What's a Microcontroller?

Student Guide for Experiments #1 through #6

Version 1.9

Warranty

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

14-Day Money Back Guarantee

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

Copyrights and Trademarks

This documentation is copyright 1999 by Parallax, Inc. BASIC Stamp is a registered trademark of Parallax, Inc. If you decided to use the name BASIC Stamp on your web page or in printed material, you must state that "BASIC Stamp is a registered trademark of Parallax, Inc." Other brand and product names are trademarks or registered trademarks of their respective holders.

Disclaimer of Liability

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

Internet Access

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

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

Internet BASIC Stamp Discussion List

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

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

Preface ... 3
Audience and Teacher's Guides ... 3
Future Experiments ... 4
Copyright and Reproduction ... 4
Special Contributors ... 4

Experiment #1: What's a Microcontroller? ... 7
Parts Required ... 9
Build It! ... 9
Program It! ... 14
Questions ... 22
Challenge! ... 23
What have I learned? ... 24
Why did I learn it? ... 25
How can I apply this? ... 25

Experiment #2: Detecting the Outside World ... 27
Parts Required ... 27
Build It! ... 28
Program It! ... 30
Questions ... 36
Challenge! ... 37
What have I learned? ... 38
Why did I learn it? ... 39
How can I apply this? ... 39

Experiment #3: Micro-controlled Movement: ... 41
Parts Required ... 42
Build it! ... 42
Program It! ... 45
Questions ... 52
Challenge! ... 53
What have I learned? ... 54
What did I learn it? ... 55
How can I apply this? ... 55

Experiment #4: Simple Automation 57
Parts Required 58
Built It! 58
Program It! 60
Questions 66
Challenge! 67
What have I learned? 68
Why did I learn it? 69
How can I apply this? 69

Experiment #5: Measuring an Input 71
Parts Required 72
Build It! 72
Program It! 75
Questions 79
Challenge! 80
What have I learned? 81
Why did I learn it? 82
How can I apply this? 82

Experiment #6: Manual to Digital 83
Parts Required 84
Build It! 84
Program It! 87
Questions 93
Challenge! 94
What have I learned? 95
Why did I learn it? 96
How can I apply this? 96
Parts Listing 97

Appendix A: Parts Listing and Sources 97
Appendix B: Reading the Resistor Color Code 103
Appendix C: DS1804 Datasheet 105
Appendix D: PBASIC Quick Reference Guide 113

Preface

In 1979 Chip Gracey had his first introduction to programming and electronics: the Apple II computer. Chip was instantly interested in the new machine, writing BASIC code to display graphics and removing the case to see the electronic components. This experience quickly led to dismantling video game source code and household electronic hardware, and trying to use these devices for purposes other than originally intended. Hobby transformed into a business, and by the time he was a senior in high school Chip was running a small business from his bedroom making software duplication hardware for the Commodore 64 computer.

High schools offered no software or hardware classes in 1982, and when Chip graduated in 1986 college just didn't seem like the right place to start running a business. Instead, he and a friend Lance Walley started "Parallax" from their apartment. The first products included sound digitizers for the Apple II and 8051 programmers.

The business grew slowly until 1992 when Parallax released the first BASIC Stamp. Parallax knew the BASIC Stamp would be special – it was the tool they needed to do their own hobby projects. The fact that the BASIC Stamp would create it's own industry was probably unknown by the Parallax founders, but it quickly became apparent that the small computer had it's own group of enthusiasts. It let ordinary people program a microcontroller for the first time, and gave them powerful I/O commands that made it easy to connect to other electronic components. By the end of 1998 Parallax sold over 125,000 BASIC Stamp modules and distributed a complete series of supporting peripherals through over 40 world-wide sales channels.

What does this story have to do with Stamps in Class curriculum? The Stamps in Class curriculum is designed to introduce students and teachers to microcontrollers using software basics and simple hardware, integrating the two without a tremendous investment (the curriculum is free and Parallax has educational prices for the hardware). It starts from the bottom, with hands-on projects and programming. This is the way Chip learned microcontrollers, and gained enough engineering experience to design the BASIC Stamp.

Audience and Teacher's Guides

The "What's a Microcontroller?" curriculum was created for ages 15-18 with a purpose of providing a entry-level background to microcontroller programming and interfacing, but it is also appropriate for college classes with additional instructor supplementary material. The curriculum combines software and hardware, first showing the student how to build the circuit, then program the microcontroller, and finally challenging them to improve the design. Teacher's guides for each lesson are available by e-mail request to stampsinclass@parallaxinc.com.

Of course some electronic background would be helpful. Electronic principles are not explained in depth since there are many great texts to be read side-by-side with this curriculum. The Radio Shack Forrest Mimm's Getting Started in Electronics are the best accompaniment. Some other more advanced resources include the Scott Edwards' Programming and Customizing the BASIC Stamp Computer and the Parallax BASIC Stamp Manual Version 1.9. See Appendix A for book sources..

Future Experiments

The "What's a Microcontroller?" series is comprised of the first six experiments included in this book. Future lessons will also be published in sets of six. The contents of these experiments are based on feedback from students and instructors. If you have ideas for future experiments please send them to the editorial team at stampsinclass@parallaxinc.com. We also accept experiment contributions, just be sure to see the editorial guidelines from our Stamps in Class web site.

Copyright and Reproduction

Stamps in Class What's a Microcontroller? lessons are Copyright © Parallax 2000. Parallax grants every person a conditional right to download, duplicate, and distribute this text without our permission. The condition is that this text, or any portion thereof, should not be duplicated for commercial use resulting in expenses to the user beyond the marginal cost of printing. That is, *nobody* would profit from duplication of this text. Preferably, duplication would have no expense to the student. Any educational institution wishing to produce duplicates for their students may do so without our permission. This text is also available in printed format from Parallax. Because we print the text in volume, the consumer price is often less than typical xerographic duplication charges. This text may be translated to any foreign language with the permission of Parallax, Inc.

Special Contributors

This curriculum was originally written by I-Four of Grass Valley, California. I-Four's primary author is Matt Gilliland. Parallax stumbled across Matt on the internet. Matt is a true educational proponent - he personally tested these experiments on a group of enthusiastic neighborhood kids. This initial feedback was important since it resulted in substantial revisions (if you find any mistakes or areas for improvement please let us know by e-mail to stampsinclass@parallaxinc.com). I-Four also authored portions of the Full Option Science System (FOSS) junior-high level electronics program with The Lawrence Hall of Science at University of California, Berkeley. Since Matt's original work, many educators have provided detailed comments for our incorporation into the text. We've made many improvements on the circuits, source code, and explanations due to their help. This is the seventh version of What's a Microcontroller.

Special thanks also to the Parallax team who provided ideas and content for the program. The Parallax staff who designs, manufacturers, and accepts orders and packages the Stamps in Class products is a key part of the Stamps in Class program. Of course, it took a BASIC Stamp to make the program possible. Thank you Chip for creating this unique product and new industry.

Experiment #1: What's a Microcontroller?

Most of us know what a computer looks like. It usually has a keyboard, monitor, CPU (Central Processing Unit), printer, and a mouse. These types of computers, like the Mac or PC, are primarily designed to communicate (or "interface") with humans.

Database management, financial analysis, or even word-processing are all accomplished inside the "big box" that contains the CPU, memory, hard drive, etc. The actual "computing", however, takes place within the CPU.

If you think about it, the whole purpose of a monitor, keyboard, mouse, and even the printer is to "connect" the CPU to the outside world.

What's a...

CPU:
Central Processing Unit. This term specifically refers to the integrated circuit (contained inside the large computer "box") that does the "real computing". However, sometimes the term is used (although incorrectly) to include everything inside the "box", including the hard & floppy drives, CD-ROM, power supply & motherboard.

Microcontroller:
An integrated circuit that contains many of the same items that a desktop computer has, such as CPU, memory, etc., but does not include any "human interface" devices like a monitor, keyboard, or mouse. Microcontrollers are designed for machine control applications, rather than human interaction.

But did you know that there are computers all around us, running programs and quietly doing calculations, not interacting with humans at all? These computers are in your car, on the Space Shuttle, in your kid brother's toy, and maybe even inside your hairdryer.

We call these devices "microcontrollers". *Micro* because they're small, and *controller* because they "control" machines, gadgets, whatever. Microcontroller's by definition then, are designed to connect to machines, rather than people. They're cool because, you can build a machine or device, write programs to control it and then let it work for you *automatically*.

There is an infinite number of applications for microcontrollers. Your imagination is the only limiting factor!

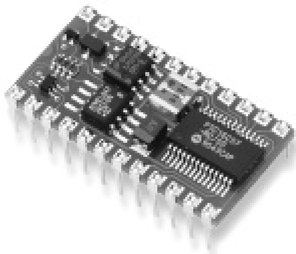
Hundreds (if not thousands) of different variations of microcontrollers are available. Some are programmed once and produced for specific applications, such as controlling your microwave oven. Others are "re-programmable", which means they can be used over and over for *different* applications. Microcontrollers are incredibly versatile – the same device may control a model rocket, a toaster, or even your car's antilock braking system.

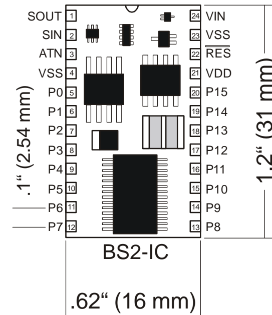
This experiment will introduce us to one very popular microcontroller called the BASIC Stamp. The BASIC Stamp is a sophisticated array of circuitry, all assembled onto a very small printed circuit board (PCB). In fact, the PCB is the same size as many other types of "integrated circuits". The BASIC Stamp is shown on the following page in Figure 1.1.

What's a...

PCB:
Printed Circuit Board. Complex electronic circuits require many electrical connections between components. A printed circuit board is simply a rigid piece of (usually) fiberglass that has many copper wires embedded on (or sometimes *in*) it. These wires carry the signals between individual components in the circuit.

Figure 1.1: BASIC Stamp II
This is a small picture of the BASIC Stamp II module. The actual module is about the size of a postage stamp.





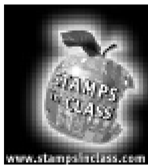
Writing programs for the BASIC Stamp is accomplished with a special version of the BASIC language (called PBASIC). Most other microcontrollers require some form of programming that may be very difficult to learn. With the BASIC Stamp, you can create simple circuits and programs in a matter of minutes (which we're about to do!). However, do not be misled into thinking that all the BASIC Stamp can do is "simple stuff". Many sophisticated commercial products have been created and sold, using the BASIC Stamp as a "brain".

When we create devices that have a microcontroller acting as a "brain", in many ways we are attempting to mimic how our own bodies operate.

Your brain relies on certain information in order to make decisions. That information is gathered through various senses such as, sight, hearing, touch, etc. These senses detect what we'll call the "real world", and send that information to your brain for "processing". Conversely, when your brain makes a decision, it sends signals throughout your body to do something *to* the "real world". Utilizing the "inputs" from your senses, and the "outputs" from your legs, arms, hands, etc., your brain is *interfaced and interacting* with the real world.

As you're driving down the road, your eyes detect a deer running out in front of you. Your brain analyzes this "input", makes a decision, and then "outputs" instructions to your arms and hands, turning the steering wheel to avoid hitting the animal. This "*input / decision / output*" is what microcontrollers are all about. We call this input/output, or "I/O" for short.

This first lesson will introduce you to the output function of the BASIC Stamp, and each following lesson will introduce new ideas and experiments for you to try. You will be able to use the ideas from these lessons to invent your own applications for microcontroller programs and circuits.



Parts Required

For each experiment, you need an IBM-compatible PC running DOS 2.0 or higher, Win95/98/2000 or NT4.0. For Experiment #1 you will need the following:

(1) BASIC Stamp II module
(1) Board of Education
(1) Programming Cable
(2) LED's (light emitting diodes)
(2) 470 ohm, ¼ watt resistors (yellow, violet, brown)
(1) 9 volt battery or wall transformer connected to the Board of Education
(6) Jumper wires

Build It!

Any microcontroller (or computer) system consists of two primary components: hardware and software. The hardware is the actual physical components of the system. The software is a list of instructions which reside *inside* the hardware. We will now create the hardware, and then write a software program to "control it".

In order for our microcontroller to interact with the real world, we need to assemble some "hardware". We'll be using a PCB called the "Board of Education". This board was created to simplify connecting "real world stuff" to the BASIC Stamp. Connectors are provided for power (wall transformer or 9 volt battery), the programming cable, and the Input / Output pins of the BASIC Stamp. There is also a "prototyping area" or breadboard (the white board with all the holes in it). It is this area that we'll be building our circuitry. See Figure 1.2.

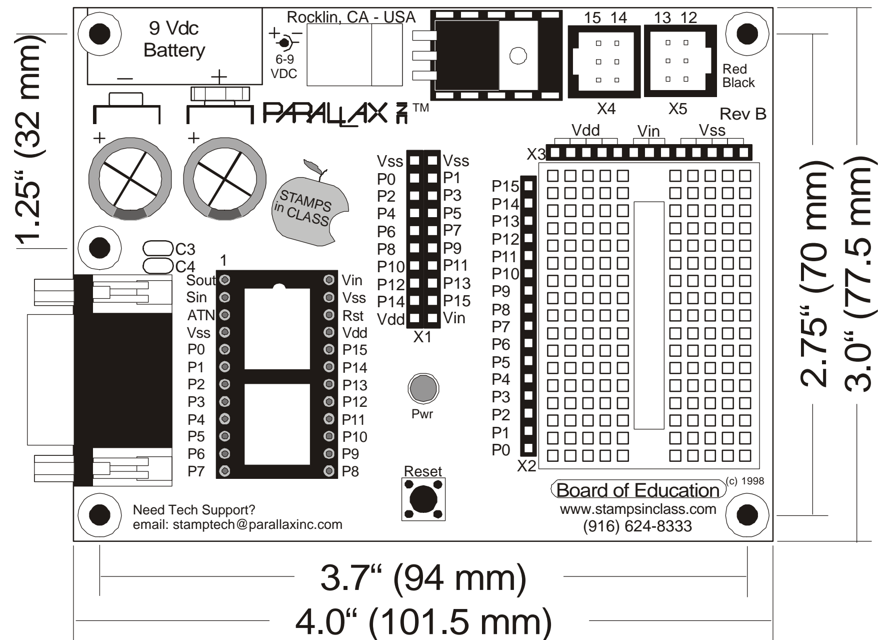


Figure 1.2:
Board of Education Rev. B
This is where we will build our circuit. The socket is for the BASIC Stamp module, and the breadboard is for your projects. The BASIC Stamp is oriented with the large chip closest towards the "AppMod Connector"

What's a...
LED:
Light Emitting Diode. A special type of semi-conductor diode, which when connected to an electronic circuit (with a current limiting resistor) emits visible light. LED's use very little power, & are ideally suited for connecting to devices such as the Stamp.

In this experiment we will be connecting two Light Emitting Diodes (LED's) to the BASIC Stamp. LED's are a special form of lamp, that for various reasons, are easily connected to microcontroller devices.

There are two *very important* things to remember when connecting LED's to the BASIC Stamp. The first is *always be sure that there is a resistor connected*, as shown in Figure 1.3 below. In this experiment the resistors should be rated at 470 ohms, ¼ watt. See Appendix C for additional information.

Secondly, be certain that the *polarity* of the LED is correct. There is a flat spot on the side of the LED that should be connected as shown in Figure 1.3. If the polarity is reversed, the LED will not work. The flat side also has the shortest LED lead.

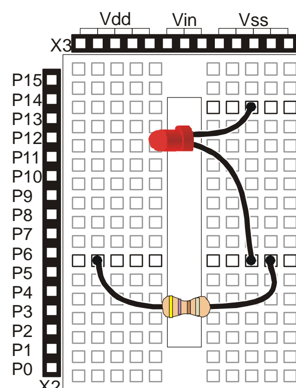


Figure 1.3: LED on Breadboard
Shows LED and resistor "plugged" into breadboard. No connections have been made yet to the BASIC Stamp's I/O pins.

When inserting an LED into the breadboard, bend the leads at right angles a short distance from the body, because some LEDs do not hold up well to stress on the plastic.

Understanding the Breadboard

The BASIC Stamp has a total of 24 pins, as shown in Figure 1.1. Some of these signals are used to connect the BASIC Stamp to the PC and the 9 volt battery (or wall pack). Sixteen of these signals (P0 through P15) are available for us to connect to the "real world".

On the Board of Education, you can follow a "trace" from the BASIC Stamp module to the line sockets on the left of the breadboard. Each BASIC Stamp I/O pin is brought to the edge of the breadboard, and with wires you can "jumper" from the sockets onto the breadboard.

Connecting an LED:
Never connect an LED to the Stamp, without having a resistor (of the proper value) in the circuit . The resistor limits the amount of current flow in the circuit to a safe level, thereby protecting both the LED and the Stamp.

It's important to understand how a breadboard works. The breadboard has many metal strips which run underneath in rows. These strips connect the sockets to each other. This makes it easy to connect components together to build an electrical circuit.

To use the breadboard, the legs of the LED and resistor will be placed in the sockets. These sockets are made so that they will hold the component in place. Each hole is connected to one of the metal strips running underneath the board. You can connect different components by plugging them into common nodes. Figure 1.4 is a small pictorial of this concept.

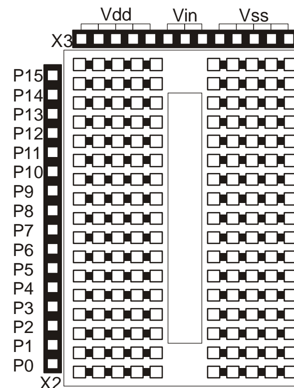


Figure 1.4: Breadboard connections
The horizontal black lines show how the "sockets" are connected underneath the breadboard. This means you don't have to plug two wires into one socket since the socket to the right or left is connected.

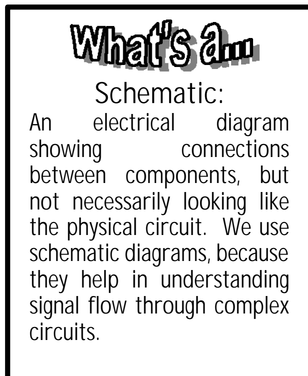
Vdd is +5 Volts, and Vin is an unregulated voltage from your power supply. For example if you use a 9-volt battery Vin is + 9 Volts. Vss is Ground.

Each BASIC Stamp pin has a "signal name" associated with it. For example pin #24 is VIN (which stands for "voltage in"). This is one of the connections for the 9 volt battery. When you plug in the battery, a connection is made from the battery to this pin via a copper wire that is embedded on the Board of Education.

The pins / signals that we will be working with for this experiment are as follows:

Pin #	Signal Name
5	P0
6	P1
21	Vdd (+5 volts)

When we program the BASIC Stamp, we will refer to the Signal Name, rather than the actual pin number.



OK, lets build the circuit! Do not connect the power supply (9 volt battery or wall transformer) yet.

Figures 1.5 and 1.6 are two different methods to show an electrical diagram. Figure 1.5 is a "schematic" diagram of the circuit. Figure 1.6 is the same circuit, but drawn as a pictorial to show what the circuit *physically* looks like. In each experiment you will be shown a schematic and a pictorial until we progress to more advanced lessons.

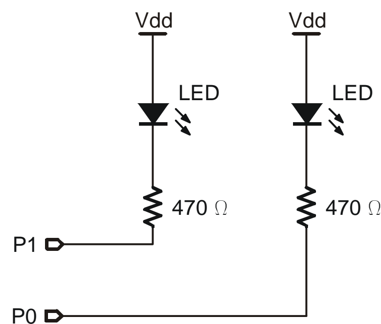


Figure 1.5: Schematic Electrical diagram for circuit shown on the right side.

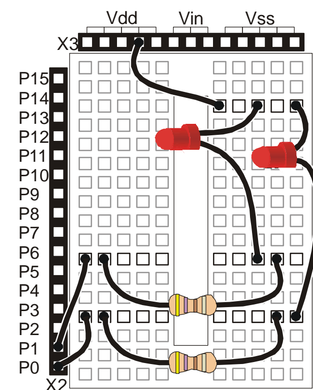


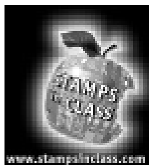
Figure 1.6: Pictorial What the circuit physically looks like after you build it. The flat side of the LED is closest to the resistor.

Connect the first LED:

1. Plug a wire into P0 and then into the breadboard as shown. Then plug a resistor into the breadboard adjacent to the wire, and plug the other end of the resistor into the other side of the breadboard.
2. Plug the LED in the breadboard adjacent to the resistor. Make sure that the lead next to the flat side of the LED connects to the resistor.
3. Plug the remaining lead on the LED to Vdd (+5v) on the Board of Education.

Connect the second LED:

1. Plug a wire into the the P1 position and connect it on the breadboard. Then plug a resistor into the breadboard adjacent to the wire, and plug the other end of the resistor into the right side of the breadboard.
2. Plug the LED into the breadboard adjacent to the resistor. Make sure that the lead next to the flat side of the LED connects to the resistor.
3. Connect the remaining lead from the LED to Vdd (+5v) of the Board of Education, using a connecting wire.



Program It!

Connect the Board of Education to the PC:

1. Plug one end of the programming cable into the Board of Education.
2. Plug the other end of the programming cable into an available serial port connector on the PC.

> **What's a...**
> **Program:**
> A sequence of instructions that are executed by a computer or microcontroller in a specific sequence to carry out a task. Programs are written in different types of "languages", such as Fortran, "C", or BASIC.

> **What's a...**
> **Bug:**
> An error in your program or hardware. To "debug" your program, is to track down & eliminate errors in your code. There may also be hardware errors such as reversing an LED that causes the system not to function.

