  GOTO Show_Menu


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  cmd = 0
  SERIN RxD, Baud96, [WAIT ("?"), HEX cmd]

  ' check for menu request
  IF (cmd = CMenu) THEN Show_Menu

  ' convert command for branching
  LOOKDOWN cmd, [CID, CSet, CTmp, CTmpC, CTmpF, CStat, CLEDs], cmd

  ' branch to requested routine
  BRANCH cmd, [Show_ID, Set_ID, Show_Temp, Show_Temp_C, Show_Temp_F]
  cmd = cmd - 5
  BRANCH cmd, [Show_Status, Set_LEDs]

BadCommand:
  SEROUT TxD, Baud96, ["Invalid Command: ", HEX2 cmd, CR]
  GOTO Main


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

Show_Menu:
  SEROUT TxD, Baud96, [CLS]
  SEROUT TxD, Baud96, ["========================", CR]
  SEROUT TxD, Baud96, ["   StampWorks Monitor   ", CR]
  SEROUT TxD, Baud96, ["========================", CR]
  SEROUT TxD, Baud96, ["?FF - Show Menu", CR]
  SEROUT TxD, Baud96, ["?F0 - Display ID", CR]
  SEROUT TxD, Baud96, ["?F1 - Set ID", CR]
  SEROUT TxD, Baud96, ["?A0 - DS1620 (Raw count)", CR]
```

```
  SEROUT TxD, Baud96, ["?A1 - Temperature (C)", CR]
  SEROUT TxD, Baud96, ["?A2 - Temperature (F)", CR]
  SEROUT TxD, Baud96, ["?B0 - Display LED Status", CR]
  SEROUT TxD, Baud96, ["?B1 - Set LEDs", CR, CR]
  SEROUT TxD, Baud96, ["Please enter a command.", CR, CR]
  GOTO Main


Show_ID:
  SEROUT TxD, Baud96, ["ID="]                   ' label output
  eeAddr = ID                                   ' point to first character of ID

Get_EE:
  READ eeAddr, eeData                           ' read a character from EEPROM
  SEROUT TxD, Baud96, [eeData]                  ' print the character
  eeAddr = eeAddr + 1                           ' point to next character
  IF (eeData <> CR) THEN Get_EE                 ' if not CR, read another
  GOTO Main


Set_ID:
  eeAddr = ID                                   ' point to ID location

Get_Char:
  SERIN RxD, Baud96, [char]                     ' get character from PC
  WRITE eeAddr, char                            ' write character to EEPROM
  eeAddr = eeAddr + 1                           ' point to next location
  IF (char <> CR) THEN Get_Char                 ' if not CR, wait for another
  GOTO Show_ID                                  ' confirm new ID


Show_Temp:
  GOSUB Get_Temp
  ' send raw temp to PC
  SEROUT TxD, Baud96, ["DS1620=", DEC tmpIn, CR]
  GOTO Main


Show_Temp_C:
  GOSUB Get_Temp
  IF (sign = 0) THEN No_Neg_C
  tmpIn = 0                                     ' only temps above freezing

No_Neg_C:
  ' convert raw count to 10ths C
  tempC = tmpIn * 5
  SEROUT TxD, Baud96, ["TempC=", DEC (tempC/10), ".", DEC (tempC // 10), CR]
```

```
  GOTO Main


Show_Temp_F:
  GOSUB Get_Temp
  IF (sign = 0) THEN No_Neg_F
  tmpIn = 0

No_Neg_F:
  tempF = (tmpIn * 9) + 320                     ' convert raw count to 10ths F
  SEROUT TxD, Baud96, ["TempF=", DEC (tempF / 10), ".", DEC (tempF // 10), CR]
  GOTO Main


Show_Status:
  SEROUT TxD, Baud96, ["Status=", BIN4 LEDs, CR]
  GOTO Main


Set_LEDs:
  ' wait for output bits
  ' - as binary string
  '
  SERIN RxD, Baud96, [BIN param]
  LEDs = param.LowNib                           ' set the outputs
  GOTO Show_Status                              ' confirm new outputs


