Experiments for the Propeller Quickstart

By Jon Titus

VERSION 1.0

Table of Contents

An Introduction

You've opened this book because you want to learn more about electronics. I won't let you down. If you're a teenager with a keen interest in how things work, you'll enjoy this book and its hands-on lab experiments. Young and not-so-young adults might enjoy working through experiments with you. Enjoy and have fun.

This book takes a different approach to electronics. Instead of building projects, you will learn how electronic components and devices work so you can create projects of your own. The experiments introduce basic electronic concepts and show how to apply them – and other ideas – to electronic circuits. So, instead of simply flashing an LED, for example, you'll learn why it flashes, how the flasher circuit works, how to change the flash rate, how to calculate the values of components for a new rate, and so on. In a later experiment you will use the same circuit to help communicate information over an invisible infrared light beam.

I started to write a short introduction here but decided to add extra information along the way. I hope you find it helpful.

IMPORTANT: None of the experiments in this book require direct connections with mains power – 120 or 240 volts AC. Power supplies, computers, and other devices connect to line voltages, of course. But the maximum voltage used in my experiments never exceeds 12 volts DC. That's the voltage across eight 1.5-volt flashlight batteries connected in series.

Prerequisites

Algebra and numbering systems. As a prerequisite you should have experience with introductory algebra so you can solve an equation such as:

$I = E / R$

for the value R when you know the values of I and E. Some experience with scientific notation ($6.02 * 10^6$) and a bit of experience with binary numbers will help, too. If these topics sound new, refer to the math tutorial at: http://www.mathsisfun.com/numbers/scientific-notation.html for help with scientific notation. Binary numbers get covered at: http://www.mathsisfun.com/binary-number-system.html.

If you haven't used electronic components before, refer to a tutorial that described how diagrams represent circuits and components: https://learn.sparkfun.com/tutorials/how-to-read-a-schematic/schematic-symbols-part-1. People have posted circuit-diagram symbol charts on the Internet, so find one you like and print it as a handy reference. Experiments in this book use only a few symbols, which you'll find easy to learn.

If you have not used a solderless breadboard, go through the SparkFun Electronics tutorial: https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard. Inexpensive breadboards let you wire and test circuits without first soldering components to a printed-circuit board (PCB). And they make it easy to modify and correct circuits.

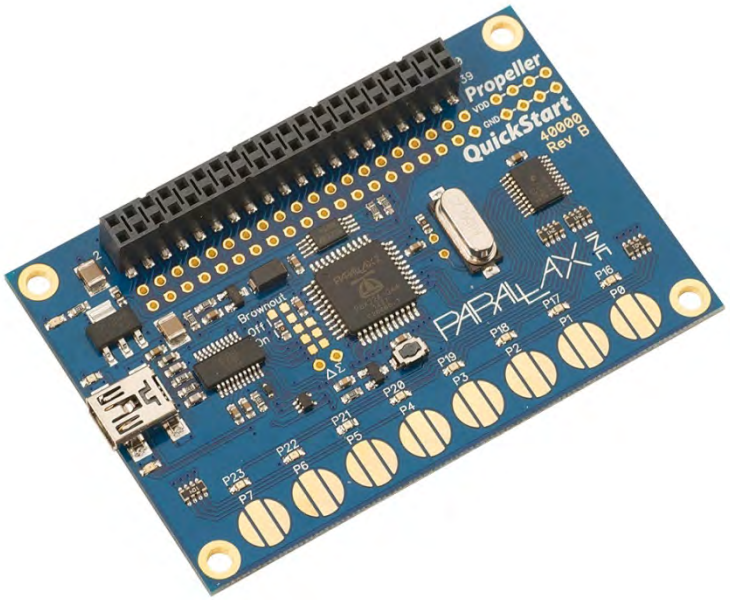
Caveat: My mention of the companies, brands, and products that follows does not imply or offer an endorsement. I have used many of the products but the decision to purchase and use them remains yours alone. Prices and Web-site addresses were current at the time of writing. I cannot guarantee you will always find a site or a product at a specified price or from a mentioned distributor. But the parts required for experiments are available from several sources.