That does it! We've just created a "hardware" circuit. But it doesn't do anything yet. That's why we need to...

How many of you already know how to write a computer program? If you've done it before, then the first part of this section may be review. But if you're a "newbie", don't worry! It's really not that hard.

A computer program is nothing more than a list of instructions that a computer (or in our case, a microcontroller) executes. We create a program for the microcontroller by typing it into a PC (utilizing the keyboard & monitor), then we send this "code" through the programming cable, to the microcontroller. This program (or list of instructions) then runs or "executes" inside the BASIC Stamp.

If we've written the program correctly, it will do what we want it to do. However, if we make a mistake, then the device won't work (or works poorly), and we need to "debug it". Debugging can be one of the most hair-pulling experiences in the entire process, therefore, the more careful you are in creating the program, theoretically the easier it'll be to debug. A software "bug" is an error in your program. Therefore, debugging is the art of "bug" removal!

PBASIC for the BASIC Stamp has a bunch of *commands* to choose from; 36 to be exact. A complete listing and description on each of these commands can be obtained from the Basic Stamp Manual Version 1.9, but each command used in these lessons is further described in Appendix E, PBASIC Quick Reference.

For the purposes of this experiment we're going to look at only four commands.

These are: **`OUTPUT, PAUSE, GOTO,`** and **`OUT`**.

As mentioned above, a program is a list of instructions that are executed in a sequence determined by the structure of the program itself. Therefore, as we write a program, it is very important to keep in mind the sequence of execution that we desire.

For example, if we want to buy a soda from a vending machine, our brain executes a list of commands to accomplish this. Perhaps something like...

1. Insert $1.00 into slot.
2. Wait for green light to come on.
3. Push button for soda type.
4. Watch soda fall into tray.
5. Pick up soda from tray.
6. Open soda.
7. Drink soda.
8. Burp.

Now, that seems pretty straightforward, but only because we've done it before.

If however, your brain was sending out the following "program":

1. Push button for soda type.
2. Open soda.
3. Insert $1.00 into slot.
4. Pick up soda from tray.
5. Burp
6. Drink soda.
7. Wait for green light to come on.
8. Watch soda fall into tray.

Not much would happen. All the proper commands are there, but they're in the *wrong order*. Once you've pushed the button for "soda type" (step #1), your brain (program) would "hang" or stall because it can't execute "open soda", because there's no soda to open!

This is a "bug". We humans can modify our brain "program" as the situation is happening, and we can of course ultimately figure out how to get that soda.

Microcontrollers, however, don't have the capacity to "adapt" and modify their own set of instructions – they're only able to execute the *exact sequence* of instructions that we give them.

Ok, enough background, let's program this microcontroller to do something!

Plug the BASIC Stamp II into the Board of Education, with the big chip towards the bottom of the board. Connect the 9 volt battery or wall pack to the Board of Education. Connect the serial cable to your PC.

Turn on your PC. BASIC Stamp software runs in DOS and Windows 95/98/NT4.0. We'll assume that you're using a computer with Windows 95. You first need to copy the contents of the disk onto your PC desktop, or into a folder.

What's a...

Download:
After a microcontroller program has been created on the PC, it is sent from the PC *down* a cable, & *loaded* into the micro-controller's memory. The program is then executed from within the Stamp.

Double click on the BASIC Stamp icon.

You should now be running a program called the "Stamp Editor". This is a program that was created to help you write and download programs to the BASIC Stamp microcontroller. The screen will look something like Figure 1.7:

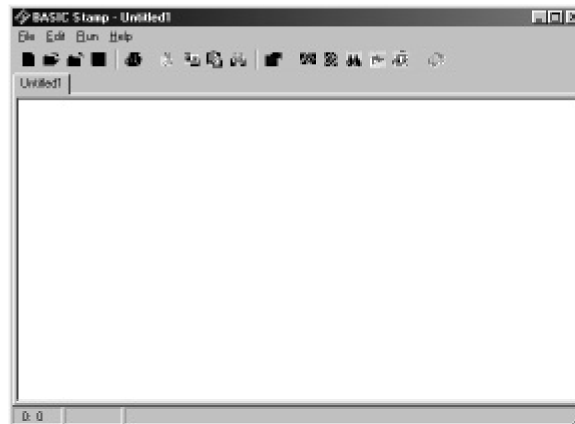


Figure 1.7: BASIC Stamp Software
Double-click on the BASIC Stamp icon to run the software. The opening screen will look like this.

The screen, except for a few words across the top, is blank. This is where you will create your programs. Now remember, we are going to write our program utilizing the "human interface" equipment (monitor, keyboard, etc.) that is part of your PC. The program that we will write will not run on the PC, but rather will be "downloaded" or sent to the microcontroller. Once the program has been received, the BASIC Stamp will execute the instructions exactly as we've created them.

Type the following program into the BASIC Stamp editor so it looks like Figure 1.8:

```
output 0
reblink:
    out0 = 0
pause 1000
    out0 = 1
    pause 1000
goto reblink
```

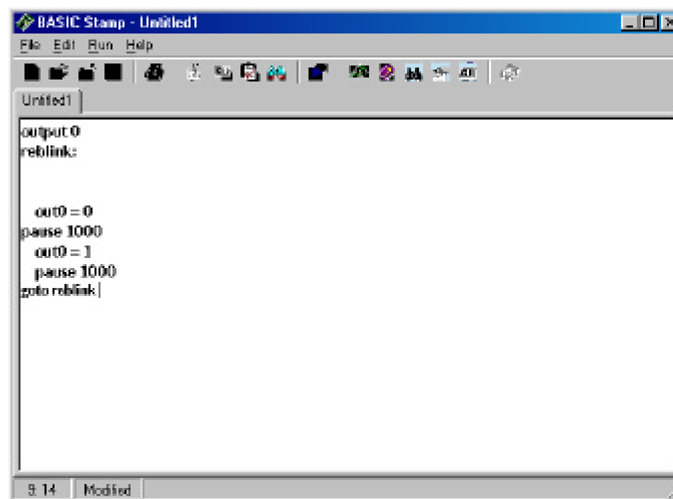


Figure 1.8:
BASIC Stamp Software
Type the code into the editor so it looks like this screen.

Now while holding the "ALT" key down, type the letter "R" (for "run") and then press "ENTER" when the menu shows the RUN command. If everything went well, the LED that is connected to P0 (pin #5 on the Board of Education) should be blinking on and off. The second LED won't blink yet because we have not written any code to control it.

FYI:
The Stamp Editor:
If you are using the DOS version, pressing the "F1" key will first show you how many variables you have used. Pressing the spacebar moves between (1) variable, (2) overall program memory, and (3) detailed program memory. To find out how big your program is, simply hold down the ALT key & press "m".

If you get a message that says, "Hardware not found", re-check the cable connections between the PC and Board of Education, & also make sure that a power supply is connected to the Board of Education. If it still does not work, check under the EDIT menu, PREFERENCES option, and EDITOR OPERATION tab. The default COM port setting should be AUTO.

Try downloading again (hold down the ALT key, & then press "r"). If it still doesn't work, you may have a bug! Re-check your program to be certain you've typed it correctly.

If after trying this, you're still having problems, ask you instructor for help.

Now lets dissect, and look at our program:

The first command used is **`output`**. Each signal (P0 & P15) can be setup as an "input" or an "output". Since we want the microcontroller to "turn on and off" an LED, the microcontroller is *manipulating* the "real world". Therefore, by definition, we want P0 to be an "**`output`**".

Result of the first command: **`output 0`** makes P0 an output. (Hint: If we had wanted to make P1 an output, the command would have been "output 1").

The next item in the program **`reblink:`**, isn't really a command. It's just a label, or a marker for a certain point in the program. We'll get back to this in a moment.

Pin #5 on the BASIC Stamp is P0 as we call it, and is an output. In the world of computers, voltages on these pins can be either "high" or "low", meaning a high voltage or a low voltage. Another way to refer to high & low is "1 & 0". "1" being high and "0" being low.

FYI:
Out:
Technically speaking, "Out" isn't really a command, it's a "register". We use the "out register" to make an output either high or low. In a future experiment, we'll explore registers in greater detail.

Think of a light switch on the wall, when the switch is in one position the lamp is on, & when it is in the other position, the lamp is off. It's binary – there are only two possible combinations, on or off, "1" or "0". No matter how hard you try, you can never put the light switch "in between" on and off positions.

If we want to turn the LED on, we need to cause P0 to go low (or become a 0). P0 is acting as a switch that can be "flipped" on or off, under program control! Simplified circuits are shown in Figure 9 (LED off) and Figure 10 (LED on). Current flow is from +voltage through the resistor, LED, and into P0, where P0 is "connected " to ground.

Figure 1.9: LED off
When P0 is "high" there is no current flow

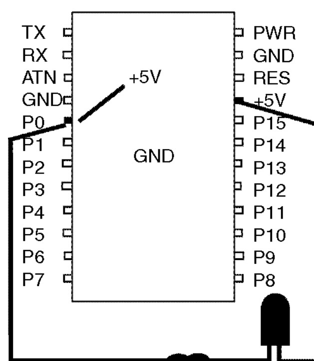
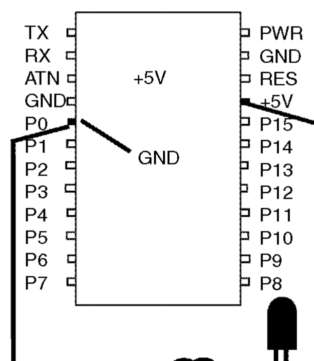


Figure 1.10: LED on
When P0 is "low" and current flows, the LED is on.



This is the purpose for the second command: "**`Out0=0`**". This will cause P0 to go "low", which causes the LED to turn on.

Keep in mind that microcontrollers execute their programs very quickly. In fact, the BASIC Stamp will execute about 4000 instructions *per second*.

If we were to turn the LED off with the next command, it would happen too quickly for us to see. Therefore, we need to "slow" the program down, so that we can see whether or not it's operating properly.

That's the purpose of the next command: "**`Pause 1000`**". This command causes the program to wait for 1000 milliseconds, or 1 second.

The next command is "**`out0=1`**". This command causes the P0 to go high, and turn the LED "off" because there is no current flow.

Next we "**`pause 1000`**" (for another second). The LED is still "off".

"**`Goto`**" is pretty much self-explanatory. During the course of program execution, when the "goto" command is encountered, the program "goes to" some other point in the program. In our example, we tell the program to "**`goto reblink`**". Wherever **`reblink`** appears, the program will "jump to".

In our program, the label **reblink** is on the second line. Therefore when the instruction **goto reblink** is reached, the program jumps back to the second line, and "loops" or does it again. (Hint: The program loops over and over each time it encounters the **goto reblink** command. This is what causes the LED's to continuously flash on and off).

A good habit to establish is to **remark** your programs. Remarking or documenting your programs makes them easier to follow and debug if there's a problem.

The apostrophe (') is used to tell the microcontroller to "disregard the following information", it's only for human benefit. In other words, anything in a program line written after an apostrophe is not part of the instruction "code".

What's a...
Remark:
"Remarks" in your program are not executed like commands. They are ignored by the micro-controller. The purpose of a remark is to allow us humans to more easily understand what the commands in the program are doing.

'So, our program could be "remarked" like this:

```
output 0            'make PO an output
reblink:            'this is where the loop begins
 out0 = 0           'turn on the LED
 pause 1000         'wait for 1 second, with the LED on
 out0 = 1           'now turn off the LED
 pause 1000         'leave the LED off for 1 second
goto reblink        'go back, and blink the LED again
```

The program will still operate exactly the same way, the "remarks" after the apostrophes are only for our benefit in understanding what we've created.

Note that throughout this experiment we have used the **pause** command to wait for x milliseconds. Keep in mind that instructions also require execution time. For example, the setup time for **low**, **high**, and **pause** commands are about 0.15 milliseconds each. On average the BASIC Stamp executes 4,000 instructions per second.

FYI: A Simpler Way

Remember that each of the pins on the Stamp (P0-P15) can be configured as an input or an output. In order to make the pin an output, we use the command: **output**. Once the pin is an output, we can make it go "low" (a logic level 0) or "high" (a logic level 1), with the **OUT0=0** statement (for low) or **OUT0=1** (for high). Using these commands, it takes two lines in our program to make the pin an output & then make it go high or low.

PBASIC has made it even simpler to do this. If you wish to make P0 an *output and high* (at the same time), simply use the command: **high 0**; and conversely, to make P0 an *output and low* (at the same time) use: **low 0**.

Our example program now would look like this:

```
reblink:
  low 0
  pause 1000
  high 0
  pause 1000
goto reblink
```

The program functions exactly the same, it's just that the new commands not only cause the pin to go high or low (like "**out0=0**" and "**out1=1**") but they also cause the pin to become an output. In simple cases (like this program), either method will suffice, but in more complicated programming, one method may be more appropriate than the other. We'll explore this in a future lesson.

Questions

1. How does a microcontroller differ from a computer?

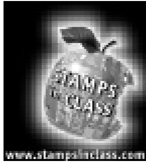
2. What is the difference between hardware & software?

3. Why is a microcontroller like your brain?

4. What does "debug" mean?

5. The following program should turn on the LED on P0 for 2 seconds, then off for 2 seconds, & then repeat. How many "bugs" are in the program, & what corrections are needed?

```
output 0
reblenk:
  out0 = 0
  pause 200
  out1 = 1
  pause 2000
goto reblink
```



Challenge!

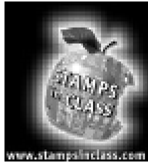
Rewrite the program in Question #5 above to do the following. Each program should be loaded into the BASIC Stamp and tested.

1. Blink both LED's on and off at the same time. When you've finished, enter the program into the PC (just like you did before), & try it.

2. Alternately blink the LED's on and off. In other words, while one LED is on, the other is off, and vice versa – just like a railroad-crossing signal.

3. Turn on the first LED on for 2 seconds, then off. Wait 5 seconds, and then turn on the second LED for 1 second & then off. Wait 3 seconds, then repeat.

4. Turn on the first LED for 1.5 seconds, then off. Wait 2 seconds, and then turn on the second LED for 1.5 seconds and then off. Wait for 2 seconds. Then blink both LED's on for ½ second and off for 2 seconds. Recycle and repeat this ½ on, 2 second off blinking.



What have I learned?

On the lines below, insert the appropriate words from the list on the left.

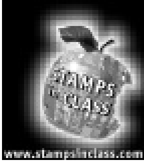
embedded
executed
microcontrollers
adapt
programmable
sequence
download

______________________ are all around us. Even when they don't look like a computer. (Who would have ever thought that a kid's toy would have a computer embedded inside of it?)

Microcontrollers consist of hardware and _______________ software. We create programs on a PC, a computer that is designed for human interaction (with a keyboard, monitor, etc.) & then _______________ the program to a microcontroller, where it is actually ___________________ ("run").

A microcontroller program is only as smart as the programmer. Unlike the human brain, the microcontroller program will not ___________ itself, and change the order of program instructions. The microcontroller will execute a set of instructions in the exact ______________ in which they were created.

Many microcontrollers are versatile and easy to use, because they may be re-usable, _________________, and can be designed into an unlimited number of products and innovations, from robots to toasters.



Why did I learn it?

The real versatility of microcontrollers is that they can be programmed to control just about anything the human mind can conceive. Model airplanes, "smart home" controllers, or battery operated remote weather data collection systems, are just a few examples.

Microcontrollers must have two components brought together, in order for the device to work. The first component is hardware. Many people make their living designing microcontroller hardware for an infinite variety of products. The second component is software. Programmers specialize in writing "control code" for cell phones, pagers, toys, or even industrial equipment.

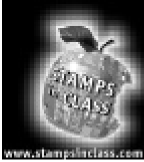
How can I apply this?

The neat part about microcontrollers (and something you may consider as a future career), is that the world of "smart devices" is expanding at an incredible rate, and doesn't show any sign of slowing down. As technology advances in all areas of our lives, we're surrounded by an ever-increasing number of technologically advanced machines and gadgets. You can help develop these, and perhaps invent the next "great product", or just have fun, building "stuff". The technology is the same, it's just applied differently!

Look around you and think about how you could use a microcontroller to create a flasher for your bicycle taillights, running lights for a truck, or an art project which uses some lights to interact with viewers.

Brainstorm with your friends, develop a nifty product and start a business...

...Who knows?



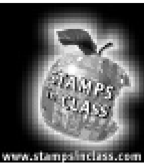
Experiment #2: Detecting the Outside World

Making Decisions. Our brain does it all the time. We make decisions based on what we see, hear, touch, etc. As we learned in Experiment #1 (What's a Microcontroller?), microcontrollers act like our brain – they manipulate the "real world" based on "inputs".

Experiment #2 will focus on how we can design a microcontroller system that can change its "outputs", depending on what kind of digital "inputs" it detects.

Microcontrollers are programmable devices. That is, they contain a certain list of instructions (called a "program" or code), that tells it what to do, given certain circumstances.

The BASIC Stamp is programmable in BASIC, a computer language that is easy to learn, and yet has some very powerful features. Let's explore how we can make microcontrollers react to, and control the "real world".



Parts Required

Experiment #2 requires the following parts:

(1) Programming Cable
(2) LED's (light emitting diodes)
(2) Push button switches
(1) BASIC Stamp II
(1) "Board of Education"
(2) 470 ohm, ¼ watt resistors (yellow, violet, brown)
(2) 10k ohm, ¼ watt resistors (brown, black, orange)
(1) 9 volt battery or wall transformer
(6) Connecting wires
(1) BASIC Stamp Editor program, either the DOS or Win95 version

What's a...

Sensor:
A sensor is an input device used to detect or measure physical presence. Examples include sensors that detect light, heat, temperature, stress, and chemicals (such as carbon monoxide).

There are an infinite variety of sensors that we can connect to the BASIC Stamp. This experiment will include a push button switch (a type of sensor) and an LED (an output device).

Build It!

Using the Board of Education shown in Figure 2.1, build the circuit as shown in Figures 2.2 and 2.3. Figure 2.2 is the pictorial (what the circuit physically looks like), and Figure 2.3 is the schematic representation of the same circuit. Some of this will be familiar from Experiment #1.

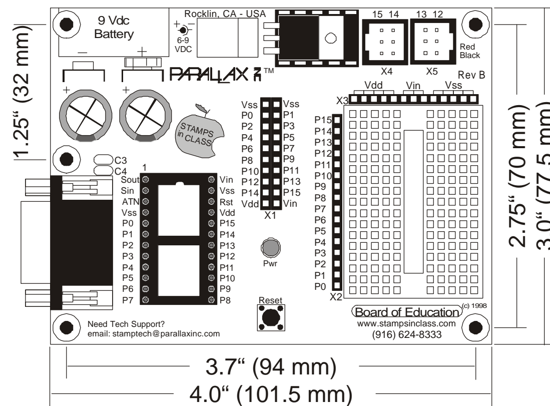


Figure 2.1: Board of Education Construction platform for Experiment 2.

Remember that the LED has to be connected with the flat side connected to the P0 and P1 output pins. Be sure that there is a 470 ohm resistor in series with each LED, and that there is a 10,000 ohm resistor connected to the “high side” of each push button switch.

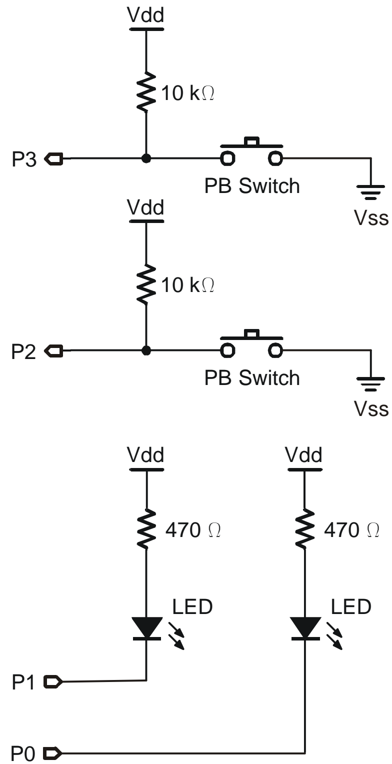


Figure 2.3: Schematic
Electrical drawing of circuit.

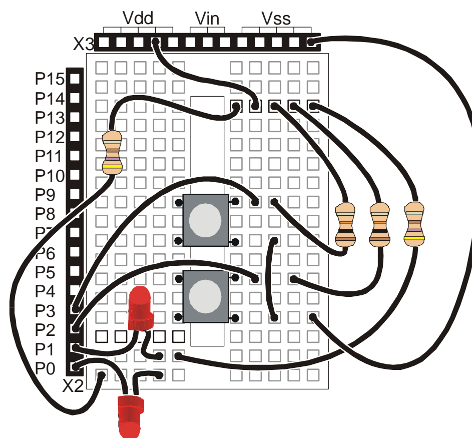


Figure 2.2: Pictorial
What the circuit should look like after you build it. The flat side of the LED is nearest the BASIC Stamp I/O pin.

Since this experiment uses two different values of resistors, and they look the same, how can you tell them apart? By reading the color-coded bands. If you don't know how to read the "code", jump to Appendix C to find out how.

This circuit has one type of sensor (the push button switch) and one type of output device (the LED).

Program It!

Once you have all the components installed into the prototype area, as shown in the figures, attach the programming cable from the Board of Education to your PC and connect a power supply.

Start the BASIC Stamp II Windows Editor by clicking on the icon.

As we learned in Experiment #1, the screen, except for a few words across the top, is blank. This is where we will write our control programs.

Also recall that the program that we will write will not run on the PC, but rather will be "downloaded" or sent to the microcontroller. Once the program has been received, the BASIC Stamp will execute the instructions exactly as we've created them.

