Applied Sensors

Student Guide

VERSION 2.0

WARRANTY

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax 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 Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

COPYRIGHTS AND TRADEMARKS

This text is available in printed format from Parallax Inc. Because we print the text in volume, the consumer price is often less than typical retail duplication charges.

2.0.0-08.10.06-SCP

ISBN 978-1-928982-47-0

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, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. 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 DISCUSSION LISTS

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Forums menu. These are the forums that we operate from our web site:

- **Propeller Chip™** – This forum is for users of the multiprocessing Parallax Propeller chip.
- **BASIC Stamp®** – This forum is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- **SX Microcontrollers and SX-Key** – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3rd party BASIC and C compilers.
- **Stamps in Class®** – Created for educators and students, members discuss the use of the Stamps in Class series in their courses. Students, educators and hobbyists are welcome to participate.
- **Javelin Stamp** – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.
- **Robotics** – Designed exclusively for Parallax robots, this forum is intended to be an open dialogue for robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, Penguin™, and Propeller QuadRover™ robots are discussed here.
- **HYDRA** – A place for enthusiasts of the Propeller-based HYDRA game development system.
- **Parallax Educators** – A private forum exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this forum for educators to provide feed back and to obtain, develop, and share teaching materials. Educators may email stampsinclass@parallax.com for enrollment information.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, www.parallax.com. Please check the individual product page's free downloads for an errata file.

Table of Contents

Preface

Interfacing sensors, the focus of the activities in this text, is integral to many types of fieldwork. Think of yourself as a geologist, wanting to know more about El Niño, and how this famous phenomenon in the waters off the coast of South America changes weather patterns all over the world. You are going to need lots of measurements. Or, think of yourself as the operator of a water treatment plant, where a city full of people is counting on you to deliver pure water day and night. You are going to have to monitor the water and operate a computer-controlled plant to pump it across the city. Or, think of yourself as responsible for an orchard of apples. You need to keep close track of the weather so that you will keep one step ahead on irrigation and pest control to bring a healthy crop to market.

Think about home appliances such as clothes dryers, ovens and room thermostats. They all use microcontrollers for measurement and control, as do instruments in the factory, the laboratory, the hospital, and beyond earth out into space. The techniques of measurement in these different settings are all similar. What you learn here will apply to many fields.

About Version 2.0

This revision of *Applied Sensors* (formerly titled *Earth Measurements*) was necessary because the low-voltage pump used to develop the activities in Chapter 6 is no longer available. Though we no longer supply a pump or the related components in the Applied Sensors Parts Kit v2.0, we have kept Chapter 6 in the book for your reference and adaptation to commercially available low-voltage pumps. As Parallax identifies sources of compatible and reasonably inexpensive low-voltage pumps, we will post links on the Applied Sensors product pages at www.parallax.com. We also invite our customers to submit information about low-voltage pump resources, or project adaptations to other types of pumps, to editor@parallax.com for potential posting on our website.

Other typographical corrections and product reference updates have been made. Pagination of Chapters 1 through 5 should be similar, if not identical, to Version 1.4.

AUDIENCE

Applied Sensors was created for ages 17+ as a subsequent text to *What's a Microcontroller?* Like all Stamps in Class texts, *Applied Sensors* teaches new techniques and circuits with only minimal overlap between the other texts. New topics introduced in this text are a closed-loop feedback control system, serial communication, use of the BASIC Stamp EEPROM for data logging, calibration of sensors, conductivity in water, and the use of a sound transducer for human feedback. Instructors are invited to participate in the private Parallax Educators Forum to obtain support and additional related educational materials for this text if they are available. Email stampsinclass@parallax.com for enrollment instructions.

FOREIGN TRANSLATIONS

Parallax educational texts may be translated to other languages with our permission under our Volunteer Translators Program. Please email translations@parallax.com for details.

SPECIAL CONTRIBUTORS

The Applied Sensors text was written by Tracy Allen Ph.D. Dr. Allen is with Electronically Monitored Ecosystems, located in Berkeley, California (http://www.emesystems.com). EME Systems designs and manufactures instruments for environmental science. Some of their products are off-the-shelf, and others are customized systems for individual clients. The commercially available OWL2C On-site Weather Logger uses a BASIC Stamp 2 or 2pe microcontroller, providing programmable capabilities for a customer who doesn't want to use the default program. Dr. Allen has particular interest in programs that address integrated pest management on the farm, efficient use of natural resources, and understanding of endangered species or ecosystems. A recent project of Dr. Allen's consists of measuring the surface temperature of dairy cows to evaluate milk productivity. Dr. Allen is a frequent contributor to the Parallax forums, and Parallax is very appreciative of his continuing involvement with the Stamps in Class program.

Thanks also go to everyone on the Parallax Team, for those who provided ideas and content for this book, and in particular to Rich Allred for the technical graphics and to Jen Jacobs for the cover graphics. Aristides Alvarez gets credit for updating the book format to the Stamps in Class style, and for rewriting the programs to PBASIC 2.5. The whole Parallax Team that designs, manufactures, accepts orders, and packages the Stamps in Class products is recognized as integral to the success of the Stamps in Class program.

Chapter 1: Piezo and Temperature Transducer

Applied Sensors will guide you as you build, program, test, and calibrate a multi-sensor instrument with a data logger. With this instrument you will measure ambient temperature, water temperature, and light level. You will build and apply a conductivity sensor to various materials, and detect salinity in a cup of water. In an optional final experiment, you will maintain the water level in a cup with a pump and conductivity sensor. Feedback about the operation of the BASIC Stamp will be conveyed to you audibly with a piezo transducer, and collected data will be displayed on your computer screen. If this sounds exciting, great! If this sounds intimidating, don't worry. You will be introduced to each subsystem one at a time, and integrate them in small steps. The first phase of this progressive experiment includes:

- A piezo transducer that converts electrical impulses from the BASIC Stamp into musical tones
- Programming the BASIC Stamp to send Morse code via the piezo transducer
- A digital temperature sensor which is another transducer that converts temperature into a coded form that the BASIC Stamp can understand
- Programming the BASIC Stamp to take temperature readings and display them on the computer screen in the Debug Terminal

Temperature is of the first importance in any process. We all know from personal experience that temperature is important to our well-being. You are probably sitting in a comfortable room, in the range of 17 to 30 degrees Celsius (63 to 86 degrees Fahrenheit). There may be a thermostat in the room that holds the temperature at that comfortable value, using a heater or an air conditioner (or maybe not!?). What do you think the temperature is right now where you are? How about outside? If you don't have a thermometer, don't worry; you will have one before this experiment is over. We need only a transducer to measure the temperature, and another transducer to convey the temperature readings to our eyes and ears.

We live on a planet that is just the right distance from the sun and has the right kind of atmosphere to offer temperatures conducive to life, as we know it. Through our human technology and industry, from clothing and housing all the way through to modern electronic environmental controls, we have extended the range of temperatures where we can live.

It is not far-fetched to say that every process on earth depends on temperature in some way. Think of erosion of mountains. Every year water seeps into cracks in the rocks, freezes, expands, and breaks off pieces. Snow, rain, clouds, wind - nearly every aspect of the weather depends critically on temperature. A few tenths of a degree change in the temperature of the water in the South Pacific Ocean (El Niño) can affect the weather all over the world. How apples grow on trees, how the worms grow in the apples, how mosquitoes thrive in stagnant pools, how tadpoles survive to eat the mosquitoes, everything relating to agriculture and biology is dependent on temperature. Add to that the environment in factories, hospitals, laboratories, schools, homes, museums, and on and on. Suffice it to say that if you want to go into any career related to microcontrollers, you are going to have to know how to measure temperature.

Preparation

To complete the experiments in this text, you will need to have your BASIC Stamp Editor v 2.4 or higher installed and running on your computer. Then, you will need to hook up your BASIC Stamp 2 and Board of Education, or your BASIC Stamp HomeWork Board, to your computer with a programming cable. If you are using a USB board or USB to Serial Adapter, you will need to have the appropriate USB VCP drivers installed on your computer as well. A complete listing of the components required for all of the experiments can be found in Appendix A.

Parts Required

(1) Piezo transducer
(1) DS1620 Temperature Sensor
(1) 1 kΩ resistor (brown black red)
(1) 0.1 µF capacitor
(6) Jumper wires

Building the Circuit

It's always good to start out with a simple project, just to get into the swing of things. That is going to be the pattern in this series of experiments. You will start out with a warm-up project, and then move on to the main focus of the experiment. The warm-up to start this experiment is simply a buzzer, a sound output device. In fancy terms, it is an "annunciator" or a "piezoelectric transducer." It will be a big part of our user interface in the projects to come in *Applied Sensors*. Sure, we can also see results on the computer screen when your BASIC Stamp is hooked up to it via its serial cable. However, having

the annunciator will allow us to stand up and walk away from the computer, around the room, into the dark, outside into the sunlight, and still be able to "hear" what is going on.

Piezoelectric transducers Piezo comes from a Greek word that means "to squeeze or press", and electric comes from a Greek word that refers to amber, a mineral that can accumulate a charge of static electricity when rubbed. Crystals, such as quartz and also some ceramic and plastic materials, generate electricity when they are flexed back and forth. This is the piezoelectric effect. Electrical wires attached to the surface of such materials can pick up that electricity. This is the basis of some kinds of microphones.

A microphone is a transducer (Latin for "lead across") that transforms sound into electricity. The piezoelectric effect works in reverse too. If electricity is applied across some piezoelectric materials, they bend in response. They can be fabricated as a thin disk, with electrical connections on both faces, and wires attached. The disk is like a tiny drumhead. When connected to a rapidly alternating electrical voltage, it flexes back and forth, compressing the air which emits sound waves, and thereby becomes a piezo transducer. It turns electricity into sound. The electrical voltage has to be in the right frequency range to resonate with the natural tone of the tiny drumhead.

Sometimes a piezo transducer is packaged along with some electrical circuitry, so that all you have to do is connect it to a battery or to a power supply and it will buzz at one preset pitch. Such a device is called a piezo buzzer. The device we are using here is a simple piezo transducer. It will not buzz if we connect it directly to a battery. It will only produce sound when we provide audio frequency electrical impulses from the BASIC Stamp.

The piezoelectric transducer you will find in your parts kit is a black plastic cylinder with two pins sticking out the bottom and a sound hole in the top. The top of the case above one of the pins is labeled with a + sign.

- √ Build the circuit shown by the schematic in Figure 1-1 and the wiring diagram in Figure 1-2.
- √ Verify that your piezo transducer is positioned the same as the one shown in the wiring diagram in Figure 1-2.
- √ Set aside the DS1620 Temperature Sensor, 1 kΩ resistor, 0.1 µF capacitor, and the 4 remaining jumper wires to use later in the chapter.

FOLLOW THE WIRING DIAGRAMS EXACTLY! The six experiments in this *Applied Sensors* series will progress from unit to unit by adding new circuits onto the old ones already built on your Board of Education or HomeWork Board. To avoid having to rewire things later, please follow the suggested parts placement shown in each wiring diagram.

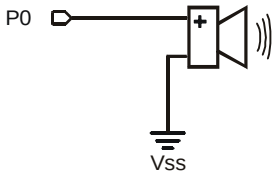


Figure 1-1
Piezo Transducer Schematic

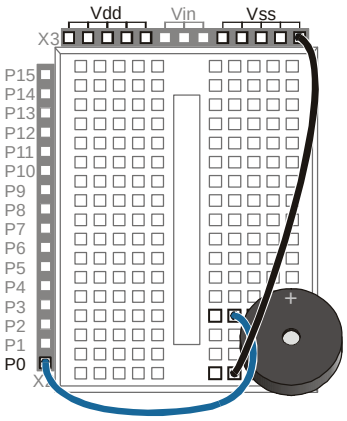


Figure 1-2
Piezo Transducer Wiring Diagram

- *Position and orient the piezo transducer as shown*

- *P0 connects to the + pin of the piezo.*

- *Other piezo pin wired to Vss (straight, or in two steps if necessary)*

Please note. The six experiments in this Applied Sensors series will progress from unit to unit by adding new circuits onto the old ones already built on the Board of Education or HomeWork Board. To avoid having to rewire things later, please follow the suggested parts placement.

Programming the Project

This experiment consists of three smaller sections that cover the piezo transducer, Morse code, and temperature measurement. The project is progressive.

Piezo Transducer

Now, to make noise with the piezo transducer, the BASIC Stamp has to supply a high frequency signal from P0. The PBASIC command to do this is **`FREQOUT`**. That's short for "frequency output."

√ Start the BASIC Stamp Editor on your computer.
√ Enter the program FirstSound.bs2
√ Make sure the Board of Education or the HomeWork Board is connected by its cable to the PC and to the power supply or battery.

√ Download the program to the BASIC Stamp. You can do this in three ways: While holding the CTRL key down, press the letter R, for Run, or press F9, or use the mouse to click on the ► button on the BASIC Stamp Editor's tool bar.

```
' Applied Sensors - FirstSound.bs2
' One line Program. (This is a comment)
' {$STAMP BS2}      (This is a compiler directive)
' {$PBASIC 2.5}     (This is a compiler directive)

FREQOUT 0, 1000, 1900
```

If all is well, you should hear a high-pitch beep. Each time you press the Reset button on the Board of Education or the HomeWork Board, you will hear it again. The reset button is easily found on the board, and is clearly labeled Reset. You can press it as often as you want, no worries. Pressing the button starts your program over again but will not erase it.

In case of difficulty during download: If RUN gives you a message about "hardware not found" or "communication error", then check to be sure that the cable that connects the PC to the Board of Education or the HomeWork Board is okay. Also check to be sure that the Board has a good power supply and that the power supply indicator light on the Board is glowing. If you need help, contact Parallax Tech Support for free at support@parallax.com, or call 1-888-99-STAMP in the US, or (916) 624-8333 outside the US. Or, visit our Stamps in Class forum at http://forums.parallax.com.

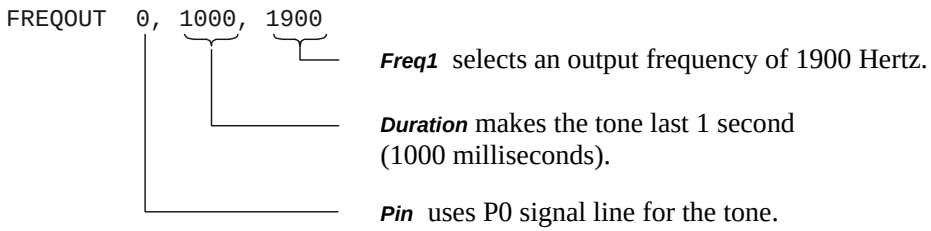
If you see a message indicating an error in your program, then check your typing. If the program is okay and CTRL-R is accepted without an error message, but it simply won't work, then check the wiring on your Board. Compare it to the wiring shown on the Wiring Diagram.

As you know from *What's a Microcontroller?* the comments at the beginning of the program are for the programmer's future reference. The compiler directives:

```
{$STAMP BS2}
{$PBASIC 2.5}
```

…identify the model of BASIC Stamp you are using and the language version. Beyond these comments and directives, this program consists of only one line of code, using the **FREQOUT** command.

There are three arguments in the **FREQOUT** command:



We can observe the voltage on P0 during the **FREQOUT** command using an oscilloscope such as the Parallax USB Oscilloscope (formerly the OPTAscope 81M). You will find out that the voltage goes back and forth from 0 to 5 volts very rapidly, and what comes out is fundamentally a 1900-Hertz sine wave that lasts for 1 second. To learn more about **FREQOUT** or any other PBASIC commands, you may click on the book icon on the BASIC Stamp Editor's tool bar, then select PBASIC Reference.

Figure 1-3 shows a screen capture of this signal taken with the Parallax USB Oscilloscope. The characteristics of the signal can be measured with the cursors, which are the red and blue lines at the peaks of the sine wave. In our actual measurement, the frequency was around 1.86 KHz, which you can read in the Cursors display. If you want to learn to use an oscilloscope, we recommend the Stamps in Class tutorial *Understanding Signals* listed in the Further Investigation section at the end of this chapter.

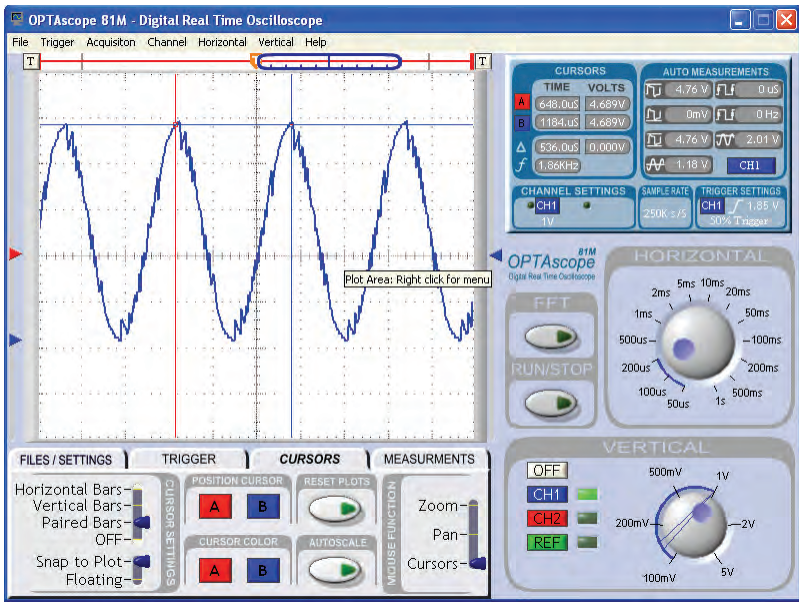


Figure 1-3
OPTAscope
Screen Capture

Argument: An argument is a number that governs the behavior of a command or a process. In the **FREQOUT** command, the arguments of the command specify what pin to use, how long the sound will be, and what the frequency will be.

Now it's time to experiment!

√ Modify the program by changing the ***Freq1*** argument from 1900 to 3800, resulting in a higher pitch:

```
FREQOUT 0, 1000, 3800
```

√ Download the modified program to your BASIC Stamp.
√ Pay attention that what you hear is a higher pitch.
√ Listen to it a couple of times, by re-running the program from the BASIC Stamp Editor and by pushing the Reset button on your Board. Don't be afraid you are going to wear out the BASIC Stamp by reprogramming it lots of times. You can reprogram the BASIC Stamp at least a million times.
√ Now try changing the ***Duration*** argument to make the tone last longer:

```
FREQOUT 0, 2000, 3800
```

√ Change ***Freq1*** back to 1900, and add the optional ***Freq2*** argument to the **FREQOUT** command to play two tones at once.

```
FREQOUT 0,2000,1900,2533
```

The number 2533 is equal to 1900 times 4/3, the musical interval "fourth."

√ And try the following frequency combination:

```
FREQOUT 0, 2000,1900,1903
```

√ How do you explain what you hear? Try changing ***Freq2*** to 1901, then 1902, then 1903 again. Do you hear the pattern?

√ And try a very short duration, to make a click 2 ms long:

```
FREQOUT 0, 2, 1900, 3804
```

Feel free to experiment. By experimenting with individual BASIC Stamp commands, you can become aware of possibilities that may be of use in programs later on.

Morse Code

An "audio annunciator" is a device that gives sound feedback about what is going on in a system. Having an audio annunciator on the Board of Education or the HomeWork Board is going to be very useful throughout these experiments in *Applied Sensors*. In Chapter 2, we will program it to send numbers using Morse code, shown in Table 1-1, and use the code to annunciate the temperature readings. Morse code is a fine way to send messages using sound.

Table 1-1: Morse Code Numerals						
Numeral:	**Morse Code**					**Binary**
0	dah	dah	dah	dah	dah	11111
1	dit	dah	dah	dah	dah	01111
2	dit	dit	dah	dah	dah	00111
3	dit	dit	dit	dah	dah	00011
4	dit	dit	dit	dit	dah	00001
5	dit	dit	dit	dit	dit	00000
6	dah	dit	dit	dit	dit	10000
7	dah	dah	dit	dit	dit	11000
8	dah	dah	dah	dit	dit	11100
9	dah	dah	dah	dah	dit	11110

The Morse code is based on sending patterns of short and long sounds. The long sound is always three times as long as the short sound. The short sound is called "dit" and the long sound is called "dah." The numerals are all made up of five dits and dahs. The letters of the alphabet have from one to four sounds, and the most common letters have the shortest patterns (for example, e = dit, t = dah, s = dit dit dit, q = dah dah dit dah). Punctuation marks have six sounds, e.g. period = dit dit dah dah dit dit. Within one letter or numeral, the time between sounds is supposed to be the same length as the dit. The time between different digits in a sequence like "50" is supposed to be the same length as a dah. The "binary" column is there just to show how you might think of Morse code as a binary number.

In these experiments, we will use only the numerals. TwoDigitMorse.bs2 is a program that sends the two-digit number "50" as Morse code. You do not have to type in the remarks, but you have to include the compiler directives. Recall that remarks are the apostrophe (') and everything that follows it on the line.

√ Enter the program TwoDigitMorse.bs2. into your BASIC Stamp Editor.

```
' Applied Sensors - TwoDigitMorse.bs2
' Morse code two digits test.

'{$STAMP BS2}
'{$PBASIC 2.5}
```

```
Dit              CON        70          ' Short span of time in milliseconds.
Dah              CON        3*Dit       ' Longer time, 3 times the above.
index            VAR        Nib         ' Index.

FOR index=1 TO 5                        ' Send 5 sounds.
  FREQOUT 0, Dit, 1900                  ' Send a dit.
  PAUSE Dit                             ' Short silence.
NEXT

PAUSE Dah                               ' Longer silence between digits.

FOR index=1 TO 5                        ' Send 5 sounds.
  FREQOUT 0, Dah, 1900                  ' Send a Dah.
  PAUSE Dit                             ' Short silence.
NEXT
```

- √ Run the program.
- √ Press the Reset button on your board if you want to hear the number 50 again.

Can you modify your program to send the most famous Morse code message of all, SOS?

You should already be familiar with the **FOR-NEXT** loop from the *What's a Microcontroller?* text. Think about how the program incorporates the rules of the Morse code. Note how it starts off by defining a constant named **Dit** in milliseconds, and then **Dah** is defined as a constant equal to three times **Dit**. PBASIC allows you to do that, to define one constant mathematically in terms of another. That's convenient, because it allows you to change the overall speed by changing only the **Dit** constant, and **Dah** will fall into place.

- √ Modify TwoDigitMorse.bs2 by changing the **Dit** constant from 70 to 140.
- √ Do it again, changing **Dit** to 35.
- √ Listen to the effect on the overall speed.

The important thing to note is that the ratio between the **Dit** and the **Dah** is always going to be 1:3. This is only an introduction. We will write a serious Morse program in Chapter 2, to annunciate temperature readings.

Temperature Readings from the DS1620

Now for a complete change of pace! Let's move on to the main topic, acquiring some temperature readings. In engineering, we usually use the word *acquire*, instead of *get*,

when we refer to data or readings. Your Board of Education or your HomeWork Board is going to become your data acquisition system.

The DS1620 is a modern temperature transducer (portions of the DS1620 data sheet are included in Appendix D). There is that word *transducer* again. Here, it refers to a device that transforms temperature into an electrical signal. The DS1620 takes temperature as its input, and transduces that value into a digital code that the BASIC Stamp can understand. The digital code represents the temperature of the DS1620 chip.

- √ Disconnect the battery or power supply to your Board of Education or HomeWork Board. A word to the wise: always do this before you change a circuit, as it is all too easy to touch a wire in the wrong place and risk burning something out.
- √ Locate the parts that you set aside at the beginning of this chapter: the DS1620 Temperature Sensor, 1 kΩ resistor, 0.1 µF capacitor, and the 4 remaining jumper wires.
- √ Build the circuit for the DS1620 as shown in Figure 1-4, following the positioning for the wiring diagram shown in Figure 1-5. Please note that the piezo circuit built previously remains in place in the schematic and wiring diagram, as it should on your Board.

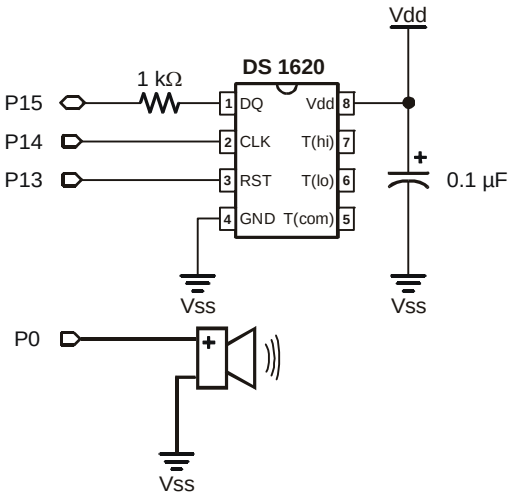


Figure 1-4
DS1620 Schematic

Schematic of the wiring diagram depicted in Figure 1-5. Remember – the piezo transducer portion of the circuit has already been built.

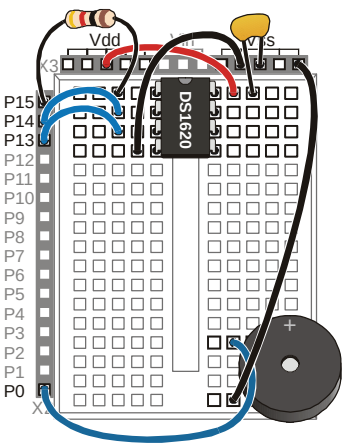


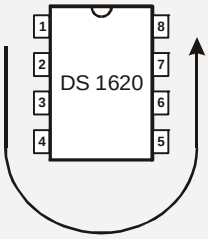
Figure 1-5
DS1620 Wiring Diagram

- *Plug the DS1620 in at the edge of the breadboard, taking care to orient it properly.*
- *0.1 µF capacitor from Vdd to Vss*
- *DS1620 pin 4 wired to Vss.*
- *DS1620 pin 8 wired to Vdd.*
- *1 kΩ resistor connects P15 to DS1620 pin 1.*
- *DS1620 pin 2 wired to BASIC Stamp P14.*
- *DS1620 pin 3 wired to BASIC Stamp P13.*

√ Make sure you lined up the DS1620 in at the very end of the breadboard when you plugged it in, observing that there is an indicator at one end of the DS1620 package to indicate where pin 1 is connected. Be careful not to reverse the power supply connections!

√ Double check your wiring, or better yet, have someone else check it, before you reconnect the power.

Which Way is Up? The DS1620 is an 8-pin DIP package. The indicator denoting the DS1620's pin 1 is a small notch on top of the chip. On parts like the DS1620 and BASIC Stamp, the pins are always counted counterclockwise starting from the mark. The mark can be a bump, round depression, notch, beveled edge, etc.



Now it's time to program the DS1620, literally. The DS1620 is itself a little computer. More accurately, it's a smart sensor. It can remember certain settings and do some pretty nifty tricks all on its own. Smart sensors are being used more and more in electronics and in the fields of environmental and industrial monitoring and control.

√ Enter the program DS1620Configuration.bs2 into your BASIC Stamp Editor.

```
' Applied Sensors - DS1620Configuration.bs2
' Configure the DS1620 for CPU continuous conversion.
'{$STAMP BS2}
'{$PBASIC 2.5}

LOW 13                              ' Puts the DS1620 in the waiting state.
FREQOUT 0, 1000, 3800               ' Sound shows us the program is running.

HIGH 13                             ' Tells the DS1620 a command is coming.
SHIFTOUT 15, 14, LSBFIRST, [12,2]  ' Command  to set DS1620 configuration 2.
LOW 13                              ' Completes the command cycle.

END                                 ' End of program.
```

- √ Double-check your typing.
- √ Download DS1620Configuration.bs2 into your BASIC Stamp and run it.

You will hear the one-second tone. That's all. But a lot has happened. The **SHIFTOUT** command sends two bytes, 12 and 2, to the DS1620. The 12 is a command to the DS1620 to get ready for the configuration, and the 2 is the actual configuration. Here are the four possible configurations:

0: No CPU, continuous conversion
1: No CPU, one-shot conversion
2: Yes CPU, continuous conversion
3: Yes CPU, one-shot conversion

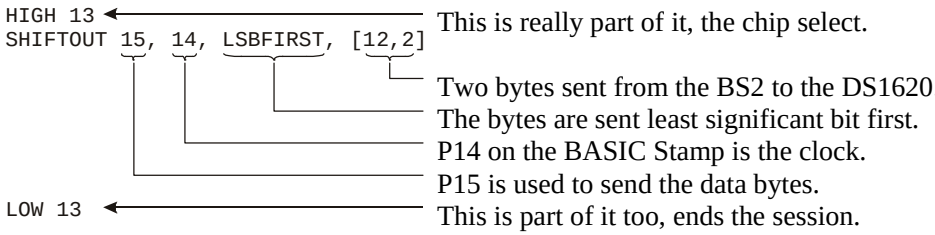
What does that mean? By selecting configuration 2, we are telling the DS1620 that we want it to send its readings to a CPU (Central Processing Unit—the BASIC Stamp). The alternative is for it to sit there and monitor temperature on its own, and not send back any readings. What good would that be? We asserted that the DS1620 is a smart sensor. Those other pins we are not using on the DS1620 could be wired up to a fan or heater, and set to regulate the temperature in a room or in a terrarium. The DS1620 also has a command that allows you to set a desired temperature. You will hear more about regulation of temperature in Chapter 6. But that is the way we are using it here, and we have not connected anything to those pins.

By setting the CPU option as "Yes" the DS1620 will send data back on the serial line when it receives commands. The term "continuous conversion" means that it will read temperature over and over and always have a current value available. The term "one-shot" (which we are not using) means that it will read the temperature once and then stop

until it receives a new command. The one-shot mode is used when an engineer needs to get the best battery life.

Now that we have sent the configuration, the DS1620 will not forget the setting. It is stored in memory inside the DS1620 in a kind of memory (EEPROM, like the BASIC Stamp program memory) that is not lost when the power supply is turned off.

The heart of the DS1620Configuration.bs2 PBASIC program is the **SHIFTOUT** command. The sequence is an example of synchronous serial communication. It will pay for you to understand how it works. Lots of modern electronics found in everything from pagers to satellites use these ideas. One main reason for this popularity is that devices that use serial communication can be made very small, because there don't have to be many wires connecting them. Here are the arguments of the command:



To explain how it works, I'll try an analogy using a stick figure dance. Please refer to Figure 1-6. The BASIC Stamp is at the bottom and the DS1620 is at the top. The DS1620 starts off with a zero as its configuration in memory.

- The BASIC Stamp starts the **SHIFTOUT** dance by raising the left hand. That is a wake-up call to the DS1620, and it means get ready, this message is for you.
- Then the BASIC Stamp taps out the first 8 beats on the clock pin with its foot.
- On each tap, the BASIC Stamp holds his right hand either low to signal a zero, or high to signal a one. Those are the digits of a binary number, sent out, least significant bit (LSB) first on the data pin.
- The DS1620 watches BASIC Stamp's right hand at each tap.
- After eight taps, DS1620 has the binary number 12 and recognizes it as a command. The BASIC Stamp knows in advance that DS1620 will interpret 12 as a command. (The command set is determined by the engineers at Dallas Semiconductor, the manufacturer of this part).

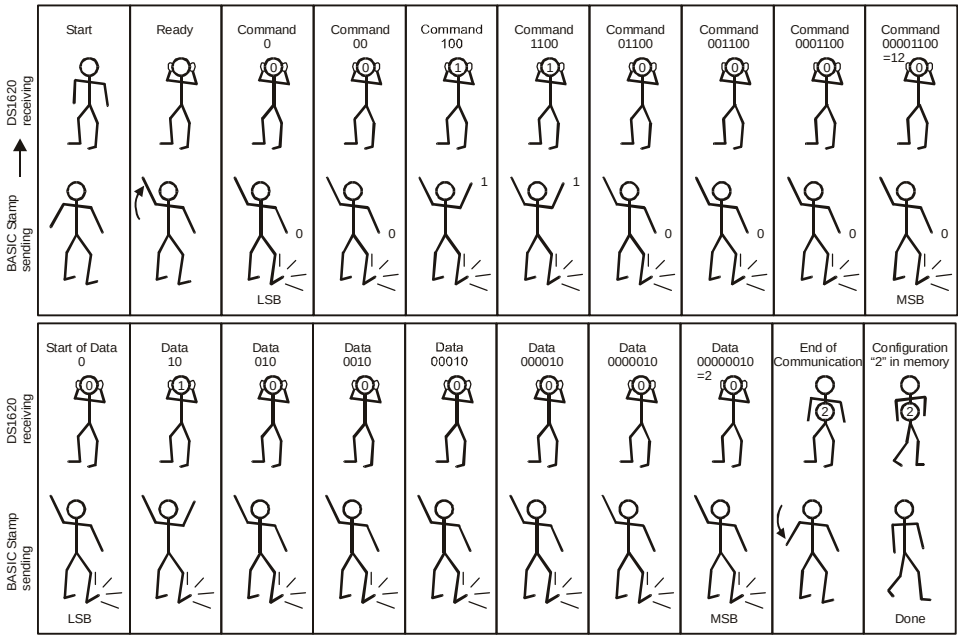


Figure 1-6: SHIFTOUT Dance

It isn't over yet!

- The DS1620 is now waiting for another binary number to follow the 12.
- The BASIC Stamp taps out 8 more beats.
- The DS1620 watches BASIC Stamp's right hand at each tap. This time it gets the number 2.
- The DS1620 stores the 2 in its EEPROM memory. Now the DS1620 is configured.
- The BASIC Stamp puts down its left hand to signal that the sequence is finished.
- The BASIC Stamp and the DS1620 are no longer in communication.

All that signaling is taken care of automatically, in less than 1/1000 second, by the **SHIFTOUT** command. Figure 1-7 shows the same thing in a timing diagram as an engineer might draw it.

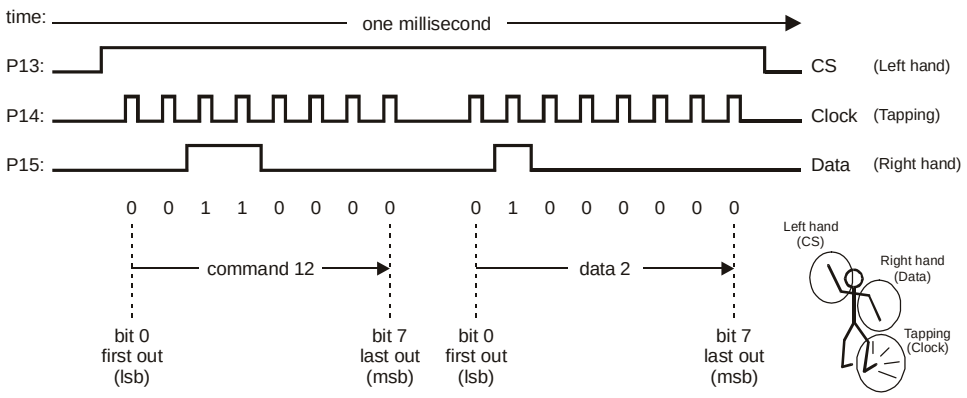


Figure 1-7: Timing Diagram

Note that 12 decimal = 00001100 binary, and 2 decimal is 00000010 binary.

When reviewing the timing diagram from Figure 1-7 consider the following:

- P13 starts the exchange by going from 0 to 5 volts. The command ends when P13 goes back down from 5 to 0 volts. P13 is often called the chip select or chip enable.
- P14 is the clock and puts out a series of 16 pulses, 0 to 5 volts, in two groups of 8.
- P15 is the data line and puts out either 0 or 5 volts in each time slot, synchronized with the clock pulses on P14. The first group forms the 12 (00001100 in binary), and the second group forms the 2 (00000010) in binary.
- Note the **`LSBFIRST`** argument for the **`SHIFTOUT`** command. The least significant bit comes first in the time sequence.
- This whole transmission of 16 clock cycles takes about 1 millisecond, 1/1000 of a second, and it happens automatically under the **`SHIFTOUT`** command.

If you want more explanation, please refer to the *BASIC Stamp Manual* where it describes the **`SHIFTOUT`** command in detail, and where there is also an application note. This is called synchronous serial communication, because the data is synchronized with the clock ticks that come from the BASIC Stamp. The BASIC Stamp is commonly referred to as the master and the DS1620 as the slave. That is because the clock pulses and commands originate in the BASIC Stamp.

Now for the main event: to read the room temperature from the DS1620.

√ Enter the program DS1620.bs2 into your BASIC Stamp Editor.
√ Download the program to your BASIC Stamp.

```
' Applied Sensors - DS1620.bs2
' Obtain temperature readings from the DS1620.
'{$STAMP BS2}
'{$PBASIC 2.5}

x         VAR        Byte          ' General purpose variable, byte.
degC      VAR        Byte          ' Variable to hold degrees Celsius.

' Note: DS1620 has been preprogrammed for mode 2.

OUTS=%0000000000000000             ' Define the initial state of all pins,
     'FEDCBA9876543210
DIRS=%1111111111111111             ' as low outputs.

FREQOUT 0, 20, 3800                ' Beep to signal that it is running.

HIGH 13                            ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [238]   ' Send the "start conversions" command.
LOW 13                             ' Do the command.

DO                                 ' Going to display once per second.
  HIGH 13                          ' Select the DS1620.
  SHIFTOUT 15, 14, LSBFIRST, [170] ' Send the "get data" command.
  SHIFTIN 15, 14, LSBPRE, [x]      ' Get the data.
  LOW 13                           ' End the command.

  degC = x / 2                     ' Convert the data to degrees C.
  DEBUG ? degC                     ' Show the result on the PC screen.
  PAUSE 1000                       ' 1 second pause.
LOOP                               ' Read & display temperature again.
```

In case of difficulty from an error in your program: If you get a message about an error in your program, you may have typed something wrong. The BASIC Stamp Editor program will position the cursor near where the error occurred, and will often display a message giving you a hint as to the problem. Look for any error near the cursor. If the error message you see is about "hardware not found" or "communication error", then be sure your Board has power and that the cable to the PC is connected properly. If all that goes okay, but the program does not work, then you will have to decide whether the problem is in the program or in your wiring of the DS1620.

- √ Look at the Debug Terminal to see the current temperature readings appear once per second. The readings are in units of degrees Celsius.
- √ Hold your finger on top of the DS1620 chip; you should see the temperature rise.
- √ Put your Board under a lamp or in the sun, and observe the time it takes for it to heat up some more.
- √ Move it away from the heat source and watch it cool down.
- √ Cool it down faster by fanning air across it.
- √ Measure the temperature near the floor.
- √ Measure the temperature on top of your PC.
- √ Measure the temperature next to your body.
- √ Use your new data acquisition device to find any other interesting warm or cool spots near your computer, and measure those! Experiment!

Which one of these temperatures (if they are different) will be the one you call the room temperature? Usually, HVAC engineers (Heating, Ventilation and Air Conditioning) prefer to use a temperature reading that is taken in the shade at a position not too close to sources of heat, like computers and bodies. This is called a representative temperature. In the real world, there can be lots of variation over even small distances and short times. You always have to make some choice about where and when is the best place and time to make a measurement.

What is going on in the program? First a word about the **OUTS** and **DIRS** statements:

```
OUTS=%0000000000000000   ' Define the initial state of all pins
     'FEDCBA9876543210
DIRS=%1111111111111111   ' as low outputs.
```

When using the BASIC Stamp, or any microcontroller, there will be pins connected to the outside world, and those pins can be either an input or an output, and if it is an output it can be either output high or output low. You are already familiar with the **OUT** and **DIR** variables from *What's a Microcontroller?* Here, with an "S" on the end, the statements control all 16 I/O pins, numbered from 0 to F (Note the apostrophe in front of the "F"--above that makes it a remark – and it is just there for reference.) The BASIC Stamp I/O pins are numbered from P0 to P15, where A = 10,...,F = 15.

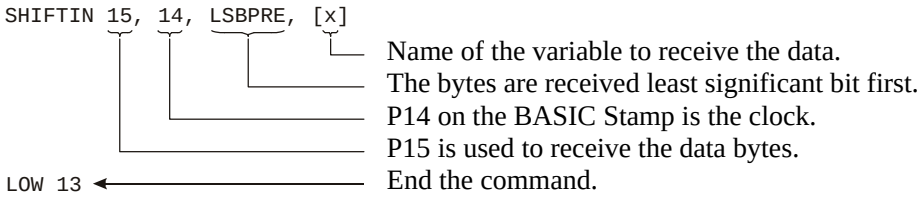
It is good programming practice to start off every serious program by putting all of the microcontroller pins into a known, desirable state. When the BASIC Stamp is first turned on or reset, all of the pins are configured by default as inputs. This is a fine state for a

microcontroller to start up in. You, the designer, are in charge of making the pins outputs as needed. On the other hand, if a pin is not connected to anything, it is not a good idea to leave it as input. Unconnected inputs may cause the microcontroller to behave erratically or to draw excessive power from the battery. The above instructions turn all of the pins on the BASIC Stamp into low outputs. That is what we want at first for the piezo transducer and for the DS1620. All the other pins are made low outputs just as a matter of principle. Reasons to do otherwise will arise as we progress through these experiments. For more information on the **DIRS** and **OUTS** command, please refer to the *BASIC Stamp Manual* or the Help file in your BASIC Stamp Editor.

The main action in the temperature program comes from the **SHIFTOUT** and **SHIFTIN** commands.

The first **SHIFTOUT** should look familiar. You see the familiar sequence: It sets P13 high, and then sends one byte, 238, out to the DS1620, and then sets P13 low again to end the sequence. Inside the DS1620, the 238 is a command that tells it to start converting temperature into digital codes. The 238 command needs to be sent at least once after the DS1620 is powered on. Unlike the configuration command, this one is not stored in the permanent memory of the chip.