Get_Temp:
  HIGH Reset                                    ' alert the DS1620
  SHIFTOUT DQ, Clock, LSBFirst, [RTmp]          ' read temperature
  SHIFTIN DQ, Clock, LSBPre, [tmpIn\9]          ' get the temperature
  LOW Reset
  RETURN
```

Behind The Scenes

Without asynchronous serial communications the world would not be what it is today. Businesses would be hard pressed to exchange information with each other. There would be no ATMs for checking our bank accounts and withdrawing funds. There would be no Internet.

Previous experiments have used synchronous serial communications. In that scheme, two lines are required: clock and data. The benefit is the automatic synchronization of sender and receiver. The downside is that it requires at least two wires to send a message.

Asynchronous serial communications requires only a single wire to transmit a message. What is necessary to allow this scheme is that both the sender and receiver must agree on the communications speed before the transmission can be received. Some "smart" systems can detect the communications speed (baud rate), the BASIC Stamp cannot.

In this experiment we'll use **SEROUT** to send information to a terminal program and **SERIN** to take data in. The input will usually be a command and sometimes the command will be accompanied with new data.

After initializing the LED outputs and the DS1620, the program enters the main loop and waits for input from the terminal program. First, **SERIN** waits for the "**?**" character to arrive, ignoring everything else until that happens. The question mark, then, is what signifies the start of a query. Once a question mark arrives, the **HEX** modifier causes the BASIC Stamp to look for valid hex characters (0 - 9, A - F). The arrival of any non-hex character (usually a carriage return [Enter] when using a terminal) tells the BASIC Stamp to stop accepting input (to the variable called **param** in our case) and continue on.

What actually has happened is that the BASIC Stamp has used the **SERIN** function to do a text-to-numeric conversion. Now that a command is available, the program uses **LOOKDOWN** to decode the command and **BRANCH** to jump to the requested subroutine if the command was valid. If the command isn't valid, a message and the offending input is displayed.

The BASIC Stamp responds to a request sending a text string using **SEROUT** set to 9600 baud (so we can use the BASIC Stamp's DEBUG terminal as the host). Each of the response strings consists of a label, the equal sign, the value of that particular parameter and finally, a carriage return. When using a terminal program, the output is easily readable. Something like this:

```
ID=Parallax BS2
```

The carriage return at the end of the output gives us a new line when using a terminal program and serves as an "end of input" when we process the input with our own program (similar to BASIC Stamp Plot Lite). The equal sign can be used as a delimiter when another computer program communicates with the BASIC Stamp. We'll use it to distinguish the label from its value.

Most of the queries are requests for information. Two of them, however, can modify information that is stored in the BASIC Stamp.

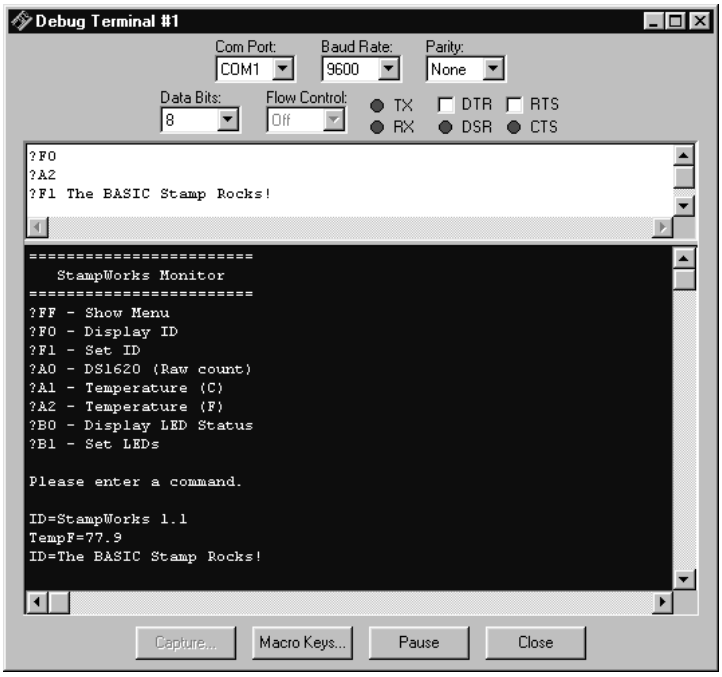
The first one is "**?F1**" which will allow us to write a string value to the BASIC Stamp's EEPROM (in a location called ID). When F1 is received as a command value, the program jumps to the subroutine called **Set_ID**. On entry to **Set_ID**, the EE pointer called **addr** is initialized, then the BASIC Stamp waits for a character to arrive. Notice that no modifier is used here. Since terminal programs and the BASIC Stamp represent characters using ASCII codes, we don't have to do anything special. When a character does arrive, **WRITE** is used to put the character into EEPROM and the address pointer is incremented. If the last character was a carriage return (13), the program outputs the new string (using the code at **Show_ID**), otherwise it loops back and waits for another character.

The second modifying query is "**?B1**" which allows us to set the status of four LEDs. Take a look at the subroutine called **Set_LEDs**. This time, the **BIN** modifier of **SERIN** is used so that we can easily define individual bits we wish to control. By using the **BIN** modifier, our input will be a string of ones and zeros (any other character will terminate the binary input). In this program, a "1" will cause the LED to turn on and a "0" will cause the LED to turn off. Here's an example of using the B1 query.