Type in the following program:

```
output 0
out0=1
Input 2
recheck:
  if in2=0 then blink
goto recheck

blink:
  low 0
  pause 200
  high 0
  pause 200
goto recheck
```

Now while holding the "ALT" key down, type the letter "r" (for "run").

If the program does not download properly and you get a message saying "error" you may have a bug. If you get a message saying "hardware not found" check the cable connection between the Board of Education and

the PC. The BASIC Stamp II Windows program will help you find bugs. If it finds one it will tell you during download, but you can also press ALT-M to find syntax and typos.

FYI:

Input:

Technically, we don't really have to give the BASIC Stamp this command, because "at power-up" (when you first apply power or push the reset button), every pin is automatically configured to be an input. However, as you develop increasingly complex circuitry, it is possible to change the status of pins from an input to an output, and vice-versa, all under program control. We're including this command to help clarify the difference between input & output.

Try downloading again (hold down the ALT key, & then press "r"). If it still doesn't work, you may have a bug! Re-check your program to be certain you've typed it correctly.

If after trying this, you're still having problems, ask you instructor for help.

If your program is working properly, the LED should blink on and off only while you're pushing the switch.

Now lets dissect, and look at our program:

Our first command, **`output 0`**, makes the P0 pin an "output".

The instruction **`Out0=1`**, actually sets the output register of P0 to a value equal to "1", or "high". Since the LED is connected to Vdd (high) and the P0 pin is also set to "high" there is no current flow, therefore the LED is off.

"**`Input 2`**" tells the BASIC Stamp that P2 should be configured as input.

Recall from Experiment #1 that a command like **`recheck`**: really isn't a command, it's a "label" – simply a marker or pointer to a certain point in our program. When our program eventually reaches the command **`goto recheck`**, it (the program) looks for the label **`recheck`**, jumps to it, and then continues execution from that point on.

The command **`if in2=0 then blink`** instructs the microcontroller to "check the status of (the pin called) "P2". When a microcontroller "checks the status" of a particular pin, what it is really doing is reading the digital value of that pin.

The electronics field is generally divided into two different "signal type" domains – Digital & Analog. Our switch input is either a "0" or a "1" (open or closed). This is a digital type of sensor. Measuring the level of a swimming pool and a dimmer light switch are examples of an analog input. Sensors for this type of application convert a measurement, for example, of 4.3" to a digital value that a microcontroller can understand. We'll explore this fascinating world of Analog to Digital Conversion in an upcoming experiment.

What's a...

Analog:

A continuously variable value. In stead of either 1 or 0 (+5 or ground), analog values can be anywhere in between two extremes. Since microcontrollers only understand inputs if they're digital values, our sensors and associated "interface" circuitry need to convert analog values (voltages) to digital equivalents.

Binary:

The number system used by all microcontrollers (as well as full-fledged computer systems). We normally use the 10 digits (0-9). Digital electronic systems only work (on the very lowest levels) with two digits, 0 & 1.

In digital (binary) electronics, any value other than 0 or 1 is considered to be invalid. Therefore, when the microcontroller "reads" the status on P2, it will see a value of either 0 or 1. We're looking for P2 to be a "0". If (when the microcontroller checks it) it's a "1" then this command will do nothing, and the program will continue executing the next command (in this case, "goto recheck" – which causes the program to loop back & continually look for P1 to become a "0"). Once P2 becomes a "0" then the conditions for this command are met & the program jumps to "blink" (a label defining the location of another "routine").

The "blink:" routine should look familiar to you. It is simply a short program that causes the LED to turn on (for .2 seconds) and then turn off (for .2 seconds). After blinking the LED, the command "goto recheck" causes the program to go back and check the status of P2 again, and "do it again".

Our program with remarks, now looks like this:

```
output 0                    ' make P0 an output
out0=1                      ' make P0 "high"
Input 2
recheck:                    'a label
if in2=0 then blink         'check to see if P2 is a "0", if it is
                            'then go an blink the LED
goto recheck                ' if P2 was a "1", then go back & 'check again

blink:                      'a label
low 0                       'turn on the LED
pause 200                   'wait .2 seconds
high 0                      'turn off the LED
pause 200                   'wait .2 seconds
goto recheck                'loop back & do it again
```

Let's write a new program. Type the following into the BASIC Stamp Editor:

```
output 0
output 1
Input 2
Input 3
out0=1
out1=1
recheck:
if in2=0 then blink
if in3=0 then double_blink
goto recheck
blink:
low 0
pause 200
high 0
pause 200
goto recheck

double_blink:
low 0
low 1
pause 200
high 0
high 1
pause 200
goto recheck
```

Before you run this program, can you tell what it's going to do?

The program is going to "make a decision", based on which button is pressed. Once either button is pressed, the program will jump to the appropriate routine. The microcontroller is sensing an input, making a decision, and then creating an output. The decision (should I flash one LED or two), is made based on which button is pressed.

The command **`if in2=0 then blink`**, looks directly at the pin and then makes a decision based on the status of that pin. The command **`if in3=0 then double_blink`** checks the other button. What happens if both buttons are pushed? Why?

There may be times when we need to get the status of an input (high or low) and then continue executing the program without making a decision just yet. Maybe several different inputs need to be in a certain state, before an action should occur.

If we wish to have an LED blink only when both buttons are pushed, our current program won't work quite the way we'd like.

A variable allows us to store a certain piece of information (collected now), for later analysis.

> **What's a... Variable:**
> A variable is a symbol that contains a certain value. That value can be changed under program control, and therefore the variable's value may change, but the symbol name doesn't. Variables can store certain pieces of information now, for use at a later time in the program.

Variables must be "declared" before using them in a program. Declaring a variable is simply a statement in your program that tells the microcontroller the name of the variable and how big it is.

Let's modify our program to demonstrate one way in which a variable can be used. Type in the following program:

```
x var bit
y var bit

output 0
output 1

out0=1
out1=1
Input 2
Input 3

get_status:
x=in2
y=in3

if x+y=0 then double_blink
goto get_status

double_blink:
low 0
low 1
pause 200
high 0
high 1
pause 200
goto get_status
```

The first lines, **`x var bit`**, tells the microcontroller that we're going to use a variable called x, and it will be a bit in size. Since we're only interested in what the state of the pin is ("0" or "1"), a single bit is all we need for the variable length. In the second line, the same goes for "y".

In the **`get_status:`** routine, two things happen. In **`x=in2`** the microcontroller will look at the input on P2, then (unlike our previous program), will not jump to another location but rather it will store the value (again "0" or "1") in our variable called "x". The program does the same for "y" in the next line.

Now, at this point in program execution (after pins P2 & P3 are read), it doesn't matter what happens to the P2 and P3 inputs, the values that they were when read, are stored as variables x and y.

We can now perform some "operations" on these variables. For example, in our program we're adding the value of "x" plus the value of "y" together. If the answer is zero, then our program will jump to the label in our program called **`double_blink`**. Therefore, both buttons need to be pushed in order for the **`double_blink`** routine to be called.

Variables don't have to be single characters. In our sample program, we're simply using the letters "x" and "y".

Try this: Use different names for x and y, maybe "Bonnie" & "Clyde". When you create more complicated programs, giving a variable a name that means something to you can help in understanding how your program is operating (especially if you need to debug it!).

FYI: More about Variables

In PBASIC, variable names can be up to 32 characters in length. The length of the name has no bearing on the execution speed of the program. For example, the statement: x = in6, will have the same execution speed as: this_is_a_very_long_name = in6.

Variables can be declared in 4 different sizes: bit (1 bit), nib (nibble - 4 bits), byte (8 bits), & word (16 bits)

You should always declare your variables in the smallest size that is appropriate for the data it will contain. The Stamp2 has a limit of 208 bits of variable "storage". These are arranged into 13 words (consisting of 16 bits each). These bits can be used in any combination of the above sizes. For example, your program could have 10 word variables (160 bits), 10 nibble variables (40 bits), & 8 bit variables (8 bits), or any other combination so long as the total doesn't exceed 208 bits.

See the BASIC Stamp Manual Version 1.9 for additional information about using variables efficiently.

Questions

1. How does a microcontroller make a decision?

2. What's a sensor & why does a microcontroller need one? Name some different types of sensors.

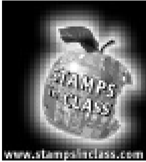
3. Define a variable, & describe how they might be used in a program.

4. Write the code to declare a variable called "status". The variable could be either "0" or a "1".

5. Add appropriate remarks to the following program:

```
output 0                          _______________________________________
output 1                          _______________________________________
out0=1                            _______________________________________
out1=1                            _______________________________________
recheck:                          _______________________________________
    if in2=0 then blink           _______________________________________
    if in3=0 then double_blink    _______________________________________
    goto recheck                  _______________________________________
blink:                            _______________________________________
    low 0                         _______________________________________
    pause 200                     _______________________________________
    high 0                        _______________________________________
    pause 200                     _______________________________________
    goto recheck                  _______________________________________
double_blink:                     _______________________________________
    low 0                         _______________________________________
    low 1                         _______________________________________
    pause 200                     _______________________________________
    high 0                        _______________________________________

    high 1                        _______________________________________
    pause 200                     _______________________________________
    goto recheck                  _______________________________________
```



Challenge!

1. Write a program (complete with remarks) that will blink LED P0 on and off (every ½ second), as long as switch P2 is pressed. When the button is not pressed, LED P1 is on, but goes off when LED P0 is flashing.

2. Write a program that will blink both LED's (every 1.2 seconds) when either switch is pressed. If no switches are pressed the LED's are on & if both switches are pressed both LED's are off.

3. Write a program that will alternately blink the LED's on and off (continuously) every ½ second, but only after switch P2 has been pressed first (and released) , and then switch P3 is pressed. Then, note (remark) in your program, what would change if you wish to reverse the "switch press" order.

4. Write a program that will blink the LED's (every .2 seconds) whenever switch P2 is pressed. Then while switch P2 is still depressed, LED P1 is turned off when switch P3 is pressed – but LED P0 is still blinking.

What have I learned?

On the lines below, insert the appropriate words from the list on the left.

- I/O pin
- declared
- inputs (or sensors)
- if in1=0
- program
- activated
- debug
- variables

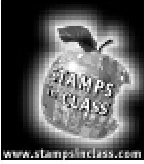
Microcontrollers need ________ in order to know what is going on in the "real world". By using PBASIC commands such as _______________", our program can determine what kind of output is appropriate.

There are an infinite variety of sensors that can be interfaced to the BASIC Stamp. Although the switches that we used in this experiment were ________________by us pushing them, they could just as easily have been the switch sensors on an elevator door – the ones that keep you from getting squished as the door closes.

_________________ are used to hold information, enabling our program to gather data now (perhaps from several different inputs), and then to make decisions at a more appropriate time

Variables can be _______________, or "set up", in 4 different sizes. If we wish to check the status (high or low) of a particular __________, then we can set the variable up as a single "bit".

Variables can be up to 32 characters in length. The important thing to remember when using variables, is to give them a name that means something to you. The length of the name will have no influence on how quickly your __________ executes, but a "very descriptive" name will make it much easier to _______ your program, should the need arise.



Why did I learn it?

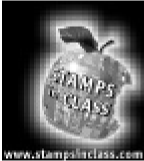
The very heart of microcontrollers is their ability to make decisions based on inputs. Inputs to a microcontroller have to be in a digital format, but many types of "real world" situations are analog in nature.

Sensor technology is one of the most challenging areas of electronics. There are hundreds of different types of sensors on the Space Shuttle & the satellites that it carries into orbit.

Many people specialize in designing sensors that interface to microcontrollers. If you like to work on "hardware", instead of writing programs (creating software), this could be a very exciting and ever challenging field.

Any microcontroller (or for that matter, computer) system relies on digital inputs, in order to make correct decisions. It is important to realize that microcontroller decisions are only as good as the program that it's running & the quality of the sensory inputs it gets.

The more you look around, the more applications you'll see for microcontroller & sensor technology.



How can I apply this?

Many retail stores have some sort of "door beeper" that chimes every time somebody goes in or out of the door. Every time the beeper goes off, the proprietor has to look and see who came in.

Using a proximity sensor that detects the presence of an object (similar to a button being pushed), you could detect when somebody goes through the door. By using three sensors you could determine which direction they are travelling. Then, using a another tone, the system could alert you differently when somebody was only leaving.

Experiment #3: Micro-*controlled* Movement

3

So we're already on Experiment #3 and "all we've done" is blinked a few LED's on and off. Hang in there, something is about to *move*!

As you know, an LED is an "output" device. A microcontroller can blink LED's as well as control all sorts of other (*sometimes movable*) output devices under program control.

> **What's a...**
>
> Interface circuitry:
> Microcontrollers operate on very small voltages & signal levels. They don't have enough "drive" capability to operate large, heavy duty types of output devices.
>
> Consider your "Walkman" as a microcontroller. It can drive small outputs (like head phones) by itself but to control a large device (like big speakers) you will need an interface circuit – an amplifier.
>
> The BASIC Stamp can control small motors on your table-top robot, or with the appropriate interface circuitry, it can operate the motors that open the flood gates on Hoover dam. It all depends on your "interface circuitry".

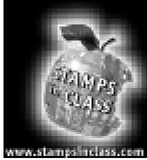
Although the control methods are very similar, other types of devices such as motors can give us a much more tangible example of "real world" manipulation.

There are many different types of motors that the BASIC Stamp can control. Most motors however, require some type of external "interface circuitry" which enables our microcontroller to control them. In this experiment, we're going to use a specialized type of DC motor. It's called a "servo".

What's a servo? A servo is a DC motor which has some "interface circuitry" already built in. This makes it extremely easy to connect to a microcontroller (such as the BASIC Stamp). The type of servo that we'll be using was originally designed for use in radio-controlled cars, boats, and planes.

Rather that continually rotating, like a standard type of hobby motor, a servo is "position-able". You can, by sending the appropriate signals from the BASIC Stamp, have the servo rotate to a specific point, and stay there.

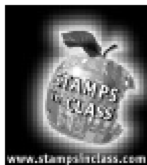
Servos have many applications, as we're about to explore.



Parts Required

Experiment #3 requires the following parts:

(1) BASIC Stamp II
(1)"Board of Education"
(1) Three pin "double male" connector
(1) Programming Cable
(1) RC servo
(1) LED (light emitting diode)
(1) 470 ohm, ¼ watt resistor
(1) 3000 microfarad electrolytic capacitor (only required for Rev. A Board of Educations)
(1) 9 volt battery or wall transformer
(6) Connecting wires
(1) BASIC Stamp Editor program, either the DOS or Win 95 version



Build It!

A picture of a typical servo is shown in Figure 3.1. Servos come in many shapes and sizes, depending on their application.

Using the Board of Education, create the hardware circuit as shown in the figures below.

> **What's a...**
>
> Vdd & Vss:
> These are the designations that are used for *plus voltage and ground.* In our circuity (on the Board of Education) Vdd is equal to +5 volts, & Vss is equal to zero volts. This is a fairly common set of values for most computer systems, however these values may vary depending on what other types of electronic devices may be in the circuit.

Figure 3.1: Servo
Radio control (R/C) servo

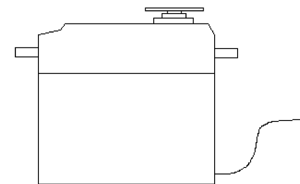


Figure 3.2 is the pictorial (what the circuit physically looks like), and Figure 3.3 is the schematic representation.

Depending on which model of servo you have, the color coding on the wires may vary. In all cases (with the servos you get from Parallax), the black wire is connected to Vss and the red wire is connected to Vdd. The remaining (third) wire may be white or yellow (or something else). This is the control input wire which we'll be conneting to the P1 signal on the BASIC Stamp.

3

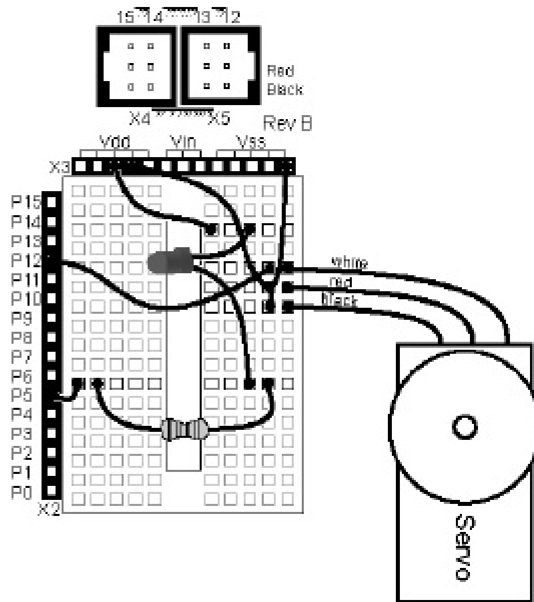


Figure 3.2: Pictorial of Servo Connection to Rev. B. Board of Education

The Rev. B Board of Education uses the three-pin header to connect the servo into the breadboard, where the white I/O must be jumpered to the I/O P12. Do not use the header as it supplies Vin (unregulated 9V supply will damage the servos).

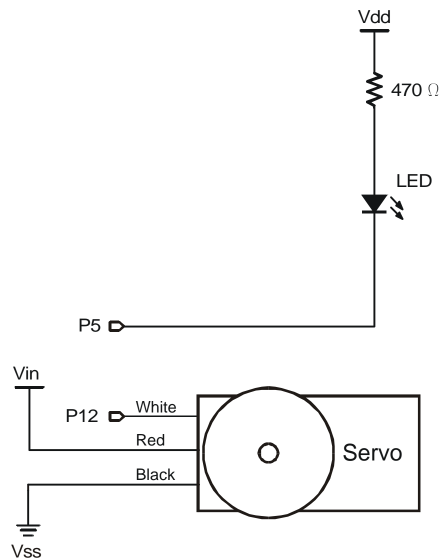
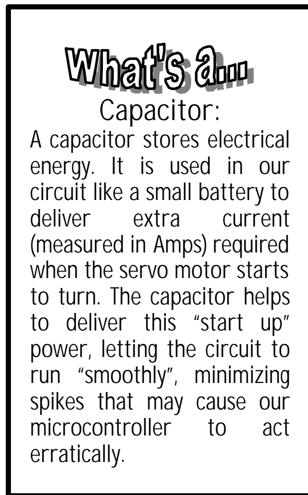


Figure 3.3: Schematic of servo connection for Rev B. Board of Education

Note: Rev A. Schematic is on the following page


What's a...

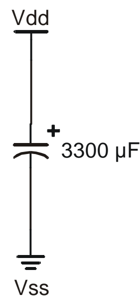
Capacitor:

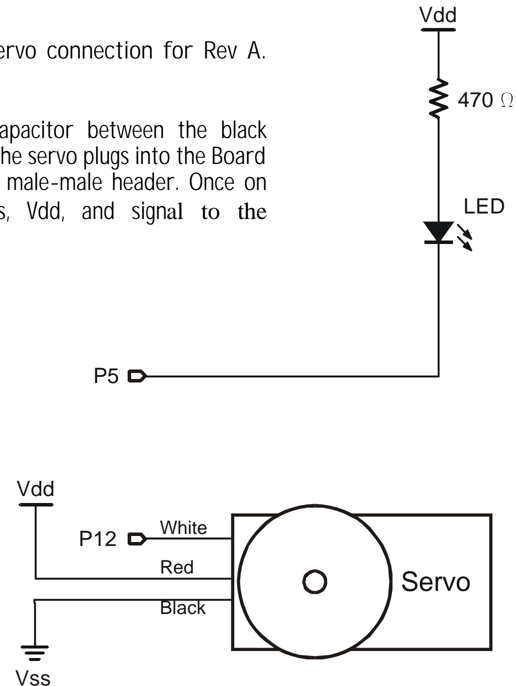
A capacitor stores electrical energy. It is used in our circuit like a small battery to deliver extra current (measured in Amps) required when the servo motor starts to turn. The capacitor helps to deliver this "start up" power, letting the circuit to run "smoothly", minimizing spikes that may cause our microcontroller to act erratically.

Figure 3.3: Schematic of servo connection for Rev A. Board of Education

Note: Place the 3300 uF capacitor between the black header's Vdd and Vss points. The servo plugs into the Board of Education using the 3-pin male-male header. Once on the board, jumper the Vss, Vdd, and signal to the appropriate locations.

Capacitor Required on Board of Education Rev A





Be sure that there is a 470 ohm resistor in series with the LED. As we learned in a prior experiment, this will limit the current flowing through the LED to a safe amount. Too much current flowing through the LED will burn it out and may damage the BASIC Stamp as well.

The capacitor (the cylinder with two wires) has a polarity designation on it as well, and is required when building this project on the Rev. A Board of Educations (it is not required for Rev. B and subsequent Board of Educations). For Rev. A boards it is important that you connect the minus (-) lead of the capacitor to Vss and the positive (+) lead to Vdd. Reversing this connection could damage the capacitor. See Figure 3.3 for the additional schematic that applies to Rev. A Board of Educations.

This circuit has two types of output devices (an LED and the servo).

Once you have all the components installed into the prototype area, (as shown in the figures), attach the programming cable from the Board of Education to your PC & connect *either* a 9 volt battery or a 9 volt DC wall transformer to the Board. Since the servo requires a lot of current (much more than an LED), battery life will be quite limited, so use the transformer if you have one.

Program It!

Turn on your PC, and double click on the BASIC Stamp icon.

You should now be running a program called the "BASIC Stamp Editor". This is a program that was created to help you write and download programs to the microcontroller.

Type in the following program:

```
output 5
here:
out5=1
pause 200
out5=0
pause 200
goto here
```

Now while holding the "ALT" key down, type the letter "r" (for "run") and press "enter".

