BASIC Stamp® Programming Manual

Version 1.8

PARALLAX® INC

This manual is valid with the following software and firmware versions:

BASIC Stamp I:
STAMP.EXE software version 2.0
Firmware version 1.4

BASIC Stamp II:
STAMP2.EXE software version 1.1
Firmware version 1.0

Newer versions will usually work, but older versions may not. New software can be obtained for free on our BBS and Internet web and ftp site. New firmware, however, must usually be purchased in the form of a new BASIC Stamp. If you have any questions about what you may need, please contact Parallax.

Thank you for purchasing a BASIC Stamp product. We've been making BASIC Stamp computers for years, and most customers find them useful and fun. Of course, we hope your experience with BASIC Stamps will be useful and fun, as well. If you have any questions or need technical assistance, please don't hesitate to contact Parallax or the distributor from which you purchased your BASIC Stamps.

This manual is divided into two sections. The first section deals with the BASIC Stamp I, and the second section deals with the BASIC Stamp II. The BASIC Stamp I has been around for some time, and therefore has more data in the way of application notes. If you have prior experience with BASIC Stamp I, you should consult Appendix C, for details on converting to the Basic Stamp II.

PBASIC Language: the BASIC Stamps are programmed in a simple version of the BASIC language, called *PBASIC*. We developed PBASIC to be easy to understand, yet well-suited for the many control and monitoring applications that BASIC Stamps are used in. The PBASIC language includes familiar instructions, such as GOTO, FOR...NEXT, and IF...THEN, as well as specialized instructions, such as SERIN, PWM, BUTTON, COUNT, and DTMFOUT.

Hardware: the BASIC Stamps discussed in this manual are the "BS1-IC" and "BS2-IC." Both represent the latest versions of the BASIC Stamp I and BASIC Stamp II. Both include a small circuit board with a PBASIC interpreter chip, EEPROM, 5-volt regulator, reset circuit, and resonator. These five components form a complete computer in a very small space. The modular design of the BS1-IC and BS2-IC makes them perfect for use in breadboards and printed circuit boards.

Each of the BASIC Stamp modules has a corresponding "carrier board." The carrier boards provide 9-volt battery clips, connectors for programming, and a small prototyping area. Although they are optional, we recommend that you purchase at least one carrier board as a means of easily programming your BASIC Stamps.

Important Information

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 return the product with a description of the problem and a copy of your invoice (if you do not have your invoice, please include your name and telephone number). We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax (for instance, if you ship your product via overnight express, we will do the same).

This warranty does not apply if the product has been modified or damaged by accident, abuse, or misuse.

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 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.

BBS/Internet Access

We maintain BBS and Internet systems for your convenience. 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
Ftp:	ftp.parallaxinc.com (same file selection as BBS)
Web:	http://www.parallaxinc.com
BBS:	(916) 624-7101 (300-14400 baud, 8 data bits, 1 stop bit, no parity)

Internet BASIC Stamp Discussion List

We maintain an email discussion list for people interested in BASIC Stamps. 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 issues.

To subscribe to the Stamp list, send email to ***majordomo@parallaxinc.com*** and write ***subscribe stamps*** in the body of the message.

BASIC Stamp I:

BASIC Stamp II:

Contents

System Requirements

To program the BASIC Stamp I, you'll need the following computer system:

- IBM PC or compatible computer
- 3.5-inch disk drive
- Parallel port
- 128K of RAM
- MS-DOS 2.0 or greater

If you have the BASIC Stamp I carrier board, you can use a 9-volt battery as a convenient means to power the BASIC Stamp. You can also use a 5-15 volt power supply (5-40 volts on the BS1-IC rev. b), but you should be careful to connect the supply to the appropriate part of the BASIC Stamp. A 5-volt supply should be connected directly to the +5V pin, but a higher voltage should be connected to the PWR pin.

Connecting a high voltage supply (greater than 6 volts) to the 5-volt pin can permanently damage the BASIC Stamp.

Packing List

If you purchased the BASIC Stamp Programming Package, you should have received the following items:

- BASIC Stamp manual (this manual)
- BASIC Stamp I programming cable (parallel port DB25-to-3 pin)
- BASIC Stamp II programming cable (serial port DB9-to-DB9)
- 3.5-inch diskette

If you purchased the BASIC Stamp II Starter Kit, you should have received the following items:

- BASIC Stamp Manual (this manual)
- BASIC Stamp II programming cable (serial port DB9-to-DB9)
- 3.5-inch diskette

If any items are missing, please let us know.

Connecting to the PC

To program a BASIC Stamp I, you'll need to connect it to your PC and then run the editor/downloader software. In this section of the manual, it's assumed that your BASIC Stamp is a BS1-IC, and that you have the corresponding carrier board.

To connect the BASIC Stamp to your PC, follow these steps:

1) Plug the BS1-IC onto the carrier board. The BS1-IC plugs into a 14-pin SIP socket, located near the battery clips on the carrier. When plugged onto the carrier board, the components on the BS1-IC should face the battery clips.

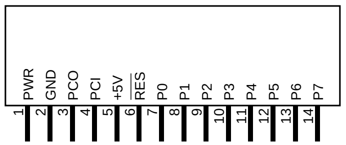
2) In the BASIC Stamp Programming Package, you received a cable to connect the BASIC Stamp to your PC. The cable has two ends, one with a DB25 connector and the other with a 3-pin connector. Plug the DB25 end into an available **parallel** port on your PC.

3) Plug the remaining end of the cable onto the 3-pin header on the carrier board. On the board and the cable, you'll notice a double-arrow marking; the markings on the cable and board should match up.

4) Supply power to the carrier board, either by connecting a 9-volt battery or by providing an external power source.

With the BASIC Stamp connected and powered, run the editor/downloader software as described later in this manual.

BS1-IC



Shown at 125% of actual size

PWR **Unregulated power in:** accepts 6-15 VDC (6-40 VDC on BS1-IC rev. b), which is then regulated to 5 volts. May be left unconnected if 5 volts is applied to the +5V pin.

GND **System ground:** connects to PC parallel port pin 25 (GND) for programming.

PCO **PC Out:** connects to PC parallel port pin 11 (BUSY) for programming.

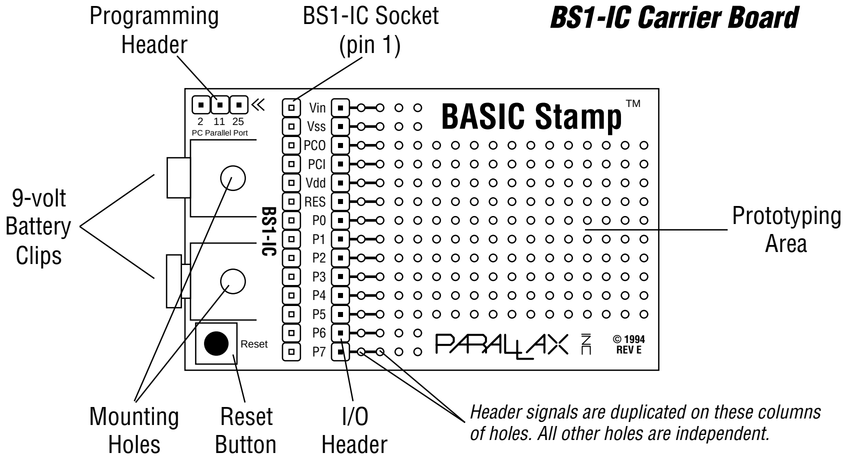
PCI **PC In:** connects to PC parallel port pin 2 (D0) for programming.

+5V **5-volt input/output:** if an unregulated voltage is applied to the **PWR** pin, then this pin will output 5 volts. If no voltage is applied to the **PWR** pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.

RES **Reset input/output:** goes low when power supply is less than 4 volts, causing the BS1-IC to reset. Can be driven low to force a reset. Do not drive high.

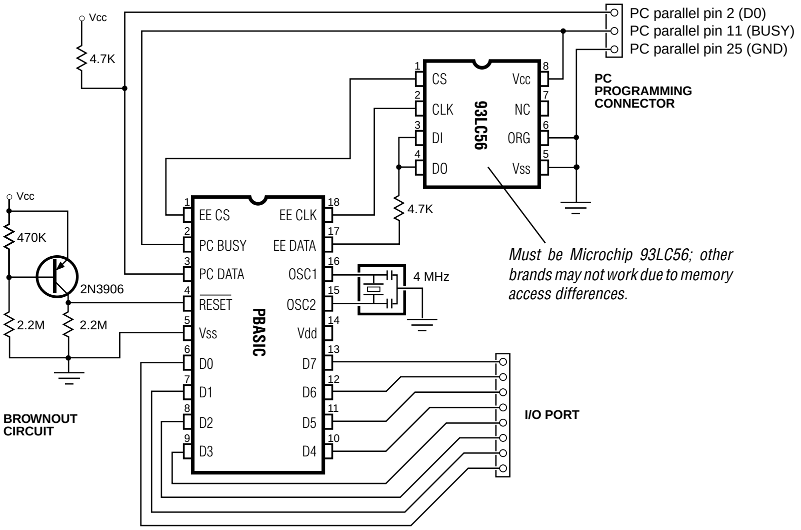
P0-P7 **General-purpose I/O pins:** each can sink 25 mA and source 20 mA. However, the total of all pins should not exceed 50 mA (sink) and 40 mA (source).

BS1-IC Carrier Board



Header signals are duplicated on these columns of holes. All other holes are independent.

General Stamp Schematic*:



Must be Microchip 93LC56; other brands may not work due to memory access differences.

* The BS1-IC has a slightly different schematic (it uses a different reset circuit, and it includes a 5-volt regulator). However, this schematic serves as an example of how simple the BASIC Stamp circuit is to implement.

Current Limits of the On-Board Regulator

In some cases, you may want to know how much current the BS1-IC can handle with its on-board regulator. At higher supply voltages, the regulator can handle less current. The BS1-IC itself takes 1-2 mA, so any current "left over" can be used to drive external circuits. The table below shows the approximate current limits at various voltages:

Power Supply (volts)	Total Current (mA)
5-9	50
12	40
25	10
40	2-3

We recommend a supply voltage on the low end (5-15 VDC). However, the BS1-IC will run at higher voltages, as shown.

The BASIC Stamp I has 16 bytes of RAM devoted to I/O and the storage of variables. The first two bytes are used for I/O (1 for actual pins, 1 for direction control), leaving 14 bytes for data. This arrangement of variable space is shown below:

Word Name	Byte Names	Bit Names	Special Notes
Port	Pins Dirs	Pin0-Pin7 Dir0-Dir7	*I/O pins; bit addressable.* *I/O pin direction control; bit addressable.*
W0	B0 B1	Bit0-Bit7 Bit8-Bit15	*Bit addressable.* *Bit addressable.*
W1	B2 B3		
W2	B4 B5		
W3	B6 B7		
W4	B8 B9		
W5	B10 B11		
W6	B12 B13		*Used by GOSUB instruction.* *Used by GOSUB instruction.*

The PBASIC language allows a fair amount of flexibility in naming variables and I/O pins. Depending upon your needs, you can use the variable space and I/O pins as bytes *(Pins, Dirs, B0-B13)* or as 16-bit words *(Port, W0-W6)*. Additionally, the I/O pins and the first two data bytes can be used as individual bits *(Pin0-Pin7, Dir0-Dir7, Bit0-Bit15)*. In many cases, a single bit may be all you need, such as when storing a status flag.

Port is a 16-bit word, which is composed of two bytes, **Pins** and **Dirs:**

> **Pins** (byte) and **Pin0-Pin7** (corresponding bits) are the I/O port pins. When these variables are read, the I/O pins are read directly. When these variables are written to, the corresponding RAM is written to, which is then transferred to the I/O pins before each instruction.
>
> **Dirs** (byte) and **Dir0-Dir7** (corresponding bits) are the I/O port direction bits. A "0" in one of these bits causes the corresponding I/O pin to be an input; a "1" causes the pin to be an output. This byte of data is transferred to the I/O port's direction register before each instruction.
>
> ***When you write your PBASIC programs, you'll use the symbols described above to read and write the BASIC Stamp's 8 I/O pins.***
>
> Normally, you'll start your program by defining which pins are inputs and which are outputs. For instance, "dirs = %00001111" sets bits 0-3 as outputs and bits 4-7 as inputs (right to left).
>
> After defining which pins are inputs and outputs, you can read and write the pins. The instruction "pins = %11000000" sets bits 6-7 high. For reading pins, the instruction "b2 = pins" reads all 8 pins into the byte variable b2.
>
> Pins can be addressed on an individual basis, which may be easier. For reading a single pin, the instruction "Bit0 = Pin7" reads the state of I/O pin 7 and stores the reading in bit variable Bit0. The instruction "if pin3 = 1 then start" reads I/O pin 3 and then jumps to start (a location) if the pin was high (1).

The BASIC Stamp's editor software recognizes the variable names shown on the previous page. If you'd like to use different names, you can start your program with instructions to define new names:

```
symbol switch = pin0              'Define label "switch" for I/O pin 0
symbol flag = bit0                'Define label "flag" for bit variable bit0
symbol count = b2                 'Define label "count" for byte variable b2
```

Can I expand the BASIC Stamp's program memory?:

No; the PBASIC interpreter only addresses 8 bits of program space, which results in the 256-byte limitation. Using a larger EEPROM, such as the Microchip 93LC66, won't make any difference.

What voltage range can I use to power the BASIC Stamp:

We encourage people to use a 9-volt battery to power the BASIC Stamp, especially if they have the carrier board. The battery is simple and can power the BASIC Stamp for days, even weeks if sleep mode is used.

However, if you want to use an external power supply, you can use anything that supplies 5-15 volts DC (5-40 VDC on BS1-IC rev. b) at a minimum of 2 mA (not including I/O current needs).

If you have a 5-volt supply, connect it to the BASIC Stamp's +5V pin. This will route power directly to the BASIC Stamp circuit, bypassing the voltage regulator.

If you have a 6-15 (6-40 VDC on BS1-IC rev. b) volt supply, connect it to the BASIC Stamp's PWR pin. This will route power through the on-board 5-volt regulator.

Can I use the Stamp to power external circuits?:

Yes; if you need to supply 5 volts, connect your circuit to the BASIC Stamp's **+5V** pin. If you need the unregulated input voltage, connect your circuit to the **PWR** pin.

How long can the BASIC Stamp run on a 9-volt battery?:

This depends on what you're doing with the BASIC Stamp. If your program never uses sleep mode and has several LED's connected to I/O lines, then the BASIC Stamp may only run for several hours. If, however, sleep mode is used and I/O current draw is minimal, then the BASIC Stamp can run for weeks.

What are the sink and source capabilities of the BASIC Stamp's I/O lines?:

The I/O pins can each sink 25 mA and source 20 mA. However, the total sink and source for all 8 I/O lines should not exceed 50 mA (sink) and 40 mA (source).

Does the BASIC Stamp support floating point math?:

No; the BASIC Stamp only works with integer math, which means that no fractions are allowed. Expressions must be given as integers, and any results are given as integers. For instance, if you gave the BASIC Stamp an instruction to divide 5 by 2, it would return a result of 2, not 2.5; the remainder (.5) is simply lost.

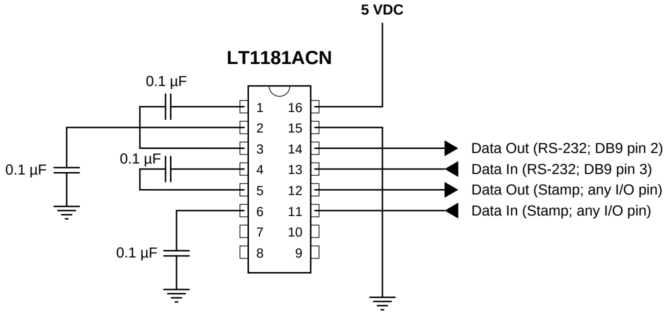
How does the BASIC Stamp evaluate mathematical expressions?:

Mathematical expressions are evaluated strictly left to right. This is important, since you may get different results than you expect. For instance, under normal rules, the expression 2 + 3 x 4 would be solved as 2 + (3 x 4), since multiplication takes priority over addition. The result would be 14. However, since the BASIC Stamp solves expressions from left to right, it would be solved as (2 + 3) x 4, for a result of 20.

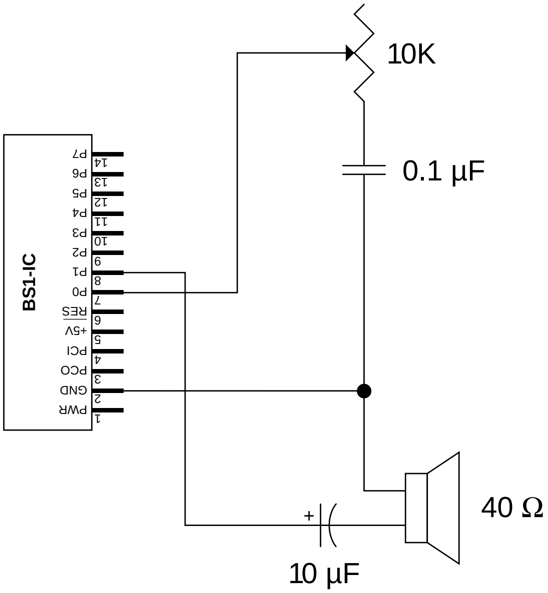
When writing your programs, please remember that the left-to-right evaluation of expressions may affect the results.

What do I need to make the BASIC Stamp support RS-232 voltages?

The BASIC Stamp's I/O pins operate at TTL voltages (0-5 volts), so the SERIN and SEROUT instructions operate at these voltages. This is fine for most applications, such as BASIC Stamps communicating with other BASIC Stamps. However, some PCs may not accept TTL voltages, especially when the PC is receiving data. If you need real RS-232 voltages, you can use the circuit shown below. The LT1181ACN is available from various distributors, including Digi-Key (call 800-344-4539).


5 VDC
LT1181ACN
0.1 µF
0.1 µF
0.1 µF
0.1 µF
1
2
3
4
5
6
7
8
16
15
14
13
12
11
10
9
Data Out (RS-232; DB9 pin 2)
Data In (RS-232; DB9 pin 3)
Data Out (Stamp; any I/O pin)
Data In (Stamp; any I/O pin)

This page shows a simple application using a BS1-IC. The purpose of the application is to read the value of the potentiometer and then generate a corresponding tone on the speaker. As the potentiometer value changes, so does the tone. For interesting variations, the potentiometer could easily be changed to a thermistor or photocell.



```
loop:
        pot 0,100,b2                    'Read potentiometer on pin 0 and
                                        'store result in variable b2.

        b2=b2/2                         'Divide result so highest value
                                        'will be 128.

        sound 1,(b2,10)                 'Generate a tone using speaker
                                        'on pin 1. Frequency is set by
                                        'value in b2. Duration of tone
                                        'is set to 10.

        goto loop                       'Repeat the process.
```

Starting the Editor

With the BASIC Stamp connected and powered, run the editor software by typing the following command from the DOS prompt:

```
STAMP
```

Assuming you're in the proper directory, the BASIC Stamp software will start running after several seconds. The editor screen is dark blue, with one line across the top that names various functions.

Program Formatting

There are few restrictions on how programs are entered. However, you should know the rules for entering constants, labels, and comments, as described in the following pages:

- ***Constants:*** constant values can be declared in four ways: decimal, hex, binary, and ASCII.

  Hex numbers are preceded with a dollar sign ($), binary numbers are preceded with a percent sign (%), and ASCII values are enclosed in double quotes ("). If no special punctuation is used, then the editor will assume the value is decimal. Following are some examples:

```
100                 'Decimal
$64                 'Hex
%01100100           'Binary
"A"                 'ASCII "A" (65)
"Hello"             'ASCII "H", "e", "l", "l", "o"
```

  Most of your programs will probably use decimal values, since this is most common in BASIC. However, hex and binary can be useful. For instance, to define pins 0-3 as outputs and pins 4-7 as inputs, you could use any of the following, but the binary example is the most readable:

```
dirs = 15               'Decimal
dirs = $0F              'Hex
dirs = %00001111        'Binary
```

- ***Address Labels:*** the editor uses labels to refer to addresses (locations) within your program. This is different from some versions of BASIC, which use line numbers.

  Generally speaking, label names can be any combination of letters, numbers, and underscores (_), but the first character of the name must not be a number. Also, label names cannot use reserved words, such as instruction names (serin, toggle, goto, etc.) and variable names (port, w2, b13, etc.)

  When first used, label names must end with a colon (:). When called elsewhere in the program, labels are named without the colon. The following example illustrates how to use a label to refer to an address:

```
loop:   toggle 0                        'Toggle pin 0

        for b0 = 1 to 10
        toggle 1                        'Toggle pin 1 ten times
        next

        goto loop                       'Repeat the process
```

- ***Value Labels:*** along with program addresses, you can use labels to refer to variables and constants. Value labels share the same syntax rules as address labels, but value labels don't end with a colon (:), and they must be defined using the "symbol" directive. The following example shows several value labels:

```
symbol start = 1                        'Define two constant
symbol end = 10                         'labels

symbol count = b0                       'Define a label for b0

loop:   for count = start to end
        toggle 1                        'Toggle pin 1 ten times
        next
```

- ***Comments:*** comments can be added to your program to make it more readable.

  Comments begin with an apostrophe (') and continue to the end of the line. You can also designate a comment using the standard REM statement found in many versions of BASIC...

```
symbol relay = 3                      'Make label for I/O pin 3
symbol length = w2                    'Make label for w2

          dirs = %11111111            'Make all pins outputs
          pins = %00000000            'Make all pins low

REM this is the main loop

main:     length = length + 10        'Increase length by 10
          gosub sub                   'Call pulse out routine
          goto main                   'Loop back

sub:      pulsout relay,length : toggle 0 : return
```

- ***General Format:***

  The editor is not case sensitive, except when processing strings (such as “hello”).

  Multiple instructions and labels can be combined on the same line by separating them with colons (:).

  The following example shows the same program as separate lines and as a single-line...

  Multiple-line version:

```
          dirs = 255                  'Make all pins outputs
          for b2 = 0 to 100           'Count from 0 to 100
          pins = b2                   'Make pins = count (b2)
          next                        'Continue counting til 100
```

  Single-line version:

```
          dirs = 255 : for b2 = 0 to 100 : pins = b2 : next
```

- ***Mathematical Operators:*** the following operators may be used in mathematical expressions:

+	add
-	subtract
*	multiply (returns low word of result)
**	multiply (returns high word of result)
/	divide (returns quotient)
//	divide (returns remainder)
min	keep variable greater than or equal to value
max	keep variable less than or equal to value
&	logical AND
|	logical OR
^	logical XOR
&/	logical AND NOT
|/	logical OR NOT
^/	logical XOR NOT

Some examples:

```
count = count + 1               'Increment count
timer = timer * 2               'Multiply timer by 2
b2 = b2 / 8                     'Divide b2 by 8
w3 = w3 & 255                   'Isolate lower byte of w3
b0 = b0 + 1 max 99              'Increment b0, but don't
                                'allow b0 to exceed 99
b3 = b3 - 1 min 10              'Decrement b3, but don't
                                'allow b3 to drop below 10
```

Entering & Editing Programs

As covered in the previous pages, there are some rules to remember about the use of constants, labels, and comments. However, for the most part, you can format your programs as you see fit.

We've tried to make the editor as intuitive as possible: to move up, press the *up arrow*; to highlight one character to the right, press *shift-right arrow*; etc.

Most functions of the editor are easy to use. Using single keystrokes, you can perform the following common functions:

- Load, save, and run programs.
- Move the cursor in increments of one character, one word, one line, one screen, or to the beginning or end of a file.
- Highlight text in blocks of one character, one word, one line, one screen, or to the beginning or end of a file.
- Cut, copy, and paste highlighted text.
- Search for and/or replace text.

Editor Function Keys

The following list shows the keys that are used to perform various functions:

Alt-R	Run program in BASIC Stamp *(download the program on the screen, then run it)*
Alt-L	Load program from disk
Alt-S	Save program on disk
Alt-Q	Quit editor and return to DOS
Enter	Enter information and move down one line
Tab	Same as Enter
Left arrow	Move left one character
Right arrow	Move right one character

Up arrow	Move up one line
Down arrow	Move down one line
Ctrl-Left	Move left to next word
Ctrl-Right	Move right to next word
Home	Move to beginning of line
End	Move to end of line
Page Up	Move up one screen
Page Down	Move down one screen
Ctrl-Page Up	Move to beginning of file
Ctrl-Page Down	Move to end of file
Shift-Left	Highlight one character to the left
Shift-Right	Highlight one character to the right
Shift-Up	Highlight one line up
Shift-Down	Highlight one line down
Shift-Ctrl-Left	Highlight one word to the left
Shift-Ctrl-Right	Highlight one word to the right
Shift-Home	Highlight to beginning of line
Shift-End	Highlight to end of line
Shift-Page Up	Highlight one screen up
Shift-Page Down	Highlight one screen down
Shift-Ctrl-Page Up	Highlight to beginning of file
Shift-Ctrl-Page Down	Highlight to end of file
Shift-Insert	Highlight word at cursor
ESC	Cancel highlighted text
Backspace	Delete one character to the left
Delete	Delete character at cursor
Shift-Backspace	Delete from left character to beginning of line
Shift-Delete	Delete to end of line
Ctrl-Backspace	Delete line
Alt-X	Cut marked text and place in clipboard
Alt-C	Copy marked text to clipboard
Alt-V	Paste (insert) clipboard text at cursor
Alt-F	Find string (establish search information)
Alt-N	Find next occurrence of string
Alt-P	Calibrate potentiometer scale *(see POT instruction for more information)*

Running Your Program

To run the program shown on the screen, press Alt-R. The editor software will check all available parallel ports, searching for a BASIC Stamp. If it finds one, it will download and run your program. Note that any program already in the BASIC Stamp will be overwritten. If the editor is unable to locate a BASIC Stamp, it will display an error.

Assuming that you have a BASIC Stamp properly connected to your PC, the editor will display a bargraph, which shows how the download of your program is progressing. Typical downloads take only several seconds, so the graph will fill quickly.

As the graph fills, you'll notice that some of the graph fills with white blocks, while the remainder fills with red blocks. These colors represent how much of the BASIC Stamp's EEPROM space is used by the program. White represents available space, and red represents space occupied by the program.

When the download is complete, your program will automatically start running in the BASIC Stamp. If you used the debug directive in your program, it will display its data when it's encountered in the program.

To remove the download graph from the screen, press any key.

Loading a Program from Disk

To load a PBASIC program from disk, press Alt-L. A small box will appear, prompting you for a filename. If you entered the filename correctly, the program will be loaded into the editor. Otherwise, an error message will be displayed.

If you decide not to load a program, press ESC to resume editing.

Saving a Program on Disk

To save a PBASIC program on disk, press Alt-S. A small box will appear, prompting you for a filename. After the filename is entered, the editor will save your program.

Using Cut, Copy, and Paste

Like most word processors, the editor can easily cut, copy, and paste text. If you need to make major changes to your program, or your program has many repetitive routines, these functions can save a lot of time.

The function of the cut, copy, and paste routines is to cut or copy highlighted text to the *clipboard* (the clipboard is an area of memory set aside by the editor). Text in the clipboard can later be pasted (inserted) anywhere in your program. Both *cut* and *paste* copy text to the clipboard, but *cut* also removes the text from its current location.

Please note that *cutting* text is different from *deleting* it. While both functions remove text from its current location, only *cut* saves the text to the clipboard – *delete* removes it entirely.

As an example of cut and paste, let's cut a section of text and then paste it elsewhere. The following steps will guide you through the process:

- First, you need to **highlight some text**. For this example, let's highlight everything from the cursor to the end of the line. To do this, press Shift-End (everything from the cursor to the end of the line should become highlighted).
- Second, with the line highlighted, **press Alt-X** (cut). The text should disappear.
- Third, move the cursor to another location – anywhere is fine. Then, **press Alt-V** (paste). The text should appear where the cursor was, pushing any following text down as necessary.

The first step could be replaced with copy (Alt-C), instead of cut (Alt-X). The only difference would be that the text would appear in its original location, as well as the pasted location.

Using Search & Replace

The editor has a function that allows you to search for and/or replace text. In many instances, this function can be very useful. For example, you may decide to change a variable name throughout your program. Doing so manually would take a lot of time, but with search and replace, it takes just seconds.

To set the search criteria, **press Alt-F** (find). A small box will appear in the center of the screen, requesting a search string and an optional replacement string. To perform the search, follow these steps:

- **Enter the search string**. If you want to search for a string that contains the *Tab* or *Return* keys, you can do so by typing *Ctrl-Tab* or *Ctrl-Return*; "•" will appear for each tab, "↓" for each return.
- **Enter the replacement string, if necessary**. If you enter a replacement string, it will be copied to the *clipboard* (the clipboard is an area of memory set aside by the editor). During the search, you will have the option to replace individual occurrences of the search string with the replacement string.

  If you only want to search (without the option to replace), just press *Enter* for the replacement string.
- The editor will remove the search criteria box and highlight the **first occurrence** of the search string.

  To replace the highlighted string with the replacement string, press Alt-V (paste).

  To find the next occurrence of the search string, **press Alt-N**.

BRANCHING

IF...THEN *Compare and conditionally branch.*

BRANCH *Branch to address specified by offset.*

GOTO *Branch to address.*

GOSUB *Branch to subroutine at address. Up to 16 GOSUB's are allowed.*

RETURN *Return from subroutine.*

LOOPING

FOR...NEXT *Establish a FOR...NEXT loop.*

NUMERICS

{LET} *Perform variable manipulation, such as A=5, B=A+2, etc. Possible operations are add, subtract, multiply, divide, max. limit, min. limit, and logical operations AND, OR, XOR, AND NOT, OR NOT, and XOR NOT.*

LOOKUP *Lookup data specified by offset and store in variable. This instruction provides a means to make a lookup table.*

LOOKDOWN *Find target's match number (0-N) and store in variable.*

RANDOM *Generate a pseudo-random number.*

DIGITAL I/O

OUTPUT *Make pin an output.*

LOW *Make pin output low.*

HIGH *Make pin output high.*

TOGGLE *Make pin an output and toggle state.*

PULSOUT *Output a timed pulse by inverting a pin for some time.*

INPUT *Make pin an input*

PULSIN *Measure an input pulse.*

REVERSE *If pin is an output, make it an input. If pin is an input, make it an output.*

BUTTON *Debounce button, perform auto-repeat, and branch to address if button is in target state.*

SERIAL I/O

SERIN *Serial input with optional qualifiers and variables for storage of received data. If qualifiers are given, then the instruction will wait until they are received before filling variables or continuing to the next instruction. Baud rates of 300, 600, 1200, and 2400 are possible. Data received must be with no parity, 8 data bits, and 1 stop bit.*

SEROUT *Send data serially. Data is sent at 300, 600, 1200, or 2400 baud, with no parity, 8 data bits, and 1 stop bit.*

ANALOG I/O

PWM *Output PWM, then return pin to input. Used to output analog voltages (0-5V) using a capacitor and resistor.*

POT *Read a 5-50K potentiometer and scale result.*

SOUND

SOUND *Play notes. Note 0 is silence, notes 1-127 are ascending tones, and notes 128-255 are white noises.*

EEPROM ACCESS

EEPROM *Store data in EEPROM before downloading BASIC program.*

READ *Read EEPROM byte into variable.*

WRITE *Write byte into EEPROM.*

TIME

PAUSE *Pause execution for 0–65536 milliseconds.*

POWER CONTROL

NAP *Nap for a short period. Power consumption is reduced.*

SLEEP *Sleep for 1-65535 seconds. Power consumption is reduced to approximately 20 µA.*

END *Sleep until the power cycles or the PC connects. Power consumption is reduced to approximately 20 µA.*

PROGRAM DEBUGGING

DEBUG *Send variables to PC for viewing.*

BRANCH *offset,(address0,address1,...addressN)*

Go to the address specified by offset (if in range).

- **Offset** is a variable/constant that specifies the address to branch to (0–N).
- **Addresses** are labels that specify where to branch.

Branch works like the ON x GOTO command found in other BASICs. It's useful when you want to write something like this:

```
if b2 = 0 then case_0  ' b2=0: go to label "case_0"
if b2 = 1 then case_1  ' b2=1: go to label "case_1"
if b2 = 2 then case_2  ' b2=2: go to label "case_2"
```

You can use Branch to organize this into a single statement:

```
BRANCH b2,(case_0,case_1,case_2)
```

This works exactly the same as the previous IF...THEN example. If the value isn't in range (in this case if b2 is greater than 2), Branch does nothing. The program continues with the next instruction after Branch.

Branch can be teamed with the Lookdown instruction to create a simplified SELECT CASE statement. See Lookdown for an example.

Sample Program:

```
Get_code:
    serin 0,N2400,("code"),b2          ' Get serial input.
                                       ' Wait for the string "code",
                                       ' then put next value into b2.
    BRANCH b2,(case_0,case_1,case_2)   ' If b2=0 then case_0
                                       ' If b2=1 then case_1
                                       ' If b2=2 then case_2
goto Get_code                          ' If b2>2 then Get_code.

    case_0:          ...               ' Destinations of the
    case_1:          ...               ' Branch instruction.
    case_2:          ...
```

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

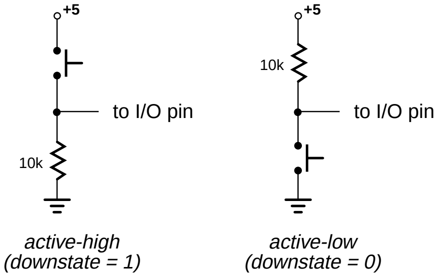
Debounce button input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high (see the diagram on the next page).

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Downstate** is a variable/constant (0 or 1) that specifies which logical state is read when the button is pressed.
- **Delay** is a variable/constant (0–255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the Button routine. Delay has two special settings: 0 and 255. If set to 0, the routine returns the button state with no debounce or auto-repeat. If set to 255, the routine performs debounce, but no auto-repeat.
- **Rate** is a variable/constant (0–255) that specifies the auto-repeat rate. The rate is expressed in cycles of the Button routine.
- **Bytevariable** is the workspace for Button. It must be cleared to 0 before being used by Button for the first time.
- **Targetstate** is a variable/constant (0 or 1) that specifies which state the button should be in for a branch to occur (0=not pressed, 1=pressed).
- **Address** is a label that specifies where to branch if the button is in the target state.

When you press a button or flip a switch, the contacts make or break a connection. A burst of electrical noise occurs as the contacts bounce against each other. Button's debounce feature prevents this noise from being interpreted as more than one switch action.

Button also lets the Stamp react to a button press the way your PC keyboard does to a key press. When you press a key, a character appears on the screen. If you hold the key down, there's a delay, then a rapid-fire stream of characters appears on the screen. Button's autorepeat function can be set up to work the same way.

Button is designed to be used inside a program loop. Each time through the loop, Button checks the state of the specified pin. When



Example button circuits.

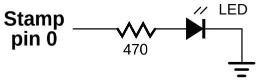
it first matches *downstate,* Button debounces the switch. Then, in accordance with *targetstate,* it either branches to *address* (targetstate=1) or doesn't (targetstate = 0).

If the switch is kept in *downstate,* Button tracks the number of program loops that execute. When this count equals *delay,* Button again triggers the action specifed by *targetstate* and *address*. Hereafter, if the switch remains in *downstate,* Button waits *rate* number of cycles between actions.

The important thing to remember about Button is that it does not stop program execution. In order for its delay and autorepeat functions towork, Button must execute from within a loop.

