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.

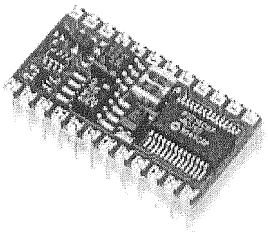
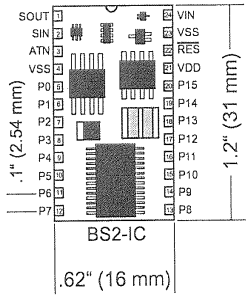


Figure 1.1: BASIC Stamp II
This is a small picture of the BASIC Stamp II module. The actual module is about the size of a postage stamp.



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.

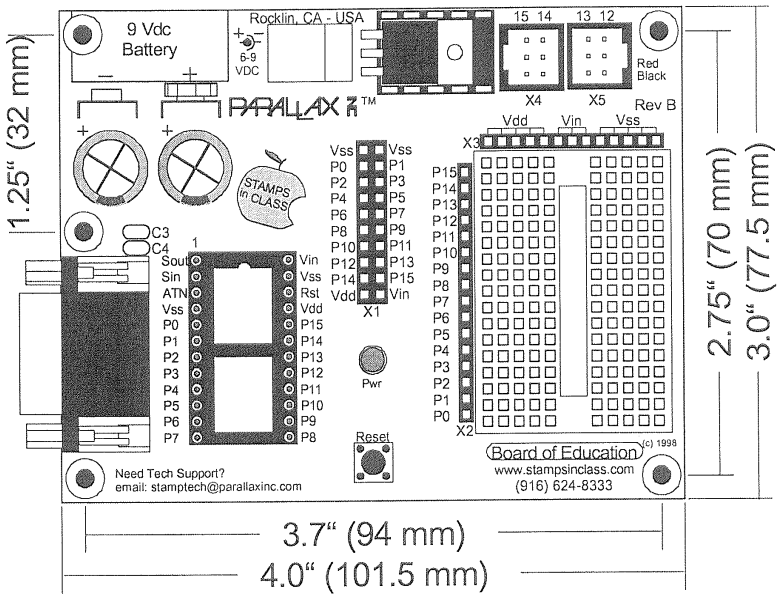


Figure 1.2:
Board of Education Rev. B
This is where we will build our circuit. The socket is for the BASIC Stamp module, and the breadboard is for your projects. The BASIC Stamp is oriented with the large chip closest towards the "AppMod Connector"

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.

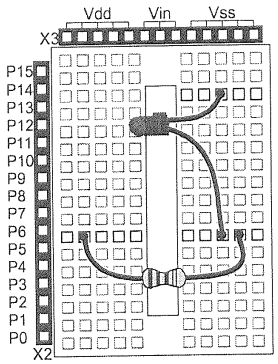


Figure 1.3: LED on Breadboard
Shows LED and resistor "plugged" into breadboard. No connections have been made yet to the BASIC Stamp's I/O pins.

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.

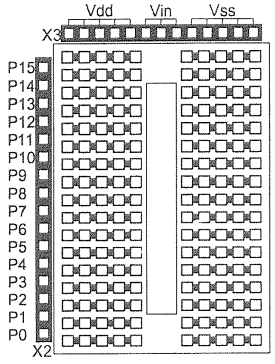
Connecting an LED:
Never connect an LED to the Stamp, without having a resistor (of the proper value) in the circuit . The resistor limits the amount of current flow in the circuit to a safe level, thereby protecting both the LED and the Stamp.

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.

What's a...

Schematic:
An electrical diagram showing connections between components, but not necessarily looking like the physical circuit. We use schematic diagrams, because they help in understanding signal flow through complex circuits.

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.

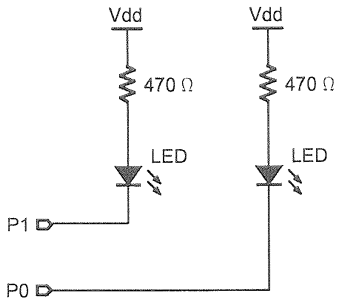


Figure 1.5: Schematic
Electrical diagram for circuit shown on the right side.