Next comes the heart of the routine, to read the temperature from the DS1620. Again you see the familiar sequence: It sets P13 high, and then sends one byte, 170, out to the DS1620. So far so good. The DS1620 interprets the 170 as a command for it to get the current temperature reading and send it back to the BASIC Stamp. Now things get interesting. The DS1620, in response to the 170 command, takes control of the data line. The BASIC Stamp moves on to the **SHIFTIN** command. Here are the arguments:



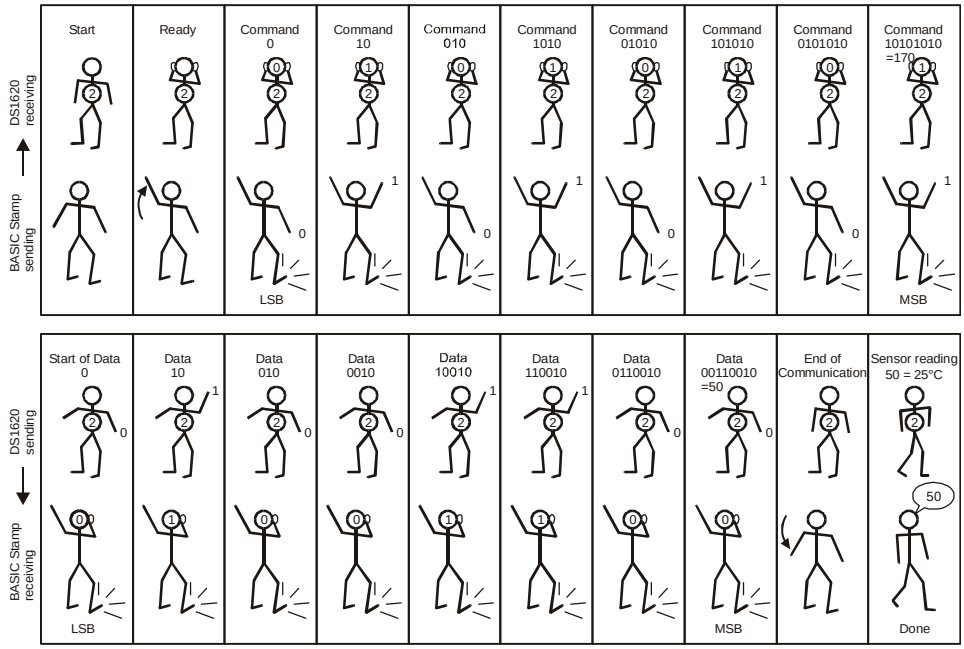


Figure 1-8: SHIFTIN Dance

P15 on the BASIC Stamp is now an input, whereas for **`SHIFTOUT`** it was an output. The BASIC Stamp is now ready to receive data from the DS1620. This is pictured once in Figure 1-8, and again as an engineering timing diagram in Figure 1-9.

Observe that the BASIC Stamp is still in charge of the timing. The BASIC Stamp is still the master and the DS1620 is the slave.

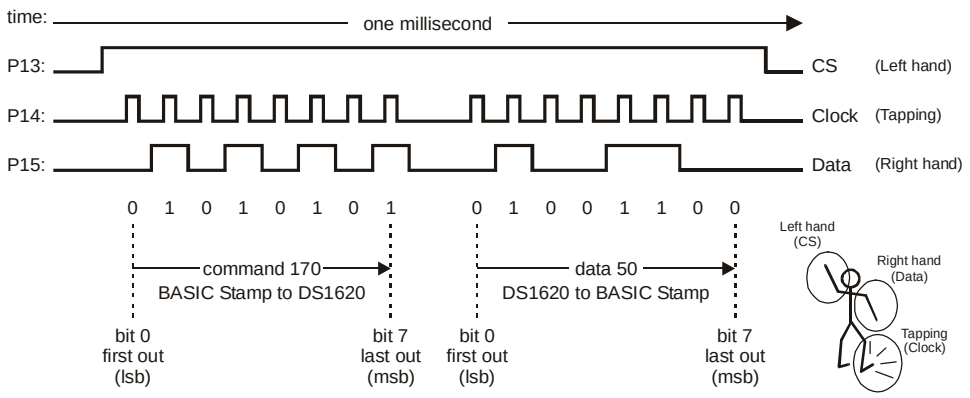


Figure 1-9: Timing Diagram for SHIFTIN

Each time the BASIC Stamp sends out a pulse on the clock line P14 (taps its foot), the DS1620 signals the next bit of the temperature byte. It starts with the least significant bit first. The **LSBPRE** means that the BASIC Stamp looks for the least significant bit before it sends out the first clock pulse. It goes like this: get 1st bit, pulse clock, get second bit, pulse clock, and so on until it has all 8 bits. The BASIC Stamp stores the data it receives from the DS1620 in the variable, **x**.

If the temperature is 25 degrees Celsius, the DS1620 sends back the value 50, which is two times the temperature. In binary, 50 is 00110010. The bytes that the DS1620 sends out are always two times the temperature. If the temperature is 25.5 degrees C, then the byte that the DS1620 sends back will be 51. Each step in **x** represents 0.5 degrees C. That is the *resolution*, the smallest change in temperature that the sensor detects.

Our program then converts the raw value of **x** to temperature:

```
degC = x / 2                ' Convert the data to degree C.
```

The BASIC Stamp uses integer arithmetic. It throws away the 0.5 degree remainder. Both 50/2 and 51/2 come out as degC = 25, and 52 and 53 both come out as degC = 26, and so on. There are ways to keep the half degree resolution, but we won't pursue that here. (But you can do so as a challenge!)

The temperature is sent to the Debug Terminal by this command:

```
DEBUG ? degC                        ' Show the result.
```

The "?" makes the BASIC Stamp send "**degC =**" and then the actual value of **degC** to the Debug Terminal, with each entry on a new line.

What are the operational limits of the DS1620?

The DS1620 is perfectly capable of measuring temperatures below zero, down to -25 Celsius. That would be important if you were out doing research on snow in Alaska, or if you were designing a control system for a freezer. The trouble is, the program we just wrote does not handle negative temperatures correctly. When the temperature goes to -1 degrees C, our reading would be degC = 127 instead of degC = -1. In order to read negative temperatures, we would have to take a couple more steps, which would complicate the program more than we want to get into at this time.

As it stands, zero degrees Celsius is the operational limit on the low end of our sensor circuit. Operational limits are everywhere in engineering, and they come up for all kinds of reasons, both in the software and hardware and in the properties of materials. This particular operational limit comes from a short cut we have taken in writing the software. That will be justified so long as the temperature is above freezing, but becomes a "bug" if we try to go below freezing. A famous software operational limit is the Y2K bug, where a software shortcut taken in the latter half of the 20th century led to an operational failure or glitches in the year 2000.

Now for a valuable experiment like this one, you should save the program you have just typed in and debugged. In this series of experiments, we are going to build up a large program, one piece at a time. This is the first piece you will be able to reuse. If you didn't do so already, you may want to enter the remarks attached to the program. That will reinforce your understanding, and it will also make it easier for you to pick up the program the next time you look at it, in Chapter 2.

Decide what you want to name the program. Your instructor will have directions, depending on how your class is set up to share the PCs. The program will have an extension of "bs2." Let's say you decide to name the program "DS1620.bs2."

This is how you save the program in the Parallax BASIC Stamp Editor:

- √ From the BASIC Stamp Editor menu bar, click File.
- √ Click on Save from the drop-down menu.

- √ In the Save As window, navigate to the directory where you want to save the program.
- √ Type in the name of your program. The .bs2 file extension will be added automatically unless you choose another option.
- √ Click Save.

Your program can be re-opened and re-named from inside the BASIC Stamp Editor when it comes time to add to it in the upcoming exercises.

Challenge!

1. Write a program using a sequence of **FREQOUT** commands to play a simple tune. Look up the **FREQOUT** command in the *BASIC Stamp Manual*. You will find an example of how to play Mary Had a Little Lamb. (Okay, you can try Stairway to Heaven, or Beethoven's 5th, if you prefer. You will discover some of the high fidelity limitations of the piezo transducer!) If you want to explore microcontroller music more thoroughly, see *What's a Microcontroller v2.0* or later for details.

2. Define a variable **degF** for Fahrenheit. Display both degrees Celsius and degrees Fahrenheit on the Debug Terminal. Use either formula:

   ```
   degF = degC * 9 / 5 + 32    or          degF = x * 9 / 10 + 32
   ```

   Is one formula better than the other? Why? Observe how the readings change as you gradually change the temperature of the DS1620 chip.

3. Display degrees Celsius resolved to 0.5 degrees. Recall that the result that comes from the DS1620 is a binary number where each bit represents 0.5 degrees. To get degrees, we divided by 2 and lost a bit of information (literally, one bit). You can display the result as 205 to represent 20.5 degrees C. Hint: multiply by 5 instead of divide by 2.

4. If the temperature is greater than (you choose a value), play an alarm tone on the piezo transducer. Make the alarm stop when the temperature goes back down. Then modify it so that the alarm continues, even when the temperature goes back down. Under what circumstances would each kind of alarm be appropriate?

Further Investigation

"What's a Microcontroller ?", Student Guide, Version 2.2, Parallax, Inc., 2004

Written by Andy Lindsay of Parallax, Inc., this text begins with detailed instructions for setting up and using your BASIC Stamp and Board of Education or HomeWork Board for the first time. Chapter 8 explores frequency and sound, including microcontrolled music. It is available online from www.parallax.com.

"Understanding Signals", Student Guide, Version 1.0, Parallax, Inc., 2003

Written by Doug Pientak, formerly of Optimum Designs, Inc., this introduction to the basics of digital oscilloscopes features the Parallax USB Oscilloscope (formerly the OPTAscope 81M). The student learns about signals through building circuits that generate and manipulate different types of waveforms with the BASIC Stamp 2, and then measures and analyzes them with the Parallax USB Oscilloscope. The book and two software platforms for the Parallax USB Oscilloscope are available online from www.parallax.com.

Chapter 2: Data Logging

The theme of the Data Logging experiment is best answered by the question: What is data logging and why is it important when using sensors? During the activities in this experiment you will:

- Design a user interface by adding a pushbutton to your existing setup on the Board of Education or HomeWork Board, then implement single click, double click and long click to do different things
- Learn the basics of **`READ`** and **`WRITE`** with the BASIC Stamp's EEPROM
- Implement a "talking (Morse code) thermometer"

Constancy punctuated by change: that is one prevalent view of the real world. In order to understand and predict events, people often need to keep a record of variables that affect the action. In laboratories, factories, and field research, the data logger, or data acquisition system, or DAQ for short, is an essential tool. It is a machine that automatically takes readings and stores them at regular intervals of time (or on some other basis) into the memory of the machine for later retrieval.

Data is stored in a log file. The term comes from nautical history, where readings of position and depth soundings on a ship were regularly noted in the Captain's log-book. In fact, some data was collected by throwing a log (not the book!) off the bow of the boat and counting the time it took for the log to reach the stern of the boat. Then they could calculate speed.

These days, much logging is done by computers with sensors attached. Computers are well suited to data logging - they never get bored or tired, and they can work reliably and very rapidly if required. It can be difficult, boring, or downright impossible for a real human being to exist in the place and time where data needs to be collected. Data loggers are found out on buoys floating in the ocean, high on windy mountaintops, on spacecraft, in collars on grizzly bears, in the stomachs of whales, out in orchards and vineyards, and in innumerable industrial settings.

Another "buzz word" these days is SCADA, for Small Computer Aided Data Acquisition. That usually refers to something fancier, a network of sensors and computers, but the general idea is the same. Data loggers may even communicate to a central hub via TCP/IP connections to the Internet, or via long-distance radio links.

In this experiment you will learn important details about the EEPROM memory in the BASIC Stamp 2. This is in preparation for logging readings of temperature, light and water level in the experiments to come. Also, you will improve on your DS1620 thermometer from the previous experiment, and make it talk (in Morse code). And as a warm-up, you will work with one pushbutton and the piezo speaker, to make a user interface.

Everyone who has a computer understands what you mean by a mouse, and the actions of click, double-click, and click-and-hold. These actions are central to the modern computer's user interface. Have you ever wondered how a program implements those actions? How hard would it be to implement them on the BASIC Stamp? Well, it is not too hard at all, and we are going to do it, to enable one button on your Board to perform multiple tasks. There is not going to be room for multiple pushbuttons. One button, along with feedback from the piezo transducer, is going to have to do it all for our user interface when the Board is not docked to the PC.

Parts Required

The *Applied Sensors* experiments are progressive and build on the previous projects. Therefore, you'll be adding parts to your Board of Education or HomeWork Board. This experiment requires the following parts:

(1) Pushbutton
(1) 10 kΩ resistor
(1) 220 Ω resistor
(2) Jumper wires

Building the Circuit

In the *What's a Microcontroller?* Student Guide you learned how to use one or two buttons to make decisions, and to control light emitting diodes. In this experiment you will build on that project and on the previous *Applied Sensors* experiment. You already have an audio annunciator for output. Now, let's install a pushbutton for input.

√ Build the circuit as shown in the schematic (Figure 2-1) and wiring diagram (Figure 2-2).

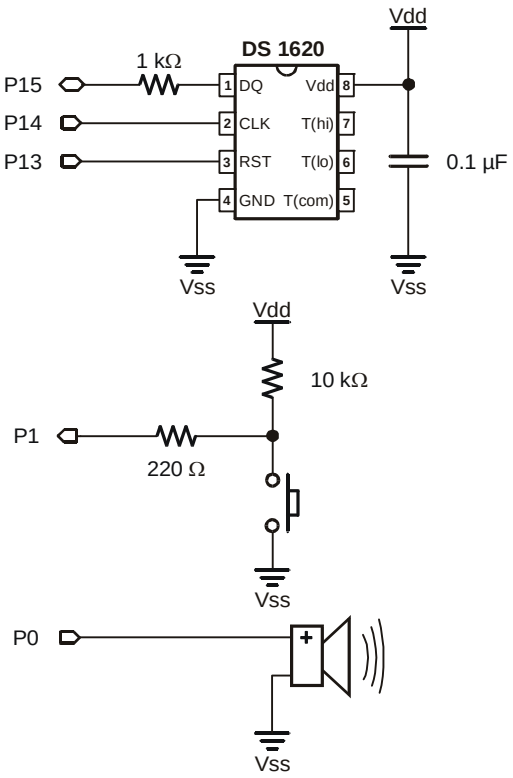


Figure 2-1
Data Logger Schematic

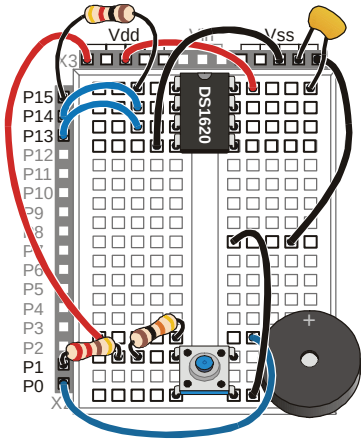


Figure 2-2
Data Logger Wiring Diagram

Install the pushbutton at the very edge of the breadboard.

Programming the Project

The wiring has a pushbutton connected to a pull-up resistor, and the junction between the resistor and the switch connected to P1 on the BASIC Stamp, through a protective resistor in series. When the pushbutton is not pressed, the voltage at the BASIC Stamp pin is 5 volts (= Vdd) through the pull-up resistor. But when the button is pressed, the voltage at the BASIC Stamp pin is low, zero volts (= Vss).

√ Enter the program TestPushbutton.bs2 into your BASIC Stamp Editor.

```
' Applied Sensors - TestPushbutton.bs2
' Test pushbutton wiring.
'{$STAMP BS2}
'{$PBASIC 2.5}

DO
  DEBUG BIN IN1
LOOP
```

√ Download the program to your BASIC Stamp.
√ Run it and observe the Debug Terminal as you push and release the pushbutton.

The program is going around and around the **`DO…LOOP`**, spewing out the level that it finds at the input. The variable is **`IN1`**. It is either high = 1, or low = 0. The reading should go to zero immediately when you press the pushbutton, and it should go to one when you release it. Yes? Go on to the next step. No? Is the problem in the program, the connection to the BASIC Stamp, or in the wiring of your pushbutton?

What's all this DEBUG stuff?

In these experiments, you will be seeing the **DEBUG** command very often, to put data on the computer screen in the Debug Terminal. Its name comes from the notion of debugging. You can put information on the screen that helps you see what is going on in your program. Moreover, you can ask the **DEBUG** command to send any messages or data you want to the Debug Terminal, it does not have to be especially for debugging.

The **DEBUG** command lets you display the data on screen in quite a few different ways, using modifiers and screen control commands. Previously we used commands like this to display temperature data:

```
DEBUG ? degC
```

That is a combination command that does 3 things: it prints the variable name and an equals sign; it prints the decimal value of the variable; and it moves the cursor down to a new line. The result looks something like this:

```
degC = 25
```

The current little program has a different form of the **DEBUG** command:

```
DEBUG BIN IN1
```

This prints the binary value of the variable **IN1**. Yes, **IN1** is a variable, the state of input pin P1, either low or high, 0 or 1. This form of the **DEBUG** command prints only the "0" or the "1", and not the name " **IN1** ", nor the "**=**", nor any spaces between the 1s and 0s, nor does it move down to a new line (until it hits the full width of the screen). The result looks something like this:

11111111000000000001111111111111110000000

00000011111111111111111111000000011111...

As we come to new forms, we will describe them briefly, and refer you to the *BASIC Stamp Manual.* You may always look up commands in the BASIC Stamp Editor's Help file.

Now, let's make the pushbutton produce a continuous sound while it is pressed down.

√ Enter the program Buzzer.bs2.

```
' Applied Sensors - Buzzer.bs2
' Read pushbutton to control the piezospeaker.
'{$STAMP BS2}
'{$PBASIC 2.5}

DO                                    ' Repeat forever.
  IF IN1=0 THEN FREQOUT 0, 8, 2500    ' Buzz if button is pressed.
LOOP
```

√ Run it, and press and release the pushbutton.

You should hear a sound that may remind you of a cricket chirping. What is going on? If the button is up, nothing happens, because the **IF** statement sees a 0 in the input pin, so **FREQOUT** is not executed and simply sends the program back to the top of the **DO…LOOP** loop. If the button is down, the **IF** statement sees a zero on the input pin. The program then executes the **FREQOUT** statement. Then it loops back to the top. So long as the button stays down, the loop with the **FREQOUT** is executed over and over.

Recall that the argument 8 in the **FREQOUT** command is the duration of the tone in milliseconds. The tone is 2500 Hertz, so in 8 milliseconds, there are 20 cycles of the tone (0.008 seconds * 2500 cycles per second = 20 cycles). Then the tone stops briefly, while the program goes back up to the top and tests the state of the P1 pin again. The tone is not produced during that time, because the BASIC Stamp can only execute one command at a time (This is an important fact to remember!). If the pin is still low, though, it soon is back to the **FREQOUT** command.

So the sound looks something like this: ||||||||.||||||||.||||||||.||||||||.||||||||. What you hear is not the pure 2500 Hertz tone, but a tone with repeated brief interruptions. These add the low sub-tone you hear in the sound, at about 110 Hertz (about 9 milliseconds for the loop, 1/.009 = 111). This is indeed kind of like a cricket's stridulation (song), which is produced when the insect rubs a file on one of its forewings against a ridge on the other forewing, producing a high pitch, with brief pauses in the back and forth motion of the wings.

Let's try some variations on the above program:

√ Modify Buzzer.bs2. by changing the **FREQUOT** command's ***Duration*** argument to 1.
√ Download the modified program.
√ Press and release the pushbutton, observing the result.
√ Repeat this modification for ***Duration*** values of 4, 50, 500 and 5000.

After you have run the program each time and listened, can you explain why each variation sounds as it does? At the long interval, 500 and especially 5000, note that the tone can go on long after the pushbutton is released. Why is that? Why doesn't the tone stop immediately when you release the pushbutton?

√ Modify Buzzer.bs2 once again by changing the **FREQUOT** command's ***Duration*** argument back to 8.

√ Tap on the pushbutton to send the number 50, or the signal "SOS" in Morse code. (dit dit dit dit dit= "5" and dah dah dah dah dah = "0", dit dit dit = "S", dah dah dah = "O").

It is already a useful program - a Morse code keyer! Refer back to Table 1-1 if you want to send other numeric messages.

√ Try inserting a **PAUSE 6** command on the line after the **IF** command.

That gives a ||||||||.....||||||||.....||||||||.....|||||||| pattern that may seem even more cricket-like. Crickets, in addition to their "output transducer" (the wings), also have an "input transducer" (an ear). It is a membrane located on their front legs! Crickets are very sensitive to repeating patterns and pulses of sounds. It is their "Morse code." Their songs are part of their courtship and male rivalry. Entomologists have studied insect stridulations by reproducing sounds on speakers, and watching what arguments of the sound evoke what behaviors from the crickets.

Sometimes you don't want an action to keep going all the time the button is down. You want it to happen once and only once each time the button is pressed.

√ Modify the program so that is reads as SingleClickDown.bs2.

```
' Applied Sensors - SingleClickDown.bs2
' Single click on pushbutton, action on button down.
'{$STAMP BS2}
'{$PBASIC 2.5}

DO                              ' Repeat forever.

  DO                            ' Do nothing.
  LOOP UNTIL (IN1=0)            ' until button is pressed.

  FREQOUT 0, 100, 2500          ' Buzz if button is pressed.

  DO                            ' Do nothing
  LOOP UNTIL (IN1=1)            ' until button is released.

LOOP
```

As in the previous program, nothing happens until the button is pressed down. Then the tone plays for 100 milliseconds. Then there is a second holding loop, where the program stays looping until the pushbutton is released. This concept was introduced in the *What's*

a Microcontroller? Student Guide. When that occurs the program goes back up to the top, ready for the button to be pressed again. One press, one action.

That is fine, but think about how a mouse click usually works. Most mouse clicks do not perform their action until you release the mouse button. That's easy. Move the **FREQOUT** down after the second "Do nothing" loop:

```
' Applied Sensors - SingleClickUp.bs2
' Single click on pushbutton, action on button up.
'{$STAMP BS2}
'{$PBASIC 2.5}

DO                                ' Repeat forever.

  DO
  LOOP UNTIL (IN1=0)              ' Loop here until button is pressed.

  DO                              ' Do nothing.
  LOOP UNTIL (IN1=1)              ' until button is released.

  FREQOUT 0, 50, 1900             ' Buzz when button is released
  FREQOUT 0, 100, 3800            ' and while we're at it, a better sound!

LOOP
```

Now a rising note should occur when the button is released. Logical, right? Be sure you understand totally how this works.

Now let's make the pushbutton take one action if you click it, and a different action if you hold it down for a long time. This is similar to the action of some computer menus that only appear if you hold the mouse button down for a longer period of time. Or you may have seen this in a car radio, where you press a preset button briefly to select a station, but you hold the button down for a longer time (until you hear a beep), to program a station you want into the preset memory. Appliances from wristwatches to VCRs, and yes, instruments sold in catalogs, all use tricks like this to get to the configuration menus.

Such a program needs a variable to keep track of the time you hold down the button. Try this:

- √ Enter and download the program LongClick.bs2.

```
' Applied Sensors - LongClick.bs2
' Single click on pushbutton, actions on button up or long click.
'{$STAMP BS2}
'{$PBASIC 2.5}

n       VAR    Word               ' Variable to keep  the time.

DO                                 ' Main loop.

  DO                               ' Do nothing
  LOOP UNTIL (IN1=0)               ' until button is pressed.

  n = 0                            ' Variable initialization.
  DO                               ' Loop to track pressing time.
    n = n + 1                      ' Increment counter.
  LOOP UNTIL (IN1=1 OR n>500)      ' Conditions to stop the loop.

  IF (n>=500) THEN
    FREQOUT 0, 5, 3800, 2533       ' Sound for long click.

    DO                             ' Do nothing.
    LOOP UNTIL (IN1=1)             ' Until button is released.

  ELSE
    FREQOUT 0, 50, 1900            ' Buzz twice when button is released
    FREQOUT 0, 100, 3800           ' after a standard click.
  ENDIF

LOOP
```

√ Quickly press and release the button, noting the effect.
√ Now press and hold the button, noting a different effect.

How does this work? The program initializes the variable **n** after you press the button. While the button is down, the program goes around and around incrementing the value of **n** in the loop. The statement:

```
LOOP UNTIL (IN1=1 OR n>500)
```

…keeps the loop going repeatedly so long as the pushbutton remains down or the counter variable remains smaller than or equal to 500. Each time around the loop, the timer variable **n** increases by one. It is a race to see which happens first. Do you release the button first, or does the timer reach 500 first? If the button is released first, well, that is just a single click, as above. The program plays the tones and goes back up to the top to await another button action. But if the timer **n** reaches 500 before you release the button, the program executes the **`FREQOUT 0, 5, 3800, 2533`** statement. There it plays one

short chirp, to let you know that you've gotten there, and then waits for you to release the button. And then it goes back to the top.

Where does the magic number 500 come from? The simple answer is "trial and error." The programmer (you!) tries different numbers until it feels right. Approximately how long (in milliseconds) do you have to hold the button down before it is identified as a long click? Try experimenting - substitute different values in place of 500.

Advanced Topic: Detecting a Double-Click with the BASIC Stamp

Can the BASIC Stamp detect a double click? Sure, it's not too hard. At the end of a single click, the program has to wait a fraction of a second to see if you are going to press the button again. If you do, then it is a double click. If you don't, it is a single click. The interval of time is so short that you don't really notice it. The actual interval is determined by trial and error, a "user preference." This too needs a timer variable. We will recycle the same timer variable, **n**, from the last program. Just for fun, we also modified the program so that it plays a constant chirp that continues until the button is released. Try this:

√ Enter and download the program Doubleclick.bs2.

```
' Applied Sensors - DoubleClick.bs2
' Double and long click on pushbutton.
'{$STAMP BS2}
'{$PBASIC 2.5}

n       VAR    Word              ' Variable to keep the time.

DO                               ' Main loop.

  DO                             ' Do nothing
  LOOP UNTIL (IN1=0)             ' until button is pressed.

  n = 0                          ' Variable initialization.
  DO                             ' Loop to track pressing time.
    n = n + 1                    ' Increment counter.
  LOOP UNTIL (IN1=1 OR n>500)    ' Conditions to stop the loop.

  ' First IF.
  IF (n>=500) THEN               ' Long click?
    FREQOUT 0, 5, 3800, 2533     ' Sound for long click.

    DO                           ' Do nothing
    LOOP UNTIL (IN1=1)           ' until button is released.
```

```
  ELSE                              ' Short click?
    n=0                             ' Initialization to check double click.
    DO                              ' Loop to track double click.
      n = n + 1                     ' Increment counter.
    LOOP UNTIL (IN1=0 OR n>150)     ' Conditions to stop the loop.

    ' Second IF.
    IF n>150 THEN                   ' Single click?
      FREQOUT 0, 50, 1900           ' Buzz twice when button is released
      FREQOUT 0, 100, 3800          ' after a single click.

    ELSE                            ' Double click?
      DO                            ' Wait until button is released.
      LOOP UNTIL (IN1=1)

      FREQOUT 0, 50, 3800           ' Play a unique falling sound
      FREQOUT 0, 50, 2533           ' indicating double click.
      FREQOUT 0, 50, 1900
    ENDIF                           ' End of the second IF.
  ENDIF                             ' End of the first IF.

LOOP
```

√ Give the pushbutton a single click. What do you hear?
√ Give the pushbutton a double click. What do you hear now?

If you press the pushbutton once and quickly release it, the program will generate the increasing pitch tones. Now there is another race between the button and the timer. This time the button is up to begin with. If you quickly press the pushbutton a second time before the timer reaches 150, that means you intend a double click. But if the timer reaches 150 first, that means you just want a single click (or you have slow fingers and need to reset the preference to a longer time, say 200).

√ Save the program DoubleClick.bs2, (or rename it if requested by your instructor) as we will be using "snippets" of what you learned here in programs to come.

What is a Snippet? You can "snip" an action from one program, and use it (with changes?) in another. Pieces of programs that perform specific actions are called snippets. Snippets often do not stand on their own as complete programs. Programmers often exchange ideas in the form of snippets.

At this point we are expanding on what you learned in the *What's a Microcontroller?* Student Guide. If you want, you could extend this logic to make a routine respond to a

triple click, like some word processing programs use to select an entire paragraph. We'll let that be a challenge!

Now, let's move on.

The Basics of Learning to READ and WRITE

In this series of experiments, we are going to program the BASIC Stamp to collect readings of temperature and other variables. We want to log them, that is, collect them at regular intervals of time and store them in a file, and read them out later for comparisons, charts and graphs. We'll take this a step at a time. First, it is important to understand how the memory on the BASIC Stamp is organized.

You know from *What's a Microcontroller?* that the memory available in the BASIC Stamp 2 is of two kinds, RAM and EEPROM. To understand the difference, it may help you to think about these kinds of memory if you know where they are located physically. Take a look at Figure 2-3, which shows a top view of the BASIC Stamp 2.

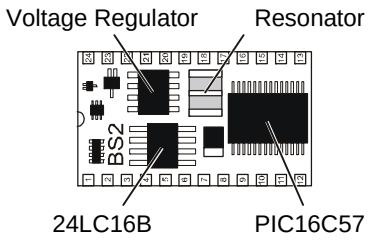


Figure 2-3
BASIC Stamp Memory

The PIC16C57 chip is the BASIC Stamp's RAM and central processor. The 24LC16B is the EEPROM, which holds your PBASIC program and data we will be storing.

Variables are created in the RAM (Random Access Memory). You store numbers in RAM with statements like this, using named variables:

```
x       VAR     Byte
x = 76
```

Variables are very versatile. They can be added and subtracted and used in lots of other kinds of arithmetic, and they can be arguments in all sorts of commands that are described in the *BASIC Stamp Manual*. It is very fast to manipulate data in RAM (~200 microseconds per operation), and RAM does not wear out with use. Trouble is, there isn't very much RAM available on the BASIC Stamp, only 26 bytes. It is not suitable for

storing lots of data. Also, the contents of RAM are lost when the BASIC Stamp loses power, or when the Reset button is pressed. RAM is not suitable for storing "valuable" data that you would want to survive when the power is disconnected.

Then there is EEPROM. A greater amount of EEPROM memory is available on the BASIC Stamp: 2048 bytes. Although part of the EEEPROM is used for your PBASIC program code, there will be some left over for data storage. One great advantage of EEPROM is that it is semi-permanent. The EEPROM memory retains its contents with or without power and through resets.

Two minor limitations of EEPROM are that it is relatively slow (~10 milliseconds to save a byte of data), and, it will wear out after something like 10,000,000 changes at one spot. To put this in perspective, if one certain location in EEPROM is reprogrammed over and over, once per second, it would take you about 116 days to get near the 10,000,000 mark. How many seconds are there in 116 days? On the other hand, at once per hour, it would take 1142 years to reach that same mark. (How many hours are there in 1142 years?) It is something to think about in planning. In *Applied Sensors* we may write to a single location a hundred times at most, nowhere near ten million.

There are three instructions the BASIC Stamp 2 uses for interacting with EEPROM: **DATA**, which stores initial values during a program download, **WRITE,** which stores values during program run time**, and READ**, which retrieves values. Once you **WRITE** data to the EEPROM, you must **READ** it into a variable again before using it in a calculation or as an argument in a command. The main reason we use EEPROM is to store larger quantities of data, if we won't have to change them too often, and to safely store data it will stay as long as we want them to.

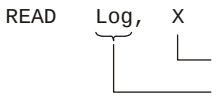
DATA, WRITE, and **READ** are most easily used to move byte at a time, as shown below and in the following example programs. However, PBASIC 2.5 allows you to use the **Word** modifier to handle word-sized variables, and also allows you to move multiple variables in one line of code. To learn more about the power of these commands, read about them in your BASIC Stamp Editor's Help file.

In PBASIC, the **DATA** directive reserves an area in the EEPROM, and gives it a name:

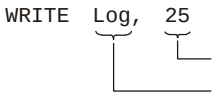
```
Log   DATA   7
```

The value 7 is loaded into EEPROM at address, **Log.**
Name for the address in EEPROM where the data is located.

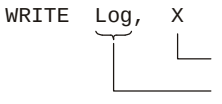
READ retrieves a byte from an address (in EEPROM) and copies its value to a variable (in RAM). The value of the byte in EEPROM is not affected by reading it.



WRITE may be used in a program to change the byte stored at an address in EEPROM.



or, with a variable,



Do not confuse the address, **Log** in this case, with the data that is stored there!

√ Enter and download EEPROMExample.bs2.

```
' Applied Sensors - EEPROMExample.bs2
' Distinction of constant, data and variable.
' {$STAMP BS2}
' {$PBASIC 2.5}

Dit        CON     70           ' Define a constant.

x          VAR     Byte         ' Define two variables.
y          VAR     Byte

Log        DATA    7            ' Reserve a byte in EEPROM, initially 7.
Worm       DATA    240          ' Reserve a byte in EEPROM, initially 240.

READ Log, x                     ' Read data from EEPROM into the variables.
READ Worm, y

DEBUG ? Dit, ? Log,             ' Show all quantities.
      ? x, ? Worm, ? y
```

The value of **Dit** is 70, an ordinary constant. The name **Dit** refers to the value itself. The values of **Log** and **Worm** are constants too, but they have values of 0 and 1, not 7 and 240. The names **Log** and **Worm** refer indirectly to the data. To read the 7 and the 240, there are two **READ** commands in the program. One **READ** gets the 7 from EEPROM address **Log = 0** and puts it in the RAM variable **x**, and the second **READ** gets the 240 from EEPROM address **Worm = 1** and puts it in RAM variable **y**. The labels **Log** and **Worm** have the addresses 0 and 1 because PBASIC assigns addresses for data statements starting at 0.

Now, let's try a modified version of EEPROM.bs2 that has four more lines at the end.

√ Enter and download the program WriteEEPROM.bs2.

```
' Applied Sensors - WriteEEPROM.bs2
' Writing a variable.
' {$STAMP BS2}
' {$PBASIC 2.5}

Dit        CON     70           ' Define a constant.

x          VAR     Byte         ' Define two variables.
y          VAR     Byte

Log        DATA    7            ' Reserve a byte in EEPROM, initially 7.
Worm       DATA    240          ' Reserve a byte in EEPROM, initially 240.

READ Log, x                     ' Read data from EEPROM into the variables.
READ Worm, y

DEBUG ? Dit, ? Log,             ' Show all quantities.
      ? x, ? Worm, ? y

x = x + 1                       ' Make a new value for x.
y = y / 2                       ' Make a new value for y.

WRITE Log, x                    ' Change the value stored at Log.
WRITE Worm, y                   ' Change the value stored at Worm.
```

√ Press the Reset button on your Board a couple of times with the Debug Terminal active. You should see the values of **x** increase by 1 each time, and the value of **y** halved each time.

√ Disconnect the power momentarily, and reconnect it. The first value you see on the Debug Terminal should be the next one in the series, showing that the EEPROM retains its data when the power is off.

What happened to the 7 and the 240 that were there when you first ran the program? They are gone. The **WRITE** statement changed those values. The only way to restore the initial condition is to download the program again from your computer.

- √ Re-download the program WriteEEPROM.bs2 from the BASIC Stamp Editor.
- √ Verify that the initial variables stored by the DATA directives have returned.

The EEPROM is often used to store settings and calibration constants that may need to be changed occasionally. It might be a argument that tells how hot the temperature has to be before turning on a fan, or how many seconds have to pass before recording data in a log file. Here is a fun demo program that plays a musical scale when you single-click the button. How many notes it plays depends on an argument that is stored in the EEPROM.

- √ Enter and download the program SaveSetting.bs2.

```
' Applied Sensors - SaveSetting.bs2
' Saving a setting in EEPROM.
' {$STAMP BS2}
' {$PBASIC 2.5}

Dit        CON     70           ' Define a constant.

many       VAR     Word         ' RAM variable for number of sounds.
n          VAR     Word         ' Multipurpose variable.
tone       VAR     Word         ' Frequency of the sound.

How        DATA    1            ' Initial number of sounds.

DO                              ' Main loop.

  DO
  LOOP UNTIL (IN1=0)            ' Loop here until button is pressed.

  n = 500                       ' Variable initialization.
  DO                            ' Loop to track pressing time.
    n = n - 1                   ' Decrement the counter.
  LOOP UNTIL (IN1=1 OR n=0)     ' Conditions to stop the loop.

  IF (n=0) THEN                 ' Code for long click.
    DO                          ' Repeat these actions.
      FREQOUT 0, 2, 3800        ' Short tick.
      PAUSE 400                 ' Short delay (time for response).
      n = n + 1                 ' Increment counter.
    LOOP UNTIL (IN1=1)          ' Until button is released.

    WRITE How, n                ' Store the new argument.
```

```
  ELSE                          ' Code for short click.
    tone = 4519                 ' Play tones, this is the first tone.
    READ How, many              ' Get how many to play from EEPROM

    FOR n=1 TO many             ' and play them.
      FREQOUT 0, Dit, tone      ' Sound, duration Dit, frequency tone.
      PAUSE Dit                 ' Brief silence.
      tone = tone ** 61858      ' Next note of chromatic scale.
    NEXT                        ' End of FOR-NEXT.

  ENDIF

LOOP
```

- √ Press and hold the pushbutton for a few moments, then release it.
- √ Give the pushbutton a quick press-and release click.
- √ Repeat several times, holding the button down for varying intervals.

What happened each time? Try to figure out how it works in detail. It consists of snippets from the foregoing button and memory routines. (The mathematical formula, tone = tone**61858, generates the chromatic scale, but you don't have to understand that here.) Do understand the role of **READ** and **WRITE**. There is one **READ** command to fetch the number of notes to play, and one **WRITE** command to store the new number selected by the user.

To test your understanding, modify the program SaveSetting.bs2 as follows:

- √ Add a **DATA** directive with a label of **Dur** and make 70 milliseconds its initial value.
- √ Change **Dit** from a constant to a byte variable named **dit** (because variables are formatted to start with a lowercase letter).
- √ At the outset of the program, use **READ** to move the value from **Dur** into the variable **dit**. At this point, the program should run, just as it does now.
- √ At the end of the long click routine, before the **ELSE** statement, have it wait for you to press and release the button a second time.
- √ During this second time the button is down, have it increment the value of **n** each time around a loop.
- √ When the button is released, write the value of **n** into the address **Dur**.
- √ Verify that the program runs, and that it allows you to change both the number of notes to be played, and the duration of the notes.

Talking Thermometer, Morse Code Revisited

We will begin this next activity by re-using the program DS1620.bs2 that you saved in Chapter 1. This program reads the temperature from the DS1620 chip and displays it on the Debug Terminal.

- √ Open the program DS1620.bs2 (or whatever you named it).
- √ Download and run it to make sure that it still works.

You never can be sure, maybe you accidentally bumped a wire on your breadboard, or maybe someone was fooling around with your program on disk. It is a wise practice to start each step of building a complex system at a point where you know everything is working.

As it stands, the program displays the temperature on the Debug Terminal once per second. Let's modify it to make the piezo transducer send the temperature using Morse code. The Morse code in the first experiment of *Applied Sensors* was an introduction - it only sends the number 50. We need a subroutine that can sound out any arbitrary two-digit number we throw at it. We'll also change the program so that your new pushbutton will initiate the temperature reading. Starting with the program from Chapter 1, we developed the next program.

- √ Save your program under the new name DS1620MorseCode.bs2.
- √ Enter the new name and description in the Title section of your program in the BASIC Stamp Editor.
- √ Continue entering the new code, being careful to note that elements of the old program have been separated into different sections, and following the hints in the new comments.

```
' -----[ Title ]-------------------------------------------------------------
' Applied Sensors - DS1620MorseCode.bs2
' Talking thermometer, using Morse code.
'{$STAMP BS2}
'{$PBASIC 2.5}

' -----[ Constants ]-----------------------------------------------------
Dit       CON        70            ' Milliseconds for Morse dit.
Dit2      CON        2*Dit         ' Constants related to Dit.
Dah       CON        3*Dit         ' Ditto.

' -----[ Declarations ]--------------------------------------------------
mc        VAR        Byte          ' Temporary for Morse pattern.
```

```
xm        VAR        Byte          ' Morse input variable.
j         VAR        Nib           ' Index for digits to send.
i         VAR        Nib           ' Index for dits and dahs.

x         VAR        Byte          ' General purpose variable, byte.
degC      VAR        Byte          ' Variable to hold degrees Celsius.

' -----[ Initializations ]----------------------------------------------
' Note: DS1620 has been preprogrammed for mode 2.
' If not, uncomment the instructions on the next line on the first RUN
' HIGH 13: SHIFTOUT 15,14,[12,2]: LOW 13

OUTS=%0000000000000000             ' Define the initial state of all pins
     'FEDCBA9876543210
DIRS=%1111111111111101             ' as low outputs,
                   '^--------------- except P1, an input for pushbutton.

FREQOUT 0, 20, 3800                ' Beep to signal that it is running.

HIGH 13                            ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [238]   ' Send the "start conversions" command.
LOW 13                             ' Do the command.

' -----[ Main Routine ]-------------------------------------------------
DO                                 ' Start of the main loop.

  DO
  LOOP UNTIL (IN1=0)               ' Loop here until button is pressed

  DO
  LOOP UNTIL (IN1=1)               ' Loop here until button is released

  HIGH 13                          ' Select the DS1620.
  SHIFTOUT 15, 14, LSBFIRST, [170] ' Send the "get data" command.
  SHIFTIN 15, 14, LSBPRE, [x]      ' Get the data.
  LOW 13                           ' End the command.

  degC = x / 2                     ' Convert the data to degrees C.
  DEBUG ? degC                     ' Show the result on the PC screen.
  xm = degC                        ' Morse routine expects data in xm.
  GOSUB Morse                      ' To the subroutine.

LOOP                               ' Back to wait for button again.

' -----[ Subroutines ]--------------------------------------------------
Morse:                             ' Emits byte xm as Morse code.
  FOR j=1 TO 0                     ' Send 2 digitS, Tens then ones.
    mc = xm DIG j                  ' Pick off the (j+1)th digit.
    mc = %11110000011111 >> mc     ' Set up pattern for Morse code.
    FOR i=4 TO 0                   ' 5 dits and dahs.
      ' Send pattern from bits of mc.
```

```
      FREQOUT 0, Dit2*mc.BIT0(i) + Dit, 1900

      PAUSE Dit                     ' Short silence.
    NEXT                            ' Next I, dit or dah of five.

    PAUSE Dah                       ' Interdigit silence.
  NEXT                              ' Next j, digit of two.
RETURN                              ' Back to main program.
```

- √ Double-check your work against the program in the text above.
- √ Run the program and try it by clicking the button.
- √ Heat up the DS1620 with your fingertip or a lamp, and then cool it off by fanning it.
- √ Listen to the Morse code as you make the temperatures go up and down.
- √ Watch the numbers display simultaneously in the Debug Terminal.
- √ Save the program DS1620MorseCode.bs2 on a disk, or under the name and directory given by your instructor.

If you are not a ham or Navy radio operator, you may need a little practice to hear the numbers of the Morse code, but it shouldn't take long. This talking thermometer is a useful instrument already. A visually impaired person could use it. Or, how about a biologist doing research on bats in a dark cave? (You would need to listen on an earphone—bats are very sensitive to high-frequency sound!) Can you think of other situations where this device might be useful?

Now let's look at the program step by step. Several variables and constants are defined at the top of the program. Some of these you will recognize from Chapter 1, where they appeared in the routine to send the number 50 as Morse code. There is the basic length of the **`Dit`** in milliseconds, and the **`Dah`**, which is defined as three times the length of the **`Dit`**, and a new one, **`Dit2`**, which is defined as twice the length of the **`Dit`**. There are a couple of other variables, too, **`xm`** and **`mc`**, that we'll talk about in connection with the **`Morse`** subroutine below.