Sample Program:

```
' This program toggles (inverts) the state of an LED on pin 0 when the
' active-low switch on pin 7 is pressed. When the switch is held down, Button
' waits, then rapidly autorepeats the Toggle instruction, making the LED
' flash rapidly. When the switch is not pressed, Button skips the Toggle
' instruction. Note that b2, the workspace variable for Button, is cleared
' before its first use. Don't clear it within the loop.

        let b2 = 0                              ' Button workspace cleared.
Loop:
        BUTTON 7,0,200,100,b2,0,skip            ' Go to skip unless pin7=0.
        Toggle 0                                ' Invert LED.
        ...                                     ' Other instructions.
skip:
        goto Loop                               ' Skip toggle and go to Loop.
```



LED hookup for sample program.

DEBUG *variable{,variable}*

Displays the specified variable (bit, byte, or word) in a window on the screen of a connected PC. Debug works only after a “run” (ALT-R) download has finished.

Debug accepts formatting modifiers as follows:

- No modifiers: prints “variable = value”
- # before variable, as in #b2, prints the decimal value, without the “variable =” text.
- $ before variable, as in $b2, prints hex value.
- % before variable, as in %b2, prints binary value.
- @ before variable, as in @b2, prints the ASCII character corresponding to the value of the variable.
- Text in quotes appears as typed.
- cr (carriage return) causes printing in the Debug window to start a new line.
- cls (clear screen) clears the Debug window.
- commas must separate all variables used with Debug.

Samples:

```
DEBUG b2                          ' Print "b2 = " + value of b2
DEBUG #b2                         ' Print value of b2
DEBUG "reading is ",b2            ' Print "reading is " & value of b2
DEBUG #%b2                        ' Print value of b2 in binary
DEBUG #@b2                        ' Display the ASCII character
                                  ' corresponding to the value in b2.
DEBUG "inputs ",b2,b3,cr          ' Print "inputs" & value of b2 & value
                                  ' of b3 & carriage return.
```

EEPROM *{location},(data,data,...)*

Store values in EEPROM before downloading the BASIC program.

- **Location** is an optional variable/constant (0–255) that specifies the starting location in the EEPROM at which data should be stored. If no location is given, data is written starting at the next available location.
- **Data** are variables/constants (0–255) to be stored sequentially in the EEPROM.

EEPROM is useful for storing values to be used by your program. One application is to store long messages for use by Serout as shown below:

Program Sample 1:

```
' Sends the text "A very long message indeed..." then reads address 255 for
' the last instruction location of the program.
        serout 0,N2400,("A very long message indeed...")
        read 255,b2              ' Get last program location (reflects length of program)
        debug b2                 ' Display it on the screen.
```

Program Sample 2:

```
' Sends the text "A very long message indeed..." then reads address 255 for
' the last instruction location of the program.
        EEPROM 0,("A very long message indeed...")
        for b2 = 0 to 28         ' Send message 1 char at a time.
        read b2,b3               ' Read data at location b2 of
        serout 0,N2400,(b3)      ' EEPROM into b3. Transmit b3.
        next                     ' Send next character.
        read 255,b2              ' Get last program location (reflects length of program)
        debug b2                 ' Display it on the screen.
```

The first program sample shows an endpoint of 197, meaning that it uses 58 bytes of program memory to send the 29-byte message. Sample 2 has an endpoint of 232 (23 bytes of program memory used). When you add 29 bytes for the storage of the message, sample 2 is 6 bytes more efficient. The savings are greater when the messages are used at several points in a program.

END

Enter sleep mode indefinitely. The Stamp wakes up when the power cycles or the PC connects. Power consumption is reduced to about 20 µA, assuming no loads are being driven.

If you do leave Stamp pins in an output-high or output-low state driving loads when End executes, two things will happen:

- The loads will continue to draw current from the Stamp's power supply.
- Every 2.3 seconds, current to those loads will be interrupted for a period of approximately 18 milliseconds (ms).

The reason for the output glitch every 2.3 seconds has to do with the design of the PBASIC interpreter chip. It has a free-running clock called the "watchdog timer" that can periodically reset the processor, causing a sleeping Stamp to wake up. Once awake, the Stamp checks its program to determine whether it should remain awake or go back to sleep. After an End instruction, the Stamp has standing orders to go back to sleep.

Unfortunately, the watchdog timer cannot be shut off, so the Stamp actually gets its sleep as a series of 2.3-second naps. At the end of each nap, the watchdog timer resets the PBASIC chip. Among other things, a reset causes all of the chip's pins to go into input mode. It takes approximately 18 ms for the PBASIC firmware to get control, restore the pins to their former state, and put the Stamp back to sleep.

If you use End, Nap, or Sleep in your programs, make sure that your loads can tolerate these periodic power outages. The easy solution is often to connect pull-up or pull-down resistors as appropriate to ensure a continuing supply of current during the reset glitch.

FOR *variable = start TO end {STEP {-} increment}*...NEXT *{variable}*

Establish a For...Next loop. *Variable* is set to the value *start*. Code between the For and Next instructions is then executed. *Variable* is then incremented/decremented by *increment* (if no increment value is given, the variable is incremented by 1). If *variable* has not reached or passed the value *end*, the instructions between For and Next are executed again. If *variable* has reached or passed *end*, then the program continues with the instruction after Next. The loop is executed at least once, no matter what values are given for *start* and *end*.

Your program can have as many For...Next loops as necessary, but they cannot be nested more than eight deep (in other words, your program can't have more than eight loops within loops).

- **Variable** is a bit, byte, or word variable used as an internal counter. *Start* and *end* are limited by the capacity of *variable* (bit variables can count from 0 to 1, byte variables from 0 to 255, and word variables from 0 to 65535).
- **Start** is a variable/constant which specifies the initial value of *variable*.
- **End** is a variable/constant which specifies the ending value of *variable*.
- **Increment** is an optional variable/constant by which the counter increments or decrements (if negative). If no step value is given, the variable increments by 1.
- **Variable** (after Next) is optional. It is used to clarify which of a series of For...Next loops a particular Next refers to.

Program Samples:

Programmers most often use For...Next loops to repeat an action a fixed number of times, like this:

```
FOR b2 = 1 to 10                        ' Repeat 10 times.
  debug b2                              ' Show b2 in debug window.
NEXT                                    ' Again until b2>10.
```

Don't overlook the fact that all of the parameters of a For...Next loop can be variables. Not only can your program establish these values itself, it can also modify them while the loop is running. Here's an example in which the step value increases with each loop:

```
let b3 = 1
FOR b2 = 1 to 100 STEP b3          ' Each loop, add b3 to b2.
debug b2                           ' Show b2 in debug window.
let b3 = b3+2                      ' Increment b3.
NEXT                               ' Again until b2>15.
```

If you run this program, you may notice something familiar about the numbers in the debug window (1,4,9,16,25,36,49...). They are all squares (1^2=1, 2^2=4, 3^2=9, 4^2=16, etc.), but our program used addition, not multiplication, to calculate them. This method of generating a polynomial function is credited to Sir Isaac Newton.

There is a potential bug in the For...Next structure. PBASIC uses 16-bit integer math to increment/decrement the counter variable and compare it to the *end* value. The maximum value a 16-bit variable can hold is 65535. If you add 1 to 65535, you get 0 (the 16-bit register rolls over, much like a car's odometer does when you exceed the maximum mileage it can display).

If you write a For...Next loop whose *step* value is larger than the difference between the *end* value and 65535, this rollover will cause the loop to execute more times than you expect. Try the following:

```
FOR w1 = 0 to 65500 STEP 3000      ' Each loop add 3000 to w1.
  debug w1                         ' Show w1 in debug window.
NEXT                               ' Again until w1>65500.
```

The value of w1 increases by 3000 each trip through the loop. As it approaches the stop value, an interesting thing happens: 57000, 60000, 63000, 464, 3464... It passes the *end* value and keeps going. That's because the result of the calculation 63000 + 3000 exceeds the maximum capacity of a 16-bit number. When the value rolls over to 464, it passes the test "is w1 > 65500?" used by Next to determine when to end the loop.

The same problem can occur when the step value is negative and larger (in absolute value) than the difference between the *end* value and 0.

GOSUB *address*

Store the address of the instruction following Gosub, branch to *address*, and continue execution there. The next Return instruction takes the program back to the stored address, continuing the program on the instruction following the most recent Gosub.

- **Address** is a label that specifies where to branch. Up to 16 GOSUBs are allowed per program.

PBASIC stores data related to Gosubs in variable w6. Make sure that your program does not write to w6 unless all Gosubs have Returned, and don't expect data written to w6 to be intact after a Gosub.

If a series of instructions is used at more than one point in your program, you can turn those instructions into a subroutine. Then, wherever you would have inserted that code, you can simply write Gosub *label* (where *label* is the name of your subroutine).

Sample Program:

```
' In this program, the subroutine test takes a pot measurement, then performs
' a weighted average by adding 1/4 of the current measurement to 3/4 of a
' previous measurement. This has the effect of smoothing out noise.
      for b0 = 1 to 10
      GOSUB test              ' Save this address & go to test.
      serout 1,N2400,(b2)     ' Return here after test.
      next                    ' Again until b0 > 10.
      end                     ' Prevents program from running into test.
test:
      pot 0,100,b2            ' Take a pot reading.
      let b2 = b2/4 + b4      ' Make b2 = (.25*b2)+b4.
      let b4 = b2 * 3 / 4     ' Make b4 = .75*b2.
return                        ' Return to previous address+1 instruction.
```

The Return instruction at the end of *test* sends the program to the instruction immediately following Gosub *test*; in this case Serout.

Make sure that your program cannot wander into a subroutine without Gosub. In the sample, what if End were removed? After the loop , execution would continue in *test*. When it reached Return, the program would jump back into the the For...Next loop at Serout because this was the last return address assigned. The For...Next loop would execute forever.

GOTO *address*

Branch to *address,* at which point execution continues.

- **Address** is a label that specifies where to branch.

Sample Program:

```
abc:
        pulsout 0,100             ' Generate a 1000-µs pulse on pin 0.
        GOTO abc                  ' Repeat forever.
```

HIGH *pin*

Make the specified pin output high. If the pin is programmed as an input, it changes to an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin.

You can think of the High instruction as the equivalent of:

```
output 3            ' Make pin 3 an output.
let pin3 = 1        ' Set pin 3 high.
```

Notice that the Output command accepts the pin number (3), while Let requires the pin's variable name *pin3*. So, in addition to saving oneinstruction, High allows you to make a pin output-high using only its number. When you look at the sample program below, imagine how difficult it would be to write it using Output and Let.

This points out a common bug involving High. Programmers sometimes substitute pin names like *pin3* for the pin number. Remember that those pin names are really bit variables. As bits, they can hold values of 0 or 1. The statement "High pin3" is a valid BASIC instruction, but it means, "Get the state of *pin3*. If *pin3* is 0, make *pin 0* output high. If *pin3* is 1, make *pin 1* output high."

Sample Program:

```
' One at a time, change each of the pins to output and set it high.
    for b2 = 0 to 7         ' Eight pins (0-7).
    HIGH b2                 ' Set pin no. indicated by b2.
    pause 500               ' Wait 1/2 second between pins.
    next                    ' Do the next pin.
```

IF *variable ?? value {AND/OR variable ?? value...}* THEN *address*

Compare variable(s) to value(s) and branch if result is true.

- **??** is one of the following operators: = (equal), <> (not equal), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)
- **Variable** is a variable that is compared to value(s)
- **Value** is a variable or constant for comparison
- **Address** is a label that specifies where to branch if the result of the comparison(s) is true

Unlike those in some other flavors of BASIC, this If...Then statement can only go to an address label. It does not support statements like "IF x > 30 THEN x = 0." To do the same thing neatly in PBASIC requires a little backwards thinking:

```
IF x <= 30 THEN skip              ' If x is less than or equal
let x = 0                         ' to 30, don't make x=0.
skip: ...                         ' Program continues.
```

Unless x > 30, the program skips over the instruction "let x = 0."

PBASIC's If...Then can evaluate two or more comparisons at one time with the conjunctions *And* and *Or*. It works from left to right, and does not accept parentheses to change the order of evaluation. It can be tricky to anticipate the outcome of compound comparisons. We suggest that you set up a test of your logic using debug as shown in the sample program below.

Sample Program:

```
' Evaluates the If...Then statement and displays the result in a debug window.
      let b2 = 7                          ' Assign values.
      let b3 = 16
      IF b3 < b2 OR b2 = 7 THEN True      ' B3 is not less than b2, but
                                          ' b2 is 7: so statement is true.
      debug "statment is false"           ' If statement is false, goto here.
end
True:
      debug "statement is true"           ' If statement is true, goto here.
end
```

INPUT *pin*

Make the specified pin an input. This turns off the pin's output drivers, allowing your program to read whatever state is present on the pin from the outside world.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

There are several ways to set pins to input. When a program begins, all of the Stamp's pins are inputs. Input instructions (Pulsin, Serin) automatically change the specified pin to input and leave it in that state. Writing 0s to particular bits of the variable *dirs* makes the corresponding pins inputs. And then there's the Input instruction.

When a pin is set to input, your program can check its state by reading its value. For example:

```
Hold:      if pin4 = 0 then Hold              ' Stay here until pin4 is 1.
```

The program is reading the state of pin 4 as set by external circuitry. If nothing is connected to pin 4, it could be in either state (1 or 0) and could change states apparently at random.

What happens if your program writes to a pin that is set up as an input? The state is written to the output latch assigned to the pin. Since the output drivers are disconnected when a pin is an input, this has no effect. If the pin is changed to output, the last value written to the latch will appear on the pin. The program below shows how this works.

Sample Program:

```
' To see this program in action, connect a 10k resistor from pin 7 to +5V.
' When the program runs, a debug window will show you the state at pin 7
' (a 1, due to the +5 connection); the effect of writing to an input pin (none);
'  and the result of changing an input pin to output (the latched state appears
' on the pin and may be read by your program). Finally, the program shows
' how changing pin 7 to output writes a 1 to the corresponding bit of dirs.
      INPUT 7                               ' Make pin 7 an input.
      debug "State present at pin 7: ",#pin7,cr,cr
      let pin7 = 0                          ' Write 0 to output latch.
      debug  "After 0 written to input: ",#pin7,cr,cr
      output 7                              ' Make pin 7 an output.
      debug  "After pin 7 changed to output: ",#pin7,cr
      debug "Dirs (binary): ",#%dirs        ' Show contents of dirs.
```

{LET} *variable = {-}value ?? value...*

Assign a value to the variable and/or perform logic operations on the variable. All math and logic is done at the word level (16 bits).

The instruction "Let" is optional. For instance, "A=10" is identical to "Let A=10".

- **??** is one of the following operators:

+	add
–	subtract
*	multiply (returns low word of result)
**	multiply (returns high word of result)
/	divide (returns quotient)
//	divide (returns remainder)
min	keep variable greater than or equal to value
max	keep variable less than or equal to value
&	logical AND
|	logical OR
^	logical XOR
&/	logical AND NOT
|/	logical OR NOT
^/	logical XOR NOT

- **Variable** is assigned a value and/or manipulated.
- **Value(s)** is a variable/constant which affects the variable.

When you write programs that perform math, bear in mind the limitations of PBASIC's variables: all are positive integers; bits can represent 0 or 1; bytes, 0 to 255; and words, 0 to 65535. PBASIC doesn't understand floating-point numbers (like 3.14), negative numbers (–73), or numbers larger than 65535.

In most control applications, these are not serious limitations. For example, suppose you needed to measure temperatures from -50° to +200°F. By assigning a value of 0 to –50° and 65535 to +200° you would have a resolution of 0.0038°!

The integer restriction doesn't mean you can't do advanced math withthe Stamp. You just have to improvise . Suppose you needed to use the constant π (3.14159...) in a program. You would like to write:

```
Let w0 = b2 * 3.14
```

However, the number "3.14" is a floating-point number, which the Stamp doesn't understand. There is an alternative. You can express such quantities as fractions. Take the value 1.5. It is equivalent to the fraction 3/2. With a little effort you can find fractional substitutes for most floating-point values. For instance, it turns out that the fraction 22/7 comes very close to the value of π. To perform the calculation *Let w0 = b2 * 3.14*, the following instruction will do the trick:

```
Let w0 = b2 * 22 / 7
```

PBASIC works out problems from left to right. You cannot use parentheses to alter this order as you can in some other BASICs. And there is no "precedence of operators" that (for instance) causes multiplication to be done before addition. Many BASICs would evaluate the expression "2+3*4" as 14, because they would calculate "3*4" first, then add 2. PBASIC, working from left to right, evaluates the expression as 20, since it calculates "2+3" and multiplies the result by 4. When in doubt, work up an example problem and use debug to show you the result.

Sample Program:

```
pot 0,100,b3        ' Read pot, store result in b3.
LET b3=b3/2         ' Divide result by 2.
b3=b3 max 100       ' Limit result to 0-100.
                    ' Note that "LET" is not necessary.
```

LOOKDOWN *target,(value0,value1,...valueN),variable*

Search *value(s)* for *target* value. If target matches one of the values, store the matching value's position (0–N) in *variable*.

If no match is found, then the variable is unaffected.

- **Target** is the variable/constant being sought.
- **Value0, value1,...** is a list of values. The target value is compared to these values
- **Variable** holds the result of the search.

Lookdown's ability to convert an arbitrary sequence of values into an orderly sequence (0,1,2...) makes it a perfect partner for Branch. Using Lookdown and Branch together, you can create a SELECT CASE statement.

Sample Program:

```
' Program receives the following one-letter instructions over a serial
' linkand takes action: (G)o, (S)top, (L)ow, (M)edium, (H)igh.
Get_cmd: serin 0,N2400,b2' Put input value into b2.
      LOOKDOWN b2,("GSLMH"),b2          ' If b2="G" then b2=0 (see note)
                                        ' If b2="S" then b2=1
                                        ' If b2="L" then b2=2
                                        ' If b2="M" then b2=3
                                        ' If b2="H" then b2=4
      branch b2,(go,stop,low,med,hi)    ' If b2=0 then go
                                        ' If b2=1 then stop
                                        ' If b2=2 then low
                                        ' If b2=3 then med
                                        ' If b2=3 then hi
goto Get_cmd                            ' Not in range; try again.
go:   ...                  ' Destinations of the
stop: ...                  ' Branch instruction.
low:  ...
med:  ...
hi:       ...
' Note: In PBASIC instructions, including EEPROM, Serout, Lookup and
' Lookdown, strings may be formatted several ways. The Lookdown command
' above could also have been written:
'     LOOKDOWN b2,(71,83,76,77,72),b2   ' ASCII codes for "G","S","L"...
' or
'     LOOKDOWN b2,("G","S","L","M","H"),b2
```

LOOKUP *offset,(value0,value1,...valueN),variable*

Look up data specified by *offset* and store it in *variable*. For instance, if the values were 2, 13, 15, 28, 8 and *offset* was 1, then *variable* would get the value "13", since "13" is the second value in the list (the first value is #0, the second is #1, etc.). If *offset* is beyond the number of values given, then *variable* is unaffected.

- **Offset** specifies the index number of the value to be looked up.
- **Value0, value1,...** is a table of values.
- **Variable** holds the result of the lookup.

Many applications require the computer to calculate an output value based on an input value. When the relationship is simple, like "out = 2*in", it's no problem at all. But what about relationships that are not so obvious? In PBASIC you can use Lookup.

For example, stepper motors work in an odd way. They require a changing pattern of 1s and 0s controlling current to four coils. The sequence appears in the table to the right.

Repeating that sequence makes the motor turn. The program below shows how to use a Lookup table to generate the sequence.

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

Sample Program:

```
' Output the four-step sequence to drive a stepper motor w/on-screen simulation.
    let dirs = %00001111                    ' Set lower 4 pins to output.
Rotate:
    for b2 = 0 to 3
     LOOKUP b2,(10,9,5,6),b3                ' Convert offset (0-3)
                                            ' to corresponding step.
     let pins = b3                          ' Output the step pattern.
     LOOKUP b2,("|/-\"),b3                  ' Convert offset (0-3)
                                            ' to "picture" for debug.
     debug cls,#%pins,"  ",#@b3             ' Display value on pins,
    next                                    ' animated "motor."
goto Rotate                                 ' Do it again.

' Note: In the debug line above, there are two spaces between the quotes.
```

LOW *pin*

Make the specified pin output low. If the pin is programmed as an input, it changes to an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

You can think of the Low instruction as the equivalent of:

```
output 3            ' Make pin 3 an output.
let pin3 = 0        ' Make pin 3 low.
```

Notice that the Output command accepts the pin number (3), while Let requires the pin's variable name *pin3*. So, in addition to saving one instruction, Low allows you to make a pin output-low using only its number. When you look at the sample program below, imagine how difficult it would be to write it using Output and Let.

This also points out a common bug involving Low. Programmers sometimes substitute pin names like *pin3* for the pin number. Remember that those pin names are really bit variables. As bits, they can hold values of 0 or 1. The statement "Low pin3" is a valid PBASIC instruction, but it means, "Get the state of *pin3*. If *pin3* is 0, make *pin 0* output low. If *pin3* is 1, make *pin 1* output low."

Sample Program:

```
' One at a time, change each of the pins to output and make it low.
    for b2 = 0 to 7         ' Eight pins (0-7).
    LOW b2                  ' Clear pin no. indicated by b2.
    pause 500               ' Wait 1/2 second between pins.
    next                    ' Do the next pin.
```

NAP *period*

Enter sleep mode for a short period. Power consumption is reduced to about 20 µA, assuming no loads are beingdriven.

- **Period** is a variable/constant which determines the duration of the reduced power nap. The duration is (2^period) * approximately 18 ms. *Period* can range from 0 to 7, resulting in the nap lengths shown in the table.

Period	2^{period}	Nap Length
0	1	18 ms
1	2	36 ms
2	4	72 ms
3	8	144 ms
4	16	288 ms
5	32	576 ms
6	64	1152 ms
7	128	2304 ms

Nap uses the same shutdown/startup mechanism as Sleep, with one big difference. During sleep, the Stamp compensates for variations in the speed of the watchdog timer that serves as its alarm clock. As a result, longer sleep intervals are accurate to about ±1 percent. Naps are controlled by the watchdog timer without compensation. Variations in temperature, voltage, and manufacturing of the PBASIC chip can cause the actual timing to vary by as much as –50, +100 percent (i.e., a period-0 nap can range from 9 to 36 ms).

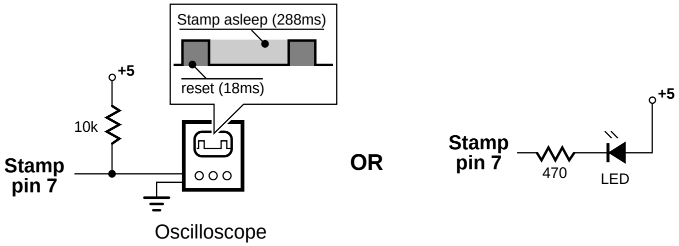
If your Stamp application is driving loads (sourcing or sinking current through output-high or output-low pins) during a nap, current will be interrupted for about 18 ms when the Stamp wakes up. The reason is that the reset that awakens the Stamp also switches all of the pins input mode for about 18 ms. When PBASIC regains control, it restores the I/O direction dictated by your program.

When you use End, Nap, or Sleep, make sure that your loads can tolerate these glitches. The simplest way is often to connect resistors high or low (to +5V or ground) as appropriate to ensure a continuing supply of current during reset.

The sample program on the next page can be used to demonstrate the effects of the nap glitch with either an LED and resistor, or an oscilloscope, as shown in the diagram.

Sample Program:

```
' During the Nap period, the Stamp will continue to drive loads connected to
' pins that are configured as outputs. However, at the end of a Nap, all pins
' briefly change to input, interrupting the current. This program may be
' used to demonstrate the effect.

        low 7                   ' Make pin 7 output-low.
Again:
        NAP 4                   ' Put the Stamp to sleep for 288 ms.
goto Again                      ' Nap some more.
```



Use either of these circuits to observe the output glitch when the Stamp awakens from a Nap. Pin 7 is output low while the Stamp is asleep. When it resets, all pins switch to input, allowing the resistor to pull pin 7 high (left) or causing the LED to blink off (right).

OUTPUT *pin*

Make the specified pin an output.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

When a program begins, all of the Stamp's pins are inputs. If you want to drive a load, like an LED or logic circuit, you must configure the appropriate pin as an output.

Output instructions (High, Low, Pulsout, Serout, Sound and Toggle) automatically change the specified pin to output and leave it in that state. Although not technically an output instruction, Pot also changes a pin to output. Writing 1s to particular bits of the variable *Dirs* causes the corresponding pins to become outputs. And then there's Output.

When a pin is configured as an output, you can change its state by writing a value to it, or to the variable *Pins*. When a pin is changed to output, it may be a 1 or a 0, depending on values previously written to the pin. To guarantee which state a pin will be in, either use the High or Low instructions to change it to output, or write the appropriate value to it immediately before switching to output.

Sample Program:

```
' To see this program in action, connect a 10k resistor from pin 7 to the +5
' power-supply rail. When the program runs, a debug window will show you the
' the state at pin 7 (a 1, due to the +5 connection); the effect of writing
' to an input pin (none); and the result of changing an input pin to output
' (the latched state appears on the pin and may be read by your program).
' Finally, the program will show how changing pin 7 to output wrote
' a 1 to the corresponding bit of the variable Dirs.

    input 7                                     ' Make pin 7 an input.
    debug "State present at pin 7: ",#pin7,cr,cr
    let pin7 = 0                                ' Write 0 to output latch.
    debug  "After 0 written to input: ",#pin7,cr,cr
    OUTPUT 7                                    ' Make pin 7 an output.
    debug  "After pin 7 changed to output: ",#pin7,cr
    debug "Dirs (binary): ",#%dirs
```

PAUSE *milliseconds*

Pause program execution for the specified number of milliseconds.

- **Milliseconds** is a variable/constant (0–65535) that specifies how many milliseconds to pause.

The delays produced by the Pause instruction are as accurate as the Stamp's ceramic resonator timebase, ±1 percent. When you use Pause in timing-critical applications, keep in mind the relatively low speed of the BASIC interpreter (about 2000 instructions per second). This is the time required for the PBASIC chip to read and interpret an instruction stored in the EEPROM.

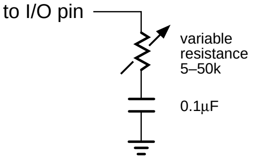
Since the PBASIC chip takes 0.5 milliseconds to read in the Pause instruction, and 0.5 milliseconds to read in the instruction following it, you can count on loops involving Pause taking at least 1 millisecond longer than the Pause period itself. If you're programming timing loops of fairly long duration, keep this (and the 1-percent tolerance of the timebase) in mind.

Sample Program:

```
abc:
        low 2               ' Make pin 2 output low.
        PAUSE 100           ' Pause for 0.1 second.
        high 2              ' Make pin 2 output high.
        PAUSE 100           ' Pause for 0.1 second.
goto abc
```

POT *pin,scale,variable*

Read a 5–50k potentiometer, thermistor, photocell, or other variable resistance. The pin specified by Pot must be connected to one side of a resistor, whose other side is connected through a capacitor to ground. A resistance measurement is taken by timing how long it takes to discharge the capacitor through the resistor. If the pin is an input when Pot executes, it will be changed to output.



- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Scale** is a variable/constant (0–255) used to scale the instruction's internal 16-bit result. The 16- bit reading is multiplied by (scale/ 256), so a scale value of 128 would reduce the range by approximately 50%, a scale of 64 would reduce to 25%, and so on. The Alt-P option (explained below) provides a means to find the best scale value for a particular resistor.
- **Variable** is used to store the final result of the reading. Internally, the Pot instruction calculates a 16-bit value, which is scaled down to an 8-bit value. The amount by which the internal value must be scaled varies with the size of the resistor being used.

Finding the best Pot scale value:

- To find the best scale value, connect the resistor to be used with the Pot instruction to the Stamp, and connect the Stamp to the PC.
- Press Alt-P while running the Stamp's editor software. A special calibration window appears, allowing you to find the best value.
- The window asks for the number of the I/O pin to which the resistor is connected. Select the appropriate pin (0-7).
- The editor downloads a short program to the Stamp (this overwrites any program already stored in the Stamp).
- Another window appears, showing two numbers: scale and value. Adjust the resistor until the smallest possible number is shown for scale (we're assuming you can easily adjust the resistor, as with a potentiometer).

Once you've found the smallest number for scale, you're done. This number should be used for the scale in the Pot instruction.

Optionally, you can verify the scale number found above by pressing the spacebar. This locks the scale and causes the Stamp to read the resistor continuously. The window displays the value. If the scale is good, you should be able to adjust the resistor, achieving a 0–255 reading for the value (or as close as possible). To change the scale value and repeat this step, just press the spacebar. Continue this process until you find the best scale.

Sample Program:

```
abc:
        POT 0,100,b2                    ' Read potentiometer on pin 0.
        serout 1,N300,(b2)              ' Send potentiometer reading
                                        ' over serial output.
goto abc                                ' Repeat the process.
```

PULSIN *pin,state,variable*

Change the specified pin to input and measure an input pulse in 10µs units.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **State** is a variable/constant (0 or 1) that specifies which edge must occur before beginning the measurement.
- **Variable** is a variable used to store the result of the measurement. The variable may be a word variable with a range of 1 to 65535, or a byte variable with a range of 1 to 255.

Many analog properties (voltage, resistance, capacitance, frequency, duty cycle) can be measured in terms of pulse durations. This makes Pulsin a valuable form of analog-to-digital conversion.

You can think of Pulsin as a fast stopwatch that is triggered by a change in state (0 or 1) on the specified pin. When the state on the pin changes to the state specified in Pulsin, the stopwatch starts counting. When the state on the pin changes again, the stopwatch stops.

If the state of the pin doesn't change (even if it is already in the state specified in the Pulsin instruction), the stopwatch won't trigger. Pulsin waits a maximum of 0.65535 seconds for a trigger, then returns with 0 in *variable*.

The variable can be either a word or a byte. If the variable is a word, the value returned by Pulsin can range from 1 to 65535 units of 10µs. If the variable is a byte, the value returned can range from 1 to 255 units of 10µs. Internally, Pulsin always uses a 16-bit timer. When your program specifies a byte, Pulsin stores the lower 8 bits of the internal counter into it. Pulse widths longer than 2550µs will give false, low readings with a byte variable. For example, a 2560 µs pulse returns a Pulsin reading of 256 with a word variable and 0 with a byte variable.

Sample Program:

```
PULSIN 4,0,w2           ' Measure an input pulse on pin 4.
serout 1,n300,(b5)      ' Send high byte of 16-bit pulse measurement
 . . .                  '  over serial output.
```

PULSOUT *pin,time*

Generate a pulse by inverting a pin for a specified amount of time. If the pin is an input when Pulsout is executed, it will be changed to an output.

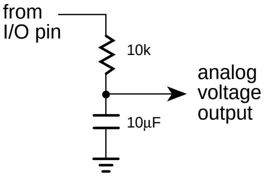
- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Time** is a variable/constant (0–65535) that specifies the length of the pulse in 10µs units.

Sample Program:

```
abc:
        PULSOUT 0,3           ' Invert pin 0 for 30
                              ' microseconds.
        pause 1               ' Pause for 1 ms.
goto abc                      ' Branch to abc.
```

PWM *pin,duty,cycles*

Output pulse-width-modulation on a pin, then return the pin to input state. PWM can be used to generate analog voltages (0-5V) through a pin connected to a resistor and capacitor to ground; the resistor-capacitor junction is the analog output (see circuit). Since the capacitor gradually discharges, PWM should be executed periodically to update and/or refresh the analog voltage.



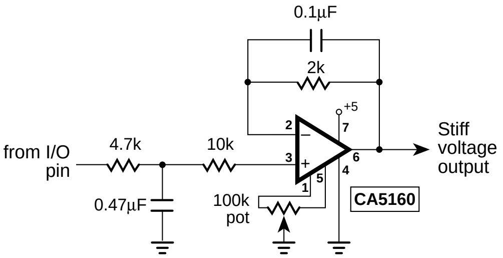
- **Pin** is a variable/constant (0–7) which specifies the I/O pin to use.
- **Duty** is a variable/constant (0–255) which specifies the analog level desired (0–5 volts).
- **Cycles** is a variable/constant (0–255) which specifies the number of cycles to output. Larger capacitors require multiple cycles to fully charge. Each cycle takes about 5 ms.

PWM emits a burst of 1s and 0s whose ratio is proportional to the *duty* value you specify. If *duty* is 0, then the pin is continuously low (0); if *duty* is 255, then the pin is continuously high. For values in between, the proportion is *duty*/255. For example, if *duty* is 100, the ratio of 1s to 0s is 100/255 = 0.392, approximately 39 percent.

When such a burst is used to charge a capacitor arranged as shown in the schematic, the voltage across the capacitor is equal to (*duty*/255) * 5. So if *duty* is 100, the capacitor voltage is (100/255) * 5 = 1.96 volts.

This voltage will drop as the capacitor discharges through whatever load it is driving. The rate of discharge is proportional to the current drawn by the load; more current = faster discharge. You can combat this effect in software by refreshing the capacitor's charge with frequent doses of PWM.

You can also buffer the output to greatly reduce the need for frequent PWM cycles. The schematic on the next page shows an example. Feel free to substitute more sophisticated circuits; this "op-amp follower" is merely a suggestion.



Op-amp buffer for PWM.

If you use a buffer circuit, you will still have to refresh the capacitor from time to time. When the pin is configured to input after PWM executes, it is effectively disconnected from the resistor/capacitor circuit. However, leakage currents of up to 1µA can flow into or out of this "disconnected" pin. Over time, these small currents will cause the voltage on the capacitor to drift. The same applies for leakage current from the op-amp's input, as well as the capacitor's own internal leakage. Executing PWM occasionally will reset the capacitor voltage to the intended value.

One more thing: The name "PWM" may lead you to expect a neat train of fixed-width pulses for a given duty value. That's not the case. When viewed on an oscilloscope, the PWM output looks like a noisy jumble of varying pulsewidths. The only guarantee is that the overall ratio of highs to lows is in the proportion specified by *duty*.

Sample Program:

```
abc:
      serin 0,n300,b2          ' Receive serial byte.
      PWM 1,b2,20              ' Output an analog voltage proportional to
                               ' the serial byte received
```

RANDOM *wordvariable*

Generate the next pseudo-random number in *wordvariable*.

- **Wordvariable** is a variable (0–65535) that acts as the routine's workspace and its result. Each pass through Random leaves the next number in the pseudorandom sequence.

The Stamp uses a sequence of 65535 essentially random numbers to execute this instruction. When Random executes, the value in wordvariable determines where to "tap" into the sequence of random numbers. If the same initial value is always used in *wordvariable*, then the same sequence of numbers is generated. Although this method is not absolutely random, it's good enough for most applications.

To obtain truly random results, you must add an element of uncertainty to the process. For instance, your program might execute Random continuously while waiting for the user to press a button.

Sample Program:

```
loop:
      RANDOM w1          ' Generate a 16-bit random number.
      sound 1,(b2,10)    ' Generate a random tone on pin 1 using the low
                         ' byte of the random number b2 as the note number.
      goto loop          ' Repeat the process
```

READ *location,variable*

Read EEPROM location and store value in *variable*.

- **Location** is a variable/constant (0–255) that specifies which location in the EEPROM to read from.
- **Variable** receives the value read from the EEPROM (0–255).