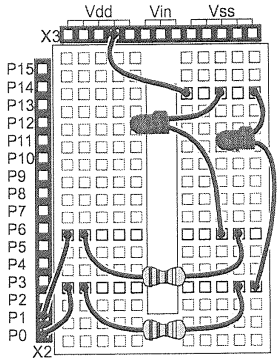


Figure 1.6: Pictorial
What the circuit physically looks like after you build it. The flat side of the LED is closest to the resistor.

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!

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

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.

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:

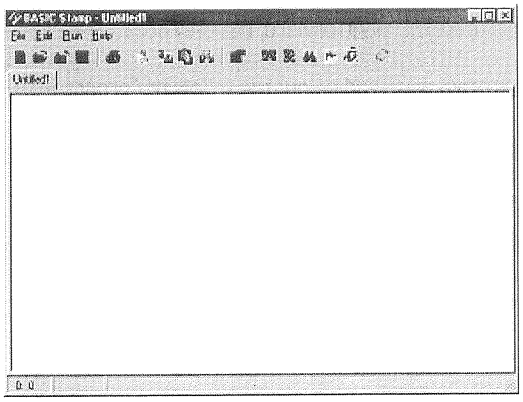


Figure 1.7: BASIC Stamp Software
Double-click on the BASIC Stamp icon to run the software. The opening screen will look like this.

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
```

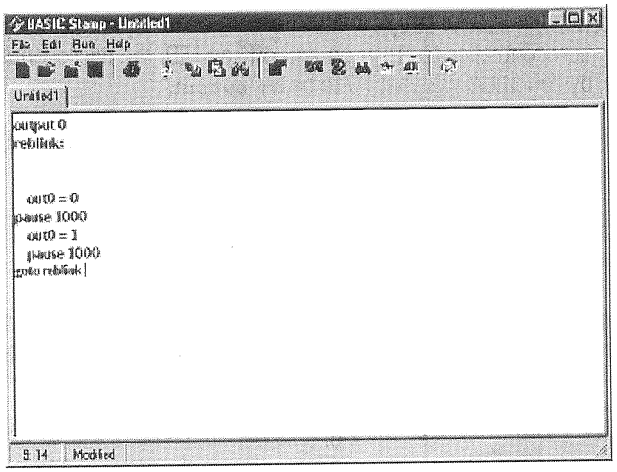


Figure 1.8:
BASIC Stamp Software
Type the code into the editor so it looks like this screen.

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

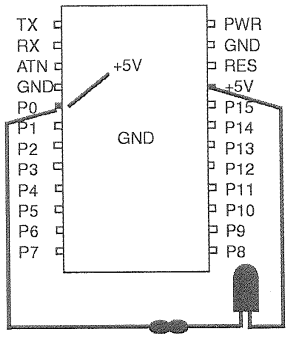
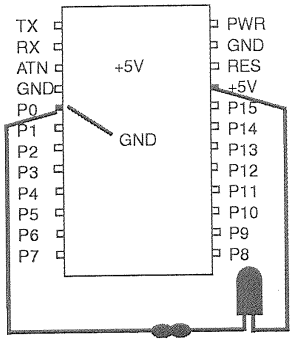


Figure 1.10: LED on
When P0 is "low" and current flows, the LED is on.



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".

Remark:

"Remarks" in your program are not executed like commands. They are ignored by the microcontroller. 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 P0 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
```

Using Your Basic Stamp to read out a Geiger Tube

In this part of the lab, we are going to use a the Basic Stamp as a data acquisition system for a Geiger counter.

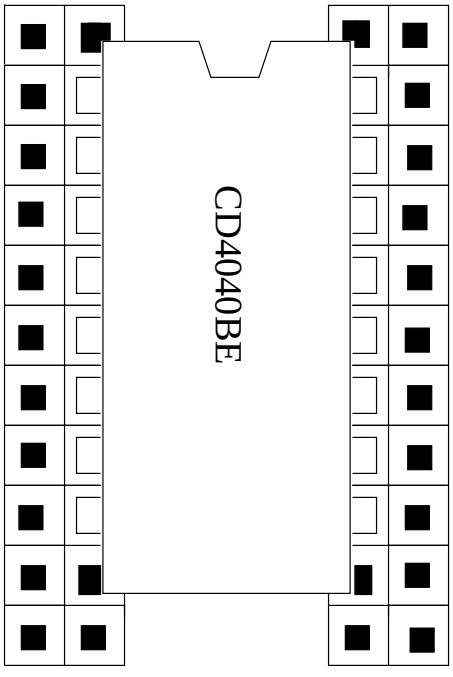
You need:

1) Board of Education with Basic Stamp 2
2) A power supply (or 9V Battery)
3) The interface cable to the computer (a DB9 serial cable)
4) Compter running DOS (need a serial port!)
5) A CD4040BE Ripple Carry Binary Counter (This is a Chip)
6) Model 528 Eberline Geiger counter and
7) The special connector cable
8) The Geiger Tube program disk (may already be in your computer)
9) Some jumpers for making connections
10) An amplifier in a clear plastic box (just an OP AMP really)
11) A few resistors and LED’s
12) A radioactive source (ask your instructor)

Procedure

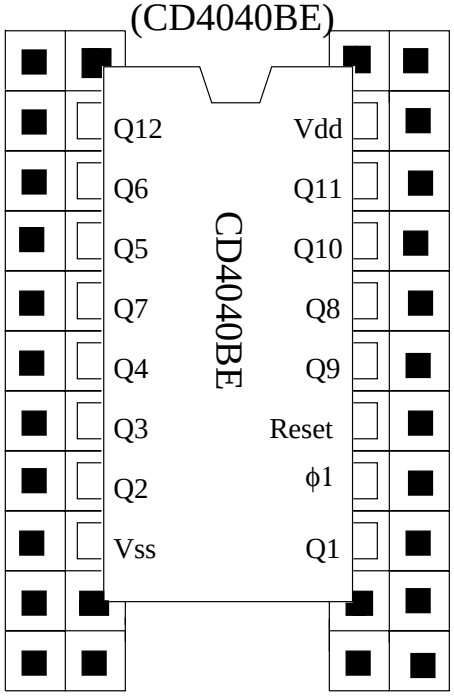
Set up the stamp as you did in the first part of the "What is a Microcontroller" Lab but leave the breadboard bare.

Now, put the CD4040BE on the Breadboard



Make the following connections:

Chip
(CD4040BE)



Chip	Board of Education
Vdd	Vdd
Vss	Vss
Reset	P0
ϕ1	P1
Q1	P4
Q2	P5
Q3	P6
Q4	P7
Q5	P8
Q6	P9
Q7	P10
Q8	P11
Q9	P12
Q10	P13
Q11	P14
Q12	P15

The idea is that we are going to pulse the ϕ1 line up and down.

The CD4040 outputs a binary number that is supposed to be the number of times the ϕ1 pin was pulsed.

We are going to use this feature to count pulses from a Geiger tube, but first we need to check and make sure everything was hooked up correctly. We are going to do this by pulsing P0 enough times to check each binary output. Q1 corresponds to the 0 bit.

In case you've never been exposed to binary numbers, they work like this for the CD4040 (and lots of other things too)

(In the table below, 1 means the pin is "high" or about +4V, 0 means the pin is "low" or about 0V)

Q12.....Q1	Number of Counts
000000000001	1
000000000010	2
000000000011	3
000000000100	4
000000000101	5
000000000110	6
000000000111	7

Q12.....Q1	Number of Counts
100000000000	2048
110000000000	3072
111000000000	3584
101000000000	2560
111111111111	4095
000000000000	0

A number is the sum of 2's, raised to the power of the bit position. You can actually watch this happening.

#counts = 2048*Q12+1024*Q11+512*Q10+256*Q9+128*Q8+64*Q7+32*Q6+16*Q5+8*Q4+4*Q3+2*Q2+Q1