```
?B1 0011 <CR>
```

The figure below shows an actual on-line session using the BASIC Stamp's DEBUG terminal. To run the experiment, follow these steps:

1. Remove components from previous experiment.
2. Enter and download the program
3. Remove power from StampWorks lab board and build the circuit
4. Move the programming cable to the RS-232 Interfacing port
5. Open a DEBUG window by clicking on the DEBUG icon
6. Set the StampWorks lab board power switch to on.



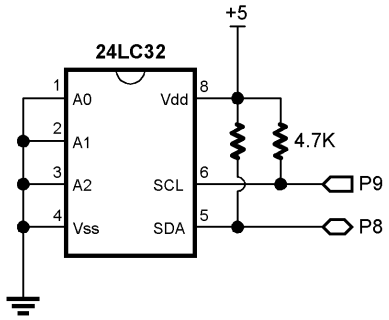
Challenge (for PC programmers)

Write a PC program that interfaces with this experiment.

Experiment #32: I²C Communications

This experiment demonstrates the BASIC Stamp's ability to communicate with other devices through the use of the popular Philips I²C protocol. The experiment uses this protocol to write and read data to a serial EEPROM and the low-level I²C routines can be used to communicate with any I²C device.

Building The Circuit




```
' =============================================================================
'
'   File..... Ex32 - 24LC32.BS2
'   Purpose... 24LC32 control via I2C
'   Author.... Parallax
'   E-mail.... stamptech@parallaxinc.com
'   Started...
'   Updated... 01 MAY 2002
'
'   {$STAMP BS2}
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------
'
' This program demonstrates essential I2C routines and communication with the
' Microchip 24LC32 serial EEPROM.
'
' The connections for this program conform to the BS2p I2CIN and I2COUT
```

```
' commands.  Use this program for the BS2, BS2e or BS2sx.  There is a separate
' program for the BS2p.


' -----------------------------------------------------------------------------
' I/O Definitions
' -----------------------------------------------------------------------------
'
SDA             CON     8                       ' I2C serial data line
SCL             CON     9                       ' I2C serial clock line


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------

DevType         CON     %1010 << 4              ' device type
DevAddr         CON     %000 << 1               ' address = %000 -> %111
Wr2432          CON     DevType | DevAddr | 0   ' write to 24LC32
Rd2432          CON     DevType | DevAddr | 1   ' read from 24LC32

ACK             CON     0                       ' acknowledge bit
NAK             CON     1                       ' no ack bit