The EEPROM is used for both program storage (which builds downward from address 254) and data storage (which builds upward from address 0). To ensure that your program doesn't overwrite itself, read location 255 in the EEPROM before writing any data. Location 255 holds the address of the last instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data.

Sample Program:

```
        READ 255,b2             ' Get location of last program instruction.
loop:
        b2 = b2 - 1             ' Decrement to next available EEPROM location
        serin 0,N300,b3         ' Receive serial byte in b3
        write b2,b3             ' Store received serial byte in next EEPROM location
if b2 > 0 then loop             ' Get another byte if there's room left to store it.
```

RETURN

Return from subroutine. Return branches back to the address following the most recent Gosub instruction, at which point program execution continues.

Return takes no parameters. For more information on using subroutines, see the Gosub listing.

Sample Program:

```
for b4 = 0 to 10
gosub abc                     ' Save return address and then branch to abc.
next
abc:
      pulsout 0,b4            ' Output a pulse on pin 0.
                              ' Pulse length is b4 x 10 microseconds.
      toggle 1                ' Toggle pin 1.
      RETURN                  ' Return to instruction following gosub.
```

REVERSE *pin*

Reverse the data direction of the given pin. If the pin is an input, make it an output; if it's an output, make it an input.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin.

See the Input and Output commands for more information on configuring pins' data directions.

Sample Program:

```
dir3 = 0                ' Make pin 3 an input.
REVERSE 3               ' Make pin 3 an output.
REVERSE 3               ' Make pin 3 an input.
```

SERIN *pin,baudmode,(qualifier,qualifier,...)*

SERIN *pin,baudmode,{#}variable,{#}variable,...*

SERIN *pin, baudmode, (qualifier,qualifier,...), {#}variable, {#}variable,...*

Set up a serial input port and then wait for optional qualifiers and/or variables.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Baudmode** is a variable/constant (0–7) that specifies the serial port mode. *Baudmode* can be either the # or symbol shown in the table. The other serial parameters are preset to the most common format: no parity, eight data bits, one stop bit, often abbreviated N81. These cannot be changed.

#	Symbol	Baud Rate	Polarity
0	T2400	2400	true
1	T1200	1200	true
2	T600	600	true
3	T300	300	true
4	N2400	2400	inverted
5	N1200	1200	inverted
6	N600	600	inverted
7	N300	300	inverted

- **Qualifiers** are optional variables/constants (0–255) which must be received in exact order before execution can continue.
- **Variables** (optional) are used to store received data (0–255). If any qualifiers are given, they must be satisfied before variables can be filled. If a # character precedes a variable name, then Serin will convert numeric text (e.g., numbers typed at a keyboard) into a value to fill the variable.

Serin makes the specified pin a serial input port with the characteristics set by *baudmode*. It receives serial data one byte at a time and does one of two things with it:

- Compares it to a *qualifier*.
- Stores it to a *variable.*

In either case, the Stamp will do nothing else until all qualifiers have been exactly matched in the specified order and all variables have been filled. A single Serin instruction can include both variables to fill and qualifiers to match.

Here are some examples:

```
SERIN 0,T300,b2
```

Stop the program until one byte of data is received serially (true polarity, 300 baud) through pin 0. Store the received byte into variable b2 and continue. For example, if the character "A" were received, Serin would store 65 (the ASCII character code for "A") into b2.

```
SERIN 0, T1200,#w1
```

Stop the program until a a numeric string is received serially (true polarity, 1200 baud) through pin 0. Store the value of the numeric string into variable w1. For example, suppose the following text were received: "XYZ: 576%." Serin would ignore "XYZ: " because these are non-numeric characters. It would collect the characters "5", "7", "6" up to the first non-numeric character, "%". Serin would convert the numeric string "576" into the corresponding value 576 and store it in w1. If the # before w1 were omitted, Serin would receive only the first character, "X", and store its ASCII character code, 88, into w1.

```
SERIN 0,N2400,("A")
```

Stop the program until a byte of data is received serially (inverted polarity, 2400 baud) through pin 0. Compare the received byte to 65, the ASCII value of the letter "A". If it matches, continue the program. If it doesn't match, receive another byte and repeat the comparison. The program will not continue until "A" is received. For example, if Serin received "LIMIT 65,A", program execution would not continue until the final "A" was received.

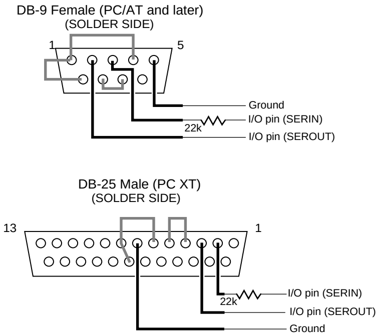
```
SERIN 0,N2400,("SESAME"),b2,#b4
```

Stop the program until a string of bytes exactly matching "SESAME" is received serially (inverted polarity, 2400 baud) through pin 0. Once the qualifiers have been received, store the next byte into b2. Then receive a numeric string, convert it to a value, and store it into b4. For example, suppose Serin received, "...SESESAME! *****19*". It would ignore the string "...SE", then accept the matching qualifier string "SESAME". Then Serin would put 33, the ASCII value of "!", into b2. It would ignore the non-numeric "*" characters, then store the characters "1" and "9". When Serin received the first non-numeric character ("*"), it would convert the text "19" into the value 19 and store it in b4. Then, having matched all qualifiers and filled all variables, Serin would permit the Stamp to go on to the next instruction.

Speed Considerations. The Serin command itself is fast enough to catch multiple bytes of data, no matter how rapidly the host computer sends them. However, if your program receives data, stores or processes it, then loops back to perform another Serin, it may miss data or receive it incorrectly because of the time delay. Use one or more of the following steps to compensate for this:

- Increase the number of stop bits at the sender from 1 to 2 (or more, if possible).
- Reduce the baud rate.
- If the sender is operating under the control of a program, add delays between transmissions.
- Reduce the amount of processing that the Stamp performs between Serins to a bare minimum.

Receiving data from a PC. To send data serially from your PC to the Stamp, all you need is a 22k resistor, some wire and connectors, and terminal communication software. Wire the connector as shown in the diagram for Serin. The wires shown in gray disable your PC's hardware handshaking, which would normally require additional connections to control the flow of data. These aren't required in communication with the Stamp, because you're not likely to be sending a large volume of data as you might to a modem orprinter.



When you write programs to receive serial data using this kind of hookup, make sure to specify "inverted" baudmodes, such as N2400.

If you don't have a terminal program, you can type and run the following QBASIC program to configure the serial port (2400 baud, N81) and transmit characters typed at the keyboard. QBASIC is the PC dialect of BASIC that comes with DOS versions 5 and later.

QBASIC Program to Transmit Data:

```
' This program transmits characters typed at the keyboard out the PC's
' COM1: serial port. To end the program, press control-break.
' Note: in the line below, the "0" in "CD0,CS0..." is a zero.

OPEN "COM1:2400,N,8,1,CD0,CS0,DS0,OP0" FOR OUTPUT AS #1
CLS
Again:
        theChar$ = INKEY$
        IF theChar$ = "" then Goto Again
        PRINT #1,theChar$;
GOTO Again
```

Sample Stamp Program:

```
' To use this program, download it to the Stamp. Connect
' your PC's com1: port output to Stamp pin 0 through a 22k resistor
' as shown in the diagram. Connect a speaker to Stamp pin 7 as
' shown in the Sound entry. Run your terminal software or the QBASIC
' program above. Configure your terminal software for 2400 baud,
' N81, and turn off hardware handshaking. The QBASIC
' program is already configured this way. Try typing random
' letters and numbers--nothing will happen until you enter
' "START" exactly as it appears in the Stamp program.
' Then you may type numbers representing notes and
' durations for the Sound command. Any non-numeric text
' you type will be ignored.

        SERIN 0,N2400,("START")             ' Wait for "START".
        sound 7,(100,10)                    ' Acknowledging beep.
Again:
        SERIN 0,N2400,#b2,#b3               ' Receive numeric text and
                                            ' convert to bytes.
        sound 7,(b2,b3)                     ' Play corresponding sound.
        goto Again                          ' Repeat.
```

SEROUT *pin,baudmode,({#}data,{#}data,...)*

Set up a serial output port and transmit data.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Baudmode** is a variable/constant (0–15) that specifies the serialport mode. *Baudmode* can be either the # or symbol shown in the table. The other serial parameters are preset to the most common format: no parity, eight data bits, one stop bit, often abbreviated N81. These cannot be changed.
- **Data** are byte variables/constants (0–255) that are output by Serout. If preceded by the # sign, data items are transmitted as text strings up to five characters long. Without the #, data items are transmitted as a single byte.

#	Symbol	Baud Rate	Polarity and Output Mode
0	T2400	2400	true always driven
1	T1200	1200	true always driven
2	T600	600	true always driven
3	T300	300	true always driven
4	N2400	2400	inverted always driven
5	N1200	1200	inverted always driven
6	N600	600	inverted always driven
7	N300	300	inverted always driven
8	OT2400	2400	true open drain (driven high)
9	OT1200	1200	true open drain (driven high)
10	OT600	600	true open drain (driven high)
11	OT300	300	true open drain (driven high)
12	ON2400	2400	inverted open source (driven low)
13	ON1200	1200	inverted open source (driven low)
14	ON600	600	inverted open source (driven low)
15	ON300	300	inverted open source (driven low)

Serout makes the specified pin a serial output port with the characteristics set by *baudmode*. It transmits the specified data in one of two forms:

- A single-byte value.
- A text string of one to five characters.

Here are some examples:

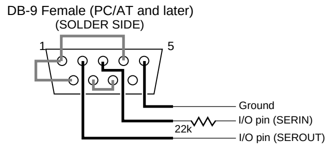
```
SEROUT 0,N2400,(65)
```

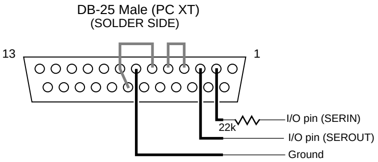
Serout transmits the byte value 65 through pin 0 at 2400 baud, inverted. If you receive this byte on a PC running terminal software, the character "A" will appear on the screen, because 65 is the ASCII code for "A".

```
SEROUT 0,N2400,(#65)
```

Serout transmits the text string "65" through pin 0 at 2400 baud, inverted. If you receive this byte on a PC running terminal software, the text “65” will appear on the screen. When a value is preceded by the # sign, Serout automatically converts it to a form that reads correctly on a terminal screen.

When should you use the # sign? If you are sending data from the Stamp to a terminal for people to read, use #. If you are sending data to another Stamp, or to another computer for further processing, it’s more efficient to omit the #.





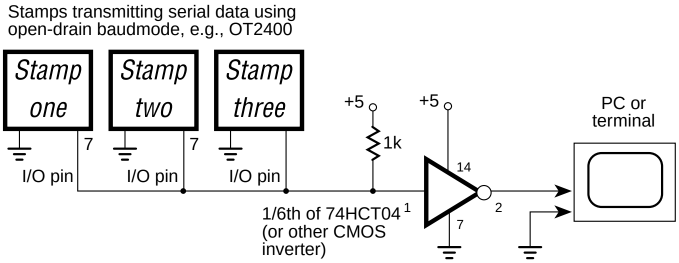
Sending data to a PC. To send data serially to your PC from the Stamp, all you need is some wire and connectors, and terminal communication software. Wire the connector as shown in the Serout connections in the diagram at right and use the inverted baudmodes, such as N2400. Although the Stamp’s serial output can only switch between 0 and +5 volts (not the ±10 volts of legal RS-232), most PCs receive it without problems.

If you don’t have a terminal program, you can type and run the following QBASIC program to configure the serial port and receive characters from the Stamp.

QBASIC Program to Receive Data:

```
' This program receives data transmitted by the Stamp through the PC's
' COM1: serial port and displays it on the screen. To end the program,
' press control-break. Note: in the line below, the "0" in "CD0,CS0..." is a zero.

OPEN "COM1:2400,N,8,1,CD0,CS0,DS0,OP0" FOR INPUT AS #1
CLS
Again:
      theChar$ = INPUT$(1,#1)
      PRINT theChar$;
GOTO Again
```

Open-drain/open-source signaling. The last eight configuration options for Serout begin with “O” for open-drain or open-source signaling. The diagram below shows how to use the open-drain mode to connect two or more Stamps to a common serial output line to form a network. You could also use the open-source mode, but the resistor would have to be connected to ground, and a buffer (non-inverting driver) substituted for the inverter to drive the PC.



To understand why you must use the "open" serial modes on a network, consider what would happen if you didn't. When none of the Stamps are transmitting, all of their Serout pins are output-high. Since all are at +5 volts, no current flows between the pins. Then a Stamp transmits, and switches to output-low. With the other Stamps' pins output-high, there's a direct short from +5 volts to ground. Current flows between the pins, possibly damaging them.

If the Stamps are all set for open-drain output, it's a different story. When the Stamps aren't transmitting, their Serout pins are inputs, effectively disconnected from the serial line. The resistor to +5 volts maintains a high on the serial line. When a Stamp transmits, it pulls the serial line low. Almost no current flows through the other Stamps' Serout pins, which are set to input. Even if two Stamps transmit simultaneously, they can't damage each other.

Sample Program:

```
abc:
     pot 0,100,b2                        ' Read potentiometer on pin 0.
     SEROUT 1,N300,(b2)                  ' Send potentiometer
                                         ' reading over serial output.
     goto abc                            ' Repeat the process.
```

SLEEP *seconds*

Enter sleep mode for a specified number of seconds.

- **Seconds** is a variable/constant (1–65535) that specifies the duration of sleep in seconds. The length of sleep can range from 2.3 seconds (see note below) to slightly over 18 hours. Power consumption is reduced to about 20 µA, assuming no loads are being driven.

Note: The resolution of Sleep is 2.304 seconds. Sleep rounds the *seconds* up to the nearest multiple of 2.304. Sleep 1 causes 2.3 seconds of sleep, while Sleep 10 causes 11.52 seconds (5 x 2.304).

Sleep lets the Stamp turn itself off, then turn back on after a specified number of seconds. The alarm clock that wakes the Stamp up is called the watchdog timer. The watchdog is an oscillator built into the BASIC interpreter. During sleep, the Stamp periodically wakes up and adjusts a counter to determine how long it has been asleep. If it isn't time to wake up, the Stamp goes back to sleep.

To ensure accuracy of sleep intervals, the Stamp periodically compares the period of the watchdog timer to the more accurate resonator timebase. It calculates a correction factor that it uses during sleep. Longer sleep intervals are accurate to ±1 percent.

If your Stamp application is driving loads during sleep, current will be interrupted for about 18 ms when the Stamp wakes up every 2.3 seconds. The reason is that the reset that awakens the Stamp causes all of the pins to switch to input mode for approximately 18 ms. When the BASIC interpreter regains control, it restores the I/O direction dictated by your program.

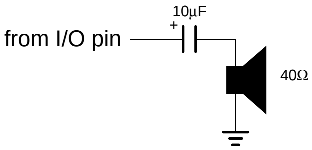
If you plan to use End, Nap, or Sleep in your programs, make sure that your loads can tolerate these periodic power outages. The simplest solution is to connect resistors high or low (to +5V or ground) as appropriate to ensure a supply of current during reset.

Sample Program:

```
SLEEP 3600          ' Sleep for about 1 hour.
goto xyz            ' Continue with program
                    ' after sleeping.
```

SOUND *pin,(note,duration,note,duration,...)*

Change the specified pin to output, and generate square-wave notes with given durations. The output pin should be connected as shown in the diagram. You may substitute a resistor of 220 ohms or more for the capacitor, but the speaker coil will draw current even when the speaker is silent.



- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.
- **Note(s)** are variables/constants (0–255) which specify type and frequency. Note 0 is silent for the given duration. Notes 1-127 are ascending tones. Notes 128-255 are ascending white noises, ranging from buzzing (128) to hissing (255).
- **Duration(s)** are variables/constants (1–255) which specify how long (in units of 12 ms) to play each note.

The notes produced by `Sound` can vary in frequency from 94.8 Hz (1) to 10,550 Hz (127). If you need to determine the frequency corresponding to a given note value, or need to find the note value that will give you best approximation for a given frequency, use the equations below.

Sample Program:

```
for b2 = 0 to 256
      SOUND 1,(25,10,b2,10)                    ‘ Generate a constant tone (25)
                                               ‘ followed by an ascending tone
                                               ‘ (b2). Both tones have the
next                                           ‘ same duration(10).
```

$$\textbf{Note} = 127 - \frac{\frac{1}{\textbf{Frequency (Hz)}} - 95 \times 10^{-6}}{83 \times 10^{-6}}$$

$$\textbf{Frequency (Hz)} = \frac{1}{95 \times 10^{-6} + \left(\left(127 - \textbf{Note}\right) \times 83 \times 10^{-6}\right)}$$

TOGGLE *pin*

Make pin an output and toggle state.

- **Pin** is a variable/constant (0–7) that specifies the I/O pin to use.

Sample Program:

```
for b2 = 1 to 25
TOGGLE 5                    'Toggle state of pin 5.
next
```

WRITE *location,data*

Store data in EEPROM location.

- **Location** is a variable/constant (0–255) that specifies which EEPROM location to write to.
- **Data** is a variable/constant (0–255) that is stored in the EEPROM location.

The EEPROM is used for both program storage (which builds downward from address 254) and data storage (which builds upward from address 0). To ensure that your program doesn't overwrite itself, read location 255 in the EEPROM before writing any data. Location 255 holds the address of the first instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data.

Sample Program:

```
read 255,b2               ' Get location of last
                          ' program instruction.
loop:
      b2 = b2 - 1         ' Decrement to next
                          ' available EEPROM location
      serin 0,N300,b3     ' Receive serial byte in b3.
      WRITE b2,b3         ' Store received serial
                          ' byte in next EEPROM location.
if b2 > 0 then loop       ' Get another byte if there's room.
```

Introduction. This application note presents a program in PBASIC that enables the BASIC Stamp to operate as a simple user-interface terminal.

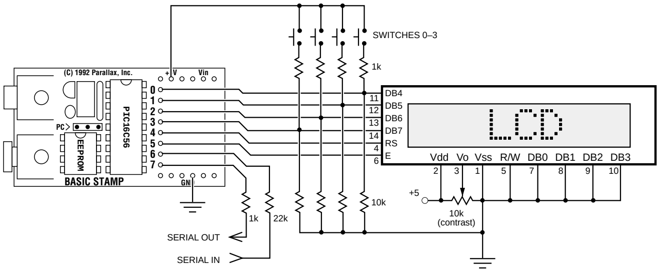
Background. Many systems use a central host computer to control remote functions. At various locations, users communicate with the main system via small terminals that display system status and accept inputs. The BASIC Stamp's ease of programming and built-in support for serial communications make it a good candidate for such user-interface applications.

The liquid-crystal display (LCD) used in this project is based on the popular Hitachi 44780 controller IC. These chips are at the heart of LCD's ranging in size from two lines of four characters (2x4) to 2x40.

How it works. When power is first applied, the BASIC program initializes the LCD. It sets the display to print from left to right, and enables an underline cursor. To eliminate any stray characters, the program clears the screen.

After initialization, the program enters a loop waiting for the arrival of a character via the 2400-baud RS-232 interface. When a character arrives, it is checked against a short list of special characters (backspace, control-C, and return). If it is not one of these, the program prints it on the display, and re-enters the waiting-for-data loop.

If a backspace is received, the program moves the LCD cursor back one



Schematic to accompany program TERMINAL.BAS.

space, prints a blank (space) character to blot out the character that was there, and then moves back again. The second move-back step is necessary because the LCD automatically advances the cursor.

If a control-C is received, the program issues a clear instruction to the LCD, which responds by filling the screen with blanks, and returning the cursor to the leftmost position.

If a return character is received, the program interprets the message as a query requiring a response from the user. It enters a loop waiting for the user to press one of the four pushbuttons. When he does, the program sends the character (“0” through “3”) representing the button number back to the host system. It then re-enters its waiting loop.

Because of all this processing, the user interface cannot receive characters sent rapidly at the full baud rate. The host program must put a little breathing space between characters; perhaps a 3-millisecond delay. If you reduce the baud rate to 300 baud and set the host terminal to 1.5 or 2 stop bits, you may avoid the need to program a delay.

At the beginning of the program, during the initialization of the LCD, you may have noticed that several instructions are repeated, instead of being enclosed in for/next loops. This is not an oversight. Watching the downloading bar graph indicated that the repeated instructions actually resulted in a more compact program from the Stamp’s point of view. Keep an eye on that graph when running programs; it a good relative indication of how much program space you’ve used. The terminal program occupies about two-thirds of the Stamp’s EEPROM.

From an electronic standpoint, the circuit employs a couple of tricks. The first involves the RS-232 communication. The Stamp’s processor, a PIC 16C56, is equipped with hefty static-protection diodes on its input/output pins. When the Stamp receives RS-232 data, which typically swings between -12 and +12 volts (V), these diodes serve to limit the voltage actually seen by the PIC’s internal circuitry to 0 and +5V. The 22k resistor limits the current through the diodes to prevent damage.

Sending serial output without an external driver circuit exploits another loophole in the RS-232 standard. While most RS-232 devices expect the signal to swing between at least -3 and +3V, most will accept the 0 and +5V output of the PIC without problems.

This setup is less noise-immune than circuits that play by the RS-232 rules. If you add a line driver/receiver such as a Maxim MAX232, remember that these devices also invert the signals. You'll have to change the baud/mode parameter in the instructions serin and serout to T2400, where T stands for true signal polarity. If industrial-strength noise immunity is required, or the interface will be at the end of a mile-long stretch of wire, use an RS-422 driver/receiver. This will require the same changes to serin and serout.

Another trick allows the sharing of input/output pins between the LCD and the pushbuttons. What happens if the user presses the buttons while the LCD is receiving data? Nothing. The Stamp can sink enough current to prevent the 1k pullup resistors from affecting the state of its active output lines. And when the Stamp is receiving input from the switches, the LCD is disabled, so its data lines are in a high-impedance state that's the next best thing to not being there. These facts allow the LCD and the switches to share the data lines without interference.

Finally, note that the resistors are shown on the data side of the switches, not on the +5V side. This is an inexpensive precaution against damage or interference due to electrostatic discharge from the user's fingertips. It's not an especially effective precaution, but the price is right.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' PROGRAM: Terminal.bas
' The Stamp serves as a user-interface terminal. It accepts text via RS-232 from a
' host, and provides a way for the user to respond to queries via four pushbuttons.

Symbol   S_in     =    7                    ' Serial data input pin
Symbol   S_out    =    6                    ' Serial data output pin
Symbol   E        =    5                    ' Enable pin, 1 = enabled
Symbol   RS       =    4                    ' Register select pin, 0 = instruction
Symbol  keys      =    b0                   ' Variable holding # of key pressed.
Symbol   char     =    b3                   ' Character sent to LCD.
Symbol  Sw_0    =    pin0                   ' User input switches
Symbol  Sw_1    =    pin1                   ' multiplexed w/LCD data lines.
Symbol  Sw_2    =    pin2
Symbol  Sw_3    =    pin3

' Set up the Stamp's I/O lines and initialize the LCD.
begin:   let pins = 0                       ' Clear the output lines
         let dirs = %01111111               ' One input, 7 outputs.
         pause 200                          ' Wait 200 ms for LCD to reset.

' Initialize the LCD in accordance with Hitachi's instructions for 4-bit interface.
i_LCD:   let pins = %00000011               ' Set to 8-bit operation.
         pulsout E,1                        ' Send data three times
         pause 10                           ' to initialize LCD.
         pulsout E,1
         pause 10
         pulsout E,1
         pause 10
         let pins = %00000010               ' Set to 4-bit operation.
         pulsout E,1                        ' Send above data three times.
         pulsout E,1
         pulsout E,1
         let char = 14                      ' Set up LCD in accordance with
         gosub wr_LCD                       ' Hitachi instruction manual.
         let char = 6                       ' Turn on cursor and enable
         gosub wr_LCD                       ' left-to-right printing.
         let char = 1                       ' Clear the display.
         gosub wr_LCD
         high    RS                         ' Prepare to send characters.

' Main program loop: receive data, check for backspace, and display data on LCD.
main:    serin S_in,N2400,char              ' Main terminal loop.
         goto bksp
out:     gosub wr_LCD
         goto main

' Write the ASCII character in b3 to LCD.
wr_LCD:  let pins = pins & %00010000
         let b2 = char/16                   ' Put high nibble of b3 into b2.
         let pins = pins | b2               ' OR the contents of b2 into pins.
         pulsout E,1                        ' Blip enable pin.
         let b2 = char & %00001111          ' Put low nibble of b3 into b2.
         let pins = pins & %00010000        ' Clear 4-bit data bus.
         let pins = pins | b2               ' OR the contents of b2 into pins.
         pulsout E,1                        ' Blip enable.
         return

' Backspace, rub out character by printing a blank.
```

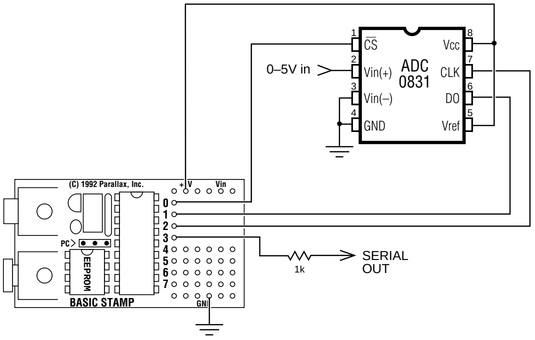
```
bksp:   if char > 13 then out           ' Not a bksp or cr? Output character.
        if char = 3 then clear          ' Ctl-C clears LCD screen.
        if char = 13 then cret          ' Carriage return.
        if char <> 8 then main          ' Reject other non-printables.
        gosub back
        let char = 32                   ' Send a blank to display
        gosub wr_LCD
        gosub back                      ' Back up to counter LCD's auto-
                                        ' increment.
        goto main                       ' Get ready for another transmission.

back:   low RS                          ' Change to instruction register.
        let char = 16                   ' Move cursor left.
        gosub wr_LCD                    ' Write instruction to LCD.
        high RS                         ' Put RS back in character mode.
        return

clear:  low RS                          ' Change to instruction register.
        let b3 = 1                      ' Clear the display.
        gosub wr_LCD                    ' Write instruction to LCD.
        high RS                         ' Put RS back in character mode.
        goto main

' If a carriage return is received, wait for switch input from the user. The host
' program (on the other computer) should cooperate by waiting for a reply before
' sending more data.
cret:   let dirs = %01110000            ' Change LCD data lines to input.
loop:   let keys = 0
        if Sw_0 = 1 then xmit           ' Add one for each skipped key.
        let keys = keys + 1
        if Sw_1 = 1 then xmit
        let keys = keys + 1
        if Sw_2 = 1 then xmit
        let keys = keys + 1
        if Sw_3 = 1 then xmit
        goto loop

xmit:   serout S_out,N2400,(#keys,10,13)
        let dirs = %01111111            ' Restore I/O pins to original state.
        goto main
```

Introduction. This application note presents the hardware and software required to interface an 8-bit serial analog-to-digital converter to the Parallax BASIC Stamp.

Background. The BASIC Stamp's instruction pot performs a limited sort of analog-to-digital conversion. It lets you interface nearly any kind of resistive sensor to the Stamp with a minimum of difficulty. However, many applications call for a true voltage-mode analog-to-digital converter (ADC). One that’s particularly suited to interfacing with the Stamp is the National Semiconductor ADC0831, available from Digi-Key, among others.

Interfacing the ’831 requires only three input/output lines, and of these, two can be multiplexed with other functions (or additional ’831’s). Only the chip-select (CS) pin requires a dedicated line. The ADC’s range of input voltages is controlled by the VREF and VIN(–) pins. VREF sets the voltage at which the ADC will return a full-scale output of 255, while VIN(–) sets the voltage that will return 0.

In the example application, VIN(–) is at ground and VREF is at +5; however, these values can be as close together as 1 volt without harming the device’s accuracy or linearity. You may use diode voltage references or trim pots to set these values.



Schematic to accompany program AD_CONV.BAS.

How it works. The sample program reads the voltage at the '831's input pin every 2 seconds and reports it via a 2400-baud serial connection. The subroutine conv handles the details of getting data out of the ADC. It enables the ADC by pulling the CS line low, then pulses the clock (CLK) line to signal the beginning of a conversion. The program then enters a loop in which it pulses CLK, gets the bit on pin AD, adds it to the received byte, and shifts the bits of the received byte to the left. Since BASIC traditionally doesn't include bit-shift operations, the program multiplies the byte by 2 to perform the shift.

When all bits have been shifted into the byte, the program turns off the ADC by returning CS high. The subroutine returns with the conversion result in the variable data. The whole process takes about 20 milliseconds.

Modifications. You can add more '831's to the circuit as follows: Connect each additional ADC to the same clock and data lines, but assign it a separate CS pin. Modify the conv subroutine to take the appropriate CS pin low when it needs to acquire data from a particular ADC. That's it.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' PROGRAM: ad_conv.bas
' BASIC Stamp program that uses the National ADC0831 to acquire analog data and
' output it via RS-232.

Symbol  CS      =   0
Symbol  AD      =   pin1
Symbol  CLK     =   2
Symbol  S_out   =   3
Symbol  data    =   b0
Symbol  i       =   b2

setup:  let pins = 255                  ' Pins high (deselect ADC).
        let dirs = %11111101            ' S_out, CLK, CS outputs; AD
                                        ' input.

loop:   gosub conv                      ' Get the data.
        serout S_out,N2400,(#b0,13,10)  ' Send data followed by a return
```

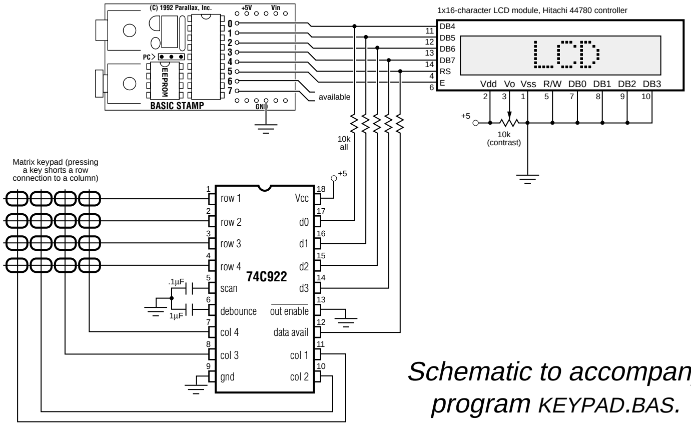
```
                                                  ' and linefeed.
          pause 2000                              ' Wait 2 seconds
          goto loop                               ' Do it forever.

conv:     low CLK                                 ' Put clock line in starting state.
          low CS                                  ' Select ADC.
          pulsout CLK, 1                          ' 10 us clock pulse.
          let data = 0                            ' Clear data.
          for i = 1 to 8                          ' Eight data bits.
          let data = data * 2                     ' Perform shift left.
          pulsout CLK, 1                          ' 10 us clock pulse.
          let data = data + AD                    ' Put bit in LSB of data.
          next                                    ' Do it again.
          high CS                                 ' Deselect ADC when done.
          return
```

Introduction. This application note presents a program in PBASIC that enables the BASIC Stamp to read a keypad and display keypresses on a liquid-crystal display.

Background. Many controller applications require a keypad to allow the user to enter numbers and commands. The usual way to interface a keypad to a controller is to connect input/output (I/O) bits to row and column connections on the keypad. The keypad is wired in a matrix arrangement so that when a key is pressed one row is shorted to one column. It's relatively easy to write a routine to scan the keypad, detect keypresses, and determine which key was pressed.

The trouble is that a 16-key pad requires a minimum of eight bits (four rows and four columns) to implement this approach. For the BASIC Stamp, with a total of only eight I/O lines, this may not be feasible, even with clever multiplexing arrangements. And although the programming to scan a keypad is relatively simple, it can cut into the Stamp's 255 bytes of program memory.

An alternative that conserves both I/O bits and program space is to use the 74C922 keypad decoder chip. This device accepts input from a 16-key pad, performs all of the required scanning and debouncing, and



Schematic to accompany program KEYPAD.BAS.

outputs a "data available" bit and 4 output bits representing the number of the key pressed from 0 to 15. A companion device, the 74C923, has the same features, but reads a 20-key pad and outputs 5 data bits.

Application. The circuit shown in the figure interfaces a keypad and liquid-crystal display (LCD) module to the BASIC Stamp, leaving two I/O lines free for other purposes, such as bidirectional serial communication. As programmed, this application accepts keystrokes from 16 keys and displays them in hexadecimal format on the LCD.

When the user presses a button on the keypad, the corresponding hex character appears on the display. When the user has filled the display with 16 characters, the program clears the screen.

The circuit makes good use of the electrical properties of the Stamp, the LCD module, and the 74C922. When the Stamp is addressing the LCD, the 10k resistors prevent keypad activity from registering. The Stamp can easily drive its output lines high or low regardless of the status of these lines. When the Stamp is not addressing the LCD, its lines are configured as inputs, and the LCD's lines are in a high-impedance state (tri-stated). The Stamp can then receive input from the keypad without interference.

The program uses the button instruction to read the data-available line of the 74C922. The debounce feature of button is unnecessary in this application because the 74C922 debounces its inputs in hardware; however, button provides a professional touch by enabling delayed auto-repeat for the keys.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' PROGRAM: Keypad.bas
' The Stamp accepts input from a 16-key matrix keypad with the help of
' a 74C922 keypad decoder chip.
Symbol   E       =    5                        ' Enable pin, 1 = enabled
Symbol   RS      =    4                        ' Register select pin, 0 = instruction
Symbol   char    =   b1                     ' Character sent to LCD.
Symbol   buttn   =   b3                     ' Workspace for button command.
Symbol   lngth   =   b5                     ' Length of text appearing on LCD.
Symbol   temp    =   b7                     ' Temporary holder for input character.