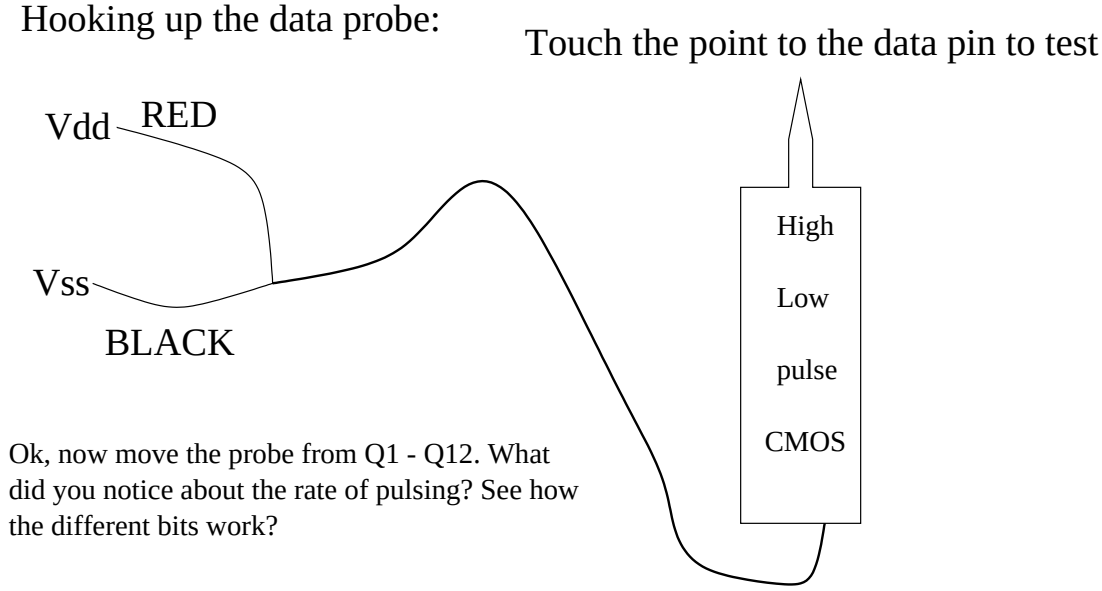
Once you have the chip hooked up, load and run the program: geigerte.bs2

The program is set up to pulse 128 times, read the binary output, and then form a word from the binary data. You can put different numbers into the program to test different pins. I.e. you could try 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048. This would let you know if you mixed up any pins.

Question: How many counts do you read if you pulse 4096 times? Why?

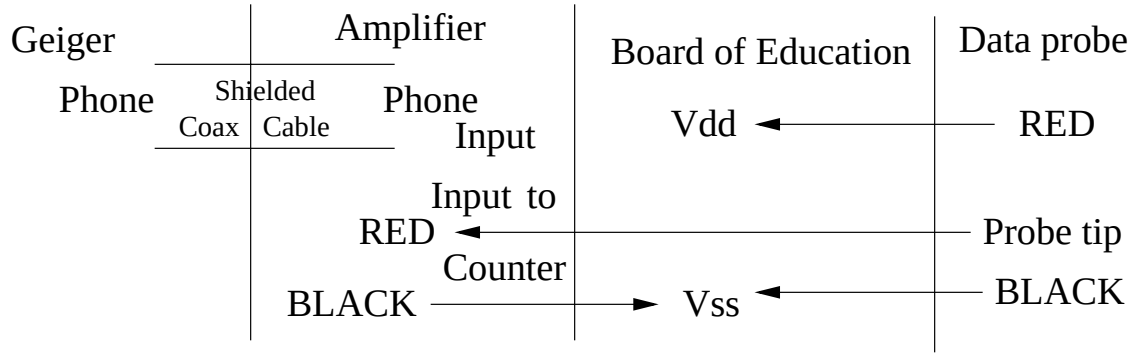
You can set the number of pulses very high and watch the operation of the counter with a data probe. You may need to vary the amount of time you pause between the high and low pin settings so you can hear or see individual pulses.

Hooking up the data probe:



If you are convinced that your counter is working, you can start to connect the Geiger Counter. Lets make sure the Geiger counter is working, at some level.

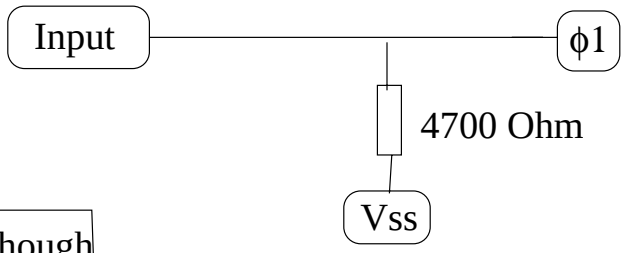
1) Check the battery by turning knob to BATT
2) OK? Turn knob so that X1 is selected
3) Have your instructor bring you a source. Co60 will be a good choice.
4) Bring counter close to source. Meter should raise a little. You don't need more than 2000/min. (Note: you may need to press RESET first)
5) Hook up the geiger counter to the Data Probe using these connections:



Repeat step 4, did you get pulses in the data probe?