P1 is now an input, for the pushbutton. P1 is set to input by making its bit in **`DIRS`** equal to zero. The following statements fix the input and output state of all 16 pins of the BASIC Stamp.

```
OUTS=%0000000000000000    ' Define initial state of all pins
     'FEDCBA9876543210
DIRS=%1111111111111101    ' as low outputs.
                  ' ^------ this is now an input for the button
```

Note the single change from the original program. If we do not set that bit in **DIRS** equal to zero, then the program cannot read the pushbutton. If you don't believe it, try it and see what happens. You may wonder about the programs in the first part of this experiment, where we were reading the state of the pushbutton very well with neither a **DIRS** nor an **OUTS** command. The reason is that the BASIC Stamp always starts up with all its pins as inputs. As a matter of good programming, we are turning them all into outputs, except the ones we truly need to be inputs. When we make a pin like P1 into an input, it doesn't matter what the state of the corresponding **OUTS** bit is. The **OUTS** bit has no effect when the pin is defined as an input.

The central idea of the **Morse** subroutine is held in the binary pattern, %11110000011111. The % sign marks it as a binary number. This is the pattern of zeros and ones as they are actually stored in binary brain of the BASIC Stamp. This binary number does have a standard numerical value (it happens to be 15391), but the numerical value is not important here. Quite often in computer science, you have to think of computer data as something other than a standard numerical value. Think of this as a pattern on an audio tape. If you put a playback head (by analogy) at the far left and play back 5 bits moving to the right, you come up with 11110. This is going to translate in Morse code to dah dah dah dah dit, a nine. (It is not a binary number nine, which would be 1001 - instead, it is a pattern for Morse code number 9 - there are many ways to represent numbers!) Take a look at the way the different digits overlap in the binary pattern in Figure 2-4. Depending on where you start on the "tape" different code patterns result, in fact, the total pattern is arranged to give the code patterns for the Morse code numerals numbers in order. It's a trick.

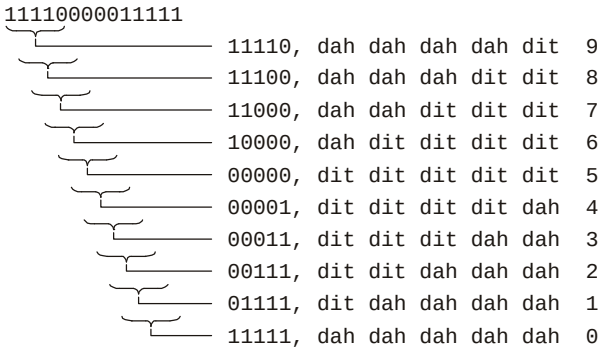


Figure 2-4
Morse Code Binary Pattern

Now let's take a closer look at how the Morse code elements fit into the program. First, you have to recognize that there is a subroutine that starts with the label **Morse**, and ends with the **RETURN** command. By writing the **Morse** section as a subroutine, we will be able to use it over again at different points in our progressive program, as it develops.

The **Morse** subroutine is called from within the Main Routine section. The Main Routine section begins by monitoring for a pushbutton press. A pushbutton release is followed by **SHIFTOUT** and **SHIFTIN** instructions which acquire the temperature reading from the DS1620, and put it into the variable **x**. Next, **x** is divided by 2 to give us the variable **degC**, and the **DEBUG** command sends this value to the Debug Terminal so we can read the temperature in degrees Celsius on the PC screen. Then, the variable **xm** is assigned the value of **degC.** This is necessary because **xm** is the variable that will be recognized by the Morse subroutine code we designed earlier. **GOSUB Morse** sends the program to that subroutine, where **xm** is converted to Morse code and played on the piezospeaker with a **FREQOUT** instruction. The **RETURN** command at the end of this subroutine sends the program back to the final **LOOP** in the Main Routine section, causing the program to go back to **DO** at the top, where it will begin monitoring for a another pushbutton press.

Let's look closer at the **Morse** subroutine. The variable **xm** is the one that will be sounded out as Morse code. In the **Morse** subroutine itself there are two **FOR…NEXT** loops, one inside the other. The outside loop has an index **j:**

```
Morse:
  FOR j=1 TO 0
    mc = xm DIG j
    mc = %11110000011111 >> mc
```

```
    'Inner FOR…NEXT loop here
  NEXT
RETURN
```

What is an Index, and what is a Pointer? An index is a variable that steps through a sequence of values. For example, **j** in the for-next loop steps through the values of 1 and 0. A pointer is a variable that specifies where in memory, or where in some ordered set, to retrieve information. For example, the variable **j** is both an index and a pointer. It points to a digit in the variable **xm.** The index **i** in this same program is a pointer to the bits (binary digits) of the variable **mc**. In experiments to come, we will use indices and pointers to refer to the data in the EEPROM log, as in, 1st reading, 2nd reading, and so on.

When the program first arrives at the **Morse** subroutine, it sets **j** equal to 1, and then continues with **j = 1** all the way through the outer loop (including everything in the inner **FOR…NEXT** loop). The keyword **NEXT** in the outer loop, is the trigger that makes the program jumps back up to the corresponding outer **FOR**, sets **j = 0**, and executes all the way through again, back to the outer **NEXT**. Note that the BASIC Stamp knows how to count backwards! After **j** has taken on the values 1 and 0, that's it, the loop ends, and the program returns to the **Main** routine.

There are two math statements in this outer **FOR…NEXT** loop. The first one is:

```
mc = xm DIG j
```

This **DIG** is a PBASIC operator, the way "plus" and "divided by" are operators. Short for digit, **DIG** returns a digit from a given position within a larger number. In our program **DIG** sits between two numbers, **xm** and **j**, and returns the (j+1)th digit of **xm,** reading from right to left. It is easiest to illustrate with a specific example: Suppose the value of **xm = 25**. On the first time through the loop, the value of **j** is 1, which gives us **j**+1 = 2. Therefore, (**mc = 25 DIG 1**) will return the 2nd digit from the right, in the tens column, which is a 2. So the instruction (**mc = 25 DIG 1**) will produce (**mc = 2**). On the second time through the loop, the result of (**mc = 25 DIG 0**) will be (**mc = 5)**, because 5 is the 1st digit from the right, in the ones column. The logic of this can be extended to larger numbers, for example, **j = 3** would point to the thousands digit. However, in this program we will only need 2 digits. Check out your Basic Stamp Editor's Help file for the full description of **DIG**.

Now we have a number between 0 and 9 inclusive in the variable **mc**. The next statement sets up the pattern for the Morse code.

```
mc = %11110000011111 >> mc
```

The symbol **>>** is another operator that goes between two numbers. The constant, **%11110000011111**, is the binary pattern we were talking about above. The **>>** operator is one that operates specifically on binary patterns. It is called a shift operator. (Shifts are very important in computer science.) It shifts the binary pattern to the right a certain number of places (**mc** places) and drops that same number of bits off the right end. Let's continue using the example from above, that had the initial value mc = 25. Continuing with first time through the loop, the digit is 2 when the program arrives at this command:

```
  mc =  11110000011111 >> 2  ' Shifting bits instruction
' mc =  111100000111          pattern shifted two to the right
'                  \11        two bits dropped
'              ^^^^^-----> 5 bits are the Morse pattern for "2"
```

And the second time through the loop, the digit is 5:

```
  mc =  11110000011111 >> 5  ' Shifting bits instruction
' mc =  111100000             pattern shifted 5 to the right
'                \11111       five bits dropped
'           ^^^^^-----> 5 bits are the Morse pattern for "5"
```

What has happened is that the Morse code pattern has ended up in bits 4 to 0 of the variable **mc**. In the example, 00111 represents 2 in Morse code, and 00000 represents 5. Earlier we talked about moving a "playback head" over the "tape"; here we have moved the "tape" over the "playback head," ready to play back the five bits on the right.

Now the Morse code pattern is in position, and we come to the inner **FOR…NEXT** loop:

```
FOR i=4 TO 0
  FREQOUT 0, Dit2*mc.BIT0(i)+Dit, 1900
  PAUSE Dit
NEXT
PAUSE Dah
```

The index here is **i**, and it runs through 5 values, counting backwards from 4 to zero. The **FREQOUT** command plays a dit or dah for each time around the inner loop. Between each sound, there is a short pause equal in width to a **Dit**. After the 5 dits and dahs of the

tens-column digit are played, there is a longer pause, equal in width to a **Dah.** Then, the program loops back to get the ones-column digit from the **DIG** operation, then plays its five-bit Morse code equivalent in the same way.

The **FREQOUT** command is familiar, except here the ***Duration*** argument is neither a constant nor a simple variable. It is an expression. PBASIC lets you do that. The expression is:

```
Dit2*mc.BIT0(i) + Dit
'     ^^^^^^^^^^-------this has a value of either 0 or 1.
```

Let's start out by stating that **mc.BIT0(i)** is a variable that has a value of either zero or one. So the statement reduces with simple multiplication and addition to either:

```
Dit2 * 0 + Dit  ==>  Dit
```

…or

```
Dit2 * 1 + Dit  ==>  3*Dit ==> Dah
```

The **FREQOUT** command plays a dit or a dah, depending on the value of the mystery variable.

So what exactly is **mc.BIT0(i)**? One powerful feature of PBASIC is that it allows you easy access to individual bits in that byte. The byte, **mc**, has 8 bits. The notation, **mc.BIT0** is called a modifier of the byte variable **mc**. It is really just name for the least significant bit of that byte. The second bit is **mc.BIT1**, and so on until **mc.BIT4**, is the 5th bit. It is simply a way of naming the bits, a syntax that is built into the PBASIC language.

There is still another way to refer to those same bits, using a variable as a pointer to bits in the byte. This notation is **mc.BIT0(i)**. For example, **mc.BIT0(4)** and **mc.BIT4** both refer to the same bit. Literally it means, "the fourth bit up from **mc.BIT0**." See the *BASIC Stamp Manual*, for more explanation. Figure 2-5 illustrates the way it works:

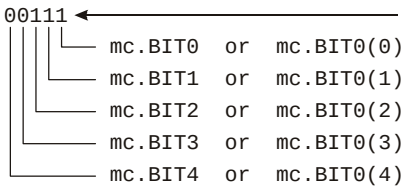


Figure 2-5
Five lower bits of the byte variable **mc.**

The variable **i** is the pointer. The power of this indirect, or array, method of naming, is that the program loop (for **i** = 4 to 0) can step through the bits of the byte variable, **mc**, one by one, and pick off the binary 0 or 1 values of the individual bits. Those are the bits that need to be sounded out as 0 → dit and 1→ dah. Here is another way we could have played the five dits and dahs, without using a **FOR…NEXT** loop:

```
FREQOUT 0, Dit2*mc.BIT0+Dit, 1900   '  first bit
PAUSE Dit                           '  short silence
FREQOUT 0, Dit2*mc.BIT1+Dit, 1900   '  second bit
PAUSE Dit                           '  short silence
FREQOUT 0, Dit2*mc.BIT2+Dit, 1900   '  third bit
PAUSE Dit                           '  short silence
FREQOUT 0, Dit2*mc.BIT3+Dit, 1900   '  fourth bit
PAUSE Dit                           '  short silence
FREQOUT 0, Dit2*mc.BIT4+Dit, 1900   '  fifth bit
PAUSE Dit                           '  short silence
```

You see, this method refers directly to each bit, one at a time. But it comes out much shorter and more elegant using the **FOR…NEXT** loop and the index as a pointer to the bits.

Whew! That was a lot of explanation for a short stretch of program. But it contains some advanced ideas: how to interpret a number as a pattern, using an index and a pointer, and how to extract decimal digits. We covered the **DIG** and shift **(>>)** operators, how to use an expression as an argument, and how to use array modifiers of PBASIC variables. These are the stuff of programming a microcontroller!

Challenge!

1. Hook up an LED to P5, so that **`HIGH 5`** will turn it on. Write a program to turn the led ON when you click the button once, and OFF when you click the button again. (Push on, push off action). Hint: although there are several ways to do this, the **`TOGGLE`** command may help. See the *BASIC Stamp Manual* for more information.

2. Write a program for your BASIC Stamp 2 that prints "working" on the Debug Terminal, and plays a sound, once each time you click a pushbutton. Hint: print a message on the Debug Terminal using commands like **`DEBUG "working", CR`** (**`CR`** stands for "carriage return").

3. Then program it so that if you hold the button down while you press and release Reset on the Board of Education or the HomeWork Board, it will not go immediately to the "working" routine. Instead it will print "I await your instructions" on the Debug Terminal, play a different sound, and delay until you click the button again. (Think about printers, how some will print a "test page" if you hold down some button on the front panel as you turn the printer on.)

4. The program DS1620MorseCode.bs2 measures the temperature in degrees Celsius. Modify the program so that it displays degrees Fahrenheit, and plays it in Morse code.

5. Modify the **`Morse`** subroutine so that it will play three digits instead of just two, in case the Fahrenheit temperature goes above 99 degrees.

6. Advanced - after accomplishing #5 above, make it so that it will not play leading zeros, that is, if the reading is 76 degrees F, it will play "7","6", not "0","7","6").

7. Then try this: start with a byte of data, initially zero, stored in EEPROM. Each time the button is single-clicked, increment the byte in EEPROM by one (**`READ`**, increment, **`WRITE`**), and display the current value on the Debug Terminal. When the value reaches 7, print the words "access denied" on the Debug Terminal, and make a sound and blink the led on and off repeatedly. At that point, if you reset the BASIC Stamp or remove the power and then restore it, the "alarm" should come on right away (**`READ`** & decision at top of program.).

8. Advanced – After accomplishing #7 above, think of a way, using a special action on the button, like holding it down for a long time, to reset the value in EEPROM to zero. That will allow access so you can click the button 7 more times before the alarm re-sounds and locks you out.

9. Write a program that plays a unique sound if you triple click the pushbutton.

Chapter 3: Temperature Probe for Micro-Environments

The theme of *Applied Sensors* Chapter 3 is to connect a temperature probe mounted at the end of a cable that can reach out away from the Board of Education or the HomeWork Board, to monitor micro-environments. A well-calibrated sensor, with good resolution, will achieve the most accurate results. The specific activities of this experiment include:

- Using a capacitor with the **`RCTIME`** command
- Temperature measurement using an AD592 probe, with calibration in an ice bath
- Comparison of calibration with the DS1620 at room temperature
- Automatic calibration using the BASIC Stamp's EEPROM
- Talking (Morse code) temperature experiments featuring solar radiation, wet-bulb/dry bulb techniques and wind chill

Analog Temperature Sensor

It is often important to extend sensors out away from the recording instruments, so that measurements can be made in micro-environments. In the natural world there can be much variation from place to place and time to time. For example, the temperature of a leaf on a plant in the sun can be significantly different from the surrounding air temperature. The leaf forms its own micro-environment. And as plants grow, they create a unique micro-environment under their canopy. Often measurements are needed in several places at once and are fed back to one centrally located instrument. For example, an agricultural weather station will measure wind high above the ground, soil moisture underground, and other arguments at points in between. This means that sensors have to be mounted on cables to reach all those separate micro-environments.

In Chapter 1 you learned about the DS1620 smart temperature sensor. One nice thing about that sensor is that it returns readings directly as digital numbers. But one disadvantage it has is that it is a chip with 8 pins, hard to turn into a probe that can be used apart from a circuit board. In this experiment you will learn about a different kind of temperature sensor, the AD592. It is easy to incorporate into a probe mounted on a single pair of wires. This AD592 is an analog temperature sensor. Analog means that its signal is a continuous electrical value (microamps), proportional to temperature. Analog is the opposite of digital; digital readings are returned as a code of discrete values (zeros and ones). The AD592 is a "classic" technology that has been proven through many

years. Many of the signals you will encounter in science, or in many fields of engineering for that matter, are analog signals. Chips like the DS1620 have analog sensors at their heart, and engineers have worked very hard to give the DS1620 its digital smarts.

Analog temperature sensor: The microampere current produced by the AD592 is what is called an "analog" of temperature. Microamps is not the same as temperature, just as apples are not the same as oranges. Considering an analogy between a capacitor and a water tank, here the analogy is between the temperature and electrical current. This is the basis of "analog" sensors. Other temperature transducers may transduce temperature to voltage or resistance or capacitance. The signals on both sides of the analogy are of a continuous nature, with infinite gradations of strength from low to high. Analog is the opposite of digital, where the signals are transmitted as digital codes of zeros and ones.

Analog sensors require a different kind of interface to the BASIC Stamp 2. In this experiment you will learn about the **`RCTIME`** command. You may know about analog-to-digital converters, a kind of electronic chip that is dedicated to doing those conversions. The **`RCTIME`** command is a rudimentary analog-to-digital converter that is built into the BASIC Stamp. To introduce the **`RCTIME`** command, you will connect a capacitor to the BASIC Stamp input and review the properties of capacitors. Once you have the **`RCTIME`** command reading the temperature sensor, you will learn how to calibrate it, so that it will read the correct temperature, despite variations in the parts that are provided to build the circuit.

Once you have the probe on a cable, you can extend it out to measure temperature in micro-environments in your surroundings.

BASIC Stamp Pins, Capacitors, Review of the BASICs

You probably already understand that the 16 general purpose I/O pins on the BASIC Stamp can be in one of three distinct states at any given time. As shown in Figure 3-1, this is like three-position switch:

1. **`HIGH`**: The switch is connected to Vdd = +5 volts, as shown here, output is high. Current can flow out of the pin, sourcing from the +5 volt (Vdd) power supply.

2. **`INPUT`**: The switch is connected as an input. Zero current flows in or out of the pin. As an input, the internal BASIC Stamp circuitry monitors the voltage at the I/O pin. Levels less than 1.3 volts are interpreted as low, or 0, those greater than 1.3 volts are seen as high, or 1.

3. **LOW**: The switch is connected to Vss = 0 volts, output is low. Current can flow into the pin, sinking current to ground (Vss).

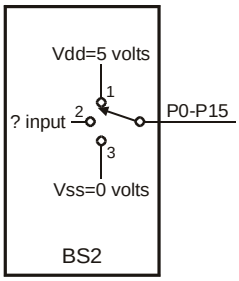


Figure 3-1
BASIC Stamp I/O Pins

Three positions exist in this switch: (1) Vdd +5V; (2) input to act as low or high; and (3) switch is connected to Vss +0V.

The simple commands **HIGH 5** or **LOW 5** or **INPUT 5** put the named I/O pin instantly into the correlating state. Many of the commands in the PBASIC language work by playing fancy games with the pins. For example, the **FREQOUT** command makes a sound by flipping the internal switch rapidly between the high and low output states. The **SHIFTIN** and **SHIFTOUT** commands work by coordinating the activity on several pins at once, some as outputs jumping from high to low, and others as inputs synchronized to the action. Here we will be introducing the **RCTIME** command, which works by switching a pin from an output to an input, and then timing how long it takes for the voltage at the I/O pin to cross the 1.3-volt threshold level.

The change in voltage is brought about by external circuitry, usually a resistor (R) and a capacitor (C). The important point we want to emphasize here is that practically zero current flows when the pin is an input—it just looks.

First, a brief review of capacitors. Please bear with us if you already understand how capacitors work. The analogy is a tank of water, with an inlet pipe and an outlet pipe. The tank stores water, analogous to how the capacitor stores electrical charge. Figure 3-2 and Figure 3-3 demonstrate this point.

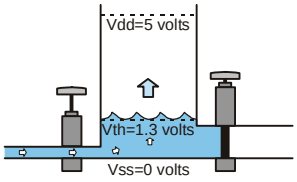


Figure 3-2
Analogy, capacitor charging

Water flows (amps) into the tank and the level (volts) rises. Higher inflow means higher rate of rise. The flow through the pipe can be limited by resistance (ohms) in the pipe, or by the water pressure at the other end of the pipe.

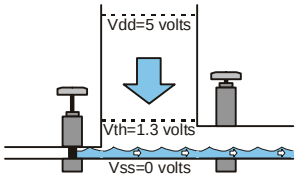


Figure 3-3
Analogy, capacitor discharging

Flow (amps) discharges from the tank and the level (volts) falls. The flow can be slow, a trickle, or fast, a deluge. If both inflow and outflow are zero, then the level stays constant. There can be unintentional flows, called leakage (amps).

Capacitors come in a wide range of sizes, measured in picofarads up to farads. This does not refer to physical size, but to the capacity to store charge, which depends on the material of which the capacitor is composed. Two capacitors of exactly the same physical size can have vastly different capacitances. We will be using values in this experiment that are 0.01 to 0.22 microfarads (µF).

Parts Required

Throughout this chapter, leave all parts from the previous experiments in place on the Board of Education or the HomeWork Board, as well as those you add in each activity. The following additional parts are required:

(1) AD592 temperature probe. See Appendix B if you would prefer to build your own.
(1) 0.1 µF monolithic capacitor
(2) 0.22 µF film capacitor
(2) 100 Ω resistor
(2) 100 kΩ
(1) Jumper wire
(2) 2-inch 4/40 stainless steel screws
(2) 16-inch pluggable jumper wires (1 red, 1 black)
(2) 2 4/40 nylon nuts
(1) Cup spanner

Building the Circuit

The first activity requires the pre-assembly of a conductivity sensor.

- √ From the Parts List on the preceding page, gather the 2-inch stainless steel screws, the 16-inch pluggable jumper wires, the nylon washers and the cup spanner.
- √ Begin by dropping the two screws through the two holes in the middle of the cup spanner.

- √ Fasten them from the other side with the nylon nuts, but do not tighten them yet.
- √ Wrap one bare end of each jumper wire around a screw between the cup spanner and the screw head.
- √ Now tighten the nylon washers to keep the wires in place and in firm contact with the screw heads.

The conductivity sensor will later be placed over the rim of a cup with the screw ends hanging down inside, and the other ends of the jumper wires will plug into the breadboard.

Simple Resistance Detector

- √ From the Parts List, gather the 0.1 μF capacitor, the 100 Ω resistor, and the conductivity sensor you just assembled.
- √ Build the circuit shown in the schematic (Figure 3-4) and the wiring diagram (Figure 3-5).
- √ Double check your circuits, using the hints next to the wiring diagram.

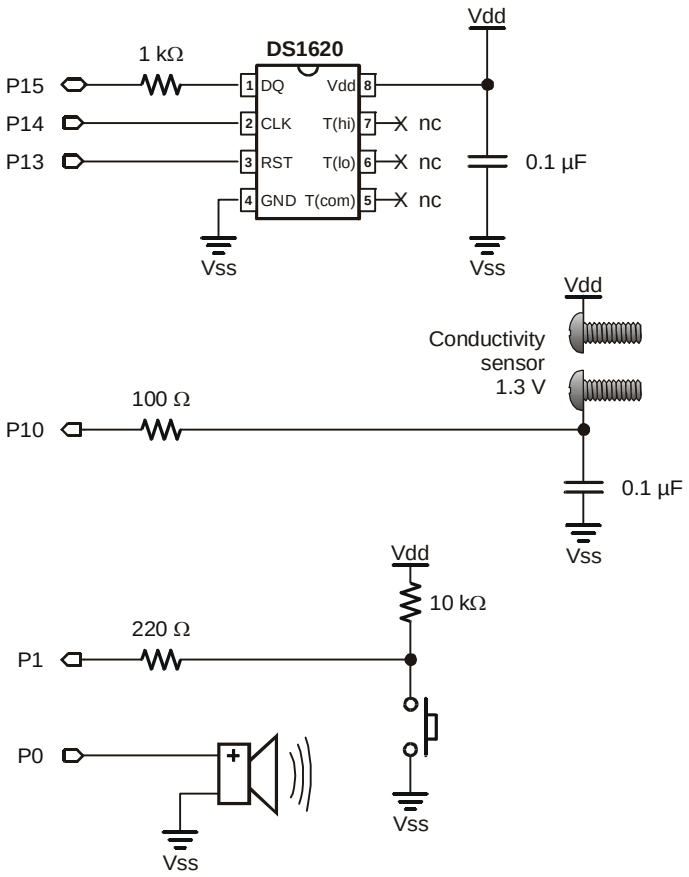


Figure 3-4
Simple Resistance Detector Schematic

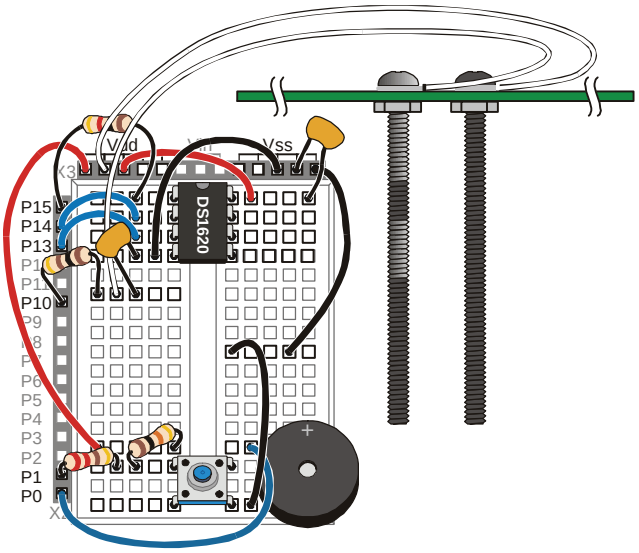


Figure 3-5
Simple Resistance Detector Wiring Diagram

- *The 100 Ω resistor connects P10 to the 0.1 µF capacitor and one lead of the conductivity sensor.*
- *The capacitor connects to Vss via the same row (node) as pin 4 of the DS1620.*
- *The other lead of the conductivity sensor connects directly to Vdd.*

√ Enter the program CapacitorDemo.bs2.
√ Let the conductivity sensor rest on a nonconductive surface, such as a pad of paper.
√ Run the program, letting the circuit run through its **DO…LOOP** for a minute or two while leaving the conductivity sensor undisturbed.

```
' Applied Sensors - CapcitorDemo.bs2
' Simple demo of a capacitor on a BS2 pin.
' {$STAMP BS2}
' {$PBASIC 2.5}

v     VAR     Bit         ' Bit-size variable for input state.

DO                        ' Beginning of the program.
  LOW 10                  ' Discharge the capacitor to 0 volts.
  FREQOUT 0, 5, 3500      ' Signal event.
  DEBUG CR                ' New line on screen.
  INPUT 10                ' Make the pin an input.

  DO                      ' Beginning of a loop.
    v = IN10              ' Read the input.
    DEBUG BIN v           ' Show it.
    PAUSE 99              ' 0.1 second pacing.
  LOOP UNTIL (v=1)        ' Repeat until the input is >1.3 V.

LOOP                      ' Back to the beginning of the program.
```

Do you hear beeps or see any 1s in the Debug Terminal? No? The first instruction in the program discharges the capacitor to zero volts. The capacitor discharges very fast, like a big pipe dumping water from the tank out onto the earth. Current from the PIC microcontroller on the BASIC Stamp can discharge the capacitor through the 100 ohm resistor in about 25 microseconds, which is much less than the ***Duration*** argument of the **`FREQOUT`** command. Then comes the **`INPUT 10`** instruction. P10 instantly becomes an input. Don't be surprised that the capacitor stays discharged, because there is no source of current to charge it. All those zeros on the screen mean that the capacitor stays discharged.

- √ Now touch the two leads of the conductivity sensor at the same time with your fingers as shown in Figure 3-6
- √ Experiment! Your finger short-circuits the capacitor. The result should depend on how sweaty or wet your fingers are (a lie detector?), and how hard you pinch. There are leakage paths through the moisture on your fingers, and through your skin and tissue.
- √ Try dipping the conductivity sensor in water.
- √ Touch it to wet paper, or
- √ Touch it to a heavy pencil line drawn on paper.
- √ Substitute the 100 kΩ resistor for the 100 Ω resistor, and test the same objects over again.

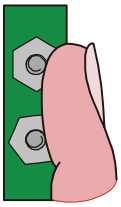


Figure 3-6
Short-Circuit

Touch the two screws that comprise the conductivity sensor. Your finger will short-circuit the capacitor.

We need to say a word here about safety. The voltage and current in this circuit are very small, five volts and a few microamperes. If you are ever unsure about a circuit, always err on the side of safety!

The input pin on the BASIC Stamp is acting as a "comparator." This is a technical term in electronics for this device that gives a yes or no answer, 1 or 0, to the question, "is the voltage level at P10 greater than or less than 1.3?" This 1.3-volt threshold is fixed by the PIC microcontroller in the BASIC Stamp 2, and there is nothing we can do to change it. Figure 3-7 shows how this works.

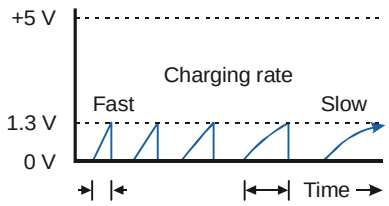


Figure 3-7
Capacitor Discharge

Over and over, the capacitor is discharged to zero volts, and then more or less rapidly charges back up to the 1.3 volt threshold. By varying the resistance of the probe, you are affecting how fast the voltage level rises.

If the voltage level on the capacitor rises to 1.3 volts, then variable **v** will become equal to 1. This will allow the program to exit the **DO…LOOP UNTIL** and go back up to the beginning, discharge the capacitor to zero volts, make a tone, and print a line return on screen. Otherwise, the program remains inside the inner **DO…LOOP UNTIL**, where it continues to read the input and print zeros on the Debug Terminal.

Resistance Detector using RCTIME

√ Now enter and run the program RCTIMEDemo.bs2,

```
' Applied Sensors - RCTIMEDemo.bs2
' Simple demo of the RCTIME command.
' {$STAMP BS2}
' {$PBASIC 2.5}

rct     VAR        Word         ' Word variable to track time.
n       VAR        Byte         ' Variable for the bar graph.

LOW 10                          ' Discharge the capacitor.

DO                              ' Beginning of the main routine.
  RCTIME 10, 0, rct             ' Time for the volts to rise to 1.3V.
  LOW 10                        ' Discharge the capacitor to 0 volts.
  DEBUG ? rct                   ' Show the time.
  n = (rct - 1) / 2048 + 1      ' Calculate length of bar graph.
  DEBUG REP "*"\n, CR           ' Display ASCII art bar graph.
  PAUSE 50                      ' Slows down the program.
LOOP                            ' Back to the beginning of main routine.
```

√ As before, experiment by varying pressure and wetness of different materials.

What sorts of **rct** values do you observe?

Here is a commented snippet of our **RCTIME** command:

```
 RCTIME 10, 0, rct    ' Original instruction
'              ^^^----> variable to hold the result (2µs units)
'           ^---------> starts with in10=0, ends when in10=1
'       ^^------------> use pin 10 for this RCTIME command
```

You may know the **RCTIME** command from the *What's a Microcontroller?* Student Guide, but we are going to refresh it here. The **RCTIME** command measures the time that it takes for the capacitor to charge up from zero to the 1.3 volt threshold. The program makes pin 10 low to start off, and that discharges capacitor to zero volts. The **RCTIME** command then turns P10 into an input, and immediately starts looking for the voltage at the I/O pin to cross the 1.3 volt threshold, while at the same time it counts up the elapsed time. The **RCTIME** command counts up in two-microsecond intervals. If the voltage at the pin does cross the 1.3 volt threshold, then the **RCTIME** command wraps up and puts the elapsed time into the variable **rct**, and the program continues with the instruction after the **RCTIME**. Here that is a **LOW 10**, which discharges the capacitor back to zero. If the voltage at the pin does not cross the 1.3 volt threshold within a tenth of a second (0.13107 second, to be exact), the **RCTIME** command gives up. It puts zero into the variable **rct** (to indicate overflow) and then the program continues with the next instruction after the **RCTIME**.

RCTIME counts in units of 2 microseconds (µs), and the maximum value of the count is 65,535 (the maximum value that will fit in a sixteen bit word), so it follows that the maximum time is 131,070 µs (2 µs x 65,535) = 0.13107 seconds. See the *BASIC Stamp Manual* for more information about **RCTIME**. To reiterate, if nothing happens within 0.13107 second, the **RCTIME** puts zero in the variable **rct**, to indicate an overflow condition.

The **RCTIME** command is useful for measuring many different things. Electrically, the circuit can be arranged so that the measured time depends on resistance, capacitance, voltage or current. Many transducers output one of those electrical quantities. For example, the temperature sensor coming up is a transducer that transduces temperature into an electrical current. A simple formula will allow us to convert the value returned by **RCTIME** immediately into temperature. Another type of temperature sensor that is well suited for use with the **RCTIME** command is the thermistor. It has a resistance that varies with temperature. It is not so convenient, because it is harder to calibrate.

Finally an explanation of the ASCII art bar graph in the RCTIMEDemo.bs2. This is a continuing education into the capabilities of the **DEBUG** command. Before the advent of computer graphic displays and printers, these ASCII graphs were the only way to produce a graphical output!

```
n = (rct - 1) / 2048 + 1      ' Calculate length of bar graph
DEBUG REP "*"\n, CR           ' Display ASCII art bar graph
```

When **rct** has a value between 0 and 65535, the value computed for **n** will be between 1 and 32. Note that 65535/2048 = 31. That defines the maximum value, and lower values fall into place. We scale it to 32 maximum simply so that the graph will fit neatly on the width of the Debug Terminal. Subtracting 1 from **rct** is a refinement. Recall that the **RCTIME** command only waits around for 0.13107 second, and then returns **rct = 0** to show that the time was longer than that. If we just graph that, then the longest overflow times end up having the shortest length on the graph. By subtracting 1, **rct = 0** becomes (**rct - 1) = 65535**. (That is how unsigned binary 16 bit math works--zero minus one equals 65535!). The graph makes more sense that way. The **DEBUG** command then uses the **REP** modifier to print **n** stars in the Debug Terminal, followed by a line return. See the *BASIC Stamp Manual* for more information about the **REP** modifier of the **DEBUG** command.

Temperature Sensor Probe using the AD592 and RCTIME

- √ Now, remove the conductivity sensor circuit that was shown in Figure 3-4 Figure 3-5, leaving all of the other circuits in place.
- √ From the Parts Required List, gather together the AD592 temperature probe, the 100 Ω resistor, the 0.22 µF film capacitor, and 1 jumper wire. Note: if you are making your own probe, see Appendix B.
- √ Build the circuit shown by the schematic Figure 3-8 and the wiring diagram in Figure 3-9.

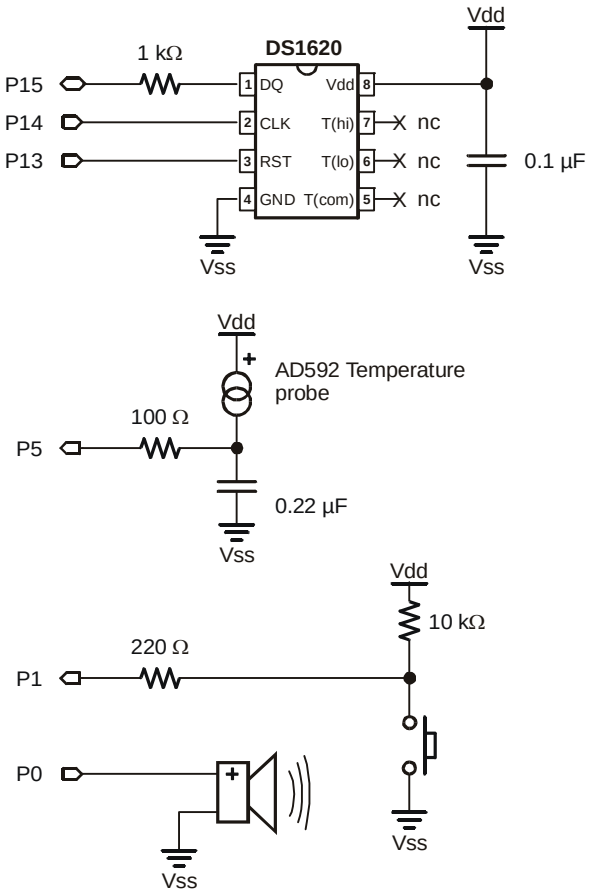


Figure 3-8
AD592 Temperature Sensor and RC-time Schematic

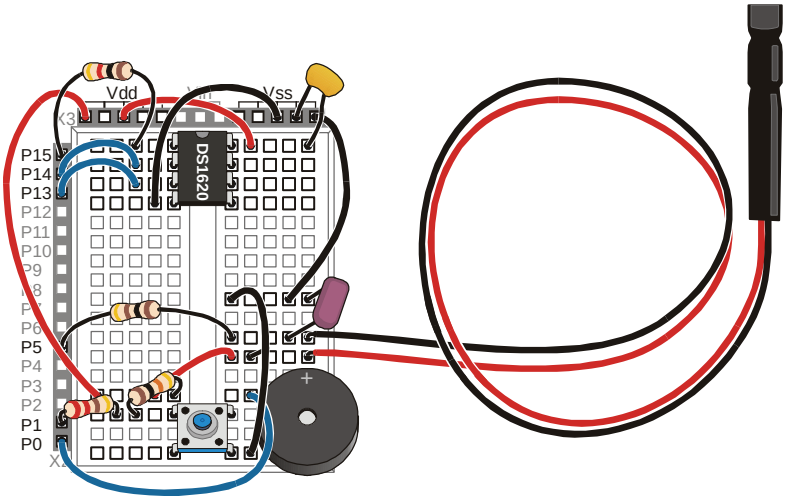


Figure 3-9: AD592 Temperature Sensor and RC-time Wiring Diagram

√ Enter and run the program AD592.bs2.

```
' Applied Sensors - AD592.bs2
' Reading the AD592 temperature sensor using the RCTIME command.
' {$STAMP BS2}
' {$PBASIC 2.5}

Kal     CON         15300               ' Constant to be determined.

rct     VAR         Word                ' A word variable.
TK      VAR         Word                ' Kelvin temperature.
TC      VAR         Word                ' Degrees Celsius.

DO                                      ' Loop forever.
  LOW 5                                 ' Discharge the capacitor.
  RCTIME 5, 0, rct                      ' Time for the volts to rise to 1.3 V.
  TK = Kal/rct*10 + (Kal//rct*10/rct)   ' Calculate Kelvin
  TC = TK - 273                         ' and Celsius.
  DEBUG DEC rct, TAB, DEC TK,           ' Show the results.
        TAB, SDEC TC, CR
  PAUSE 50                              ' Slows down the program.
LOOP                                    ' Back to the beginning of the loop.
```

The display on the Debug Terminal should show three columns, **rct**, which is raw count time (in units of two microseconds) from **RCTIME**, and the calculated Kelvin and Celsius temperatures.

√ Heat up the temperature probe in your hand or by some other means and verify that the **rct** reading on the Debug Terminal goes down as the temperature goes up.

The **TK** and **TC** readings should go up with temperature, but do not pay attention to the exact values yet. You still need to "calibrate" the sensor, which we will do shortly.

Displaying decimal values, negative numbers and numbers in columns: This form of the **DEBUG** command separates the decimal values of the variables with **TAB** characters to put them in columns. The **SDEC** modifier allows for the display of negative numbers.

A word about what sort of device the AD592 transducer is, electrically. It is a current source. The equation that governs its behavior is exceedingly simple:

Output = 1 microamp / Kelvin

That is, at 273 Kelvin (freezing, 0 °C), it produces 273 µA. At 373 Kelvin (boiling, 100 °C), it produces 373 µA. At absolute zero it would produce zero microamps, although that is actually outside its operational limit of -40 degrees Celsius.

If you look at the AD592 in terms of the analogy with a water tank, it is like a flow regulator on the input pipe. The flow does not depend on the level in the tank, nor does it depend on the pressure (voltage) that supplies the current on the other side of the regulator. This is very different from a resistor or wet fingers, where the current depends on several factors. The name **RCTIME** comes from R for resistance, C for capacitance, and the time it takes for a resistor to charge the capacitor. A current source is a very special kind of regulated resistor, one that makes the calculations relatively easy, lucky for us!

AD592 Calibration

The formula that relates the temperature to the time measured by **RCTIME** is a reciprocal: in this case **TK** is Kelvin temperature.

```
rct = constant/TK      or     TK = constant/rct
```

See the box for the theory regarding the rate of change of voltage on a capacitor. The constant will be around 153,000 when the capacitor is 0.22 µF. But it will not be exactly that value due to variations in the component values. That is why it needs to be calibrated.

Theory governing the rate of change of voltage on the capacitor

The equation governing the rate of change of voltage on the capacitor is:

dV/dt = I/C

where **I** is the current and **C** is the capacitance. If you know calculus, and assume that **I** and **C** are constant, you can easily solve for elapsed time in terms of the change in voltage and the capacitance and the current:

t = C * V / I

where **t** is in seconds, **C** is in farads, **V** is in volts, and I is in amps. If we substitute TK in Kelvin for microamps, 0.22 µf for C, 1.3 for the voltage, and 2*rct for the time in microseconds, and taking care for the units, we come up with the formula in the text:

rct = constant / TK

The constant is 153,000, when those ideal values are plugged into the formula. In reality, the capacitor will not be exactly 0.22 µF, the threshold will not be exactly 1.3 volts, and the AD592 will not have an output of exactly 1 microamp per Kelvin. Nevertheless, since there is only one free constant, we will need just one point of calibration.

In order to calibrate the sensor, we need to find the constant for this particular setup. To do this, the AD592 sensor must be put in a location where you know the temperature exactly. A good choice is an ice bath at 0 C, 273 K. With an ice bath reference, **`TK`** = 273, the constant will be (rearranging the previous equation):

```
constant = 273 * rct
```

We have to put the probe into an ice bath, let it stabilize, read the value of **`rct`**, and multiply to find the constant.

√ Prepare an ice bath. For tips on making an effective ice bath, please see the information box below.

Ice bath preparation for calibration:

The melting point of ice made with pure water is a physical constant: zero degrees Celsius, 32 degrees Fahrenheit, 273 Kelvin (Or 273.14 if you want to push the precision). You can get the best results if the ice and water mixture is:

- ✓ made with crushed ice made from distilled water;
- ✓ is held in a vacuum thermos bottle with a narrow mouth;
- ✓ stirred gently while making the measurement; and
- ✓ at least 5 cm of wire is submersed above the sensor probe tip.

Lacking a thermos bottle, you can substitute a well-insulated foam container. Careful preparation is very important if you want to achieve good results in the calibration! Watch until the reading settles down to a steady value, to equilibrium.

Metrologists (not meteorologists!) are scientists who advance the science of accurate measurements. They have to think about all possible factors that could influence the measurements.

√ Place the AD592 temperature probe into the ice bath.
√ Run the program AD592.bs2.
√ Watch the Debug Terminal until the readings equilibrate (become steady).
√ Record the reading for **`rtc`**.
`rtc` = _____.
√ Take that number and multiply it times 273. This is your AD592 calibration constant.
`rtc` x 273 = _____.

Be aware that this constant is specific for this sensor, this BASIC Stamp, and this capacitor.