CrsrXY          CON     2                       ' DEBUG Position Control


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

i2cSDA          VAR     Nib                     ' I2C serial data pin
i2cData         VAR     Byte                    ' data to/from device
i2cWork         VAR     Byte                    ' work byte for TX routine
i2cAck          VAR     Bit                     ' ACK bit from device

eeAddr          VAR     Word                    ' address: 0 - 4095
test            VAR     Nib
outVal          VAR     Byte                    ' output to EEPROM
inVal           VAR     Byte                    ' input from EEPROM


' -----------------------------------------------------------------------------
' Initialization
' -----------------------------------------------------------------------------

Initialize:
```

```
  PAUSE 250                                     ' let DEBUG open
  DEBUG CLS, "24LC32 Demo", CR, CR              ' setup output screen
  DEBUG "Address... ", CR
  DEBUG "Output.... ", CR
  DEBUG "Input..... ", CR

  i2cSDA = SDA                                  ' define SDA pin


' -----------------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------------

Main:
  FOR eeAddr = 0 TO 4095                        ' test all locations
    DEBUG CrsrXY, 11, 2, DEC eeAddr, "   "
    FOR test = 0 TO 3                           ' use four patterns
      LOOKUP test, [$FF, $AA, $55, $00], outVal
      DEBUG CrsrXY, 11, 3, IHEX2 outVal
      i2cData = outVal
      GOSUB Write_Byte
      PAUSE 10
      GOSUB Read_Byte
      inVal = i2cData
      DEBUG CrsrXY, 11, 4, IHEX2 inVal, "  "
      IF (inVal <> outVal) THEN Bad_Addr
      DEBUG "Pass "
      GOTO Next_Addr

Bad_Addr:
      DEBUG "Fail "

Next_Addr:
      PAUSE 50
    NEXT
  NEXT

  DEBUG CR, CR, "Done!"
  END


' -----------------------------------------------------------------------------
' Subroutines
' -----------------------------------------------------------------------------

' Byte to be written is passed in i2cData
' -- address passed in eeAddr
```

```
Write_Byte:
  GOSUB I2C_Start                              ' send Start
  i2cWork = Wr2432                             ' send write command
  GOSUB I2C_TX_Byte
  IF (i2cAck = NAK) THEN Write_Byte            ' wait until not busy
  i2cWork = eeAddr / 256                       ' send word address (1)
  GOSUB I2C_TX_Byte
  i2cWork = eeAddr // 256                      ' send word address (0)
  GOSUB I2C_TX_Byte
  i2cWork = i2cData                            ' send data
  GOSUB I2C_TX_Byte
  GOSUB I2C_Stop
  RETURN


' Byte read is returned in i2cData
' -- address passed in eeAddr

Read_Byte:
  GOSUB I2C_Start                              ' send Start
  i2cWork = Wr2432                             ' send write command
  GOSUB I2C_TX_Byte
  IF (i2cAck = NAK) THEN Write_Byte            ' wait until not busy
  i2cWork = eeAddr / 256                       ' send word address (1)
  GOSUB I2C_TX_Byte
  i2cWork = eeAddr // 256                      ' send word address (0)
  GOSUB I2C_TX_Byte
  GOSUB I2C_Start
  i2cWork = Rd2432                             ' send read command
  GOSUB I2C_TX_Byte
  GOSUB I2C_RX_Byte_Nak
  GOSUB I2C_Stop
  i2cData = i2cWork
  RETURN


' -----------------------------------------------------------------------------
' Low Level I2C Subroutines
' -----------------------------------------------------------------------------

' --- Start ---

I2C_Start:                                     ' I2C start bit sequence
  INPUT i2cSDA
  INPUT SCL
  LOW i2cSDA                                   ' SDA -> low while SCL high
```

```
Clock_Hold:
  IF (Ins.LowBit(SCL) = 0) THEN Clock_Hold      ' device ready?
  RETURN


' --- Transmit ---

I2C_TX_Byte:
  SHIFTOUT i2cSDA, SCL, MSBFIRST, [i2cWork\8]   ' send byte to device
  SHIFTIN i2cSDA, SCL, MSBPRE, [i2cAck\1]       ' get acknowledge bit
  RETURN