Now, remove the connection from P1 to the CD4040 (we don't need to pulse artificially now) and hook up the Geieger counter "Input" to the $\phi 1$ pin. Place the source close to the Geiger tube and look at Q1 with the data probe. Getting any action? You may want to check your connections, or you may need to hook up a circuit to help the geiger counter trigger the CD4040. A suggestion is shown below (you may want to have your instructor check your set up before you try this, and you may wait 20 sec. or so and look again.)

" Conditioning " the input this way gives the CD4040 chip a better chance of recognizing a High level and a Low level. The resistor are sets a definite low level, and the Geiger counter will pulse the $\phi 1$ pin High. You may not need this though.



Now that the Geiger counter is working, leave the source in place and load the program: geigerme.bs2 Run it. It is set up to take 12 readings of 10 seconds lengths each. If you add up these readings and divide by 2 you should roughly duplicate the meter reading on the geiger counter.

You are ready to procede with some physics measurements if your counter set up checked out ok. You may find it convenient to modify the geigerme.bs2 program to make your data taking easier. Before you modify this program though, copy it to a file you won't destroy, like mygeig.bs2.

Proceed to the following experiments.

Do experiment 2.6 first. If you have time, graph your results. If the results make sense then: Hint: Set up your program to do this part automatically. You will still have to graph the results.

Do the supplement next. Be sure you don't change the geometry between the source and the Detector once you've started. You also may have to throw away your first data point.

Note on program structure:

Most of the programs in the lab had this kind of structure:

Top: Let the computer know what variables you need
Middle: Operate on variables
Bottom: Data output

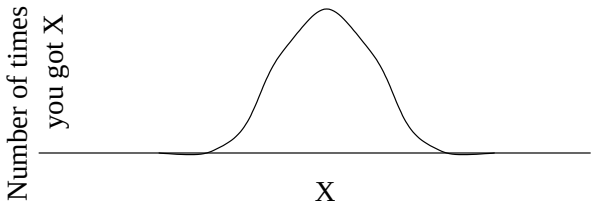
Experiment 2.6 (Supplement)

Counting Statistics

Your job in this lab is to see if your set up behaves the way you expect for a radioactive decay. The number of decays you observe in a given amount of time, for a given radioactive source and detector should follow a known statistical behavior. Lets say you were going to measure the number of decays from a Co60 source for a minute, and you did this 20 times. Each minute you wrote down the number of decays that occured in that minute. You don't expect the numbers to be exactly equal, but you do expect them to fall within some reasonable interval of counts. Statistically speaking, you'd expect the number of counts to follow a bell shaped curve (a Gaussian curve) if you got, on average, about 20 or more counts in a minute. In fact, if you define your Gaussian this way: (X is each number you wrote down)

$$\left\langle \frac{\text{Number of trials}}{\sqrt{2\,\pi\,\sigma}} \right\rangle e^{-0.5\left\langle \frac{(X-Xavg)}{\sigma} \right\rangle^2}$$

You expect to see a distribution that looks like:



If you took lots of data and made a graph of the number of times X was a certain number, you'd expect to see a curve like the one above. You'd also expect that:

$$Xavg = \frac{\sum_{i=1}^{n} Xi}{n}$$

n= # of trials

Xi = number of counts in a single trial

and

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (Xi - Xavg)^2}{n}}$$

Now, what is pretty cool about counting statistics is that

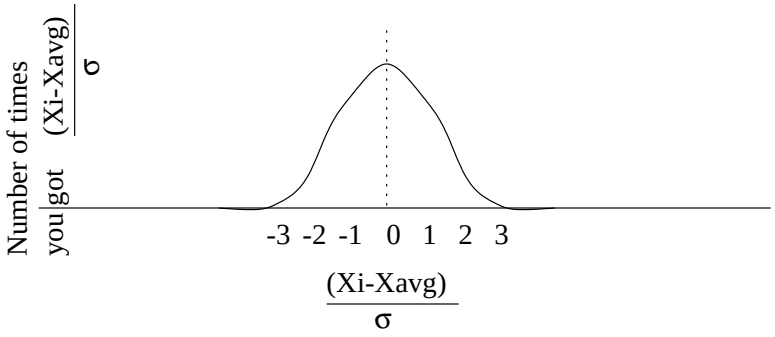
σ should be pretty close to $\sqrt{Xavg}$

for large n, and Xavg of around 20 or higher, and your detector worked, etc.