' Set up the Stamp's I/O lines and initialize the LCD.
begin:   let pins = 0                       ' Clear the output lines
         let dirs = %01111111               ' One input, 7 outputs.
         pause 200                          ' Wait 200 ms for LCD to reset.
         let buttn = 0
         let lngth = 0
         gosub i_LCD
         gosub clear

keyin:   let dirs = %01100000               ' Set up I/O directions.
loop:    button 4,1,50,10,buttn,0,nokey     ' Check pin 4 (data available) for
                                            ' keypress.
         lngth = lngth + 1                  ' Key pressed: increment position
counter.
         let temp = pins & %00001111        ' Strip extra bits to leave only key data.
         if temp > 9 then hihex             ' Convert 10 thru 15 into A thru F (hex).
         let temp = temp + 48               ' Add offset for ASCII 0.
LCD:     let dirs = %01111111               ' Get ready to output to LCD.
         if lngth > 16 then c_LCD           ' Screen full? Clear it.
cont:    let char = temp                    ' Write character to LCD.
         gosub wr_LCD
nokey:   pause 10                           ' Short delay for nice auto-repeat
                                            ' speed.
         goto keyin                         ' Get ready for next key.

hihex:  let temp = temp + 55                ' Convert numbers 10 to 15 into A - F.
         goto LCD
c_LCD:   let lngth = 1                      ' If 16 characters are showing on LCD,
         gosub clear                        ' clear the screen and print at left edge.
         goto cont

' Initialize the LCD in accordance with Hitachi's instructions
' for 4-bit interface.
i_LCD:   let pins = %00000011               ' Set to 8-bit operation.
         pulsout E,1                        ' Send above data three times
         pause 10                           ' to initialize LCD.
         pulsout E,1
         pulsout E,1
         let pins = %00000010               ' Set to 4-bit operation.
         pulsout E,1                        ' Send above data three times.
         pulsout E,1
         pulsout E,1
         let char = 12                      ' Set up LCD in accordance w/
```

```
          gosub wr_LCD                    ' Hitachi instruction manual.
          let char = 6                    ' Turn off cursor, enable
          gosub wr_LCD                    ' left-to-right printing.
          high RS                         ' Prepare to send characters.
          return

' Write the ASCII character in b3 to the LCD.
wr_LCD:   let pins = pins & %00010000
          let b2 = char/16                ' Put high nibble of b3 into b2.
          let pins = pins | b2            ' OR the contents of b2 into pins.
          pulsout E,1                     ' Blip enable pin.
          let b2 = char & %00001111       ' Put low nibble of b3 into b2.
          let pins = pins & %00010000     ' Clear 4-bit data bus.
          let pins = pins | b2            ' OR the contents of b2 into pins.
          pulsout E,1                     ' Blip enable.
          return

' Clear the LCD screen.
clear:    low RS                          ' Change to instruction register.
          let char = 1                    ' Clear display.
          gosub wr_LCD                    ' Write instruction to LCD.
          high RS                         ' Put RS back in character mode.
          return
```

Introduction. This application note presents a program in PBASIC that enables the BASIC Stamp to control pulse-width proportional servos and measure the pulse width of other servo drivers.

Background. Servos of the sort used in radio-controlled airplanes are finding new applications in home and industrial automation, movie and theme-park special effects, and test equipment. They simplify the job of moving objects in the real world by eliminating much of the mechanical design. For a given signal input, you get a predictable amount of motion as an output.

Figure 1 shows a typical servo. The three wires are +5 volts, ground, and signal. The output shaft accepts a wide variety of prefabricated disks and levers. It is driven by a geared-down motor and rotates through 90 to 180 degrees. Most servos can rotate 90 degrees in less than a half second. Torque, a measure of the servo's ability to overcome mechanical resistance (or lift weight, pull springs, push levers, etc.), ranges from 20 to more than 100 inch-ounces.

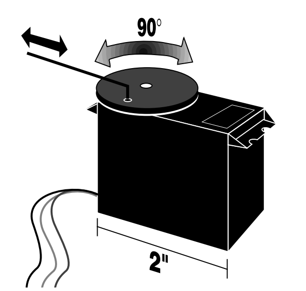


Figure 1. A typical servo.

To make a servo move, connect it to a 5-volt power supply capable of delivering an ampere or more of peak current, and supply a positioning

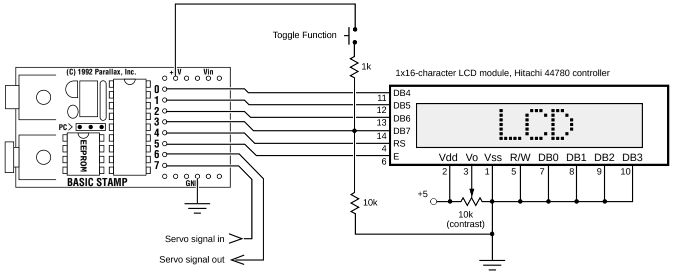


Figure 2. Schematic to accompany program SERVO.BAS.

signal. The signal is generally a 5-volt, positive-going pulse between 1 and 2 milliseconds (ms) long, repeated about 50 times per second. The width of the pulse determines the position of the servo. Since servos' travel can vary, there isn't a definite correspondence between a given pulse width and a particular servo angle, but most servos will move to the center of their travel when receiving 1.5-ms pulses.

Servos are closed-loop devices. This means that they are constantly comparing their commanded position (proportional to the pulse width) to their actual position (proportional to the resistance of a potentiometer mechanically linked to the shaft). If there is more than a small difference between the two, the servo's electronics will turn on the motor to eliminate the error. In addition to moving in response to changing input signals, this active error correction means that servos will resist mechanical forces that try to move them away from a commanded position. When the servo is unpowered or not receiving positioning pulses, you can easily turn the output shaft by hand. When the servo is powered and receiving signals, it won't budge from its position.

Application. Driving servos with the BASIC Stamp is simplicity itself. The instruction pulsout pin, time generates a pulse in 10-microsecond (µs) units, so the following code fragment would command a servo to its centered position and hold it there:

```
servo:    pulsout 0,150
          pause 20
          goto servo
```

The 20-ms pause ensures that the program sends the pulse at the standard 50 pulse-per-second rate.

The program listing is a diagnostic tool for working with servos. It has two modes, pulse measurement and pulse generation. Given an input servo signal, such as from a radio-control transmitter/receiver, it displays the pulse width on a liquid-crystal display (LCD). A display of "Pulse Width: 150" indicates a 1.5-ms pulse. Push the button to toggle functions, and the circuit supplies a signal that cycles between 1 and 2 ms. Both the pulse input and output functions are limited to a resolution of 10µs. For most servos, this equates to a resolution of better than 1 degree of rotation.

The program is straightforward Stamp BASIC, but it does take advantage of a couple of the language's handy features. The first of these is the EEPROM directive. EEPROM address,data allows you to stuff tables of data or text strings into EEPROM memory. This takes no additional program time, and only uses the amount of storage required for the data. After the symbols, the first thing that the listing does is tuck a couple of text strings into the bottom of the EEPROM. When the program later needs to display status messages, it loads the text strings from EEPROM.

The other feature of the Stamp's BASIC that the program exploits is the ability to use compound expressions in a let assignment. The routine BCD (for binary-coded decimal) converts one byte of data into three ASCII characters representing values from 0 (represented as "000") to 255.

To do this, BCD performs a series of divisions on the byte and on the remainders of divisions. For example, when it has established how many hundreds are in the byte value, it adds 48, the ASCII offset for zero. Take a look at the listing. The division (/) and remainder (//) calculations happen before 48 is added. Unlike larger BASICs which have a precedence of operators (e.g., multiplication is always before addition), the Stamp does its math from left to right. You cannot use parentheses to alter the order, either.

If you're unsure of the outcome of a calculation , use the debug directive to look at a trial run, like so:

```
let BCDin = 200
let huns = BCDin/100+48
debug huns
```

When you download the program to the Stamp, a window will appear on your computer screen showing the value assigned to the variable huns (50). If you change the second line to let huns = 48+BCDin/100, you'll get a very different result (2).

By the way, you don't have to use let, but it will earn you Brownie points with serious computer-science types. Most languages other than BASIC make a clear distinction between equals as in huns = BCDin/100+48 and if BCDin = 100 then...

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' PROGRAM: Servo.bas
' The Stamp works as a servo test bench. It provides a cycling servo signal
' for testing, and measures the pulse width of external servo signals.

Symbol   E       =    5                   ' Enable pin, 1 = enabled
Symbol   RS      =    4                   ' Register select pin, 0 = instruction
Symbol   char    =    b0                  ' Character sent to LCD.
Symbol   huns    =    b3                  ' BCD hundreds
Symbol   tens    =    b6                  ' BCD tens
Symbol   ones    =    b7                  ' BCD ones
Symbol   BCDin   =    b8                  ' Input to BCD conversion/display
routine.
Symbol   buttn   =    b9                  ' Button workspace
Symbol   i       =    b10                 ' Index counter

' Load text strings into EEPROM at address 0. These will be used to display
' status messages on the LCD screen.
EEPROM 0,("Cycling... Pulse Width: ")

' Set up the Stamp's I/O lines and initialize the  LCD.
begin:   let pins = 0                     ' Clear the output lines
         let dirs = %01111111             ' One input, 7 outputs.
         pause 200                        ' Wait 200 ms for LCD to reset.

' Initialize the LCD in accordance with Hitachi's instructions
' for 4-bit interface.
i_LCD:   let pins = %00000011             ' Set to 8-bit operation.
         pulsout E,1                      ' Send above data three times
         pause 10                         ' to initialize LCD.
         pulsout E,1
         pulsout E,1
         let pins = %00000010             ' Set to 4-bit operation.
         pulsout E,1                      ' Send above data three times.
         pulsout E,1
         pulsout E,1
         let char = 12                    ' Set up LCD in accordance w/
```

```
          gosub wr_LCD                  ' Hitachi instruction manual.
          let char = 6                  ' Turn off cursor, enable
          gosub wr_LCD                  ' left-to-right printing.
          high RS                       ' Prepare to send characters.

' Measure the width of input pulses and display on the LCD.
mPulse:   output 3
          gosub clear                   ' Clear the display.
          for i = 11 to 23              ' Read "Pulse Width:" label
            read i, char
            gosub wr_LCD                ' Print to display
          next
          pulsin 7, 1, BCDin            ' Get pulse width in 10 us units.
          gosub BCD                     ' Convert to BCD and display.
          pause 500
          input 3                       ' Check button; cycle if down.
          button 3,1,255,10,buttn,1,cycle
          goto mPulse                   ' Otherwise, continue measuring.

' Write the ASCII character in b3 to LCD.
wr_LCD:   let pins = pins & %00010000
          let b2 = char/16              ' Put high nibble of b3 into b2.
          let pins = pins | b2          ' OR the contents of b2 into pins.
          pulsout E,1                   ' Blip enable pin.
          let b2 = char & %00001111     ' Put low nibble of b3 into b2.
          let pins = pins & %00010000   ' Clear 4-bit data bus.
          let pins = pins | b2          ' OR the contents of b2 into pins.
          pulsout E,1                   ' Blip enable.
          return

clear:    low RS                        ' Change to instruction register.
          let char = 1                  ' Clear display.
          gosub wr_LCD                  ' Write instruction to LCD.
          high RS                       ' Put RS back in character mode.
          return

' Convert a byte into three ASCII digits and display them on the LCD.
' ASCII 48 is zero, so the routine adds 48 to each digit for display on the LCD.
BCD:      let huns= BCDin/100+48        ' How many hundreds?
          let tens= BCDin//100          ' Remainder of #/100 = tens+ones.
          let ones= tens//10+48         ' Remainder of (tens+ones)/10 = ones.
          let tens= tens/10+48          ' How many tens?
          let char= huns                ' Display three calculated digits.
          gosub wr_LCD
          let char = tens
          gosub wr_LCD
          let char = ones
          gosub wr_LCD
          return
```

```
' Cycle the servo back and forth between 0 and 90 degrees. Servo moves slowly ' in
one direction (because of 20-ms delay between changes in pulse width) and quickly
' in the other. Helps diagnose stuck servos, dirty feedback pots, etc.
cycle:    output 3
          gosub clear
          for i = 0 to 9                      ' Get "Cycling..." string and
            read i, char                      ' display it on LCD.
            gosub wr_LCD
          next i
reseti:   let i = 100                         ' 1 ms pulse width.
cyloop:   pulsout 6,i                         ' Send servo pulse.
          pause 20                            ' Wait 1/50th second.
          let i = i + 2                       ' Move servo.
          if i > 200 then reseti              ' Swing servo back to start position.
          input 3                             ' Check the button; change function if
                                              ' down.
          button 3,1,255,10,buttn,1,mPulse
          goto cyloop                         ' Otherwise, keep cycling.
```

Introduction. This application note explores several applications for the BASIC Stamp's unique pulsin command, which measures the duration of incoming positive or negative pulses in 10-microsecond units.

Background. The BASIC Stamp’s pulsin command measures the width of a pulse, or the interval between two pulses. Left at that, it might seem to have a limited range of obscure uses. However, pulsin is the key to many kinds of real-world interfacing using simple, reliable sensors. Some possibilities include:

tachometer
speed trap
physics demonstrator
capacitance checker
duty cycle meter
log input analog-to-digital converter

Pulsin works like a stopwatch that keeps time in units of 10 microseconds (µs). To use it, you must specify which pin to monitor, when to trigger on (which implies when to trigger off), and where to put the resulting 16-bit time measurement. The syntax is as follows:

pulsin *pin, trigger condition, variable*

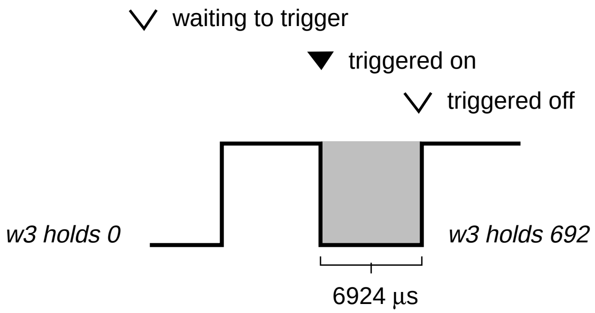


Figure 1. Timing diagram for `pulsin 7,0,w3`.

Pin is a BASIC Stamp input/output pin (0 to 7). Trigger condition is a variable or constant (0 or 1) that specifies the direction of the transition that will start the pulsin timer. If trigger is 0, pulsin will start measuring when a high-to-low transition occurs, because 0 is the edge's destination. Variable can be either a byte or word variable to hold the timing measurement. In most cases, a word variable is called for, because pulsin produces 16-bit results.

Figure 1 shows how pulsin works. The waveform represents an input at pin 7 that varies between ground and +5 volts (V).

A smart feature of pulsin is its ability to recognize a no-pulse or out-of-range condition. If the specified transition doesn't occur within 0.65535 seconds (s), or if the pulse to be measured is longer than 0.65535 s, pulsin will give up and return a 0 in the variable. This prevents the program from hanging up when there's no input or out-of-range input.

Let's look at some sample applications for pulsin, starting with one inspired by the digital readout on an exercise bicycle: pulsin as a tachometer.

Tachometer. The most obvious way to measure the speed of a wheel or shaft in revolutions per minute (rpm) is to count the number of

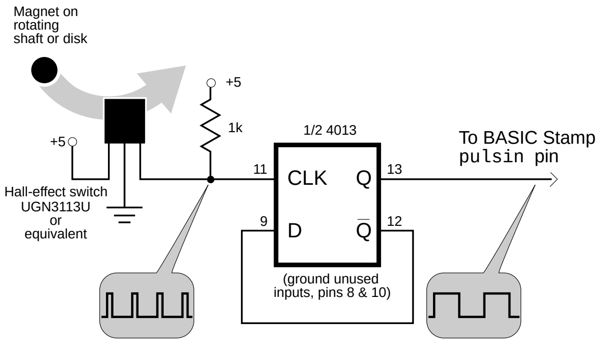

Magnet on rotating shaft or disk
+5
1k
+5
1/2 4013
To BASIC Stamp
pulsin pin
11
CLK
Q
13
9
D
Q̅
12
Hall-effect switch UGN3113U or equivalent
(ground unused inputs, pins 8 & 10)

Figure 2. Schematic to accompany listing 1, TACH.BAS.

revolutions that occur during 1 minute. The trouble is, the user probably wouldn't want to wait a whole minute for the answer.

For a continuously updated display, we can use pulsin to measure the time the wheel takes to make one complete revolution. By dividing this time into 60 seconds, we get a quick estimate of the rpm. Listing 1 is a tachometer program that works just this way. Figure 2 is the circuit that provides input pulses for the program. A pencil-eraser-sized magnet attached to the wheel causes a Hall-effect switch to generate a pulse every rotation.

We could use the Hall switch output directly, by measuring the interval between positive pulses, but we would be measuring the period of rotation minus the pulses. That would cause small errors that would be most significant at high speeds. The flip-flop, wired to toggle with each pulse, eliminates the error by converting the pulses into a train of square waves. Measuring either the high or low interval will give you the period of rotation.

Note that listing 1 splits the job of dividing the period into 60 seconds into two parts. This is because 60 seconds expressed in 10-µs units is 6 million, which exceeds the range of the Stamp's 16-bit calculations. You will see this trick, and others that work around the limits of 16-bit math, throughout the listings.

Using the flip-flop's set/reset inputs, this circuit and program could easily be modified to create a variety of speed-trap instruments. A steel ball rolling down a track would encounter two pairs of contacts to set and reset the flip-flop. Pulsin would measure the interval and compute the speed for a physics demonstration (acceleration). More challenging setups would be required to time baseballs, remote-control cars or aircraft, bullets, or model rockets.

The circuit could also serve as a rudimentary frequency meter. Just divide the period into 1 second instead of 1 minute.

Duty cycle meter. Many electronic devices vary the power they deliver to a load by changing the duty cycle of a waveform; the proportion of time that the load is switched fully on to the time it is fully off. This approach, found in light dimmers, power supplies, motor controls and amplifiers, is efficient and relatively easy to implement with digital components. Listing 2 measures the duty cycle of a repetitive pulse train by computing the ratio of two pulsin readings and presenting them as a percentage. A reading approaching 100 percent means that the input is mostly on or high. The output of figure 2's flip-flop is 50 percent. The output of the Hall switch in figure 2 was less than 10 percent when the device was monitoring a benchtop drill press.

Capacitor checker. The simple circuit in figure 3 charges a capacitor, and then discharges it across a resistance when the button is pushed. This produces a brief pulse for pulsin to measure. Since the time constant of the pulse is determined by resistance (R) times capacitance (C), and R is fixed at 10k, the width of the pulse tells us C. With the resistance values listed, the circuit operates over a range of .001 to 2.2µF. You may substitute other resistors for other ranges of capacitance; just be sure that the charging resistor (100k in this case) is about 10 times the value of the discharge resistor. This ensures that the voltage at the junction of the two resistors when the switch is held down is a definite low (0) input to the Stamp.

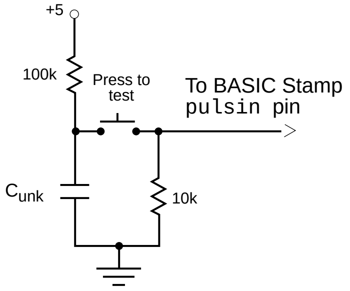


Figure 3. Schematic for listing 3, CAP.BAS.

Log-input analog-to-digital converter (ADC). Many sensors have convenient linear outputs. If you know that an input of 10 units (degrees, pounds, percent humidity, or whatever) produces an output of 1 volt, then 20 units will produce 2 volts. Others, such as thermistors and audio-taper potentiometers, produce logarithmic outputs. A Radio Shack thermistor (271-110) has a resistance of 18k at 10° C and 12k at 20°C. Not linear, and not even the worst cases!

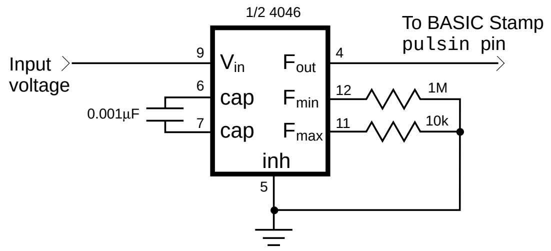


Figure 4. Schematic for listing 4, VCO.BAS.

While it’s possible to straighten out a log curve in software, it’s often easier to deal with it in hardware. That’s where figure 4 comes in. The voltage-controlled oscillator of the 4046 phase-locked loop chip, when

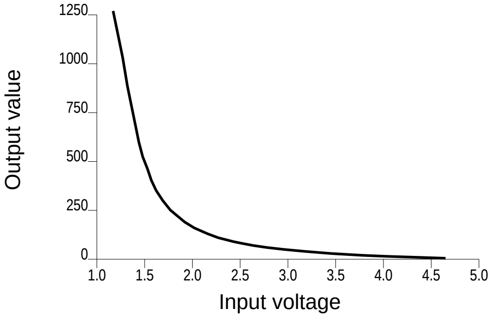


Figure 5. Log response curve of the VCO.

wired as shown, has a log response curve. If you play this curve against a log input, you can effectively straighten the curve. Figure 5 is a plot of the output of the circuit as measured by the pulsin program in listing 4. It shows the characteristic log curve.

The plot points out another advantage of using a voltage-controlled oscillator as an ADC; namely, increased resolution. Most inexpensive ADCs provide eight bits of resolution (0 to 255), while the VCO provides the equivalent of 10 bits (0 to 1024+). Admittedly, a true ADC would provide much better accuracy, but you can't touch one for anywhere near the 4046's sub-$1 price.

The 4046 isn't the only game in town, either. Devices that can convert analog values, such as voltage or resistance, to frequency or pulse width include timers (such as the 555) and true voltage-to-frequency converters (such as the 9400). For sensors that convert some physical property such as humidity or proximity into a variable capacitance or inductance, pulsin is a natural candidate for sampling their output via an oscillator or timer.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

A Note about the Program Listings

All of the listings output results as serial data. To receive it, connect Stamp pin 0 to your PC's serial input, and Stamp ground to signal ground. On 9-pin connectors, pin 2 is serial in and pin 5 is signal ground; on 25-pin connectors, pin 3 is serial in and pin 7 is signal ground. Set terminal software for 8 data bits, no parity, 1 stop bit.

```
' Listing 1: TACH.BAS
' The BASIC Stamp serves as a tachometer. It accepts pulse input through pin 7,
' and outputs rpm measurements at 2400 baud through pin 0.
        input 7
        output 0
Tach:   pulsin 7,1,w2 ' Read positive-going pulses on pin 7.
        let w2 = w2/100                 ' Dividing w2/100 into 60,000 is the
                                        ' same as dividing
        let w2 = 60000/w2               ' w2 into 6,000,000 (60 seconds in 10
                                        ' us units).
' Transmit data followed by carriage return and linefeed.
         serout 0,N2400,(#w2," rpm",10,13)
         pause 1000                          ' Wait 1 second between readings
         goto Tach
```

```
' Listing 2: DUTY.BAS
' The BASIC Stamp calculates the duty cycle of a repetitive pulse train.
' Pulses in on pin 7; data out via 2400-baud serial on pin 0.
         input 7
         output 0
Duty:    pulsin 7,1,w2                       ' Take positive pulse sample.
         if w2 > 6553 then Error             ' Avoid overflow when w2 is multiplied
by 10.
         pulsin 7,0,w3                       ' Take negative pulse sample.
         let w3 = w2+w3
         let w3 = w3/10                      ' Distribute multiplication by 10 into two
         let w2 = w2*10                      ' parts to avoid an overflow.
         let w2 = w2/w3                      ' Calculate percentage.
         serout 0,N2400,(#w2," percent",10,13)
         pause 1000                          ' Update once a second.
         goto Duty

' Handle overflows by skipping calculations and telling the user.
Error:   serout 0,N2400,("Out of range",10,13)
         pause 1000
         goto Duty
```

```
' Listing 3: CAP.BAS
' The BASIC Stamp estimates the value of a capacitor by the time required for it to
' discharge through a known resistance.
         input 7
         output 0
Cap:     pulsin 7,1,w1
         if w1 = 0 then Cap                  ' If no pulse, try again.
         if w1 > 6553 then Err               ' Avoid overflows.
         let w1 = w1*10
         let w1 = w1/14                      ' Apply calibration value.
         if w1 > 999 then uF                 ' Use uF for larger caps.
         serout 0,N2400,(#w1," nF",10,13)
         goto Cap

uF:      let b4 = w1/1000                    ' Value left of decimal point.
         let b6 = w1//1000                   ' Value right of decimal point.
         serout 0,N2400,(#b4,".",#b6," uF",10,13)
         goto Cap
```

```
Err:    serout 0,N2400,("out of range",10,13)
        goto Cap
```

```
' Listing 4: VCO.BAS
' The BASIC Stamp uses input from the VCO of a 4046 phase-locked loop as a
logarithmic
' A-to-D converter. Input on pin 7; 2400-baud serial output on pin 0.
        input 7
        output 0
VCO:    pulsin 7,1,w2 ' Put the width of pulse on pin 7 into w2.
        let w2 = w2-45                  ' Allow a near-zero minimum value
                                        ' without underflow.
        serout 0,N2400,(#w2,10,13)
        pause 1000                      ' Wait 1 second between measure-
                                        ' ments.
        goto VCO
```

Introduction. This application note demonstrates simple hardware and software techniques for driving and controlling common four-coil stepper motors.

Background. Stepper motors translate digital switching sequences into motion. They are used in printers, automated machine tools, disk drives, and a variety of other applications requiring precise motions under computer control.

Unlike ordinary dc motors, which spin freely when power is applied, steppers require that their power source be continuously pulsed in specific patterns. These patterns, or step sequences, determine the speed and direction of a stepper's motion. For each pulse or step input, the stepper motor rotates a fixed angular increment; typically 1.8 or 7.5 degrees.

The fixed stepping angle gives steppers their precision. As long as the motor's maximum limits of speed or torque are not exceeded, the controlling program knows a stepper's precise position at any given time.

Steppers are driven by the interaction (attraction and repulsion) of magnetic fields. The driving magnetic field "rotates" as strategically placed coils are switched on and off. This pushes and pulls at permanent magnets arranged around the edge of a rotor that drives the output

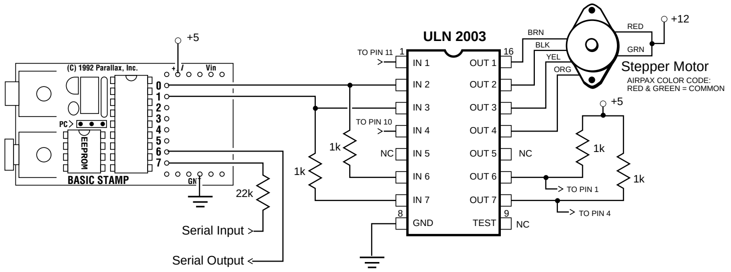
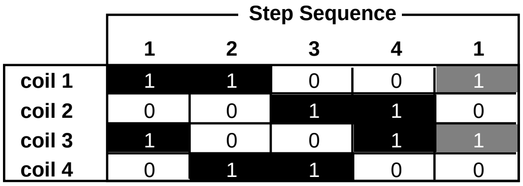


Figure 1. Schematic for the serial stepper controller.

shaft. When the on-off pattern of the magnetic fields is in the proper sequence, the stepper turns (when it's not, the stepper sits and quivers).

The most common stepper is the four-coil unipolar variety. These are called unipolar because they require only that their coils be driven on and off. Bipolar steppers require that the polarity of power to the coils be reversed.

The normal stepping sequence for four-coil unipolar steppers appears in figure 2. There are other, special-purpose stepping sequences, such as half-step and wave drive, and ways to drive steppers with multiphase analog waveforms, but this application concentrates on the normal sequence. After all, it's the sequence for which all of the manufacturer's specifications for torque, step angle, and speed apply.



	Step Sequence				
	1	2	3	4	1
coil 1	1	1	0	0	1
coil 2	0	0	1	1	0
coil 3	1	0	0	1	1
coil 4	0	1	1	0	0

Figure 2. Normal stepping sequence.

If you run the stepping sequence in figure 2 forward, the stepper rotates clockwise; run it backward, and the stepper rotates counterclockwise. The motor's speed depends on how fast the controller runs through the step sequence. At any time the controller can stop in mid sequence. If it leaves power to any pair of energized coils on, the motor is locked in place by their magnetic fields. This points out another stepper motor benefit: built-in brakes.

Many microprocessor stepper drivers use four output bits to generate the stepping sequence. Each bit drives a power transistor that switches on the appropriate stepper coil. The stepping sequence is stored in a lookup table and read out to the bits as required.

This design takes a slightly different approach. First, it uses only two output bits, exploiting the fact that the states of coils 1 and 4 are always the inverse of coils 2 and 3. Look at figure 2 again. Whenever coil 2 gets a 1, coil 1 gets a 0, and the same holds for coils 3 and 4. In Stamp designs, output bits are too precious to waste as simple inverters, so we give that job to two sections of the ULN2003 inverter/driver.

The second difference between this and other stepper driver designs is that it calculates the stepping sequence, rather than reading it out of a table. While it's very easy to create tables with the Stamp, the calculations required to create the two-bit sequence required are very simple. And reversing the motor is easier, since it requires only a single additional program step. See the listing.

How it works. The stepper controller accepts commands from a terminal or PC via a 2400-baud serial connection. When power is first applied to the Stamp, it sends a prompt to be displayed on the terminal screen. The user types a string representing the direction (+ for forward, – for backward), number of steps, and step delay (in milliseconds), like this:

```
step>+500 20
```

As soon as the user presses enter, return, or any non-numerical character at the end of the line, the Stamp starts the motor running. When the stepping sequence is over, the Stamp sends a new step> prompt to the terminal. The sample command above would take about 10 seconds (500 x 20 milliseconds). Commands entered before the prompt reappears are ignored.

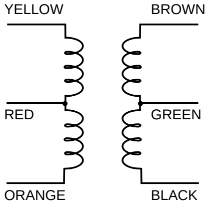


Figure 3. Color code for Airpax steppers.

On the hardware side, the application accepts any stepper that draws 500 mA or less per coil. The schematic shows the color code for an Airpax-brand stepper, but there is no standardization among different brands. If you use another stepper, use figure 3 and an ohmmeter to translate the color code. Connect the stepper and give it a try. If it vibrates instead of turning, you have one or more coils connected incorrectly. Patience and a little experimentation will prevail.

```
' Program STEP.BAS
' The Stamp accepts simply formatted commands and drives a four-coil stepper.
Commands
' are formatted as follows: +500 20<return> means rotate forward 500 steps with 20
' milliseconds between steps. To run the stepper backward, substitute - for +.

Symbol    Directn = b0
Symbol    Steps = w1
Symbol    i = w2
Symbol    Delay = b6
Symbol    Dir_cmd = b7

          dirs = %01000011 : pins = %00000001 ' Initialize output.
          b1 = %00000001 : Directn = "+"
          goto Prompt                         ' Display prompt.

' Accept a command string consisting of direction (+/-), a 16-bit number
' of steps, and an 8-bit delay (milliseconds) between steps. If longer
' step delays are required, just command 1 step at a time with long
' delays between commands.

Cmd:      serin 7,N2400,Dir_cmd,#Steps,#Delay ' Get orders from terminal.
          if Dir_cmd = Directn then Stepit    ' Same direction? Begin.
          b1 = b1^%00000011
' Else reverse (invert b1).

Stepit:   for i = 1 to Steps
' Number of steps.
          pins = pins^b1
' XOR output with b1, then invert b1
          b1 = b1^%00000011
' to calculate the stepping sequence.
          pause Delay                         ' Wait commanded delay between
                                              ' steps.
          next
          Directn = Dir_cmd
' Direction = new direction.

Prompt:   serout 6,N2400,(10,13,"step> ")     ' Show prompt, send return
          goto Cmd                            ' and linefeed to terminal.
```

Program listing: *As with the other application notes, this program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.*

Introduction. This application note shows how to measure temperature using an inexpensive thermistor and the BASIC Stamp's `pot` command. It also discusses a technique for correcting nonlinear data.

Background. Radio Shack offers an inexpensive and relatively precise thermistor—a component whose resistance varies with temperature. The BASIC Stamp has the built-in ability to measure resistance with the `pot` command and an external capacitor. Put them together, and your Stamp can measure the temperature, right? Not without a little math.

The thermistor's resistance decreases as the temperature increases, but this response is not linear. There is a table on the back of the thermistor package that lists the resistance at various temperatures in degrees celsius (°C). For the sake of brevity, we won't reproduce that table here, but the lefthand graph of figure 1 shows the general shape of the thermistor response curve in terms of the more familiar Fahrenheit scale (°F).

The `pot` command throws us a curve of its own, as shown in figure 1 (right). Though not as pronounced as the thermistor curve, it must be figured into our temperature calculations in order for the results to be usable.

One possibility for correcting the combined curves of the thermistor and `pot` command would be to create a lookup table in the Stamp's EEPROM. The table would have to be quite large to cover a reasonable temperature range at 1° precision. An alternative would be to create a smaller table at 10° precision, and figure where a particular reading

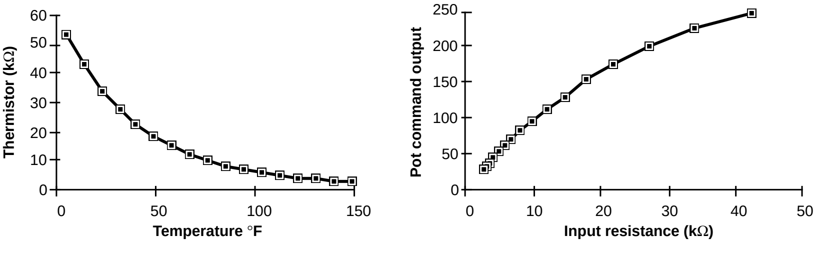


Figure 1. Response curves of the thermistor and `pot` *command.*

might lie within its 10° range. This is *interpolation*, and it can work quite well. It would still use quite a bit of the Stamp's limited EEPROM space, though.

Another approach, the one used in the listing, is to use a power-series polynomial to model the relationship between the `pot` reading and temperature. This is easier than it sounds, and can be applied to many nonlinear relationships.

Step 1: Prepare a table. The first step is to create a table of a dozen or so inputs and outputs. The inputs are resistances and outputs are temperatures in °F. Resistance values in this case are numbers returned by the `pot` function. To equate `pot` values with temperatures, we connected a 50k pot and a 0.01 µF capacitor to the Stamp and performed the calibration described in the Stamp manual. After obtaining a scale factor, we pressed the space bar to lock it in.

Now we could watch the pot value change as the potentiometer was adjusted. We disconnected the potentiometer from the Stamp and hooked it to an ohmmeter. After setting the potentiometer to 33.89k (corresponding to a thermistor at 23 °F or –5 °C), we reconnected it to the Stamp, and wrote down the resulting reading. We did this for each of the calibration values on the back of the thermistor package, up to 149 °F (65 °C).

Step 2: Determine the coefficients. The equation that can approximate our nonlinear temperature curve is:

$$\text{Temperature} = C0 + C1 \bullet (\text{Pot Val}) + C2 \bullet (\text{Pot Val})^2 + C3 \bullet (\text{Pot Val})^3$$

where C0, C1, C2, and C3 are coefficients supplied by analytical software, and each Cn • $(\text{Pot Val})^n$ is called a term. The equation above has three terms, so it is called a *third-order* equation. Each additional term increases the range over which the equation's results are accurate. You can increase or decrease the number of terms as necessary, but each additional coefficient requires that Pot Val be raised to a higher power. This can make programming messy, so it pays to limit the number of terms to the fewest that will do the job.

The software that determines the coefficients is called GAUSFIT.EXE and is available from the Parallax ftp site. To use it, create a plain text file called GF.DAT. In this file, which should be saved to the same subdirectory as GAUSFIT, list the inputs and outputs in the form *in,out<return>*. If there are values that require particular precision, they may be listed more than once. We wanted near-room-temperature values to be right on, so we listed 112,68 (`pot` value at 68 °F) several times.

To run the program, type GAUSFIT *n* where *n* is the number of terms desired. The program will compute coefficients and present you with a table showing how the computed data fits your samples. The fit will be good in the middle, and poorer at the edges. If the edges are unacceptable, you can increase the number of terms. If they are OK, try rerunning the program with fewer terms. We were able to get away with just two terms by allowing accuracy to suffer outside a range of 50 °F to 90 °F.

Step 3: Factor the coefficients. The coefficients that GAUSFIT produces are not directly useful in a BASIC Stamp program. Our coefficients were: C0 = 162.9763, C1 = –1.117476, and C2 = 0.002365991. We plugged the values into a spreadsheet and computed temperatures from `pot` values and then started playing with the coefficients. We found that the following coefficients worked almost as well as the originals: C0 = 162, C1 = –1.12, and C2 = 0.0024.