Over the years, my friends Ken and Chuck Gracey at Parallax have given me hardware to experiment with and I greatly appreciate their support. The projects in this book use many components from Parallax and those I recommend highly.

These days engineers, hobbyists, students, and experimenters use small microcontrollers (MCUs) to add "intelligence" to a circuit or product. Many experiments in this book use an MCU and include programs you can run and modify as you wish. I used the Parallax Propeller MCU for the experiments.

Why the Parallax Propeller MCU?

I chose the Parallax Propeller P8X32A board – about $US 32 – for the experiments because I found it so easy to use.



I looked at many MCU boards from semiconductor companies such as Texas Instruments, Freescale Semiconductor, STmicroelectronics, and Microchip Technology. These companies sell many inexpensive MCU boards that help engineers learn about new capabilities, the software "tools" used to write programs, and so on. Unfortunately, these boards didn't offer capabilities that would make them easy for most non-programmers to use, as explained below. No matter which MCU you use, this information might help you make a better-informed decision.

1. Easy to use software. The free "Propeller Tool" software makes programming and debugging easy. The tool includes a Parallax Serial Terminal that lets people easily communicate with the Propeller chip and see results on a PC monitor. The Propeller Tool program runs on a standard PC with an unused USB port. Parallax also has an open-source C compiler for those who like the C language.

2. Easy access to I/O pins through header receptacles. The P8X32A board provide all 32 I/O signals. Although some of these signals connect to LEDs and touch controls on the board, you can still use them as you wish. Parallax sells a "QuickStart Pin Finder" overlay that identifies pin signals on the P8X32A board. Very handy.

3. Easy-to-understand code. I like the Propeller "Spin" language that handles many details behind the scenes. If you want to control four I/O pins as outputs, for example, you need only two statements:

```
DIRA[27..24] := %1111  'Set pins 27 through 24 as outputs
```

```
OUTA[27..24] := %0000  'Set the four outputs to logic 0
```

Information about I/O pins for a large semiconductor company's MCU requires 19 pages in the chip's manual!

4. Nothing extra needed, except for a USB cable. The Propeller P8X32A board gives you eight processors, ready to go. Yes, the Propeller chip includes eight (8) independent processors.

5. Inexpensive. The P8X32A board costs $US 32.

6. Lots of free support and an extensive library of contributed algorithms and code in the Propeller Object Exchange: http://obex.parallax.com/. Several experiments in this book use code from the Exchange. You can freely use the objects in the OBEX. Just give credit where it's due.

Software

Experiments that involve software for a Propeller MCU include complete, commented code listings. The code files are found as a separate download from this text; go to http://www.parallax.com/downloads/experiments-propeller-quickstart-code or search “Experiments for the Propeller Quickstart” in the downloads search page on www.parallax.com. Each experiment has a separate folder for its code, if any. Folders for experiments that do not require code contain a "No software.txt" file so you know nothing has gone missing. The experiment folders include some code I downloaded from the Parallax Object Exchange (OBEX). In a few cases I modified the OBEX code, but you have access to my changes and to the original code from the OBEX site. Some of the comments might vary slightly between those in this book and those in the code due to format restrictions on a printed page. The software remains the same, though.

A few lines of code in the print version of the experiments "wrap" onto the next line when printed in the experiments. You can print programs from the PST window or copy them into a word processor or text editor and print them from there. I recommend you use a print page in the landscape orientation.

You must pay careful attention to the indentation of code statements. The Propeller Tool window shows a gray line to indicate code sections. Use this line to help you understand how code will run. Improper indention can lead to unexpected results.

Thoughts about Troubleshooting

Even the best engineers make mistakes in circuit designs and software, so don't feel badly if an experiment doesn't work properly when you try it. I have checked and rechecked diagrams, explanations and Propeller programs several times, but errors do occur. Check the Propeller Forum for any corrections or updates. Below I offer suggestions you can use to troubleshoot circuits and code:

1. Always ensure you have the proper voltage set on your power supply or supplies BEFORE you make connections. Check the power connections at the breadboard power buses and at components and IC leads. It's easy to forget a power or ground connection, or to be "off by one" in a column of breadboard receptacles when you connect a component with power or ground. I recommend you place a resistor and LED in series between the power and ground power buses to provide a visual indication that you have power turned on. (If you don't know how to do that, you'll see plenty of LED-circuit examples in the experiments.)