Now, in the write up from the Ortec Corporation for this lab, the definitions of the variables they use are a little challenging.

When they say R as a rate, what they really mean is that R is the number of counts in a SET interval. For instance, if you took 320 counts in one minute, R=320 and NOT 5.3/sec. The idea is that you keep the amount of time you look constant, and use the number of counts in that constant, set, time as R.

In the histogram of $\frac{(Xi-Xavg)}{\sigma}$ the units on the bottom should actually be dimensionless.



What you are doing is seeing if your estimate of the σ is reasonable for your data. You should see a Gaussian distribution of width 1. Sometimes this is called a Normal distribution.

EXPERIMENT 2.6

Counting Statistics

Purpose

As is well known, each measurement made for a radioactive sample is independent of all previous measurements, because radioactive decay is a random process. However, for a large number of individual measurements the deviation of the individual count rates from what might be termed the "average count rate" behaves in a predictable manner. Small deviations from the average are much more likely than large deviations. In this experiment we will see that the frequency of occurrence of a particular deviation from this average, within a given size interval, can be determined with a certain degree of confidence. Fifty independent measurements will be made, and some rather simple statistical treatments of the data will be performed.

The average count rate for N independent measurements is given by

$$\bar{R} = \frac{R_1 + R_2 + R_3 + \cdots R_N}{N} \qquad (12)$$

where R_1 = the count rate for the first measurement, etc., and N = the number of measurements.

In summation, notation $\bar{R}$ would take the form

$$\bar{R} = \frac{\sum_{i=1}^{i=N} R_i}{N} . \qquad (13)$$

The deviation of an individual count from the mean is $(R - \bar{R})$. From the definition of $\bar{R}$ it is clear that

$$\sum_{i=1}^{i=N} (R_i - \bar{R}) = 0. \qquad (14)$$

The standard deviation $\sigma = \sqrt{\bar{R}}$.

Procedure

1. Set the operating voltage of the Geiger counter at its proper value.

2. Place the ^{60}Co source far enough away from the window of the GM tube so that ~~4000~~ counts can be obtained in a time period of 0.5 min. 100 - 1000

3. Without moving the source, take ~~50~~ 20-50 independent 0.5-min runs, and record the values in Table 2.2. (Note that you will have to extend Table 2.2; we have shown only ten entries.)

The counter values, R, may be recorded directly in the table since for this experiment R is defined as the number of counts recorded for a 0.5-min time interval.

4. With a calculator determine $\bar{R}$ from Eq. (12). Fill in the values of $R - \bar{R}$ in Table 2.2. It should be noted that these values can be either positive or negative. You should indicate the sign in the data entered in the table.

Table 2.2*

Run	R	σ	$R - \bar{R}$	$(R - \bar{R})/\sigma$		$(R - \bar{R})/\sigma$ (Rnd'd Off)	
				Typical	Measured	Typical	Measured
1				− 0.15		0	
2				+ 1.06		+1.0	
3				+ 0.07		0	
4				− 1.61		−1.5	
5				− 1.21		−1.0	
6				+ 1.70		+1.5	
7				− 0.03		0	
8				− 1.17		−1.0	
9				− 1.67		−1.5	
10				+ 0.19		0	

*Typical values of $(R - \bar{R})/\sigma$ and $(R - \bar{R})/\sigma$ rounded off; listed for illustrative purposes only.

EXERCISES

a. Calculate σ, and fill in the values for σ and $(R - \bar{R})/\sigma$ in the table, using only two decimal places. Round off the values for $(R - \bar{R})/\sigma$ to the nearest 0.5 and record these values in the table. Note that in Table 2.2 we have shown some typical values of $(R - \bar{R})/\sigma$ and the rounded-off values.

b. Make a plot of the frequency of the rounded-off events $(R - \bar{R})/\sigma$ vs the rounded-off values. Figure 2.3 shows this plot for an ideal case.

Note that at zero there are eight events, etc. This means that in our complete rounded-off data in Table 2.2 there were eight zeros. Likewise, there were seven values of +0.5, etc. Does your plot follow a normal distribution similar to that in Fig. 2.3?