Problem?

If you are using the DOS BASIC Stamp editor and you get a message that says, "Hardware not found", re-check the cable connections between the PC and Carrier Board, & also make sure that the 9 volt battery (or wall transformer) is connected & charged.

Try downloading again (hold down the ALT key, & then press "r"). If it still doesn't work, you may have a bug! Re-check your program to be certain you've typed the program correctly.

After checking your connections, press ALT "r" again. If you still receive the "hardware not found" message, then make sure your computer is running in DOS, not Win95. If it is running in Win 95, then press the Start button (on the monitor), and select "Restart in MS-DOS mode".

If after trying this, you're still having problems, ask your instructor for help.
After checking your connections, press ALT "r" again. If you still receive the "hardware not found" message, then make sure your computer is running in DOS, not Win95. If it is running in Win 95, then press the Start button (on the monitor), and select "Restart in MS-DOS mode".

If your program is working properly, the LED should be blinking.

But we've done this before. It's just a simple LED blinking program, why are we doing it again?

The answer is that we are about to use a more sophisticated PBASIC command, and this simple blinking routine will help us to understand how the new command works.

What's a...

Millisecond:

Computers and micro-controller systems operate at very fast rates. As humans we are used to time measurements in the seconds range, or in the case of athletic competition, 10ths or even 100ths of a second. A millisecond is 1 / 1000 of a second, i.e. there are 1000 milliseconds in one second. This seems like a very small amount of time, but it is actually quite long in the micro-electronic world of computers. In fact, your personal computer (that you're using to write these PBASIC programs) is probably operating in the *millionths* of a second range!

Timing Diagram:

Computers operate on a series of pulses, usually between 0 and 5 volts. A timing diagram is simply a visual way to show what the pulses look like. You "read" a timing diagram from left to right - which is really a *duration* of time. In our sample diagram, we see that the voltage (on our output pin P1) starts at 0 volts. After a short time period, we see that P1 pulses high for a duration of between 1 and 2 milliseconds, at which time it returns to 0 volts. After approximately 10 milliseconds, P1 pulses high again. Unless otherwise noted in the diagram, you can assume that the process repeats itself, i.e. when you get to the right side of the diagram, go back to the left, & start over again.

Try changing both **pause** statements to values of only 100 (instead of 200). Now change the pauses to 50. Now 30. Now 20. Now 5. What's happening?

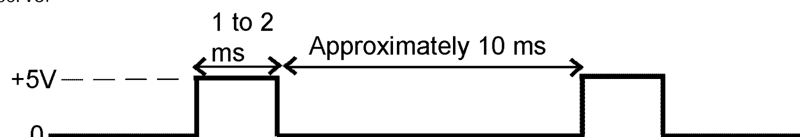
The LED is blinking faster and faster because the time of each pause is getting shorter each time you decrease (the Pause) values. When you reach a certain blink rate, our eyes see the LED as on all the time. It really isn't. It's just blinking at such a high rate, that our eyes can't see the individual *pulses* of light.

Ok, so what? Well, a servo is controlled by a stream of pulses that are *between* 1 and 2 milliseconds in length. This pulse recurs about every 10 milliseconds.

Recall that the **pause** command is set in milliseconds, and that the smallest pause length we can have is 1 millisecond. The next (available value) is 2 milliseconds (ms).

So what about the servo? A servo needs to have a stream of pulses (on the white or yellow "control" wire) that vary *between* 1 and 2 ms in length. With a stream of pulses that are a constant 1 ms in length, the servo will be positioned at one extreme of its rotation. As the pulse width increases (1.1ms, 1.2ms, 1.3ms... etc), the servo changes its position. When the pulse width reaches 2.0ms the servo is at the other extreme of its rotation. These pulses need to occur at about 10 ms intervals. Figure 3.4 is a timing diagram of the pulses needed by the servo.

Figure 3.4: The pulse stream for a typical servo: Timing diagram of pulses needed by the servo.



Ok, armed with this information lets write a program that will make the servo move to one (extreme) position, stay there for a short time and then move to another position, remain there for a short time, and then repeat.

3

Type in the following program:

```
x var word
output 12
here:
for x = 1 to 100
  pulsout 12,500
  pause 10
next
pause 500
for x = 1 to 100
  pulsout 12,1000
  pause 10
next
pause 500
goto here
```

Now while holding the "ALT" key down, type the letter "r" (for "run").

If your program is working properly, the servo should be rotating from one (extreme) end of its rotation to the other, then returning back and doing it again.

Servos are not designed to fully rotate (like a standard motor that you might use on a robot's drive wheels). Instead, they are used for positioning types of applications. Examples would include opening and closing valves, or a robotic manipulator arm. However, if you continue your study of microcontrollers you'll find that servos are often "hacked" so they can rotate continuously for use in robotics.

FYI:

Servo Modifications:
Although they're not specifically designed for full rotation, servo's can be modified to allow them full rotary motion. A method for this modification is outlined in "Programming & Customizing the BASIC Stamp", by Scott Edwards. See the Appendix for more information.

Let's explore the program:

`x var word`

Recall that in order for the BASIC Stamp to know what variables are being used, we need to "declare" them in our program. This command tells the BASIC Stamp that we will be using a variable called "x", and that it will be one "word" in size. A "word" variable 16 bits and can hold a value between 0 and 65,536 in our decimal number system. Because we're only using "100" as the maximum value in our program, we could have set this variable up as a *byte* variable, using 8 bits and capable of storing a value between 0 and 255 (decimal). The word bit comes from binary digit.

`output 12`

This we already know – it makes P12 an output.

`here:`

Simply a label, marking a place in the program.

`for x = 1 to 100`

For those of you who have written programs in (other types of) BASIC, this command may look familiar. This is the beginning of a "**`for . . . next`**" loop. It simply says that the first time this command line is encountered, that our variable "x" will be set to the value of "1". The program goes on to the next command and continues program execution until it encounters the command called "**`next`**".

Upon reaching "**`next`**", the program loops back up to the "**`for x = 1 to 100`**" command, and increments the value of "x" by one. The program then continues to loop over and over (incrementing "x" each time) until the value of "x" = 100. When "x" = 100 (i.e. when this part of program has "looped" 100 times) the program will exit the "loop" and execute the command immediately after "**`next`**".

We are sending a string of 100 pulses to the servo to allow it enough time to mechanically react to the signal stream. The microcontroller can operate much faster that any "real world" mechanical device, and by looping 100 times, we're giving the servo enough time to "catch up" to the BASIC Stamp.

3

`pulsout 12,500`

This is a very handy command in the I/O world. Many times we need to have a very stable output pulse generated by our microcontroller in order to precisely control hi-tech devices (such as our servo). To implement what this command does using the techniques that we used to blink the LED really isn't feasible because our servo requires pulse widths of *between* 1 and 2 ms. "Pause" just can't provide the resolution that we need – it jumps *from* 1 *to* 2 milliseconds. This is the reason we created the LED blinker program earlier. What took 4 or 5 lines of code (with inadequate resolution) can be accomplished with this single command. And with a resolution that is measured in *microseconds!*

`Pulsout 12` does exactly what its name implies. It creates a *single* pulse output on I/O pin P12. The "500" is a value that determines the duration of the pulse. As mentioned above, this duration is measured in microseconds. **`Pulsout`** has a resolution of 2 microseconds, therefore a value of 500, would yield a pulse length of *500 times 2 microseconds*, or 1000 microseconds (which equals 1 millisecond – the value required for the servo). A value of 1000 would create a pulse length of 1000 x 2 microseconds = 2 milliseconds – the required pulse width for the servo's other extreme.

`pause 10`

Nothing new here – we already know what **`pause`** does, but the reason that we're pausing here may not be readily apparent. The specifications for servo control (at least for the servo's we're using in this experiment) dictate that the stream of pulses going into the servo must be approximately 10 milliseconds apart. By pausing 10 ms at this point, we're controlling the flow of pulses to fit the servo's specifications. Again, see the *timing diagram* in Figure 3.4.

`next`

At this point the program will loop back to the prior "**`for x = 1 to 100`**" command and output the next pulse, unless it's already looped (in this example) 100 times. If "x" has reached 100 at this stage, the program will continue execution beyond this command.

`pause 500`

This command is executed when the (above) **`for…next`** loop has finished. This is just a pause so we can see the servo stop before it turns again.

`for x = 1 to 100`

We're headed into another loop. This one is identical the first loop, with the exception that the pulsout length is 2 milliseconds. This causes the servo to rotate to its other extreme.

`pulsout 12,1000`

Create a single pulse with a duration of 2 milliseconds.

`pause 10`

Again, we need to wait for 10 milliseconds before continuing in our "loop".

`next`

If "x" hasn't incremented to 100 yet, the program will loop back up to the prior "**`for x=1 to 100`**" command. Note that it will loop back to the "for" command that is immediately prior to this "next" statement. (***Not*** *all the way back up the first "for...next" loop*).

`goto here`

Go back up and do it all over again.

Ok, let's recap what our program is doing.

> **What's a...**
>
> Initialization:
>
> The first part of many programs is sometimes referred to as the "initialization routine". All this means is that this portion of the program "sets up" all the various parameters that the program will be using.

After initialization, the program will send a stream of 100 pulses, each pulse being 1 millisecond in length. This will cause the servo to rotate to one extreme end of its rotation.

Then, the BASIC Stamp will send out another series of 100 pulses (again, utilizing the "**`for...next`**" loop), this time however, the pulse widths are 2 milliseconds in length. This causes the servo to rotate to it's other extreme position.

The program loops back and does it all over again.

Now, let's try something interesting. Since the position of the servo is controlled by the pulse length (generated by the **`pulsout`** command), try changing the first **`pulsout`** command to:

`pulsout 12,750`

What happened and why?

Recall that to rotate the servo to a particular position, just change the value of the pulse width. By changing our first width to 750, this yields a pulse width of 750 x 2 microseconds, or 1.5 milliseconds. The servo will rotate to about the middle of its rotation, and cycle back and forth between the middle and one extreme.

The servo has it's own internal potentiometer (more on this in future experiments) which compares the pulse width sent by the BASIC Stamp to it's center position, and responds by rotating in either direction.

Try different combinations of pulse widths (at both extremes) to rotate the servo to different positions.

Do you understand what we're doing here? Your program is able to move (or in this case rotate) a mechanical device, *in the real world.* If this were a bigger servo, you could use it to move the arm of an industrial robot, or open the door automatically at the supermarket! Servos like this one are also used to control the eyes and facial expressions of most "creatures" made by special effects experts for movies.

Questions

1. How is a servo different than a motor?

2. What command do we use (in PBASIC) to control a servo's rotation?

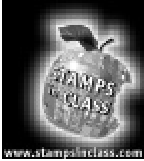
3. Why can't we use the **`Pause`** command to create the pulse lengths necessary to control a servo?

4. Describe the way a "**`for…bext`**" loop operates.

5. Add appropriate *remarks* to the following program:

```
x var word              ________________________________
output 1                ________________________________
here:                   ________________________________
for x = 1 to 100        ________________________________
pulsout 12,500          ________________________________
pause 10                ________________________________
next                    ________________________________
for x = 1 to 100        ________________________________
pulsout 12,1000         ________________________________
pause 10                ________________________________
next                    ________________________________
goto here               ________________________________
```

3



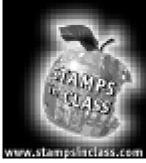
Challenge!

1. Write a program (complete with remarks) that will turn on the LED (on P5) every time the servo reaches one extreme of its travel, and then turn the LED off when it reaches the other extreme.

2. Write a program (with remarks) that rotate the servo from one extreme to the other (back and forth), but stopping for a short "pause" in the middle of its rotation each time.

3. Write a program (with remarks) that will move the servo to one extreme to the midpoint, return back, then rotate all the way to the other extreme, and then recycle.

4. Write a program that will cause the LED to blink 3 times and then rotate the servo from one extreme to the other. Pause for a moment and then repeat. This would be like a "warning" indicator that an automatic piece of machinery was about to start.



What have I learned?

On the lines below, insert the appropriate words from the list on the left.

- cycles
- decimal
- motor
- interface
- For…Next
- pulsout
- servos
- pulses
- milliseconds
- hardware

_______________ are a special type of DC _____________ which is well suited for connecting directly to a microcontroller. A servo is designed to react to a series of _____________ on its control wire. As the width of these pulses changes from 1 to 2 _______________, the servo's internal circuitry causes the motor to rotate to the appropriate position.

We use the _____________ command to output a specific pulse width for the control line input of the servo. In our application, we varied the pulse width between 1 and 2 milliseconds, using the command: `Pulsout 12, X`; where X is a _____________ value between 500 and 1000. Since the pulsout command has a resolution of 2 microseconds, this gave us a pulse width output of 1000 and 2000 microseconds, respectively.

Servos can be large or small, depending upon the application. The _____________circuitry (which is built into the servo housing) eliminates the need for us to connect many other _____________components for proper circuit operation.

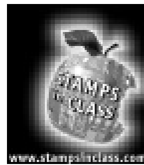
A ___________ loop is a convenient method to loop through a certain portion of our program for a pre-determined number of ____________. In our sample program, we looped 100 times, but this number could have been easily changed to accommodate other loop lengths, depending on the requirements of the program and hardware.

Why did I learn it?

To many of us, having a microcontroller blink on and off an LED might seem like no big deal, but making a motor or mechanical device move *under program control* is where microcontrollers really start to get interesting.

Although the microcontroller doesn't know what the output device is (LED or servo), making something move in the real world gives *us* a much more tangible example of *real world manipulation*.

There are microcontrollers (some of them BASIC Stamps!) all around us controlling servos, AC and DC motors, solenoids and other types of motive devices. These range from the little vibrator device inside your "silent" pager, to the automatic doors at the supermarket, to the robotic manipulators in use by hobbyists and professional developers alike.

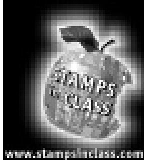


How can I apply this?

Although additional interface circuitry is usually required for most other types of motion devices (for connecting to the BASIC Stamp), the principles outlined in this experiment use essentially the same control techniques. Many people make their living designing microcontroller based systems that mechanically manipulate our world. Even if you don't end up doing this type of work as a career, you'll still have a appreciation for what goes into making your pager vibrate, or the supermarkets doors open for you automatically.

Now that we know how to control a servo, you could develop a control system for a model plane that would be similar to an "autopilot" function on a full sized aircraft. If you added a digital altimeter as an "input" to the BASIC Stamp, then the craft could be flown automatically.

In fact, you could design in some sort of "override" safety system that would allow a novice to fly the plane, but when he was about to crash into the ground (and destroy the plane!), your autopilot system could "take over" and prevent the catastrophe!



Experiment #4: Simple Automation

In Experiment #3 we used a servo (a specialized type of motor) to demonstrate how a microcontroller can manipulate a mechanical device in the "real world". The program that we wrote (and downloaded to the BASIC Stamp's memory) controlled the rotation and position of the servo.

The program caused the servo to swing back and forth between the servo's two different extremes of rotation. This was an example of how a microcontroller can cause a motive device to operate.

However in Experiment #3, the BASIC Stamp was "blind". All the servo did was respond to our code. Remember that the very heart of a microcontroller is its ability to make decisions based on inputs and then manipulate the "real world" with outputs.

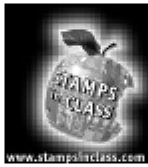
In this experiment we're going to do just that. Yes, we're going to move the servo again, but only if the proper input conditions are met. You could think of this experiment as the small equivalent of an automatic door at the supermarket. The door is closed most of the time until somebody – or something – comes near, then the door automatically opens. There is apparently nothing that we need to do in order for the door to open. We're not pushing any buttons, just by being near the door is enough to cause it to open. This is a very basic form of automation.

> **What's a...**
>
> **Automation:**
> In this experiment, the term automation means that something is being done, without any "human interaction". In our example (the supermarket automatic door), this isn't exactly true. Although we're not physically pushing any buttons, we are, by our presence, figuratively "pushing the light detectors button". This however, appears to be completely automatic, since we don't have to "think" about doing any thing other than walking up to the door.
>
> In its truest sense, automation is the microcontrollers ability to make things happen with no interaction on our part.

Some of the sensors that are used for this type of application are quite sophisticated, and others are quite simple. However, they all have one thing in common and that is that they sense an input and deliver the signal to a microcontroller so that it can make a decision, in this case "opening the door".

As our "detection sensor", we're going to use a device called a "photoresistor". It's a device designed to detect different light levels. It's a type of "optical" sensor, a part that is classified as an optoelectronic device.

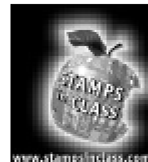
Let's automate! Experiment #4 requires the following parts:



Parts Required

For this experiment you will need the following:

(1) BASIC Stamp II
(1)"Board of Education"
(1) Three pin connector (if using Rev. A Board of Education)
(1) Programming Cable
(1) R/C servo
(1) LED
(1) 470 ohm, ¼ watt resistor
(1) 3300 microfarad electrolytic capacitor (if you are using a Rev. A Board of Education)
(1) Photoresistor, Cadmium Photocell (CdS)
(1) 10K ohm, ¼ watt resistor
(1) 9 volt battery or wall transformer
(misc.) connecting wires
(1) Personal Computer running DOS 2.0 or greater, with an available serial port.
(1) BASIC Stamp Editor program



Build It!

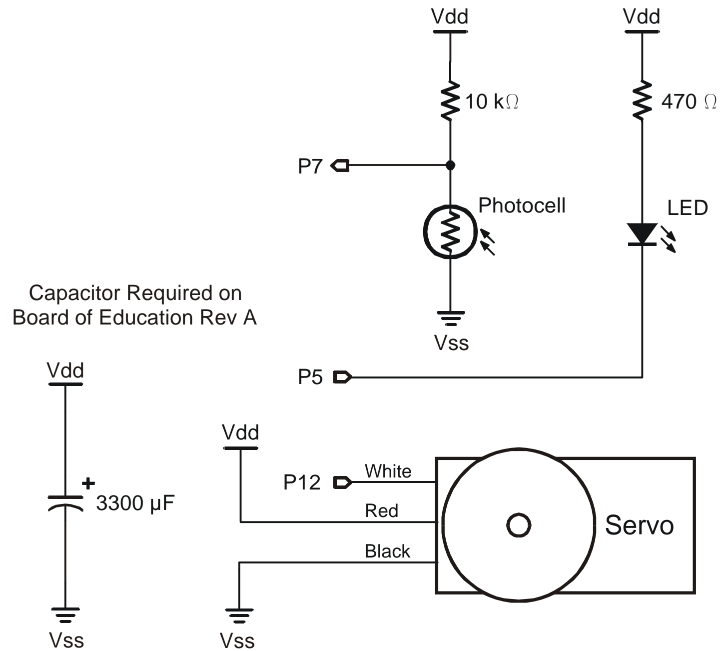
Using the Board of Education, create the hardware circuit as shown in the figures below.

Figure 4.1 is the schematic and Figure 4.2 is the pictorial (what the circuit physically looks like).

Depending on which model of servo you have, the color coding on the wires may vary. In all cases (with the servos you get from Parallax), the black wire is connected to Vss and the red wire is connected to Vdd. The remaining (third) wire may be white or yellow (or something else). This is the control input wire which we'll be conneting to the P1 signal on the BASIC Stamp.

Figure 4.1: Schematic for Experiment #4 on a Rev B Board of Education:

Note: Rev. A Board of Educations require the 3300 uF capacitor across the Vdd and Vdd. This schematic is for the Rev B. Board of Education.



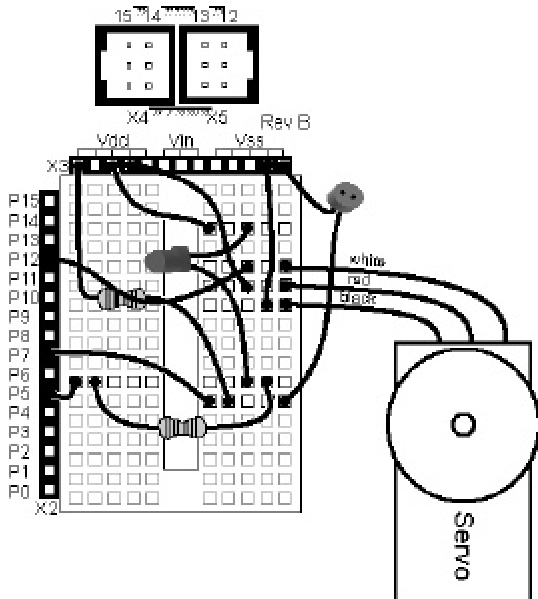
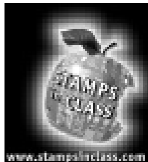


Figure 4.2: Physical picture for of circuit for Rev. B Board of Educations

This circuit has two types of output devices (the servo and the LED) and one type of input device (the photoresistor).

Remember to connect the LED properly!



Program It!

Once you have all the components installed into the Board of Education's prototype area (as shown in Figures 4.1 and 4.2) attach the programming cable from the Board of Education to your PC and connect either a 9 volt battery or a 9 volt DC wall transformer to the Board. Since the servo requires a lot of current (much more than an LED), battery life will be quite limited, so use the transformer if you have one.

What's a...

Program space:
Microcontrollers may have several types memory which they use to carry out their tasks. In the case of the BASIC Stamp 2, we're limited to 2048 bytes of (EEPROM) memory storage. This amount of space is used for both program *and data* storage. If you write a program that automatically gathers data over a period of time (such as a remote weather station), you'll want to make your program as small and as efficient as possible to allow as much room for data storage as you can.