2. You MUST have a common ground between circuits unless specifically told otherwise. Lack of a ground in common with circuits will cause unexpected behavior and component failures. Check and recheck ground connections.

3. The LEDs on a microcontroller board simplify troubleshooting. The Propeller P8X32A board comes with eight LEDs ready to use. You can include LED-control statements in a program to let you know when the MCU reaches a specific point in a program. If the chosen LED never turns on, you can move the statement elsewhere to determine where the MCU got "stuck," perhaps in an endless loop, waiting for something to happen, or testing for a condition that never occurs.

We also have a powerful tool in the Parallax Serial Terminal (PST) built into the Propeller Tool software. When you need more than an "MCU reached here" LED, use print statements in your code to display a statement or value in the PST window. In a program you could use statements in a program to turn on LEDs at specific points. An Internet search will uncover a few commercial and freeware Propeller debugging tools available from third parties.

What You Need for the Experiments

Each experiment starts with a list of needed components. And an appendix list the materials needed to do *all* experiments. We call this list a *bill of materials*, or BOM. If you plan to start with only a few experiments, read the requirements list for each and create your own list of parts. Some experiments list specific manufacturers and part numbers. If you choose different suppliers or device models, experiments might not work properly. This is particularly true for infrared-LED experiments 18 through 20.

Every home or school lab should have some basic equipment and I recommend the following items. Don't panic, though. You probably don't need to have them all at once.

Solderless breadboards. Inexpensive and easy to use. Buy the types that include power buses along the top and bottom edges. Some manufacturers color code the power rails so you can quickly identify power (red) and ground (blue) connections. I like that.

A power supply. I like power supplies that have adjustable output voltages from 0 to 15 volts, but experiments can use supplies with a fixed-voltage output, too. An adjustable power supply does not need analog or digital meters – they just add to the cost. You can use a digital multimeter (described shortly) to measure the voltage output. I saw Tekpower DC HY-1502 variable power supplies rated for an output between 1.5 and 15 V volts on Amazon for about $35. They look like decent units. (The description notes, "...ideal for tattoo use." Hmmm.) I would buy two supplies because several experiments require two voltages.

If you don't want to buy power supplies, consider alternatives:

1. Batteries. Buy several battery holders and connect them in series to deliver 3, 4.5, and 6 volts. D-size cells provide a lot of energy. Many 3.3-volt devices work well with a 3-volt supply. Likewise, many

5-volt ICs can operate at 4.5 volts. You can use a 6-volt source such as four D cells to deliver power through a high-current silicon diode. A 1N5400 diode, for example, can pass a current as high as 3 amperes with a 1-volt drop across the diode. *Voila*, a 5-volt output. Ajax Scientific sells D-cell battery holders that clip together in either series or parallel. The holders cost about $US 8 for a package of eight. Look for battery holders at Amazon.

Or add another battery or two to provide 7.5 or 9 volts and then use a 5-volt regulator such as a 3-terminal 7805 to provide a stable 5-volt power output. You'll need a couple of capacitors. Find 7805 circuits on the Internet. A Fairchild data sheet shows a good circuit for a fixed-output regulator: www.fairchildsemi.com/datasheets/LM/LM7805.pdf.

2. Use modular power cubes, called "wall warts" by many hobbyists and experimenters. These cubes plug into a power outlet and deliver a DC voltage. Electronic distributors sell new cubes for about $US 10, but you or friends might have ones you no longer use. Typical outputs include 5, 9, 12, and 24 volts. I have seen several cubes that provide a 3.3-volt output. You want a *linear* regulator. *Switching* regulators can produce a higher voltage than specified when not connected to a circuit. Anecdotal evidence suggests they can damage semiconductors when first connected to a circuit.