' --- Receive ---

I2C_RX_Byte_Nak:
  i2cAck = NAK                                  ' no ACK = high
  GOTO I2C_RX

I2C_RX_Byte:
  i2cAck = ACK                                  ' ACK = low

I2C_RX:
  SHIFTIN i2cSDA, SCL, MSBPRE, [i2cWork\8]      ' get byte from device
  SHIFTOUT i2cSDA, SCL, LSBFIRST, [i2cAck\1]    ' send ack or nak
  RETURN


' --- Stop ---

I2C_Stop:                                       ' I2C stop bit sequence
  LOW i2cSDA
  INPUT SCL
  INPUT i2cSDA                                  ' SDA --> high while SCL high
  RETURN
```

Behind the Scenes

The I²C-bus is a two-wire, synchronous bus that uses a Master-Slave relationship between components. The Master initiates communication with the Slave and is responsible for generating the clock signal. If requested to do so, the Slave can send data back to the Master. This means the data pin (SDA) is bi-directional and the clock pin (SCL) is [usually] controlled only by the Master.

Experiment #32: I²C Communications

The transfer of data between the Master and Slave works like this:

Master sending data
- Master initiates transfer
- Master addresses Slave
- Master sends data to Slave
- Master terminates transfer

Master receiving data
- Master initiates transfer
- Master addresses Slave
- Master receives data from Slave
- Master terminates transfer

The I^2C specification actually allows for multiple Masters to exist on a common bus and provides a method for arbitrating between them. That's a bit beyond the scope of what we need to do so we're going to keep things simple. In our setup, the BS2 (or BS2e or BS2sx) will be the Master and anything connected to it will be a Slave.

You'll notice in I^2C schematics that the SDA and SCL lines are pulled up to Vdd (usually through 4.7K). The specification calls for device bus pins to be open drain. To put a high on either line, the associated bus pin is made an input (floats) and the pull-up takes the line to Vdd. To make a line low, the bus pin pulls it to Vss (ground).

This scheme is designed to protect devices on the bus from a short to ground. Since neither line is driven high, there is no danger. We're going to cheat a bit. Instead of writing code to pull a line low or release it (certainly possible – I did it), we're going to use **SHIFTOUT** and **SHIFTIN** to move data back and forth. Using **SHIFTOUT** and **SHIFTIN** is faster and saves precious code space. If you're concerned about a bus short damaging the Stamp's SDA or SCL pins during **SHIFTOUT** and **SHIFTIN**, you can protect each of them with a 220 ohm resistor. I've been careful with my wiring and code and haven't found this necessary.

Low Level I²C Code

At its lowest level, the I²C Master needs to do four things:

- Generate a Start condition
- Transmit 8-bit data to the Slave
- Receive 8-bit data from Slave – with or without Acknowledge
- Generate Stop condition

A Start condition is defined as a HIGH to LOW transition on the SDA line while the SCL line is HIGH. All transmissions begin with a Start condition. A Stop condition is defined as a LOW to HIGH transition of the SDA line while the clock line is HIGH. A Stop condition terminates a transfer and can be used to abort it as well.

There is a brief period when the Slave can take control of the SCL line. If a Slave is not ready to transmit or receive data, it can hold the SCL line low after the Start condition. The Master can monitor this to wait for the Slave to be ready. At the speed of the BS2, monitoring the clock line usually isn't necessary but I've built the clock-hold test into the I2C_Start subroutine just to be safe.

Data is transferred eight bits at a time, sending the MSB first. After each byte, the I²C specification calls for the receiving device to acknowledge the transmission by bringing the bus low for the ninth clock. The exception to this is when the Master is the receiver and is receiving the final byte from the Slave. In this case, there is no Acknowledge bit sent from Master to Slave.

Sending and receiving data from a specific slave always requires a Start condition, sending the Slave address and finally, the Stop condition. What happens between the Slave address and the Stop are dependent on the device and what we're doing.

What you'll need to do is get the data sheet for the I²C device you want to connect to. I have found, without exception, that data sheets for I²C-compatible parts have very clear protocol definitions – usually in graphic form – that makes implementing our low-level I²C routines very simple.