EEPROM:
This stands for 'electrically erasable, programmable, read only memory. Although a sophisticated development in the "memory industry", it's really quite simple to use. We can store our programs & data in EEPROM with very simple commands. Then when the power is removed, both the program & data is retained. What sets the EEPROM apart from most other types of "solid state" memory is that it can be very easily erased ("automatically") & re-written to, again and again.

Turn on your PC, and double click on the BASIC Stamp icon.

You should now be running a program called the "Stamp Editor". This is a program that was created to help you write and download programs to the BASIC Stamp microcontroller.

Type in the following program:

```
this_place:
high 5
pause 200
low 5
pause 200
goto this_place
```

Now while holding the "ALT" key down, type the letter "r" (for "run") and press "enter".

Just another blinker program? Well yes and no. Notice that there is no "**output 5**" command in the program at all. If you've been reading through the appendices throughout these lessons you'll discover the the "**high**" and "**low**" commands automatically make the pin an output.

This saves some keystrokes, and more importantly saves program space on the BASIC Stamp. We really don't need to worry about running out of program space with our (small) experiment programs, but as you begin to create larger and more complex programs, it's a good habit to sharpen your programming skills to turn out "high quality" programs. Not only will you be less likely to run out of program space, but your code will actually run faster, resulting in a quicker execution time.

Modify the program to look like this:

```
n var bit
n=0

this_place:
n=0
low 5
debug ? n
pause 1000
high 5
n=1
debug ? n
pause 1000
goto this_place
```

Run the program.

Not only should your LED be blinking but you should also have an "information box" on your PC's monitor, alternately displaying "n=1" and "n=0".

Let's dissect, and see what's going on...

`n var bit`

A variable called "n", one bit in size.

`this_place:`

A label in the program

`n=0`

Something new. We're going to "set the value of 'n' to 0"

`low 5`

Make P5 low, thereby turning on the LED.

debug ? n

Debug? That word sounds familiar. Remember that to "debug" means to remove the errors in your program. Well, the PBASIC language has a command called "**debug**", which can really aid in getting rid of all those program glitches.

Normally we send the program from our PC down the programming cable to the BASIC Stamp. It's essentially been (until now) a one-way trip.

Debug is a very specialized command that allows the BASIC Stamp to send information ("data") back "up" the cable, and display it on your PC's monitor. In this manner we can "look inside the BASIC Stamp" and see the data that the BASIC Stamp is working with. In this case, we set the value of "n" to 0 in a previous command. When the debug command is encountered, it "prints" the value of "n" onto the PC's debug window.

> **What's a...**
>
> Debug (command):
>
> A very useful tool for "seeing" what your program is doing down inside the Stamp.
>
> The Debug command has a tremendous amount of flexibility built in. Check out the appendix for a complete description of each of the available features.

The "?" is an abbreviation for "print", so the command literally says: "Open the debug window on the PC and print the value of "n" on the screen".

pause 1000

Easy by now, right?

high 5

Turn off the LED.

> **What's a...**
>
> Hard-coded:
>
> A value or parameter that is set *absolutely* to a specific value. There are other ways to set values to a predetermined value (such as using the **con** command), that make it easier to change their value in the future. In many cases the variables used in a program are *dynamic* in that they constantly change their value during program execution.

n=1

Here we're changing the value of "n" to 1.

debug ? n

We're sending the value of "n" back to the PC. Since the value of "n" has changed, debug "prints" the change on your PC's screen.

pause 1000

Yep. We know what this does.

goto this_place

Go do it all over again.

Now some of you may be thinking "Why do we need to see the value of "n" in our debug window? Since our program (that we wrote) is setting the value of "n", we already know what the value of "n" is Why bother displaying it back to us?

That's true, the value of "n", in this example is "hard-coded" – the program sets the value with no variance. However, when a variable is set by external events, such as when an input changes due to a switch being pushed for example, then debug allows us to see the changing input data, and then ascertain whether or not our program is reacting properly.

Let's try it. Modify the program:

```
n var bit

this_place:
n=in7
debug ? n
pause 100
goto this_place
```

Now instead of "hard-coding" the value of "n", we're allowing "n" to be equal to the value of whatever "in7" is. Since P7 is connected to our photo-resistor, when the amount of light that the photo-resistor sees changes, the input voltage (on P7) also changes.

Move your hand over the photo-resistor (no need to touch, although you won't hurt it). The debug value on your PC should be changing from a "1" to a "0", depending on whether the photo-resistor sees a light or dark environment. Be sure the lights are on in your room.

This part of the circuit is called a resistive divider, and we'll explore this in greater detail in an upcoming experiment. Suffice it to say at this point, that as the photo-resistor changes its resistance (due to varying light levels), the voltage changes on P7. This voltage change is an analog signal. Now, since microcontroller input pins only recognize binary (digital) values, when the voltage reaches a certain point, the value that P7 sees will be either a "1" (+ 5 volts) or a "0" (0 volts).

What we've just done is created a "switch" that doesn't need to be pushed! It's a sensor that reacts automatically to external light levels.

Now let's do something fun…

Change the program to this:

```
x var word
n var bit
output 12
```

```
close_the_door:
for x = 1 to 100
  pulsout 12,500
  pause 10
next
pause 10

look_for_people:
n=in7
if n = 1 then open_the_door
pause 100
goto look_for_people

open_the_door:
for x = 1 to 100
  pulsout 12,1000
  pause 10
next
pause 10

n=in7
if n = 0 then close_the_door
goto open_the_door
```

What do you think this program is going to do?

Don't be discouraged that the program may be getting a little longer. Let's just break if down into manageable, byte sized chunks (pun!)

This is the initialization part of the program, we already know what these three commands do.

```
x var word
n var bit
output 12
```

We used this routine in Experiment #3. It sends out a series of pulses that makes the servo rotate to one end of its travel.

```
close_the_door:
for x = 1 to 100
  pulsout 12,500
  pause 10
next
pause 200
```

This part of the program simply looks at whether or not the photo-resistor detects a "shadow", & if it does, causes the program to go to the routine that will "open the door".

```
look_for_people:
n=in7
if n=1 then open_the_door
pause 100
goto look_for_people
```

If our servo was larger and connected to the supermarket's front door, this routine would open it.

```
open_the_door:
for x = 1 to 100
  pulsout 12,1000
  pause 10
next
pause 200
```

This is for "safety". As long as the person is standing anywhere near the door, keep the door open. (You don't want to crush too many customers, they'll start shopping somewhere else!)

```
n=in7
if n =0 then close_the_door
goto open_the_door
```

Questions

1. What is automation?

2. What does the "debug" command do, and why is it useful?

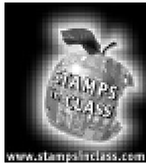
3. What does the command **`n = in7`** do? How does the command **`is_person_there = in7`** differ in execution?

4. In this experiment, how does the microcontroller know when to "open the door"?

5. Add appropriate remarks to the following program:

```
x var word               ____________________________________
n var bit                ____________________________________
output 1                 ____________________________________
close_the_door:          ____________________________________
for x = 1 to 100         ____________________________________
pulsout 12,500           ____________________________________
pause 10                 ____________________________________
next                     ____________________________________
pause 200                ____________________________________
look_for_people:         ____________________________________
n=in7                    ____________________________________
if n=1 then open_the_door ___________________________________
pause 100                ____________________________________
goto look_for_people     ____________________________________
open_the_door:           ____________________________________
for x = 1 to 100         ____________________________________
pulsout 12,1000          ____________________________________
pause 10                 ____________________________________
next                     ____________________________________
pause 200                ____________________________________
n=in7                    ____________________________________
```

```
if n = 0 then close_the_door      ___________________________________
goto open_the_door                ___________________________________
```



Challenge!

1. Write a program (complete with remarks) that will turn on the LED (on P5) every time the photo-resistor is in a shadow.

2. Write a program (with remarks) that will blink the LED twice and then "open the door" (rotate the servo), when the sensor detects a shadow. Then recycle and do it again.

3. Write a program (with remarks) that will blink the LED twice and then "open the door (rotate the servo), when the sensor detects a shadow. Then while the sensor is still detecting a shadow, blink the LED continuously, until the shadow goes away. Then recycle and do it again.

4. Write a program that will cause the LED to blink continuously, until a shadow is detected by the photo-resistor. Once the shadow is detected, the LED is turned on while the "door is being opened". Once the door is open, the LED is off, until the shadow goes away. Then recycle and do it again.

5. Think of conditions where the program for this experiment would not work correctly.

What have I learned?

On the lines below, insert the appropriate words from the list on the left.

automatic

photo-resistor

efficiently

automation

response

input

optical

execute

action

data

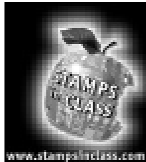
memory

_______________is a fascinating application for microcontrollers. Without any intentional _______________by humans, the BASIC Stamp can make things happen based solely on sensory inputs. The supermarket's _______________door is a great example of a typical microcontroller application.

There are many different types of sensors that can detect movement in the "real world". In this experiment we used a _______________whose value is dependent upon how much light it detects. Since this sensor detects light (or its absence), it is sometimes referred to as an "_______________" sensor.

The "Debug" command allows the BASIC Stamp to send _______________back to the PC, so that we can determine how well our program is operating, and whether or not it's making the appropriate "decisions". In this experiment the BASIC Stamp sent the value of "n" back to the PC, which was assigned to the value of the _______________ input on P7.

As we develop larger programs, it becomes increasingly important to write our "code" as _______________ as possible. This is important for several reasons. First, most microcontrollers (like the BASIC Stamp) have a limited amount of _______________ for program and data storage. The fewer instructions that we can use to accomplish a given task, means the more features that we can add to our program. Secondly, fewer instructions to accomplish a given task allows our program to _______________ faster – yielding a faster _______________time to "real world" situations. As an example, if our program was poorly written & the door didn't open quickly, the customer might walk right into the door. Not exactly great customer relations!



Why did I learn it?

Knowing how to "automate" certain tasks can take the drudgery (& sometimes danger) out of many types of jobs. Automation in the automobile industry has significantly improved many facets of the assembly process. Painting and welding, for example, are now done by automated robots with more consistency at lower cost and with less risk to employees.

On the down side, many people during this "automation era", are being displaced. Retraining is necessary, but hopefully the next job they get won't be nearly as hazardous or tedious.

On the plus side, there is a great opportunity for new products and processes that are accomplished automatically. This has spawned a whole new type of "innovation industry". No longer do you need to "put nut A on to bolt B", hour after hour, day after day. Now you can use your imagination to develop new and ever changing products that help improve life for everyone.

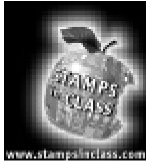
How can I apply this?

There are many opportunities to improve what we might consider to be rather mundane tasks.

For example, you could design a supermarket door that not only opened automatically, but also kept track of how many people actually entered (or for that matter, exited) the store.

You could keep track of this during specific time periods throughout the day, so that your microcontroller would alert the manager of the store that he's going to need additional check-out clerks, because there's more people shopping at this time.

This would improve customer service because the employees wouldn't be caught "off guard" with a bottleneck at the checkout stands. Your system would be a form of "shopping crowd early warning device"!



Experiment #5: Measuring an Input

As we've learned so far, each of the BASIC Stamp's 16 "real world" pins can be configured as either an input or an output. If the pin was configured as an input, there are 7 different instructions in the PBASIC language that can be used. Each of these commands is suited for specific types of input conditions.

For example, in Experiment #2 we learned how to use a command called "input". If this command is used in your program, it causes the specified pin to become an input (gee, that makes sense!). Then, anytime we wanted to check the status of the pin (whether it was "high" or "low") we used the statement **`if in2=0 then blink`**.

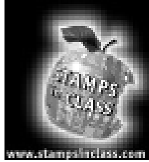
> **What's a...**
>
> 7 different "input" instructions:
>
> They are:
> - Button
> - Count
> - Input
> - Pulsin
> - Rctime
> - Serin
> - Shiftin
>
> We've already used "input" earlier. In this experiment, we'll be exploring "Pulsin"

This line of code "looked" at the pin, and returned a value. If the value on that pin was "0", then the code would branch off to another point in the program where it would blink the LED. If the value of the input was a "1" then it would continue program execution with the next line of code.

In any event, the values that could be detected with this code were binary - either a "1"or a "0". This is suitable for detecting whether or not a switch has been pushed (Exp. #2), or even to detect light or dark on a photocell, as we did in Experiment #4.

The PBASIC language has some other commands that offer a greater level of sophistication when it comes to detecting inputs. If you haven't already, download a free copy of the BASIC Stamp Manual from www.stampsinclass.com. In it you'll find a complete description and application information of all the commands available in the PBASIC language.

In this Experiment we're not only going to take an advanced look at "input detection", but we're also going to use a popular integrated circuit named the '555 timer.

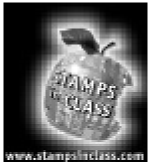


Parts Required

For this experiment you will need the following:

(1) BASIC Stamp II
(1)"Board of Education"
(1) Programming Cable
(1) LED
(1) CMOS 555 timer IC
(1) 10 microfarad, 25 volt electrolytic capacitor
(1) 1 microfarad, 25 volt electrolytic capacitor
(1) 470 ohm, ¼ watt resistor
(1) 100K potentiometer (variable resistor)
(1) 15K ohm, ¼ watt resistor
(1) 1 K ohm, ¼ watt resistor
(1) 9 volt battery or wall transformer
(misc.) connecting wires
(1) Personal Computer running DOS 2.0 or greater, with an available serial port.
(1) BASIC Stamp Editor program

Sources for these materials are listed in Appendix A.



Build It!

This hardware circuit uses an Integrated Circuit called a "555 timer". The '555' is a actually a "bunch of electronic circuitry" (that used to fill up a large area on a printed circuit board) that has been miniaturized and encased into the little "8 pin dip" package that we're using today. Although it's a sophisticated array of circuits on the inside of the plastic case, the '555' is really quite simple to use for many different applications. In fact, in the many years since it's development, the '555' has been designed into an untold number and variety of devices because it can do so many different things. Although it's not "programmable" like the BASIC Stamp, the '555' can be configured, with different combinations of resistors and capacitors, to accomplish many different tasks.

What's a...

Integrated Circuit:
"IC's" as they're commonly called, are electronic circuits that have been miniaturized and combined into one small, convenient package. Many different types of IC's have been created for untold numbers of applications. The '555' timer that we're using in this experiment is a member of the "Linear" family of IC's. The Stamp's CPU is a "Digital" IC.

8 pin dip:
This refers to the package style of the IC. The '555' has 8 Pins, and these pins are arranged in a Dual Inline Package.

What's a...

Astable multivibrator:
A fancy name for a circuit that with "no outside intervention" (by other circuitry or devices), will continually output a stream of pulses. Remember when we created the pulse stream for the servo control in Experiments 3 & 4? Same thing here, except that the '555' will alternate high and low, without us having to write a program. It's a hardware version of the software "Pulsout" command.

An IC (integrated circuit) needs to have some sort of identification on its package to tell us where pin #1 is. The identifier is usually a notch or indentation located on one end of the plastic package. See Figure 5.1.

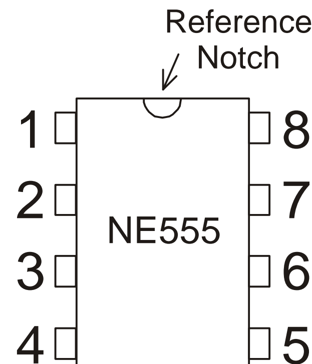


Figure 5.1: 555 Timer IC
Note the notch on one end of the chip package.

As with the BASIC Stamp, each pin on the '555' has a particular purpose. Although the '555' is fairly "bullet proof", connecting an improper electrical signal to the wrong pin can damage the device, so be careful and follow the diagrams closely.

The type of circuit that we're building here is called an "astable multivibrator". Don't be put off by its complicated name! All this really means is that the output of the '555' alternates from high to low. Recall that in Experiment #1 we used the 'high' and 'low' commands to blink an LED on and off. In reality, that's all that our '555' circuit is doing. It's just "oscillating" on and off. The '555' circuit that we're building, is the hardware equivalent of Experiment #1.

The rate at which the output (on pin #3 of the '555') blinks is controlled by the values of a resistor and capacitor. As the values of these devices change, the "blink" rate of the '555' changes.

We've used resistors in the past. They control the amount of current flowing through a given circuit. Since we want to (conveniently) change the rate at which the '555' blinks, we're going to use a variable resistor, also known as a potentiometer. If you've ever adjusted the volume on a radio, you've used one. By turning the dial on the "pot", you change the value of the variable resistor. This in turn changes the rate at which the '555' blinks.

What's a...

Potentiometer:
A "pot" is just a resistor that changes its value as you manually rotate (or in some cases slide) its shaft or dial. Recall that resistors have two "leads" or connections. A pot has three connections. The center lead is connected to a wiper that "wipes" across a resistive element. The closer the wiper gets to either end of the element, the lower the resistance between the wiper and the end it's approaching. Pots come in many different values, such as 5K, 10k, 100k, 1 meg ohms, and more. They also come in many different physical configurations to accommodate different product designs. But they all operate essentially the same way – mechanical movement of the wiper element changes the resistive value of the device.

As you're connecting the potentiometer, you'll notice that there are three terminals or connections available. One of these is the "wiper" and the other two are the ends of the resistive element. We only need to connect one end and the wiper contact to our circuit, as shown in Figure 5.2.

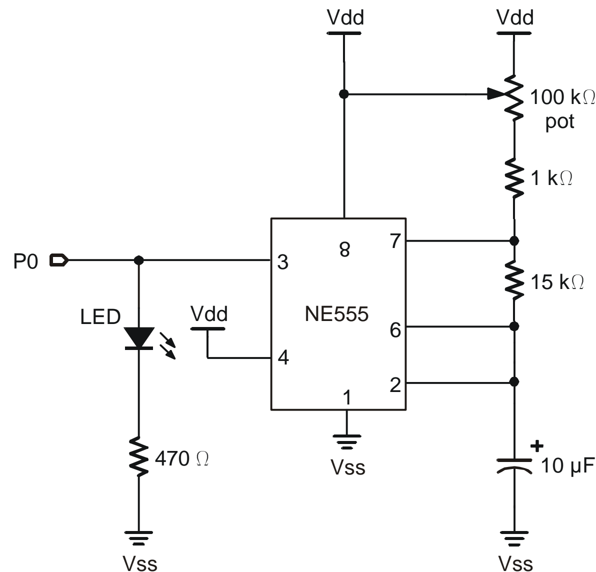


Figure 5.2: 555 Timer Schematic
Schematic for Experiment #5 on the Board of Education.

FYI:

About the schematics:
It is common practice on schematic diagrams to draw the pins of an IC wherever they make the diagram easiest to read.

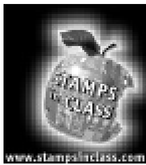
When you insert the '555' timer IC into the breadboard area of the Board of Education, be sure to have the device "span" the "dividing trench", so that the pins are not shorted together. Once you've completed the circuit shown in Figure 5.2, go ahead and power up the Board of Education.

When you power up the Board of Education you should find your LED blinking. Turn the knob on the potentiometer. What happened? You're probably wondering why it's already working and you haven't even written a program! That's the beauty of using a 555 timer in your circuit – it doesn't necessarily need to communicate with the BASIC Stamp to operate on it's own.

You see – Pin #4 (on the '555') is a "reset" pin. It is an "input" to the '555' and as long as this pin sees a 'high' (1), the '555' will operate. In order for our circuit to work without interaction from the BASIC Stamp, we connected pin #4 (on the '555') directly to Vdd (high). This kept pin #4 in a high state, which allowed the '555' to blink the LED. Now, we are going to have the BASIC Stamp take control of the '555'.

Remove the power source from the Board of Education. Move the end of the wire from Vdd (Pin 4 on the 555) to P1 (on the BASIC Stamp) to give the BASIC Stamp control. Now we are going to write a program that controls the '555' from P1. Remember, when Pin 4 (on the '555') sees high, it will start working.

5



Program It!

Start the BASIC Stamp Editor. If you don't remember how to do this refer back to an earlier experiment. As mentioned above, we now want the BASIC Stamp to control the 555 circuit, so be sure that you've connected the 555's Pin 4 to BASIC Stamp P1.

Pin #4 (on the '555' is a "reset" pin. It is an "input" to the '555' and as long as this pin sees a 'high', the '555' will operate. In order for our circuit to work without interaction from the BASIC Stamp, we connected pin #4 (on the '555') directly to Vdd (high). This kept pin #4 in a high state, which allowed the '555' to blink the LED.

What's a...

Reset:
As mentioned, this is a control pin on the '555' timer IC. If we connect this pin to P0 on the Stamp, & P0 is configured as an "*input*', then the '555' circuitry may in fact operate (although perhaps unreliably). In this situation we have two inputs (P0 on the Stamp and 'reset' on the '555') connected together. A pin configured as an input on the Stamp will tend to "float" high. This is a "floating condition, however and is not guaranteed to be a true "high". When we make P1 an output, and cause it to go "high", it does so, and *drives* pin 4 (on the '555") high, rather that just "floating" it up.

Type in the following program:

```
here:
high 1
pause 5000
low 1
pause 5000
goto here
```

Now while holding the "ALT" key down, type "r" (for "run") and press "enter".

What's happening, and why? If everything is working properly, you should see the LED blink on, off, on, off, etc. (for a period of 5 seconds) and then be completely off for 5 seconds. Then the program recycles and does it again.

Since P1 (on the BASIC Stamp) is connected to the reset pin on the '555', every time P1 goes 'high' it allows the '555' to blink the LED on and off. And whenever P1 goes low (under our program control), it shuts off the '555' circuit.

Ok you say, but so what?