√ Now round off the constant to the nearest 10, and drop the final digit (a zero). This should be a five-digit number. This will be the value of **`Kal`** you need to substitute in the program.
`Kal` = _____.
√ Put this value in your AD592.bs2 program in place of the 15300 "default" value.
√ Re-run the program. You should see **`TK`** and **`TC`** show the temperature of the calibration bath; 273 Kelvin, 0 Celsius.

Now, to explain the peculiar formula needed to calculate **TK**. Unlike big computers, where the computer language has lots of fancy math available, you will need to stretch the BASIC Stamp's math brain. The reason we need the trick is that the constant, ~153000 (or whatever you found) is larger than the highest possible number that the BASIC Stamp can work with (2^16=65536).

Recall how you did long division in elementary school. This is the same thing, really, but the notation is a little different. Here are examples of the two essential elements in BASIC Stamp math:

BASIC Stamp Notation:	..meaning
1432/524 = 2	Single slash means INTEGER DIVISION (524 goes twice into 1432) and there is a remainder.
1432//524 = 384	Double slash means remainder after INTEGER DIVISION: 1432-(2*524) = 384. Remainder always less than divisor, 384< 524.

Observe that:

(2 x 524) +384 = 1432, that is, the quotient times the divisor, plus the remainder is equal to the original number. That is really the definition of division.

Now, think how you would solve the problem of 143220/524, using elementary school arithmetic:

Equation:	Steps in elementary school arithmetic
27 524) 143220 14148 174	First step in a long division. 524 goes 27 times into 14322, and the remainder is 174. The BS2 knows how to divide into numbers that have numerators less than 65536, so it has no trouble in figuring out that 14322/524 = 27 in one step.
273 524) 143220 14148 1740 1572 168	Next step, the zero is brought down to the right of the 174 remainder. That effectively multiplies the 174 times 10. Then 1740 is divided by 524, and the result, 3 is put after the quotient, which becomes 273. This too effectively multiplies the 27 times 10, as it moves up by one significant figure. The fractional remainder, 168/524, is dropped.

Here is how the BASIC Stamp 2 denotes the same problem:

```
TK = 14322/524*10 + (14322//524*10/524)
    '^^^^^^^^^^^^----------------------first step of division
                   '^^^^^^^^^^^^^^-----the remainder times ten
                                 '^^^-div. to get next digit
    '^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-quotient is 273
    'final remainder 168 is dropped
```

The result is that the BASIC Stamp has calculated 143220/524 = 273.

If you like math, great, you understand or can figure out how it works. If you are one of those who draws a blank when you see math, or you don't have the time to sit down and think it through, well, just take the formula, use it, and turn the crank. A lot of math in computer science is like that. It comes in libraries you just use without thought when you write programs. You assume that the wizards in the tower have gotten it right. Nonetheless, understanding it can be helpful and rewarding, or a career, if you are good at it.

What is the smallest change in temperature that we can detect? Look at Table 3-1 for some typical **`rct`** values and **`TK`** values, if the constant is 143000.

Table 3-1: Temperature Resolution			
Raw Conversion	**Real**	**Kelvin**	**Celsius**
143000/484	= 295.5	295	22
143000/485	= 294.8	294	21
143000/486	= 294.2	294	21
143000/487	= 293.6	293	20
143000/488	= 293.0	293	20
143000/489	= 292.4	292	19

The real temperature resolution is about 0.6 Kelvin. That is, each step in **`rct`** is at best a step of 0.6 Kelvin in temperature. We are rounding it off to 1 Kelvin, losing a little bit of information, but with knowledge beforehand.

If we were to use a larger capacitor (say 0.33µF) in the circuit, the constant would be larger, and the resolution would be improved. On the other hand, with a smaller capacitor (like 0.1µF), the resolution would be worse.

In the next experiment, you will check on the calibration of the AD592 probe in comparison with the DS1620 from Chapters 1 and 2.

Resolution: Suppose you are measuring a quantity that can take on any value between zero and 100. If your instrument can only tell the difference between "greater than 50" and "less than 50" then it has one bit of resolution, that is, the measurement is a sort of "yes/no." On the other hand, if your instrument can tell the difference between 1 and 2 and 3 and so on up to 100, then the resolution is 1%, or about 7 bits (7 bits, because 2^7 = 128). Resolution is not the same thing as accuracy. If your instrument reads 50 when the true value is 52.3, then it is not accurate, or at least it needs to be calibrated. That is true whether it has one bit or 7 bits or more of resolution.

Talking Thermometer Revisited, Two Channels

Now let's combine this new sensor with the DS1620 talking thermometer. The next program you will use is a variation of DS1620MorseCode.bs2, which you may have saved from Chapter 2. If you are careful, you can reload that program, rename it and make the modifications so it matches TwoChannelsThermometer.bs2 shown below. Look for new variables in the Declarations section, one line of code in the Initializations section, and a routine for the AD592 within the Main Routine section.

The first thing you are going to test is to see if the AD592 probe and the DS1620 have the same reading at "room temperature" so you will want them to be at the same temperature.

- √ Position the AD592 probe in contact with the DS1620 on your Board of Education or HomeWork Board.
- √ Enter the program TwoChannelsThermometer.bs2.
- √ In place of the **`Kal`** value shown in the program, enter value you calculated above.

```
' -----[ Title ]-----------------------------------------------------------
' Applied Sensors - TwoChannelsThermometer.bs2
' Talking thermometer, two channels.
'{$STAMP BS2}
'{$PBASIC 2.5}

' -----[ Constants ]-------------------------------------------------------
Dit       CON        70               ' Milliseconds for Morse dit.
Dit2      CON        2*Dit            ' Constants related to Dit.
Dah       CON        3*Dit            ' Ditto.

' -----[ Declarations ]----------------------------------------------------
mc        VAR        Byte             ' Temporary for Morse pattern.
xm        VAR        Byte             ' Morse input variable.
j         VAR        Nib              ' Index for digits to send.
i         VAR        Nib              ' Index for dits and dahs.

x         VAR        Byte             ' General purpose variable, byte.
degC      VAR        Byte             ' Variable to hold degrees Celsius.

rct       VAR        Word             ' Reading from RCTIME.
TK        VAR        Word             ' Kelvin temperature.
TC        VAR        Word             ' Degrees Celsius.

Kal       CON        15300            ' Constant to be determined.

' -----[ Initializations ]------------------------------------------------
' Note: DS1620 has been preprogrammed for mode 2.
' If not, uncomment the instructions on the next line on the first RUN.
' HIGH 13: SHIFTOUT 15,14,[12,2]: LOW 13

OUTS=%0000000000000000               ' Define the initial state of all pins.
     'FEDCBA9876543210
DIRS=%1111111111111101               ' As low outputs
                  '^----------------  except P1, an input for a pushbutton.

FREQOUT 0, 20, 3800                  ' Beep to signal that it is running.

HIGH 13                              ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [238]     ' Send the "start conversions" command.
```

```
LOW 13                                ' Do the command.
LOW 5                                 ' Discharge the capacitor.

' -----[ Main Routine ]------------------------------------------------------
DO                                    ' Start of the main loop.

  DO
  LOOP UNTIL (IN1=0)                  ' Loop here until button is pressed.

  DO
  LOOP UNTIL (IN1=1)                  ' Loop here until button is released.

DS1620:                               ' DS1620 temperature sensor code.
  HIGH 13                             ' Select the DS1620.
  SHIFTOUT 15, 14, LSBFIRST, [170]    ' Send the "get data" command.
  SHIFTIN 15, 14, LSBPRE, [x]         ' Get the data.
  LOW 13                              ' End the command.
  degC = x / 2                        ' Convert the data to degrees C.
  DEBUG ? degC                        ' Show the result on the PC screen.
  xm = degC                           ' Morse routine expects data in xm.
  GOSUB Morse                         ' to the subroutine.

  PAUSE 100

AD592:                                ' AD592 temperature sensor code.
  RCTIME 5, 0, rct                    ' Get the AD592 count.
  LOW 5                               ' Discharge the capacitor.
  TK = Kal/rct*10 + (Kal//rct*10/rct) ' Calculate Kelvin
  TC = TK - 273                       ' and Celsius.
  DEBUG DEC rct, TAB, DEC TK,         ' Show the results.
        TAB, SDEC TC, CR
  PAUSE 50                            ' Slows down the program.

LOOP                                  ' Back to wait for button again

' -----[ Subroutines ]--------------------------------------------------------
Morse:                                ' Emits byte xm as Morse code.
  FOR j=1 TO 0                        ' Send 2 digits, tens then ones.
    mc = xm DIG j                     ' Pick off the (j+1)th digit.
    mc = %11110000011111 >> mc        ' Set up pattern for Morse code.
    FOR i=4 TO 0                      ' 5 dits and dahs.
                                      ' Send pattern from bits of mc.
      FREQOUT 0, Dit2*mc.BIT0(i)+Dit, 1900
      PAUSE Dit                       ' Short silence.
    NEXT                              ' Next i, dit or dah of five.

    PAUSE Dah                         ' Interdigit silence.
  NEXT                                ' Next j, digit of two.
RETURN                                ' Back to main.
```

- √ If the Morse code becomes obnoxious to you or your classmates, simply unplug the wire from P1 to turn off the buzzer, or put an apostrophe in front of the **`GOSUB Morse`** and turn it into a comment.
- √ Run this with the AD592 probe in direct contact with the DS1620 on your Board (and no direct sunlight). If you have some, you can put some heat sink compound (thermally conductive grease) between the two to improve the contact.
- √ Be sure the readings are constant, and record the readings in degrees Celsius:

  DS1620:_____
  AD592 :_____

They should be pretty close to one another. You just calibrated the AD592 in an ice bath, and the DS1620 data sheet specifies that its reading will be within ±0.5 degree.

Save the program TwoChannelsThermometer.bs2 on disk, following your teacher's instructions.

Automatic Calibration (Advanced Topic)

One feature of many modern instruments is automatic calibration. For example, since we know that the DS1620 has ±0.5 degree accuracy, we might like to skip the preparation of an ice bath, which, after all, requires quite a few materials and effort to do it right. We could use the DS1620, at room temperature, as the calibration reference. You could attach a new AD592 temperature probe to the Board of Education or HomeWork Board, put it in contact with the DS1620, and let it sit for a few minutes, and then press a button to enter the calibration value. Voilá! The BASIC Stamp would calculate the correct calibration value and stores it in EEPROM for you.

The program ThermometerCalibration.bs2 can do just that. The program also directs you storing and retrieving word-size data in the EEPROM. You can modify TwoChannelsThermometer.bs2 using the instructions below, and the complete program ThermometerCalibration.bs2 is also listed for your reference.

- √ Open TwoChannelsThermometer.bs2.
- √ Save as ThermometerCalibration.bs2.
- √ Remove the calibration constant:

```
Kal   CON  15300
```

√ Replace the calibration constant you just removed with:

```
EKal  DATA Word 15300 ' constant to be determined
kal   VAR  Word       ' for calibration constant
```

The value **EKal** points to a value in the EEPROM, and that value will be transferred to and from the variable **kal**, using the BASIC Stamp's **READ** and **WRITE** statements. The calibration constant is a word value, but the EEPROM stores only bytes. PBASIC 2.5 automatically stores words in two successive bytes of EEPROM when you follow the **DATA** directive with the **Word** modifier. This also works with **READ** and **WRITE**.

√ Enter the following line in the program, just before the **RCTIME** instruction.

```
READ EKal, Word kal  ' Get calibration constant
    '^^^^------------ read from location EKal
    '      ^^^^^^^^--- word variable kal
```

√ Also add the section to the Main Routine so that if you hold the button down for a long time, it will branch to a special calibration routine. (Do you recognize this from the LongClick.bs2 program from Chapter 2?)

```
  x = 0                             ' Counter initialization.

  DO                                ' Loop beginning.
    PAUSE 100                       ' 0.1 second pacing.
    x = x + 1                       ' Increment counter.
    IF x>30 THEN GOSUB Calibrate    ' Calibrate if long click.
  LOOP UNTIL (IN1=1)                ' Until button is released.
```

√ Finally, add the following subroutine at the end of the program:

```
Calibrate:
  FREQOUT 0, 5, 3400                  ' Show we got here.

  DEBUG "The probe should be in contact", CR
  DEBUG "with the DS1620",CR

  TK = degC + 273                     ' Kelvin from DS1620.
  kal = TK/10*rct + (TK//10*rct+5/10) ' Compute and round kal.

  DEBUG ? kal                         ' Show value of kal.
```

```
        WRITE EKal, Word kal                ' Write kal on EEPROM.

        FREQOUT 0, 5, 1900                  ' Show finished.

        x = 0                               ' Reset counter.

      RETURN                                ' Back to main program.
```

√ Check your work against the program ThermometerCalibration.bs2 below.

```
' -----[ Title ]-----------------------------------------------------------
' Applied Sensors - ThermometerCalibration.bs2
' Talking thermometer, two channels, with calibration.
'{$STAMP BS2}
'{$PBASIC 2.5}

' -----[ Declarations ]-----------------------------------------------------
Dit       CON        70               ' Milliseconds for Morse dit.
Dit2      CON        2*Dit            ' Constants related to Dit.
Dah       CON        3*Dit            ' Ditto.

mc        VAR        Byte             ' Temporary for Morse pattern.
xm        VAR        Byte             ' Morse input variable.
j         VAR        Nib              ' Index for digits to send.
i         VAR        Nib              ' Index for dits and dahs.

x         VAR        Byte             ' General purpose variable, byte.
degC      VAR        Byte             ' Variable to hold degrees Celsius.

rct       VAR        Word             ' Reading from RCTIME.
TK        VAR        Word             ' Kelvin temperature.
TC        VAR        Word             ' Degrees Celsius.
kal       VAR        Word             ' Calibration constant.
EKal      DATA       Word 15300       ' Initial value of Constant on EEPROM.

' Note: DS1620 has been preprogrammed for mode 2.
' If not, uncomment the instructions on the next line on the first RUN.
' HIGH 13: SHIFTOUT 15,14,[12,2]: LOW 13

OUTS=%0000000000000000               ' Define the initial state of all pins.
     'FEDCBA9876543210
DIRS=%1111111111111101               ' As low outputs
                  '^----------------   except P1, an input for a pushbutton.

FREQOUT 0, 20, 3800                  ' Beep to signal that it is running.

READ EKal, Word kal                  ' Get calibration constant.
LOW 5                                ' Discharge the capacitor.
```

```
HIGH 13                                 ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [238]        ' Send the "start conversions" command.
LOW 13                                  ' Do the command.

' -----[ Main Routine ]-------------------------------------------------
DO                                      ' Start of the main loop.

  DO
  LOOP UNTIL (IN1=0)                    ' Loop here until button is pressed.

  x = 0                                 ' Counter initialization.
  DO                                    ' Loop to track pressing time.
    PAUSE 100                           ' 0.1 second pacing.
    x = x + 1                           ' Increment counter.
    IF x>30 THEN GOSUB Calibrate        ' Calibrate if button held > 3 seconds.
  LOOP UNTIL (IN1=1)                    ' Loop here until button is released.

DS1620:                                 ' DS1620 temperature sensor code.
  HIGH 13                               ' Select the DS1620.
  SHIFTOUT 15, 14, LSBFIRST, [170]      ' Send the "get data" command.
  SHIFTIN 15, 14, LSBPRE, [x]           ' Get the data.
  LOW 13                                ' End the command.
  degC = x / 2                          ' Convert the data to degrees C.
  DEBUG ? degC                          ' Show the result on the PC screen.
  xm = degC                             ' Morse routine expects data in xm.
  GOSUB Morse                           ' To Morse subroutine.

  PAUSE 100

AD592:                                  ' AD592 temperature sensor code.
  RCTIME 5, 0, rct                      ' Get the AD592 count.
  LOW 5                                 ' Discharge the capacitor.
  TK = kal/rct*10 + (kal//rct*10/rct)   ' Calculate Kelvin
  TC = TK - 273                         ' and Celsius.
  DEBUG DEC rct, TAB, DEC TK,           ' Show the results.
        TAB, SDEC TC, CR
  PAUSE 50                              ' Slows down the program.

LOOP                                    ' Back to wait for button again

' -----[ Subroutines ]---------------------------------------------------
Morse:                                  ' Emits byte xm as Morse code.
  FOR j=1 TO 0                          ' Send 2 digits, tens then ones.
    mc = xm DIG j                       ' Pick off the (j+1)th digit.
    mc = %11110000011111 >> mc          ' Set up pattern for Morse code.
    FOR i=4 TO 0                        ' 5 dits and dahs.
                                        ' Send pattern from bits of mc.
      FREQOUT 0, Dit2*mc.BIT0(i)+Dit, 1900
      PAUSE Dit                         ' Short silence.
    NEXT                                ' Next i, dit or dah of five.
```

```
    PAUSE Dah                           ' Interdigit silence.
  NEXT                                  ' Next j, digit of two.
RETURN                                  ' Back to main program.

Calibrate:
  FREQOUT 0, 5, 3400                    ' Signal to show we got here.

  DEBUG "The probe should be in contact", CR
  DEBUG "with the DS1620",CR

  TK = degC + 273                       ' Kelvin temperature of DS1620.
  kal = TK/10*rct + (TK//10*rct+5/10) ' Compute and round off kal.

  DEBUG ? kal                           ' Show value of kal.

  WRITE EKal, Word kal                  ' Write kal on EEPROM.

  FREQOUT 0, 5, 1900                    ' Show finished.

  x = 0                                 ' Reset counter.

RETURN                                  ' Back to main program.
```

When you first run the program, the temperatures returned from the AD592 will be incorrect, due to incorrect default value of **`EKal`**.

- √ Run the program ThermometerCalibration.bs2.
- √ Put the AD592 in contact with the DS1620. IMPORTANT: make good contact between the AD592 and the DS1620, using a wire to hold them together, and improve the contact with silicone heat sink grease if you have some. Be sure there are no nearby sources of heat.
- √ Click the button and watch the reading until you see that it has settled down.
- √ When ready, press and hold the button until you hear the calibration click.

When you release the button, the readings should suddenly become correct in comparison to the DS1620. The AD592 probe can now be extended out to measure other temperatures. This kind of auto-calibration capability is especially important for instruments that read things like conductivity or pH (acidity), where the sensors need frequent recalibration.

Let's verify that the calibration routine worked, by taking the temperature in the ice bath.

- √ Place your AD592 probe in the ice bath.

√ DO NOT press the calibration button.
√ Read the temperature in the Debug Terminal.

It should read close to zero, within the ±1 degree resolution of your Board of Education or HomeWork Board measurement system. Remember, the calibration routine depends on having the AD592 at the same temperature as the DS1620! With this auto-calibration routine, do not press the calibration button when the probe is in the ice bath!

The calibration constant comes from the equation:

```
constant = (true Kelvin temperature) * rct
```

We are assuming that the DS1620 gives us the "true" temperature. Suppose the DS1620 is at 25 degrees Celsius, 298 Kelvin. Suppose the value of **`rct`** is 591. So,

```
constant = (298 * 591) = 176134,
```

. . .and the value that must go into the EEPROM is the top five digits of that, rounded off as before. The trick is to get this result on the BASIC Stamp without overflowing 16 bits. It takes two steps, rewriting the reference temperature as:

```
298  =  29 * 10 + 8
```

…or in the notation of the BS2, for any Kelvin temperature:

```
  TK  =  (TK/10)*10 + (TK//10)
'        ^^^^^^^^^^------------Integer division,
'                     ^^^^^^^^-plus remainder.
```

Multiplying both sides by **`rct`**, then dividing by ten (to get the top 5 digits of the product) we end up with the formula in the program:

```
kal = TK/10*rct + (TK//10*rct+5/10) ' Compute and round kal
```

The 5 added before the final division is for rounding. Think it through.

The **`WRITE`** statement stores the word **`kal`** in the location, **`EKal`**. Subsequently, the **`READ`** command retrieves the calibration value in exactly the same way. The calibration constant stays there unchanged in EEPROM until (1) you press the calibration button

again, or (2) until you re-download the program again by running it from within the BASIC Stamp Editor.

Some Field Research: Temperature Experiments

Investigate the temperatures you find around and about your environment. Your instructor may have specific instructions. Measure the temperature in:

- √ Hot and cold tap water
- √ An aquarium
- √ Out in the open sun
- √ Under trees or bushes
- √ Underground

Note that it is possible to extend the length of the temperature probe, if you want, simply by adding more wire. Look for those micro-environments. Where are the sources of heat that lead to variation in temperatures in microenvironments?

Here are a few specific experiments you can try. These merely illustrate how a temperature probe can be used to measure more than just temperature.

The Psychrometer: Measuring Humidity

- √ Measure and record the air temperature in the shade.
  Dry bulb temperature:_____.
- √ Wrap wicking, gauze, cloth or a piece of paper towel around the temperature sensor and hold it in place with rubber bands or wire. Try to make the wrapping tight and compact, and not too much of it!
- √ Make the covering wet.
- √ Fan air across the wet sensor, or spin the probe rapidly around on its wire. It will cool down to its final value quicker if you have constructed a compact wet bulb.
- √ Measure an record the new final temperature:
  Wet bulb temperature:_____.
- √ Subtract the wet measurement from the dry measurement.

You might expect 4 or 5 degrees Celsius of wet bulb depression in a room at 50% relative humidity. Everyone knows that a wet body cools off in the breeze. The cooling effect is greatest in dry air. This is called the wet bulb depression. It depends mainly on the relative humidity in the air, and on wind speed. At higher wind speeds it depends only on

relative humidity. An instrument to measure humidity using a wet and a dry thermometer in this fashion is called a "psychrometer" (from the Greek root, psychros, meaning "cold"). By spinning the wet bulb through the air, it becomes a "sling psychrometer." Psychrometric charts are where you would go to look up the humidity as a function of the dry and wet bulb temperatures. Try this both inside and out of doors. For interest, an example of a psychrometric chart is shown in Figure 3-10. This chart was designed by Hong Kong University in order to classify building comfort zones.

- "y" axis on the right side is absolute humidity.
- "x" axis is dry-bulb temperature (°C).

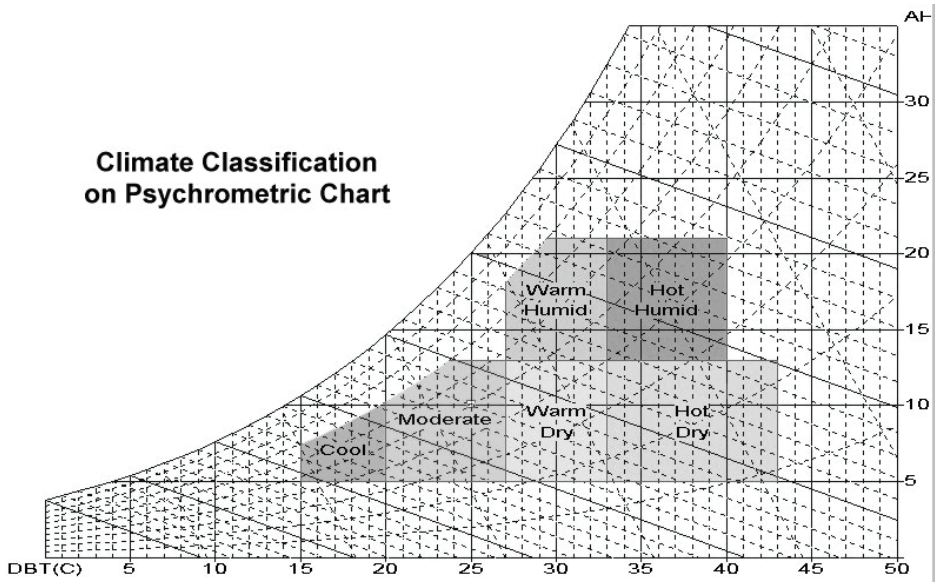


Figure 3-10: Example of Psychrometric Chart

Ice Point Depression in Salt Water: Measuring Salinity

You have an ice bath from your calibration of the temperature probe. When ordinary table salt is mixed with ice water, what happens to the temperature of the mixture? Think about the role of salt water in relation to ice cream makers, icy roads and icebergs.

- √ Set up your ice bath again, if necessary.
- √ Design and complete a quantitative experiment that features varying amounts of the same type of salt added to the ice bath.
- √ Design and complete an experiment that compares different types of salt added to the salt bath.

Be sure to use a well-insulated container for best results. What conclusions can you draw about the relationship between salinity and the freezing point of water?

The Pyranometer: Measuring Solar Radiation

- √ Wrap a cylinder of aluminum foil around the temperature sensor, twisted at the end.
- √ Tie a thread to the foil so that you can pull it put it off the probe.
- √ Put it in the sun, inside a clear plastic or glass jug to cut down the wind.
- √ Record the temperature reading when it settles down.
  Reading with foil wrap: _____.
- √ Use the thread to pull off the foil.
- √ Let the temperature re-equilibrate.
- √ Record the new temperature reading with and without the foil.
  Reading with black sensor: _____.
- √ Compare the difference between the two readings.

Everyone knows that dark objects can get hot in the sun. A device that measures radiation by looking at the temperature difference between a black and a white surface is called a "black and white" pyranometer. (Pyr is a Greek root that means "fire" but you already knew that!)

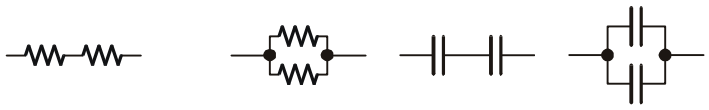
The Hot Probe Anemometer: Measuring Wind Speed

- √ Allow the black temperature sensor to become hot in the sun.
- √ Record the temperature _____.
- √ Slowly spin the probe around on the end of its wire for 30 seconds.
- √ Record the "slow spin" temperature _____.
- √ Allow the temperature sensor to heat up in the sun to its previously recorded temperature.
- √ Quickly spin the probe around on the same length of wire for 30 seconds.
- √ Record the "quick spin" temperature _____.

√ Find the difference between your "slow spin" and "quick spin" temperature measurements.

Everyone knows that warm bodies cool off in the breeze. This shows that temperature can be used to measure winds speed. "Hot wire anemometers" use a platinum wire both as the sensing element (its resistance changes with temperature) and as the heating element (electrical current passing through it makes it heat up).

Challenge!

1. Hook up a circuit as in Figure 3-5, except use 100 kΩ resistor and a 0.22 µF capacitor. Measure the **`rct`** value, and insert it in the center cell of the table below. You have two 0.22 µF capacitors in your kit, and two 100 kΩ resistors. By making parallel and series connections of those parts, you can fill in the rest of the entries in the chart.
2. 

2 R Ω	R/2 Ω	C/2 µF	2C µF

RCTIME vs. R and C	50 kΩ	100 kΩ	200 kΩ
0.11 µF			
0.22 µF			
0.44 µF			

3. Modify the program AD592.bs2 so that it shows its result in degrees Fahrenheit and degrees Rankine, instead of degrees Celsius and Kelvin. (Rankine = Kelvin*1.8.). Do not just convert Kelvin to degrees Rankine though. Calculate a new constant for Rankine = constant/rct.

4. Pushbutton application: sometimes the Morse code sound may be annoying. You can unplug the wire from P0 to shut it off. But the challenge here is to make a way to turn it off in software. Think of a way using the pushbutton to toggle the sound on and off.

5. Hook up a 0.1 µF capacitor and 100 Ω resistor to P10, as in Figure 3-5, with the conductivity sensor. Also hook up an LED and resistor to P8, so that the BASIC Stamp can turn the LED on and off. Then (a) make a program that turns on the LED only when the probe is dipped in water; (b) Make a program that pauses for 10 seconds, then tests the input and blinks the led if the probe was dipped in water anytime during the 10 second pause. Then recharge the capacitor, turn the pin to an input, then go back to the pause; and (c) Modify the program so that it counts up how many times you dip the probe in water, one count possible for each pause. This program shows how the capacitor helps to monitor things like

rain gages and traffic counters–switches that close unpredictably for short periods of time. The capacitor is used as a "memory" remembering the event until the BASIC Stamp gets around to looking at the input.

Chapter 4: Light on Earth and Data Logging

The theme of the Light on Earth and Data Logging experiment is "light, and its importance for everything under the sun." To demonstrate this, we'll build a light meter/data logger. The activities we'll perform in this experiment include:

- A light sensor photodiode using **`RCTIME`**, and observations of the orders of magnitude scales of intensity
- A combined temperature and light meter
- A pushbutton data logger for temperature and light
- A couple of experiments using the light meter/data logger

The sun is the driving force of most of the weather and physical processes on earth. Where would we be without photosynthesis? People have been taking measure of the sun since the dawn of prehistory. At Stonehenge, at the Caracol of Chichen Itza, and around the world, the ancients took the measure of the solar cycle of the seasons in relation to agriculture and to temporal and spiritual life.

Temperature is a relatively simple variable in comparison to light. Light comes in a spectrum of colors, both visible and invisible, and the spectrum extends out to fuzzy limits of wavelength. It has polarization and direction. Many aspects of light have special significance. Certain wavelengths are responsible for sunburn; other wavelengths are special for the ripening of fruit. There are subtle patterns of light. For example, bees can see patterns of deep blue on flowers in the ultraviolet range that the human eye cannot perceive, and hummingbirds' vision extends farther into the red, infrared, than our own. Light is important to us in a tremendous range of intensities, from solar energy for electricity and heating, to bioluminescence of creatures in the deep oceans.

Like temperature, light is often used to measure other things. For example, instruments for detecting air quality and CO_2 are often based on lasers, or on the fact that gases absorb light at characteristic wavelengths. Astronomers use the spectra to deduce the chemical composition of the stars and interstellar gasses. On the other end of the scale of size, light is used to probe chemical processes in DNA and the mechanisms of the living cell. In a practical arena, light is used for motion detectors, for indicators, and of course for illumination, which is in itself a whole specialty of engineering.

One fundamental law is that the intensity of light from a point source falls off with the square of the distance. That is, at double the distance from a light bulb (or from the sun), the light intensity will be 1/4 of its value at the first location. The same amount of energy is spread over 4 times the area. Using the light meter you build in this lesson, you will have a tool to investigate that law, as well as to explore light variation in your environment. This concept of light attenuation is illustrated in Figure 4-1

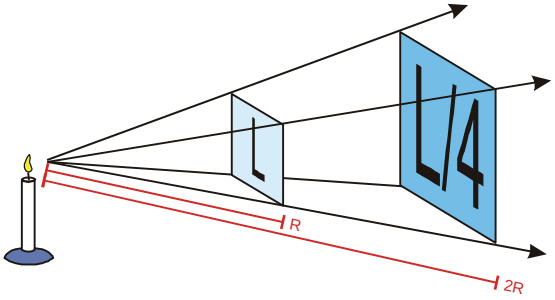


Figure 4-1
Light Attenuation

At twice the distance from a bulb (or from the sun), the light intensity will be 1/4 of its value at the first location. We'll investigate this law with our microcontroller-based light meter.

Parts Required

For this experiment we'll be leaving parts on the Board of Education or HomeWork Board that we installed in Chapter 3. The following parts are required for Chapter 4:

(1) Photodiode
(4) 100 Ω resistor (brown black brown)
(2) 0.01 µF poly capacitor
(1) 0.22 µF poly capacitor
(2) 100 pF ceramic capacitor (101)
(1) Red LED
(1) Green LED
(1) 9 V battery (not included)
A 50 watt R20 spotlight, if available, for experiments (not included)

Building the Circuit

Photodiode as a Light Transducer

What's a Microcontroller? introduced one kind of light-responsive sensor called a photoresistor, which has a cadmium sulfide-coated surface that becomes less resistant to current as it is exposed to brighter light. In this lesson, we will use a different type of

photodetector, a photodiode. A photodiode passes an electrical current (in amps) that is directly proportional to light intensity. This characteristic makes it especially well suited for quantitative measurements.

You may also be familiar with the light emitting diode (LED), which turns electrical current into light. The LED emits light when the current flows in the direction of the diode arrow. You may know that electrons (negative charges, e-) actually flow in the opposite direction. But in an accident of historical interpretation, in electronics we usually think of current as if it were carried by positive charges. Again, the LED emits light when the (positive) current flows in the direction of the arrow.

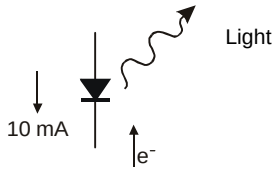


Figure 4-2
LED

The LED emits light when the (positive) current flows in the direction of the arrow.

It also works in reverse. Light falling on a diode produces electricity. If you connect a voltmeter to a diode exposed to light, you will measure a fraction of a volt, with the polarity as indicated. Electrons accumulate at the cathode end.

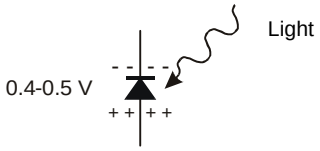


Figure 4-3
Photodiode

A photodiode produces electricity with light.

When you connect the photodiode into a circuit loop with a wire, current flows around the loop. The amount of current is proportional to the intensity of the light. This is fundamental. It is the light that generates the charges that make the current. The only way they can get back together is to flow around the external circuit. Observe that the electrons flow clockwise around the circuit, as shown in Figure 4-4. The conventional current (positive charges) flows in the opposite direction, against the direction of the diode arrow. This is called a photocurrent, and it is a reverse current. Compare this to the forward current that lights up an LED. This business of the arrow can be confusing, but in electricity, it is all relative. In this circuit, the voltage across the diode is zero–it is short-circuited.

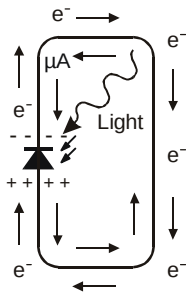


Figure 4-4
Photocurrent

Photocurrent occurs when conventional current (the positive charges) flow in the opposite direction, against the direction of the diode arrow.

Sensitivity to light is a fundamental property of transistors and diodes. In most places, that would be an undesirable side effect. Transistors and integrated circuits are usually potted in a plastic, ceramic or metal case, and one reason to do that is to keep out light that would greatly affect their performance. Photodiodes are made especially to accentuate the sensitivity to light. Look at the photodiode in your kit. It has a clear epoxy top, and underneath that you can see a small black square of silicon, with wires attached at the sides. Electrical charge builds up on the top and the bottom, where the wires pick it up. The difference between a solar panel and a photodiode is largely in the area of the diode. Solar panels have huge areas, square feet or square meters, so they can intercept a lot of light and produce lots of current and power, measured in amps and watts. The photodiode is made of especially pure material for measurement, not for energy production.

One thing that makes photodiodes very useful for measurement is that a simple equation governs their behavior as a transducer:

*i = constant * light intensity*

The sensor is linear. That means that if the light level increases, say, by a factor of 1000, then the current through the diode will also increase by that same factor. For the photodiode, this holds true over several orders of magnitude, over several powers of ten. It is that characteristic that makes it so useful for measurement.

The simple equation also holds true when the diode is hooked up in a reverse voltage circuit as in Figure 4-5. At the same light intensity, the amount of current is exactly the same here as it would be in the short circuit of Figure 4-4. The current through the photodiode charges the capacitor. The charge accumulates on the capacitor as shown, and the voltage across the capacitor gradually increases. The BASIC Stamp 2 program will

measure the time it takes for the voltage at P6 to fall from 5 V down to 1.3 V as shown in Figure 4-5.

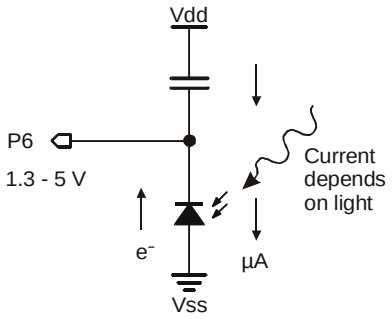


Figure 4-5
Photodiode Circuit

With this circuit the BASIC Stamp can measure the time it takes for the charge to fall from 5 V to 1.3 V. This is called a resistor/capacitor circuit.

√ From the Parts Required list above, gather together the 0.01 µF capacitor, the photodiode, and a 100 Ω resistor.

√ Add the circuit to your breadboard as shown in the schematic (Figure 4-6) and wiring diagram (Figure 4-7).

√ Add an optional resistor between the piezo transducer and BASIC Stamp PO to quiet down the speaker if your instructor requests it.

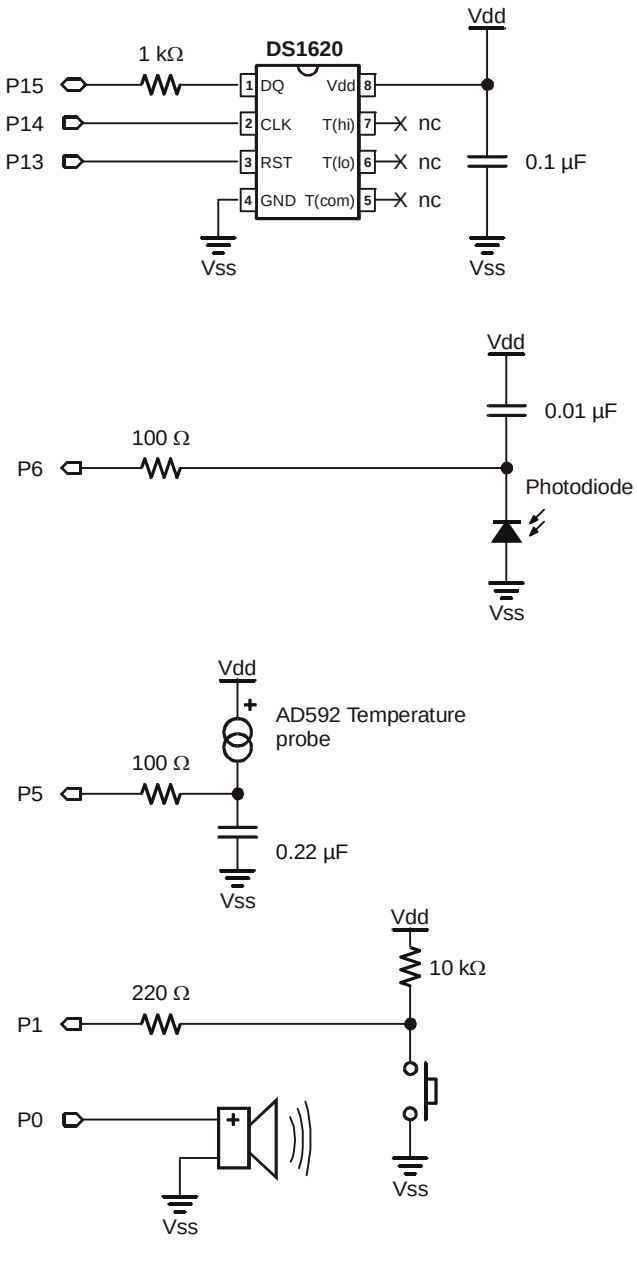


Figure 4-6
Photodiode Light Transducer Schematic.

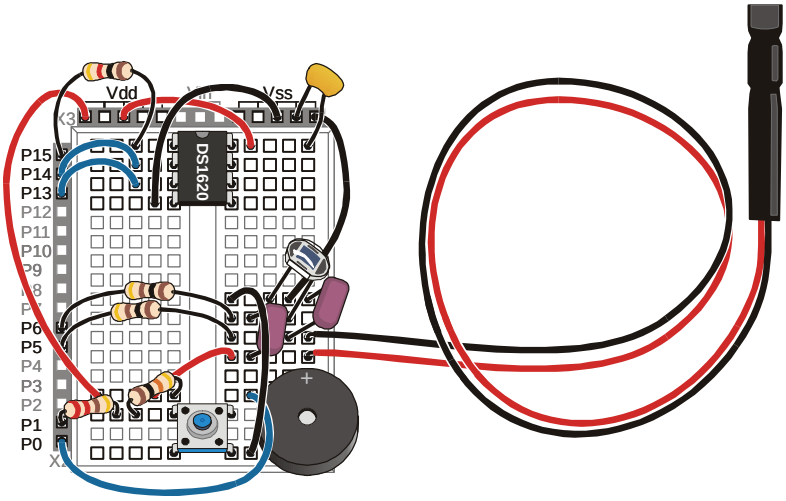


Figure 4-7: Photodiode Light Transducer Wiring Diagram

Node: A node consists of all the points in a circuit that are connected together. Each row of 5 holes on the Board of Education or the HomeWork Board is a node, because all of the holes are connected electrically. One node in Figure 4-7 is the row where the diode, the capacitor and P6 (via the resistor) meet.

√ Enter the program LightMeter.bs2.

```
' Applied Sensors – LightMeter.bs2
' Sound out light levels from the photodiode.
' {$STAMP BS2}
' {$PBASIC 2.5}

rct     VAR     Word          ' Variable for RCTIME.

HIGH 6                        ' Discharge the capacitor.

DO
  DO
    RCTIME 6, 1, rct          ' Time for volts to fall to 1.3V.
    HIGH 6                    ' Discharge the capacitor.
  LOOP UNTIL (rct<>0)         ' No sound if RCTIME overflows.
  FREQOUT 0, 1, 3400          ' Make a click.
LOOP                          ' Repeat the main loop again.
```

- √ Download and run the program.
- √ Expose the light sensor to dim light, such as under your desk, and brighter light, such as a lamp.
- √ Read the next couple of paragraphs before you get too carried away.

Light levels can vary tremendously in the natural environment. Our eyes have an amazing capability to accommodate both dim and bright light. Sensitivity is the amount of light that it takes to get a response. Sometimes we need a sensitivity adjustment, or a switch for "high" and "low" sensitivity. Cameras and eyes have an iris that opens or closes to adjust the amount of incoming light, to extend the range of sensitivity.

Observe that the program jumps back to the top without making a sound, if the value of **`rct`** is equal to zero. That is at the dim end of the range. You may have to cover the sensor with a box or something to make it dark enough to see this effect. The capacitor takes too long to charge and the **`RCTIME`** command does not see a transition from 1 to 0 at P6 within its 0.13107-second limit. At the other extreme, in bright light, the clicking becomes very high-pitched and bunched up, so you can't distinguish differences.

- √ Find your 0.22 µ capacitor and your 100 pF capacitor.
- √ Replace the 0.01 µF capacitor with 100 pF capacitor to make it more sensitive and responsive in dim light.
- √ Try taking measurements in dim light again.
- √ Try fastening a piece of tissue paper, if available, over the photodiode to decrease its sensitivity.
- √ Try taking measurements back in brighter light.
- √ Take out the 100 pF capacitor, and put in a 0.22 µF capacitor.
- √ Take measurements in brighter light again.

Now that you have figured out when to swap out capacitors to obtain the proper sensitivity to the given light conditions you are measuring, let's explore your surroundings. Have fun with this!