The problem that remained was how to use these values in a Stamp program. The Stamp deals in only positive integers from 0 to 65,535. The trick is to express the numbers to the right of the decimal point as fractions. For example, the decimal number 0.75 can be expressed as 3/4. So to multiply a number by 0.75 with the BASIC Stamp, first multiply the number by 3, then divide the result by 4. For less familiar decimal values, it may take some trial and error to find suitable fractions. We found that the 0.12 portion of C1 was equal to 255/2125, and that C2 (0.0024) = 3/1250.

Step 4: Plan the order of execution. Just substituting the fractions for the decimal portions of the formula still won't work. The problem is that portions of terms, such as 3•Pot Val2/1250, can exceed the 65,535 limit. If Pot Val were 244, then 3•244^2 would equal 178,608; too high.

The solution is to factor the coefficients and rearrange them into smaller problems that can be solved within the limit. For example (using PV to stand for Pot Val):

$$\frac{PV \cdot PV \cdot 3}{1250} = \frac{PV \cdot PV \cdot 3}{5 \cdot 5 \cdot 5 \cdot 5 \cdot 2} = \frac{PV}{25} \cdot \frac{PV \cdot 3}{50}$$

The program in the listing is an example of just such factoring and rearrangement. Remember to watch out for the lower limit as well. Try to keep intermediate results as high as possible within the Stamp's integer limits. This will reduce the effect of truncation errors (where any value to the right of the decimal point is lost).

Conclusion. The finished program, which reports the temperature to the PC screen via the debug command, is deceptively simple. An informal check of its output found that it tracks within 1 °F of a mercury/glass bulb thermometer in the range of 60 °F to 90 °F. Additional range could be obtained at the expense of a third-order equation; however, current performance is more than adequate for use in a household thermostat or other noncritical application. Cost and complexity are far less than that of a linear sensor, precision voltage reference, and analog-to-digital converter.

If you adapt this application for your own use, component tolerances will probably produce different results. However, you can calibrate the program very easily. Connect the thermistor and a stable, close-tolerance 0.1-µF capacitor to the Stamp as shown in figure 2. Run the program and note the value that appears in the `debug` window. Compare it to a known accurate thermometer located close to the thermistor. If the thermometer says 75 and the Stamp 78, reduce the value of C0 by 3. If the thermometer says 80 and the Stamp 75, increase the value of C0 by 5. This works because the relationship between the thermistor resistance and the temperature is the same, only the value of the capacitor is different. Adjusting C0 corrects this offset.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

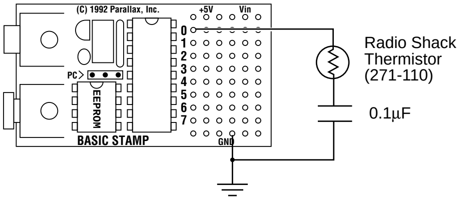


Figure 2. Schematic to accompany THERM.BAS.

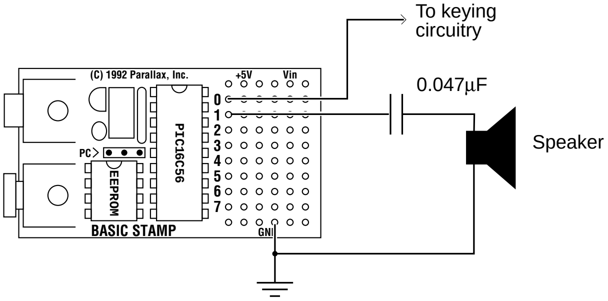
```
' Program THERM.BAS
' This program reads a thermistor with the BASIC
' pot command, computes the temperature using a
' power-series polynomial equation, and reports
' the result to a host PC via the Stamp cable
' using the debug command.

' Symbol constants represent factored portions of
' the coefficients C0, C1, and C2. "Top" and "btm"
' refer to the values' positions in the fractions;
' on top as a multiplier or on the bottom as a
' divisor.
Symbol co0    = 162
Symbol co1top = 255
Symbol co1btm = 2125
Symbol co2bt1 = 25
Symbol co2top = 3
Symbol co2btm = 50

' Program loop.
Check_temp:
        pot 0,46,w0                         ' 46 is the scale factor.

' Remember that Stamp math is computed left to
' right--no parentheses, no precedence of
' operators.
        let w1 = w0*w0/co2bt1*co2top/co2btm
        let w0 = w0*co1top/co1btm+w0
        let w0 = co0+w1-w0
        debug w0
        pause 1000                          ' Wait 1 second for next
goto Check_temp                             ' temperature reading.
```

Introduction. This application note presents a technique for using the BASIC Stamp to send short messages in Morse code. It demonstrates the Stamp's built-in `lookup` and `sound` commands.

Background. Morse code is probably the oldest serial communication protocol still in use. Despite its age, Morse has some virtues that make it a viable means of communication. Morse offers inherent compression; the letter E is transmitted in one-thirteenth the time required to send the letter Q. Morse requires very little transmitting power and bandwidth compared to other transmitting methods. And Morse may be sent and received by either human operators or automated equipment.

Although Morse has fallen from favor as a means for sending large volumes of text, it is still the legal and often preferred way to identify automated repeater stations and beacons. The BASIC Stamp, with its ease of programming and minuscule power consumption, is ideal for this purpose.

The characters of the Morse code are represented by sequences of long and short beeps known as dots and dashes (or dits and dahs). There are one to six beeps or *elements* in the characters of the standard Morse code. The first step in writing a program to send Morse is to devise a compact way to represent sequences of elements, and an efficient way to play them back.



Schematic to accompany program MORSE.BAS.

The table on the next page shows the encoding scheme used in this program. A single byte represents a Morse character. The highest five bits of the byte represent the actual dots(0s) and dashes (1s), while the lower three bits represent the number of elements in the character. For example, the letter F is dot dot dash dot, so it is encoded 0010x100, where x is a don't-care bit. Since Morse characters can contain up to six elements, we have to handle the exceptions. Fortunately, we have some excess capacity in the number-of-elements portion of the byte, which can represent numbers up to seven. So we assign a six-element character ending in a dot the number six, while a six-element character ending in a dash gets the number seven.

The program listing shows how these bytes can be played back to produce Morse code. The table of symbols at the beginning of the program contain the timing data for the dots and dashes themselves. If you want to change the program's sending speed, just enter new values for `dit_length`, `dah_length`, etc. Make sure to keep the timing

Morse Characters and their Encoded Equivalents

Char	Morse	Binary	Decimal	Char	Morse	Binary	Decimal
A	•–	01000010	66	S	•••	00000011	3
B	–•••	10000100	132	T	–	10000001	129
C	–•–•	10100100	164	U	••–	00100011	35
D	–••	10000011	131	V	•••–	00010100	20
E	•	00000001	1	W	•––	01100011	99
F	••–•	00100100	36	X	–••–	10010100	148
G	––•	11000011	195	Y	–•––	10110100	180
H	••••	00000100	4	Z	––••	11000100	196
I	••	00000010	2	0	–––––	11111101	253
J	•–––	01110100	116	1	•––––	01111101	125
K	–•–	10100011	163	2	••–––	00111101	61
L	•–••	01000100	68	3	•••––	00011101	29
M	––	11000010	194	4	••••–	00001101	13
N	–•	10000010	130	5	•••••	00000101	5
O	–––	11100011	227	6	–••••	10000101	133
P	•––•	01100100	100	7	––•••	11000101	197
Q	––•–	11010100	212	8	–––••	11100101	229
R	•–•	01000011	67	9	––––•	11110101	245

relationships roughly the same; a dash should be about three times as long as a dot.

The program uses the BASIC Stamp's `lookup` function to play sequences of Morse characters. `Lookup` is a particularly modern feature of Stamp BASIC in that it is an object-oriented data structure. It not only contains the data, it also "knows how" to retrieve it.

Modifications. The program could readily be modified to transmit messages whenever the Stamp detects particular conditions, such as "BATTERY LOW." With some additional programming and analog-to-digital hardware, it could serve as a low-rate telemetry unit readable by either automated or manual means.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program MORSE.BAS
' This program sends a short message in Morse code every
' minute. Between transmissions, the Stamp goes to sleep
' to conserve battery power.
Symbol   Tone = 100
Symbol   Quiet = 0
Symbol   Dit_length = 7                 ' Change these constants to
Symbol   Dah_length = 21                ' change speed. Maintain ratios
Symbol   Wrd_length = 42                ' 3:1 (dah:dit) and 7:1 (wrd:dit).
Symbol   Character = b0
Symbol   Index1 = b6
Symbol   Index2 = b2
Symbol   Elements = b4

Identify:
output  0: output 1
for Index1 = 0 to 7
' Send the word "PARALLAX" in Morse:
         lookup Index1,(100,66,67,66,68,68,66,148),Character
         gosub Morse
next
sleep 60
goto Identify

Morse:
```

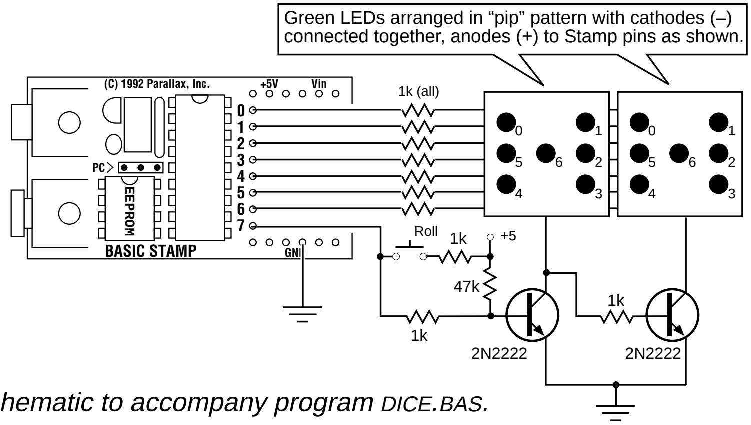
```
let Elements = Character & %00000111
if Elements = 7 then Adjust1
if Elements = 6 then Adjust2
Bang_Key:
for Index2 = 1 to Elements
            if Character >= 128 then Dah
            goto Dit
  Reenter:
            let Character = Character * 2
next
gosub char_sp
return
Adjust1:
Elements = 6
goto Bang_Key

Adjust2:
Character = Character & %11111011
goto Bang_Key
end

Dit:
high 0
sound 1,(Tone,Dit_length)
low 0
sound 1,(Quiet,Dit_length)
goto Reenter

Dah:
high 0
sound 1,(Tone,Dah_length)
low 0
sound 1,(Quiet,Dit_length)
goto Reenter

Char_sp:
sound 1,(Quiet,Dah_length)
return

Word_sp:
sound 1,(Quiet,Wrd_length)
return
```

Introduction. This application note describes an electronic dice game based on the BASIC Stamp. It shows how to connect LED displays to the Stamp, and how to multiplex inputs and outputs on a single Stamp pin.

Background. Much of BASIC's success as a programming language is probably the result of its widespread use to program games. After all, games are just simulations that happen to be fun.

How it works. The circuit for the dice game uses Stamp pins 0 through 6 to source current to the anodes of two sets of seven LEDs. Pin 7 and the switching transistors determine which set of LEDs is grounded. Whenever the lefthand LEDs are on, the right are off, and vice versa. To light up the LEDs, the Stamp puts die1's pattern on pins 0-6, and enables die1 by making pin 7 high. After a few milliseconds, it puts die2's pattern on pins 0-6 and takes pin 7 low to enable die2.

In addition to switching between the dice, pin 7 also serves as an input for the press-to-roll pushbutton. The program changes the pin to an input and checks its state. If the switch is up, a low appears on pin 7 because the base-emitter junction of the transistor pulls it down to about 0.7 volts. If the switch is pressed, a high appears on pin 7. The 1k resistor puts a high on pin 7 when it is an input, but pin 7 is still able to pull the base of the transistor low when it is an output. As a result, holding the switch down doesn't affect the Stamp's ability to drive the display.


Green LEDs arranged in “pip” pattern with cathodes (–) connected together, anodes (+) to Stamp pins as shown.

Schematic to accompany program DICE.BAS.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program DICE.BAS
' An electonic dice game that uses two sets of seven LEDs
' to represent the pips on a pair of dice.

Symbol  die1 = b0                       ' Store number (1-6) for first die.
Symbol  die2 = b1                       ' Store number (1-6) for ssecond die.
Symbol  shake = w3                      ' Random word variable
Symbol  pippat = b2                     ' Pattern of "pips" (dots) on dice.
Symbol  Select = 7                      ' Pin number of select transistors.

high Select
let dirs = 255                          ' All pins initially outputs.
let die1 = 1                            ' Set lucky starting value for dice (7).
let die2 = 4                            ' (Face value of dice = die1+1, die2+1.)

Repeat:                                 ' Main program loop.
let pippat = die1
gosub Display                           ' Display die 1 pattern.
let pippat = die2                       ' Now die 2.
gosub Display
input Select                            ' Change pin 7 to input.
if pin7 = 1 then Roll                   ' Switch closed? Roll the dice.
let w3 = w3+1                           ' Else stir w3.
Reenter:                                ' Return from Roll subroutine.
output Select                           ' Restore pin 7 to output.
goto Repeat

Display:                                ' Look up pip pattern.
lookup pippat,(64,18,82,27,91,63),pippat
let pins = pins&%10000000
toggle Select                           ' Invert Select.
let pins = pins|pippat                  ' OR pattern into pins.
pause 4                                 ' Leave on 4 milliseconds.
return

Roll:
random shake                            ' Get random number.
let die1 = b6&%00000111                 ' Use lower 3 bits of each byte.
let die2 = b7&%00000111
if die1 > 5 then Roll                   ' Throw back numbers over 5 (dice>6).
if die2 > 5 then Roll
goto Reenter                            ' Back to the main loop.
```

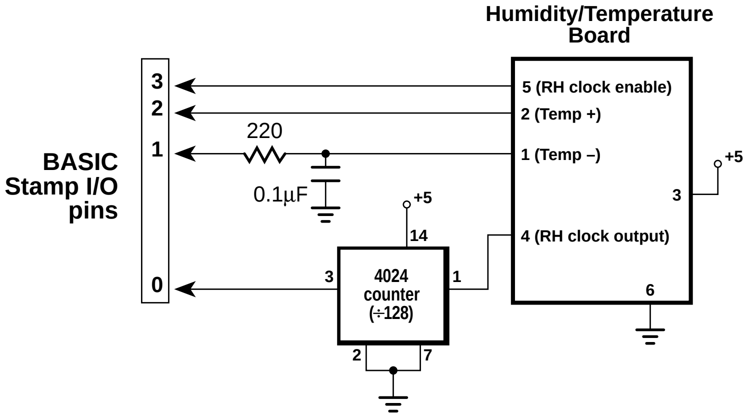
Introduction. This application note shows how to interface an inexpensive humidity / temperature sensor kit to the Stamp.

Background. When it's hot, high humidity makes it seem hotter. When it's cold, low humidity makes it seem colder. In areas where electronic components are handled, low humidity increases the risk of electrostatic discharge (ESD) and damage. The relationship between temperature and humidity is a good indication of the efficiency of heavy-duty air-conditioning equipment that uses evaporative cooling.

Despite the value of knowing temperature and humidity, it can be hard to find suitable humidity sensors. This application solves that problem by borrowing a sensor kit manufactured for computerized home weather stations.

The kit, available from the source listed at the end of this application note for $25, consists of fewer than a dozen components and a small (0.5" x 2.75") printed circuit board. Assembly entails soldering the components to the board. When it's done, you have two sensors: a temperature-dependent current source and a humidity-dependent oscillator.

Once the sensor board is complete, connect it to the Stamp using the circuit shown in the figure and download the software in the listing. The



Schematic to accompany program HUMID.BAS.

debug window will appear on your PC screen showing values representing humidity and temperature. To get a feel for the board's sensitivity, try this: Breathe on the sensor board and watch the debug values change. The humidity value should increase dramatically, while the temperature number (which decreases as the temperature goes up) will fall a few counts.

How it works. The largest portion of the program is devoted to measuring the temperature, so we'll start there. The temperature sensor is an LM334Z constant-current source. Current through the device varies at the rate of 0.1µA per 1° C change in temperature. The program in the listing passes current from pin 2 of the Stamp through the sensor to a capacitor for a short period of time, starting with 5000 µs. It then checks the capacitor's state of charge through pin 1. If the capacitor is not charged enough for pin 1 to see a logical 1, the Stamp discharges the capacitor and tries again, with a slightly wider pulse of 5010 µs.

It stays in a loop, charging, checking, discharging, and increasing the charging pulse until the capacitor shows as a 1 on pin 1's input. Since the rate of charge is proportional to current, and the current is proportional to temperature, the width of the pulse that charges the capacitor is a relative indication of temperature.

Sensing humidity is easier, thanks to the design of the kit's hardware. The humidity sensor is a capacitor whose value changes with relative humidity (RH). At a relative humidity of 43 percent and a temperature of 77° F, the sensor has a value of 122 pF ± 15 percent. Its value changes at a rate of 0.4 pF ± 0.05 pF for each 1-percent change in RH.

The sensor controls the period of a 555 timer wired as a clock oscillator. The clock period varies from 225 µs at an arid 10-percent RH to 295 µs at a muggy 90-percent RH. Since we're measuring this change with the Stamp's pulsin command, which has a resolution of 10 µs, we need to exaggerate those changes in period in order to get a usable change in output value. That's the purpose of the 4024 counter.

We normally think of a counter as a frequency divider, but by definition it's also a period multiplier. By dividing the clock output by 128, we create a square wave with a period 128 times as long. Now humidity is represented by a period ranging from 28.8 to 37.8 milliseconds. Since `pulsin` measures only half of the waveform, the time that it's high, RH values range from 14.4 to 18.9 milliseconds. At 10-µs resolution, `pulsin` expresses these values as numbers ranging from 1440 to 1890. (Actually, thanks to stray capacitance, the numbers returned by the circuit will tend to be higher than this.)

In order to prevent clock pulses from interfering with temperature measurements, the RH clock is disabled when not in use. If you really need the extra pin, you can tie pin 5 of the sensor board high, leaving the clock on continuously. You may need to average several temperature measurements to eliminate the resulting jitter, however.

Since the accuracy of both of the measurement techniques is highly dependent on the individual components and circuit layout used, we're going to sidestep the sticky issue of calibration and conversion to units. A recent article in *Popular Electronics* (January 1994 issue, page 62, "Build a Relative-Humidity Gauge") tells how to calibrate RH sensors using salt solutions. Our previous application note (Stamp #7, "Sensing Temperature with a Thermistor") covers methods for converting raw data into units, even if the data are nonlinear.

Program listing and parts source. These programs may be downloaded from our ftp site at ftp.parallaxinc.com, or through our web site at http://www.parallaxinc.com. The sensor kit (#WEA-TH-KIT) is available for $25 plus shipping and handling from Fascinating Electronics, PO Box 126, Beaverton, OR 97075-0126; phone, 1-800-683-5487.

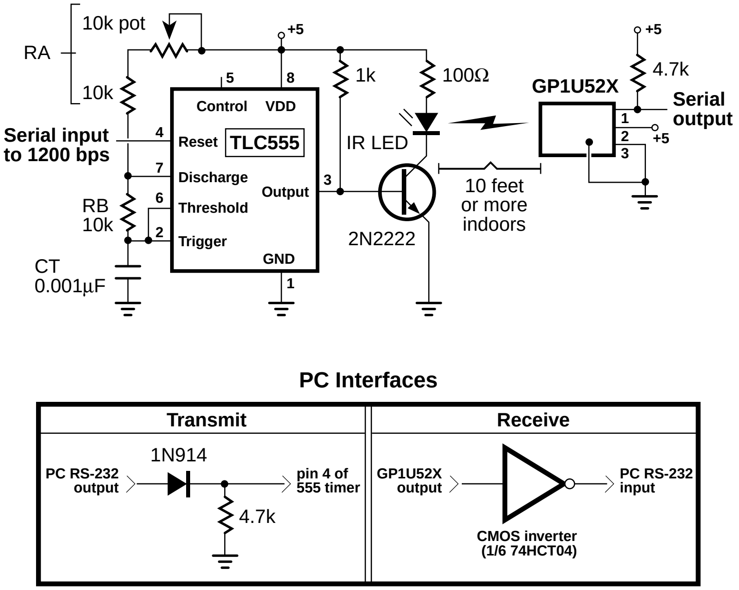
```
' Program HUMID.BAS
' The Stamp interfaces to an inexpensive temperature/humidity
' sensor kit.

Symbol   temp = w4                    ' Temperature
Symbol   RH = w5                      ' Humidity

' The main program loop reads the sensors and displays
' the data on the PC screen until the user presses a key.

Loop:
input 0:input 2: output 3
```

```
low 2: low 3
let temp = 500                          ' Start temp at a reasonable value.

ReadTemp:
        output 1: low 1
        pause 1                         ' Discharge the capacitor.
        input 1                         ' Get ready for input.
        pulsout 2,temp                  ' Charge cap thru temp sensor.
        if pin1 = 1 then ReadRH         ' Charged: we're done.
        let temp = temp + 1 ' Else try again
        goto ReadTemp                   ' with wider pulse.

ReadRH:
        high 3                          ' Turn on the 555 timer
        pause 500                       ' and let it stabilize.
        pulsin 0,1,RH                   ' Read the pulse width.
        low 3                           ' Kill the timer.
        debug temp:debug RH             ' Display the results.
        goto Loop                       ' Do it all again.
```

Introduction. This application note shows how to build a simple and inexpensive infrared communication interface for the BASIC Stamp.

Background. Today’s hottest products all seem to have one thing in common; wireless communication. Personal organizers beam data into desktop computers and wireless remotes allow us to channel surf from our couches. Not wanting the BASIC Stamp to be left behind, we devised a simple infrared data link. With a few inexpensive parts from your neighborhood electronics store you can communicate at 1200 baud over distances greater than 10 feet indoors. The circuit can be modified for greater range by the use of a higher performance LED.

How it works. As the name implies, infrared (IR) remote controls transmit instructions over a beam of IR light. To avoid interference from other household sources of infrared, primarily incandescent lights, the beam is modulated with a 40-kHz carrier. Legend has it that 40 kHz was selected because the previous generation of ultrasonic remotes worked



Schematic to accompany program IR.BAS.

at this frequency. Adapting their circuits was just a matter of swapping an LED for the ultrasonic speaker.

The popularity of IR remotes has inspired several component manufacturers to introduce readymade IR receiver modules. They contain the necessary IR detector, amplifier, filter, demodulator, and output stages required to convert a 40-kHz IR signal into 5-volt logic levels. One such module is the GP1U52X, available from your local Radio Shack store as part no. 276-137. As the schematic shows, this part is all that’s required for the receiving section of our application.

For the transmitting end, all we need is a switchable source of 40-kHz modulation to drive an IR LED. That’s the purpose of the timer circuit in the schematic. Putting a 1 on the 555’s reset pin turns the 40-kHz modulation on; a 0 turns it off. You may have to fiddle with the values of RA, RB, and CT. The formula is Frequency $= 1.44/((RA+2*RB)*CT)$. With RB at 10k, the pot in the RA leg of the circuit should be set to about 6k for 40-kHz operation. However, capacitor tolerances being what they are, you may have to adjust this pot for optimum operation.

To transmit from a Stamp, connect one of the I/O pins directly to pin 4 of the ’555 timer. If you use pin 0, your program should contain code something like this:

```
low 0                          ' Turn off pin 0's output latch.
output 0                       ' Change pin 0 to output.
...                            ' other instructions
serout 0,N1200,("X")           ' Send the letter "X"
```

To receive with another Stamp, connect an I/O pin to pin 1 of the GP1U52X. If the I/O pin is pin 0, the code might read:

```
input 0                        ' Change pin 0 to input.
...                            ' other instructions
serin 0,T1200,b2               ' Receive data in variable b2.
```

To receive with a PC, you’ll need to verify that the PC is capable of receiving 5-volt RS-232. If you have successfully sent RS-232 from your Stamp to the PC, then it’s compatible. As shown in the schematic, you’ll need to add a CMOS inverter to the output of the GP1U52X. Don’t use a TTL inverter; its output does not have the required voltage swing. To transmit from a PC, you'll need to add a diode and resistor ahead of the '555 timer as shown in the schematic. These protect the timer from the negative voltage swings of the PC's real RS-232 output.

Modifications. I'm sure you're already planning to run the IR link at 2400 baud, the Stamp's maximum serial speed. Go ahead, but be warned that there's a slight detection delay in the GP1U52X that causes the start bit of the first byte of a string to be shortened a bit. Since the serial receiver bases its timing on the leading edge of the start bit, the first byte will frequently be garbled.

If you want more range or easier alignment between transmitter and receiver, consider using more or better LEDs. Some manufacturers' data sheets offer instructions for using peak current, duty cycle, thermal characteristics, and other factors to calculate optimum LED power right up to the edge of burnout. However, in casual tests around the workshop, we found that a garden-variety LED driven as shown could reliably communicate with a receiver more than 10 feet away. A simple reflector or lens arrangement might be as beneficial as an exotic LED for improving on this performance.

If you find that your IR receiver occasionally produces "garbage characters" when the transmitter is off, try grounding the metal case of the GP1U52X. It is somewhat sensitive to stray signals. If you build the transmitter and receiver on the same prototyping board for testing, you are almost certain to have this problem. Bypass all power connections with 0.1-µF capacitors and use a single-point ground. And be encouraged by the fact that the circuit works much better in its intended application, with the transmitter and receiver several feet apart.

Program listing. There's no program listing this time; however, you may download programs for other application notes from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

Introduction. This application note presents a circuit that allows the BASIC Stamp to measure distances from 1 to 12 feet using inexpensive ultrasonic transducers and commonly available parts.

Background. When the November 1980 issue of *Byte* magazine presented Steve Ciarcia's article *Home in on the Range! An Ultrasonic Ranging System*, computer hobbyists were fascinated. The project, based on Polaroid's SX-70 sonar sensor, allowed you to make real-world distance measurements with your computer. We've always wanted to build that project, but were put off by the high cost of the Polaroid sensor ($150 in 1980, about $80 today).

If you're willing to give up some of the more advanced features of the

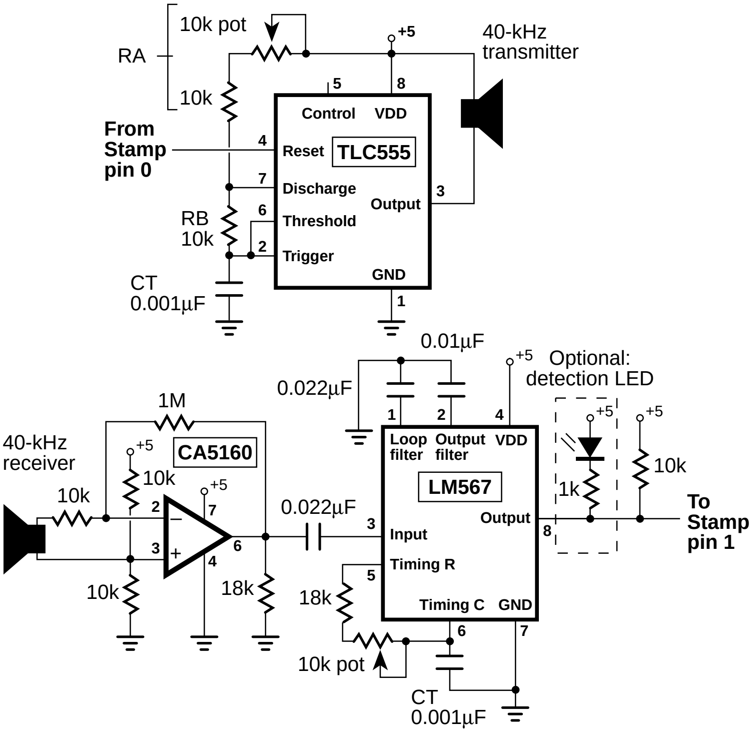


Figure 1. Schematic to accompany program SONAR.BAS.

Polaroid sensor (35-foot range, multi-frequency chirps to avoid false returns, digitally controlled gain) you can build your own experimental sonar unit for less than $10. Figure 1 shows how.

Basically, our cheap sonar consists of two sections; an ultrasonic transmitter based on a TLC555 timer wired as an oscillator, and a receiver using a CMOS op-amp and an NE567 tone decoder. The Stamp controls these two units to send and receive 40-kHz ultrasonic pulses. By measuring the elapsed time between sending a pulse and receiving its echo, the Stamp can determine the distance to the nearest reflective surface. Pairs of ultrasonic transducers like the ones used in this project are available from the sources listed at the end of this application note for $2 to $3.50.

Construction. Although the circuits are fairly self-explanatory, a few hints will make construction go more smoothly. First, the transmitter and receiver should be positioned about 1 inch apart, pointing in the same direction. For reasons we'll explain below, the can housing the transmitter should be wrapped in a thin layer of sound-deadening material. We used self-adhesive felt from the hardware store. Cloth tape or thin foam would probably work as well. Don't try to enclose the transducers or block the receiver from hearing the transmitter directly; we count on this to start the Stamp's timing period. More on this later. For best performance, the oscillation frequency of the TLC555 and the NE567 should be identical and as close to 40 kHz as possible. There are two ways to achieve this. One way is to adjust the circuits with a frequency counter. For the '555, temporarily connect pin 4 to +5 volts and measure the frequency at pin 3. For the '567, connect the counter to pin 5.

If you don't have a counter, you'll have to use ±5-percent capacitors for the units marked CT in the '555 and '567 circuits. Next, you'll need to adjust the pots so that the timing resistance is as close as possible to the following values. For the '555: Frequency = 1.44/((RA + 2*RB)* CT), which works out to $40\text{x}10^3 = 1.44/((16\text{x}10^3 + 20\text{x}10^3) \text{ x } 0.001\text{x}10^{-6})$.

Measure the actual resistance of the 10k resistors labeled RA and RB in the figure and adjust the 10k pot in the RA leg so that the total of the equation RA + 2*RB is 36k. Once the resistances are right on, the frequency of oscillation will depend entirely on CT. With 5-percent tolerance, this puts you in the ballpark; 38.1 to 42.1 kHz.

For the ’567 the math comes out like so: Frequency = 1/(1.1*R*CT); $40x10^3 = 1/(1.1 \text{ x } 22.73x10^3 \text{ x } 0.001x10^{-6})$

Adjust the total resistance of the 18k resistor and the pot to 22.73k. Again, the actual frequency of the ’567 will depend on CT. With 5-percent tolerance, we get the same range of possible frequencies as for the ’555; 38.1 to 42.1 kHz.

Once you get close, you can fine-tune the circuits. Connect the LED and resistor shown in the figure to the ’567. Temporarily connect pin 4 of the ’555 to +5 volts. When you apply power to the circuits, the LED should light. If it doesn’t, gradually adjust the pot on the ’555 circuit until it does. When you’re done, make sure to reconnect pin 4 of the ’555 to Stamp pin 0. Load and run the program in the listing. For a test run, point the transducers at the ceiling; a cluttered room can cause a lot of false echoes. From a typical tabletop to the ceiling, the Stamp should return `echo_time` values in the range of 600 to 900. If it returns mostly 0s, try adjusting the RA pot very, very slightly.

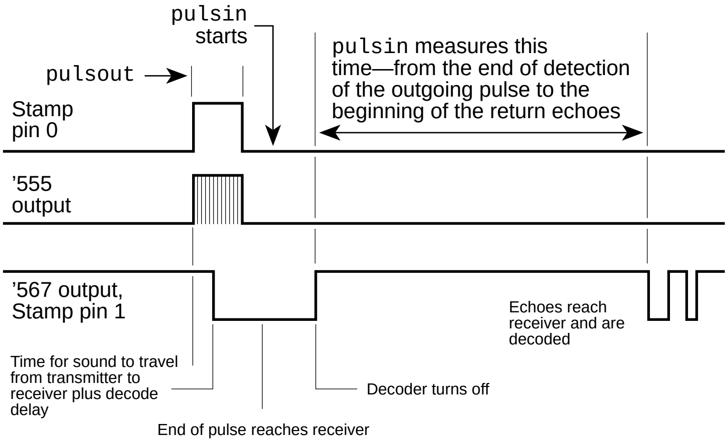

pulsin starts
pulsout
pulsin measures this time—from the end of detection of the outgoing pulse to the beginning of the return echoes
Stamp pin 0
’555 output
’567 output, Stamp pin 1
Echoes reach receiver and are decoded
Time for sound to travel from transmitter to receiver plus decode delay
Decoder turns off
End of pulse reaches receiver

Figure 2. Timing diagram of the sonar-ranging process.

How it works. In figure 1, the TLC555 timer is connected as a oscillator; officially an *astable multivibrator*. When its reset pin is high, the circuit sends a 40-kHz signal to the ultrasonic transmitter, which is really just a specialized sort of speaker. When reset is low, the ′555 is silenced.

In the receiving section, the ultrasonic receiver—a high-frequency microphone—feeds the CA5160 op amp, which amplifies its signal 100 times. This signal goes to an NE567 tone decoder, which looks for a close match between the frequency of an incoming signal and that of its internal oscillator. When it finds one, it pulls its output pin low.

Figure 2 illustrates the sonar ranging process. The Stamp activates the ′555 to send a brief 40-kHz pulse out through the ultrasonic transmitter. Since the receiver is an inch away, it hears this initial pulse loud and clear, starting about 74 µs after the pulse begins (the time required for sound to travel 1 inch at 1130 feet per second). After the ′567 has heard enough of this pulse to recognize it as a valid 40-kHz signal, it pulls its output low.

After `pulsout` finishes, the transmitter continues to ring for a short time. The purpose of the felt or cloth wrapping on the transmitter is to damp out this ringing as soon as possible. Meanwhile, the Stamp has issued the `pulsin` command and is waiting for the ′567 output to go high to begin its timing period. Thanks to the time required for the end of the pulse to reach the receiver, and the pulse-stretching tendency of the ′567 output filter, the Stamp has plenty of time to catch the rising edge of the ′567 output.

That's why we have to damp the ringing of the transmitter. If the transmitter were allowed to ring undamped, it would extend the interval between the end of `pulsout` and the beginning of `pulsin`, reducing the minimum range of the sonar. Also, if the ringing were allowed to gradually fade away, the output of the ′567 might chatter between low and high a few times before settling high. This would fool `pulsin` into a false, low reading.

On the other hand, if we prevented the receiver from hearing the transmitter at all, `pulsin` would not get a positive edge to trigger on. It would time out and return a reading of 0.

Once pulsin finds the positive edge that marks the end of the NE567's detection of the outgoing pulse, it waits. Pulsin records this waiting time in increments of 10 µs until the output of the '567 goes low again, marking the arrival of the first return echo. Using debug, the program displays this delay on your PC screen.

To convert this value to distance, first remember that the time pulsin measures is the round-trip distance from the sonar to the wall or other object, and that there's an offset time peculiar to your homemade sonar unit. To calibrate your sonar, carefully measure the distance in inches between the transmitter/receiver and the nearest wall or the ceiling. Multiply that number by two for the roundtrip, then by 7.375 (at 1130 feet/second sound travels 1 inch in 73.746 µs; 7.375 is the number of 10-µs pulsin units per inch). Now take a Stamp sonar reading of the distance. Subtract your sonar reading from the calculated reading. That's the offset.

Once you have the offset, add that value to pulsin's output before dividing by 7.375 to get the round-trip distance in inches. By the way, to do the division with the Stamp's integer math, multiply the value plus offset by 10, then divide by 74. The difference between this and dividing by 7.375 will be about an inch at the sonar's maximum range. The result will be the round-trip distance. To get the one-way distance, divide by two.