Well, think about it this way. Microcontrollers are only capable of doing one thing at any one time. If we want to blink an LED on and off as a "warning indicator", then while the BASIC Stamp is doing its "high - pause - low – pause - repeat" routine (to blink the LED), the BASIC Stamp is not able to do anything else.

Now, as shown in the following program, you can turn on the "LED blinker circuit" and continue on (in your program) and do something else "more important". Try it.
'

```
x var word
low 1
here:
high 1              'turn the blinker on
for x = 1 to 500
debug ? x           'count to 500 on the screen
next                'while the LED blinks
pause 3000
low 1               'turn the blinker off
pause 2000
goto here           'go back and do it again
```

What we've done here is "off-loaded" the task of actually blinking the LED (on, off, on, off, etc.) from the BASIC Stamp. The action of "blinking" is accomplished by the '555' timer circuit. All the BASIC Stamp needs to do is enable or disable the "blinker circuitry". The BASIC Stamp can then go on and do other more important tasks. In this example, the "more important task is to count up to 500 and display the numbers on the screen. In the real world, however, you might be looking for other "input conditions" to be met (on some other pin on the BASIC Stamp).

What's a...

Microfarad:

A unit of measurement for the amount of "charge" that can be stored in a capacitor. Similar to the "ohm" value for resistors, the microfarad (for capacitors) is available in a wide range of values. 1 microfarad is equal to 1/1000000 of a farad. We'll explore capacitors in an upcoming experiment, but for now, understand that the lower the value, the lower the charge that you can store on the capacitor, which results in faster oscillation of the 555 circuit.

In this circuit, so far, we've been using a 10 microfarad capacitor. Shut off the power and replace the 10 microfarad capacitor with a 1 microfarad cap. Be sure to observe the proper polarity on the capacitors. Go ahead & re-apply power.

By reducing the value of the capacitor (C1) we've increased the blink rate of the LED (in this case, 10 fold). Even though it may be difficult to see visually, the LED is still blinking, but at a much higher rate.

In Experiment #4 (where we controlled the rotation of a servo), we used a command called **`pulsout`**. Recall that **`pulsout`** generated a single pulse output of a length determined by one of the commands' parameters.

For example, to create a pulse length of 1 millisecond (on P1), the command was: **`Pulsout 1, 500`**. The value of 500 is the number of two

microsecond increments. Therefore 500 times 2 microseconds = 1000 to microseconds or one millisecond.

We're now going to use a new command called **`pulsin`**.

`Pulsin` is the input counterpart to **`pulsout`**. Rather than generating an output pulse of a predetermined length, **`pulsin`** looks at a particular input pin and measures the length of an incoming pulse and returns the length of that pulse in a variable.

Try the following program:

```
x var word

high 1

here:
pulsin 0,1,x
debug ? x
goto here
```

What's happening? Pin #3 of the '555' timer circuit is connected to the LED. The LED blinks at the rate determined by the value of the potentiometer (and the capacitor). We've also connected the output of the '555' to P0 on the BASIC Stamp.

`x var word`

This simply sets up a variable called "x", that is one word (or 16 bits) in size. This means that x can go as high as 65,536 (decimal).

`high 1`

This causes P1 to go high, which in turn (since its connected to the reset pin #4 on the '555'), allows it to oscillate.

`here:`

A label to jump back to . . .

`pulsin 0,1,x`

This single command tells the BASIC Stamp to:

Look at an incoming pulse on P1.

Wait for that pulse to go from low to high.

As soon as it does, start a "stopwatch", and continue to monitor the pin.

As soon as the pulse goes back to low, then stop the "stopwatch" and return a the value in a variable called "x". This value is in two microsecond increments.

`debug ? x`

This displays the value of "x" on the PC.

`goto here`

Do it again.

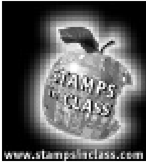
Now, try adjusting the value of the pot. What's happening to the value of "x"? Can you explain what's happening?

Whenever you change the value of the pot, the blink rate of the LED is changed. **`Pulsin`** can only measure up to a maximum of 131 milliseconds, that's why we increased the blink rate of the LED (by lowering the value of the capacitor). With the 10 microfarad capacitor, the blink rate was just too slow for **`pulsin`** to be able to measure it.

You can now actually measure the value of the blink rate of the LED. Take the value of "x" displayed on your screen, multiply it by two (remember that **`pulsin`** measures in 2 microsecond intervals) and you'll get the length (in microseconds) of each "blink".

The **`pulsin`** command is a significantly more advanced "input detector" command than a simple statement like "in" (or input), but they both have their appropriate uses – it just depends on the application.

Be sure and check out Appendix B and for a complete listing of PBASIC commands see the BASIC Stamp Manual Version 1.9. The Application Notes also show different ways to use the **`pulsin`** command.



Questions

1. What is a potentiometer, and what is one typically used for?

2. Why might we want to use a '555' timer to blink an LED instead of using the BASIC Stamp?

3. What does the **`pulsin`** command do?

4. What does the Reset pin on the '555' do, and how do you connect it to the BASIC Stamp?

5. Add appropriate remarks to the following program:

```
low 1              ______________________________
here:              ______________________________
high 1             ______________________________
for x = 1 to 500   ______________________________
debug ? x          ______________________________
next               ______________________________
pause 3000         ______________________________
low 0              ______________________________
pause 2000         ______________________________
goto here          ______________________________
```

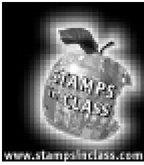
Challenge!

1. Write a program (complete with remarks) that will allow the LED to blink whenever a switch is connected to P8. (You'll need to build this circuit in hardware – if you need a hint, refer to Experiment #2 on connecting a switch to an input pin).

2. Draw the schematic diagram of your circuitry from Challenge #1.

3. Replace the switch part of the circuit in Challenge #1, with the photosensor circuit from Experiment #4. Connect the photosensor to P10 and write a program that will enable the '555' blinker circuit for a period of 3 seconds, whenever the sensor "sees a shadow".

4. Write a program that will display (using debug) the measured value of **`pulsin`** and whenever the value falls below 10000, the reset pin is brought low, shutting off the blinking circuit.



What have I learned?

On the lines below, insert the appropriate words from the list on the left.

- microfarads
- output
- volume level
- simultaneously
- values
- .131 seconds
- hardware
- debug
- variable
- integrated
- pulses
- Pulsout
- microseconds
- resistance
- LED

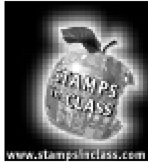
The '555' timer is an ______________ circuit that can be used for many different applications. In this Experiment, we used it to create a stream of ______________that caused an LED to blink. We then connected the ________________of the '555' to the BASIC Stamp and were able to measure the length of each pulse in ________________.

A potentiometer is a mechanical version of a _____________ resistor. To increase or decrease the ______________ of the pot, you physically rotate the shaft, not unlike changing the __________________on your home stereo.

The "blink rate" of the '555' timer circuit is determined by the _______ of a resistor (measured in ohms) and a capacitor (measured in ________________). A microfarad is equal to 1 / 1000000 of a farad.

The Pulsin command is the "input equivalent" to the ________________ command. Pulsin can measure pulses up to ______________ in length. In our program, using the _____________ command, we could actually measure the length of each pulse that was blinking the _____________.

Utilizing hardware to accomplish simple things is sometimes the best solution to accomplish a given task. If you have an application that needs to do two things ______________, you will need to weigh the benefits / disadvantages of adding additional ______________ circuitry to your design.



Why did I learn it?

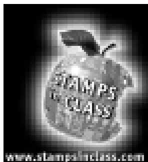
This Experiment demonstrates the interfacing of other types of integrated circuits to a microcontroller. Microcontrollers are only capable of doing one thing at any one time. In many cases, this restriction isn't a problem because the microcontroller operates at such a high rate of speed. If however, you absolutely need to be doing more than one thing at a time, the challenge can be easily solved by using additional circuitry as we did with the " 555" timer integrated circuit.

The "555" timer has been used in innumerable applications and products throughout the years. In this experiment, we used the timer in what we call "astable multivibrator" mode. This was a relatively simple example of how to off-load some of the processing that ordinarily would have to be accomplished by the microcontroller. Many products that use a microcontroller as their central processing unit (CPU), rely heavily on additional circuitry to accomplish certain tasks.

This is not to say that a microcontroller cannot do the job, but rather it is sometimes quicker and more cost-effective to use additional circuitry, to accomplish a given task. As you design your own circuits, you'll need to make decisions between additional code or additional hardware to come up with the most appropriate solution.

In some upcoming experiments, we will be connecting many other types of IC's to the BASIC Stamp which significantly enhance its operation and capabilities to interact with the "Real World".

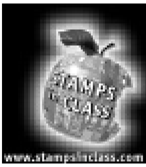
And of course, knowing how to interface different types of integrated circuits and components together is one of the foundational disciplines required of an electronics engineer.



How can I apply this?

As you continue to experiment with microcontrollers, you'll discover many different ways to interface or connect things. Some of these methods may be from some "application note" that was developed by a semiconductor company, still others might be your own creation. In any event, knowing the basic methods of connecting IC's together to form a reliable product is a very valuable skill.

Many of you have pagers or cell phones. These, as we've mentioned before have microcontrollers as their basic "brain". But in order for these devices to realize their true potential, they need "support circuitry" (not unlike our 555 timer). And the ability to design a suitable hardware solution will always be in demand.



Experiment #6: Manual to Digital

In our last experiment, we used one of the most popular integrated circuits of all time, the '555' timer. With it, we built an "astable multivibrator" – a fancy name for a circuit that blinked an LED.

Recall that the blink rate was controlled by the values of two components: a capacitor and a resistor. In order for us to conveniently change the blink rate, we substituted a potentiometer in place of the (normally) fixed value resistor. By manually rotating the shaft of the pot, we changed its resistance.

Some devices are now available that allow us to eliminate the "manual" element of changing a resistive value.

You may be familiar with cellular phones that require you to press a button rather than spin a dial to adjust the speaker volume. In many cases, this is done with a circuit similar to the one that we're going to create. Instead of changing the volume of a speaker, we're going to return to our '555' blinker circuit, and not only be able to turn the circuit on and off, but also vary the blink rate of the LED.

What's a...

Economic Decision

As you begin to create your own circuits, whether or not it's a commercial product, the cost of "electronic hardware" can rise rapidly – especially if you do this as an avid hobby outside of a normal classroom. It becomes increasingly important to decide which is the best approach to solve a particular task. Sometimes the best and cheapest aren't the same option. As we'll discover, many times you'll be able to solve a task both ways. It might take you longer to do it in software, but it (not counting your time) will almost always be cheaper.

Remember, all this is accomplished in hardware. It is important to realize that in every design, you'll make tradeoffs – either to do it in code or implement the function in hardware. There isn't just one correct answer. In many cases you could do both, and then it becomes an economic decision – which method would be least expensive? And, could the code control all of the functions reliably?

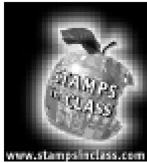
These are questions that are asked throughout the design process. As we'll discover in this Experiment, there are many different methods to accomplish a certain task, and sometimes it is better to let the microcontroller do the "really hard stuff", like calculations for example, and leave the mundane (blink the LED at a different rate) tasks to a simple hardware circuit.

Get out your Board of Education and make something happen!

Parts Required

For this experiment you will need the following:

(1) BASIC Stamp II module
(1)"Board of Education"
(1) Programming Cable
(1) LED
(1) 555 timer IC
(1) DS1804-100 digital resistor (IC)
(1) 10 microfarad, 25 volt electrolytic capacitor
(1) 1 microfarad, 25 volt electrolytic capacitor
(1) 470 ohm, ¼ watt resistor
(1) 100K potentiometer (variable resistor)
(1) 15K ohm, ¼ watt resistor
(1) 1 K ohm, ¼ watt resistor
(1) 9 volt battery or wall transformer
(misc.) connecting wires
(1) BASIC Stamp Editor program



Build It!

Look at the circuit as shown in Figure 6.1. As you'll notice, this is essentially the same the LED Blinker circuit that we created in Experiment #5. Recall that the LED "blink rate" was changed by turning the potentiometer.

What's a...

DS1804:
Many different number and letter combinations are used throughout the semiconductor industry to refer to individual components. These are simply reference numbers for specific types of devices.

This particular IC is manufactured by a company named "Dallas Semiconductor".

Now build the circuit shown in Figure 6.1. Connect the DS1804 integrated circuit in the potentiometer's place. Also notice that the reset line on the 555 (pin #4) is now connected to P0 on the BASIC Stamp.

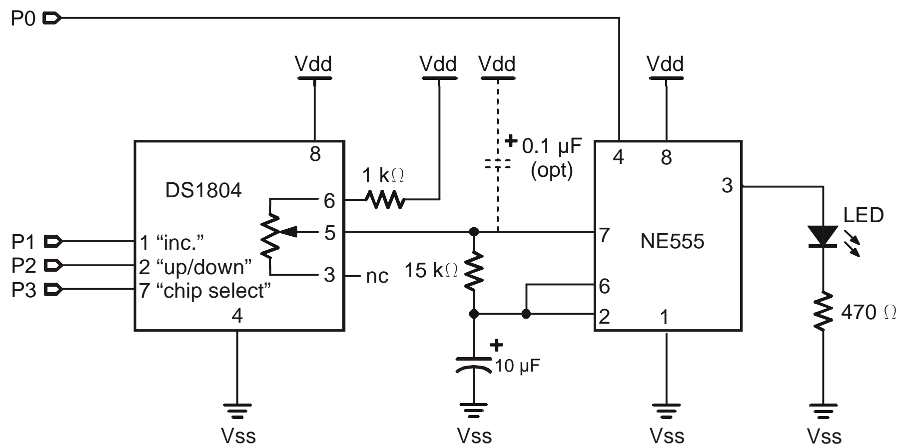
Ok, what is the DS1804? Remember that a potentiometer is nothing more that a variable resistor that changes its resistance when you rotate, or mechanically move its wiper or contact arm. This change in resistance is responsible for the change in the blink rate of the 555 timer circuit.

The DS1804 is a "digitally controlled" potentiometer. You can change its resistance (just like you did with the manual pot), but instead of mechanically moving the wiper, you're able to send digital pulses to it from the BASIC Stamp. These digital pulses change the location of the wiper and therefore, change the blink rate of the 555.

Figure 6.1:Digital Potentiometer. Replace the manual potentiometer with the Dallas Semiconductor "digital resistor" in Experiment 6, revising the circuit as shown.

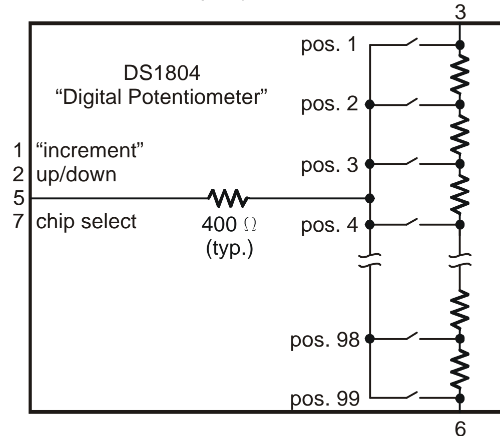
The dotted lines indicate an "optional" 0.1 uF capacitor which may be needed for '555' manufacturers other than SGS / Thompson or TI.

6



Take a look at Figure 6.2. This is an internal diagram of what's located on the inside of the DS1804 integrated circuit. The resistive element has a total value of 100k ohms, and it's separated into 99 segments. The wiper, though it's not totally "continuous", can be positioned to any one of 100 discrete positions, including above and below all the segments. The wiper itself is has about 400 ohms of resistance.

Figure 6.2:Inside the DS1804. This "digitally controlled" potentiometer takes the place of the manual version.



At position 1, only the wiper is contributing to the "output resistance", which you can measure as the resistance between pin 3 and pin 5. When the wiper is in position 2, the value of 1 segment on the resistive element is added to the wiper's resistance. So how much resistance is in a segment? It can be calculated by dividing the number of segments into the total value of all the segments on the internal resistive element:

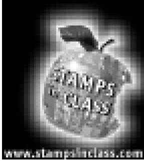
Segment resistance = total resistance ÷ 99 segments = 100 k-ohm ÷ 99 segments
= 1010.1010... ohms ÷ segment ≈ 1010 ohms/segment.

Position 2 will be the wiper resistance plus the value of one segment. That's 400 ohms + 1010 ohms = 1410 ohms. Position 3 will be 400 ohms + (2 × 1010) ohms = 2420 ohms. Since 400 ohms is 0.4 k-ohms, position 100 will be 0.4 k-ohms + 100 k-ohms ≈ 100.4 k-ohms.

FYI:

Sources of Error:

There are resistances in the wires, leads and connections inside the IC, as well as actual resistances in the silicon itself. We can usually disregard these types of "errors" because, in the case of the DS1804, the overall resistance is 100k ohms. In the DS1804, the wiper arm itself introduces approximately 400 ohms into the circuit. The value 400 ohms is the "typical value". According to the DS1804's data sheet, the wiper resistance could go as high as a "maximum value" of 1000 ohms..



In Experiment #5 we created a circuit that could enable (or disable) a 555 timer / blinker circuit. However, the only way that we could alter the blink rate was to manually rotate the mechanical pot. With the DS1804 we can now have the BASIC Stamp not only enable or disable the circuit, but it can control the blink rate as well.

Make sure that your programming cable is connected to the Board of Education and then go ahead and apply power.

Trouble?

If you get a message that says, "Hardware not found", re-check the cable connections between the PC and Carrier Board, & also make sure that the 9 volt battery (or wall transformer) is connected & charged.

Try downloading again (hold down the ALT key, & then press "r"). If it still doesn't work, you may have a bug! Re-check your program to be certain you've typed the program correctly.

After checking your connections, press ALT "r" again. If you still receive the "hardware not found" message, then make sure your computer is running in DOS, not Win95. If it is running in Win 95, then press the Start button (on the monitor), and select "Restart in MS-DOS mode".

If after trying this, you're still having problems, ask you instructor for help.

Type the following program into the BASIC Stamp Editor:

```
x var word
y var word
output 0
output 1
output 2
output 3
high 0

low 3

low 2
for x = 1 to 99
  high 1
  low 1
next

high 2
for y = 1 to 2
  high 1
  low 1
next

here:
goto here
```

Ok, what's the program doing? If it's operating correctly, your LED should be blinking (fairly quickly). If not, check your code and make sure you typed it in correctly.

Now try this: Change the value of "y" from 2 to 75.

Your code (with remarks), should now look like this:

```
x var word
y var word

output 0            '555 reset
output 1            'increment
output 2            'UP / down
output 3            'chip select

high 0              'enable the blinker
low 3               'select the DS1804

'this section resets the variable resistor to zero ohms

low 2               'the pulses cause the DS1804 to go "down"
for x = 1 to 99
  high 1            'pulse it
  low 1
next                'cycle 99 times

'this routine sets the value of the variable resistor to a value
'determined by 'y' in the For…Next Loop.

high 2              'the pulses cause the DS1804 to go "up"
for y = 1 to 75
  high 1
  low 1
next                'cycle 75 times (set the resistor up 75 positions)

here:               'after setting the resistor, stay here and do nothing
goto here
```

Run the program.

What happened? Why?

Let's take a closer look at our code:

```
x var word
y var word
```

Nothing new here, all we're doing is setting up a couple of variables called "x"" and "y".

```
output 0            '555 reset
```

This output is connected to the timer's reset line. Therefore by changing the value of P0's output, we can turn the blinker circuit on or off.

```
output 1              'increment
```

This command we've done many times before, but what is P1 connected to? The answer can be found in the data sheet for the DS1804 variable resistor. The increment signal (pin #1 on the DS1804) is where we want to send in our pulses to move the wiper. As you'll see, we'll use the 'high' and 'low' commands to pulse this signal line.

```
output 2       'UP / down
```

Another output from the BASIC Stamp that is connected to a signal called "UP / down" on the DS1804. When we program the BASIC Stamp to make P2 high then any pulses on the "increment pin" will cause the variable resistor to increase in value. Conversely if P2 is low, then any pulses on the increment signal line will cause the DS1804 to decrease in value.

6

```
output 3       'chip select
```

The chip select signal is simply a way of allowing the DS1804 to have its value changed. If the chip select signal (on pin 7 of the DS1804) is high, then any pulses or "UP / down" signals are simply ignored. The signal (coming from P3 on the BASIC Stamp) is "active low".

```
high 0         'enable the blinker
```

OK, this sets P0 to high, therefore enabling the blinker circuit.

```
low 3          'select the DS1804
```

Since P3 is "chip select" (and it's active low), making P3 a "0" enables the DS1804 to receive pulses and modify its resistance setting.

```
low 2          'the pulses cause the DS1804 to go "down"
```

Whenever the DS1804 is first turned on, we don't have any idea where the wiper is set because we can't visually see it. Therefore, this routine (beginning with this command) will send out enough pulses to set the resistive wiper element all the way back to the beginning. Since there are 99 discrete positions on the resistive element, by pulsing it "down" (setting the direction with the command "low 2") at least 99 times, ensures that it's at the beginning. It doesn't really matter if you pulse it "down" (or "up" for that matter) more than 99, the DS1804 will just "bottom out" and disregard the extra (unnecessary pulses).

```
for x = 1 to 99
```

This sends the wiper all the way to the "bottom" of the resistive element.

```
high 1          'pulse it
low 1
next            'cycle 99 times
```

These commands make P0 go high and low, resulting in a single pulse, and because we're in a For...Next loop that cycles 99 times, this creates 99 pulses. The "digital resistor" is now set to "0" ohms.

```
high 2          'the pulses cause the DS1804 to go "up"
```

The P2 signal that was set "low" above (causing the wiper to move "down") is now set "high" so that any pulses (on pin 1 of the DS1804) from this point on in the program, will cause the wiper to move "up".

```
for y = 1 to 75
```

This loop length can be changed from anywhere between 1 and 99. Try different values. Each time you change the value (that the For...Next loop cycles), run the program. You'll see that the blinker operates at a different rate, depending upon your value.

```
high 1
low 1
next            'cycle 75 times (set the resistor up 75 positions)
```

In this case we're cycling 75 times, which causes the wiper to move up "75 positions".

```
here:           'after setting the resistor, stay here and do nothing
goto here
```

At this point our program just sits there and loops, doing nothing. In a "real application" however, your program would continue doing other tasks, meanwhile the blink rate (that was set by your microcontroller) continues, with no other BASIC Stamp interaction required.