- √ Disconnect your Board of Education or HomeWork Board from the programming cable.
- √ Power your board with a 9-volt battery.
- √ Carry along the 100 pF and a 0.22 µF capacitor and change the sensitivity whenever you feel it is necessary.
- √ Explore your surroundings, indoors and outside, if possible:

- Try pointing the sensor both up and down, to detect the direct light and the reflected light and patterns of light and shade.
- Try scanning close across the objects on a desk, both shiny and dull.
- Try scanning close to the bold pattern on a curtain or clothing.
- Try scanning the flickering pattern on your computer screen or TV set.
- Try scanning a relatively dark place using the 100 pF capacitor.
- Try scanning outside in sunlight, if available, using the 0.22 µF capacitor.

More about Measuring Light Intensity

A **Pyranometer** quantifies light intensity by the light energy hitting a surface per unit time. This is the right measurement if you are designing a solar panel system or a solar water heater, or if you are an architectural engineer thinking about the energy efficiency of a building. This kind of light intensity is measured in watts per square meter. The power input from the sun at the earth's surface, on a clear summer day, is a little over 1000 watts per square meter, or 75 watts per square foot. The solar energy input above the atmosphere is nearly 1400 watts per square meter, which is often given in other units as 2 Langleys per minute. (1 Langley = 1 calorie per square centimeter). Sometimes we are interested only in the energy of certain wavelengths. For example, ultraviolet light at about 300 to 320 nanometers causes sunburn, not only in people, but in other life on earth too, like coral in the ocean. This UVB light can be separated out and measured. It amounts to less than 0.1% of the total, less than 1 watt per square meter. But it is a very significant 0.1%. More and more UVB is getting down to the earth as the ozone in the upper atmosphere is depleted, due it seems to human activity, the use of certain CFC chemicals.

A **Photometer** quantifies light intensity as our eyes see it. This is the subject of illumination engineering, and of physiology. How does an owl see? Our eyes are most sensitive to bright light in the yellow-green range, and the sensitivity falls off in the red and the blue. The question of light intensity is still one of energy per unit area per unit time, but now it includes only energy at the wavelengths we can see. It is measured in special units, *lux* or *foot-candles*. The light intensity looking at full sunlight is about 110,000 lux, but of course, looking at direct sunlight is not something that we do. It is too intense. In contrast, a 100 watt light bulb, viewed from 1 meter distance, is about 100 lux. That too is perceived as bright. Normal room lighting is measured in 10s of lux. There is a tremendous range of values that sensors, including our eyes, have to deal with – around 7 or 8 orders of magnitude. But that is nothing compared to the range of light levels arriving from celestial objects, which is detectable in over 20 orders of magnitude.

A **PAR meter** quantifies light intensity as it affects the growth of plants. This is of great interest to farmers, and aquaculturists, and botanists. Photosynthesis occurs in special band of wavelengths, called the photosynthetic action spectrum. PAR stands for Photosynthetically Active Radiation. Measurement of PAR allows botanists to estimate how much growth would be possible for a given plant, if light were the limiting factor. Light comes in packets of energy, called quanta, and each unit of photosynthesis takes one quanta of light. The units of PAR are micromoles of quanta per square meter per second. Full sun illuminosity is about 2000 moles per square meter per second. A lot of plant biology has to with plant adaptations to light levels.

A **Spectrophotometer** is the most versatile of the bunch. It tells you how much light energy is in each narrow band of wavelengths across the spectrum. In contrast, the photocurrent in your photodiode is due to the convolved effect of many different wavelengths. The spectrophotometer can be used to characterize and calibrate almost any of the other light measuring instruments, but, needless to say, it can be a much more complicated and expensive instrument. An economical version of this is a colorimeter that you might find in a paint store for matching colors.

How LightMeter.bs2. Works

Let's go back to our own light meter, and talk more about the BASIC Stamp program that makes it work. Note that one side of the capacitor is connected to Vdd (+5) instead of being connected to Vss (ground) as it was in the AD592 temperature sensor circuit. Here, LightMeter.bs2 starts off with `HIGH 6` to discharge the capacitor. This might seem

strange, making the pin high to discharge the capacitor, but note that a capacitor is said to be "discharged" when the voltage from one side to the other is equal to zero. The top of the capacitor is connected to +5 volts, so **`HIGH 6`** makes both sides equal to +5 volts, so it is discharged. Figure 4-8 illustrates the time course of the voltage at P6.

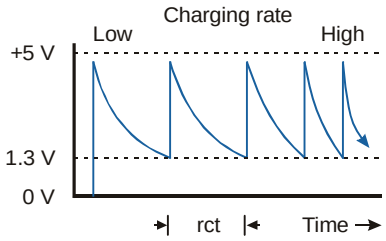


Figure 4-8
Resistor/Capacitor Discharge

*Light strikes the photodiode, causing it to sink current from the discharged capacitor. As the capacitor charges, the voltage at P6 decays from nearly 5 V down to 1.3 V. The **`RCTIME`** command holds P6 as an input during that time. As soon as the **`RCTIME`** detects the 1.3 volt level, the program moves to the **`HIGH 6`** and quickly discharges the capacitor, and the voltage at P6 quickly returns to +5 volts.*

Look carefully at the **`RCTIME`** command in LightMeter.bs2. The second argument is now a 1. That argument was 0 in the programs in Chapter 3 with the AD592 temperature transducer. This instructs the **`RCTIME`** routine to count time while P6 is equal to one, and to stop when P6 makes the transition to zero. Try changing the second argument from 1 to 0. It doesn't work, does it? That is, is it sensitive to light? Do you hear a very high tone, a very low tone, or no tone at all? For debugging, it is good to think these "what if..." kinds of questions.

The BASIC Stamp accepts both forms of the command, with the second argument either 0 or 1, because there are situations where one or the other will be the best, or the only, choice. For example, the AD592 temperature sensor works fine in the circuit of Chapter 3, but it would fail to work in this one. That has to do with the voltage requirements of the AD592. We can't go into all the advantages and disadvantages of one circuit over the other, but it is good to be aware of this flexibility when you get down to the fun of designing your own serious projects.

Observe that the pitch of the tone from the annunciator goes higher in brighter light. Why is that, when the time to discharge the capacitor, **`rct`**, goes lower in brighter light? Be sure you understand the reason, which is that with lower values of **`rct`**, the program loop cycles faster, which sounds a higher tone. (Note that this program is using a constant for the **`FREQUOT`** command's ***`Freq1`*** argument – it is the more rapid cycling that causes the

tone to sound higher in pitch, but the frequency generated by the BASIC Stamp remains the same.)

Photodiode and the BASIC Stamp as a Digital Light Meter

The previous program was an analog meter, because the audible frequency output was an analog of the light input. Both go up and down in a similar manner. Analog meters are great conveying information directly to our senses. This time, let's look at the actual numbers on the Debug Terminal. This is the digital connection.

√ Enter and download the program DigitalLightMeter.bs2

```
' Applied Sensors – DigitalLightMeter.bs2
' Numerical light levels from the photodiode.
' {$STAMP BS2}
' {$PBASIC 2.5}

rct     VAR     Word            ' Variable for RCTIME.
light   VAR     Word            ' Variable light intensity.

HIGH 6                          ' Discharge the capacitor.

DO

  RCTIME 6, 1, rct              ' Time for volts to fall to 1.3V.
  HIGH 6                        ' Discharge the capacitor.

  light = 65535 / rct           ' Calculating light.

  DEBUG DEC rct, TAB,           ' Display values.
        DEC light, TAB,
        BIN light, CR

  PAUSE 400                     ' Slow things down.

LOOP                            ' Repeat the main loop again.
```

The constant 65535 is arbitrary. The important thing is that the light is proportional to 1/**`rct`**. The exact value of the proportionality constant will be determined when we calibrate the sensor against a light source of known intensity, like the sun, or a standard light bulb.

The Theory Behind it:

For those of you who are interested, the theory here is the same as for Chapter 3. The voltage across the capacitor here must change by 3.7 volts (see Figure 4.8), instead of 1.3 volts (see Figure 3.7). The formula is:

2*rct = C * 3,700,000 / i

(2*rct) is in microseconds, C is in microfarads, and i is in microamps. This shows the theoretical inverse relation between the photocurrent, i, and the rct variable that comes out of the **RCTIME** command. The photocurrent is proportional to the light intensity falling on the photodiode. However, unlike the AD595 temperature probe, where the calibration constant is 1 µA/K, there is not an exact equality between standard units of light and photocurrent. But it doesn't matter. All the constants can be lumped into one that can be determined at the time of calibration:

rct = constant / (light level) or **light level = constant / rct**

- √ Run the program.
- √ Expose the light meter to dim light.
- √ Now expose it to bright light.

Notice how the numbers in the second column in the Debug Terminal increase in bright light. Notice also that the numbers in the first column decrease as the capacitor charges more rapidly from the larger photocurrent.

The final column is included, not so much so you can see the binary value of the light level, but because the length of the binary number, from 1 to 15 binary digits, is proportional to the logarithm of the light level. One digit is added for every doubling of the light intensity. Try taking your light meter from very dim to very bright to see what we mean.

Logarithms are useful for dealing with phenomena that vary over huge ranges. As another example, the VU meter on a stereo sound system shows you a logarithmic graph of sound level. Our ears, like our eyes, can accommodate a tremendous range of sound levels. In technical terms, the length of the binary number shows the integer part of the logarithm to base 2. The same is true for the decimal number in the second column; it adds one digit for each factor of 10 in the increase in light level, but that is harder to perceive. You can try replacing **BIN light** in the **DEBUG** statement with **REP 42\ NCD light**. This prints a string of stars, instead of the actual binary number. The **NCD** operator is the closest thing the BASIC Stamp has to a logarithm function.

Let's look now at the effect of the capacitor on the reading:

- √ Start with a 0.01 µF capacitor in your light meter circuit.
- √ Place your Board in a place where the light level is constant, and note the reading.
- √ Now find the second 0.01 µF capacitor.
- √ Install it on your breadboard in parallel with the first one, as shown in Figure 4-9.
- √ Now (with the same light level as before) observe the **`rct`** and the light level readings.

The **`rct`** reading should be about twice what it was before, and the light reading in the second column should be about 1/2 of what it was. This is because the value of the capacitor enters into the calculation of the calibration constant, as shown in the theory box. If the current from the photodiode is constant, then doubling the capacitor value also doubles the time it takes for the capacitor to charge down to the threshold.

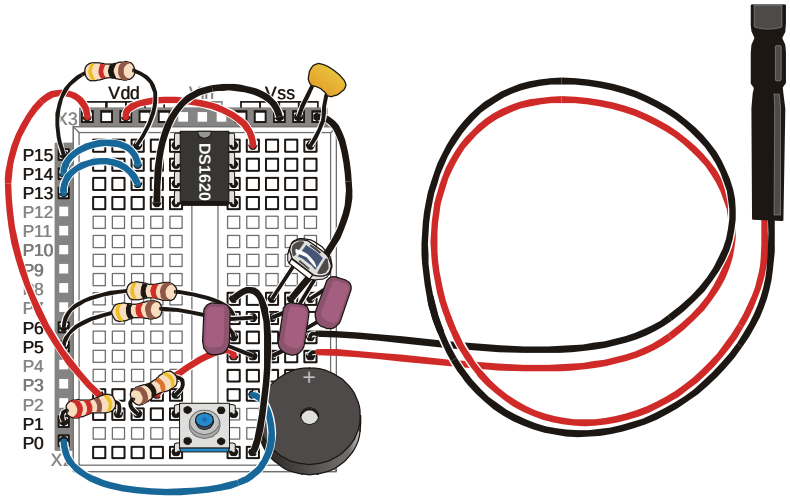


Figure 4-9
Parallel Capacitors

By adding the second capacitor in the circuit in a parallel fashion, the ***`rct`*** *reading will double from what we've measured before. The value of the capacitor enters into the calculation of the calibration constant.*

The capacitors we are using are of a type called "polyester film" capacitors. They are desirable because of their stability. Temperature changes do not affect their capacitance. We need a stable capacitor like that, so that the **`RCTIME`** value will only depend on the current from the photodiode.

- √ Now remove both of the 0.01 µF capacitors, so that there is no capacitor at all in the photodiode circuit. (Be sure you are getting the photodiode capacitors, not the temperature capacitor).
- √ Expose the light meter to very dim light.

It still works! The actual numbers don't mean anything, but you should find that it acts just as if there were a capacitor in the circuit. It will be sensitive to very low levels of light. As a matter of fact, there is a capacitor in the circuit. The input gate on the PIC 16C57 microcontroller on the BASIC Stamp has a built-in capacitance of about 50 pF (picofarads). In addition to that, the wiring of the white block breadboard contributes capacitance. Remember, capacitance exists whenever electrical conductors come close to one another, intended or not. This is called stray capacitance, because it was not really intended to be there as part of your circuit. The combination of the PIC input capacitance and the capacitance of the white block wiring add up to the equivalent of about 250 pF of stray capacitance.

- √ Put in a 100 pF capacitor where you removed the 0.01 µF capacitor.
- √ Check the reading in the same dim light.

The reading will not change too much. The capacitance has only gone from about 250 pF to 350 pF, not from zero to 100 pF as you might have expected.

- √ Put a second 100 pF capacitor in parallel with the first.
- √ Check you reading again.

The readings will not change by a factor of 2, because the capacitance has gone from about 350 pF up to 450 pF, not from 100 pF to 200 pF. Often when things do not turn out the way you expect, it is due to stray circuit elements for which you are not accounting.

- √ Now remove the two 100 pF capacitors.
- √ Restore the single 0.01 µF capacitor in the light meter circuit.
- √ Hold the light meter in a bright light source. If you have it, use a 50-watt R20 spotlight (the kind used in track lighting). If you don't have this type of bulb an alternative would be a 100 watt bulb. The light intensity of such a light source facing the center of the beam at one-meter distance is about 425 lux (40 candlepower).

If you tried to write down the light reading you see in the second column in the Debug Terminal, you might be waiting there forever it to settle on a value. You may find that it fluctuates up and down quite a bit, making it hard to decide what the "reading" really is. These fluctuations come from a couple of different sources. Not the least of them is that the light level really is fluctuating very fast, faster than your eye can perceive it. The intensity of the lamp depends on the power line voltage, so we should really emphasize again that its intensity is *approximately* 425 lux at one meter. The AC line voltage that drives the lamp goes from zero to 170 V, 120 times per second. As this happens, the intensity flickers. The filament in the lamp stays glowing hot, due to its thermal mass, but the output does fluctuate on a time scale of $1/120^{th}$ of a second, about 10 milliseconds. Along comes the photodiode and the BASIC Stamp, to sample the light in less than one millisecond. Sometimes it samples the highs, and sometimes it samples the lows.

Make a rough estimate of your average value:

- √ Look at the readings for a bit to note a minimum value.
  Raw Reading Minimum: _____.
- √ Now scan the readings for a maximum value
  Raw Reading Maximum: _____.
- √ Average them to make an estimate halfway in between.
  Reading Mean: _____.
- √ Now, see if you can find a scale factor so that the program displays the numerical value in standard units of 425 lux, instead of the raw value in arbitrary units, when the sensor is in position in front of the lamp.

  *(Reading Mean) * (scale factor) = 425 lux*

For example, if your Raw Reading Mean happens to be 168, then on a calculator you would multiply that times 2.53. That number is found by using

(scale factor) = 425/ Reading Mean
(scale factor) = 425/168 = 2.53

Subsequently you can move the light meter into an unknown area and find the actual light level there in units of lux, because the light meter is calibrated.

*(new light level in lux) = (new raw reading) * (scale factor)*
*(new light level in lux) = (new raw reading) * 2.53*

The trouble is, the BASIC Stamp (like most microcontrollers) uses integer math. It does not have fractions. Well, not quite true. The BASIC Stamp 2 has a peculiar math operator called */ that is called fractional multiply. The catch is that the fraction has to be one of these specific values: 0, 1/256, 2/256 and so on, up to 256/256 (unity) and so on: 257/256 (one + 1/256th), up to 65535/256 (255 + 255/256ths). All these fractions have a denominator of 256. The factor that goes on the right side of the */ is the numerator of the fraction, and the denominator of 256 is implied. Here are some examples:

```
Y = X */ 256  ' is the same as Y = X, because 256/256=1
Y = X */ 128  ' is the same as Y = X*1/2, because 128/256 = 1/2
Y = X */ 384  ' is the same as Y = X*3/2, because 384/256=3/2
Y = X */ 647  ' is the same as Y = X*647/256 ...
```

It so happens that 647/256 is close to 2.53, which is the scale factor we need. Try it:

647/256 = _____ *close to 2.53?*

*168 * (647/256) = _____* *close to 425?*

Here is an example of how to find the value to put after the */. You have your own Reading Mean, say it is 168. You have the known value of light intensity, nominally 425 lux. Then the factor to put after the */ is 425*256/168 = 647.

*425 * 256 / (Reading Mean) = calibration constant for indoor light*
*425 * 256 / 168 = 647*

The equation to use in the DigitalLightMeter.bs2 program now becomes:

```
light = 65535/rct*/647      ' Computes light from left to right
```

- √ Use your own Reading Mean to calculate your calibration constant for indoor light.
- √ Insert the numerical value you found for your calibration constant for indoor light in the appropriate line of code in the DigitalLightMeter.bs2 program.
- √ Re-download and run the program.
- √ Take a reading from the same light source at the same distance you used to obtain your Minimum and Maximum readings.

The Debug Terminal should display 425 in the second column, when the sensor is placed one meter in front of the calibration source you originally used. Now, as you take the light meter around the room, the reading will be displayed in standard units of lux. This is

a "ballpark" calibration. But it demonstrates the idea, and how the */ operator can help with the math See the *BASIC Stamp Manual* for the full details on PBASIC operators. Calibration of analog sensors often involves multiplying a raw reading times a fraction, so understanding how to use the */ operator can be a great help.

Temperature and Light Meter

Do you still have your temperature sensor probe hooked up? We hope so. If not, hook it back up as shown in Figure 3-9.

- √ Enter the program LightTemperature.bs2.
- √ Enter your own calibration constant for indoor light value in place of 647 in **`Lical CON 647`**.
- √ Enter your own constant, **`Kal`**, for the AD592 temperature sensor, from page 70. That number goes in the place of the 15068 in **`Kal CON 15086`**.

```
' Applied Sensors – LightTemperature.bs2
' Light intensity and temperature meter.
' {$STAMP BS2}
' {$PBASIC 2.5}

Kal     CON     15068           ' Calibration constant for AD592.
Lical   CON     647             ' Calibration constant for photodiode.
                                ' Use Your Own Calibration Constants!!
rct     VAR     Word            ' Variable for RCTIME.
light   VAR     Word            ' Variable light intensity.
TC      VAR     Word            ' Variable for degrees Celsius from AD592.

LOW 5                           ' Discharge AD592 capacitor.
HIGH 6                          ' Discharge photodiode
DO
  RCTIME 5, 0, rct              ' Read temperature probe.
  LOW 5                         ' Discharge AD592 capacitor.

  ' Calculate Celsius.
  TC = Kal / rct * 10 + (Kal // rct * 10 / rct) – 273

  RCTIME 6, 1, rct              ' Read photodiode.
  HIGH 6                        ' Discharge photodiode capacitor.

  light = 65535 / rct */ Lical  ' Calculate lux.

  DEBUG DEC TC, " C", TAB,      ' Display values.
        DEC light, " lux", CR
  PAUSE 400                     ' Slow things down.
LOOP                            ' Repeat the main loop again.
```

Now you have both temperature (first column) and light readings (second column) in the Debug Terminal, with units. That's progress!

- √ Compare the two **RCTIME** instructions that go with the temperature and the light level.
- √ Be sure you understand why the two instructions are different, in relation to how the circuits are set up.
- √ Get this working before you proceed to the next section.

Light and Temperature Logger, using RAM Memory

Now let's make the program store up a bunch of readings. We have already discussed in earlier lessons why data logging is important. It's time to get down and do some of it. The ultimate goal is to log data in the EEPROM memory of the BASIC Stamp. But for now to keep it simple we will log data in the RAM memory of the chip. Recall from Chapter 1 that there are 26 bytes of RAM available for our general purpose use in the BASIC Stamp 2. We will set aside 18 bytes of that for the data log shown in Figure 4-10.

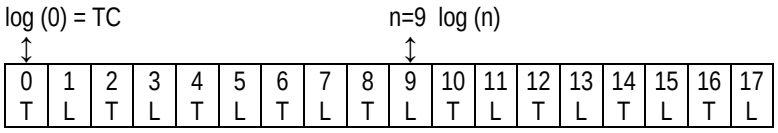


Figure 4-10 Allocation of Memory for Log File

The instruction to reserve 18 bytes in RAM will be:

```
log    VAR  Byte(18)
```

These bytes are like 18 bins in a row, numbered from 0 to 17, and we are going to store temperature values in the even numbered bins and light intensity values in odd numbered bins. In the program, will refer to these bins by using an index inside parentheses:

```
log(0)=TC           ' stores temperature reading in first bin
light=log(9)        ' retrieves light reading from 10th bin.
light=log(n)        ' bin number is held in a variable, n.
                    ' when n=9, retrieves the light value
                    ' from the 10th bin.
```

On signal from the pushbutton, the program will acquire a temperature and a light level reading, and it will store the numbers in the next two available bins. When all the bins are full, the program will make a special protest beep to annunciate the fact, "memory full."

When you make a special long press on the pushbutton, all of the values in the 18 bytes will be displayed on the Debug Terminal. To erase the data and start over, all you have to do is press the reset button. That "re-zeros" all the bins and the pointer.

√ Enter the program LightTemperatureLogger.bs2.

```
' -----[ Title ]-----------------------------------------------------------
' Applied Sensors – LightTemperatureLogger.bs2
' Temperature and light intensity logging in RAM.
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Constants ]-------------------------------------------------------
Kal     CON     15068             ' Calibration constant for AD592.
Lical   CON     647               ' Calibration constant for photodiode.
                                  ' Use Your Own Calibration Constants!!
' -----[ Declarations ]----------------------------------------------------
log     VAR     Byte(18)          ' 18 bytes reserved for the log file.
rct     VAR     Word              ' Variable for RCTIME.
light   VAR     Word              ' Variable light intensity.
TC      VAR     Word              ' Variable for degrees Celsius from AD592.
n       VAR     Byte              ' Counter for the pushbutton.
ptr     VAR     Byte              ' Pointer to next entry in data log file.

' -----[ Initializations ]--------------------------------------------------
OUTS=%0000000001000000            ' Now put in the outs and dirs statements.
     'fedcba9876543210
DIRS=%1111111111111101            ' All are low outputs
             ' ^-----------------   except P6 is high output to discharge C
                  ' ^------------   and P1 is input for pushbutton.
DEBUG CLS
DEBUG "Ready to log data!", CR
FREQOUT 0, 200, 2550
FREQOUT 0, 400, 3400
ptr = 0                           ' Pointer initialization.

' -----[ Main Routine ]-----------------------------------------------------
DO                                ' Main loop.

  DO
  LOOP UNTIL (IN1=0)              ' Loop here until button is pressed

  n = 0                           ' Variable initialization.
  DO                              ' Loop to track pressing time.
    PAUSE 100                     ' Time the button in 0.1 sec increments.
    n = n + 1                     ' Increment counter.
  LOOP UNTIL (IN1=1 OR n>12)      ' Conditions to stop the loop.

  IF (n>=12) THEN                 ' Long click?
```

```
    GOSUB Long_Click
  ELSEIF ptr>17 THEN              ' Short click but memory full?
    GOSUB Memory_Full
  ELSE                            ' Short click instructions.
    GOSUB Get_Data
  ENDIF

LOOP                              ' Jump to the main loop.

' -----[ Subroutines ]----------------------------------------------------
Long_Click:
  FREQOUT 0, 50, 2550             ' Feedback sound.
  FREQOUT 0, 100, 3400

  ' Message on screen print units of measurement
  DEBUG CLS, "logged data!", CR, "degC",TAB,"Lux",CR

  FOR n=0 TO 16 STEP 2            ' Will show 9 records.
    TC = log(n)                   ' Get temperature.
    light = log(n+1) * 2          ' Get light.
    DEBUG DEC TC, TAB,            ' Display.
          DEC light, CR
  NEXT                            ' Next record of 9.

  DO                              ' Do nothing
  LOOP UNTIL (IN1=1)              ' Until button is released

  DEBUG CR, "press RESET to erase data", CR
RETURN

Memory_Full:
  DEBUG CR, "memory full"         ' Message.
  FREQOUT 0, 50, 3400             ' Audio indication.
  FREQOUT 0, 200, 2000, 2100
RETURN

Get_Data:
  FREQOUT 0, 10, 1900             ' Sound to show we got here.

  RCTIME 5, 0, rct                ' Read temperature probe.
  LOW 5                           ' Discharge AD592 capacitor.

  ' Calculate Celsius.
  TC = Kal / rct * 10 + (Kal // rct * 10 / rct) – 273

  log(ptr) = TC                   ' Store temperature.
  ptr = ptr + 1                   ' Point to next bin.

  RCTIME 6, 1, rct                ' Read photodiode.
  HIGH 6                          ' Discharge photodiode capacitor.
```

```
  ' Calculate lux.
  light = 65535 / rct */ Lical

  log(ptr) = light/2 MAX 255      ' Store light intensity/2.
  ptr = ptr + 1                   ' Point to next bin.

  DEBUG DEC TC, " C", TAB,        ' Display values.
        DEC light, " lux", CR

  PAUSE 400                       ' Slow things down
RETURN
```

- √ Run the program.
- √ Press the pushbutton 9 times to collect 9 records of temperature and light level.
- √ Listen for the program to make a "memory full" beep.
- √ Hold down the button for >1.2 seconds to make all 9 of the records display in the Debug Terminal.

You can leave the Debug Terminal open on the screen, and go off and do an experiment to collect 9 records, and then come back to the computer to play them back. When you are ready to start over, press the Reset button on your Board.

Records and fields: Each line, or row, of data is a record. Each reading across is a field. The first field here is temperature in units of degrees Celsius; the second field is light in units of lux. The fields line up in columns. The file here consists of up to 9 records. This terminology is commonly heard in relation to spreadsheets and databases

How LightTemperatureLogger.bs2 Works

We'll comment on the program itself, then move on to some experiments you might want to try.

This program starts off with the **`OUTS`** and the **`DIRS`** statements. They replace the **`HIGH 5`** and **`LOW 6`** that were in previous programs. Note that the sixth position in the **`OUTS`** statement is 1, and the fifth position is zero. That makes P6 high and P5 low, as is required to discharge the two capacitors. The P1 position in the **`DIRS`** statement is a zero, which makes P1 an input. The other pins on the BASIC Stamp are all set to be low outputs, as a matter of good programming practice.

The program starts off with familiar single click and long click code for the pushbutton, so you should recognize the code. When the button is down, there is a race between the timer and the button. If the timer passes the 1.2 second mark, then the program jumps to the **Long_Click** subroutine.

But if the button is released before the 1.2 seconds, then the program goes right into the **Get_Data** subroutine. There, it reads the temperature probe and the light probe just as in program LightTemperature.bs2. Then it puts the temperature value in the bin that is pointed to by the variable **ptr**, then **ptr** is increased by one. Next it puts the value of **light/2** into this next bin. Again, **ptr** is increased by one to point to the next empty bin.

Before jumping to the **Get_Data** subroutine, the program tests the value of the pointer, and jumps to the memory full message if the pointer is greater than 17. The program does not allow itself to collect more data than will fit in the allotted space. You might try taking out this condition, just to observe what sort of error will occur!

Why log the value of light divided by 2, instead of simply light as it is? The bins only hold byte-size quantities, less than or equal to 255. That is fine for Celsius, which will be in the range of 0-100. But the light level might get higher than that. It was calibrated at 425 lux. By dividing by 2, light values of up to 511 lux can be stored. The downside is a loss of resolution, but it is not significant in light of our "ballpark" calibration. When we read the light values from the log, we will multiply them by two to get back the original value. Note that **light/2** is followed by **max 255**. The 255 is a signal that your data is out of range.

The **Long_Click** routine uses a **FOR…NEXT** loop to step through all 9 records.

```
for n=0 to 16 step 2                    ' Will show 9 records
```

The "**step 2**" makes the value of the index, **n**, take on the even values, 0,2,4,6,...,16. That is a total of 9 steps. The value of temperature is read first from **log(0)**, and light from **log(1)**, and these values are displayed in the Debug Terminal. Note that light is multiplied by 2 to reconstruct the original value. Then the **FOR…NEXT** loop bumps up the value of the index to 2, and fetches and displays **log(2)** and **log(3)**, and so on up through **log(16)** and **log(17)**.

Why does the program use **n** as the pointer, instead of **ptr**? First, understand that it doesn't matter what variable goes in parentheses after the **log()**. The only thing that matters is the numerical value of what is in the parentheses. By using "**n**" as the pointer, the value of "**ptr**" is not disturbed. Let's say you have already acquired 5 readings, and then you do the long click to read out what you already have for those 5. You can then continue where you left off and collect readings 6 to 10, and then do the long click to read them all out. It is just a small refinement.

At the end of the playback routine, the program reminds you that you have to press the Reset button on your Board in order to start over with a clean slate.

This program uses every single RAM variable available in the BASIC Stamp. Two bytes each are used for the word variables, **rct**, **TC** and **light**, and one byte each for the indexes **n** and **ptr**, and 18 bytes for the log file. That adds up to 26. If you tried to add one more variable to the program, you would get an error message, "out of variable space" when you ran it.

Experiments with the Data Logger

Verification of $1/r^2$ dependence for the light source

- √ Prepare a string with knots at one meter, 1.5 meters, 2 meters, 2.5 meters and so on up to 4 meters.
- √ Connect the string to the side of the fixture holding the 50 watt, R20 spotlight.
- √ Press Reset on your Board to clear the data log.
- √ Collect data at each distance from the light source, taking care to stay in the center of the beam.
- √ Upload your data to the Debug Terminal.
- √ Graph the readings as a function of position.
- √ Verify that the intensity of light falls off as $1/r^2$.

This is a "ballpark" experiment. Think of some factors that might make the results less than perfect. Don't forget about the fluctuations we discussed earlier!

Investigation of the light distribution from the spotlight

- √ Set up a protractor with the string and spotlight from the previous experiment.
- √ Using your ingenuity, design and execute an experiment that involves holding the light sensor at 10 different angles around the center of the beam from the spotlight.

√ Collect the readings and graph them.

Rate of heating and cooling, timed logging

√ Hold the temperature probe a few inches from the lamp, and press the button at regular 15 second intervals.
√ Then take the sensor away from the light and take 4 more readings at the same rate.
√ Upload the data and graph the temperature as a function of time.

Wouldn't it be nice to have the data logger press the button for you, at regular intervals? Easily done! Change the first nested **DO…LOOP** in the Main Routine section of the LightTemperatureLogger.bs2 program as follows:

```
DO                          ' Changes to program
  n = 0                     ' Initialize the time counter
  DO                        ' Loop here until button or time
    PAUSE 1000              ' One second pacing.
    n = n + 1               ' Count time
  LOOP UNTIL (n=15 OR in1=0)' Get data at 15 second intervals
                            ' Can press button to get data too.
  n = 0                     ' ... and so on as in the original
```

√ Download the modified program.
√ Press Reset on your Board to start your experiment.

The routine still accepts clicks for data logging. The number **n** can be up to 65535 seconds–more than 18 hours between readings–if you care to start a long term experiment!

Alternative Scale for the Light Sensor in Bright Light, Easy Method

If you want to measure brighter light, outdoors in sunlight for example, here is an easy "ballpark" way to go about it.

√ Insert the 0.22 µF capacitor in place of the 0.01 µF capacitor in the photodiode circuit.
√ Put a factor of 22 in the three lines that calculate the light intensity:

```
  light = log(n+1)*44               ' Get light.
```

…and

```
      ' Calculate lux
      light = 65535/rct*/Lical*22
```

…and

```
      log(ptr)= light/44 MAX 255       ' Store light intensity/44
```

This works because the new capacitance is 22 times the old capacitance. The **`rct`** value that used to be produced at 100 lux is now produced at 2200 lux. Recall the effect of doubling the capacitance earlier in this lesson. The old range of measurement was 0 to 512 lux. That is now 0 to 11264 lux.

Alternative Scale for the Light Sensor, Calibration in Full Sun

This will give a light sensor reading in PAR, which are units of micromoles of quanta per square meter per second. If you read the section about light intensity, you know that this is the measurement used for plant growth. This is again a ballpark calibration! Our photodiode does not have the filters that would limit it to the wavelengths best for plant growth.

- √ Put the 0.22 µF capacitor in place of the 0.01 µF capacitor.
- √ Temporarily make the constant **`Lical`** equal to 256 and take out the divide and multiply factors in the light calculations:

```
Lical      CON     256    ' Calibration constant for photodiode
log(ptr) = light MAX 255  ' Store raw light intensity.
light = log(n+1)          ' Get light.
```

Recall that in using the */ operator, */**`256`** is like doing nothing. It is the fraction 256/256, unity. Recall also that when we calculated a calibration constant for units in lux, we assumed a known value of 425 lux at one meter from our spotlight. Now we will assume a known value of 2000 PAR in full direct sunlight.

- √ Place the photodiode sensor directly facing the outdoor sun. Of course, you may have to go outside with your Board running on the battery on a sunny day to do this! You can log data by pressing the button, or by using the timed data mode, depending on how you have your current program configured.
- √ Upload the data to the Debug Terminal. (Note: if the light readings are 255, that means it is out of range, too bright. Put a piece or two of tissue paper over the sensor held with a rubber band, and try again.)
- √ Record the raw value of light shown in the Debug Terminal in the second column.

√ Multiply the raw value times 2000, and then divide by 256. Insert this as the new value for **`Lical`**. For example, given a raw value of 188:

*2000*256/188 = 2723.*

```
Lical     CON    2723          ' Cal. Constant for photodiode
```

√ Change this line of code again so that when it is stored, it is reduced to a byte-sized value by dividing by 10:

```
log(ptr) = light/10 MAX 255  ' Store PAR/10
```

√ And change this line of code so that when it is retrieved for use, it is re-multiplied by 10.

```
light = log(n+1)*10          ' Get PAR
```

Of course, we lose some resolution, because the final digit used will always be a zero regardless of our initial measurement.

√ Also, change the units of measurement in the **`DEBUG CLS`** statement from "**`lux`**" to "**`PAR`**."

Now, when the light sensor is in full sun, it should log a reading of about 2000 PAR.

Use your logger to explore the temperatures and light levels in the outdoor environment!

√ Modify your program to log readings automatically once every two hours (7200 seconds).
√ Come back in 18 hours to see what happened to light levels and temperature while you were away!

Challenge!

1. Do you understand **energy per unit area**? A certain laser puts out a total energy of one milliwatt into a beam with a cross sectional area of one square millimeter. How does the intensity of that light compare with the intensity of sunlight, which is about 1000 watts per square meter?

2. **Assignment: Pluto.** You are planning to visit the planet Pluto, and you want to know how bright the light will be there. Guess–would it be enough to read this page comfortably?

    a. Estimate Pluto's daytime illumination in lux. (The earth is 149,500,000 kilometers from the sun, while Pluto averages 5,920,000,000 kilometers from the sun. On Earth, with the sensor pointed directly at the sun, we measure about 110,000 lux. Approximately what value in lux will you measure when you point the light meter at the sun from Pluto?
    b. Using your calibrated BS2 light meter, find a place in your environment where the ambient light level falling on this page would be comparable to what you will experience outdoors on the sunny side of Pluto.

3. **Reaction Time Tester** Install a light emitting diode and 470 ohm resistor on your Board so that a **`HIGH 9`** instruction can turn it on, and **`LOW 9`** can turn it off. Write a program that does the following to test your reaction time. When you press and hold down the pushbutton, the program waits a random amount of time from 1 to 15 seconds, and then turns on the LED. Then you have to release the pushbutton as fast as you can. The program should use the **`RCTIME`** instruction to measure the time it takes to release the button. Then it displays your reaction time in milliseconds on the Debug Terminal, turns off the LED, and goes back to the top to await another round. The program should test to see if you released the button before the LED goes on, and call you a "cheater" if you do.

4. **Colorimeter** In your kit you have a red and a green light emitting diode. Not only can these diodes emit light, they can also act as photodiodes to receive light. That is, the reverse current in the LED is proportional to light level hitting it. They respond best at the same color they emit. So a red LED responds most to red light, and green to green. Hook up the red and the green LEDs as shown in Figure 4-5 except use BASIC Stamp I/O pins P8 and P9, and use 100 pF capacitors. The current produced by the LEDs is very small. You may reverse

the position of the diodes and the capacitors to get more sensitivity if needed. Write a program that reads the output of both sensors in succession and displays the result on the Debug Terminal. With the sensors in bright white light, the two readings will be different, because two diodes will have different natural sensitivities. Adjust the amount of light reaching the diodes, or adjust the scaling in the program, so that both readings are the same in white light. Then try putting red and green filters in front of the diodes. Have the diodes look at different colors of paper or through different filters, or at the light from a prism.

Chapter 5: The Liquid Environment

The theme of the Liquid Environment experiment is "level and conductivity of water as examples of sensors of the liquid environment." This is a more difficult kind of sensor, but we'll continue with the data logging experiments. The activities associated with this experiment consist of:

- Conductivity using ON-OFF input or **`RCTIME`**
- Adding a 555 oscillator as an input to your Board of Education or HomeWork Board
- Using the 555 oscillator for measurement of conductivity in water using stainless steel probes
- Adding a conductivity measurement to the data logger from Chapter 3

The view of Earth from our moon in 1969 cleared up for once and for all that we live on a water planet. Scientists, farmers, emergency response agencies, the general public, everybody needs to know some "how, when, or why" about water. When is it going to rain? How deep is it? How cold, how hot, how clear, how clean? What minerals, what organic materials does it contain? How fast is it moving? How much is underground? How long has it been there? How much water is in the ice caps, the oceans, the rivers, in living tissue? How do raindrops form? What makes an El Niño event? What happens inside a cloud when it snows? Is there danger of landslides, droughts, floods, or famines? Can a cactus survive here? Can a frog, or a mouse? Can I drink it? Are our wetlands disappearing? Should we care?

So, water will be the third variable we get to in this book. What can you measure about water? I am sure that you can think of hundreds of things right off the bat. We will concentrate on a couple. The first is to detect its presence, or its level. This is the sort of measurement that is needed in order to monitor or control the level of water in a stream, or a fish farm, or a water treatment plant, or when it is a question of when to irrigate a field or a potted plant. The second type of measurement will be the electrical conductivity of water. This is a measurement that is used to detect the presence of salt and minerals in water, and it is also used to assess the quality of drinking water or to study the mixing of fresh and salt water in a tideland or an estuary. There are many kinds of water measurements that require different sensors, like how acidic it might be, or how clear it might be. It is a big field, with lots of research going into the development of sensors that can detect the quality of water.

Measurements in the liquid medium are more problematic than measurements of temperature or light. The sensor probes that detect temperature or light do not actually have to contact the medium electrically. In contrast, wetness sensors often do have to come in direct contact and are subject to corrosion and all sorts of electrical interactions with metals, ions and currents in the liquid medium itself.

Following that observation, we want to make an IMPORTANT precautionary note. Water and electricity don't mix, usually, without planning. Do not, we repeat, do not, by any mistake, spill water on your Board of Education or HomeWork Board! And always be careful about your own safety when working around electricity and water.

Parts Required

The following parts are required for this experiment:

(1) LMC555 CMOS timer
(4) Jumper wires
(2) 0.1 µF capacitor
(1) 100 Ω resistor
(2) 100 kΩ resistor
(1) Conductivity sensor
(1) Cup (not included)
Water
Salt

Building the Circuit

Wetness Alarm

In Chapter 3 you experimented with the conductivity sensor in your kit, as a way of introducing the **`RCTIME`** command. We removed it from your breadboard after that experiment, but now we will be using it again.

√ Build the circuit shown in the schematic (Figure 5-1) and wiring diagram (Figure 5-2.

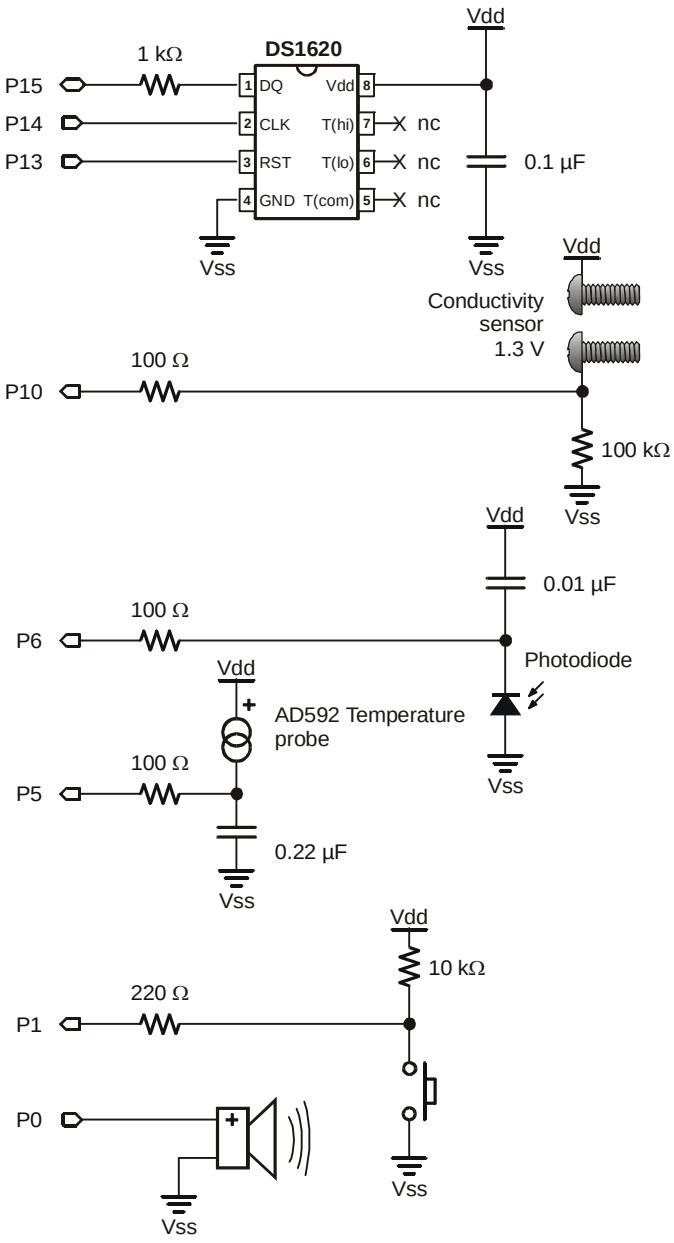


Figure 5-1
Conductivity Sensor Schematic

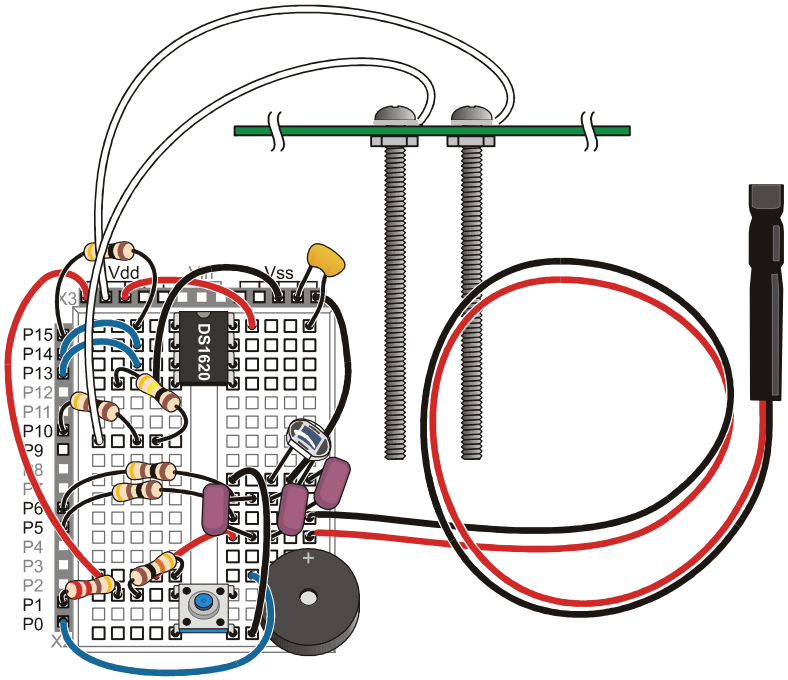


Figure 5-2: Conductivity Sensor Wiring Diagram

√ Now enter the program WetnessAlarm.bs2.

```
' Applied Sensors – WetnessAlarm.bs2
' Wetness alarm.
' {$STAMP BS2}
' {$PBASIC 2.5}

DO
 DEBUG BIN IN10
 IF IN10=1 THEN FREQOUT 0, 6, 2550
 PAUSE 50
LOOP
```

√ Fill your cup with water to within 2 inches of the rim.
√ Leave the conductivity sensor on a non-conducting surface.