The experiment uses the low-level I²C routines to implement the **Write_Byte** and **Read_Byte** routines. The sequence for these routines was lifted right from the 24LC32 data sheet. Notice that each routine begins with an I²C Start condition and is terminated with the Stop condition. The code in between sends the device command/type code, the address to deal with and then actually deals with (writes or reads) the data. While this takes a few lines of code, it is actually very straightforward.

Experiment #32: I²C Communications

Most I²C routines follow a very similar structure; varying only in the internal address and for a few devices, the way the device code is transmitted (there are a few devices that carry an address setting in the device code byte).

Challenge

From the hundreds of I²C devices available, pick one that will be most useful for your projects and write the high-level code necessary to communicate with it.

Striking Out on Your Own

Congratulations, you're a BASIC Stamp programmer! So what's next? Well, that's up to you. Many new programmers get stuck when it comes to developing their own projects. Don't worry, this is natural – and there are ways out of being stuck. The following tips will help you succeed in moving your ideas to reality.

Plan Your Work, Work Your Plan

You've heard it a million times: plan, plan, and plan. Nothing gets a programmer into more trouble than bad or inadequate planning. This is particularly true with the BASIC Stamp as resources are so limited. Most of the programs we've fixed were "broken" due to bad planning and poor formatting which lead to errors.

Talk It Out

Talk yourself through the program. Don't just think it through, *talk it through*. Talk to yourself–out loud–as if you were explaining the operation of the program to a fellow programmer. Often, just hearing our own voice is what makes the difference. Better yet, talk it out as if the person you're talking to *isn't* a programmer. This will force you to explain details. Many times we take things for granted when we're talking to ourselves or others of similar ability.

Write It Out

Design the details of your program on a white (dry erase) board before you sit down at your computer. And use a lot of colors. You'll find working through a design visually will offer new insights, and the use of this medium allows you to write code snippets within your functional diagrams.

Design With "Sticky Notes"

Get out a pad of small "sticky notes". Write module names or concise code fragments on individual notes and then stick them up on the wall. Now stand back and take a look. Then move them around. Add notes, take some away; just do what feels right to you. This exercise works particularly well with groups. How do you know when you're done? When the sticky notes stop moving! It's a good idea to record the final outcome before starting your editor. Another tip: this trick works even better when combined with trick #2. You can draw lines between and around notes to indicate program flow or logical groupings. If it's not quite right, just erase the lines or move some notes. Try this trick; it really does work.

Going Beyond The Box

By now, your appetite for BASIC Stamp projects has probably grown well beyond what you ever expected. So where do you turn now? Don't worry, there are many BASIC Stamp and related resources available, both in print and on the Internet. Here's a list to get you started:

Books & Magazines

- *Microcontroller Application Cookbook* By Matt Gilliland
- *Microcontroller Projects with BASIC Stamps* By Al Williams
- *Programming and Customizing the BASIC Stamp Computer* By Scott Edwards
- *BASIC Stamp* By Claus Kühnel and Klaus Zahnert
- *Getting Started In Electronics* By Forrest Mims
- *Engineer's Notebook* By Forrest Mims
- *Nuts & Volts* Magazine "Stamp Applications" column

Internet Sites

www.parallaxinc.com	Parallax main site
www.stampsinclass.com	Parallax educational site
www.al-williams.com/awce/index.htm	Al Williams web site
www.seetron.com	Scott Edwards Electronics web site
www.hth.com/losa	List of Stamp Applications – great idea source
www.emesystems.com/BS2index.htm	Tracy Allen's Stamp resources – very technical

Appendix A:
BASIC Stamp II Manual Version 2.0c

Pages 198-344 of the BASIC Stamp Manual are included in this appendix. The entire manual (and future updates) is available for purchase or download from www.parallaxinc.com.