3. Use a surplus PC power supply. Late-model supplies have outputs for +3.3, +5, +12, and -12 volts. Find the color code for the outputs on the Internet. See for example: http://www.smps.us/power-connectors.html. In many PC supplies you must connect the green wire to ground (one of the black wires) to turn on the power. In a PC, the green wire (labeled PS_ON#) connects to the PC's on-off switch or an on-off circuit. For more information, run an Internet search for the terms: PC power supply pinout. I found several new supplies on eBay for about $25. A local PC store that does repairs might sell a power supply for less.

I highly recommend color-coded wires for power-supply connections. Most engineers use red wires for power and black wires for all grounds. If you use two or three supplies routinely, use different colors for power from them. If you don't have colored wire, wrap some colored tape on the supply wires at the power supply and use the same color at the other end of the power wire. I often use green wire for +5-volt supply connections, orange for -12 volts, and red for +12 volts. Stay with black for all ground wires.

While on the subject of wires, unless noted otherwise, you MUST have a ground connection in common with all circuits. I cannot emphasize enough that they need a common return path.

Digital multimeter (DMM). You can buy a DMM from Amazon or from an electronics distributor. Good brand name products include those from Extech, Fluke, Triplett, Klein Tools, and Amprobe (formerly Wavetek).

Choose a meter that offers DC voltage scales of 0-to-200 millivolts, 0-to-2 volts, 0-to-20 volts, and possibly higher voltage ranges. For current measurements I saw a meter that has a 0-to-100 milliamp range and a 0-to-10 ampere range with no ranges between. Not worth buying. You need some intermediate current ranges, too. The Klein Tools MM200 Auto Ranging Multimeter looks like a good instrument (see image below). Auto-ranging means the meter determines the best measurement span for an applied signal and the signal polarity – positive or negative. You don't have to move the control after you set it for amperes, volts, resistance, or capacitance. Read customer reviews before you buy a DMM. And make sure the DMM comes with test leads and probes. Probes that come with detachable alligator clips offer a lot of flexibility when you connect a meter to wires.

Oscilloscope. You DO NOT need an oscilloscope, or "scope," to complete any experiment. I have used a Parallax PropScope USB oscilloscope ($US 180 as of this writing) in several experiments to illustrate something important.

You can find other USB-based instruments from many manufacturers and distributors. A search of eBay found over 15,000 listings that include the word oscilloscope, so you can find some bargains among used equipment.

Locate manuals online before you make a purchase and ensure you can return an item for a refund if it doesn't work. You want 1X/10X probes for any scope. The 10X setting attenuates higher voltages by a factor of 10 so their output falls within one of the scope ranges. (In several places I use a screen image from a professional-type scope from Tektronix simply because it can handle four inputs.)

Hand tools. You will need small screwdrivers, a pair of wire strippers, a small pair of needle-nose pliers (4.5 inches, 11.5 cm), and a small diagonal cutter – also called a side cutter – (4 inches, 10 cm). Those length dimensions apply to my lab tools, so you get an idea of size. A pair of forceps (tweezers) can simplify pressing component leads into breadboards. And you might need an adjustable wire stripper. Again, Amazon offers a wide variety of small, inexpensive tools, none of which should cost more than $US 5.00 or 6.00 each. When you clip wires wear safety glasses. A piece of wire in an eye can ruin your day and your sight.

Hookup wire. I recommend either 24- or 22-gauge *solid* hookup wire for solderless breadboards. You can find wire kits that include spools of wire with various insulation colors. Or, you might like a set of pre-stripped breadboard wires in several lengths. You will also find kits of breadboard jumper wires that have pins soldered on each end. Distributors as well as Amazon and eBay sellers offer jumper-wire kits.

Clip Leads. I also recommend you have several clip leads – short pieces of stranded hookup wire with an electrical clip at each end. I use mainly small grabber clips rather than larger alligator clips. Amazon and eBay usually list assortments of clips alone or wired sets of clips. Buy them in several colors so you can quickly distinguish connections from one another. ApogeeKits, for example, sells a set of 10 clip leads for about $9. http://www.apogeekits.com/. I like the miniature grabbers similar to those shown below.