- √ Run the program. Did you hear anything?
- √ Now place the sensor over your cup of water, with the two probes (screws) hanging into the water while the cup spanner rests on the rim.
- √ Run the program again. What did you hear this time?

This is your basic water detector and alarm. When you run the program, you will not hear anything until you dip the probes in the water. The program here is like the pushbutton routines you studied in Chapter 2, where pushing down the button made the "cricket" sound. Here, the conductivity sensor's probes in water take on the role of the pushbutton.

- √ Recall the discussion of the conductivity sensor as a variable resistor in Chapter 3.
- √ Explain what is going on by adding remarks to program WetnessAlarm.bs2.
- √ Try replacing the **`IN10=0`**, with **`IN10=1`**, and see what happens.
- √ Think of a situation where that flavor of alarm might be useful.

Why is a 100 kΩ resistor chosen for the circuit? The resistor sets the sensitivity. With higher resistance values, we would run the risk of the circuit saying "wet!" even if a little condensation forms on the wiring. With lower resistance values, that type of error becomes less likely, but on the other hand, the sensor might fail to say "wet" when it should, if the water happens to be especially pure and non-conductive. It comes down to trial and error.

- √ Remove the 100 kΩ resistor from the conductivity sensor circuit.
- √ Replace it with a 1 kΩ resistor.
- √ See how much moisture across the probes is needed to trigger the sensor.

You will find that you have to get the probes much wetter than before, to get the alarm. This kind of wetness alarm is used to train toddlers not to wet their bed. A pad absorbs urine and sets off the alarm. A similar circuit is used to set off the alarm in industrial plants if there is a spill.

Imagine expanding this circuit to control water level. If the water is spilled, we could turn on a pump clean it up. Then when sensor tells us the level is down, we turn the pump off. That is how a sump pump works, to keep water out of someone's basement, or a bilge pump to keep water out of a boat. But we are getting ahead of the game. That is the topic for Chapter 6. At this point the game is to look into quantitative analog measurements, not just "yes/no" but "how much water" and "water of what quality?"

Measurement of Conductance using RCTIME

- √ Remove the 1 kΩ resistor (where the 100 kΩ resistor used to be) from the conductivity sensor circuit.
- √ Replace it with a 0.1 µF capacitor. This is now precisely the circuit shown in Figure 3-4 Figure 3-5.
- √ Enter the program Conductivity.bs2.

```
' Applied Sensors – Conductivity.bs2
' RCTIME measures conductivity.
' {$STAMP BS2}
' {$PBASIC 2.5}

rct     VAR      Word       ' Word variable for RCTIME.
n       VAR      Byte       ' Variable for the bar graph.

LOW 10                      ' Discharge the capacitor to 0 volts.

DO
  RCTIME 10, 0, rct         ' Time for the volts to rise to 1.3V.
  LOW 10                    ' Discharge the capacitor to 0 volts.

  rct = rct – 1             ' Calculate length of bar graph.

  DEBUG DEC rct, TAB,       ' Display ASCII art bar graph.
        REP "*"\NCD rct, CR

  PAUSE 1000                ' Slow it down to 1 per second.
LOOP
```

- √ Run the program. Now you have a digital output that reflects the resistance of the water between the probes.
- √ Hold the conductivity sensor's probes in the water to the depths indicated in Table 5-1. You will have to figure out how to determine the depth in the water. Maybe you can put marks on the side of the cup, or on the sensor itself.
- √ Measure and record your findings in Table 5-1.
- √ Observe how the numbers change as you change the depth of the probe in the water.

Table 5-1: Depth/Resistance Relationship	
Probe Location	**RC-time Reading**
Probe not in water	
Probe tip only touching water	
Probe 1cm in water	
Probe 2cm in water	
Probe 3cm in water	

Do you see a trend?

√ Try repeating the measurements a few times, and write down the numbers.
√ Allow the probe to sit at depth for a minute or two in the water.

Are the readings repeatable? That is, do you get the same result each time?

Here are a couple of program notes. First, why include the formula, **`rct=rct-1`**? The reason is to make the graph look better. As you pull the probe out of the water, the number **`rct`** gets larger and larger, but suddenly when the probe leaves the water, **`rct`** suddenly goes back to zero. That is due to a peculiarity of the **`RCTIME`** command, which returns the value "**`0`**" as a kind of error message when it runs over its maximum value. By putting in the formula **`rct=rct-1`**, we are in effect making "aces high."

When you subtract 1 from zero in integer arithmetic, you get 65535. When you use microcontrollers, or really, when you program any computer, you often have to compensate for the little peculiarities of the commands available to you.

How about the graph? It uses a **`DEBUG`** modifier:

```
REP "*"\NCD rct
```

You know **`rct`** is the variable. **`REP`** is short for "repeat." It repeats printing the character "*" on the Debug Terminal, and the number of times it repeats is given by the number after the "\" For example:

```
DEBUG REP "*"\12
```

would print 12 stars in a row in the Debug Terminal. You could also program that as:

```
DEBUG "*************"
```

But doing it with **REP** is more concise. And the number after the "\" can be a variable, which can be very useful. Here the variable after the "\" is actually an expression that results in a number. The expression is **NCD rct**. The **NCD** is a math operator unique to the BASIC Stamp. The result is just the length of the number **rct** in binary form. For example, if **rct=35** in decimal, its binary form is **rct=%100011**. The length of that binary number is 6 binary digits. Run that through the **NCD** operator, and **NCD rct** returns the value, 6, and six stars are printed on the Debug Terminal. Maybe you will never have to use that command! But there it is, to add to your bag of programming tricks. The number of stars increases by one for each doubling of the value of **rct.**

We will take one more measurement series for the above table, with the electrical connection to the conductivity sensor reversed:

- √ Take out the probe wire that connects to Vdd.
- √ Place it in the breadboard where the other probe wire normally belongs.
- √ Connect the probe wire you just removed from the breadboard to Vdd instead.
- √ Make the series of measurements again as listed in Table 5-1.

You will probably find that the numbers are a little different. The difference is due to what is going on in the liquid medium, as electricity passes through it from one probe to the other.

The effect you are seeing is called "polarization." Chemical reactions are actually changing the electrode. This is not a big deal for the simple on-off kind of sensor, but it is a disaster for quantitative measurements. Polarization occurs because the electrical current is constantly moving in one direction through the probe. It is direct current, DC. The simple cure is to drive the sensor with current first one direction and then other. That is alternating current, AC. Many of these chemical reactions are reversible, up to a point, so the alternating current leads to much more stable readings. The BASIC Stamp can't supply the necessary AC signal. An external chip can help here.

Measurement of Conductance using the 555 Timer IC

- √ Remove the conductivity sensor circuit.

- √ From the Parts Required list at the beginning of the chapter, gather a 100 Ω resistor, a 100 kΩ resistor, a 0.1 µF capacitor, the jumper wires, and the 555 timer IC chip.
- √ Install the 555 timer IC circuit shown in the schematic (Figure 5-3).
- √ Be attentive to follow the parts placement in the wiring diagram (Figure 5-4) since the wiring is getting tight!

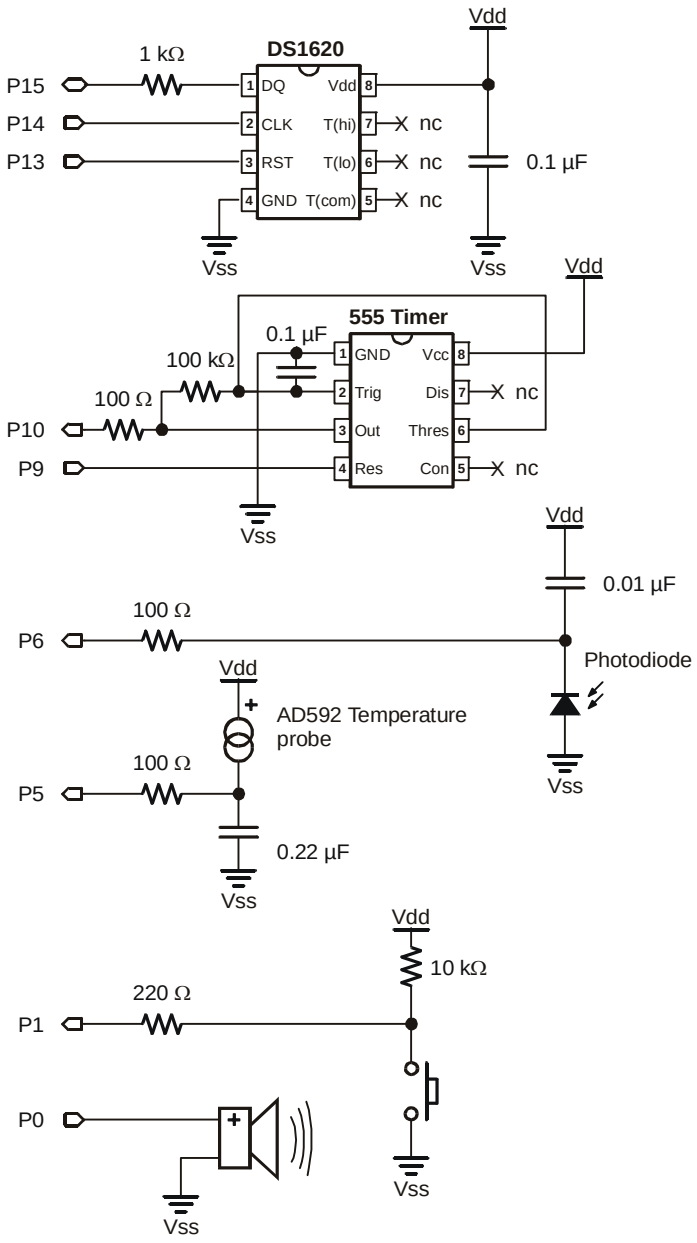


Figure 5-3
Conductance Using the 555 Timer Schematic

Space on the breadboard is becoming very limited. Follow the wiring diagram below exactly to make the project fit on the breadboard.

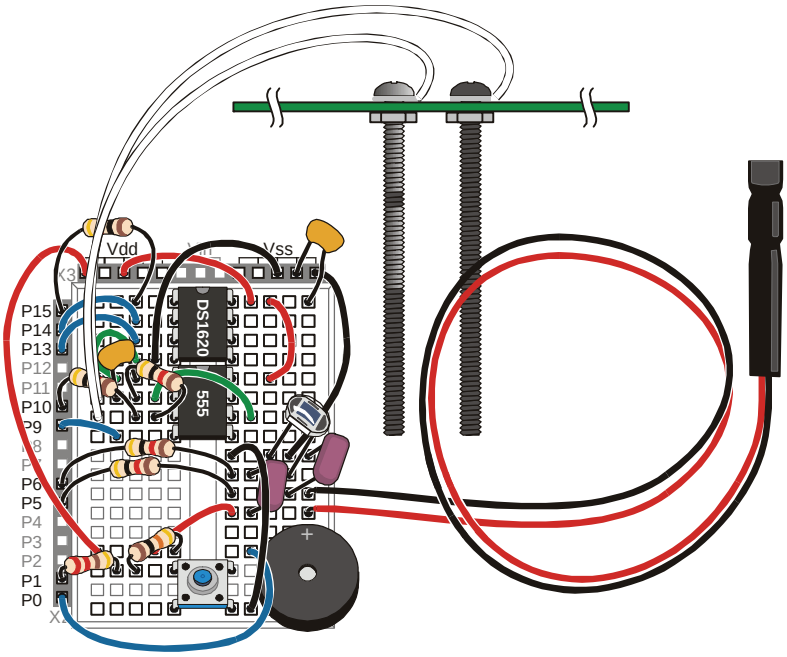


Figure 5-4
Conductance Using the 555 Timer IC Wiring Diagram

This circuit is an astable multivibrator. That is terminology held over from the early days of electronics. All it means is that the circuit output (pin 3 of the 555 timer) alternates from high to low repeatedly on its own. The resistor from pin 3 to pin 2, along with the capacitor from pin 2 to pin 1, determine the frequency of oscillation. P10 on the BASIC Stamp will be configured as an input so that it can monitor the frequency produced by the 555 timer. I/O pin 9 will be configured as an output that can turn the 555 timer ON or OFF. When P9 is high, the 555 timer is ON. There are several ways to hook up the 555 timer chip, in fact, there are entire books devoted to nothing but the 555 timer!

√ Enter and run the Test555.bs2 program:

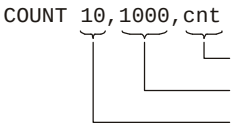
```
' Applied Sensors – Test555.bs2
' Test of the 555 oscillator.
' {$STAMP BS2}
' {$PBASIC 2.5}

cnt    VAR   Word       ' Word variable for count.

HIGH 9                  ' Turn on the 555 oscillation.

DO
  COUNT 10, 1000, cnt   ' Count for one second.
  DEBUG DEC cnt, CR     ' Show values.
LOOP
```

Here are the arguments of the BASIC Stamp 2 **`COUNT`** command:



RAM variable for the result of counting.
Count this long in milliseconds, duration.
Pin to use for counting, a BASIC Stamp input.

√ When the ***`Duration`*** argument is 1000, the reading you see in the Debug Terminal should be about 75. Write down your reading.

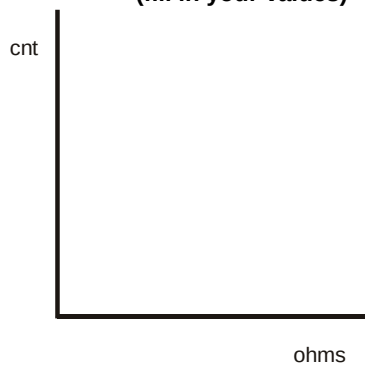
`cnt` = _____ for **`COUNT`** ***`Duration`*** of 1000, 100 kΩ resistor, 0.1 µF capacitor

√ Now place a second 100 kΩ resistor in parallel with the first, side by side in your breadboard. The parallel combination of two 100 kΩ resistors is a resistance of 50 kΩ (the series combination gives 200 kΩ). The frequency should be about double.
√ Enter the value in Table 5-2 below.
√ Now put the two 100 kΩ resistors in series from pin 2 to pin 3 of the 555. The frequency should now by 1/2 of the original value.
√ Also record this value in Table 5-2.
√ Calculate the value of 1/R, which is called the conductance, and has units of siemens (an older, more colorful term for conductance is the mho, or ohm spelled backwards: 1 siemen = 1 mho).
√ Make a quick graph of the frequency versus resistance, and a graph of frequency vs. conductivity in the space provided below.

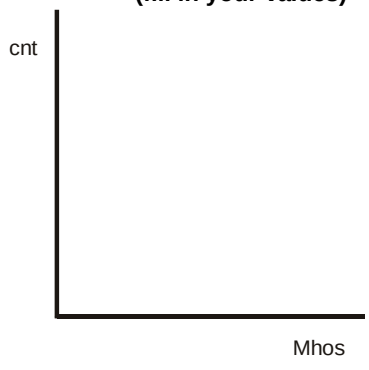
Table 5-2: 555 Timer Test		
R, resistance, ohms	**G, conductance, mho (= 1/R)**	**cnt, from BASIC Stamp COUNT command**
50 k		
100 k		
200 k		

You'll need to calculate G = 1/R, and measure **`cnt`** from Test555.bs2

Frequency versus Resistance
(fill in your values)



Frequency versus Conductivity
(fill in your values)



√ Observe which graph is more linear.

Why do we have to have to talk about both resistance and conductance, if one is just the inverse of the other? Get used to it! There is a separate term for the inverse of everything in electronics. In electronics it is more common to talk about resistance. However, in materials science, chemistry, and environmental instrumentation, it is more common to hear the term conductance. Maybe that is because in the liquid medium there are many, many different paths between any two points. The different paths are like many resistors in parallel, and changes in the liquid medium tend to change those parallel elements. So it is easier to talk about conductance, where conductances add in parallel. Figure 5-5 shows how resistors are placed in series and parallel to measure resistance I and conductance (G).

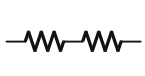


Figure 5-5
Resistance and Conductance Formulas

R + R ohms	1 / (1/R + 1/R) ohms
1 / (1/G + 1/G) mhos	G + G mhos

Resistance R, and conductance G, of series and parallel resistors (conductance). The formula for parallel resistors is easier in terms of conductance.

- √ Restore your circuit to the original single 100 kΩ resistor.
- √ Modify the program Test555.bs2 by changing the ***Duration*** argument from 1000 ms to 500 ms.
- √ Run the modified program and observe the reading.
- √ Now repeat with a ***Duration*** of 2000 milliseconds.

Did you see that the reading changes by a factor of close to 2 each way?

Let's look at this another way. Just above you wrote down a value of **cnt,** the value that came out when a 100 kΩ resistor and a 0.1 µF capacitor were in the circuit, and the duration argument was equal to 1000 in the **COUNT** command. What ***Duration*** argument would you have to put in the **COUNT** command in order to make the reading come out at 100 instead of at 75 (or your reading _____)? Well, you just have to make the ***Duration*** argument proportionately longer. A longer ***Duration*** gives you a higher count, right?

- √ So calculate:

Duration = 1000 * (100/75) = 1333, but you need to use your own reading:

Duration = 1000 * (100/_____) =_____ (use your reading to obtain the result)

That is your Duration calibration constant. You will need it again.

- √ Modify Test555.bs2 again, using your Duration calibration constant as the **COUNT** command's ***Duration*** argument.
- √ Run the program and observe the results.

Now when you run the program with 10^{-5} siemens (100 kΩ) in the circuit, it should display 100 in the Debug Terminal instead of the original value. The point of this is that

the ***Duration*** argument in the **COUNT** command can be used to scale the result, so that it appears directly in siemens. We want you to think quantitatively!

Now let's change the **DEBUG** command so that it shows the units. And while we're at it we might as well calculate the resistance in ohms and display that too. The resulting new program is given below.

√ Enter the program Calibrate555.bs2.


```
' Applied Sensors – Calibrate555.bs2
' Calibration of the 555 oscillator.
' {$STAMP BS2}
' {$PBASIC 2.5}

cnt    VAR    Word      ' Word variable for count.
R      VAR    Word      ' Word variable for resistance.

HIGH 9                  ' Turn on the 555 oscillation.

DO

  COUNT 10, 1333, cnt   ' Count for about one second.
          '^^^^--------  You use your own constant here!!!
  R = 50000/cnt*2       ' Calculate resistance R = 1/G.

  DEBUG DEC cnt, "E-7", ' Show values.
        TAB, DEC R, "00", CR

LOOP
```

√ Run the program and observe the results.

The display should now show 100E-7 (for $100*10^{-7}$ siemens), and in the second column it should show 100000 (for Ω).

√ Verify the calibration by putting the extra 100 kΩ resistor in parallel to with the first again to get a resistance value of 50 kΩ back in the circuit.
√ Run the program again and observe the change in your readings.

The display should show 200E-7 siemens and 50000 Ω. Notice that the resistance reading appends two zeros to the value of R, to make it come out in Ω.

Theory behind using a 555 timer to measure conductivity:

There are many references that talk about how the 555 timer circuit works, and how to apply it, even whole books on nothing but the 555 timer! The important point for measuring conductivity is that the current through the resistor in this circuit goes back and forth, first one direction and then the other direction, equal and opposite. As we discussed above, that is what we are looking for in a probe to put in the liquid medium. There is a balance of current flow in each direction, to forestall corrosion, plating, and polarization. The theory for the 555 timer is very similar to the theory for **`RCTIME`**, but we don't want to get into it here. The equation for the output frequency is approximately: f = 3/4*R*C. With R = 100000 Ω and C=0.1µF, that comes out to 75 hertz.

Conductance in Water

Now it is time to dive in!

- √ Gather up a cup full of water if you don't have one now, and keep a spoon and a couple of pinches of table salt handy.
- √ Re-install the conductivity sensor. Figure 5-6 shows how to connect the probe to the 555 timer on your breadboard.
- √ Leave the Duration calibration constant you calculated for your setup in place for the **`COUNT`** command's ***`Duration`*** argument.

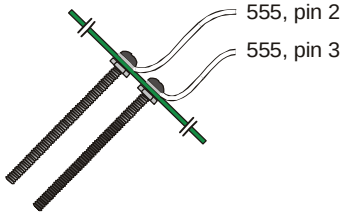


Figure 5-6
Conductivity Sensor

Replace the 100 kΩ resistor with the conductivity sensor.

- √ With the conductivity sensor in the circuit, run program Calibrate555.bs2 again.
- √ Place your fingers across the probe to see if you get a reading in siemens and in Ω.
- √ Wet your fingers and take another reading. How do you explain the result in terms of conductances?
- √ Without touching the probe with your fingers, touch the probe to the leads of the 100 kΩ resistor, to confirm that the meter still reads correctly. It should read 100E-7 siemens, 100 kΩ.
- √ Immerse the probes in the center of the cup of tap water to a depth of 4 cm.
- √ Read the conductance from the Debug Terminal.

- √ Record the reading in Table **5-3**.
- √ Repeat and record the conductance measurements at depths of 3, 2, and 1 cm, completing Table 5-3.

Table 5-3: Distilled or Tap Water Conductance vs. Depth

Water level	Conductance
1 cm	
2 cm	
3 cm	
4 cm	

- √ Keeping the sensor at one constant depth, move it over until it is near the side of the cup.
- √ Look for a change in the conductivity reading.

What happens to the reading, and can you explain why in terms of conductances in parallel?

- √ Move the sensor back to the center of the cup.
- √ Note the reading.
- √ Bring a metal object such as the back of a metal spoon up near the sensor probe.
- √ Look for a change in the conductivity reading.

How does that affect the reading, and why is it different from bringing the probe near the side of the cup?

- √ Drop a pinch of salt crystals into the cup, but do not stir.
- √ Look at the conductivity reading.
- √ Stir to disperse and dissolve the salt crystals.
- √ Look at the reading again. Surprised?
- √ Hold the probes in the saltwater at a depth of one centimeter.
- √ Observe and record the reading in Table 5-2.
- √ Repeat the measurements for each depth until Table 5-2 is completed.

Can you see the pattern emerging? How do you think this may be useful?

Table 5-4: 555 Timer Test	
Pinch of Salt Dissolved in Water, Conductance versus Depth	
Water level	**Conductance**
1 cm	
2 cm	
3 cm	
4 cm	

Conductivity is often used to determine the salinity of water (how much salt it contains per unit volume), or more generally, how much mineral content is present. If you used tap water, you might try the experiment again using distilled water. By holding the depth constant, you could use this probe to measure salinity, which is closely related, through a rather complicated formula, to conductivity.

Note that in each case the conductance is proportional to depth, either in the tap water, or in the salt water. You can use this device to measure the depth of water.

However, the two measurements are confounded. To use the device to measure depth, you have to be sure that the amount of salt in the water is going to be constant, or you have to acquire a separate measurement of conductivity in order to compensate. On the other hand, in order to measure conductance, you have to be careful to keep the sensor at a constant depth.

Design of professional instruments is largely concerned with overcoming or compensating for the effects of confounding variables. For conductivity measurements, great care is taken to confine the solution to a fixed volume, and to use stable electrode materials, and to monitor the temperature at the point of measurement. Professional water depth meters are seldom based on the conductance principle because of these difficulties.

The conductivity of water in the natural environment spans many orders of magnitude. Conductivity is measured in units of siemens per centimeter. Ocean water may have a conductivity of 50000 siemens per cm, whereas pure distilled water may have a conductivity of mere microsiemens per cm.

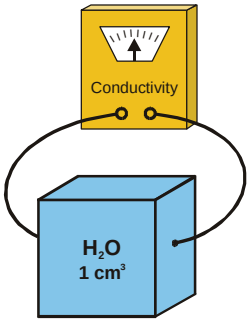


Figure 5-7
Conductivity Measurement

Conductivity is measured, in theory, with a block of material, 1 centimeter on a side. Electrodes are fastened to two opposite faces of the block, and the conductance is measured. Since the sample block is one cm long, the conductivity is in units of siemens per cm. Conductivity measurements are reported as if they were made in this setup, but the actual measuring setup is a lot more complicated!

To calibrate a conductivity instrument, you would need to have a standard salt solution that lets you make the leap from conductance (measured by your BASIC Stamp across its particular sensor) to conductivity (a property of the water being measured, independent of peculiarities of the measuring instrument). You have to find a constant, called the "cell constant" or the "instrument constant" that accounts for the actual geometry of the probe you are using. It is a constant of proportionality. We leave the calibration to the exercises that follow this experiment.

What is the difference between conductance and conductivity?

Conductivity is a property of materials. Materials that conduct electricity well, like metals, have a high conductivity, while insulators have low conductivity. If you take a thin wire one meter long, it will have a certain resistance from end to end, and its conductance will be simply one over the resistance. A thicker wire made of the same material and of the same length will have a lower resistance and a higher conductance. The conductivity of the wire is the same in both cases. It is a property of the material itself, not the quantity of the material.

Data Logging Continued: Drying of Soil

In the natural world, the phenomenon of evaporation is very important. Water evaporates from the soil, and water is also lost in the transpiration of plants. The rate of evaporation and other mechanisms of water loss depend on the solar intensity, the wind speed, the temperature, the humidity and the ground cover. You can use your BASIC Stamp-controlled data logger to study evaporation. On a practical basis, you can use your logger to tell you when to water your houseplants or your garden!

√ Enter the program RAMDataLogger.bs2 (which is a modification of LightTemperatureLogger.bs2) as shown below.

```
' -----[ Title ]-----------------------------------------------------------
' Applied Sensors – RAMDataLogger.bs2
' Temperature and light intensity logging in RAM.
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Constants ]-------------------------------------------------------
Kal        CON     15068          ' Calibration constant for AD592.
Lical      CON     647            ' Calibration constant for photodiode.
CondCal    CON     1333           ' Calibration constant for conductance.
' Use Your Own Calibration Constants!!
Interval   CON     10             ' Get data once per ten seconds
                                  ' you can choose whatever interval you need.
' -----[ Declarations ]----------------------------------------------------
log        VAR      Byte(18)      ' 18 bytes reserved for the log file.
rct        VAR      Word          ' Variable for RCTIME.
light      VAR      Word          ' Variable light intensity.
TC         VAR      Word          ' Variable for degrees Celsius from AD592.
n          VAR      Byte          ' Counter for the pushbutton.
ptr        VAR      Byte          ' Pointer to next entry in data log file.
cnt        VAR      rct           ' Variable for conductivity  ALIAS of rct.

' -----[ Initializations ]--------------------------------------------------
OUTS=%0000000001000000            ' Now put in the OUTS and DIRS statements.
     'fedcba9876543210
DIRS=%1111101111111101            ' All are low outputs.
     '     ^   ^    ^-------------  except P1 is input for pushbutton
     '     ^   ^------------------  P6 is high output to discharge C
     '     ^----------------------  and P10 is input for conductivity (555).

ptr = 0                           ' Pointer initialization.
DEBUG CLS
DEBUG "Ready to log data!", CR,
      "degC", TAB,                ' Display units.
      "lux", TAB, "siemens", CR
FREQOUT 0, 200, 2550              ' Initialization sound.
FREQOUT 0, 400, 3400

' -----[ Main Routine ]----------------------------------------------------
DO                                ' Main loop.

  n = 0                           ' Initialize counter.
  DO                              ' Loop here until button or time.
    PAUSE 1000                    ' One second pacing.
    n = n + 1                     ' Increment counter.
  LOOP UNTIL (IN1=0 OR n=Interval)' Time out or button pressed.

  FREQOUT 0, 5, 3400              ' Tick for button down or time out.

  n = 0                           ' Initialize counter.
  DO                              ' Loop to track pressing time.
```

```
    PAUSE 100                     ' Time the button in 0.1 sec increments.
    n = n + 1                     ' Increment counter.
  LOOP UNTIL (IN1=1 OR n>12)      ' Conditions to stop the loop.

  IF (n>=12) THEN                 ' *Long click?
    GOSUB Long_Click
  ELSEIF ptr>17 THEN              ' *Short click or time out
    GOSUB Memory_Full             ' but memory full?
  ELSE                            ' *Short click or time out?
    GOSUB Get_Data
  ENDIF

LOOP                              ' Jump to the main loop.

' -----[ Subroutines ]------------------------------------------------------
Long_Click:
  FREQOUT 0, 50, 2550             ' Feedback sound.
  FREQOUT 0, 100, 3400

  ' Message on screen print units of measurement
  DEBUG CLS, "logged data!", CR, "degC", TAB,"Lux", TAB, "mho", CR

  FOR n=0 TO 15 STEP 3            ' Will show 6 records.
    TC = log(n)                   ' Get temperature.
    light = log(n+1) * 2          ' Get light.
    cnt = log(n+2)                ' Get conductivity.
    DEBUG DEC TC, TAB,            ' Display.
          DEC light, TAB,
          DEC cnt, CR
  NEXT                            ' Next record.

  DO                              ' Do nothing
  LOOP UNTIL (IN1=1)              ' until button is released.

  DEBUG CR, "press RESET to erase data", CR
RETURN

Memory_Full:
  DEBUG CR, "memory full"         ' Message.
  FREQOUT 0, 50, 3400             ' Audio indication.
  FREQOUT 0, 200, 2000, 2100
RETURN

Get_Data:
  FREQOUT 0, 10, 1900             ' Sound to show we got here.

  ' Temperature.
  RCTIME 5, 0, rct                ' Read temperature probe.
  LOW 5                           ' Discharge AD592 capacitor.

  ' Calculate Celsius.
```

```
  TC = Kal / rct * 10 + (Kal // rct * 10 / rct) – 273

  log(ptr) = TC                   ' Store temperature.
  ptr = ptr + 1                   ' Point to next bin.

  ' Light.
  RCTIME 6, 1, rct                ' Read photodiode.
  HIGH 6                          ' Discharge photodiode capacitor.

  ' Calculate lux.
  light = 65535 / rct */ Lical

  log(ptr) = light/2 MAX 255      ' Store light intensity/2.
  ptr = ptr + 1                   ' Point to next bin.

  ' Conductivity.
  HIGH 9                          ' Turn on the 555.
  PAUSE 100                       ' Delay for it to get up to speed.
  COUNT 10, CondCal, cnt          ' Count the frequency.
                                  ' ← use your scale factor!!!
  LOW 9                           ' Turn off the 555.

  log(ptr) = cnt                  ' Store the conductivity.
  ptr = ptr + 1                   ' Point to next bin.

  ' Display values.
  DEBUG DEC TC, TAB, DEC light,
        TAB, DEC cnt, "E-7", CR
RETURN
```

This adds conductivity to the mix. Now you will be able to store 6 readings in memory and read them out later. The total number of bytes available for logging is 18. With 3 per record, that means we are limited to 6 records total.

Recall from Chapter 4 that we ran out of variables for that program. We used 18 bytes for the data log file, and the rest of the available variables for the program. In this program we reuse the variable **`rct`** for the **`COUNT`** function for conductivity. We call it **`cnt`**, and define it in a variable alias statement at the top of the program. That means the **`cnt`** and **`rct`** are really the same physical variable. Changing one of the variables changes them both, simply because they are physically the same.

There is nothing unusual about this program. It is a straightforward expansion of the one from Chapter 4. It seems to be getting longer, but each piece has its special part to play.

In this program the **OUTS** and **DIRS** statements are modified to take account of the new pins. P10 is an input for the **COUNT** operations. P9 is an output to turn the 555 timer on and off. The new quantity **Interval** is a constant for the number of seconds between readings (0-65535). The new program has the code necessary for the conductivity probe. We have made a modification to the **Long_Click** routine to play back the conductivity data

- √ Get this program running, and tested at 10 second intervals.
- √ Insert the temperature probe and the conductivity sensor in a cup or flower pot full of vermiculite or other potting medium.
- √ Place the whole set-up in the sun, with the light sensor set for outdoor light.
- √ Leave it for 6 hours.
- √ Look at your data. Is the time interval appropriate?

If you are doing this in a classroom, different groups can do the experiment with variations. For example, some in the sun, some in the shade, some with ventilation, some not. Use a real potted plant. Experiment – that's the way you learn how to use microcontrollers!

Alternatively, for a quicker experiment, drape a wet paper towel over the conductivity sensor probe. Follow and log the conductivity and temperature of the paper towel as it dries out.

Additional Experiments to Try

Condensation Sensor

- √ Press a piece of dry plastic or glass up against the screws on the sensor. Note the conductivity reading.
- √ Place the glass or plastic surface near your face, and exhale on it slowly to fog it up.
- √ Retake the conductivity reading.

When the glass is dry, it is an insulator, and the conductivity is low. But if you breathe heavily on the glass, it will deposit condensation that will conduct electricity. Depending on the temperature and humidity, you may have to chill the surface before condensation will form. This kind of sensor is useful in agriculture, where condensation forming on

leaves of plants can lead to infection by fungus and scab diseases.

Humidity Sensor

- √ Find a length of heavy thread or light-weight cotton twine.
- √ Soak it in salt water (a pinch of salt will do).
- √ Wrap it around the stainless steel screws of the conductivity sensor.
- √ Dry it off with a hair dryer and observe the conductivity while you do this.
- √ Let the string come back to atmospheric moisture.
- √ Exhale slowly on it, and see if the conductivity will increase.

The NaCl has a transition point at about 75% humidity where it picks up lots of water. Below 75% humidity, NaCl tends to give up water to the atmosphere. Above 75% humidity, NaCl tends to absorb moisture from the atmosphere. The conductivity follows along. Different salts respond at different humidity levels.

Surface Explorer

- √ Create a shallow pool of water in a large shallow plastic or glass tray or dish.
- √ Carefully put some pieces of rock salt in the dish, but do not stir.
- √ Use the conductivity sensor's probe to explore the diffusion of salt into your "aquifer."

Incursion of salt water into fresh water marshes and aquifers is a big problem in areas where the fresh water is drawn off for uses in industry and agriculture.

- √ Scribble in an area on a piece of paper with heavy pencil or artist's charcoal.
- √ Drag the conductivity sensor across the surface.

The variation in conductivity can be used to explore the thickness and resistivity of the pattern and the density of the markings.

Temperature Dependence of Conductivity

The conductivity of salt solutions in water depends on the type of salt, and also on temperature.

- √ Dissolve some table salt in a cup of water.
- √ Log both temperature and conductance at a constant depth as you heat the water. Graph the result of your experiment.
- √ Repeat the same experiment with a different type of salt (say KOH).
- √ Repeat the experiment again with a weak acid solution made with vinegar.

You will find that each solution has its own characteristic temperature dependence. Commercial conductivity sensors always measure both temperature and conductivity. From temperature and conductivity, they can then calculate the concentration of the salt. Can you figure out how to do that calculation? Chemistry reference books, like the *Handbook of Chemistry and Physics*, contain this kind of information. Look and see. You have to know in advance what type of salt or salt mixture is in solution.

Quantitative Calibration of the Conductivity Sensor, using a Standard Solution

In order to calibrate this sensor to measure conductivity (property of the material) instead of conductance (an electrical quantity), you would have to prepare a standard solution that has a known conductivity. Such standard solutions can be purchased, or you can make them yourself in the classroom, by adding a known amount of potassium chloride (KCl) to a known amount of water. Tables of conductivity are found in chemistry or in water quality handbooks, or in references such as the *Handbook of Chemistry and Physics.*

Once you have the standard solution, you measure its conductance with your BASIC Stamp-controlled instrument. That gives you a constant of proportionality between your conductance reading and the conductivity of the solution. This constant is called the instrument constant. It has to do with the geometry of the electrodes and the cup. You also measure the temperature of the solution. With this information in hand you can proceed to measure the conductivity (and the concentration) of unknown water samples.

Ground Loop Error

- √ Connect a long piece of hookup wire to Vss (zero volts) on your Board of Education or Homework Board.
- √ Drop the free bare end of the wire into a cup of water where the conductivity probe is operating and showing its readings in the Debug Terminal.
- √ Watch as you do this to see if the reading changes.

This is due to the extra ground path provided by the wire. This kind of situation is common in large instrumentation systems, say, in a fish farm or in an industrial plant. But it is sometimes hard to track down where the interaction is coming from. There can be unplanned connections between points in the system. To avoid this problem, engineers often design "isolated" sensors, which means that signals are passed across an optical link or other barrier of that sort, so that there will be no direct electrical connection. This is also extremely important in situations where safety and shock hazard are an issue, such as in medical instrumentation.

Environmental Explorer

This experiment requires you to measure the conductivity of several samples of the same volume of water collected from various places. Be prepared to do some fieldwork to collect your samples!

- √ Measure the conductivity of a sample of distilled water at a depth of 2 cm.
- √ Record your reading.
- √ Repeat with the same volume of tap water.
- √ Repeat with the same volume of pond or lake water.
- √ Repeat with the same volume of ocean water (or beg a sample from the owner of a salt-water aquarium!).
- √ Rank your samples by conductivity.
- √ Begin with your sample of distilled water, and add ¼ teaspoon of salt at a time, stirring and then measuring the conductivity between each addition.
- √ Determine how much salt you had to add to your distilled water to give it the same conductivity as your ocean water sample.
- √ Use this information (the volume of your water sample, and the volume of salt used) to calculate the volume of salt you have to add to one liter of distilled water to make it as salty as the ocean.

Are you surprised at your answer?

Challenge!

1. Write a program that counts the number of times you press the pushbutton in 5 seconds. The BASIC Stamp should sound a "start" tone, and then count the number of button presses, play a "finish" tone, display the result in the Debug Terminal, pause for 3 seconds, and then do it again.

2. Install a red and a green LED on your breadboard, so that your BASIC Stamp can use P7 and P8 to turn them on or off. Modify the program that measures temperature, light and conductivity as follows:

    a. It should turn on the green LED if the measurements are all within normal operating range (you decide what that range is).
    b. If they go outside the normal range, the green LED should turn off, and the red LED should turn on.
    c. If they then return to normal, the green LED should come back on, but the red light should stay on to show that there has been a "problem."
    d. If the readings get way out of range, turn on the "alarm siren."

3. With the circuit of Figure 5-3, the frequency of oscillation is proportional to 1/RC, where R is the resistance, and C is the capacitance. In your kit, you have 2 of the 100 kΩ resistors and 2 of the 0.1 µF capacitors. You can put resistors in series to make 200 kΩ, and in parallel to make 50 kΩ. You can put two of the capacitors in series to make 0.05 µF, and in parallel to make 0.2 µF. Write a program to show you the frequency in Hertz for each value of resistance and capacitance in the following table:

Frequencies from 555 timer	0.05 µF	0.1 µF	0.2 µF
50 KΩ-20E-6 mho			
100 KΩ-10E-6 mho			
200 KΩ-5E-6 mho			

This will test your understanding of how the white block is connected underneath! Does it seem to be true that frequency = constant/RC? Graph frequency vs. conductivity. Graph frequency vs. resistance. Which is linear?

Chapter 6: Measurement and Control

Editor's Note: T*he low-voltage pump used to develop the activities in Chapter 6 is no longer available. Though we no longer supply the pump circuit components in the Applied Sensors Parts Kit v2.0, we have kept Chapter 6 in the book for your reference and adaptation to commercially available low-voltage pumps.*

The theme of the Measurement and Control experiment is that the BASIC Stamp microcontroller can do both measurement and control, closing the feedback loop. The final activities in this text will explore:

- Feedback to control the level of water in a cup using a pump, and a conductivity sensor as the level detector
- Simultaneous measurement and control of 4 variables

Measurement and data logging are often combined with control. Not satisfied to simply sit there and watch, your BASIC Stamp-controlled instrument reaches out and does something that affects the conditions in the outside world. It might open a door in response to an approaching pedestrian. Or it might function as a thermostat, to control a heater or a fan when the temperature gets too hot or too cold. In industry, on the farm, in public works, in scientific research, all manner of processes need to be controlled and regulated based on measurements to achieve a desired result. Some instruments may themselves require internal measurement and control. Imagine what goes on in a machine like the automated Mars rover, where robot arms and chemistry laboratories and instruments of all kinds have to function as an integrated measurement and control system far from human interaction. Many modern instruments, like DNA analyzers or automated water quality analyzers, are marvels of measurement and control.

In this experiment, you will turn on a pump to regulate the water level in a cup, or to keep up the moisture in the soil around a potted plant. You might think of this as smaller versions of a fish farm, or a water treatment plant, or a full-scale irrigation system for a vineyard.

Feedback is important here. It is possible to have control without feedback. If you have an automated system pour one cup of water on a potted plant every day, regardless of the condition of the plant, there is no feedback in that. You are in danger of over-watering the plant, and wasting water and fertilizer besides. It might not matter with one small plant in

well-drained soil, but imagine a dry-land farm, or a large greenhouse operation. If the condition of the soil or the plant is first measured, and from that decision is made of whether or not to irrigate, that is feedback at work. The result can be a happier plant as well as more efficient use of resources. This is especially important in places and times where water is scarce. Feedback can take on many forms, and involve a combination of measurements in the control decisions.