Modifications. The possibilities for modifications are endless. For those who align the project without a frequency counter, the most beneficial modification would be to borrow a counter and precisely align the oscillator and tone decoder.

Or eliminate the need for frequency alignment by designing a transmitter oscillator controlled by a crystal, or by the resonance of the ultrasonic transducer itself.

Try increasing the range with reflectors or megaphone-shaped baffles on the transmitter and/or receiver.

Soup up the receiver's amplifier section. The Polaroid sonar unit uses variable gain that increases with the time since the pulse was transmitted to compensate for faint echoes at long distances.

Make the transmitter louder. Most ultrasonic transmitters can withstand inputs of 20 or more volts peak-to-peak; ours uses only 5.

Tinker with the tone decoder, especially the loop and output filter capacitors. These are critical to reliable detection and ranging. We arrived at the values used in the circuit by calculating reasonable starting points, and then substituting like mad. There's probably still some room for improvement.

Many ultrasonic transducers can work as both a speaker and microphone. Devise a way to multiplex the transmit and receive functions to a single transducer. This would simplify the use of a reflector or baffle.

Parts sources. Suitable ultrasonic transducers are available from All Electronics, 1-800-826-5432. Part no. UST-23 includes both transmitter and receiver. Price was $2 at the time of this writing. Marlin P. Jones and Associates, 1-800-652-6733, stock #4726-UT. Price was $3.95 at the time of this writing. Hosfelt Electronics, 1-800-524-6464, carries a slightly more sensitive pair of transducers as part no. 13-334. Price was $3.50 at the time of this writing.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: SONAR.BAS
' The Stamp runs a sonar transceiver to measure distances
' up to 12 feet.

Symbol   echo_time = w2                ' Variable to hold delay time

setup:   let pins = 0                  ' All pins low
         output 0                      ' Controls sonar xmitter
         input 1                       ' Listens to sonar receiver

ping:    pulsout 0,50                  ' Send a 0.5-ms ping
         pulsin 1,1,echo_time          ' Listen for return
         debug echo_time               ' Display time measurement
         pause 500                     ' Wait 1/2 second
         goto ping                     ' Do it again.
```

Introduction. This application note shows how to use the 93LC66 EEPROM to provide 512 bytes of nonvolatile storage. It provides a tool kit of subroutines for reading and writing the EEPROM.

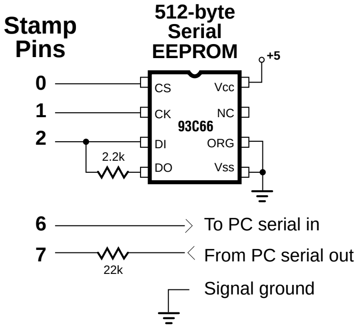
Background. Many designs take advantage of the Stamp's ability to store data in its EEPROM program memory. The trouble is that the more data, the smaller the space left for code. If only we could expand the Stamp's EEPROM!

This application note will show you how to do the next best thing; add a separate EEPROM that your data can have all to itself.

The Microchip 93C66 and 93LC66 electrically erasable PROMs (EEPROMs) are 512-byte versions of the 93LC56 used as the Stamp's program memory. (Before you ask: No, dropping a ′66 in place of the Stamp's ′56 will not double your program memory!) Serial EEPROMs communicate with a processor via a three- or four-wire bus using a simple synchronous (clocked) communication protocol at rates of up to 2 million bits per second (Mbps).

Data stored in the EEPROM will be retained for 10 years or more, according to the manufacturer. The factor that determines the EEPROM's longevity in a particular application is the number of erase/write cycles. Depending on factors such as temperature and supply voltage, the EEPROM is good for 10,000 to 1 million erase/write cycles. For a thorough discussion of EEPROM endurance, see the Microchip Embedded Control Handbook, publication number DS00092B, November 1993.

How it works. The circuit in the figure specifies a 93LC66 EEPROM, but a 93C66 will work as well. You can also subsitute the 256-byte ′56, provided you restrict the highest address to 255. The difference between the C and LC models is that the LC has a wider Vcc range (2.5–5.5 V,



Schematic to accompany EEPROM.BAS.

versus 4–5.5 V), lower current consumption (3 mA versus 4 mA), and can be somewhat slower in completing internal erase/write operations, presumably at lower supply voltages. In general, the LC type is less expensive, and a better match for the operating characteristics of the Stamp.

The schematic shows the data in and data out (DI, DO) lines of the EEPROM connected together to a single Stamp I/O pin. The 2.2k resistor prevents the Stamp and DO from fighting over the bus during a read operation. During a read, the Stamp sends an opcode and an address to the EEPROM. As soon as it has received the address, the EEPROM activates DO and puts a 0 on it. If the last bit of the address is a 1, the Stamp could end up sourcing current to ground through the EEPROM. The resistor limits the current to a reasonable level.

The program listing is a collection of subroutines for reading and writing the EEPROM. All of these rely on `Shout`, a routine that shifts bits out to the EEPROM. To perform an EEPROM operation, the software loads the number of clock cycles into `clocks` and the data to be output into `ShifReg`. It then calls `Shout`, which does the rest.

The demonstration program calls for you to connect the Stamp to your PC serial port, type in up to 512 characters of text, and hit return when you're done. Please type this sample text rather than downloading a file to the Stamp. The Stamp will miss characters of a rapidly downloaded file, though it's more than fast enough to keep up with typing. As you type in your message, the Stamp will record each character to EEPROM.

When you're finished typing, the Stamp will repeat your text back to the PC serial port. In fact, it will read all 512 bytes of the EEPROM contents back to the PC.

If you don't have the EEPROM data handy (Microchip Data Book, DS00018D, 1991), you should know about a couple of subtleties. First, when the EEPROM powers up, it is write protected. You must call `Eenable` before trying to write or erase it. It's a good idea to call `Edisbl` (disable writes) as soon as possible after you're done. Otherwise, a power glitch could alter the contents of your EEPROM.

The second subtle point is that National Semiconductor makes a series of EEPROMs with the same part numbers as the Microchip parts discussed here. However, the National parts use a communication protocol that's sufficiently different to prevent them from working with these routines. Make sure to ask for Microchip parts, or be prepared to rewrite portions of the code.

Modifications. If you're using PBASIC interpreter chips as part of a finished product, you may be contemplating buying a programmer to duplicate EEPROMs for production. If you'd prefer to avoid the expense, why not build a Stamp-based EEPROM copier? Just remember to include a 2-millisecond delay or read the busy flag between sequential writes to an EEPROM. This is required to allow the internal programming process to finish. These topics are covered in more detail in the EEPROM documentation.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: EEPROM.BAS
' This program demonstrates subroutines for storing data in a
' Microchip 93LC66 serial EEPROM. This program will not work
' with the National Semiconductor part with the same number.
' Its serial protocol is substantially different.

Symbol  CS       = 0            ' Chip-select line to pin 0.
Symbol  CLK      = 1            ' Clock line to pin 1.
Symbol  DATA     = pin2         ' Destination of Shout; input to Shin
Symbol  DATA_N   = 2            ' Pin # of DATA for "input" & "output"
Symbol  ReadEE   = $C00         ' EEPROM opcode for read.
Symbol  Enable   = $980         ' EEPROM opcode to enable writes.
Symbol  Disable  = $800         ' EEPROM opcode to disable writes.
Symbol  WriteEE  = $A00         ' EEPROM opcode for write.
Symbol  GetMSB   = $800         ' Divisor for getting msb of 12-bit no.
Symbol  ShifReg  = w1           ' Use w1 to shift out 12-bit sequences.
Symbol  EEaddr   = w2           ' 9-bit address for reads & writes.
Symbol  EEdata   = b6           ' Data for writes; data from reads.
Symbol  i        = b7           ' Index counter for EEPROM routines.
Symbol  clocks   = b10          ' Number of bits to shift with Shout.

        output DATA_N           ' EEPROM combined data connection.
        output CLK              ' EEPROM clock.
        output CS                   ' EEPROM chip select.

' Demonstration program to exercise EEPROM subroutines:
' Accepts serial input at 2400 baud through pin 7. Type a
' message up to 512 characters long. The Stamp will store
' each character in the EEPROM. When you reach 512 characters
' or press return, the Stamp will read the message back from
' the EEPROM and transmit it serially through pin 6
' at 2400 baud.

        output 6                    ' For serial output.
        input  7                    ' For serial input.
        gosub Eenabl                ' Remove EEPROM write protection.
        let EEaddr=0                ' Start at 1st (0th) address.
CharIn: serin 7,N2400,EEdata        ' Get character.
        if EEdata<32 then Done      ' If it's return, done.
        gosub Ewrite                ' Otherwise, write to EEPROM.
        let EEaddr=EEaddr+1         ' Increment addr for next write.
        if EEaddr=512 then Done     ' Memory full? Done.
        goto CharIn

Done:   gosub Edisbl                ' Protect EEPROM.
        for w4 = 0 to 511           ' Show all 512 bytes.
        let EEaddr = w4             ' Point to EEPROM address.
        gosub Eread                 ' Retrieve the data.
        serout 6,N2400,(EEdata)     ' Send it out serial port.
        next                        ' Next character.
        End                         ' Demo over.

' Write the data in EEdata to the address EEaddr.
Ewrite: let ShifReg=WriteEE         ' Get the write opcode.
        let ShifReg=ShifReg|EEaddr  ' OR in the address bits.
        let clocks = 12             ' Send 12 bits to EEPROM.
        high CS                     ' Chip select on.
        gosub Shout                 ' Send the opcode/address.
        let ShifReg = EEdata*16     ' Move bit 7 to bit 11.
        let clocks = 8              ' Eight data bits.
        gosub Shout                 ' Send the data.
        low CS                      ' Deselect the EEPROM.
        return

' Read data from EEPROM address EEaddr into EEdata.
Eread:  let ShifReg=ReadEE          ' Get the read opcode.
        let ShifReg=ShifReg|EEaddr  ' OR in the address bits.
        let clocks=12               ' Send 12 bits to EEPROM.
        high CS                     ' Chip select on.
        gosub Shout                 ' Send the opcode/address.
        gosub Shin                  ' Receive the byte.
        low CS                      ' Deselect the EEPROM.
        return

' Enable writes to the EEPROM. Upon power-up the EEPROM is
' write-protected, so this routine must be called before
' first writing to the EEPROM.
Eenabl: let ShifReg=Enable          ' Get the write-enable opcode.
        high CS                     ' Chip select on.
        let clocks = 12             ' Send 12 bits to EEPROM.
        gosub Shout                 ' Send the opcode.
        low CS                      ' Deselect the EEPROM.
        return

' Disable writes to the EEPROM.
Edisbl: let ShifReg=Disable         ' Get the write-disable opcode.
        high CS                     ' Chip select on.
        let clocks = 12             ' Send 12 bits to EEPROM.
        gosub Shout                 ' Send the opcode
        low CS                      ' Deselect the EEPROM
        return

' Shift data into EEdata.
Shin:   input DATA_N                ' Change the data line to input.
        let EEdata=0                ' Clear data byte.
        for i = 1 to 8              ' Prepare to get 8 bits.
        let EEdata=EEdata*2         ' Shift EEdata to the left.
        high CLK                    ' Data valid on rising edge.
        let EEdata=EEdata+DATA      ' Move data to lsb of variable.
        low CLK                     ' End of clock pulse.
        next i                      ' Get another bit.
        output DATA_N               ' Restore data line to output.
        return

' Shift data out of ShifReg.
Shout:  for i = 1 to clocks         ' Number of bits to shift out.
        let DATA=ShifReg/GetMSB     ' Get bit 12 of ShifReg.
        pulsout CLK,10              ' Output a brief clock pulse.
        let ShifReg=ShifReg*2       ' Shift register to the left.
        next i                      ' Send another bit.
        return
```

Introduction. This application note shows how to connect multiple Stamps together in a simple network. It explains the use of the serout open-drain and open-source baudmodes.

Background. Many Parallax customers are interested in connecting multiple Stamps together to form a network. Their applications include intelligent home control, security sytems, small-scale robotics, and distributed sensing arrangements. For these applications, the Stamp has built-in serial networking capabilities requiring a minimum of external components. Better yet, participation in a network requires only a couple of lines of Stamp code and one additional I/O line at most.

How it works. The first question that comes to mind is: "Why not just connect multiple Stamps to one serial port and make them talk one at a time? That would be a good enough network for most jobs." That's true, for the most part, but the Stamp's normal serial outputs would destroy each other. Figure 1 shows why.

In output mode, the Stamp's I/O pins act like switches connected to the power-supply rails. When the Stamp outputs a 1, it's turning on the switch connected to the +5-volt rail while turning off the one going to ground. To output a 0, it does the reverse. If you connect multiple Stamp outputs together, you set up the situation in figure 1b: a direct short to ground through a pair of Stamp output switches. This would damage the Stamp's PBASIC interpreter chip.

Now, before you run off to design a system of logic gates or diodes to fix this, listen up: The Stamp can be configured to use only one of the two switches for serial output. This eliminates the possibility of a short circuit and opens up the possibility of network hookups. See figure 2.

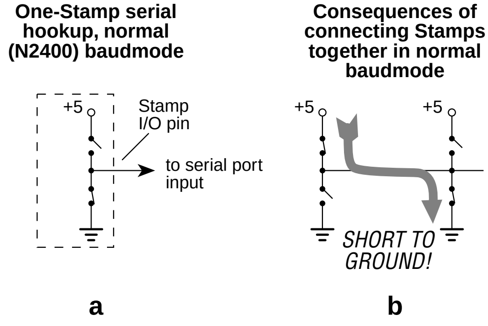

One-Stamp serial hookup, normal (N2400) baudmode
+5
Stamp I/O pin
to serial port input
a
Consequences of connecting Stamps together in normal baudmode
+5
+5
SHORT TO GROUND!
b

Figure 1

To use this technique, your program should begin by setting the serial pin to input in order to turn off the output switches. Then, when it's time for the Stamp to put some data onto the network using **serout**, the baudmode argument should begin with **OT**, as in **OT2400**. This is known as an open-drain configuration, in honor of the portion of the PBASIC interpreter's output MOSFET switch left "open" at the pin connection.

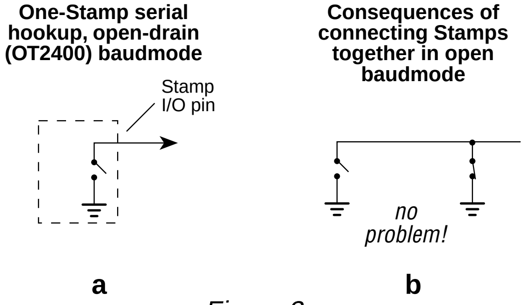

One-Stamp serial hookup, open-drain (OT2400) baudmode
Stamp I/O pin
a
Consequences of connecting Stamps together in open baudmode
no problem!
b

Figure 2

When connected Stamp pins are in different states there's no problem, because no current flows. No data flows, either, because the pins are incapable of outputting a logical 1 (+5 volts). That's easily remedied by adding a pullup resistor, however, as shown in figure 3.

The inverter/line driver shown in figure 3 can either be a CMOS type like one-sixth of a 74HCT04 or an actual RS-232 line driver, such as a MAX-232. If the Stamps will be talking to each other instead of reporting to a host computer, you can eliminate the line driver entirely.

The Stamp also supports an open baudmode that switches to +5 only instead of ground. This is the open-source configuration, selected by an argument beginning with **ON**, such as **ON2400**. To make this work, you must reverse the polarity of everything shown in figure 3. The resistor would go to ground. A non-inverting buffer (or additional inverter) would be used to straighten out the signal polarity.

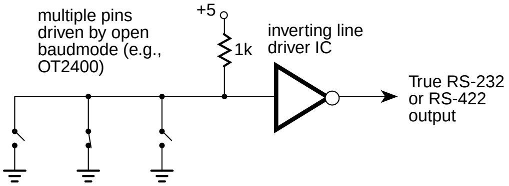

multiple pins driven by open baudmode (e.g., OT2400)
+5
1k
inverting line driver IC
True RS-232 or RS-422 output

Figure 3

Now that we have a way to safely connect multiple Stamp serial pins to a single line, how do we ensure that only one Stamp talks at once? The possibilities seem endless, and depend primarily on the nature of the data to be sent through the net. For example, each Stamp could alternate between talking and listening on the net. You could use a system of qualifiers that each Stamp would have to receive via **serin** before it could transmits onto the net. That way, one Stamp would send its data, then turn the net over to the next. That is the approach used in the demonstration programs.

Of course, if you have I/O pins available on each of the Stamps in the net, you could just have each Stamp wait for a particular logic level to tell it to transmit. Another approach would be to have one Stamp trigger its neighbor. As I said, the possibilities go on and on. If you get stuck for ideas, just look at a diagram of a local-area network (LAN). LAN designers have invented all kinds of schemes, called "network topologies," for determining who talks when. They've dreamed up good names, too, like *token rings*, *stars*, *hubs*, etc.

The example we present is a variation on the token-ring idea. Three Stamps named Moe, Larry, and Curly will share a single serial line. When they are first powered up, Moe will transmit a message concluding with "Larry." Larry, recognizing his name, will transmit a message concluding with "Curly." Curly will transmit a message, concluding with "Moe." Moe will start the process all over again. Even though the Stamps are communicating among themselves, we'll still use an inverter/driver in order to monitor the process with a PC running

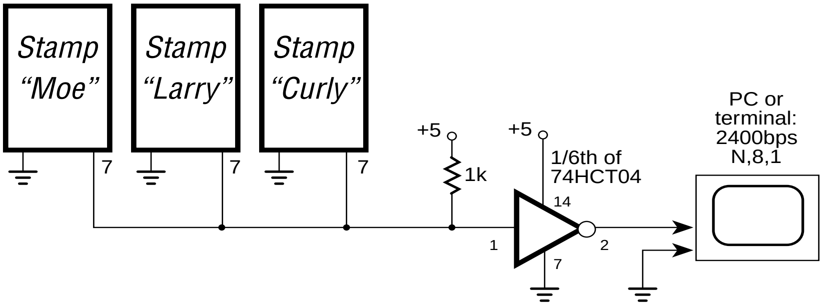


Figure 4. Serial network of Stamps using open-drain output.

terminal software. Figure 4 shows the circuit; the program listing shows the code used in the Stamps.

For your application, you'd simply substitute a real message (based on data gathered by the Stamps) for the sample messages. Make sure that your data messages cannot contain the names of the other Stamps, or you'll create chaos. A safe bet is to restrict data to numbers, and names to text. Make sure that the individual Stamps can gather data quickly enough to be ready when their names are called. If they're not ready, they may miss their cues. This can cause the entire net to hang up. Likewise, simple failure of one of the Stamps will hang the net. For critical applications, you might want to consider making one of the Stamps a supervisor whose job it is to handle these emergencies.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: Moe
' Stamp participant in a simple ring-type network. This Stamp has the job of
' starting the network up by passing an initial message before receiving a cue.
' Thereafter, Moe only transmits when cued by Curly, the last Stamp on the net.

input 7                                     ' Set pin 7 to input.
pause 1000                                  ' Give others time to wake up.
serout 7,OT2400,(10,13,"Three   ")          ' Say the line.
serout 7,OT2400,("Larry",10,13)             ' Cue next.

' Now enter the main program loop.
Loop:
      serin 7,T2400,("Moe",10,13)           ' Wait for cue.
      serout 7,OT2400,(10,13,"Three   ")    ' Say the line.
      serout 7,OT2400,("Larry",10,13)       ' Cue next.
goto Loop
```

```
' Program: Larry
' Stamp participant in a simple ring-type network. Only transmits when cued
' by Moe, the first Stamp on the net.

input 7                                     ' Set pin 7 to input.

' Main program loop:
```

```
Loop:
        serin 7,T2400,("Larry",10,13)          ' Wait for cue.
        serout 7,OT2400,("Blind   ")           ' Say your line.
        serout 7,OT2400,("Curly",10,13)        ' Cue next
goto Loop


' Program: Curly
' Stamp participant in a simple ring-type network. Only transmits when cued
' by Larry, the middle Stamp in the net.

input 7                                        ' Set pin 7 to input.

' Main program loop:
Loop:
        serin 7,T2400,("Curly",10,13)          ' Wait for cue.
        serout 7,OT2400,("Mice    ")           ' Say your line.
        serout 7,OT2400,("Moe",10,13)          ' Cue next
goto Loop
```

Introduction. This application note explains how to convert digital values to analog voltages using the BASIC Stamp command `pwm`.

Background. There's probably some misunderstanding about the pulse-width modulation (`pwm`) command. Most Stamp users know that it generates a waveform whose duty cycle (ratio of on time to off time) can be varied from 0 to 100 percent by varying the `duty` input from 0 to 255. But experience with other devices probably leads them to expect the output to look like figure 1. This is the sort of variable duty cycle output you get from most timer and counter circuits.

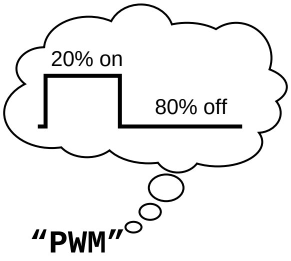


Figure 1. What users think pwm looks like.

The Stamp uses a different, more efficient algorithm to generate pwm. Its output is just as useful for generating analog voltages, but when displayed on an oscilloscope, it can look like a mess; see figure 2.

The proportion of on time to off time is the same, but instead of being separated into neat chunks, Stamp pwm is distributed over a large number of pulses of varying width.

Without getting too far into the details, the reason is this: The Stamp generates pwm by adding the duty cycle into an internal variable that we'll call the "accumulator." It doesn't care what the accumulator contains, only whether or not it overflows (generates a carry-the-one operation). If it does, the pwm pin goes high; otherwise, low.

The Stamp does this addition quite a few times. The larger the duty cycle is, the more often carries occur, and the higher the proportion of highs to lows. However, the carries occur with an irregular rhythm, so the output waveform, while perfect duty-cycle pwm, looks like fruit salad on the 'scope.

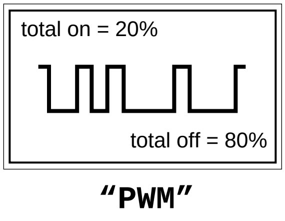


Figure 2. What pwm really looks like.

Using pwm. The primary application for pwm is to generate a voltage from 0 to 5 volts proportional to the duty cycle. An even simpler use is to control the brightness of an LED. See figure 3.

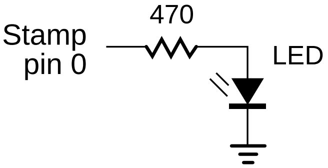


Figure 3. Controlling the brightness of an LED with pwm.

If, as shown in the figure, the LED is connected to pin 0, the following fragment of a Stamp program will gradually raise the brightness of the LED from off to fully on:

```
low 0                      ' LED completely off.
for b2 = 0 to 255          ' Loop from off to on.
      pwm 0, b2,1          ' Output one burst of pwm.
next
high 0                     ' Leave LED on.
```

Although the Stamp is sending a stream of 1s and 0s to the LED, it appears to be steadily on at varying levels of brightness. That's because your eyes integrate (smooth) the rapid flickering, just as they do the frames of a movie or television picture.

If you look at the `pwm` writeup in the Stamp manual, you'll see that in most applications you need a resistor and capacitor to integrate the output to a smoothly varying voltage. What the manual doesn't show is the effect that connected circuits can have on a simple resistor/capacitor (RC) integrator.

The fact is that if you try to draw too much current from the RC circuit, your program will have to output many cycles of pwm, and do so quite often in order to maintain the charge on the capacitor. Otherwise, the voltage level set by pwm will begin to droop.

Figure 4 shows one way to overcome this. The CA5160E operational amplifier (op amp) has an extremely high input impedance, so it draws very little current from the capacitor. Since its gain is set to 1 by the accompanying components, the voltage at its output is the same as the voltage at its input, with one big exception: The current drawn from the

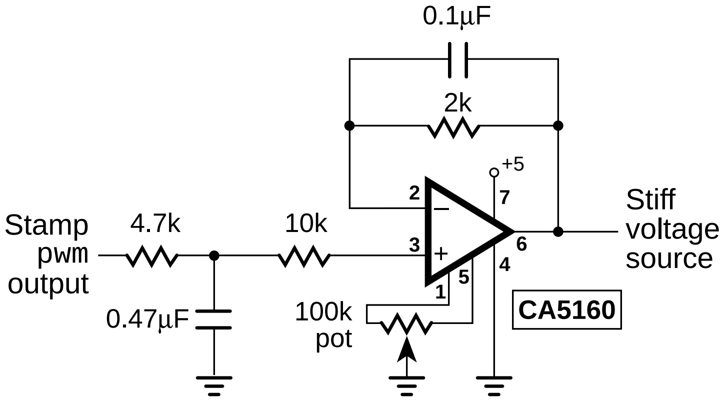


Figure 4. Example op-amp buffer circuit.

output of the op amp does not affect the charge on the capacitor in the RC integrator.

According to the op amp's specs, you can draw up to 12 mA from its output. Other op amps may offer higher current outputs, but make sure to check all the specifications. The CA5160 was used here because it is happy operating from a 5-volt, single-ended supply. Supply current is typically 50 µA (ignoring current drawn from the output). Other op amps may require split supplies of ±15 volts or more.

To drive the op-amp circuit properly, the pin used for pwm output must actually be defined as an input. This ensures that once pwm establishes a voltage level on the capacitor it disconnects itself from the circuit. The code we used to set the circuit to approximately 2.5 volts is:

```
input 0                  ' Make pin 0 an input.
pwm 0, 127,1             ' Output one burst of pwm.
```

In our tests, one burst of pwm was sufficient to charge the capacitor to the desired voltage. Once set, the voltage at the op amp's output (driving a 1k resistor load) remained steady for more than 15 minutes. It actually drifted slowly upward, probably due to slight current leakage from the Stamp I/O pin. In a real application, you should try to reestablish the voltage level more often than once every 15 minutes.

A few final notes about the circuit. The 100k pot allows you to fine-tune the op amp's output offset. Connect the circuit with an accurate voltmeter at the output. Program the Stamp to kick out a burst of PWM. The voltage appearing at the op amp output should be (duty/255) times the supply voltage (5 volts from the Stamp's regulated supply). So if the duty is 127, the output voltage should be (127/255) * 5 = 2.49 volts. Adjust the pot until the actual voltage agrees with your calculation.

You may find that your op amp won't track the input voltage all the way to +5 volts. One solution is to simply ignore this limitation, and just work within the range you do get. Another is to connect the + supply pin of the op amp (pin 7) to unregulated +9 volts from the battery. As the battery dies, you'll eventually have the same problem again, but you will get rail-to-rail performance for most of the battery's life.

Program listing. There's no program listing this time; however, you may download programs for other application notes our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

Introduction. This application note explains how to use the BASIC Stamp directive `bsave` and the program BSLOAD.EXE to enable customers to update Stamp programs without access to the source code. It also shows a method by which the Stamp can reload its own program memory from data received over RS-232.

Background. Try this: Phone Microsoft and tell them you own Excel™ or another product of theirs, and you'd like a copy of the source code. Be generous; tell them you are willing to pay for the disks and the shipping charges.

You'll probably find out how people working at a big corporation react to pranks. You'll also learn a lot of new ways to gently say "no."

If you want to keep your Stamp source code private, but still allow customers to download alternative functions, change EEPROM data, or update firmware, you need to know about `bsave`.

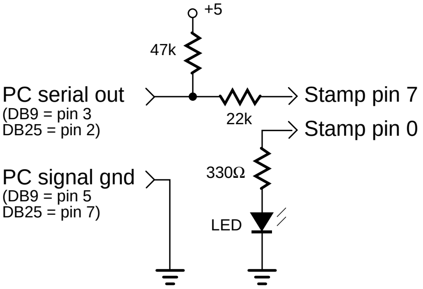
Using bsave. When you run a Stamp program using the latest versions of STAMP.EXE, the software looks for the directive `bsave` on a line by itself anywhere in the source-code listing. If `bsave` is present, the software saves a 256-byte file called CODE.OBJ to the current directory. That file contains a copy of the binary data written to the Stamp's EEPROM. You can rename and distribute that file, along with a program called BSLOAD.EXE that's available from the Parallax bulletin-board system.

If you renamed your code file UPDATE.OBJ (it's smart to retain the extension .OBJ because that is the default recognized by BSLOAD) you could distribute it, the BSLOAD software, and a Stamp cable to your customer. Instruct the customer to connect the cable from a PC to the Stamp-based product, connect power, and type BSLOAD UPDATE. A status message appears on the screen to indicate the success of the download.

This technique eliminates the need to distribute your source files and STAMP.EXE in order to update Stamp firmware.

Self-replacing programs. This is going to sound like do-it-yourself brain surgery, but it's possible to write Stamp programs that replace themselves in EEPROM program memory. This means you can download a new program to the Stamp via a serial link.

Program listings 1 and 2 are examples of self-replacing programs. The trick lies in the fact that both contain identical startup code. This code, with the assistance of the serial hookup depicted in the figure, determines whether or not a serial output is connected to the Stamp at startup. If no serial connection is present, the Stamp goes about its business—in the cases of listings 1 and 2, flashing an LED in two different ways. If a serial connection is present at startup, the Stamp receives 256 bytes of data and uses them to replace the entire contents of its EEPROM.



This means that the program overwrites itself. It doesn't crash, however, because the replacement program contains the same code in the same place; at the very beginning of the program.

Listing 3 is a QBASIC program that performs the serial downloading. You may copy and modify this program to fit your own requirements. When you write your own version, be sure to note that QBASIC must open the .OBJ file as binary data. Otherwise, chances are good that a control-Z character (ASCII 26) somewhere in the .OBJ file would cause QBASIC to end the download.

Here's how to make this demonstration work: Construct the circuit shown in the figure, but do not connect your PC's serial port to the Stamp yet. Load and run the Stamp program THROB.BAS. Because the file contains `bsave`, the Stamp software will write its binary image to the file CODE.OBJ. Make a mental note of the DOS path to this file; you'll need it for the downloading step. Next load and run BLINK.BAS. Since it does not contain `bsave`, this will not generate an object file.

Now, quit the Stamp software and disconnect the Stamp from its battery or power supply. Remember, the Stamp only looks for the serial connection at startup, otherwise, it goes into its normal loop.

Boot QBASIC or QuickBASIC and load the program REPROG.BAS. Before you run the program, type your path name into the command `OPEN "D:\CODE.OBJ" FOR BINARY AS #1`. Connect the PC's serial output as indicated in the figure and apply power to the Stamp. Now run the program.

As the download proceeds, the program will display the current byte number on the screen of your PC, and the Stamp will blink its LED in time to the arriving data. A large `FOR/NEXT` delay has been added to the downloading loop to prevent it from outrunning the EEPROM programming process.

When the download is over, the Stamp will begin running THROB.BAS. If you like, you can create an object file of BLINK.BAS and follow the procedures above to replace THROB. Or you can write your own program, include the downloading code, and replace either program with your program.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Listing 1: Blink.BAS
' This program can replace itself with a new program
' downloaded via a serial connection. It is part of a
' demonstration described in Stamp application note 16.
if pin7 = 1 then Loop                   ' No serial hookup so skip.
for b2 = 255 to 0 step -1               ' Download 256 bytes
        serin 7,N2400,b4                ' Get a byte.
        write b2,b4                     ' Write to EEPROM.
        toggle 0 ' Flash LED.
next

Loop:   ' Main program loop:
        toggle 0 ' blink LED.
        pause 50
goto Loop
```

```
' Listing 2: Throb.BAS
' This program can replace itself with a new program
' downloaded via a serial connection. It is part of a
' demonstration described in Stamp application note 16.
if pin7 = 1 then Loop                       ' No serial hookup so skip.
for b2 = 255 to 0 step -1                   ' Download 256 bytes
        serin 7,N2400,b4                    ' Get a byte.
        write b2,b4                         ' Write to EEPROM.
        toggle 0 ' Flash LED.
next

Loop:                                       ' Main program loop:
        for b2 = 0 to 255 step 5            ' make LED "throb"
        pwm 0,b2,1                          ' by varying its brightness
        next                                ' using pwm.
        for b2 = 255 to 0 step -3
        pwm 0,b2,1
        next
goto Loop
```

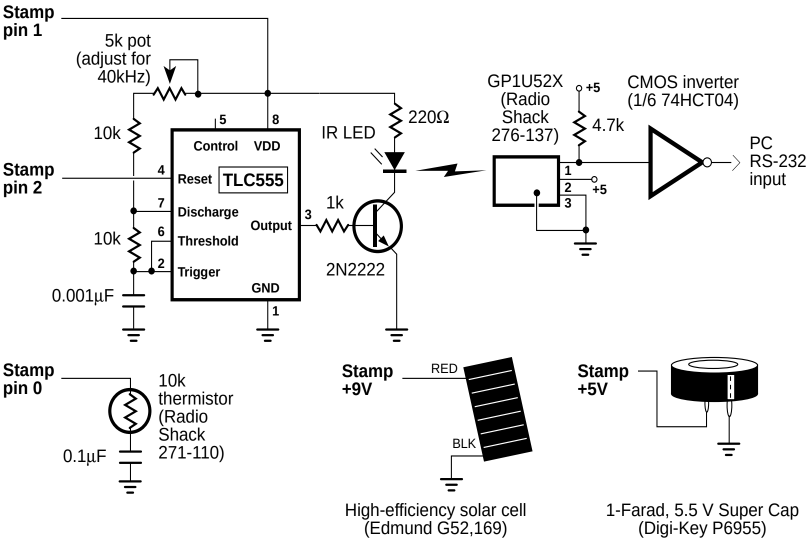
```
' Listing 3: REPROG.BAS (NOT a Stamp program)
' This is a QBASIC program that will download a Stamp
' object file via an RS-232 serial hookup. Be sure to
' enter the correct path to your CODE.OBJ file in the
' OPEN command below.
DEFINT A-Z:CLS
DIM code(255) AS INTEGER
' Load the contents of the CODE.OBJ into a variable.
' Replace "D:\CODE.OBJ" with your file's path.
OPEN "D:\CODE.OBJ" FOR BINARY AS #1
FOR i = 0 TO 255
        code(i) = ASC(INPUT$(1, 1))
NEXT i
CLOSE #1
' Send the code bytes out the serial port.
OPEN "COM1:2400,N,8,1,CD0,CS0,DS0,OP0" FOR RANDOM AS #1
FOR i = 0 TO 255
        CLS: PRINT "Sending: "; i
        PRINT #1, CHR$(code(i));
        FOR j = 1 TO 20000: NEXT j  ' Large delay.
NEXT i
CLOSE #1
END
```

Introduction. This application note shows how to operate the Stamp 24 hours a day from the power provided by a 1.5" x 2.5" solar battery. The example application takes outdoor temperature measurements every 12 seconds, then relays them to a computer indoors via an infrared link.

Background. A standard 9-volt battery can power the Stamp for a long time with the use of the **sleep** and **nap** commands. But eventually the battery *will* die, if only from old age.

Although it's usually no problem to just replace the battery, there are applications that require long periods of unattended operation. Imagine a mountaintop weather station, forest wildlife counter, or floating sensor buoy, drifting in the currents at sea. Now imagine the cost of mounting an expedition to replace the Stamp's battery. Whew!

There are also less exotic places in which independence from batteries would be a good idea. How about pollution-measuring instruments at the top of a pole, or an electronic bicycle speedometer?