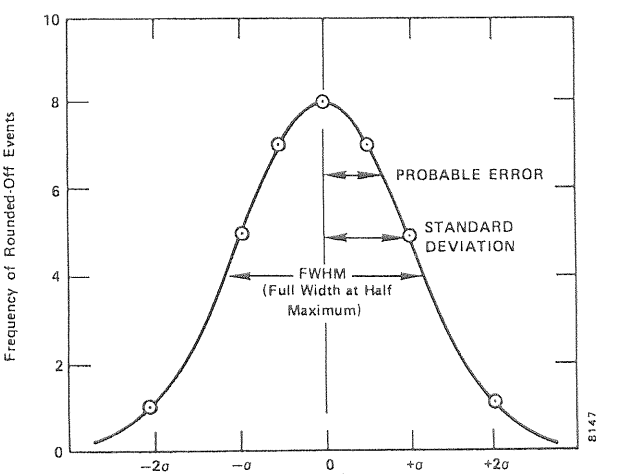


Fig. 2.3. Typical Plot of Frequency of Rounded-Off Events vs the Rounded-Off Values.

Experiment 2.6 (Supplement)

Bad things happen in 2's ?!?!?

Your job in this part of the lab is to verify one of the more bizarre consequences of the random nature of both noise and radioactive decays.

Consider a process such as particle decay where:

1) In a small interval of time, dt, there is at most a single decay
2) The probability of finding a decay in this interval of time is proportional to dt
3) A decay in dt is independent of decays at other times

The probability then of finding a decay in the time dt is $P_i(dt)=Mdt$ (M is a constant)

And the probability of not finding a decay in the time dt is $P_j(dt)=1-P_i(dt) = 1-Mdt$

If we keep in mind that the decays are independent the probability to get no decays in a long time, t+dt, can be expressed $P_j(t+dt) = P_j(t)P_j(dt)$ or, combined with the above, one can form the quantity

$$\frac{P_j(t+dt)-P_j(t)}{dt} = -MP_j(t)$$

$$\text{or } P_j(t) = P_j(0)e^{-Mt}$$

since $P_j(0) = 1$, $P_j(t) = e^{-Mt}$ ...so what?

Well, lets combine some probabilities. Figure the probability that you have no decay in time t, and one decay in dt is:

$$P_j(t)P_i(dt)=(e^{-Mt}\ Mdt)$$

And the probability/(unit length) becomes Me^{-Mt}

This is how the time between events should be distributed in a random process. You are going to verify this.

Procedure: Once you have finished with the counting statistics portion of the lab, load the program geigeran.bs2

This program loops 65000 times (about 2.5msec/loop) and checks to see if the counter has incremented. If so, you will get the number of loops since the last geiger tube hit. (You may need to put a 4700 Ohm resistor on the input to stop multiple hits)

Write down each number. Probably you need 250 or so points to do a good job. Make sure that the rate in the counter is not too high (a few/second is fine).

You can also do different rates as a check. If you completely remove the source, you can measure the background to see if it is random.

Does your data confirm the relationship Me^{-Mt} ? What is M for your data? What is the average time between events for your data?

Challenge: Can you get the computer to write the data to a file for you to analyze later?

Note on N events in time t:

It is interesting to consider what happens in over a long time scale where there is more than one decay. The probability for finding N decays in a length of time t+dt will be made up of the probability of finding N decays in t + 0 events in dt and N-1 decays in t and 1 event in dt [remember 1) from above].

$P_n(t+dt) = P_n(t)P_j(dt) + P_{n-1}(t)P_i(dt)$

$= P_n(t)(1-Mdt) + P_{n-1}(t)(Mdt)$

or

$$\frac{P_n(t+dt) - P_n(t)}{dt} = -MP_n(t) + MP_{n-1}(t)$$

Which has a solution (which I looked up):

$$P_n(t) = \frac{1}{N!}(Mt)^N e^{-Mt} \quad \text{and} \quad \sum_{N=0}^{\infty} P_n(t) = 1$$

Consider $\sum a^N P_n(t) = e^{aMt} e^{-Mt} \sum P_n(at)$

$= e^{aMt} e^{-Mt}$

Now a derivative with respect to a while you let a go to 1 gives you the average value of N, and by similar means, you can compute the standard deviation on N (which is $\sqrt{N}$).