The final project in *Applied Sensors* will be a data logger that combines the two temperature sensors (the DS1620 and the AD592), the light sensor, the conductivity sensor, and the pump control in one program. The data will be stored in the BASIC Stamp's EEPROM memory. This data logger can also be used for a variety of experiments of your own design to undertake as class projects or on your own initiative. Thank you for sticking with it!

Parts Required

The following parts are required to complete this experiment.

(1) 100 Ω resistor
(2) 16" jumper wires, one red and one black
(1) Cup, in which a ¼" hole can be punched or drilled near the bottom (not included)
(1) Watertight tray or shallow dish made of glass or plastic (not included)
Duct tape or extra-wide electrical tape (not included)
(1) 10 Ω, 1 watt resistor, heavy-duty heat-resistant (brown-black-black)*
(1) TX1049A NPN "superbeta" transistor (marked ZTX 104 9A)*
(1) 3V Submersible water pump with 1/4" tubing (not included)*
(1) 5 VDC 300 mA external power supply (not included)*

* Editor's note: The datasheets for the pump and transistor used in the development of this kit are included in Appendix D: Data Sheets for your reference. If you choose to use a pump with different specifications, be sure to select a transistor, resistor, and external power supply adequate for the pump. Be sure to protect the BASIC Stamp with a minimum 100 Ω, ¼ watt resistor between P3 and the transistor base.

For this experiment, and external power supply is used as a precaution in case the pump selected has a current draw that exceeds the capacity of the 500 mA regulator on the Board of Education (all revisions) or BASIC Stamp HomeWork Board (rev C or higher). The HomeWork Board revisions A – C have a 50 mA regulator.

Figure 6-1 shows the preparation of the pump. The lead wires from the pump are fragile, and the pump cannot tolerate water inside the motor, so we suggest these precautions:

- √ Take a piece of duct tape or extra-wide electrical tape and wrap it around the plastic pump housing where the wire leads are connected.
- √ Pinch the tape together at the top to provide strain relief for the wires and protect the pump from splashed water.
- √ Connect the 16" jumper wires to the leads of the pump, by laying them ends-together and twisting the bare portions.
- √ Wrap the connections with tape for waterproofing and to provide strain relief.

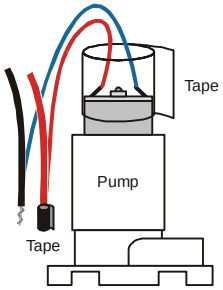


Figure 6-1
Pump Preparation

Wrap a piece of duct tape around the top of the pump to protect the wires and prevent water from getting into the pump. Join and tape the pump's wires with the long red and black wires that are included in the kit.

Building the Circuit

- √ Disconnect your board's wall-mount power supply or 9 V battery.
- √ Follow the schematic in Figure 6-2 and the wiring diagram in Figure 6-3 to add the transistor and pump circuit to your board.
- √ Be sure to orient the transmitter so that the printed face is directed away from the BASIC Stamp.
- √ As shown in the schematic in Figure 6-3, connect the collector of the transistor to the external regulated 5 VDC 300 mA power supply.
- √ The ground from the external 5 VDC 300 mA power supply must be connected to any ground pad on your Board of Education or HomeWork Board. When using split power supplies it is always important to tie the grounds together.
- √ For the moment, lay the conductivity sensor aside, but it will rest across the rim of the cup as shown in Figure 6-3 later on in the experiment.
- √ Plug in your board's original wall-mount power supply or 9 V battery.

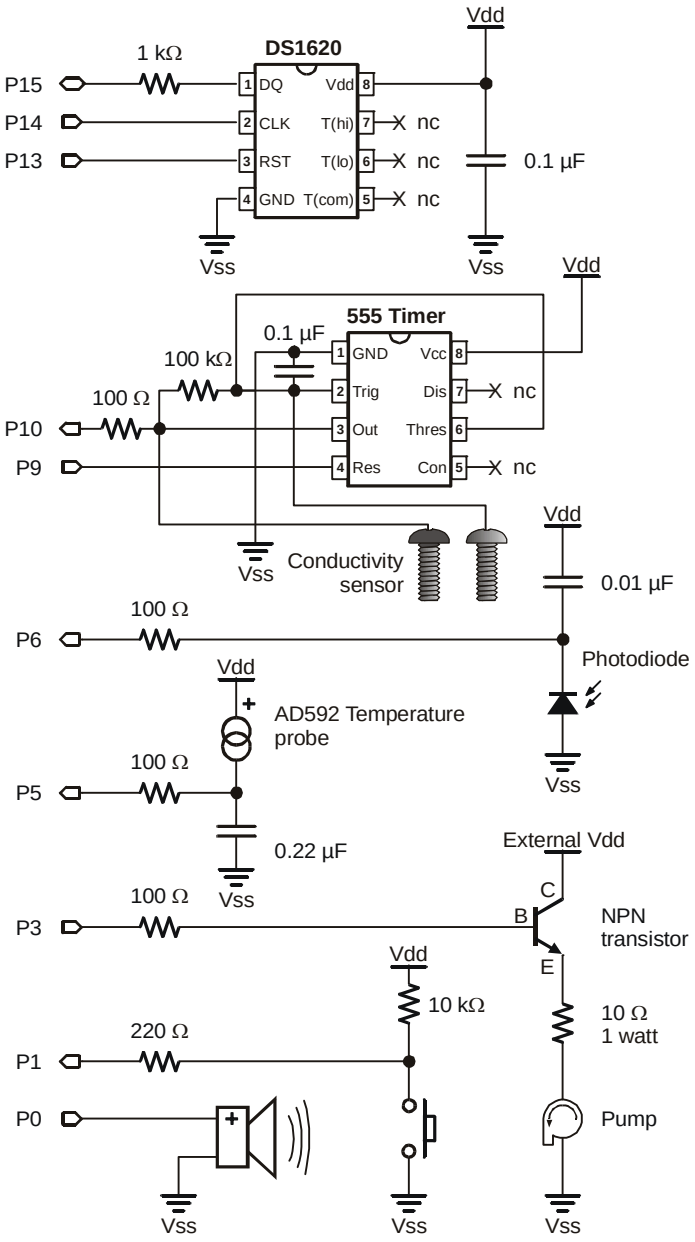


Figure 6-2
Pump Control with Transistor Wiring Diagram (for both Board of Education and HomeWork Board).

- *Transistor collector to external 5 VDC 300 mA regulated power supply*
- *Transistor base through a 100 ohm resistor to P3*
- *Transistor emitter to the 10 ohm , 1 watt resistor*
- *10 ohm 1 watt resistor the red pump wire*
- *Black pump wire to common Vss.*

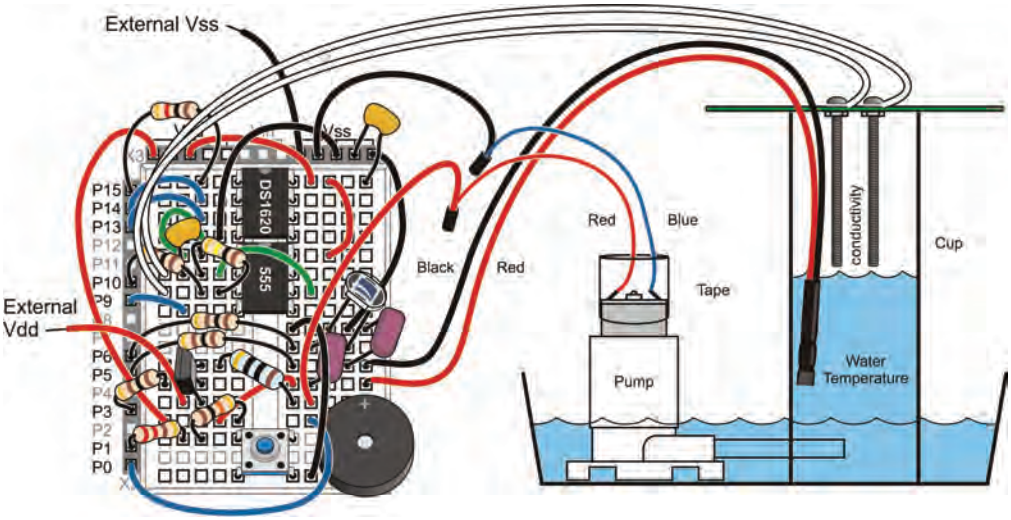


Figure 6-3: Pump Control with Transistor Wiring Diagram

On-Off Control of Pump

Now that your circuit is assembled, it is time to prepare the environment in which your instrument will work. The pump will raise the water level in the cup, but the water will drain back out when the power is off.

- √ Drill or punch a ¼ " hole in the side of your cup near the base.
- √ Insert the pump tube into this hole; it should fit snugly.
- √ Place the pump and the cup into your watertight tray.
- √ Fill the tray with tap water to cover the base of the pump. There needs to be enough water in the tray to fill the cup to the level of the conductivity probe.
- √ Keep more water and also a scoop or sponge on hand, to adjust the total water in your environment as may be necessary.

After building the circuits and setting up our water environment, we jump into the programming.

- √ Enter and run the program PumpTester.bs2.

```
' Applied Sensors – PumpTester.bs2
' Pump tester.
' {$STAMP BS2}
' {$PBASIC 2.5}

DO
  HIGH 3
  PAUSE 5000
  LOW 3
  PAUSE 2000
LOOP
```

Ideally, the pump should turn on for 5 seconds and off for 2 seconds, and repeat until you interrupt. If it works, great! If not, here are a couple of suggestions for troubleshooting.

- √ Kick the pump, but not literally! But sometimes the impeller becomes stuck if it has dried out in a way that leaves mineral deposits inside, so give it a tap.
- √ Look at the bottom of the pump, and you will see a hole and the vanes of the impeller inside. You can loosen them with the tip of a paper clip.
- √ Also, look to be sure that the tube that emerges from the side of the pump housing is not pushed in too far. If it is pushed into far it can prevent the impeller from turning.
- √ Use a wire to jump directly from the collector to the emitter of the transistor on your Board. That bypasses the program control of the pump, so you can tell if the problem is your pump or your program.
- √ If the pump still does not operate, then disconnect the pump from your breadboard and touch the pump wires to a 1.5 volt flashlight battery. That way you can tell if it is your pump or your breadboard circuit.
- √ If that doesn't do it, check the wires that connect to the pump for loose connections.
- √ If it still doesn't work, your pump may be defective or broken.

If the pump does work, but the circuit does not respond to the program, recheck the wiring.

- √ Be sure that the transistor is oriented correctly in the circuit and that the base is connected to P3.
- √ One end of the 10 Ω resistor should be in the same row as the emitter of the transistor.
- √ The wiring is getting tight. You have to check the connections carefully to be sure there is not a short circuit with one of the other parts on the Board.

Impeller: Inside the plastic housing at the bottom of the motor is a rapidly rotating disk with radial vanes. The vanes pull in water at the center through a hole you can see in the bottom of the housing, and throw the water out along the edge and force it to flow through the exit tube. The water is pulled through by the action of centrifugal force. The spinning disk with vanes is called an impeller.

Figure 6-4 shows an impeller, the rotating disk within the pump.

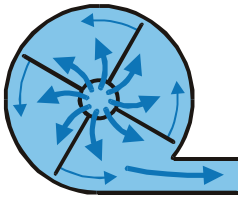


Figure 6-4
Pump Impeller

The motor in the pump draws about 300 milliamps of current at 3 volts. That is lots more than the pins of the BASIC Stamp are capable of supplying. It is necessary to use a transistor to amplify the current available from the BASIC Stamp. There are several ways to use transistors. The one here is called an "emitter follower." The voltage at the emitter of the transistor "follows" the voltage at the base. When P3 is low, at zero volts, the emitter of the transistor is also at zero volts and the motor is off. But when P3 goes high, to +5 volts, the emitter follows along (4.4 volts when P3 is at +5 volts) and the pump is turned on. The 300 milliamps needed by the pump comes through the transistor from the external power supply, not from pin P3.

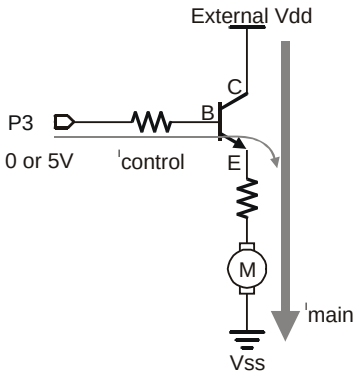


Figure 6-5
Transistor Action

e = emitter
b = base
c = collector

√ Now modify the program to look like PumpTesterButton.bs2.

```
' Applied Sensors – PumpTesterButton.bs2
' Testing the pump with the pushbutton.
' {$STAMP BS2}
' {$PBASIC 2.5}

OUTPUT 3

DO
  OUT3=~IN1
LOOP
```

Now the pump will be on only when you hold the pushbutton down. Maybe you were expecting a longer program? Look at the statement, **`OUT3=~IN1`**. What that says is, "make the output the opposite of the state of the pushbutton input." The symbol "**~**" means "not." If **`IN1`** is zero, pushbutton down, then the state of the pump will become 1, ON. If **`IN1`** is equal to 1, pushbutton up, then the state of the pump will become 0, OFF.

√ Try it. Mark a level on the side of the cup a little above the hole.
√ Hold down the button until the pump raises the water level in the cup to reach your mark.
√ Then release the button, and the water will begin to drain out.
√ Try to press and release the button so that you hold the water level close to the mark.

You are now part of a feedback loop. And soon you are going to be replaced by automation. The conductivity sensor is going to do the looking and turning the pump on and off. I think this is one job you would just as soon have automated.

A further word about the program: another way to write it would be to use **`IF…THEN`** statements, something like PumpTesterButtonIF.bs2, below.

```
' Applied Sensors – PumpTesterButtonIF.bs2
' Testing the pump with the pushbutton,
' Then selecting action with IF instruction.
' {$STAMP BS2}
' {$PBASIC 2.5}

DO
  IF IN1=0 THEN HIGH 3
  IF IN1=1 THEN LOW 3
LOOP
```

That is a fine way to write the program. It shows very well what is going on. If the pushbutton is down, the pump turns on. If the pushbutton is up, the pump turns off. The program has to take one or the other course of action, because **IN1** has to be either 0 or 1.

√ Enter PumpTesterButtonIF.bs2, and give it a try.

You may try to find other different ways to write the code. It is always good to know that there are different ways of accomplishing a task. The goal may be to make the code as compact as possible, or to make it run as fast as possible, or to make the program as easy as possible to follow in documentation. There is rarely just one solution.

Pump Control with Feedback

The objective now is to operate the pump until the water comes up to the level of the conductivity sensor's probe, and then to hold the level there, automatically, using the conductivity sensor as the level detector. In the above exercise you were using your eye as the level sensor and your finger on the pushbutton as part of the feedback loop. Now the BASIC Stamp will do the job.

√ Place the conductivity sensor on your cup so that the spanner rests on the rim and the end of the screws are above the resting water level.
√ Enter the program PumpController.bs2.
√ Use your own Duration calibration constant from page 132, divided by 10, for the **COUNT** command's ***Duration*** argument.

```
' Applied Sensors – PumpController.bs2
' Pump controller.
' {$STAMP BS2}
' {$PBASIC 2.5}

cnt       VAR     Word

DO
  HIGH 9                        ' Turn ON the 555.
  COUNT 10, 133, cnt            ' Count pulses.
            '^^---------------- ' USE YOUR CONSTANT divided by 10.
  LOW 9                         ' Turn OFF the 555.

  DEBUG DEC cnt, " umho", CR    ' Display micromho.

  IF cnt > 36 THEN LOW 3        ' Level too high, turn pump off.
  IF cnt < 30 THEN HIGH 3       ' Level too low, turn pump on.
LOOP                            ' Do it again.
```

The Duration calibration constant makes the reading come out in units of micro-mho (or micro-siemens, µS, in cgs units). What does that mean? If you take the reading from the Debug Terminal and divide it into 10^6, the result will be the resistance in ohms. It is not so important here to have real units for control of the level, but on general principles we like to remind you that there are accepted units of measurement for conductance.

√ Run the program.

The water should rise up to the level of the sensor, and then the pump should turn off. The water level drops until it no longer hits the sensor, and then the pump turns on. Remember, the count is higher as the probe is deeper in the water.

√ Observe the action: How often does the pump come on? What is the ratio of the "on" time to the "off" time?

This isn't a very efficient system, because the water leaks back out through the pump when the pump is off.

The critical level chosen for the water, the point where the pump changes from off to on and vice versa, is called the "set point." There are two set points in this program, one for on and one for off. The pump turns on when the value of **`cnt`** is less than 30, and it turns off when the value of **`cnt`** is greater than 36. Think about what happens in the program code when the value of **`cnt`** is 30 or 36. Neither of the **`IF`** statements is true, so the program goes around the loop without taking any action to change the state of the pump motor. If the motor was off, it stays off. If the motor was on, it stays on.

This kind of control, where the two set points are offset in such a way so that the effect lags behind the change, is called hysteresis. This program has 7 units of hysteresis, from 30 to 36 inclusive. This is a desirable feature in some kinds of feedback systems. For example, there are some kinds of motors and equipment that suffer if they are cycled on and off too often. It is better to let the liquid reach the upper set point, and then let the motor rest until the liquid drops below the lower set point before turning the motor back on again. That saves wear and tear on mechanical parts. Other times it is a requirement of the system, say, to let a plant dry out between waterings, or to create a surge in the water level in a fountain.

The hysteresis built into our program is expressed by the diagram in Figure 6-6. The vertical axis is the water level, or the conductance, since higher water level causes higher

conductance. The horizontal axis is the state of the pump, on or off. On the left, the water is low, and the pump is on. On the right, the water is high, and the pump is off. In operation, the system spends most of its time going around the square of hysteresis: ON up to the upper set point, then OFF, and falling down to the lower set point.

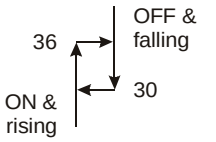


Figure 6-6
Hysteresis

You can experiment with the value of the high and the low set points by changing their values in the **`IF cnt >`** and **`IF cnt <`** instructions.

- √ Increase the upper set point in the **`IF cnt >`** instruction by 1 until you can make the water reach a high level on the conductivity sensor, without overflowing the cup.
- √ Observe the value of **`cnt`** on the Debug Terminal.
- √ Observe how often the pump cycles on and off.
- √ Now reduce the upper set point in the **`IF cnt >`** instruction by 1 (starting at 35) and observe how often the pump cycles on and off, as you approach **`IF cnt > 31.`**

What will happen to the actual level if the conductivity of the water changes?

- √ Return the program to the original **`IF`** values.
- √ Add a pinch of salt to the water.
- √ Observe how the behavior of the pump and the water level changes.

You can see that this is not a professional level sensor!

Here is an alternative way to program it that avoids the use of the **`IF…THEN`** statements. This is a matter of programming style. Try it!

- √ Enter and run the program PumpControllerEquation.bs2

```
' Applied Sensors – PumpControllerEquation.bs2
' Pump controller with equation.
' {$STAMP BS2}
' {$PBASIC 2.5}

cnt      VAR     Word          ' Variable for count.

LOW 3

DO

  HIGH 9                       ' Turn on the 555.
  COUNT 10, 200, cnt           ' Count pulses.
  LOW 9                        ' Turn off the 555.

  DEBUG DEC cnt, CR

  OUT3 = ~(cnt/36 MAX 1)

LOOP                           ' Do it again.
```

When the value of the **COUNT** is less than the set point of 36, the value of **cnt/36** will be zero. Remember, this in integer math, and the result of **cnt/36** is always an integer. When **cnt** is greater than or equal to 36, then the value of **cnt/36** will be 1 or greater. The additional operation, **MAX 1**, limits the value to 1 at most. There is a "**not**" operator, "**~**" in front of the whole expression in parentheses. The result is that when **cnt** is less than the set point, **OUT3** is high, and the pump is on. But when **cnt** is greater than or equal to the set point, then **OUT3** is low and the pump is off. Note that this program starts off with a command, **LOW 3**, that turns P3 into a low output. Otherwise P3 would be an input.

This program PumpControllerEquation.bs2 does not have any hysteresis. The pump is on for all values of the count less than 36, and off for all values greater than or equal to 36. However, we can alter the equation to add a little bit of hysteresis:

√ Modify the **OUT3** instruction so it reads:

```
OUT3=~(cnt/(30+(OUT3*6)) MAX 1)
```

√ Run the modified program.

The pump stays on until the water level reaches 36 or above, but once the pump is off, it does not turn back on until the level falls back below 30. The hysteresis is added here by

mixing the state of the output into the right-hand side of the formula. The BASIC Stamp can do that. **OUT3** is a variable like any other variable, and your program can either read or set its value. Think this through. It is tricky and not as transparent as doing it with **IF…THEN** commands. But this way of coding it is more compact. **IF…THEN** commands clutter up a program in their own way. It is an advanced technique for your bag of programming tricks!

Memory in the BASIC Stamp, Revisited

The data logger we developed in Chapters 4 and 5 had only 18 bytes of RAM memory. We could only have 9 records with 2 fields (temperature and light) in Chapter 4, or 6 records with 3 fields (temperature, light, and conductivity) in Chapter 5. Not only that, the data all disappeared if we turned off the power or pressed the Reset button and it was gone for good! This would be a serious shortcoming in a data logger for use in environmental science. A scientist or an engineer may want to collect much more data than that, and he or she most likely won't want the data to disappear prematurely. It needs to be retrieved and stored safely in an archive file before it is erased from the logger.

So we are going to switch over and store the data in the EEPROM memory instead of in the RAM, using 5 fields (ordinal record number, temperature from DS1620, temperature from AD592, light, and conductivity). Recall from Chapter 2 that there are 2048 bytes of EEPROM on the BASIC Stamp. We will reserve 250 bytes of that for our data log. That is lots more than we had available in RAM. We can reserve more than that, too, if need be for future experiments. Best of all, our data in EEPROM will survive resets and power outages.

The way we go about storing data is different in RAM that it will be in EEPROM. Here is the way it looked when recording temperature (T), light (L) and conductivity (C) in RAM:

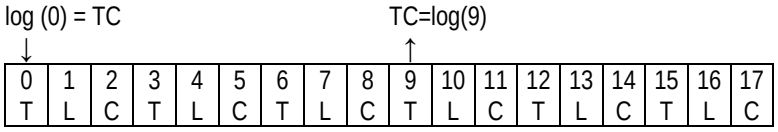


Figure 6-7
RAM log

There is an array of 18 byte-size variables, from **log(0)** to **log(17)**. We put data into bins with statements of the form, **log(i)=TC**, and we retrieve data from the bins with statements of the form, **TC=log(i)**. The value in parentheses is a variable, a pointer.

Here is the difference with EEPROM, recording ordinal record number (#), temperature from DS1620 (T), temperature from AD592 (t), light (L), and conductivity (C):

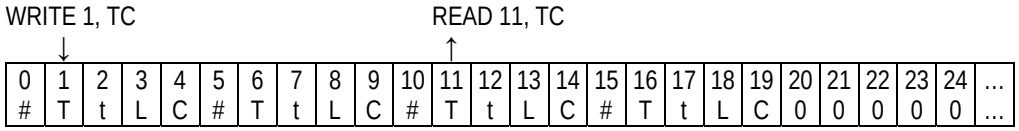


Figure 6-8: EEPROM log

TC is a byte-size quantity. As in RAM, we can use a pointer variable to reference the byte to write or to read. Recall that to write a temperature value into location 1 in the EEPROM, we use the statement:

```
WRITE 1, TC
```

…and to retrieve data from location 11 in EEPROM, we use the statement:

```
READ 11, TC
```

Reserving EEPROM Space

The following program demonstrates a couple of ways to reserve space in the EEPROM.

```
' Applied Sensors – EEPROMAllocation.bs2
' Illustration of EEPROM allocation.
' {$STAMP BS2}
' {$PBASIC 2.5}

Pad     DATA      (32)               ' Reserve 32 bytes, undefined.
Log     DATA      1(60)              ' Reserve 60 bytes, preset all = 1.
Hey     DATA      72, 101, 121, 33,  ' Reserve 8 bytes specified.
                  32, 66, 83, 50

ptr     VAR       Byte               ' Byte for pointer.
x       VAR       Byte               ' Byte for EEPROM data.

FOR ptr=0 TO 7                       ' Point to 8 locations in a row.
  READ ptr + Hey, x                  ' Read data from location.
  DEBUG DEC x, " "                   ' Show it, with a space.
NEXT

DEBUG CR
```

√ Enter the program EEPROMAllocation.bs2.
√ Run the program.

When you run this, (or press Reset on your Board), you should see the 8 numbers from the **Hey** data appear in the Debug Terminal.

This program sets aside space in EEPROM for data, 100 bytes of it to be exact. There are 32 bytes of undefined data (not preset to any particular value) starting at address zero, then 60 bytes of defined data (all preset to one) starting at address 32, and 8 bytes of numerical data at locations 92 through 99. Each of these 100 EEPROM locations contains an 8 bit data pattern. The 8 bit pattern could represent a number such as a temperature or a light level, or it could represent a letter to print in the Debug Terminal, or it could be a pattern of Morse code, or any other thing you could imagine to fit into a pattern of digital bits.

The **FOR…NEXT** loop reads and prints out the 8 numbers starting at EEPROM address **Hey**. We could have written it like this:

```
FOR ptr=92 TO 99        ' ← explicit values for the pointer
  READ ptr, x           ' ← read from those locations
' and so on.
```

But it is best to let the BASIC Stamp software keep track of the details of which number goes with which name. That makes future changes easier.

√ Now make a simple change to the **DEBUG** statement, as follows:

```
FOR ptr=0 TO 7
  READ ptr + Hey, x
  DEBUG x            ' ← change this, leave out DEC and , " ".
```

√ Run the modified program.

Now the BASIC Stamp reads the same 8 bytes from the EEPROM memory. But the **DEBUG** statement sends them as is as single bytes to the Debug Terminal. The Debug Terminal interprets them as printable characters. For example "72" sent as a single byte is the ASCII code for the letter "H" (ASCII-American Standard Code for Information Interchange). With the **DEC** modifier, the **DEBUG** statement takes the single byte with numerical value 72, and sends it to the Debug Terminal as two ASCII codes, first the one

for "7" and then the one for "2" next. If you substitute one of the other numerical indicators, you can see the numbers in binary (72 = %1001000) or in hex (72 = $48):

√ Modify and run EEPROMAllocation.bs2 3 more times, until you have used all of the variations of the **DEBUG** modifiers as shown below.

```
DEBUG x                 ' Show as ASCII text.
DEBUG DEC x, " "        ' Show decimal, with a space.
DEBUG BIN x, " "        ' Show binary, with a space.
DEBUG HEX x, " "        ' Show hex, with a space.
```

The point is that the binary pattern that is stored in the EEPROM is the same in each case. It is only the interpretation by the **DEBUG** command and by the Debug Terminal that is different. Please bear with us if you already know this. This is a confusing point for many students. We are going to use the EEPROM to store numerical data, but we usually use **DEBUG** with the decimal modifier.

We will store each point of data as one byte in the EEPROM. Each byte can represent a number from 0 to 255 decimal. Our logger will not store larger values. It is possible to do so, but it would take two EEPROM locations per point.
The declarations:

```
Pad     DATA   (32)       ' Reserve 32 bytes, undefined.
Log     DATA   1(60)      ' Reserve 60 bytes, preset all=1.
```

…are two other ways to reserve space for data in the EEPROM. The second form initializes the 60 bytes to have a value of 1, while the first form simply sets aside the bytes without specifying a value to store there.

√ Now modify the central part of the program EEPROMAllocation.bs2 once more as follows, to print out the decimal value of all 100 locations in EEPROM:

```
FOR ptr=0 TO 99         ' Read in all 100 bytes of data.
  READ ptr + Pad, x     ' Starting at pad (Pad=0).
  DEBUG DEC x, " "      ' Show data as decimal value.
NEXT
```

√ Run the modified program, and look at your Debug Terminal.

Now you should see 32 bytes of garbage, followed by 60 zeros, followed by the 8 bytes that have special meaning as ASCII text. Why do we say "garbage"? It is because the 32 bytes you see first are simply stuff that was left over in your BASIC Stamp from earlier programs and experiments. The program reserves space, but it does not send any new data to the BASIC Stamp to put in those locations.

The BASIC Stamp Editor has a very useful feature that lets you look directly at the allocation of the memory. It is an invaluable tool for program development.

√ Close the Debug Terminal if it is active on your screen.
√ Press CTRL-M (or choose Memory Map from the Run menu or from the tool bar).

Three views appear in one window, and they are labeled RAM Map, Condensed EEPROM Map, and Detailed EEPROM Map.

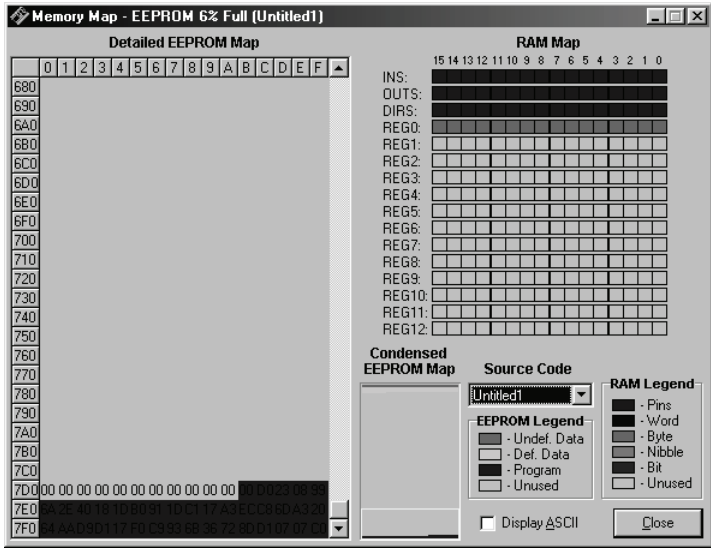


Figure 6-9
Memory Map

The left portion of the map is the EEPROM, your BASIC Stamp source code and extra EEPROM space. The right side of the map is variable storage, RAM. The lower right hand corner of the memory map defines the type of data by color that you are storing in EEPROM and RAM.

√ Look at the RAM Map. It is the view at the upper right side.

Recall that RAM is an acronym for Random Access Memory that is located inside the PIC microcontroller chip, and stores the program variables. There are 32 bytes, (16 words, 256 bits) altogether. The first 6 bytes (3 words) are dedicated to the I/O pins of the

BASIC Stamp. These variables bear the pre-assigned names, **ins**, **outs** and **dirs**. This is visible in Figure 6-9 to the left of the top three lines on the RAM Map. These pin variables appear in red on your color computer monitor. That leaves 13 words, which is 26 bytes, for the variables in our program. Program EEPROMAllocation.bs2 has only two variables, both of them defined as bytes. The RAM Map shows them in light blue color right below the pin variables. The rest of the RAM memory is not allocated in this program, and is shown in gray. Be aware that the RAM Map does not show you the actual values of the variables—that only happens when you run the program.

√ Now look at the Condensed EEPROM Map. It is in the lower middle of the Memory Map window.

Recall that the EEPROM memory is located in a separate chip, apart from the PIC microprocessor. There are 2048 bytes of EEPROM. At the top of the schema is the data in shades of blue, and at the bottom is the program code in red. Between the program and the data is empty space that will fill up as we write longer programs and reserve more space for data. You might ask what will happen if the two areas collide in the middle? If you have used the **DATA** directive to reserve log space, the problem will be detected when the program tries to compile during download and an "Out of memory" error message will appear. (Be warned, if you are using the **WRITE** command, which is executed during "run time" rather than "compile time" you won't have this error message to warn you, and you can begin to overwrite your program with data.) Notice that the data area has two different shadings. The first 32 bytes in blue are the "undefined data" or "empty data" declared with the statement:

```
Pad       DATA   (32)
```

When you load your program into the BASIC Stamp, the loading process does not touch those bytes, and that is why there was still the "garbage" you saw when you last ran Program EEPROMAllocation.bs2. In contrast, the following statements create what is called "defined data."

```
Log     DATA     1(60)
Hey     DATA     72, 101, 121, 33, 32, 66, 83, 50
```

When you run your program into the BASIC Stamp, specific bytes are loaded into the EEPROM along with the program itself.

√ Now look at the Detailed EEPROM Map.

This view shows the contents of the EEPROM byte by byte. You first see 32 dark blue zeros, followed by 60 zeros, followed by the 8 specific bytes. The display shows HEX numbers (from 00 to FF). You can press ALT-A to view the data as ASCII text.

Note that the BASIC Stamp Editor does not show you the "garbage." You only get to see that when you really run the program on the BASIC Stamp. You see, the BASIC Stamp gives you a lot of options on how to use the EEPROM resources.

Now use your mouse to move down to the bottom of the Detailed EEPROM Map. When you are looking at the bottom, you will see the actual bytes of the program itself as it is stored in the EEPROM. The program EEPROMAllocation.bs2 occupies about 40 bytes of memory. The program code is stored in a very compressed form, so don't look for an easy correspondence between the bytes in the EEPROM and the text of the program.

√ Close the window (ALT-C), to get back to the BASIC Stamp Editor.

The purpose of this digression was to help you think about the organization of the memory on the BASIC Stamp, and also to illustrate a very useful feature of the BASIC Stamp programming software.

Data Logger

Now it's time to get down to business. There are several issues that need to be addressed to make a working data logger. Rather than deal with them piecemeal, here altogether now are the design objectives.

- **Pump Controller Capability:** The data logger will also function as a pump controller, to keep the water level up in the cup. Therefore, the data logger has to do both measurement and control.

- **Pushbutton Data Sampling:** Clicking the pushbutton once will log 5 bytes in EEPROM as follows below.

Ordinal 1,2,3,. . . 50	Temperature from DS1620	Temperature from AD592	Light from photodiode	Conductance from probe	…more

- **Reconfigurable Automatic Timed Data Sampling:** The data logger needs take readings at a programmed interval from seconds to hours. For example, with hourly logging, and 50 records total, the unit could hold two days worth of data. The interval is set at the time of programming the BASIC Stamp.

- **50 records of 5 Bytes Each:** Since we are going to set aside 250 bytes for the data log, and since each record has five fields, there will be room for 50 records in the file. The log file can be made smaller or larger as needed for different projects.

- **Resume Data Logging after Interrupt:** The program can find its place in the data file even after pressing Reset, or a power outage. This is done by scanning the data file, where the next free data location will be tagged by a zero in the ordinal number field.

- **Pushbutton Data Retrieval:** press and hold down the pushbutton for 1.2 seconds to get into the routine to play back all the recorded data. This is like the RAM data logger in Chapters 4 and 5. After playback of the data, the logger can resume taking additional data where it left off.

- **Pushbutton Memory Erase:** To erase the data and start over, press and hold down the pushbutton during reset.

- **Audio Feedback:** Annunciate all the user interaction on the piezo transducer.

- **Visual Data Display:** Show data on the Debug Terminal.

- **Optional Morse Code Data Output:** Annunciate the logged data on the piezo transducer. The Morse code instructions are included in the program, but commented out. To activate this feature, you must delete the "'" from in front of the two **`GOSUB Morse`** instructions.

The final program used in this project is DataLogger.bs2, which is a compilation of portions of many programs you have already worked through. The starting point is program TwoChannelsThermometer.bs2, which you should have saved on disk. This program also uses code from program RAMDataLogger.bs2. If you wish, you can save yourself a little typing by cutting and pasting. The program is getting long, but we want to emphasize that it is built out of lots of pieces that you already know. This program just

brings them together in one place. The objective here is to give you a program you can use for further experiments.

One strategy is to simply type in all the changes, and then deal with any typos and errors later. That isn't a bad strategy where you have reason to believe that the program is basically okay (and we hope that it is!). Another strategy is to type small segments and test as you go along. That is usually the best way if you are uncertain about the pieces. You would first verify your program TwoChannelsThermometer.bs2, then add the additional variables and constants and data declarations, then the light and conductivity sensors, then the pump control routines, then the routine put data in memory, and then the timed data logging, and finally the routine to read the data out from memory.

- √ Enter the program DataLogger.bs2.
- √ Enter your own calibration values as requested in the Constants section comments. The requested values were calculated on the following pages:

  Kal AD592 calibration constant from page 70 = _____
  Lical calibration constant for indoor light from page 105 = _____
  Lical calibration constant for outdoor light from page 115 = _____
  Cntcal Duration calibration constant from page 132 = _____

```
' -----[ Title ]-----------------------------------------------------------
' Applied Sensors – DataLogger.bs2
' Temperature and light intensity logging in EEPROM,
' with simultaneous control of water level.
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Constants ]-------------------------------------------------------
' Morse code constants.
Dit         CON     50                  ' Milliseconds for Morse dit.
Dit2        CON     2*Dit               ' Constants related to Dit.
Dah         CON     3*Dit               ' Ditto.

' Sensor calibration constants. USE YOUR OWN calibration constants!
Kal         CON     15300               ' For the AD592 in Kelvin with .22uF.
Lical       CON     647                 ' For photodiode in lux with.01uf.
Cntcal      CON     1333/10             ' For conductance in umho with .1uf.

' Logging constants.
Interval    CON     600                 ' Logging interval (tenths seconds).
Nflds       CON     5                   ' Number of fields per record.
Nrecs       CON     50                  ' Number of records in file.
LogSiz      CON     Nflds * Nrecs       ' Size of the log file in bytes.
```

```
Pad         DATA    (16)                ' Pad to save wear and tear on memory.
Log         DATA    0(LogSiz)           ' Bytes in EEPROM for data log file.

' -----[ Declarations ]----------------------------------------------------
' General purpose variables.
xm          VAR     Byte                ' Morse & EEPROM input variable.
x           VAR     Byte                ' General purpose variable.
n           VAR     Word                ' Variable for time counter.

' Morse code variables.
mc          VAR     Byte                ' Temporary for Morse pattern.
j           VAR     Nib                 ' Index for digits to send.
i           VAR     Nib                 ' Index for dits and dahs.

' Sensor variables
degC        VAR     Word                ' For Celsius temperature from DS1620.
TK          VAR     Word                ' For Kelvin temperature from AD592.
TC          VAR     Word                ' Celsius from AD592.
rct         VAR     Word                ' For the RC timer.
light       VAR     Word                ' Light level from the photodiode.
cnt         VAR     Word                ' For the conductance probe.
umho        VAR     Byte                ' Conductivity.
mhoMax      VAR     Byte                ' Maximum value of conductivity.

' Logging variable.
ptr         VAR     Byte                ' Pointer to data in the log file.

' -----[ Initializations ]--------------------------------------------------
' Note: DS1620 has been preprogrammed for mode 2.
' If not, uncomment the instructions on the next line on the first RUN
' HIGH 13: SHIFTOUT 15,14,[12,2]: LOW 13

OUTS=%0000000001000000                  ' Now specify the OUTS and DIRS.
     'fedcba9876543210
DIRS=%1111101111111101                  ' P0 is output for piezo
                                        ' P1 is input for pushbutton
                                        ' P3 low for pump
                                        ' P5 is low output to discharge C
                                        ' P6 is high output to discharge C
                                        ' P9 is control of 555 ON-OFF
                                        ' P10 is input for 555
                                        ' P13-15 output for DS1620 SPI
                                        ' all unused pins are low outputs.

' -----[ Main Program ]-----------------------------------------------------
' Program execution starts here out of reset.
ptr = -5                                ' Pointer initialization.
DO                                      ' Find next free location in EEPROM.
  ptr = ptr + 5                         ' Point to a record.
  READ ptr + Log, x                     ' Read byte.
LOOP UNTIL (x=0 OR ptr>=LogSiz)         ' If x=0, this is a free record
```

```
                                    ' also test for full log, ptr=LogSiz
                                    ' pass here when free record is found
                                    ' ptr points to the next free record.

ON IN1 GOSUB Erase_Log              ' Erase log if button down at reset.

DEBUG ? ptr                         ' Show the pointer & the base address.
DEBUG "RESET+button=erase",CR       ' User message.

FREQOUT 0, 20, 1900                 ' Beep to signal that it is running.


' ----- Main Loop
DO
  n = 0                             ' Initialize counter.
  DO                                ' Loop here until button or time.
    GOSUB Pump                      ' Update the pump status.
    n = n + 1                       ' Count time.
  LOOP UNTIL (IN1=0 OR n=Interval)  ' Get data at intervals or button.
' Can press button to get data too.

  FREQOUT 0,5,2550                  ' Tick for button down.

  n = 0                             ' Initialize counter.
  DO                                ' Loop to track pressing time.
    PAUSE 100                       ' Time button in 0.1 sec increments.
    n = n + 1                       ' Increment counter.
  LOOP UNTIL (IN1=1 OR n>12)        ' Conditions to stop the loop.

  IF (n>=12) THEN                   ' Long click?
    GOSUB Playback
  ELSE                              ' Short click or time?
    GOSUB Get_Data
  ENDIF
LOOP

' -----[ Subroutines ]------------------------------------------------

' ----- Subroutine Get-Data
Get_Data:                           ' Come here to scan and log data.
  FREQOUT 0, 20, 3400               ' Got here!

  LOW 3                             ' Turn off pump, while scanning.
  xm = ptr/5 + 1                    ' First put the ordinal record.
  GOSUB Write_Data                  ' Write it in EEPROM!

  DEBUG CR, "logging data!", CR,    ' Message and units.
        "#", 32, "degC", TAB,
        "degC", TAB, "lux", TAB,
        "umho", CR, DEC xm, " "
```

```
' DS1620 temperature sensor code.
HIGH 13                                 ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [238]        ' Send "start conversions" command.
LOW 13                                  ' Finish the command.

PAUSE 450                               ' Delay for conversion.

HIGH 13                                 ' Select the DS1620.
SHIFTOUT 15, 14, LSBFIRST, [170]        ' Send the "get data" command.
SHIFTIN 15, 14, LSBPRE, [x]             ' Get the data.
LOW 13                                  ' End the command.

degC = x/2                              ' Convert the data to degrees C.

xm = degC                               ' Morse routine expects data in xm.
GOSUB Write_Data                        ' Write the degC data.

DEBUG DEC xm, TAB                       ' Show it on Debug Terminal.
' GOSUB Morse                           ' Send it as morse code (optional).

' AD592 temperature sensor code.
RCTIME 5, 0, rct                        ' Get the AD592 Count.
LOW 5                                   ' Pull input  low, discharge cap.

TK = Kal/rct*10 + (Kal//rct*10/rct)     ' Calculate Kelvin.

TC = TK-273                             ' Convert to degrees C.

xm = TC                                 ' Morse routine expects data xm.
GOSUB Write_Data                        ' Write the data to EEPROM.

DEBUG DEC xm, TAB                       ' Show it on Debug Terminal.
' GOSUB Morse                           ' Send it as Morse code (optional).

' Photodiode sensor code.
RCTIME 6, 1, rct                        ' Read the photodiode.
HIGH 6                                  ' Discharge photodiode capacitor.

light = 65535/rct */Lical               ' Calculate lux.

xm = light/2 MAX 255                    ' Ready to store it in EEPROM.
GOSUB Write_Data                        ' Store it in eeprom!

DEBUG DEC light, TAB                    ' Show it on Debug Terminal.

' Conductance sensor code.
xm = mhoMax                             ' Store max. value from Pump.
GOSUB Write_Data                        ' Data to EEPROM.

DEBUG DEC xm, CR                        ' Show max conductance in umho.
mhoMax = 0                              ' Reinitialize the accumulator.
```

```
RETURN

' ----- Subroutine Erase_Log
Erase_Log:
  FREQOUT 0, 400, 2550, 1900            ' Sound we got here.

  FOR x=0 TO ptr STEP 5                 ' Step through all used records.
    WRITE x + Log, 0                    ' Make them zero.
  NEXT

  DEBUG CLS, "data erased!", CR         ' Message on cleared screen.

  ptr = 0                               ' Reset pointer after erasing data.

  DO                                    ' Hold here until button released.
  LOOP UNTIL IN1=1
RETURN

' ----- Subroutine Playback
Playback:
  LOW 3                                 ' Pump off unconditional.
  FREQOUT 0, 50, 2550                   ' Sound we got here!
  FREQOUT 0, 100, 3400
  DEBUG CLS, "logged data!", CR,        ' Message and units.
        "#", 32, "degC", TAB,
        "degC", TAB, "lux", TAB,
        "umho", CR

  ptr = 0                               ' Point to start of data.

  READ ptr + Log, x                     ' Read first record number.
  DO WHILE (x<>0 AND ptr<LogSiz)        ' Meanwhile it's not zero or
                                        ' last record:
    DEBUG DEC x, " "                    ' show record number,
    READ ptr + 1 + log, degC            ' read temperature (DS1620),
    READ ptr + 2 + log, TC              ' read temperature (AD592),
    READ ptr + 3 + log, light           ' read light,
    READ ptr + 4 + log, umho            ' read conductance,
    light = light*2                     ' restore light units,

    DEBUG DEC degC, TAB, DEC TC,        ' show the values,
          TAB, DEC light, TAB,
           DEC umho, CR

    ptr = ptr + 5                       ' point to next record,
    READ ptr + Log, x                   ' and read next record number.
  LOOP                                  ' Back to check conditions.

  DO                                    ' Wait for button up
    DEBUG REP "-"\31, CR                ' printing horizontal line.
  LOOP UNTIL IN1=1
```

```
RETURN
' ----- Subroutine Morse
Morse:                                  ' Send xm in morse code.
  FOR j=1 TO 0                          ' Send 2 digits, tens then ones.
    mc = xm DIG j                       ' Pick off the (j+1)th digit.
    mc = %11110000011111 >> mc          ' Set up pattern for Morse code.

    FOR i=4 TO 0                        ' 5 dits and dahs
      ' Send pattern from bits of mc.
      FREQOUT 0, Dit2*mc.BIT0(i) + Dit, 1900
      PAUSE Dit                         ' Short silence.
    NEXT                                ' Next i, dit or dah of five.

    PAUSE Dah                           ' Interdigit silence.
  NEXT                                  ' Next j,digit of two.

RETURN                                  ' Back to main.

' ----- Subroutine Pump
Pump:
  HIGH 9                                ' Turn on the 555.
  COUNT 10, 100, cnt                    ' Count the frequency.
  LOW 9                                 ' Turn off the 555.

  umho = cnt * Cntcal/100 MAX 255       ' Calculate umho.
  mhoMax = umho MIN mhoMax              ' Keep the maximum value of umho.

  IF umho>99 THEN LOW 3                 ' Threshold to turn pump off.
  IF umho<50 THEN HIGH 3                ' Threshold to turn pump on.
  ' OUT3=~(umho/(OUT3*49+50) MAX 1)     ' Control the pump,  alternate.
RETURN                                  ' Back to main.

' ----- Subroutine Write-Data
Write_Data:
  IF ptr<LogSiz THEN                    ' Check for end of log.
    WRITE ptr + log, xm                 ' Write this field.
    ptr = ptr + 1                       ' Point to next field.
  ENDIF
RETURN
```

- √ Run the program.
- √ Try a short data logging experiment to test your equipment.