Stamp pin 1
5k pot (adjust for 40kHz)
10k
Stamp pin 2
10k
0.001µF
5
8
Control
VDD
4
Reset
TLC555
7
Discharge
6
Threshold
Output
2
Trigger
GND
1
3
1k
2N2222
IR LED
220Ω
GP1U52X (Radio Shack 276-137)
+5
4.7k
1
2
3
+5
CMOS inverter (1/6 74HCT04)
PC RS-232 input
Stamp pin 0
10k thermistor (Radio Shack 271-110)
0.1µF
Stamp +9V
RED
BLK
Stamp +5V
High-efficiency solar cell (Edmund G52,169)
1-Farad, 5.5 V Super Cap (Digi-Key P6955)

Schematic to accompany program SOLAR.BAS

Solar batteries can solve these problems, but only during the daytime. At night, the Stamp would have to run off an alternative power source, such as a rechargeable battery. However, rechargeables are notoriously fussy about proper care and feeding, which might become more of a job than the Stamp's primary mission.

In keeping with the minimalist philosophy of the Stamp itself, we decided to try the simplest conceivable combination of round-the-clock power; a solar battery and a really big capacitor.

How it works. For a trial application, we borrowed from two previous application notes. We took the thermistor temperature measurement scheme of note #7 and wedded it to the infrared communication setup of note #11. That way, we could show that the Stamp and some fairly current-hungry peripherals could both share our 24-hour power source. See the schematic.

For our test of the project, the Stamp, 40-kHz transmitter, and super capacitor were mounted outdoors in a small cardboard box taped to a window on the shady side of a building. The box helped protect the circuit from the elements, and provided a dark background for the IR LED. The solar battery was mounted outside the box, angled upward. On the indoor side of the window, a breadboard holding the IR receiver, CMOS inverter and power supply was pointed at the IR LED on the other side of the glass.

The project works like this: Every 12 seconds the Stamp takes a temperature reading by executing a **pot** command on pin 0, the pin to which the thermistor is connected. It converts the resulting byte of data into the current temperature using the power-series technique described in app note #7. Then the Stamp applies power to the '555 transmitter circuit. The Stamp sends a byte of data at 1200 baud to the pin 4 of the '555, causing it to transmit the data as an on/off-keyed 40-kHz infrared signal.

The infrared remote-control receiver (GP1U52X) converts the modulated light beam back into bits. A CMOS inverter reverses the polarity of the bits and provides sufficient voltage swing for reception through a PC serial port. The PC receives the data, adds a time tag, and records it to a file on the hard drive.

When the serial transmission is done, the PIC turns off the ’555 and goes to sleep for another 12 seconds.

From the standpoint of the project’s solar power source, there isn’t much to explain. The specified solar battery produces up to 10 volts at 9 mA in direct sunlight, or 8 volts and 0.075 mA in typical indoor lighting. We split the difference and mounted the battery outdoors on the shady side of a building. At noon in this location we got 10 volts at about 1 mA.

Before installing the 1-Farad super capacitor, we charged it to about 4 volts by leaving it connected to a 5-volt power supply through a 4.7k resistor for several hours. This limited the amount of charging current that the capacitor would demand from the Stamp’s voltage regulator when first connected. Once the capacitor was installed, the solar battery kept it charged.

We ran the project ’round the clock for several days, periodically reviewing the time-tagged data files for breaks or erratic data that would indicate a power failure. None occurred. The lowest voltage across the super cap, which occurred after about 10 hours of darkness, was 3.65 volts, just enough to keep the Stamp going. Less than an hour after sunrise the cap would charge back up to 5 volts.

The solar battery has plenty of excess capacity for this type of application. An interesting challenge would be to find ways to exploit this. For example, in a telemetry application, the Stamp might store data over night, then transmit it during daylight when power would be abundant.

Parts sources. The solar battery is available from Edmund Scientific, 609-573-6250. The super cap is available from Digi-Key, 800-344-4539. Many of the other components used in the circuit are available from Radio Shack electronics stores.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: SOLAR.BAS
' Program to demonstrate that the Stamp can operate 24 hours a day
' from a solar battery and 1-Farad memory-backup capacitor (super
' cap). Every 12 seconds the Stamp wakes up, takes a temperature
' reading from a thermistor, converts it to degrees F and
' transmits it (as a single byte of data) over a 1200-bps
' infrared link.

' Coefficients for the thermistor conversion/linearization routine.
' For more information, see Stamp app note #7.
Symbol co0        = 171                    ' Adjusted to match capacitor.
Symbol co1top     = 255
Symbol co1btm     = 2125
Symbol co2bt1     = 25
Symbol co2top     = 3
Symbol co2btm     = 50

' Change pins 1 and 2 to outputs and take them low. Pin 1 controls
' power to the '555 timer/40-kHz transmitter. Pin 2 is serial output
' to the 40-kHz transmitter.
low 1:low 2

' Main program loop.
Loop:
' Take a thermistor measurement using pot.
        pot 0,46,w0
' Linearize it and convert to degrees F with the equation from
' Stamp application note #6.
        let w1 = w0*w0/co2bt1*co2top/co2btm
        let w0 = w0*co1top/co1btm+w0
        let w0 = co0+w1-w0
' Now turn on the '555 timer and give it a little time to get ready.
        high 1
        pause 100
' Transmit the data.
        serout 2,N1200,(b0)
' Turn off the '555.
        low 1
' Go back to sleep.
        sleep 10
goto Loop
```

```
' Program DATA_LOG.BAS
' This is a QBASIC program to display and record the data
' from the Stamp program SOLAR.BAS. To quit this program,
' either press control-break, or press any key and wait
' for the next Stamp transmission.

DEFINT A-Z
OPEN "com1:1200,N,8,1,CD0,CS0,DS0,OP0" FOR INPUT AS #1
OPEN "c:\data.log" FOR OUTPUT AS #2
CLS
Again:
            temp$ = INPUT$(1, 1)
            PRINT ASC(temp$); CHR$(9); TIME$
            PRINT #2, ASC(temp$); CHR$(9); TIME$
IF INKEY$ = "" THEN GOTO Again
CLOSE
END
```

Introduction. This application note shows how to read multiple switches through a single input/output (I/O) pin by using the *pot* command.

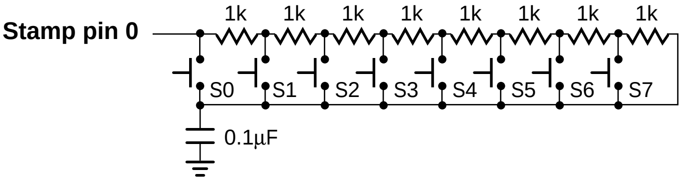
Background. If your BASIC Stamp application needs to check the status of more than a few switches, you have probably considered using external hardware to do the job. The trouble is that most hardware solutions still use more than one I/O pin, and often require considerable program overhead.

Now, consider the pot command. It reads a resistance and returns a proportional number. What if you wired your switches to vary the resistance measured by *pot*? With an appropriate lookup routine, you'd be able to determine which switch was pressed.

That's exactly the method we're going to demonstrate here.

How it works. As the figure shows, we wired up eight switches and eight 1k resistors in a sort of pushbutton-pot arrangement. When no switch is pressed, the circuit's resistance is the sum of all of the resistors in series; 8k. If you press the switch closest to the pot pin (S0), the network is shorted out, so the resistance is 0. Press S1, and the resistance is 1k. And so on.

To see this effect in action, follow these steps: Wire up the circuit in the figure, connect the Stamp to your PC, run STAMP.EXE, and press ALT-P (calibrate pot). Select the appropriate I/O pin; in this case pin 0. The PC screen will display a dialog box showing a suggested value for the pot scale factor—the number that ensures a full-scale response for the connected combination of the resistor(s) and capacitor.



Schematic to accompany program MANY_SW.BAS

Press the space bar to lock in the scale factor. Now the screen displays the actual value returned by the pot command. Press the switches and watch the value change. Write down the scale factor and the numbers returned by pressing S0 through S7. As you do so, you'll notice that the numbers vary somewhat. They tend to be steadier in the lower resistance ranges, and jumpier at higher resistances. Write down the highest number returned for each switch.

Armed with this calibration data, you can write PBASIC code to determine which switch was pressed just by looking at the pot value. The program listing shows an example. We took the numbers recorded in the step above, added a fixed amount to each, and put them in a lookup table. To identify a switch by its resistance value, the program starts searching at the lowest resistance, represented by switch 0. The program asks, "is this resistance less than or equal to the lookup entry for switch 0?" If it is, then switch 0 was pressed; if not, the program increments the switch number and repeats the question until it determines which switch was pressed.

In creating the lookup table, we added 10 to the maximum value for each of the switches. This serves as a safety margin to prevent errors in case the capacitance and resistance values wander a bit with temperature.

This scheme has some drawbacks, mostly related to the way pot works. Pot makes resistance readings by charging up the capacitor, then gradually discharging it through the series-connected pot or resistor. By measuring the time required to discharge the cap, pot can provide a pretty accurate estimate of the relative resistance. This process takes several milliseconds to complete.

If one of the switches is pressed during the pot timing cycle, the rate at which the capacitor discharges will change. The pot measurement will be wrong, and the switch number returned by the program will be wrong. To guard against this, the program ignores the first several readings after an initial switch closure. This isn't completely foolproof, but it makes misidentified switches a relatively rare event.

Another potential drawback is that the program cannot detect more than one switch closure at a time. If two switches are closed at the same time, the program will correctly identify the lower of the two switches. For example, if switches 2 and 5 are both closed, the program will recognize switch 2. You can understand this by analyzing the circuit. Switch 2 effectively shorts out all of the resistor/switch network beyond itself. Additional closed switches have no effect.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: MANY_SW.BAS (Read switches with POT command)

' This program illustrates a method for reading eight switches using
' one I/O pin by using the POT command. The switches are wired as
' shown in the accompanying application note to cut out portions of
' a network of series-connected 1k resistors. The POT command reads
' the resulting resistance value. The subroutine ID_sw compares the
' value to a table of previously determined values to determine
' which switch was pushed.

Clear:                          ' Clear counter that determines how many
let b0 = 0                      ' readings are taken before switch is ID'ed.

Again:
     pot 0,148,b2               ' Take the resistance reading.
     if b2 >= 231 then Clear    ' Higher than 230 means no switch pushed.
     goto ID_sw                 ' Value in range: identify the switch.
Display:
     debug b3                   ' Show the switch number on PC screen.
goto Clear                      ' Repeat.


' ID_sw starts with the lowest switch-value entry in the table (the 0th
' entry) and compares the POT value to it. If the POT value is less than
' or equal, then that's the switch that was pushed. If it's not
' lower, the routine checks the next switch-value entry.

' There's nothing magical about the switch values in the table below. They
' were obtained by pressing the switches and recording their POT
' values, then adding an arbitrary amount--in this case 10. The idea
' was to select numbers that would always be higher than the highest
' POT value returned when the corresponding switch was pressed, but
' lower than the lowest value returned by the next switch. This keeps
' the comparison/search required to identify the switch as simple as
' possible.

ID_sw:
	if b0 > 8 then skip		' Take 8 readings before trying to
	b0 = b0+1			' identify the switch.
	goto Again
skip:
	for b3 = 0 to 7			' Compare table entries to the current reading.
		lookup b3,(10,45,80,114,146,175,205,230),b4
		if b2 <= b4 then done	' Match? Then done.
	next
done:		goto Display		' Switch identified; display its number.
```

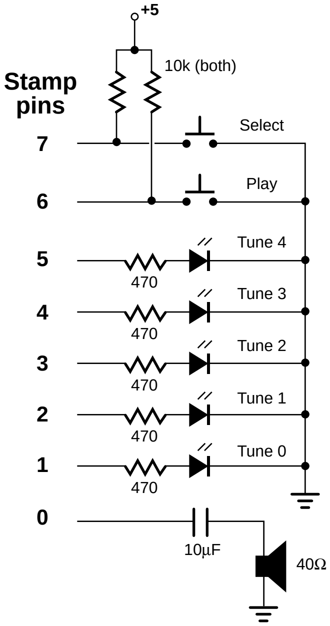
Introduction. This application note explains the *button* command and presents an example program that uses *button* in its immediate and delay/autorepeat modes.

Background. The idea is simple enough—a single command allows your PBASIC programs to read and debounce a switch. However, *button*'s myriad features and its lack of an equivalent in other BASIC dialects have led to considerable misunderstanding among Stamp users. An explanation is in order.

First of all, *button* is intended to be used inside a loop. The idea is that the program goes about its normal business, periodically checking the state of the button. If conditions set up by the button command are met, then the program goes to the address included in the button command.

Second, we should define a slippery term that's important to understanding what button does. That term is "debounce." When you press a switch, the contacts smack into each other with the action of a microscopic earthquake. For several milliseconds they bounce and shudder and grind against one another before finally settling into solid contact. This bouncing shows up as several milliseconds of rapid on/off switching that can be detected by a relatively fast device like the Stamp.

In order to keep switch bounce from seeming like several deliberate switch presses, the button command ignores these very rapid changes in the switch state. So, when we talk about switch "debouncing" in the discussions that follow, we mean button's effort to clean up the switch output.



Schematic to accompany program listings 1 and 2.

Now, let's take a look at the parameters of button in detail. The syntax, as described in the instruction manual, is this:

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

Pin is a variable or constant in the range of 0 to 7 that specifies which pin the button is connected to. Remember to use the number of the pin and not its pin name (e.g., Pin3 or Pins.3). The pin name will return the state of the specified pin (0 or 1), which is probably not what you want.

Downstate is a variable or constant that specifies what state the button will be in (0 or 1) when it is pressed.

Delay is used with button's autorepeat capability. If you hold down the A key on your PC keyboard, there's a small pause before the PC goes into machine-gun mode and starts rapidly filling the screen with AAAAAAA... With button, *delay* sets the length of this pause as a variable or constant number of loops (1 to 254) through the button command. So, if you set delay to 100, your Stamp program must loop through the button command 100 times after the initial press before autorepeat will begin. How long will this take? It depends on your program. If the delay is too short for your taste, insert a short pause in the loop that contains the button command.

You can also use the delay setting to change the way button works. A delay of 0 turns off debounce and autorepeat. A delay of 255 turns on debounce, but turns off autorepeat.

Rate is a variable or constant that specifies how fast the autorepeat will occur. Like delay, it is also specified in terms of the number of loops through the button command.

Bytevariable is button's workspace—a variable in which button stores the current state of the delay or rate counters. Make sure to give each button command you use a different workspace variable, or your buttons will interact in bizarre and undesirable ways. Also make sure that byte variables used by button commands are cleared to 0 before they are first used. After that, button will take care of them.

Targetstate is a variable or constant (0 or 1) that specifies whether the program should take action when the button is pressed (1), or when it's not pressed (0). Why the heck would you want to go to some address when the button isn't pressed? In some cases, it's simpler to skip over part of your program unless the button is pushed. This reverse logic can be a little hard to get used to, but it can help reduce the "spaghetti code" of multiple *goto*s for which BASIC is so frequently condemned.

Address is the program label that you want to go to when all of the conditions set by the button command are met.

To illustrate how all of these parameters make button work, we've designed a sample application called BTN_JUKE.BAS. It's a five-selection jukebox that uses one button to select which tune to play by scrolling through five LEDs, and a second button to trigger the currently selected tune.

How it works. The circuit in the figure should be pretty self-explanatory, but note that the switches are wired so that the Stamp pins see highs (+5 volts) when the switches are open and lows (0 volts) when they're pressed.

Now look at listing 1. The program begins by defining the variable *Select*, which will hold the number of the currently selected tune. It then sets up the pins' I/O directions. It clears both of the byte variables that will be used in the button commands (*b0* and *b1*) at one time by clearing the word variable to which they belong (*w0*). As a final setup step, it turns on the LED corresponding to a selection of 0.

The program then enters its main loop containing the button commands. The first is:

button 7,0,0,0,b0,0,no_play

This command translates to: "Read the button on pin 7. When it is pressed, there will be a logical 0 on the pin. Don't debounce or autorepeat (delay = 0). With autorepeat turned off, rate doesn't matter, so set it to 0. Use *b0* as your byte workspace. When the button is not pressed (0), go to *no_play*."

So as long as the button is not pressed, the button command will skip over the code that plays the selected tune. When the button is pressed, the tune will play.

This button command doesn't require debounce or autorepeat, because the tunes are relatively long. By the time a tune is finished playing, the user has probably already released the button. If he hasn't, the tune will simply play again without delay.

The second button command is:

button 6,0,200,60,b1,1,Pick

This translates to: "Read the button on pin 6. When it is pressed, there will be a logical 0 on the pin. Debounce the switch and delay 200 cycles through this command before starting autorepeat. Once autorepeat begins, delay 60 cycles through button between repeats. Use b1 as a workspace. When the button is pressed (1) go to the label *Pick*."

From the user's standpoint, this means that a single press of the select button lights the next LED in the sequence. Holding down the button makes the LEDs scan rapidly. Releasing the switch causes the currently lit LED to remain on.

It's hard to describe what an important difference debounce and autorepeat make in the ease and quality of a user interface. The best way is to offer a comparison. Listing 2 is the same jukebox program as listing 1, but without the button command to debounce the switches. It uses the same circuit as listing 1, so you can alternately download the two programs for an instant comparison.

When you run NO_BTN.BAS, you'll find no difference in the operation of the play button. Remember that button's debounce and autorepeat features were turned off in the original program anyway. If you need to economize on variables, you can substitute a simple *if/then* for button in cases that don't use these features.

The select button is a different story. It becomes almost impossible to select the LED you want. To make the comparison fair, we even added a brief pause to the *Pick* routine as a sort of debouncing. It helps, but not enough to make the button action feel solid and predictable. This is the kind of case that requires *button*.

Program listing. These programs may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

Listing 1: BTN_JUKE.BAS (Demonstration of the Button command)

```
' The Stamp serves as a tiny jukebox, allowing you to pick from one of
' five musical (?) selections created with the sound command. The point
' of the program is to demonstrate the proper way to use the button
' command. The juke has two buttons--one that lets you pick the tune
' by "scrolling" through five LEDs, and the other that plays the tune
' you selected. The selection button uses the debounce and autorepeat
' features of button, while the play button is set up for immediate
' response without delay or autorepeat.

SYMBOL Select = b2              ' Variable to hold tune selection, 0-4.

let dirs = %00111111            ' Pins 6 & 7 are inputs for buttons.
let w0 = 0                      ' Initialize all variables to zero
let w1 = 0                      ' (includes clearing the button variables b0,b1)
let pins = %00000010            ' Turn on the first selection LED.

' The main program loop. Main scans the two buttons and branches to
' no_play or Pick, depending on which button was pressed. Note the two
' different ways the button command is used. In the first case,
' button skips over the branch instruction that jumps to the
' appropriate tune routine _unless_ the button is pushed.
' The tunes are fairly long, so no debounce is needed, and
' autorepeat isn't appropriate (the next trip through main will
' play the tune again, anyway). The second button command, which
' scrolls through the selection LEDs, uses both debounce and auto-
' repeat. Switch bounce could cause the display to seem to skip
' over selections, and autorepeat is a nice, professional touch
' for rapidly scrolling through the display.

Main:
  button 7,0,0,0,b0,0,no_play   ' Don't play tune unless button is pushed.
  branch Select,(Tune0,Tune1,Tune2,Tune3,Tune4)
no_play:
  button 6,0,200,60,b1,1,Pick   ' When button is pushed, change selection.
goto Main
```

```
' Pick increments the variable Selection, while limiting it to a maximum
' value of 4. If Selection exceeds 4, the code resets it to 0.

Pick:
  let Select = Select + 1          ' Increment selection.
  if Select < 5 then skip          ' If Select = 5, then Select = 0.
  let Select = 0                   ' Skip this line if Select is < 3.
skip:
  lookup Select,(2,4,8,16,32),pins                ' Light appropriate LED.
goto Main                                         ' Return to main program loop.

' The tunes. Not necessarily music.

Tune0: sound 0,(100,10,110,100): goto main

Tune1: sound 0,(98,40,110,10,100,40): goto main

Tune2: sound 0,(100,10,80,100): goto main

Tune3: sound 0,(100,10,110,50,98,10): goto main

Tune4: sound 0,(98,40,100,10,110,40): goto main
```

' Listing 2: NO_BTN.BAS (Demonstration of poor debouncing)

```
' This program is identical to BTN_JUKE.BAS, except that it does not
' use button commands to read the state of the switches. Contrasting
' the operation of this program to BTN_JUKE will give you a good idea
' of the benefits of button.

' The Stamp serves as a tiny jukebox, allowing you to pick from one of
' five musical (?) selections created with the sound command. The point
' of the program is to demonstrate the proper way to use the button
' command. The juke has two buttons--one that lets you pick the tune
' by "scrolling" through five LEDs, and the other that plays the tune
' you selected.


SYMBOL Select = b2                 ' Variable to hold tune selection, 0-4.

let dirs = %00111111               ' Pins 6 & 7 are inputs for buttons.
let b2 = 0                         ' Clear the selection.
let pins = %00000010               ' Turn on the first selection LED.

' The main program loop. Main scans the two buttons and takes the
' appropriate action. If the play button on pin 7 is not pressed,
' the program skips over the code that plays a tune. If the select
' button is pressed, the program goes to the routine Pick, which
' increments the current tune selection and LED indicator.

Main:
  if pin7 = 1 then no_play          ' Don't play tune unless pin 7 button is pushed.
  branch Select,(Tune0,Tune1,Tune2,Tune3,Tune4)
no_play:
  if pin6 = 0 then Pick             ' When pin 6 button is pushed, change tune.
goto Main

' Pick increments the variable Selection, while limiting it to a maximum
' value of 4. Note that it begins with a pause of 0.15 seconds. This
' prevents the code from executing so fast that the LEDs become a blur.
' However, it's no substitute for the button command. You'll find that
' it is hard to select the particular LED you want.

Pick:
  pause 150                         ' Attempt to debounce by delaying .15 sec.
  let Select = Select + 1           ' Increment selection.
  if Select < 5 then skip           ' If Select = 5, then Select = 0.
  let Select = 0                    ' Skip this line if Select is < 3.
skip:
  lookup Select,(2,4,8,16,32),pins                  ' Light appropriate LED.
goto Main                                           ' Return to main program loop.

' The tunes. Not necessarily music.

Tune0: sound 0,(100,10,110,100): goto main

Tune1: sound 0,(98,40,110,10,100,40): goto main

Tune2: sound 0,(100,10,80,100): goto main

Tune3: sound 0,(100,10,110,50,98,10): goto main

Tune4: sound 0,(98,40,100,10,110,40): goto main
```

Introduction. This application note describes an inexpensive and accurate timebase for Stamp applications.

Background. The Stamp has remarkable timing functions for dealing with microseconds and milliseconds, but it stumbles a little when it comes to minutes, hours, and days.

The reason for this is twofold: First, the Stamp's ceramic resonator timebase is accurate to about ±1 percent, so the longer the timing interval, the larger the error. A clock that was off by 1 percent would gain or lose almost 15 minutes a day.

Second, Stamp instructions take varying amounts of time. For example, the *Pot* command reads resistance by measuring the length of time required to discharge a capacitor. The higher the resistance, the longer *Pot* takes. The math operators also take varying amounts of time depending on the values supplied to them.

The result is that even the most carefully constructed long-term timing programs end up being less accurate than a cheap clock.

An obvious cure for this might be to interface a real-time clock to the Stamp. Available units have all kinds of neat features, including calendars with leap-year compensation, alarms, etc. The trouble here is that once you write a program to handle their synchronous serial interfaces, acquire the time from the user, set the clock, read the time and convert

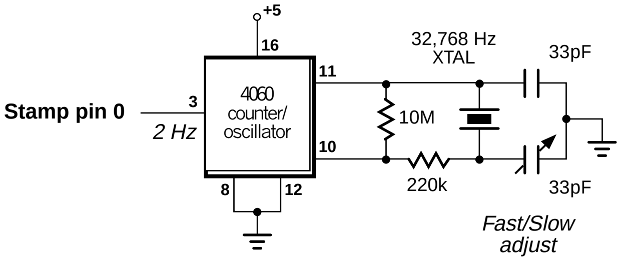


Figure 1. Schematic to accompany TIC_TOC.BAS

it to a usable form, you have pretty much filled the Stamp's EEPROM. A compromise approach is to provide the Stamp with a very accurate source of timing pulses, and let your program decide how to use them. The circuit and example program presented here do just that. For this demonstration, the Stamp counts the passing seconds and displays them using *debug*.

How it works. The circuit in figure 1 shows how to construct a crystal-controlled, 2-pulse-per-second timebase from a common digital part, the CD4060B. This part costs less than $1 from mail-order companies like the one listed at the end of this note. The 32,768-Hz crystal is also inexpensive, at just over 50 cents.

The 4060 is a 14-stage binary counter with an onboard oscillator. Although the oscillator can be used with a resistor/capacitor timing circuit, we're going for accuracy; hence the crystal. Why 32,768 Hz and not some other value, like 1 MHz? It just happens that $32{,}768 = 2^{15}$, so it's easy to use a binary counter like the 4060 to divide it down to easy fractions of one second. Since the 4060 is a 14-stage counter, the best it can do is divide by 2^{14}. The program further divides the resulting twice-a-second pulses to produce one count per second.

Take a look at the program listing. It consists of a main loop and a routine to increment the clock. In an actual application, the main loop would contain most of the program instructions. For accurate timing, the instructions within the main loop must take less than 250 milliseconds total. Even with the timing problems we've discussed, that's pretty easy to do.

Let's walk through the program's logic. In the main loop, the program compares the state of *pin0* to *bit0*. If they're equal (both 0 or both 1) it jumps to the *tick* routine.

In *tick*, the program toggles *bit0* by adding 1 to the byte it belongs to, *b0*. This makes sure that *bit0* is no longer equal to the state of *pin0*, so the program won't return to *tick* until *pin0* changes again.

B0 also serves as a counter. If it is less than 4, the program returns to the main loop. When *b0* reaches 4, *tick* clears it, adds 1 to the running total of seconds, displays the number of seconds on the screen, and jumps back to the main loop.

This is pretty elementary programming, but there's one detail that may be bothering you: If we're using a 2-Hz timebase, why count to 4 before incrementing the seconds? The reason is that we're counting transitions—changes in the state of *pin0*—not cycles. Figure 2 shows the difference.

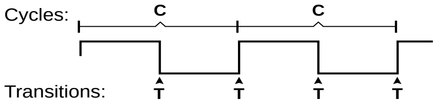


Figure 2.

This stems from our use of *bit0* to track changes in the timing pulses. As soon as *pin0 = bit0*, we drop into *tick* and toggle the state of *bit0*. This keeps us from visiting *tick* more than once during the same pulse. The next time *pin0* changes—the next transition—*pin0 = bit0*, and *tick* executes again. A side effect of this approach is that we increment the counter twice per cycle.

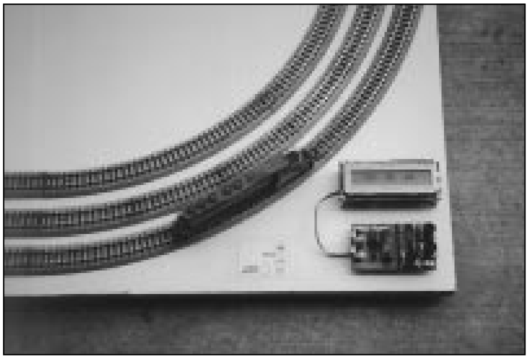
Construction notes. The circuit in figure 1 draws only about 0.5 mA, so you can power it from the Stamp's +5V supply without any problem. The resistor and capacitor values shown are a starting point, but you may have to adjust them somewhat for most reliable oscillator startup and best frequency stability. You may substitute a fixed capacitor for the adjustable one shown, but you'll have to determine the best value for accurate timing. The prototype was right on the money with a 19-pF capacitor, but your mileage may vary due to stray capacitance and parts tolerances.

Parts source. The CD4060B and crystal are available from Digi-Key (800-344-4539) as part numbers CD4060BE-ND and SE3201, respectively.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: TIC_TOC.BAS  (Increment a counter in response to a
' precision 2-Hz external clock.)

' The 2-Hz input is connected to pin0. Bit0 is the lowest bit of b0,
' so each time b0 is incremented (in tick), bit0 gets toggled. This
' ensures that tick gets executed only once per transition of pin0.

Main:
        if pin0 = bit0 then tick
          ' Other program activities--
          ' up to 250 ms worth--
          ' go here.
        goto Main

' Tick maintains a 16-bit counter to accumulate the number of seconds.
' The maximum time interval w1 can hold is 65535 seconds--a bit over
' 18 hours. If you want a minute count instead, change the second
' line of tick to read: "if b0 < 240 then Main". There are 1440 minutes
' in a day, so w1 can hold up to 65535/1440 = 45.5 days worth of to-the-
' minute timing information.

tick:
        let b0 = b0 + 1                 ' Increment b0 counter.
        if b0 < 4 then Main             ' If b0 hasn't reached 4, back to Main.
        let b0 = 0                      ' Else clear b0,
        let w1 = w1 + 1                 '  increment the seconds count,
        debug cls,#w1," sec."           '  and display the seconds.
        goto Main                       ' Do it again.
```

Introduction. This application note describes a simple model train project that we showed at the Embedded Systems Conference in 1994. The project uses a Stamp to control the speeds of three N-scale trains. The speeds are displayed on an LCD display, and can be changed using three buttons: *track select*, *up*, and *down*.



The completed Stamp-controlled train set (buildings and trees were added later).

Background. Several months before the Conference, we decided that we should have an interesting example of the Stamp's capabilities. We determined that it should be something physical, something simple, and something that people would relate to. We looked at various toys, including Legos, Erector Sets, electric race cars, and model trains. They all had their good and bad points, but we finally decided upon model trains. I always liked model trains as a child, and I was the one who had to build it, anyway.

Trains are somewhat simple to control and detect, and many people like them (more than we expected). The only drawback was the amazingly high cost of constructing a complete train set. A complete train set, with three loops of track, three trains, several buildings, and lots of trees, cost about $700! (the trains my parents bought were much less expensive). It didn't seem like that much, because I purchased the track one day, and the engines a week later, and the buildings a week later, etc. But the bookkeeper was keeping track, and indeed the simple train display was getting expensive. But, alas, it was too late to go back.

Having decided upon a train set, I had the harder task of deciding what to do with it. I had some really neat ideas, like having a Stamp in each engine, thus making each train intelligent. With a brain in each train, I would then have infrared "markers" along the track, so the trains would know their position and act accordingly. In fact, perhaps they could even communicate with a master Stamp, which could then modify their path and communicate with other trains. The possibilities were endless, especially since I hadn't run into reality, yet.

After some humbling thought, I tapered my ideas to a simple two-part goal: to control the speed of three trains, and to detect the position of the trains. I didn't know exactly how to accomplish these goals, but they seemed possible. I knew that high-current drivers existed, and could be used to run the trains. As for detecting the trains, my thoughts ranged from LED/detector pairs to Hall-effect sensors. The Hall-effect sensors seemed better, since they could be hidden (LEDs would be too obvious).

Preliminary research. Not knowing much about high-current drivers, I called Scott Edwards. He knows something about everything, and he was happy to fax a pinout of the Motorola ULN2803. The Motorola chip is an 18-pin device described as an "octal high-voltage, high-current Darlington transistor array." Other people refer to it as a "low-side driver," since it's used to drive the low (GND) side of a load. Each driver can sink 500 mA, and as you might guess from the word "octal" in the name, the ULN2803 has eight separate drivers, so you can really drive a lot of current with one chip. The chip even has internal clamping diodes to suppress transients that occur when "noisy" devices turn on and off ("noisy" devices include motors, relays, solenoids, etc.). Without diodes to suppress transients, the digital control circuitry (in this case, the Stamp) may go crazy (I think this is caused by fluctuations on the I/O pins and/or power pins). In any case, the ULN2803 makes a previously messy task very clean, simple, and inexpensive (the chips are under $1).

As for Hall-effect sensors, I ordered a selection from Digi-Key and then went to Radio Shack to buy some magnets. If you're not familiar with them, Hall-effect sensors are 3-pin, transistor-sized devices that sense magnetic fields. They sense the presence of a north or south magnetic field, depending on the individual sensor's design. Some even act as a mechanical switch: they trigger when a magnetic field is present, and then remain activated until power is removed. Others remain active until they sense another magnetic field of the same or opposite polarity. And all of the ones I found had TTL-level outputs, which was perfect for the Stamp. All in all, if you need to sense a magnetic field, there's probably one for you.

For me, the only question was: will the train's engine generate a strong enough field to trigger the sensor? After all, I wanted to place the sensor under the track, or even under the wooden board on which the train set was built. This would place the sensor 0.25 to 0.75 inches from the underside of the train. Unfortunately, the train didn't produce an adequate field at any distance, no matter how small. So, I purchased a selection of "super strong" magnets at a nearby electronics store. These small magnets were strong enough to keep themselves secured to my hand by placing one in my palm and the other on the back of my hand (fairly impressive, since my hand is at least an inch thick). And they were strong enough to activate the most sensitive Hall-effect sensor through the track *and the wooden board!* This was great, because placing the sensors on the "back" of the train set would be much easier than drilling holes in the board.

Starting construction. Having done a little preliminary research, it was time to start making something. It seemed logical to construct the basic track layout first, and then start integrating the Stamp. So, I constructed a simple layout of three oval tracks. The distance separating each track from the next was about half an inch. I thought this closeness would look attractive when all three trains were running at the same time; I even reversed the polarity of the middle track, just to make the display look especially interesting (all trains going the same direction might get boring).

With the physical layout complete, I turned to speed control. I remembered that the ULN2803 was used on our Stamp Experiment Board, so I used the experiment board for initial testing. Using a handful of micrograbber cables, I quickly connected the on-board Stamp circuit to the ULN2803 and then to the first loop of track. And, of course, nothing worked. I examined the circuit for several hours, and discovered two or three stupid mistakes. The mistakes were truly stupid (like missing ground connections), but one of them reminded me of why the ULN2803 is called a low-side driver: the driver provides a switchable ground, so my circuit must therefore provide a constant "supply" voltage to one side of the tracks (in this case, 12 VDC). With the minor bugs corrected, it worked, or at least somewhat. I hadn't written much code, so the necessary PWM routines weren't in place to vary the train's speed. However, I could toggle a Stamp I/O pin, which drove the ULN2803, which powered the train. A miracle was upon us (at least for me): the BASIC Stamp could make the train start and stop.

A foundation was forming, but there were still basic human-interface questions (how many buttons would control the system?, would there be an LCD display?, etc.). I decided upon the following design:

- Three buttons (track select, up, down)
- LCD display for track speeds

I ordered a selection of push-buttons from Digi-Key and called Scott about his new serial LCD module (it was new at the time). He had designed a 1x16 character LCD which was controlled with one line (plus power and ground). The serial LCD was a godsend, because I was running out of I/O lines on the Stamp. Controlling the track voltages took three lines, and the buttons were going to take three more. This left only two unused I/O lines, which would usually fall short of the six lines required to drive a regular intelligent LCD. But, again, the serial LCD saved the day. With the tracks, buttons, and LCD, I had one I/O line left unused.

The buttons arrived the next day, and I chose the ones that seemed best for the job (large button, small footprint). I soldered the ULN2803 and three buttons onto a BASIC Stamp, and then connected the Stamp to the train set.

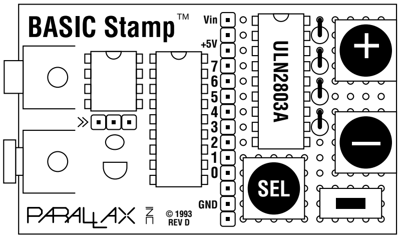
Programming custom PWM. It was time to do some real BASIC programming. Earlier, when Scott sent the ULN2803 data, he also included some routines to make the Stamp perform "custom" pulse-width-modulation (PWM). The Stamp has a built-in PWM command, but it's meant for purposes other than driving the ULN2803. To control the speed of the trains, I would need to write a program that pulsed the voltage to the tracks. Instead of varying the voltage to the tracks, which would require more complex hardware, the Stamp could simply pulse the tracks with a set voltage. Pulse-width-modulation has that name because you are varying, or *modulating*, the width of a pulse. If the pulse is on half the time and off half the time, then you have a duty cycle of 0.5, which would theoretically make the train run at half speed (of course, the engine's performance is probably not linear). Using Scott's example as a guide, I wrote a subroutine that pulsed all three tracks according to the speed set by the user. I still don't fully understand real PWM, but the basic theory of the train routine makes sense:

- A counter (or *accumulator*) is maintained for each track.
- The user sets a speed (0-99) for each track.
- For every pass through the PWM routine, the speed is added to the accumulator. The high bit (bit 7) of the accumulator is then copied to the I/O pin for the appropriate track. If the bit is a '1', then the train will receive power; if the bit is a '0', then power is removed.
- The Stamp executes the PWM routine many times per second, so the train receives a number of 'on' and 'off' states. A higher speed value causes the accumulator to overflow more often, which results in more frequent 'on' states. If power to the tracks is 'on' more often, then the trains move more quickly. If power is 'off' more often, then the trains slow down.

Connecting push-buttons. With the PWM routine working relatively well, it was time to move on to other concerns. The push-buttons and LCD were waiting to be used. The buttons seemed like the obvious thing to work on next.

The BASIC Stamp has a particularly handy instruction called BUTTON, which is used to read push-buttons and perform otherwise tedious functions, such as debounce and auto-repeat. Debounce is necessary to convert one button press into one electrical pulse (many pulses actually occur when the button's contacts are closed); auto-repeat allows the user to hold down the button and have the system act as if he were repeatedly pressing the button (most computer keyboards do this). I had never used the BUTTON instruction before, but it was relatively simple to experiment with and understand.

But, I noticed that the buttons seemed unstable; sometimes the Stamp would act as if I were still pressing a button long after I had stopped. Then I realized that I had forgotten pull-up resistors on the button inputs. In my circuit, when a button was pressed, it connected the associated I/O pin to ground, which read as ‘0’ to the BASIC program. However, when the button was not pressed, the I/O pin would “float,” since it wasn’t connected to anything. Since the pin was floating, it would randomly read as ‘0’ or ‘1’. This was solved by adding pull-up resistors to the button inputs; the resistors provide a weak connection to the 5-volt supply, so the inputs read as ‘1’ when their buttons are not pressed. The last step involving the buttons was to adjust the auto-repeat rate until it seemed right (not too fast, not too slow). The repeat rate is controlled by one of the values given in the BUTTON instruction, so it just took a few quick downloads to arrive at the right value.



Completed Stamp with ULN2803 and buttons

Connecting the LCD display. With the buttons working, the next item was the LCD. I connected the LCD to the Stamp and then entered the sample program provided with the LCD. After some minor troubleshooting, the LCD worked, and worked well! Printing text was almost as easy as using the normal PRINT instruction found in other versions of BASIC. But, since the Stamp has no PRINT instruction, the LCD is controlled with simple SEROUT instructions. For instance, SEROUT 0,N2400,(“hello”) prints the word “hello” on an LCD module connected to pin 0.

I wanted the LCD to display something fancy, but reality came into play for two reasons: 16 characters isn't that much, especially if you want to display three speeds, and I was quickly running out of program space in the Stamp. So, I decided upon a simple display of the track speeds, as shown below:

```
>00   00   00
```

The pairs of digits represent the speed of the trains, and the arrow indicates which train is currently selected (pressing the up and down buttons affects the speed of the currently selected train). This arrangement was simple to operate, and made good use of available resources.

Streamlining the program. After a day or so, I had a program that was nearly finished; it read the buttons, updated the LCD, and ran the trains. But, as I finished the LCD routine, I noticed that the performance of the trains was getting progressively worse. The trains had run smoothly before, even at slow speeds, but now they were very jerky, even at medium and fast speeds. The problem was that the LCD took a fairly long time to update. Updating the LCD meant sending 22 bytes of data, which took about 0.1 seconds. One-tenth of a second isn't much to us, but it's an eternity to the Stamp, and was quite noticeable in the trains. I spent the evening making the program more efficient, which resulted in more acceptable operation. The two changes that really helped were:

- Updating the LCD only when something changed, which looks better, anyway (less flicker).
- Calling the train PWM routine several times from within the LCD routine.

I was finally nearing the end of the project. I did a few downloading tests, and realized that I only had a few more bytes of program space in the Stamp.

There was one more function I wanted: the trains derailed a few times while running continuously, so I felt that a panic button would be a good idea. The purpose of the button would be to stop the trains in the event of an accident. Without the panic button, the operator would have to set each speed to zero, which would take some time (I imagined trains strewn about the board). The panic button was easy: all I needed to add was a single line in the beginning of the main loop, which would check the panic button and jump to an earlier line that set the speeds to zero (something that the program did upon start-up). This seemed straightforward, but it proved to be more difficult than I thought. The concept was fine, but I was short a few bytes of program space.

A few more bytes. Squeezing a few more bytes out of my program was painfully difficult. Finally, everything did fit, but only after resorting to extreme measures. For instance, if you look at the first few lines of code, you'll see the following:

```
symbol track1_speed=b2
symbol track1_accum=b1

symbol track2_speed=b3
symbol track2_accum=b7

symbol track3_speed=b4
symbol track3_accum=b6

symbol current_track = b5
.
.
.
reset: w1 = 0: w2 = 0
```

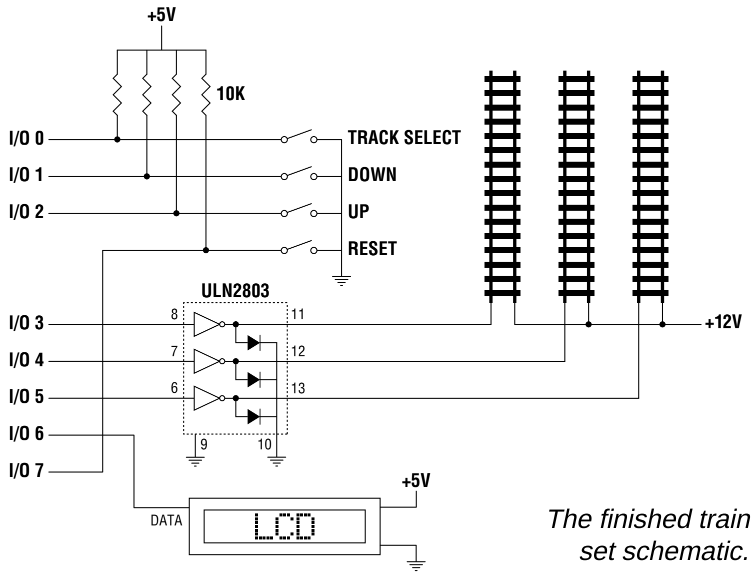
You might wonder why I didn't just use the variables b1-b7 in order, which is how I originally had them. The order shown seems random, but it actually saves program space later. The last line shown resets the track speeds and current track variable. The word variable w1 includes b2 and b3, and the word variable w2 includes b4 and b5. So, by clearing two word variables, the program clears four byte variables, which saves a byte or two of program space.

If you're really wondering about variable allocation, you might also wonder why the program doesn't store anything in b0. This is because b0 has a special role in the train PWM routine:

```
.
.
.
b0 = track1_accum
pin3 = bit7
```

In this piece of code, b0 is loaded with the accumulator for track #1. You may recall that the high bit (bit 7) of the accumulator is used to drive the track. But, how do we isolate the high bit? The easiest way, at least on the Stamp, is to copy the 8-bit accumulator value into b0, and then use the unique bit-addressable quality of b0 to drive the track I/O pin. The statement pin3 = bit7 means *make pin 3 the same state as bit 7 of b0*. The only variable that's bit-addressable is b0, so it should be saved for such cases.

Conclusion. In the end, the train project was fun and educational. It wasn't nearly as elaborate as I originally intended, but it was a good example of what the BASIC Stamp could do. We now offer a larger Stamp, which would have made the programming portion much easier. But, it wouldn't have been nearly as much fun.



The finished train set schematic.

```
' Program: TRAIN.BAS
' Uses simple 4-button interface and LCD to control voltages to three N-scale trains.


symbol track1_speed=b2                        'set up variable names
symbol track1_accum=b1

symbol track2_speed=b3
symbol track2_accum=b7

symbol track3_speed=b4
symbol track3_accum=b6

symbol current_track = b5

symbol track_btn = b8
symbol up_btn = b9
symbol down_btn = b10

pause 2000                                    'wait for lcd to wake up

serout 6,n2400,(254,1,254," ")                'clear lcd

dirs = %00111000                              'make track driver pins
                                              'outputs; all others are
                                              'inputs

reset:    w1 = 0: w2 = 0                      'set track speeds and
                                              'current track # to 0

goto update_lcd                               'update lcd

main_loop:

          if pin7 = 0 then reset              'reset everything if
                                              'reset button is pressed

          gosub run_trains                    'update track pwm

track:    button 0,0,30,6,track_btn,0,down0   'read track select button

          current_track = current_track + 1   'increment current track #
          if current_track <> 3 then go_lcd
          current_track = 0                   'reset if over 2
go_lcd:   goto update_lcd                     'update lcd

down0:    button 1,0,30,1,down_btn,0,up0      'read down button
```

Program listing: *As with the other application notes, this program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.*

```
          if current_track <> 0 then down1      'check current track #
          if track1_speed = 0 then up0
          track1_speed = track1_speed - 1       'reduce track 1 speed
          goto update_lcd                       'update lcd

down1:    if current_track <> 1 then down2      'check current track #
          if track2_speed = 0 then up0
          track2_speed = track2_speed - 1       'reduce track 2 speed
          goto update_lcd                       'update lcd

down2:    if track3_speed = 0 then up0
          track3_speed = track3_speed - 1       'reduce track 3 speed
          goto update_lcd                       'update lcd

up0:      button 2,0,30,1,up_btn,0,main_loop    'read up button

          if current_track <> 0 then up1        'check current track #
          if track1_speed = 99 then main_loop
          track1_speed = track1_speed + 1       'increase track 1 speed
          goto update_lcd                       'update lcd

up1:      if current_track <> 1 then up2        'check current track #
          if track2_speed = 99 then main_loop
          track2_speed = track2_speed + 1       'increase track 2 speed
          goto update_lcd                       'update lcd

up2:      if track3_speed = 99 then main_loop
          track3_speed = track3_speed + 1       'increase track 3 speed

update_lcd:

          serout 6,n2400,(254,130,254," ")      'move cursor and print " "
          if track1_speed > 9 then abc          'test for 1 or 2 digits
          serout 6,n2400,("0")                  'print leading zero
abc:      serout 6,n2400,(#track1_speed)        'print track 1 speed

          gosub run_trains                      'update track pwm

          serout 6,n2400,(254,134,254," ")      'move cursor and print " "
          if track2_speed > 9 then abc2         'test for 1 or 2 digits
          serout 6,n2400,("0")                  'print leading zero
abc2:     serout 6,n2400,(#track2_speed)        'print track 2 speed

          gosub run_trains                      'update track pwm

          serout 6,n2400,(254,138,254," ")      'move cursor and print " "
          if track3_speed > 9 then abc3         'test for 1 or 2 digits
          serout 6,n2400,("0")                  'print leading zero
abc3:     serout 6,n2400,(#track3_speed)        'print track 3 speed
```

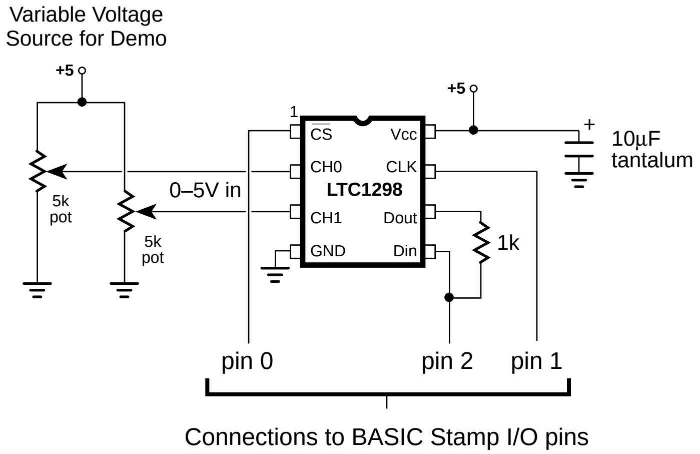
```
                gosub run_trains                'update track pwm

done:           b0 = current_track * 4 + 130    'print arrow pointing to
                serout 6,n2400,(254,b0,254,">") 'currently selected track

goto main_loop

run_trains:

                'update track 1 pwm
                track1_accum = track1_accum + track1_speed
                b0 = track1_accum
                pin3 = bit7                     'drive track 1
                track1_accum = track1_accum & %01111111

                'update track 2 pwm
                track2_accum = track2_accum + track2_speed
                b0 = track2_accum
                pin4 = bit7                     'drive track 2
                track2_accum = track2_accum & %01111111

                'update track 3 pwm
                track3_accum = track3_accum + track3_speed
                b0 = track3_accum
                pin5 = bit7                     'drive track 3
                track3_accum = track3_accum & %01111111

                return
```

Introduction. This application note shows how to interface the LTC1298 analog-to-digital converter (ADC) to the BASIC Stamp.

Background. Many popular applications for the Stamp include analog measurement, either using the Pot command or an external ADC. These measurements are limited to eight-bit resolution, meaning that a 5-volt full-scale measurement would be broken into units of 5/256 = 19.5 millivolts (mV).

That sounds pretty good until you apply it to a real-world sensor. Take the LM34 and LM35 temperature sensors as an example. They output a voltage proportional to the ambient temperature in degrees Fahrenheit (LM34) or Centigrade (LM35). A 1-degree change in temperature causes a 10-mV change in the sensor's output voltage. So an eight-bit conversion gives lousy 2-degree resolution. By reducing the ADC's range, or amplifying the sensor signal, you can improve resolution, but at the expense of additional components and a less-general design.

The easy way out is to switch to an ADC with 10- or 12-bit resolution. Until recently, that hasn't been a decision to make lightly, since more bits = more bucks. However, the new LTC1298 12-bit ADC is reasonably priced at less than $10, and gives your Stamp projects two channels



Schematic to accompany LTC1298.BAS

of 1.22-mV resolution data. It's packaged in a Stamp-friendly 8-pin DIP, and draws about 250 microamps (µA) of current.

How it works. The figure shows how to connect the LTC1298 to the Stamp, and the listing supplies the necessary driver code. If you have used other synchronous serial devices with the Stamp, such as EEPROMs or other ADCs described in previous application notes, there are no surprises here. We have tied the LTC1298's data input and output together to take advantage of the Stamp's ability to switch data directions on the fly. The resistor limits the current flowing between the Stamp I/O pin and the 1298's data output in case a programming error or other fault causes a "bus conflict." This happens when both pins are in output mode and in opposite states (1 vs. 0). Without the resistor, such a conflict would cause large currents to flow between pins, possibly damaging the Stamp and/or ADC.

If you have used other ADCs, you may have noticed that the LTC1298 has no voltage-reference (Vref) pin. The voltage reference is what an ADC compares its analog input voltage to. When the analog voltage is equal to the reference voltage, the ADC outputs its maximum measurement value; 4095 in this case. Smaller input voltages result in proportionally smaller output values. For example, an input of 1/10th the reference voltage would produce an output value of 409.

The LTC1298's voltage reference is internally connected to the power supply, Vcc, at pin 8. This means that a full-scale reading of 4095 will occur when the input voltage is equal to the power-supply voltage, nominally 5 volts. Notice the weasel word "nominally," meaning "in name only." The actual voltage at the +5-volt rail of the full-size (pre-BS1-IC) Stamp with the LM2936 regulator can be 4.9 to 5.1 volts initially, and can vary by 30 mV.

In some applications you'll need a calibration step to compensate for the supply voltage. Suppose the LTC1298 is looking at 2.00 volts. If the supply is 4.90 volts, the LTC1298 will measure (2.00/4.90) * 4095 = 1671. If the supply is at the other extreme, 5.10 volts, the LTC1298 will measure (2.00/5.10) * 4095 = 1606.

How about that 30-mV deviation in regulator performance, which cannot be calibrated away? If calibration makes it seem as though the LTC1298 is getting a 5.000-volt reference, a 30-mV variation means that the reference would vary 15 mV high or low. Using the 2.00-volt example, the LTC1298 measurements can range from (2.00/4.985) * 4095 = 1643 to (2.00/5.015) * 4095 = 1633.

The bottom line is that the measurements you make with the LTC1298 will be only as good as the stability of your +5-volt supply.

The reason the manufacturer left off a separate voltage-reference pin was to make room for the chip's second analog input. The LTC1298 can treat its two inputs as either separate ADC channels, or as a single, differential channel. A differential ADC is one that measures the voltage difference between its inputs, rather than the voltage between one input and ground.

A final feature of the LTC1298 is its sample-and-hold capability. At the instant your program requests data, the ADC grabs and stores the input voltage level in an internal capacitor. It measures this stored voltage, not the actual input voltage.

By measuring a snapshot of the input voltage, the LTC1298 avoids the errors that can occur when an ADC tries to measure a changing voltage. Without going into the gory details, most common ADCs are *successive approximation* types. That means that they zero in on a voltage measurement by comparing a guess to the actual voltage, then determining whether the actual is higher or lower. They formulate a new guess and try again. This becomes very difficult if the voltage is constantly changing! ADCs that aren't equipped with sample-and-hold circuitry should not be used to measure noisy or fast-changing voltages. The LTC1298 has no such restriction.

Parts source. The LTC1298 is available from Digi-Key (800-344-4539) for $8.89 in single quantities (LTC1298CN8-ND). Be sure to request a data sheet or the data book (9210B-ND, $9.95) when you order.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: LTC1298.BAS (LTC1298 analog-to-digital converter)
' The LTC1298 is a 12-bit, two-channel ADC. Its high resolution, low
' supply current, low cost, and built-in sample/hold feature make it a
' great companion for the Stamp in sensor and data-logging applications.
' With its 12-bit resolution, the LTC1298 can measure tiny changes in
' input voltage; 1.22 millivolts (5-volt reference/4096).

' ============================================================
'                 ADC Interface Pins
' ============================================================

' The 1298 uses a four-pin interface, consisting of chip-select, clock,
' data input, and data output. In this application, we tie the data lines
' together with a 1k resistor and connect the Stamp pin designated DIO
' to the data-in side of the resistor. The resistor limits the current
' flowing between DIO and the 1298's data out in case a programming error
' or other fault causes a "bus conflict." This happens when both pins are
' in output mode and in opposite states (1 vs 0). Without the resistor,
' such a conflict would cause large currents to flow between pins,
' possibly damaging the Stamp and/or ADC.

SYMBOL   CS = 0          ' Chip select; 0 = active.
SYMBOL   CLK = 1         ' Clock to ADC; out on rising, in on falling edge.
SYMBOL   DIO_n = 2       ' Pin _number_ of data input/output.
SYMBOL   DIO_p = pin2    ' Variable_name_ of data input/output.
SYMBOL   ADbits = b1     ' Counter variable for serial bit reception.
SYMBOL   AD = w1         ' 12-bit ADC conversion result.

' ============================================================
'                 ADC Setup Bits
' ============================================================

' The 1298 has two modes. As a single-ended ADC, it measures the
' voltage at one of its inputs with respect to ground. As a differential
' ADC, it measures the difference in voltage between the two inputs.
' The sglDif bit determines the mode; 1 = single-ended, 0 = differential.
' When the 1298 is single-ended, the oddSign bit selects the active input
' channel; 0 = channel 0 (pin 2), 1 = channel 1 (pin 3).
' When the 1298 is differential, the oddSign bit selects the polarity
' between the two inputs; 0 = channel 0 is +, 1 = channel 1 is +.
' The msbf bit determines whether clock cycles _after_ the 12 data bits
' have been sent will send 0s (msbf = 1) or a least-significant-bit-first
' copy of the data (msbf = 0). This program doesn't continue clocking after
' the data has been obtained, so this bit doesn't matter.

' You probably won't need to change the basic mode (single/differential)
' or the format of the post-data bits while the program is running, so
' these are assigned as constants. You probably will want to be able to
' change channels, so oddSign (the channel selector) is a bit variable.
```

```
SYMBOL    sglDif = 1          ' Single-ended, two-channel mode.
SYMBOL    msbf = 1            ' Output 0s after data transfer is complete.
SYMBOL    oddSign = bit0      ' Program writes channel # to this bit.


' ==============================================================
'                 Demo Program
' ==============================================================

' This program demonstrates the LTC1298 by alternately sampling the two
' input channels and presenting the results on the PC screen using Debug.

high CS                       ' Deactivate the ADC to begin.
Again:                        ' Main loop.
  For oddSign = 0 to 1        ' Toggle between input channels.
    gosub Convert             ' Get data from ADC.
    debug "ch ",#oddSign,":",#AD,cr  ' Show the data on PC screen.
    pause 500                 ' Wait a half second.
  next                        ' Change input channels.
goto Again                    ' Endless loop.

' ==============================================================
'                 ADC Subroutine
' ==============================================================

' Here's where the conversion occurs. The Stamp first sends the setup
' bits to the 1298, then clocks in one null bit (a dummy bit that always
' reads 0) followed by the conversion data.

Convert:
  low CLK                     ' Low clock—output on rising edge.
  high DIO_n                  ' Switch DIO to output high (start bit).
  low CS                      ' Activate the 1298.
  pulsout CLK,5               ' Send start bit.
  let DIO_p = sglDif          ' First setup bit.
  pulsout CLK,5               ' Send bit.
  let DIO_p = oddSign         ' Second setup bit.
  pulsout CLK,5               ' Send bit.
  let DIO_p = msbf            ' Final setup bit.
  pulsout CLK,5               ' Send bit.
  input DIO_n                 ' Get ready for input from DIO.
  let AD = 0                  ' Clear old ADC result.
  for ADbits = 1 to 13        ' Get null bit + 12 data bits.
    let AD = AD*2+DIO_p       ' Shift AD left, add new data bit.
    pulsout CLK,5             ' Clock next data bit in.
  next                        ' Get next data bit.
  high CS                     ' Turn off the ADC
return                        ' Return to program.
```

Introduction. This application note shows how to interface the DS1620 Digital Thermometer to the BASIC Stamp.

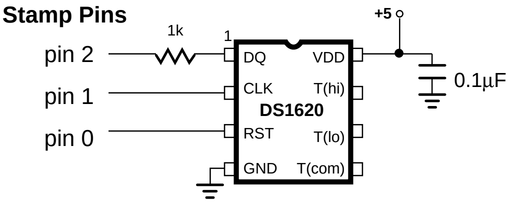
Background. In application note #7, we demonstrated a method for converting the non-linear resistance of a thermistor to temperature readings. Although satisfyingly cheap and crafty, the application requires careful calibration and industrial-strength math.

Now we’re going to present the opposite approach: throw money ($7) at the problem and get precise, no-calibration temperature data.

How it works. The Dallas Semiconductor DS1620 digital thermometer / thermostat chip, shown in the figure, measures temperature in units of 0.5 degrees Centigrade from –55° to +125° C. It is calibrated at the factory for exceptional accuracy: +0.5° C from 0 to +70° C.

(In the familiar Fahrenheit scale, those °C temperatures are: range, –67° to +257° F; resolution, 0.9° F; accuracy, +0.9° F from 32° to 158° F.)

The chip outputs temperature data as a 9-bit number conveyed over a three-wire serial interface. The DS1620 can be set to operate continuously, taking one temperature measurement per second, or intermit-



DQ—Data input/output
CLK—Clock for shifting data in/out (active-low conversion start in thermostat/ 1-shot mode)
RST—Reset; high activates chip, low disables it
GND—Ground connection
VDD—Supply voltage; +4.5 to 5.5 Vdc
T(hi)—In thermostat mode, outputs a 1 when temp is above high setpoint
T(lo)—In thermostat mode, outputs a 1 when temp is below low setpoint
T(com)—In thermostat mode, outputs a 1 when temp exceeds high setpoint and remains high until temp drops below low setpoint

Schematic to accompany DS1620.BAS

tently, conserving power by measuring only when told to.

The DS1620 can also operate as a standalone thermostat. A temporary connection to a Stamp establishes the mode of operation and high/low-temperature setpoints. Thereafter, the chip independently controls three outputs: T(high), which goes active at temperatures above the high-temperature setpoint; T(low), active at temperatures below the low setpoint; and T(com), which goes active at temperatures above the high setpoint, and stays active until the temperature drops below the low setpoint.

We'll concentrate on applications using the DS1620 as a Stamp peripheral, as shown in the listing.

Using the DS1620 requires sending a command (what Dallas Semi calls a *protocol*) to the chip, then listening for a response (if applicable). The code under "DS1620 I/O Subroutines" in the listing shows how this is done. In a typical temperature-measurement application, the program will set the DS1620 to thermometer mode, configure it for continuous conversions, and tell it to start. Thereafter, all the program must do is request a temperature reading, then shift it in, as shown in the listing's *Again* loop.

The DS1620 delivers temperature data in a nine-bit, two's complement format, shown in the table. Each unit represents 0.5° C, so a reading of 50 translates to +25° C. Negative values are expressed as two's complement numbers. In two's complement, values with a 1 in their leftmost bit position are negative. The leftmost bit is often called the *sign* bit, since a 1 means – and a 0 means +.

To convert a negative two's complement value to a positive number, you must invert it and add 1. If you want to display this value, remember to put a minus sign in front of it.

Rather than mess with two's complement negative numbers, the program converts DS1620 data to an absolute scale called *DSabs*, with a range of 0 to 360 units of 0.5° C each. The Stamp can perform calculations in this all-positive system, then readily convert the results for display in °C or °F, as shown in the listing.

Once you have configured the DS1620, you don't have to reconfigure it unless you want to change a setting. The DS1620 stores its configuration in EEPROM (electrically erasable, programmable read-only memory), which retains data even with the power off. In memory-tight Stamp applications, you might want to run the full program once for configuration, then strip out the configuration stuff to make more room for your final application.

If you want to use the DS1620 in its role as a standalone thermostat, the Stamp can help here, too. The listing includes protocols for putting the DS1620 into thermostat (*NoCPU*) mode, and for reading and writing the temperature setpoints. You could write a Stamp program to accept temperature data serially, convert it to nine-bit, two's complement format, then write it to the DS1620 configuration register.

Be aware of the DS1620's drive limitations in thermostat mode; it sources just 1 mA and sinks 4 mA. This isn't nearly enough to drive a relay—it's just enough to light an LED. You'll want to buffer this output with a Darlington transistor or MOSFET switch in serious applications.

Parts sources. The DS1620 is available from Jameco (800-831-4242) for $6.95 in single quantity as part number 114382 (8-pin DIP). Be sure to

Nine-Bit Format for DS1620 Temperature Data

Temperature		DS1620 Data		
°F	*°C*	*Binary*	*Hex*	*Decimal*
+257	+125	0 11111010	00FA	250
+77	+25	0 00110010	0032	50
+32.9	+0.5	0 00000001	0001	1
+32	0	0 00000000	0000	0
+31.1	-0.5	1 11111111	01FF	511
-13	-25	1 11001110	01CE	462
-67	-55	1 10010010	0192	402

Example conversion of a negative temperature:
-25°C = 1 11001110 in binary. The 1 in the leftmost bit indicates that this is a negative number. Invert the lower eight bits and add 1: 11001110 -> 00110001 +1 = 00110010 = 50. Units are 0.5°C, so divide by 2. Converted result is -25°C.

request a data sheet when you order. Dallas Semiconductor offers data and samples of the DS1620 at reasonable cost. Call them at 214-450-0448.

Program listing. The program DS1620.BAS is available from the Parallax bulletin board system. You can reach the BBS at (916) 624-7101. You may also obtain this and other Stamp programs via Internet: ftp.parallaxinc.com.

```
' Program: DS1620.BAS
' This program interfaces the DS1620 Digital Thermometer to the
' BASIC Stamp. Input and output subroutines can be combined to
' set the '1620 for thermometer or thermostat operation, read
' or write nonvolatile temperature setpoints and configuration
' data.

' ===================== Define Pins and Variables ================
SYMBOL  DQp = pin2        ' Data I/O pin.
SYMBOL  DQn = 2           ' Data I/O pin _number_.
SYMBOL  CLKn = 1          ' Clock pin number.
SYMBOL  RSTn = 0          ' Reset pin number.
SYMBOL  DSout = w0        ' Use bit-addressable byte for DS1620 output.
SYMBOL  DSin = w0         ' "   "   "          word "   "      input.
SYMBOL  clocks = b2       ' Counter for clock pulses.

' ===================== Define DS1620 Constants ===================
' >>> Constants for configuring the DS1620
SYMBOL  Rconfig = $AC     ' Protocol for 'Read Configuration.'
SYMBOL  Wconfig = $0C     ' Protocol for 'Write Configuration.'
SYMBOL  CPU = %10         ' Config bit: serial thermometer mode.
SYMBOL  NoCPU = %00       ' Config bit: standalone thermostat mode.
SYMBOL  OneShot = %01     ' Config bit: one conversion per start request.
SYMBOL  Cont = %00        ' Config bit: continuous conversions after start.
' >>> Constants for serial thermometer applications.
SYMBOL  StartC = $EE      ' Protocol for 'Start Conversion.'
SYMBOL  StopC = $22       ' Protocol for 'Stop Conversion.'
SYMBOL  Rtemp = $AA       ' Protocol for 'Read Temperature.'
' >>> Constants for programming thermostat functions.
SYMBOL  RhiT = $A1        ' Protocol for 'Read High-Temperature Setting.'
SYMBOL  WhiT = $01        ' Protocol for 'Write High-Temperature Setting.'
SYMBOL  RloT = $A2        ' Protocol for 'Read Low-Temperature Setting.'
SYMBOL  WloT = $02        ' Protocol for 'Write Low-Temperature Setting.'

' ===================== Begin Program ============================
' Start by setting initial conditions of I/O lines.
low RSTn        ' Deactivate the DS1620 for now.
```

```
high CLKn       ' Initially high as shown in DS specs.
pause 100       ' Wait a bit for things to settle down.

' Now configure the DS1620 for thermometer operation. The
' configuration register is nonvolatile EEPROM. You only need to
' configure the DS1620 once. It will retain those configuration
' settings until you change them—even with power removed. To
' conserve Stamp program memory, you can preconfigure the DS1620,
' then remove the configuration code from your final program.
' (You'll still need to issue a start-conversion command, though.)
let DSout=Wconfig          ' Put write-config command into output byte.
gosub Shout                ' And send it to the DS1620.
let DSout=CPU+Cont         ' Configure as thermometer, continuous conversion.
gosub Shout                ' Send to DS1620.
low RSTn                   ' Deactivate '1620.
Pause 50                   ' Wait 50ms for EEPROM programming cycle.
let DSout=StartC           ' Now, start the conversions by
gosub Shout                ' sending the start protocol to DS1620.
low RSTn                   ' Deactivate '1620.

' The loop below continuously reads the latest temperature data from
' the DS1620. The '1620 performs one temperature conversion per second.
' If you read it more frequently than that, you'll get the result
' of the most recent conversion. The '1620 data is a 9-bit number
' in units of 0.5 deg. C. See the ConverTemp subroutine below.
Again:
  pause 1000               ' Wait 1 second for conversion to finish.
  let DSout=Rtemp          ' Send the read-temperature opcode.
  gosub Shout
  gosub Shin               ' Get the data.
  low RSTn                 ' Deactivate the DS1620.
  gosub ConverTemp         ' Convert the temperature reading to absolute.
  gosub DisplayF           ' Display in degrees F.
  gosub DisplayC           ' Display in degrees C.
goto Again

' ==================== DS1620 I/O Subroutines ==================
' Subroutine: Shout
' Shift bits out to the DS1620. Sends the lower 8 bits stored in
' DSout (w0). Note that Shout activates the DS1620, since all trans-
' actions begin with the Stamp sending a protocol (command). It does
' not deactivate the DS1620, though, since many transactions either
' send additional data, or receive data after the initial protocol.
' Note that Shout destroys the contents of DSout in the process of
' shifting it. If you need to save this value, copy it to another
' register.
Shout:
high RSTn                  ' Activate DS1620.
output DQn                 ' Set to output to send data to DS1620.
```

```
for clocks = 1 to 8            ' Send 8 data bits.
  low CLKn                     ' Data is valid on rising edge of clock.
  let DQp = bit0               ' Set up the data bit.
  high CLKn                    ' Raise clock.
  let DSout=DSout/2            ' Shift next data bit into position.
next                           ' If less than 8 bits sent, loop.
return                         ' Else return.

' Subroutine: Shin
' Shift bits in from the DS1620. Reads 9 bits into the lsbs of DSin
' (w0). Shin is written to get 9 bits because the DS1620's temperature
' readings are 9 bits long. If you use Shin to read the configuration
' register, just ignore the 9th bit. Note that DSin overlaps with DSout.
' If you need to save the value shifted in, copy it to another register
' before the next Shout.
Shin:
input DQn                      ' Get ready for input from DQ.
for clocks = 1 to 9            ' Receive 9 data bits.
  let DSin = DSin/2            ' Shift input right.
  low CLKn                     ' DQ is valid after falling edge of clock.
  let bit8 = DQp               ' Get the data bit.
  high CLKn                    ' Raise the clock.
next                           ' If less than 9 bits received, loop.
return                         ' Else return.

' ================ Data Conversion/Display Subroutines ===============
' Subroutine: ConverTemp
' The DS1620 has a range of -55 to +125 degrees C in increments of 1/2
' degree. It's awkward to work with negative numbers in the Stamp's
' positive-integer math, so I've made up  a temperature scale called
' DSabs (DS1620 absolute scale) that ranges from 0 (-55 C) to 360 (+125 C).
' Internally, your program can do its math in DSabs, then convert to
' degrees F or C for display.
ConverTemp:
if bit8 = 0 then skip          ' If temp > 0 skip "sign extension" procedure.
  let w0 = w0 | $FE00          ' Make bits 9 through 15 all 1s to make a
                               ' 16-bit two's complement number.
skip:
  let w0 = w0 + 110            ' Add 110 to reading and return.
return

' Subroutine: DisplayF
' Convert the temperature in DSabs to degrees F and display on the
' PC screen using debug.
DisplayF:
let w1 = w0*9/10               ' Convert to degrees F relative to -67.
if w1 < 67 then subzF          ' Handle negative numbers.
  let w1 = w1-67
  Debug #w1, " F",cr
```

```
return
subzF:
  let w1 = 67-w1            ' Calculate degrees below 0.
  Debug "-",#w1," F",cr     ' Display with minus sign.
return

' Subroutine: DisplayC
' Convert the temperature in DSabs to degrees C and display on the
' PC screen using debug.
DisplayC:
let w1 = w0/2               ' Convert to degrees C relative to -55.
if w1 < 55 then subzC       ' Handle negative numbers.
  let w1 = w1-55
  Debug #w1, " C",cr
return
subzC:
  let w1 = 55-w1            ' Calculate degrees below 0.
  Debug "-",#w1," C",cr     ' Display with minus sign.
return
```