OK, lets modify the program as follows to make the blink rate increase and decrease visibly:

```
x var word
y var word

output 0        '555 reset

output 1        'increment
```

```
output 2              'UP / down
output 3              'chip select

high 0                'turn on the blinker

here:
low 2                 'point the wiper "down"
low 3                 'select (or enable) the DS1804 to receive data

for x = 1 to 99       'reset the value of the wiper to "0" slowly
  high 1
  low 1
pause 200
next

high 2                'point the wiper "up"
for x = 1 to 99       'make it go up 99 discrete positions
  high 1
  low 1
pause 200             'pause for a short time so that we can see each step.
Next
goto here             'do it again
```

The first part of the program is the same, with the exception of the placement of the "here" label, and the following modifications:

```
high 2                'point the wiper "up"
```

This makes the wiper go in the up direction, whenever the chip is selected, and it receives pulses on the "increment" pin.

```
for y = 1 to 99       'make it go up 99 discrete positions
```

At first thought, it appears as though we're going to set the position of the wiper all the way "up". Eventually we do, but not right away.

```
high 1
low 1
```

Here's where the pulses are created.

```
debug ? y
```

This allows us to see where the wiper is positioned

```
pause 200    'pause for a short time so that we can see each step.
```

What's a...

Ohmmeter:
An ohmmeter is a device that actually measures the value of a particular resistor. It works by forcing a current through the resistor and measuring the voltage drop across it.

In most cases, you need to have the power off (to your circuit) if you're going to use an ohmmeter. In the case of the DS1804, however, you can leave power applied as long as you only touch the three terminals (representing the potentiometer terminals) with the probes. In fact, in order to be able to measure the wiper value at any point, power must be applied. This is a rare instance – and is *not* the norm.

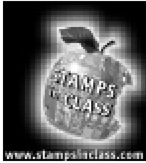
Now, by placing "pause" here, we're able to see just how the wiper is moved in the upward direction. You'll notice that the LED is blinking slower and slower as the wiper is moved (internally) on the variable resistor chip.

```
next
goto here    'do it again
```

Instead of recycling in a "do nothing" loop, we now are going back to do it all over again.

Experiment. Change some of the loop values. What happens when you don't "reset" the wiper all the way back to "low"? Try pulsing the wiper beyond its 99 position "limit".

If you have access to an ohmmeter, you can actually measure the resistance changes in the DS1804 between the wiper and either end of the "pot" terminals.



Questions

1. What does "chip select" mean?

2. How is the DS1804 different from the potentiometer that we used in Experiment #5?

3. Why does the **for…next** command come in useful in this Experiment?

4. Why is it important to know how to hook up hardware, rather than just writing programs for a microcontroller?

5. Add appropriate remarks to the following program:

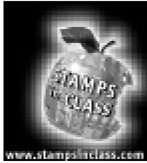
```
x var word              ______________________________
y var word              ______________________________
output 0                ______________________________
output 1                ______________________________
output 2                ______________________________
output 3                ______________________________
high 0                  ______________________________
low 3                   ______________________________


low 2                   ______________________________
for x = 1 to 99         ______________________________
high 1                  ______________________________
low 1                   ______________________________
next                    ______________________________


high 2                  ______________________________
for y = 1 to 20         ______________________________

high 1                  ______________________________
```

```
low 1       ____________________________________________
next        ____________________________________________
here:       ____________________________________________
goto        _______________________________________
here        _______________________________________
```



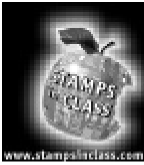
Challenge!

1. Write a program (complete with remarks) that alternates the rate of the LED blinker circuit from a slow speed to a fast speed, every 5 seconds.

2. Write a program that changes the rate of the blinker circuit every 1 second. The rate is to go from a fast blink to a slower blink. Once the circuit has cycled one complete time (after approximately 31 seconds!) have the program stop the blinker circuit and go to a "do nothing" loop. Display the wiper position as it is changing, on your PC using debug.

3. Replace the 10 microfarad capacitor in the 555 timer circuit with a 1 microfarad cap. Connect the output of the 555 to P5 on the BASIC Stamp. Now draw the complete schematic of your circuit.

4. Using the circuit from Challenge #3, modify the program from Challenge #2 to measure the length of the pulses in (2) microsecond increments using the **`pulsin`** command on the BASIC Stamp's P5 pin. Have the program recycle and do it again.



What have I learned?

On the lines below, insert the appropriate words from the list on the left.

- design
- program space
- manual
- function
- pulses
- hardware
- interface
- off-loading
- displaying
- BASIC Stamp
- rate
- microcontroller

The ________________ is capable of doing many different things. It all depends on what type of ____________ is connected to it. In this Experiment we built a 555 timer circuit and allowed the microcontroller to send out a series of _________ that not only enabled the blinking circuit, but also controlled the actual ________ of the pulses coming out of the 555.

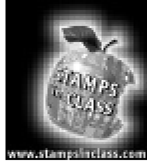
The control of the blink rate was accomplished by replacing the "__________" pot that we used in our last Experiment with the DS1804 "digital potentiometer". There are many devices, such as the DS1804, that allow microcontrollers to _____________ and control things in the "real world".

By setting the blink rate at a certain point, the BASIC Stamp was free to "go about other business", such as calculating or ____________ data on our PC.

Many times it is necessary to free up valuable (and sometimes expensive) ___________________ for more important tasks. We saw this initially in Experiment #4, and we now see that the potential of "________________" some of the processing from the microcontroller may in fact, be unlimited.

It's possible, for example that a complete control system could be built with no microcontroller at all. It would be made entirely of "discrete" logic. In fact, until the emergence of the ______________, this was how circuits were built.

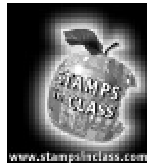
The microcontroller therefore gives us the option of choosing whether a _____________ should be solved in hardware or software. It's all part of the __________ process.



Why did I learn it?

If you happen to choose this field as a career, (whether for the first time, or later in life) there are many attributes that could give you an advantage over others. As we've learned in the last two Experiments, there is almost always more than one way to solve a problem. Knowing when to do it in "hardware" or when to write the "code", is a very desirable talent. There are an unlimited number of opportunities for innovative design engineers and especially those that know how to best "blend" hardware and software together resulting in the "best" product.

Even if you're not planning on this field as a career, the ability to be flexible in your approach to problem solving will help you stand out in whatever discipline you choose.



How can I apply this?

Why not design a BASIC Stamp controlled stereo system for your home that detects when someone walks into the room? The BASIC Stamp could detect your presence - similar to how we used the photo-resistor in an earlier Experiment. Now, since the BASIC Stamp is "always" looking to make sure that somebody is in the room, it really can't spend a whole lot of time sending out continuous control signals to the volume control on the stereo.

Therefore, utilizing a device such as the DS1804, the BASIC Stamp can monitor other inputs & based on that data, set the appropriate sound level. Then, as you enter the room (detected by the BASIC Stamp), the program causes the stereo volume to gradually increase to a pleasant level. If the phone rings, (using another type of sensor) the BASIC Stamp would automatically decrease the stereo's volume so that you could talk on the phone with no objectionable background "noise".

Parts Listing

All components (next page) used in the first six experiments are readily available from common electronic suppliers including Mouser and Digi-Key. Customers who would like to purchase a complete kit may also do so through Parallax. Parallax adds a small packaging and handling fee to the parts, partially offset by our volume purchases made to the suppliers. Customers may realize a small savings of 10% on high volumes (75+ units) of the "What's a Microcontroller?" Parts Kit by building their own component kits.

The first six experiments require the Board of Education Full Kit (#28102):

Parallax also manufactures the Board of Education Kit (#28150), consisting of the board and pluggable wires only. Use this kit if you already have a BS2-IC module and power supply. Individual pieces may also be ordered using the Parallax stock codes shown below.

Board of Education – Full Kit (#28102)

Parallax Code#	Description	Quantity
28150	Board of Education	1
800-00016	Pluggable wires	10
BS2-IC	BASIC Stamp II module	1
750-00008	300 mA 9 VDC power supply	1
800-00003	Serial cable	1

Board of Education Kit (#28150)

Parallax Code#	Description	Quantity
28102	Board of Education and pluggable wires	1
BS2-IC	Pluggable wires	10

This printed documentation is very useful for additional background information:

BASIC Stamp Documentation

Parallax Code#	Description	Internet Availability?
27919	BASIC Stamp Manual Version 1.9	http://www.stampsinclass.com
28123	"What's a Microcontroller?" Text	http://www.stampsinclass.com
27951	"Programming and Customizing the BASIC Stamp Computer"	Table of Contents only from http://www.stampsinclass.com

The first six lessons require the "What's a Microcontroller?" Parts Kit (#28123)

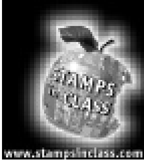
The contents of the "What's a Microcontroller?" Parts Kit is listed below. These parts are required for experiments one through six. In case you need specific replacement parts from Parallax the stock code is listed for each individual component. If you would rather purchase these components elsewhere and need assistance identifying an appropriate source for these parts, please feel free to contact us at stampsinclass@parallaxinc.com.

What's a Microcontroller? Parts Kit (#28123)

Parallax Code#	Description	Quantity
150-04710	470 ohm ¼ watt 5% resistor	6
350-00006	LED, red color	6
150-01030	10K ¼ watt 5% resistor	2
400-00002	Tact switch (4-lead pushbutton type)	2
800-00016	Pluggable jumper wires, bag of 10	1
900-00002	DC hobby servo (Hitec HS 300 or equivalent)	1
201-03080	3300 uF electrolytic capacitor**	1
451-00301	3 pin single row header	1
150-01020	1K ohm ¼ watt 5% resistor	5
350-00009	Photo-resistor (EG&G Vactec)	1
604-00006	CMOS 555 timer - 8 pin dip (use SGS/Thompson or TI)*	1
201-01050	1uF electrolytic capacitor**	1
201-01062	10 uF electrolytic capacitor**	1
150-01530	15k resistor 1/4 watt	1
152-01040	100k potentiometer	1
152-01041	100k solid state potentiometer (Dallas Semiconductor DS1804-100)	1

* If not using specified manufacturer part, you may need to connect a 0.1 uF capacitor between Pin 7 and Vdd in Experiment #6, Manual to Digital.

** Capacitor voltage ratings should be equal or greater than 16V.



Sources

The Parallax distributor network serves approximately 40 countries world-wide. A portion of these distributors are also Parallax-authorized "Stamps in Class" distributors – qualified educational suppliers. Stamps in Class distributors normally stock the Board of Education (#28102 and #28150) and sometimes the "What's a Microcontroller?" Parts Kit (#28123). Several electronic component companies are also listed for customers who wish to assemble their own "What's a Microcontroller?" Parts Kit.

Country	Company	Notes
United States	Parallax, Inc. 3805 Atherton Road, Suite 102 Rocklin, CA 95765 USA (916) 624-8333, fax (916) 624-8003 http://www.stampsinclass.com http://www.parallaxinc.com	Parallax and Stamps in Class source. Manufacturer of the BASIC Stamp.
United States	Jameco 1355 Shoreway Road Belmont, CA 94002 (650) 592-8097, fax (650) 592-2503 http://www.jameco.com	Parallax Stamps in Class distributor. Also a reliable source for components.
United States	Digi-Key Corporation 701 Brooks Avenue South Thief River Falls, MN 66701 (800) 344-4539, fax (218) 681-3380 http://www.digi-key.com	Source for electronic components. Parallax distributor. May stock Board of Education. Excellent source for components.
United States	Mouser Electronics 345 South Main Mansfield, TX 76203 (800) 346-6873, fax (817) 483-6899 http://www.mouser.com	Source for electronic components. Parallax distributor. May stock Board of Education in 1999. Excellent source for components.

Australia	Microzed Computers PO Box 634 Armidale 2350 Australia Phone +612-67-722-777, fax +61-67-728-987 http://www.microzed.com.au	Parallax distributor. Stamps in Class distributor. Excellent technical support.
Australia	RTN 35 Woolart Street Strathmore 3041 Australia phone / fax +613 9338-3306 http://people.enternet.com.au/~nollet	Parallax distributor and Stamps in Class distributor.
Canada	Aerosystems 3538 Ashby St-Laurent, QUE H4R 2C1 Canada (514) 336-9426, fax (514) 336-4383	Parallax distributor and Stamps in Class distributor.
Canada	HVW Technologies 300-8120 Beddington Blvd NW, #473 Calgary, AB T3K 2A8 Canada (403) 730-8603, fax (403) 730-8903 http://www.hvwtech.com	Parallax distributor and Stamps in Class distributor.
Germany	Elektronikladen W. Mellies Str. 88 32758 Detmold Germany 49-5232-8171, fax 49-5232-86197 http://www.elektronikladen.de	Parallax distributor and Stamps in Class distributor.
New Zealand	Trade Tech Auckland Head Office, P.O. Box 31-041 Milford, Auckland 9 New Zealand +64-9-4782323, fax 64-9-4784811 http://www.tradetech.com	Parallax distributor and Stamps in Class distributor.

Netherlands	Antratek Kanaalweg 33 2903 LR Capelle A/S Ijssel Netherlands +31-10450-4949, fax 31-10451-4955 antratek@box.nl	Parallax distributor and Stamps in Class distributor.
United Kingdom	Milford Instruments Milford House 120 High St., S. Milford Leeds YKS LS25 5AQ United Kingdom +44-1-977-683-665 fax +44-1-977-681-465 http://www.milinst.demon.co.uk	Parallax distributor and Stamps in Class distributor.

Boooks and Internet Resources

If you are new to BASIC Stamps, electronics, or programming there are several internet and printed sources you may wish to investigate.

Books and Publications

Programming & Customizing the Basic Stamp Computer by Scott Edwards. ISBN 0-07-913684-2. Available from Parallax (#27905) and Amazon (http://www.amazon.com).

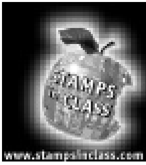
Parallax BASIC Stamp Manual Version 1.9 from Parallax (#27919) and distributors.

Nuts and Volts Magazine Stamp Applications. Published each month in Nuts and Volts magazine (http://www.nutsvolts.com), with past issues available for free download from their web site.

Internet

Parallax web site http://www.parallaxinc.com and the Parallax Stamps in Class web site http://www.stampsinclass.com include free downloadable BASIC Stamp resources.

Al Williams Consulting hosts the BASIC Stamp Project of the Month at http://www.al-williams.com.



Reading the Resistor Color Code

Most common types of resistors have colored bands that indicate their value. The resistors that we're using in this series of experiments are typically "1/4 watt, carbon film, with a 5% tolerance". If you look closely at the sequence of bands you'll notice that one of the bands (on an end) is gold. This is band #4, & the gold color designates that it has a 5% tolerance.

Bands 1 through 3 tell us what the actual value is, measured in "ohms". The higher the value, the less current is permitted to flow through it (at a given voltage).

The color values are as follows:

Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Grey	8
White	9

To determine the value of a resistor, look at the first color, determine its value from the above chart & write it down. Do the same for the second band. The third band "is the number of 0's to write down". For example:

A resistor has the following color bands:

Band #1. = Red
Band #2. = Violet
Band #3. = Yellow
Band #4. = Gold

Looking at our chart above, we see that Red has a value of 2.

So we write: "2".
Violet has a value of 7.
So we write: "27"

Yellow has a value of 4.
So we write: "27 and four zeros" or "270000".

This resistor has a value of 270,000 ohms (or 270k) & a tolerance of 5%.

DS1804 Datasheet

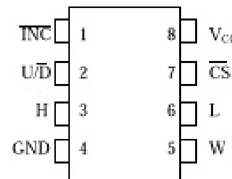
Appendix D consists of the Dallas Semiconductor 1804 datasheet. You can also download the datasheet from http://www.dalsemic.com.

DS1804
NV Trimmer Potentiometer

www.dalsemi.com

FEATURES
- Single 100-position taper potentiometer
- Nonvolatile "on-demand" wiper storage
- Operates from 3V or 5V supplies
- Up/down, increment-controlled interface
- Available in 8-pin (300-mil) DIP, 8-pin (150-mil) SOIC, 8-pin (118-mil) µSOP, and Flip Chip packages
- Operating Temperature:
  - Industrial: -40°C to +85°C
- Standard Resistance Values:
  - DS1804-010 10 kΩ
  - DS1804-050 50 kΩ
  - DS1804-100 100 kΩ

PIN ASSIGNMENT



8-Pin DIP (300-mil)
8-Pin SOIC (150-mil)
8-Pin uSOP (118-mil)
8-Pin Flip Chip Package

See Mech. Drawings Section

PIN DESCIPTION
H	-	High-Terminal of Potentiometer
L	-	Low-Terminal of Potentiometer
W	-	Wiper of Potentiometer
V_{CC}	-	3V or 5V Power Supply
$\overline{CS}$	-	Chip Select
$U/\overline{D}$	-	Up/Down Control
$\overline{INC}$	-	Increment/Decrement Wiper Control
GND	-	Ground

DESCRIPTION
The DS1804 NV Trimmer Potentiometer is a nonvolatile digital potentiometer having 100 positions. The device provides an ideal method for low-cost trimming applications using a CPU or manual control input with minimal external circuitry. Wiper position of the DS1804 can be stored in EEPROM memory on demand. The device's wiper position is manipulated by a 3-terminal port that provides an increment/ decrement counter controlled interface. This port consists of the control inputs $\overline{CS}$, $\overline{INC}$, and $U/\overline{D}$. The DS1804 is available in three resistor grades, which include a 10 kΩ, 50 kΩ, and 100 kΩ. The device is provided in an industrial temperature grade. Additionally, the DS1804 will operate from 3V or 5V supplies and is ideal for portable application requirements. Three packaging options are available and include the 8-pin (300-mil) DIP, 8-pin (150-mil) SOIC, 8-pin (118-mil) uSOP, and the Flip Chip Package.

OPERATION

The DS1804 is a single nonvolatile potentiometer. The device has a total of 100 tap-points including the L- and H- terminals. A total of 99 resistive segments exist between the L- and H- terminals. These tap-points are accessible to the W-terminal, whose position is controlled via a 3-terminal control port. A block diagram of the DS1804 is shown in Figure 1.

The 3-terminal port of the DS1804 provides an increment/decrement interface which is activated via a chip select input. This interface consists of the input signals $\overline{CS}$, $\overline{INC}$, and U/$\overline{D}$. These input signals control a 7-bit up/down counter. The output of the 7-bit up/down counter controls a 1 of 100 decoder to select wiper position. Additionally, this interface provides for a wiper storage operation using the $\overline{CS}$ and $\overline{INC}$ input control pins. The timing diagram for the 3-terminal interface control is shown in Figure 2.

PIN DESCRIPTIONS

V_{CC} - Power Supply. The DS1804 will support supply voltages ranging from +2.7 to +5.5 volts.

GND - Ground.

H - High-Terminal Potentiometer. This is the high terminal of the potentiometer. It is not required that this terminal be connected to a potential greater than the L-terminal. Voltage applied to the H-terminal can not exceed the power-supply voltage, V_{CC}, or go below ground.

L - Low-Terminal Potentiometer. This is the low terminal of the potentiometer. It is not required that this terminal be connected to a potential less than the H-terminal. Voltage applied to the L-terminal cannot exceed the power-supply voltage, V_{CC}, or go below ground.

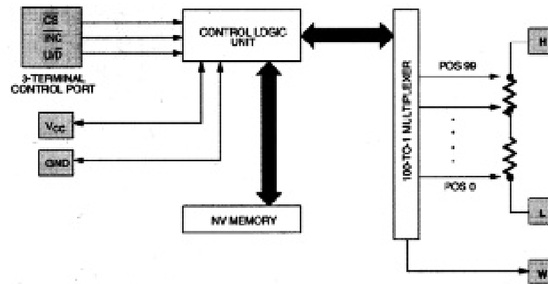
W - Wiper of the Potentiometer. This pin is the wiper of the potentiometer. Its position on the resistor array is controlled by the 3-terminal control port. Voltage applied to the wiper cannot exceed the power-supply voltage, V_{CC}, or go below ground.

$\overline{CS}$ - Chip Select. The $\overline{CS}$ input is used to activate the control port of the DS1804. This input is active low. When in a high-state, activity on the $\overline{INC}$ and U/$\overline{D}$ port pins will not affect or change wiper position.

$\overline{INC}$ - Wiper Movement Control. This input provides for wiper position changes when the $\overline{CS}$ pin is low. Wiper position changes of the W-terminal will occur one position per high-to-low transition of this input signal. Position changes will not occur if the $\overline{CS}$ pin is in a high state.

U/$\overline{D}$ - Up/Down Control. This input sets the direction of wiper movement. When in a high state and $\overline{CS}$ is low, any high-to-low transition on $\overline{INC}$ will cause a one position movement of the wiper towards the H-terminal. When in a low state and $\overline{CS}$ is low, any high-to-low transitions on $\overline{INC}$ will cause the position of the wiper to move towards the L-terminal.