The logging interval is initially set to be about 1 minute (**`interval con 600`**) in tenths of a second. It should work pretty much like the RAM data logger you made in Chapters 4 and 5. If you are working in a classroom setting, your teacher may have other suggestions for the interval and for the size and location of the log file.

√ Check the design objectives above for the details about how it is supposed to work.
√ Refer to the Troubleshooting suggestions below, if necessary.

Troubleshooting

√ Check for program resets. If the program resets frequently, and seems like it never quite gets started, try to run it without the pump plugged in.

√ Check the breadboard and DS1620 for heat by touching them carefully. The pump draws quite a bit of electrical power. The 10 Ω resistor that is in series with the motor on the breadboard will get warm. Expect the temperature of the DS1620 temperature sensor to rise when you operate the pump for a long time.

√ Check for a non-responsive DS1620. If the DS1620 stops responding you see only zeros in the second column on the Debug Terminal. Try increasing the delay in the DS1620 routine from 450 to a larger value. A delay is required after sending to the DS1620 the [238] code that starts the analog to digital conversion. If you compare carefully you will see that in TwoChannelsThermometer.bs2 that "start conversions" code was only sent once, early in the program. Unfortunately, the DS1620 is quite sensitive to noise generated by the pump. As a quick fix, we turn off the motor, and then issue the "start conversions" command. In a real engineering project, this behavior would be troublesome, and effort would be expended to isolate and resolve the problem.

√ Check your calibration. Remember, if you change the timing capacitors, you also have to recheck the calibration.

√ Check your capacitor circuits. Be sure you have the 0.22 µF capacitor for the AD592 temperature sensor, and the 0.01µF capacitor for the photodiode, and the 0.1 µF capacitor for the conductance sensor. You may, for example, want to use the light sensor in outdoor sunlight, so you will have to change over to a 0.22 µf capacitor and also change `Lical` calibration constant.

Program Notes

Time-Critical Multitasking

Note that the pump subroutine at the end of the listing is called in a couple of places. In particular, it is called repeatedly in the initial pushbutton up and pushbutton down loops.

That is because the operation of the pump is a time-critical task. This is what many kinds of complicated programs have to do. They have to perform multiple tasks at practically the same time. Here it is to both monitor the button and keep up the water level. The programmer has to be sure that both tasks are serviced in a timely manner. The conductance reading in the pump routine takes 1/10 of a second, and paces the whole process. The BASIC Stamp is a relatively slow computer compared to a PC, and it cannot do true "multitasking." The BASIC Stamp here is getting around to the pump and the pushbutton about 10 times per second. You can see the level in the cup drop when the program goes off to take readings from the sensors. Maybe even a one or two-second delay is acceptable here. But in other systems, it could matter a whole lot and you would want a faster microcontroller. Check out www.parallax.com for our full line of microcontrollers.

Pump Power and Sensor Readings

Note the **`LOW 3`** command near the top of the **`Get_Data`** routine. The pump is turned off unconditionally while reading the sensors. Otherwise, the noise and heavy power supply drain of the pump would affect the readings. Try it to see what we mean. Comment out the **`LOW 3`** command, and then run the program again. During an interval when the pump happens to be on, press the pushbutton and note the readings on the Debug Terminal. Then press the pushbutton again during an interval when the pump is off, and compare those readings with it on.

Conductance Reading

Why **`mhoMax`**? The conductance reading requires some explanation. The conductance is being used to control the water level. So the pump routine reads the conductance value often as part of the pump subroutine. That routine keeps track of the maximum value of conductance that it has detected.

```
mhoMax = umho MIN mhoMax
```

What this says is, "let the new value of **`mhoMax`** be equal to whichever is greater, the conductance (**`umho`**) or the current value of **`mhoMax`** " (min, because **`mhoMax`** is the minimum value in the match). For example, if the current value of **`umho`** is 67, and the current value of **`mhoMax`** is 65, the new value of **`mhoMax`** will be 67. It is **`mhoMax`** that goes

into the data file in the conductance routine. **mhoMax** is then reset to zero so that it can accumulate a new and different maximum value during the next interval.

Writing Data

Figure 6-10 illustrates how the program uses the EEPROM. The **Write_Data** subroutine is called from several places in the **Get_Data** routine. First it is called to store the ordinal record number, and then once more for each sensor. At the end of the **Get_Data** routine, the pointer is left pointing to the next free byte, where the next record number will go the next time the interval passes or the next time the pushbutton is pressed.

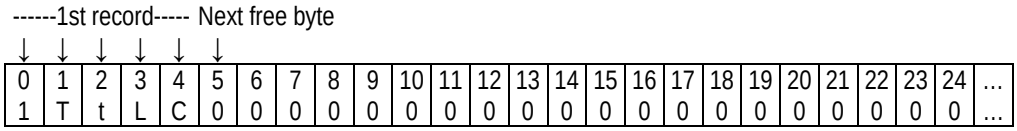


Figure 6-10: Writing Data

In professional data loggers, the record ordinal number field might be used to store the time and date.

Scanning Data Storage

The routine at the beginning of the program scans through all of the possible records in the data file, looking only at the locations reserved for record numbers: (**ptr** = 0, 5, 10, 15, ... ,245). If it finds a zero in one of those locations, the program will begin writing the next record at that place. This scan takes place each time you press Reset on your Board, or when the power is turned on. By putting tags in the data file, it can reconstruct where it was. Note that right after that is the instruction that tests to see if the pushbutton is being held down, just after reset. If so, the BASIC Stamp branches to the routine that "erases" the data file. What it really does is to put zeros in all of the ordinal number locations. That way the program starts over at the beginning of the log. The **Erase_Log** routine does not really erase all the data, just the ordinal number of each record. This helps reduce the wear and increase the life of your BASIC Stamp module's EEPROM.

Data Playback

Figure 6-11 illustrates that the routine that plays back the data is the reverse of the routine that puts it into the memory.

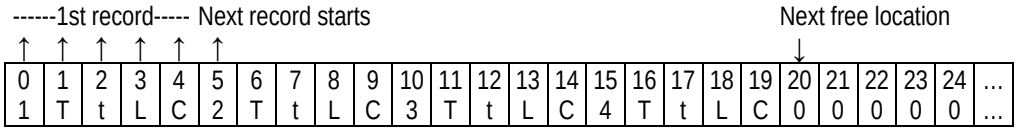


Figure 6-11: Reading Data

The routine reads out 5 locations, and then leapfrogs the pointer up to the next group of five. At the end of all the records, the pointer will be left pointing to the next free location, one containing a zero in the ordinal number field.

Pushbutton Control

The code that detects when you press and release the pushbutton should be very familiar to you by now. Here every time around the loop, while waiting for something to happen, the program checks the pump and turns if off or on as needed. The routine to read the conductivity takes about 0.1 second, so that paces the button loops too.

Timing Calibration

The timing interval for data logging is not very precise. You may want to calibrate it by changing the ballpark constants. To calibrate the timing, you need a stop watch. Set the interval for something like 1 minute, and time it (using the beeps for logging) to see how it turns out with interval = 6000. If the actual time turns out to be 1% too long, then to compensate, count for 1% less time, or interval = 5940.

EEPROM Lifespan

Can you ruin the EEPROM? Note that 16 bytes are set aside at the top of the memory. The purpose of this is to avoid overstressing those top bytes. Recall that EEPROM has a finite life in terms of the number of times it can be written to. It takes ten million or so rewrites to wear out a location. When using EEPROM for data logging, you have to remember this. It takes a long time to get to 10 million times, but don't get carried away

with rapid fire experiments with sub second timing! To check what is in memory, while in the BASIC Stamp Editor, press CTRL-M. Recall the discussion of the RAM Map and the EEPROM Map. This helps to visualize how the variables and EEPROM space are being used. What fraction of the EEPROM are the program and the data occupying?

Other Investigations

Pump Operational Limitations

Design an experiment to study the flow rate from the pump, and the height of water it is capable of supporting.

Solar Heater

- √ Make a solar heater by running water through black copper tubing under glass or in a bottle, out in the sun.
- √ Run the pump to bring water into a reservoir.
- √ Monitor the water temperature and the sunlight.
- √ Use the water temperature and the sunlight to decide when to turn on and off the pump automatically.

Pump PWM

Try to PWM the pump, that is, turn it rapidly on and off as follows:

```
x    CON    5

DO
  HIGH 3
  PAUSE x
  LOW 3
  PAUSE 10-x
LOOP
```

The constant 5 causes the pump to be on half the time and off half the time. This way the pump effectively pumps half as fast. The on and off states alternate so fast that you can't perceive a stop and start in the pumping action. Try varying the constant to see what happens at other ratios of ON to OFF time. You can also try the BASIC Stamp 2 built-in PWM command.

Challenge!

1. Write a simple program that turns on the pump when the pushbutton is pressed once, and then lets it stay on until the pushbutton is pressed again. (Push on, push off).

2. You are in charge of a public fountain that is supposed to operate only in the daytime, and only when the sun is out and the temperature is greater than 70 degrees Fahrenheit. Write a program to control the fountain.

3. Make a program that can draw a graph of conductance on the Debug Terminal. Let the program fill the cup to overflowing, and then let the level drop below the sensor tips, before starting up the pump again. All the while the conductivity reading should be graphing in the Debug Terminal.

4. A fish farm must maintain the level of water in a tank and replenish it, and keep the water stirred, and monitor for conditions that could be deadly. Write a program that keeps the water going up and down in the cup, but it will sound an alarm if the temperature of the water exceeds 80 degrees C, or if the water stops flowing for any reason, or if the conductivity of the water changes drastically.

5. Sometimes in real-world settings it is desirable to know how long or what percentage of time a motor is running versus not running. This helps with maintenance and with planning for energy efficiency. Modify program PumpController.bs2 so that it displays the percentage of time that the pump stays on. You could also add this to your data logging program, as an indication of how much water was used.

Appendix A: Parts Listing

Required Hardware

To complete the activities in **Chapters 1 through 5** of this text, you will need the following equipment:

- PC running Windows 98/2K/XP/Vista with an available serial or USB port
- BASIC Stamp Editor v2.4 (or higher) software
- Board of Education Full Kit (serial, #28103, or USB, #28803)
  -OR- a BASIC Stamp HomeWork Board and programming cable*
- Applied Sensors Parts Kit (see Table A-1 on page 180)
- Additional Household Items (see Table A-2 on page 181)

*****The Applied Sensors experiments are fully compatible with the HomeWork Board, which has a BASIC Stamp 2 built right in. The economical HomeWork Board is included in the BASIC Stamp Activity Kit (#90005) and also in 10-packs (#28158). Please contact sales@parallax.com for assistance in outfitting your classroom.

For free technical support, email support@parallax.com or call 1-888-99-STAMP from the United States. From outside the United States, call (916) 624-8333. Or, visit our Stamps in Class forum at http://forums.parallax.com.

Low-voltage Pump Circuit Not Included

The optional activities in Chapter 6 require a low-voltage pump, transistor, and resistor, which are not included. The low-voltage pump used to develop the activities in Chapter 6 is no longer available. The activities in Chapter 6 are optional and included for your reference and adaptation to commercially available low-voltage pumps. As Parallax identifies sources of compatible and reasonably inexpensive low-voltage pumps, we will post links on the Applied Sensors product pages at www.parallax.com.

We also invite our customers to submit information about low-voltage pump resources, or project adaptations to other types of pumps, to editor@parallax.com for potential posting on our website.

Applied Sensors Parts Kit

This kit contains all of the electronic components for the activities in Chapters 1–5. For your convenience, these components are pictured with labels and Parallax part numbers on the last page of this text.

Table A-1: Applied Sensors Parts Kit (#28126)
Parts and quantities subject to change without notice

Parallax Code#	Description	Quantity
150-01011	100 Ω¼ watt 5% resistor	4
150-01020	1 kΩ ¼ watt 5% resistor	1
150-01030	10 kΩ ¼ watt 5% resistor	1
150-01040	100 kΩ ¼ watt 5% resistor	2
150-02210	220 Ω ¼ watt 5% resistor	2
150-04710	470 Ω ¼ watt 5% resistor	2
200-01010	100 pF mono radial capacitor	2
200-01031	0.01 µF 50 V capacitor	1
200-01040	0.1 µF mono radial capacitor	3
200-02240	0.22 µF 50 V capacitor	3
28130	AD592 Temperature probe	1
350-00001	LED, green	1
350-00006	LED, red	1
350-00012	Photodiode, blue enhanced (Photonic Detectors)	1
400-00002	Pushbutton tact switch	1
604-00002	DS1620 Digital Thermometer	1
604-00009	555 timer, 8-pin DIP	1
700-00020	2", 4-40 stainless steel machine screws	2
700-00036	4-40 nylon nut	2
700-00058	Cup spanner	1
800-00016	3" jumper wires (bag of 10)	2
800-00021	16" red jumper wire	1
800-00022	16" black jumper wire	1
900-00001	Piezospeaker (sound transducer)	1

Additional Items Required

Some additional common items are needed to complete the activities in this text; those marked with an asterisk are required for the main experiments, and the others are used for optional activities and challenges.

Table A-2: Additional Items Needed (* required item)
Aluminum foil
Crushed ice *
Cup that can have a hole drilled or punched in it *
Distilled water *
Duct tape or wide electrical tape *
Heavy cotton thread or string
Graphite pencil
Paper
Paper towels
Potted plant
Protractor
Rock salt
Rubber band
Spoon
Spotlight - 50 watt R20, or 100 watt standard light bulb *
Table salt *
Tap water *
Thermos, or Styrofoam cups and aluminum foil *
Vinegar
Waterproof nonmetal tray at least 1.5 inches deep *
9 volt battery
Low-voltage submersible water pump with flexible tubing (Chapter 6 only)*
External power supply suited for the low-voltage pump selected (Chapter 6 only)*
NPN Transistor suited for the low-voltage pump selected (Chapter 6 only)*
Resistor suitable between transistor and low-voltage pump selected (Chapter 6 only)*

Appendix B: Building the ADS592 Temperature Probe

The *Applied Sensors* experiments use the AD592 temperature probe. The part needs to be protected before being inserted into liquid. Parallax builds a custom temperature probe (#28130), but you can do this yourself from these plans. An abbreviated datasheet for the AD592 is included in Appendix D of this text.

You'll need the following materials:

(1) AD592 Temperature Transducer in plastic TO-92 case (Newark Electronics)
(2) 16" wires, one black and one red, stripped on both ends
(2) 1" solder sleeves (Powell Electronics CWT-1502 or equivalent)
(1) 1½" length of ¼" adhesive lined heat shrink tubing (Digi-Key #EPS3316NK-ND)
(1) heat gun

To build the probe:

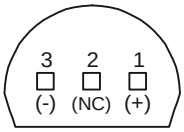


Figure B-1
Step #1

Identify the AD592's (-), NC, and (+) pins from this picture as viewed from the bottom.

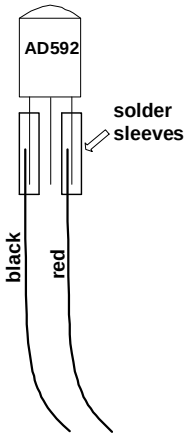


Figure B-2
Step #2

Slip the solder sleeve over the black wire and pin 3 (-). Slip another solder sleeve over the red wire and pin 1 (+). Heat up the connections until the wires are joined.

If you have no solder sleeves you can use heat shrink tubing.

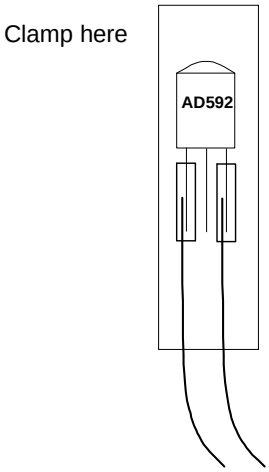


Figure B-3
Step #3

Slip the heat shrink tubing over the entire package. Fasten the package with a heat gun, and while it's still hot clamp the top portion to ensure that it stays shut.

Appendix C: Resistor Color Code

Resistor Color Code

Most common types of resistors have colored bands that indicate their value. Most of 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, and the gold color designates that it has a 5% tolerance.

The resistor color code is an industry standard in recognizing the value of resistance of a resistor. Each color band represents a number, and the order of the color band will represent a number value. The first two color bands indicate a number. The third color band indicates the multiplier, or in other words, the number of zeros. The fourth band indicates the tolerance of the resistor as +/- 5, 10, or 20 %.

Color	1st Digit	2nd Digit	Multiplier	Tolerance
black	0	0	1	
brown	1	1	10	
red	2	2	100	
orange	3	3	1,000	
yellow	4	4	10,000	
green	5	5	100,000	
blue	6	6	1,000,000	
violet	7	7	10,000,000	
gray	8	8	100,000.000	
white	9	9	1,000,000,000	
gold				5%
silver				10%
no color				20%

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 Ω (or 270 kΩ) and a tolerance of 5%.

Appendix D: Data Sheets

Appendix D consists of abbreviated data sheets for the key components used in these experiments. The full data sheets are available from the manufacturers' web sites shown below. This text includes the first two pages only of the data sheets.

Table D-1: Datasheet Locator	
Component	**Manufacturer's Website**
Analog Devices 592	http://www.analogdevices.com
Dallas Semiconductor 1620	http://www.maxim-ic.com
Edmund Scientific Pump	http://scientificsonline.com
ZTX1049A NPN Transistor	http://www.zetex.com/

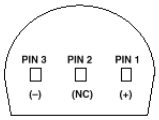
ANALOG DEVICES

Low Cost, Precision IC Temperature Transducer

AD592*

FEATURES
High Precalibrated Accuracy: 0.5°C max @ +25°C
Excellent Linearity: 0.15°C max (0°C to +70°C)
Wide Operating Temperature Range: –25°C to +105°C
Single Supply Operation: +4 V to +30 V
Excellent Repeatability and Stability
High Level Output: 1 μA/K
Two Terminal Monolithic IC: Temperature In/ Current Out
Minimal Self-Heating Errors

CONNECTION DIAGRAM



★ PIN 2 CAN BE EITHER ATTACHED OR UNCONNECTED
BOTTOM VIEW

PRODUCT DESCRIPTION

The AD592 is a two terminal monolithic integrated circuit temperature transducer that provides an output current proportional to absolute temperature. For a wide range of supply voltages the transducer acts as a high impedance temperature dependent current source of 1 μA/K. Improved design and laser wafer trimming of the IC's thin film resistors allows the AD592 to achieve absolute accuracy levels and nonlinearity errors previously unattainable at a comparable price.

The AD592 can be employed in applications between –25°C and +105°C where conventional temperature sensors (i.e., thermistor, RTD, thermocouple, diode) are currently being used. The inherent low cost of a monolithic integrated circuit in a plastic package, combined with a low total parts count in any given application, make the AD592 the most cost effective temperature transducer currently available. Expensive linearization circuitry, precision voltage references, bridge components, resistance measuring circuitry and cold junction compensation are not required with the AD592.

Typical application areas include: appliance temperature sensing, automotive temperature measurement and control, HVAC (heating/ventilating/air conditioning) system monitoring, industrial temperature control, thermocouple cold junction compensation, board-level electronics temperature diagnostics, temperature readout options in instrumentation, and temperature correction circuitry for precision electronics. Particularly useful in remote sensing applications, the AD592 is immune to voltage drops and voltage noise over long lines due to its high impedance current output. AD592s can easily be multiplexed; the signal current can be switched by a CMOS multiplexer or the supply voltage can be enabled with a tri-state logic gate.

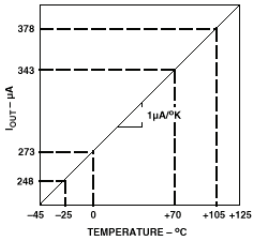
The AD592 is available in three performance grades: the AD592AN, AD592BN and AD592CN. All devices are packaged in a plastic TO-92 case rated from –45°C to +125°C. Performance is specified from –25°C to +105°C. AD592 chips are also available, contact the factory for details.

*Protected by Patent No. 4,123,698.

REV. A

PRODUCT HIGHLIGHTS

1. With a single supply (4 V to 30 V) the AD592 offers 0.5°C temperature measurement accuracy.
2. A wide operating temperature range (–25°C to +105°C) and highly linear output make the AD592 an ideal substitute for older, more limited sensor technologies (i.e., thermistors, RTDs, diodes, thermocouples).
3. The AD592 is electrically rugged; supply irregularities and variations or reverse voltages up to 20 V will not damage the device.
4. Because the AD592 is a temperature dependent current source, it is immune to voltage noise pickup and IR drops in the signal leads when used remotely.
5. The high output impedance of the AD592 provides greater than 0.5°C/V rejection of supply voltage drift and ripple.
6. Laser wafer trimming and temperature testing insures that AD592 units are easily interchangeable.
7. Initial system accuracy will not degrade significantly over time. The AD592 has proven long term performance and repeatability advantages inherent in integrated circuit design and construction.



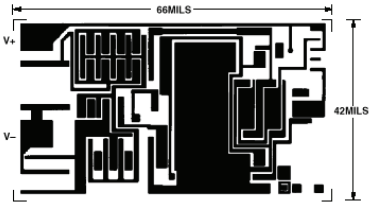
One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 617/329-4700 **Fax: 617/326-8703**

AD592–SPECIFICATIONS (typical @ T_A = +25°C, V_S = +5 V, unless otherwise noted)

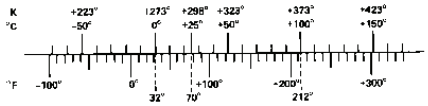
Model	AD592AN Min	AD592AN Typ	AD592AN Max	AD592BN Min	AD592BN Typ	AD592BN Max	AD592CN Min	AD592CN Typ	AD592CN Max	Units
ACCURACY										
Calibration Error @ +25°C[1]		1.5	**2.5**		0.7	**1.0**		0.3	**0.5**	°C
T_A = 0°C to +70°C										
Error over Temperature		1.8	3.0		0.8	1.5		0.4	0.8	°C
Nonlinearity[2]		0.15	0.35		0.1	0.25		0.05	0.15	°C
T_A = –25°C to +105°C										
Error over Temperature[3]		2.0	**3.5**		0.9	**2.0**		0.5	**1.0**	°C
Nonlinearity[2]		0.25	0.5		0.2	0.4		0.1	0.35	°C
OUTPUT CHARACTERISTICS										
Nominal Current Output @ +25°C (298.2K)		298.2			298.2			298.2		µA
Temperature Coefficient		1			1			1		µA/°C
Repeatability[4]			0.1			0.1			0.1	°C
Long Term Stability[5]			0.1			0.1			0.1	°C/month
ABSOLUTE MAXIMUM RATINGS										
Operating Temperature	–25		+105	–25		+105	–25		+105	°C
Package Temperature[6]	–45		+125	–45		+125	–45		+125	°C
Forward Voltage (+ to –)			**44**			**44**			**44**	V
Reverse Voltage (– to +)			**20**			**20**			**20**	V
Lead Temperature (Soldering 10 sec)			300			300			300	°C
POWER SUPPLY										
Operating Voltage Range	4		30	4		30	4		30	V
Power Supply Rejection										
+4 V < V_S < +5 V			**0.5**			**0.5**			**0.5**	°C/V
+5 V < V_S < +15 V			**0.2**			**0.2**			**0.2**	°C/V
+15 V < V_S < +30 V			**0.1**			**0.1**			**0.1**	°C/V

NOTES
[1]An external calibration trim can be used to zero the error @ +25°C.
[2]Defined as the maximum deviation from a mathematically best fit line.
[3]Parameter tested on all production units at +105°C only. C grade at –25°C also.
[4]Maximum deviation between +25°C readings after a temperature cycle between –45°C and +125°C. Errors of this type are noncumulative.
[5]Operation @ +125°C, error over time is noncumulative.
[6]Although performance is not specified beyond the operating temperature range, temperature excursions within the package temperature range will not damage the device.
Specifications subject to change without notice.
Specifications shown in boldface are tested on all production units at final electrical test. Results from those tests are used to calculate outgoing quality levels. All min and max specifications are guaranteed, although only those shown in boldface are tested on all production units.

METALIZATION DIAGRAM



TEMPERATURE SCALE CONVERSION EQUATIONS



$$°C = \frac{5}{9}(°F - 32) \qquad K = °C + 273.15$$

$$°F = \frac{9}{5}°C + 32 \qquad °R = °F + 459.7$$

ORDERING GUIDE

Model	Max Cal Error @ +25°C	Max Error –25°C to +105°C	Max Nonlinearity –25°C to +105°C	Package Option
AD592CN	0.5°C	1.0°C	0.35°C	TO-92
AD592BN	1.0°C	2.0°C	0.4°C	TO-92
AD592AN	2.5°C	3.5°C	0.5°C	TO-92

D

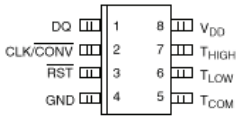
DALLAS SEMICONDUCTOR

DS1620
Digital Thermometer and Thermostat

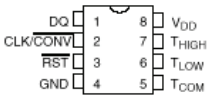
FEATURES

- Requires no external components
- Supply voltage range covers from 2.7V to 5.5V
- Measures temperatures from –55°C to +125°C in 0.5°C increments. Fahrenheit equivalent is –67°F to +257°F in 0.9°F increments
- Temperature is read as a 9–bit value
- Converts temperature to digital word in 1 second (max)
- Thermostatic settings are user–definable and non–volatile
- Data is read from/written via a 3–wire serial interface (CLK, DQ, $\overline{RST}$)
- Applications include thermostatic controls, industrial systems, consumer products, thermometers, or any thermally sensitive system
- 8–pin DIP or SOIC (208 mil) packages

PIN ASSIGNMENT



DS1620S 8–PIN SOIC (208 MIL)
See Mech Drawings Section



DS1620 8–PIN PDIP (300 MIL)
See Mech Drawings Section

PIN DESCRIPTION

DQ	–	3–Wire Input/Output
CLK/$\overline{CONV}$	–	3–Wire Clock Input and Stand–alone Convert Input
$\overline{RST}$	–	3–Wire Reset Input
GND	–	Ground
T_{HIGH}	–	High Temperature Trigger
T_{LOW}	–	Low Temperature Trigger
T_{COM}	–	High/Low Combination Trigger
V_{DD}	–	Power Supply Voltage (3V – 5V)

DESCRIPTION

The DS1620 Digital Thermometer and Thermostat provides 9–bit temperature readings which indicate the temperature of the device. With three thermal alarm outputs, the DS1620 can also act as a thermostat. T_{HIGH} is driven high if the DS1620's temperature is greater than or equal to a user–defined temperature TH. T_{LOW} is driven high if the DS1620's temperature is less than or equal to a user–defined temperature TL. T_{COM} is driven high when the temperature exceeds TH and stays high until the temperature falls below that of TL.

User–defined temperature settings are stored in non–volatile memory, so parts can be programmed prior to insertion in a system, as well as used in stand–alone applications without a CPU. Temperature settings and temperature readings are all communicated to/from the DS1620 over a simple 3–wire interface.

OPERATION–MEASURING TEMPERATURE

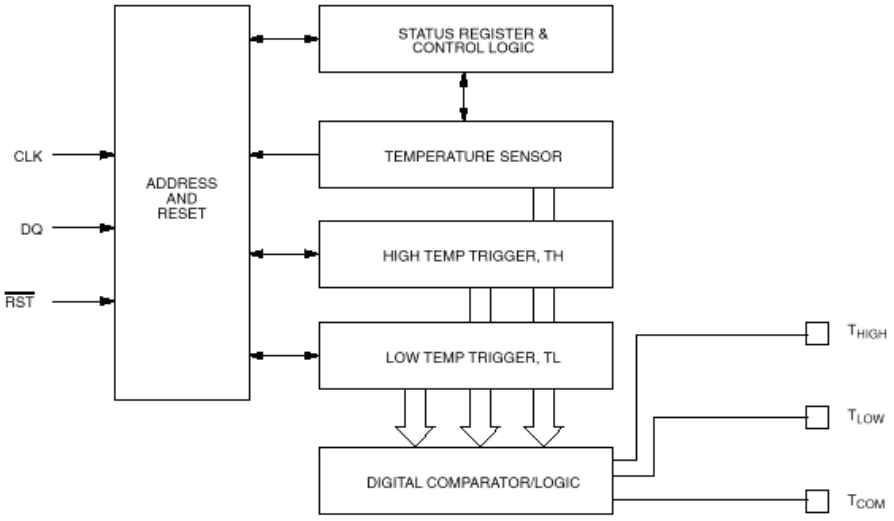
A block diagram of the DS1620 is shown in Figure 1. The DS1620 measures temperatures through the use of an on–board proprietary temperature measurement technique. A block diagram of the temperature measurement circuitry is shown in Figure 2.

The DS1620 measures temperature by counting the number of clock cycles that an oscillator with a low temperature coefficient goes through during a gate period determined by a high temperature coefficient oscillator. The counter is preset with a base count that corresponds to –55°C. If the counter reaches zero before the gate period is over, the temperature register, which is also preset to the –55°C value, is incremented, indicating that the temperature is higher than –55°C.

At the same time, the counter is then preset with a value determined by the slope accumulator circuitry. This circuitry is needed to compensate for the parabolic behavior of the oscillators over temperature. The counter is then clocked again until it reaches zero. If the gate period is still not finished, then this process repeats.

The slope accumulator is used to compensate for the nonlinear behavior of the oscillators over temperature, yielding a high resolution temperature measurement. This is done by changing the number of counts necessary for the counter to go through for each incremental degree in temperature. To obtain the desired resolution, therefore, both the value of the counter and the number of counts per degree C (the value of the slope accumulator) at a given temperature must be known.

DS1620 FUNCTIONAL BLOCK DIAGRAM Figure 1



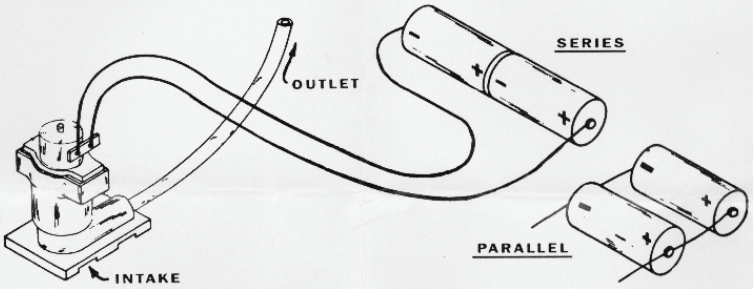
D

Edmund Scientific

INFORMATION AND INSTRUCTIONS

MINIATURE WATER PUMP
NO.50,345

This small water pump operates on 1½ to 3 volts direct current provided by batteries, either standard "D" cells or longer lasting No.6 ignition cells. The batteries may be connected in series, if you want highest capacity, or parallel for longer life. The illustration, below shows how to connect cells in series and parallel. To connect cells in series, connect the plus terminal of one to the minus terminal of the other; connect the pump wires to the remaining terminals on each battery. To connect cells in parallel, connect both plus terminals to one wire of the pump and both negative terminals to the other.



Capacity: **Approximately 1 pint/minute at 12" head.**
Current: **3 to 4 amperes at 3 volts.**

Although it will pump water from the outlet whem the impeller is rotating either way, the pump is slightly more efficient when the impeller is rotating counter clockwise (as viewed from the top).

Some suggested uses for this water pump:

1. Temperature control and circulation of darkroom chemicals
2. Table fountains and displays
3. Model waterfalls, hydroelectric installations, canal locks, etc.
4. Water filtration and aeration of fish tanks
5. Flow models of circulatory systems
6. Automatic plant watering unit
7. Humidifier

A NOTE OF CAUTION: The pump should not be placed in more than 1" of water as higher water level will flood the motor.

Edmund Scientific Co. 7785 Edscorp Bldg., Barrington, N.J. 08007

ZTX1049A

NPN SILICON PLANAR MEDIUM POWER HIGH GAIN TRANSISTOR

FEATURES

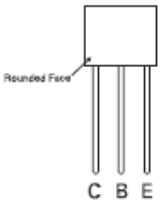
- V_{CEV} = 80V
- Very low saturation voltages
- High gain
- 20 amps pulse current

APPLICATIONS

- LCD backlight converters
- Emergency lighting
- DC-DC converters

E-Line

PINOUT



SIDE VIEW

ABSOLUTE MAXIMUM RATINGS

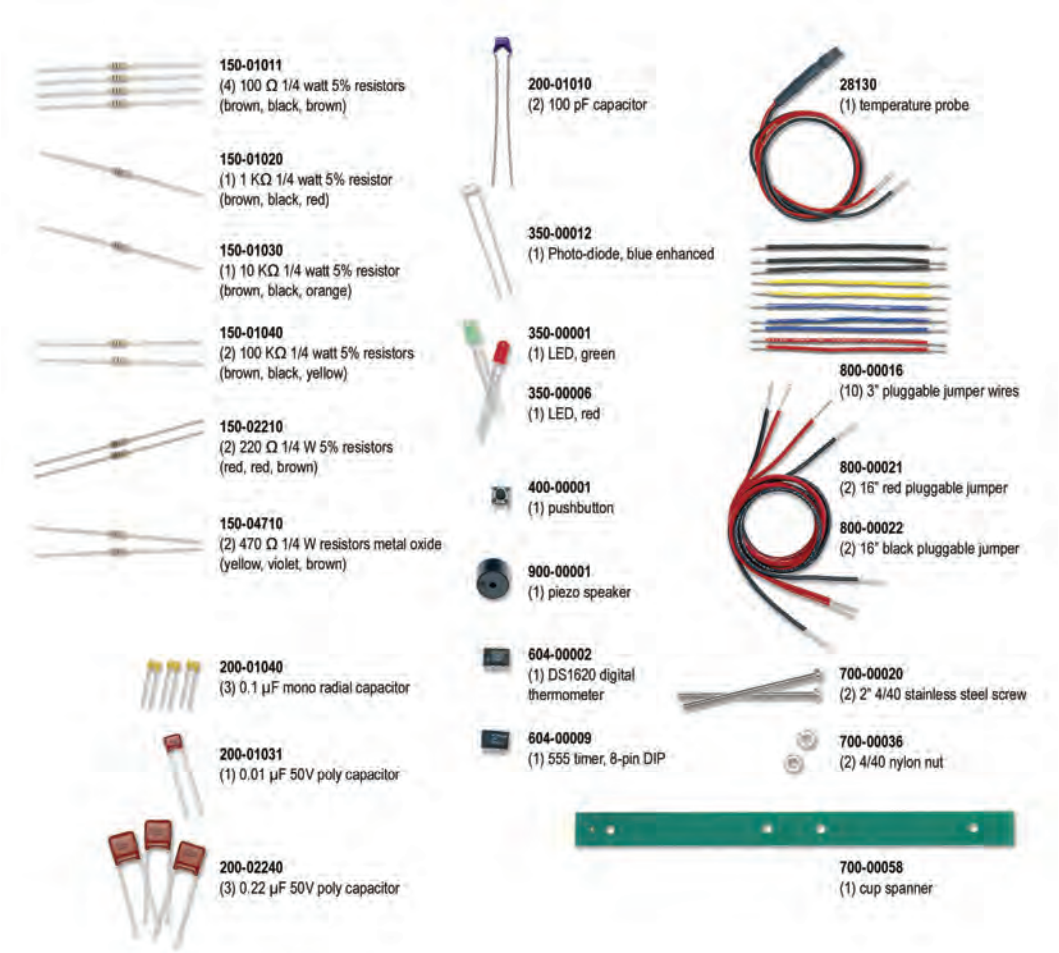
PARAMETER	SYMBOL	VALUE	UNIT
Collector-Base Voltage	V_{CBO}	80	V
Collector-Emitter Voltage	V_{CEO}	25	V
Emitter-Base Voltage	V_{EBO}	5	V
Peak Pulse Current	I_{CM}	20	A
Continuous Collector Current	I_C	4	A
Base Current	I_B	500	mA
Power Dissipation at T_{amb}=25°C	P_{tot}	1	W
Operating and Storage Temperature Range	T_j:T_{stg}	-55 to +200	°C

ELECTRICAL CHARACTERISTICS (at T_{amb} = 25°C unless otherwise stated).

PARAMETER	SYMBOL	MIN.	TYP.	MAX.	UNIT	CONDITIONS
Collector-Base Breakdown Voltage	$V_{(BR)CBO}$	80	120		V	I_C=100µA
Collector-Emitter Breakdown Voltage	V_{CES}	80	120		V	IC=100µA
Collector-Emitter Breakdown Voltage	V_{CEO}	25	30		V	IC=10mA
Collector-Emitter Breakdown Voltage	V_{CEV}	80	120		V	IC=100µA, V_{EB}=1V
Emitter-Base Breakdown Voltage	$V_{(BR)EBO}$	5	8.75		V	I_E=100µA
Collector Cut-Off Current	I_{CBO}		0.3	10	nA	V_{CB}=50V
Emitter Cut-Off Current	I_{EBO}		0.3	10	nA	V_{EB}=4V
Collector Emitter Cut-Off Current	I_{CES}		0.3	10	nA	VCES=50V
Collector-Emitter Saturation Voltage	$V_{CE(sat)}$		30 60 125 155	45 80 180 220	mV mV mV mV	I_C=0.5A, I_B=10mA* I_C=1A, I_B=10mA* I_C=2A, I_B=10mA* I_C=4A, I_B=50mA*
Base-Emitter Saturation Voltage	$V_{BE(sat)}$		890	950	mV	I_C=4A, I_B=50mA*
Base-Emitter Turn-On Voltage	$V_{BE(on)}$		820	900	mV	IC=4A, V_{CE}=2V*
Static Forward Current Transfer Ratio	h_{FE}	250 300 300 200 35	430 450 450 350 70	1200		I_C=10mA, V_{CE}=2V* I_C=0.5A, V_{CE}=2V* I_C=1A, V_{CE}=2V* I_C=4A, V_{CE}=2V* I_C=20A, V_{CE}=2V*
Transition Frequency	f_T		180		MHz	I_C=50mA, V_{CE}=10V f=50MHz
Output Capacitance	C_{obo}		45	60	pF	V_{CB}=10V, f=1MHz
Turn - On Time	t_{on}		125		ns	I_C=4A, I_B=40mA, V_{CC}=10V
Turn -Off Time	t_{off}		380		ns	I_C=4A, I_B=±40mA, V_{CC}=10V

*Measured under pulsed conditions. Pulse width=300µs. Duty cycle ≤ 2%

Index



Parts and quantities in the Applied Sensors kits are subject to change without notice. Parts may differ from what is shown in this picture. If you have any questions about your kit, please contact stampsinclass@parallax.com.