DS1804 BLOCK DIAGRAM Figure 1



POWER-UP/POWER-DOWN CONDITIONS

On power-up the DS1804 will load the value of EEPROM memory into the wiper position register (or 1 of 100 decoder). The value of this register can then be set to another wiper position if desired, by using the 3-terminal control port. On power-up, wiper position will be loaded within a maximum time period of 500 µs once the power-supply is stable. Additionally, the 3-terminal interface port will be active after 50 ms.

On power-down, the wiper position register data will be lost. On the next device power-up, the value of EEPROM memory will be loaded into the wiper position register.

On shipment from the factory, Dallas Semiconductor does not guarantee a specified EEPROM memory value. This value should be set by the customer as needed.

NONVOLATILE WIPER STORAGE

Wiper position of the DS1804 can be stored using the $\overline{INC}$ and $\overline{CS}$ inputs. Storage of the wiper position takes place, whenever the $\overline{CS}$ input transitions from low-to-high while the $\overline{INC}$ is high. Once this condition has occurred the value of the current wiper position will be written to EEPROM memory.

The DS1804 is specified to accept 50,000 writes to EEPROM before a wear-out condition. After wear-out the DS1804 will still function and wiper position can be changed during powered conditions using the 3-terminal control port. However, on power-up the wiper-position will be indeterminate.

ONE TIME PROGRAMMABILITY (OTP)

The DS1804 can be easily used as an OTP device. The user of the DS1804 can trim the desired value of the wiper position and set this position for storage as described above. Any activity through the 3-terminal port can then be prevented by connecting the $\overline{CS}$ input pin to V_{CC}. Also, an OTP application does not adversely affect the number of times EEPROM is written, since EEPROM will only be loaded and ***not written*** during a power-up or power-down condition.

On power-up the DS1804 will load the current value of EEPROM memory into the wiper position register.

ABSOLUTE MAXIMUM RATINGS*

Voltage on Any Pin Relative to Ground -1.0V to +7.0V
Operating Temperature -40 °C to +85 °C
Storage Temperature -55 °C to +125 °C
Soldering Temperature 260 °C for 10 seconds

* This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

1. Insert A (Burn-in Disclaimer)

The Dallas Semiconductor DS1804 is built to the highest quality standards and manufactured for long term reliability. All Dallas Semiconductor devices are made using the same quality materials and manufacturing methods. However, the Flip Chip Package version of the DS1804 is not exposed to environmental stresses, such as burn-in, that some industrial applications require. For specific reliability information on this product, please contact the factory in Dallas at (972) 371-4448.

RECOMMENDED DC OPERATING CONDITIONS (-40°C to +85°C)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Supply Voltage	V_{CC}	+2.7		5.5	V	1
Input Logic 1	V_{IH}	2.0		V_{CC}+0.5	V	1,2
Input Logic 0	V_{IL}	-0.5		+0.8 +0.6	V	1,15
Resistor Inputs	L,H,W	GND-0.5		V_{CC}+0.5	V	1,3

DC ELECTRICAL CHARACTERISTICS (-40°C to +85°C; V_{CC}=2.7V to 5.5V)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Supply Current	I_{CC}			400	µA	4
Input Leakage	I_{LI}	-1		+1	µA	
Wiper Resistance	R_W		400	1000	Ω	
Wiper Current	I_W			1	mA	
Standby Current: 3 Volts 5 Volts	I_{STBY}		10 20	40	µA µA	5
Wiper Load Time	t_{WLT}		500		µs	6
Power-Up Time	t_{PU}		50		ms	14

ANALOG RESISTOR CHARACTERISTICS (-40°C to +85°C; V_{CC}=2.7V to 5.5V)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
End-to-End Resistor Tolerance		-20		+20	%	8
Absolute Linearity			±0.6		LSB	9
Relative Linearity			±0.25		LSB	10
-3 dB Cutoff Frequency	f_{cutoff}				MHz	11
Temperature Coefficient			750		ppm/°C	

CAPACITANCE (25°C; V_{CC}=2.7V to 5.5V)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Input Capacitance	C_{IN}			5	pF	12
Output Capacitance	C_{OUT}			7	pF	12

AC ELECTRICAL CHARACTERISTICS (-40°C to +85°C; V_{CC}=2.7V to 5.5V)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
$\overline{CS}$ to $\overline{INC}$ Setup	t_{CI}	50			ns	13
U/$\overline{D}$ to $\overline{INC}$ Setup	t_{DI}	100			ns	13
$\overline{INC}$ Low Period	t_{IL}	50			ns	13
$\overline{INC}$ High Period	t_{IH}	100			ns	13
$\overline{INC}$ inactive to $\overline{CS}$ Inactive	t_{IC}	500			ns	13
$\overline{CS}$ Deselect Time	t_{CPH}	100			ns	13
Wiper Change to $\overline{INC}$ Low	t_{IW}			200	ns	13
$\overline{INC}$ Rise and Fall Times	t_R, t_F			500	µs	13
$\overline{INC}$ Low to $\overline{CS}$ Inactive	t_{IK}	50			ns	16
Wiper Storage Time	t_{WST}			10	ms	13,17

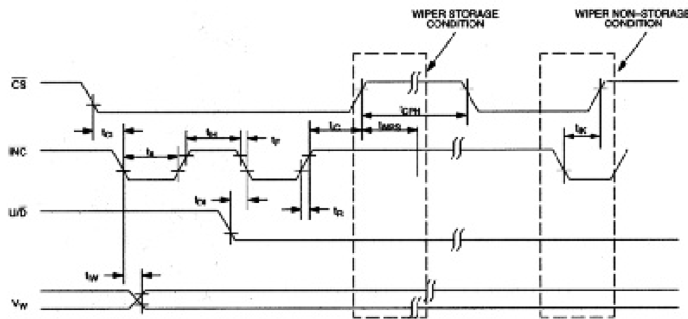
NOTES:

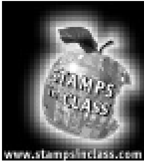
1. All voltages are referenced to ground.
2. Valid for V_{CC} = 5V only.
3. Resistor input voltages cannot go below ground or exceed V_{CC} by the amounts as shown in the table.
4. Maximum current specifications are based on the clock rate of $\overline{INC}$ input. This specification represents the current required when changing the wiper position.
5. Standby current levels apply when all inputs are driven to appropriate supply levels. $\overline{CS}$, $\overline{INC}$, U/$\overline{D}$ = V_{CC}.
6. Wiper load time is specified as the time required for the DS1804 to load the wiper position with the contents of nonvolatile memory once V_{CC} has reached a stable operating voltage equal to or greater than 2.7V.
7. The DS1804 is available in three resistor values. These include the DS1804-010; 10 kΩ, the DS1804-050 50 kΩ; and the DS1804-100 100 kΩ.
8. Valid at 25 °C only.
9. Absolute linearity is used to compare measured wiper voltage versus expected wiper voltage as determined by wiper position. The DS1804 is specified to provide an absolute linearity of ±0.60 LSB.
10. Relative linearity is used to determined the change in voltage between successive tap positions. The DS1804 is specified to provide a relative linearity specification of ±0.25 LSB.
11. -3 dB cutoff frequency characteristics for the DS1804 depend on potentiometer total resistance. DS1804-010, 1 MHz, DS1804-050; 200 kHz, and DS1804-100; 100 kHz.
12. Capacitance values apply at 25 °C.
13. See Figure 2.
14. Power-up time is specified as the time required before the 3-terminal control becomes active after a stable power supply level has been reached.
15. At V_{CC} = 2.7V, V_{IL} = 0.8V.
16. The $\overline{INC}$ low to $\overline{CS}$ inactive is specified to be 50 ns minimum. This is the transition condition which allows the DS1804 3-terminal port to become inactive without writing the EEPROM memory of the part.
17. Wiper Storage Time, t_{WST}, is the time require for the DS1804 to write EEPROM memory for storage of a new wiper position. The maximum time required to accomplish this task is specified at 10 ms.

DS1804 ORDERING INFORMATION

ORDERING NUMBER	PACKAGE	OPERATING TEMPERATURE	VERSION
DS1804-010	8L DIP (300-MIL)	-40 ℃ TO +85 ℃	10kΩ
DS1804-050	8L DIP (300-MIL)	-40 ℃ TO +85 ℃	50kΩ
DS1804-100	8L DIP (300-MIL)	-40 ℃ TO +85 ℃	100kΩ
DS1804Z-010	8L SOIC (150-MIL)	-40 ℃ TO +85 ℃	10kΩ
DS1804Z-050	8L SOIC (150-MIL)	-40 ℃ TO +85 ℃	50kΩ
DS1804Z-100	8L SOIC (150 MIL)	-40 ℃ TO +85 ℃	100kΩ
DS1804u-010	8L µSOP (118-MIL)	-40 ℃ TO +85 ℃	10kΩ
DS1804u-050	8L µSOP (118-MIL)	-40 ℃ TO +85 ℃	50kΩ
DS1804u-100	8L µSOP (118-MIL)	-40 ℃ TO +85 ℃	100kΩ
DS1804X-010	8L FCP (118-MIL)	-40 ℃ TO +85 ℃	10kΩ
DS1804X-050	8L FCP (118-MIL)	-40 ℃ TO +85 ℃	50kΩ
DS1804X-100	8L FCP (118-MIL)	-40 ℃ TO +85 ℃	100kΩ

3-TERMINAL INTERFACE TIMING DIAGRAM Figure 2





PBASIC Quick Reference Guide

The Parallax BASIC Stamp Manual Version 1.9 consists of approximately 450 pages of PBASIC command descriptions, application notes, and schematics. The entire document is available for download from http://www.parallaxinc.com and http://www.stampsinclass.com in Adobe's PDF format, but may also be purchased by students and educational institutions.

This PBASIC Quick Reference Guide is a reduced version of BASIC Stamp II commands.

BRANCHING

IF…THEN
IF *condition* THEN *addressLabel*
Evaluate condition and, if true, go to the point in the program marked by *addressLabel*.

- *Condition* is a statement, such as "x=7" that can be evaluated as true of false.
- *AddressLabel* is a label that specifies where to go in the event that the condition is true.

BRANCH
BRANCH *offset*, [*address0, address1, …address N*]
Go to the address specified by offset (if in range).

- *Offset* is a variable / constant that specifies which of the listed address to go to (0-N).
- *Addresses* are labels that specify where to go.

GOTO
GOTO addressLabel
Go to the point in the program specified by addressLabel.

- *AddressLabel* is a label that specifies where to go.

GOSUB
GOSUB *addressLabel*
Store the address of the next instruction after GOSUB, then go to the point in the program specified by *addressLabel*.

- *AddressLabel* is a label that specifies where to go.

RETURN
Return from subroutine – sends the program back to the address (instruction) immediately following the most recent GOSUB.

LOOPING

FOR...NEXT

FOR *variable* = *start* to *end* {*stepVal*} ...NEXT

Create a repeating loop that executes the program lines between FOR and NEXT, incrementing or decrementing variable according to stepVal until the value of the variable pases the end value.

- *Variable* is a bit, nib, byte, or word variable used as a counter.
- *Start* is a variable or constant that specifies the initial value of the variable.
- *End* is a variable or constant that specifies the end value of the variable. When the value of the variable passes end, the FOR . . . NEXT loop stops executing and the program goes on to the instruction after NEXT.
- *StepVal* is an optional variable or constant by which the variable increases or decreases with eachteip through the FOR / NEXT loop. If start is larger than end, PBASIC2 understands stepVal to be negative, even though no minus sign is used.

NUMERICS

LOOKUP

LOOKUP *index*, [*value0, value1,... valueN*], *variable*

Look up the value specified by the index and store it in a variable. If the index exceeds the highest index value of the items in the list, variable is unaffected. A maximum of 256 values can be included in the list.

- *index* is a constant, expression or a bit, nibble, byte or word variable.
- *value0, value1*, etc. are constants, expressions or bit, nibble, byte or word variables.
- *variable* is a bit, nibble, byte or word variable.

LOOKDOWN

LOOKDOWN *value*, {??,} [*value0, value1,... valueN*], *variable*

- *value* is a constant, expression or a bit, nibble, byte or word variable.
- *??* is =, <>, >, <, <=, =>. (= is the default).
- *value0, value1*, etc. are constants, expressions or bit, nibble, byte or word variables.
- *variable* is a bit, nibble, byte or word variable.

RANDOM
RANDOM *variable*
Generate a pseudo-random number.

- *variable* is a byte or word variable in the range 0..65535.

DIGITAL I/O

INPUT
INPUT *pin*
Make the specified pin an input (write a 0 to the corresponding bit of DIRS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

OUTPUT
OUTPUT *pin*
Make the specified pin an output (write a 1 to the corresponding bit of DIRS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

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

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

LOW
LOW *pin*
Make pin output low (write 1 to the corresponding bit of DIRS and 0 to the corresponding bit of OUTS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

HIGH
HIGH *pin*
Make the specified pin output high (write 1s to the corresponding bits of both DIRS and OUTS).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

TOGGLE
TOGGLE *pin*
Invert the state of a pin.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

PULSIN
PULSIN *pin, state, variable*
Measure an input pulse (resolution of 2 µs).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *state* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *variable* is a bit, nibble, byte or word variable.

Measurements are in 2uS intervals and the instruction will time out in 0.13107 seconds.

PULSOUT
PULSOUT *pin, period*
Output a timed pulse by inverting a pin for some time (resolution of 2 µs).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *period* is a constant, expression or a bit, nibble, byte or word
- *variable* in the range 0..65535 representing the pulse width in 2uS units.

BUTTON
BUTTON *pin, downstate, delay, rate, workspace, targetstate, label*
Debounce button, perform auto-repeat, and branch to address if button is in target state.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *downstate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *delay* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *rate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *workspace* is a byte or word variable.
- *targetstate* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *label* is a valid label to jump to in the event of a button press.

SHIFTIN

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

Shift bits in from parallel-to-serial shift register.

- *dpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the data pin.
- *cpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the clock pin.
- *mode* is a constant, symbol, expression or a bit, nibble, byte or word variable in the range 0..4 specifying the bit order and clock mode. 0 or MSBPRE = msb first, pre-clock, 1 or LSBPRE = lsb first, pre-clock, 2 or MSBPOST = msb first, post-clock, 3 or LSBPOST = lsb first, post-clock.
- *result* is a bit, nibble, byte or word variable where the received data is stored.
- *bits* is a constant, expression or a bit, nibble, byte or word variable in the range 1..16 specifying the number of bits to receive in result. The default is 8.

SHIFTOUT

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

Shift bits out to serial-to-parallel shift register.

- *dpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the data pin.
- *cpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the clock pin.
- *mode* is a constant, symbol, expression or a bit, nibble, byte or word variable in the range 0..1 specifying the bit order. 0 or LSBFIRST = lsb first, 1 or MSBFIRST = msb first.
- *data* is a constant, expression or a bit, nibble, byte or word variable containing the data to send out.
- *bits* is a constant, expression or a bit, nibble, byte or word variable in the range 1..16 specifying the number of bits of data to send. The default is 8.

COUNT

COUNT *pin, period, result*

Count cycles on a pin for a given amount of time (0 - 125 kHz, assuming a 50/50 duty cycle).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *period* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.
- *result* is a bit, nibble, byte or word variable.

XOUT

XOUT *mpin, zpin*, [*house**keyorcommand*{*cycles*} {, *house**keyorcommand*{*cycles*}… }]

Generate X-10 powerline control codes. For use with TW523 or TW513 powerline interface module.

- *mpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the modulation pin.
- *zpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the zero-crossing pin.
- *house* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying the house code A..P respectively.
- *keycommand* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15 specifying keys 1..16 respectively or is one of the commands shown for X-10 light control in the BASIC Stamp Manual Version 1.9. These commands include lights on, off, dim and bright.
- *cycles* is a constant, expression or a bit, nibble, byte or word variable in the range 2..65535 specifying the number of cycles to send. (Default is 2).

SERIAL I/O

SERIN

SERIN *rpin*{*fpin*}, *baudmode*, {*plabel*,} {*timeout*, *tlabel*,} [*inputdata*]

Serial input with optional qualifiers, time-out, and flow control. If qualifiers are given, then the instruction will wait until they are received before filling variables or continuing to the next instruction. If a time-out value is given, then the instruction will abort after receiving nothing for a given amount of time. Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data received must be N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).

- *rpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..16.
- *fpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *baudmode* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.
- *plabel* is a label to jump to in case of a parity error.
- *timeout* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 representing the number of milliseconds to wait for an incoming message.
- *tlabel* is a label to jump to in case of a timeout.
- *inputdata* is a set of constants, expressions and variable names separated by commas and optionally proceeded by the formatters available in the DEBUG command, except the ASC and REP formatters. Additionally, the following formatters are available:

1. STR bytearray\L{\E} input a string into bytearray of length L with optional end-character of E. (0's will fill remaining bytes).
2. SKIP L input and ignore L bytes.
3. WAITSTR bytearray{\L} Wait for bytearray string (ofL length, or terminated by 0 if parameter is not specified and is 6 bytes maximum).
4. WAIT (value {,value...}) Wait for up to a six-byte se-quence.

SEROUT

SEROUT *tpin*{*fpin*}, *baudmode*, {*pace*,} {*timeout*, *tlabel*,} [*outputdata*]

Send data serially with optional byte pacing and flow control. If a pace value is given, then the instruction will insert a specified delay between each byte sent (pacing is not available with flow control). Baud rates of 300 - 50,000 are possible (0 - 19,200 with flow control). Data is sent as N81 (no parity, 8 data bits, 1 stop bit) or E71 (even parity, 7 data bits, 1 stop bit).

- *tpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..16.
- *fpin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *baudmode* is a constant, expression or a bit, nibble, byte or word variable in the range 0..60657.
- *pace* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 specifying a time (in milliseconds) to delay between transmitted bytes. This value can only be specified if the fpin is not specified.
- *timeout* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 representing the number of milliseconds to wait for the signal to transmit the message. This value can only be specified if the fpin is specified.
- *tlabel* is a label to jump to in case of a timeout. This can only be specified if the fpin is specified.
- *outputdata* is a set of constants, expressions and variable names separated by commas and optionally proceeded by the formatters available in the DEBUG command.

ANALOG I/O

PWM

PWM pin, duty, cycles

Output PWM, then return pin to input. This can be used to output analog voltages (0-5V) using a capacitor and resistor.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *duty* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255.
- *cycles* is a constant, expression or a bit, nibble, byte or word variable in the range 0..255 representing the number of 1ms cycles to output.

RCTIME

RCTIME *pin, state, variable*

Measure an RC charge/discharge time. Can be used to measure potentiometers.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *state* is a constant, expression or a bit, nibble, byte or word variable in the range 0..1.
- *variable* is a bit, nibble, byte or word variable.

SOUND

FREQOUT

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

Generate one or two sinewaves of specified frequencies (each from 0 - 32767 hz.).

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range of 0..15.
- *milliseconds* is a constant, expression or a bit, nibble, byte or word variable.
- *freq1* and *freq2* are constant, expression or bit, nibble, byte or word variables in the range 0..32767 representing the correspond-ing frequencies.

DTMFOUT

DTMFOUT *pin*, {*ontime*, *offtime*,}[*key*{,*key*...}]

Generate DTMF telephone tones.

- *pin* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.
- *ontime* and *offtime* are constants, expressions or bit, nibble, byte or word variables in the range 0..65535.
- *key* is a constant, expression or a bit, nibble, byte or word variable in the range 0..15.

EEPROM ACCESS

DATA

DATA {*pointer*} DATA {@*location*,} {*WORD*} {*data*}{(*size*)} {, { *WORD*} {*data*}{(*size*)}...}

Store data in EEPROM before downloading PBASIC program.

- *pointer* is an optional undefined constant name or a bit, nibble, byte or word variable which is assigned the value of the first memory location in which data is written.
- *location* is an optional constant, expression or a bit, nibble, byte or word variable which designates the first memory location in which data is to be written.
- *word* is an optional switch which causes DATA to be stored as two separate bytes in memory.
- *data* is an optional constant or expression to be written to memory.
- *size* is an optional constant or expression which designates the number of bytes of defined or undefined data to write/reserve in memory. If DATA is not specified then undefined data space is reserved and if DATA is specified then SIZE bytes of data equal to DATA are written to memory.

READ
READ *location, variable*
Read EEPROM byte into variable.

- *location* is a constant, expression or a bit, nibble, byte or word variable in the range 0..2047.
- *variable* is a bit, nibble, byte or word variable.

WRITE
WRITE *location, data*
Write byte into EEPROM.

- *location* is a constant, expression or a bit, nibble, byte or word variable in the range 0..2047.
- *data* is a constant, expression or a bit, nibble, byte or word variable.

TIME
PAUSE
PAUSE *milliseconds*
Pause execution for 0–65535 milliseconds.

- *milliseconds* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535.

POWER CONTROL

NAP
NAP *period*
Nap for a short period. Power consumption is reduced to 50 uA (assuming no loads).

- *period* is a constant, expression or a bit, nibble, byte or word variable in the range 0..7 representing 18ms intervals.

SLEEP
SLEEP *seconds*
Sleep for 1-65535 seconds. Power consumption is reduced to approximately 50 µA.

- *seconds* is a constant, expression or a bit, nibble, byte or word variable in the range 0..65535 specifying the number of seconds to sleep.

END
END
- Sleep until the power cycles or the PC connects. Power consumption is reduced to approximately 50 µA.

PROGRAM DEBUGGING

DEBUG
DEBUG *outputdata*{,*outputdata*...}
Send variables to PC for viewing.
- *outputdata* is a text string, constant or a bit, nibble, byte or word variable. If no formatters are specified DEBUG defaults to ascii character display without spaces or carriage returns following the value.