Components. Experiments call for individual LEDs, display modules, resistors, capacitors, transistors, integrated circuits, and a few other devices. Again, read the list of requirements for an experiment, decide whether you want to run the experiment, and check off the components on the BOM. I have purchased components from suppliers such as Parallax, Sparkfun Electronics, Adafruit, Digikey, Mouser, and Jameco. All sell high-quality, reliable components.

Resistors. Instead of buying individual resistors, consider purchasing an assortment. Jameco sells an assortment of 1/4-watt carbon-film resistors with a 5-percent tolerance (gold band) for $US 15. If you want a set of drawers for them, too, the price comes to $US 30. Over the years I have purchased various assortments to replenish my lab supplies. If necessary, you can buy individual resistors of a given resistance.

I recommend you have an inexpensive resistor-substitution box. I found an Elenco Kit, Model RS-400 with 24 resistances that would work well in any lab. All Spectrum Electronics and other companies sells this kit for about $US15. Or you can find instructions on the Internet that explain how to build your own resistor-

substitution box. I like the boxes that use two 12-position rotary switches and a single-pole double-throw (SPDT) switch to select the high- or low-resistance section.

Several experiments require one or two variable resistors, also called trimmer resistors, trimmers potentiometers, or just trimmers. These devices have three terminals; one at each end of the internal resistor, and one that connects to a mechanical wiper. When you move the small control, the wiper moves from one end of the resistor to the other. For a 2000-ohm trimmer, the wiper lets you adjust a resistance from 0 ohms to 2000 ohms over about a 270° rotation. You'll need a small screwdriver to adjust a trimmer. (Engineers and others sometimes refer to trimmer resistors as "trimpots," but Bourns, Inc. owns that word as a trademark. Don't use it unless you refer to a Bourns product.)

Trimmer resistors have a short lifetime – a few hundred turns. Data sheets often list a "rotational life" for trimmers. You won't reach that limit in the experiments, but if a trimmer starts to "act up," it might indicate you need a replacement.

I store components in sets of small drawers and in Plano-brand StowAway compartmented utility boxes with movable dividers. Sporting-goods stores might sell the Plano boxes in their fishing department. (They're great for holding fresh-water flies and lures.) Label compartments to simplify locating part types.

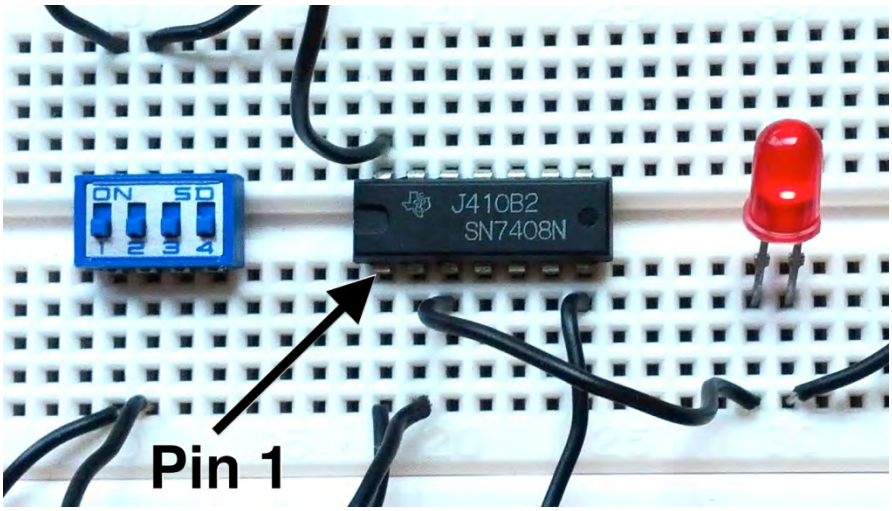
Use Google to locate a color-code chart for resistors and print it so you have a handy reference. After a while you'll know the value of each color, so it becomes easy to pick out a 2200-ohm resistor and to check breadboard circuits for the proper component values. Marlin P. Jones & Associates includes a business-card-size color-code chart with each order for components. I keep one on my lab bench.

Capacitors. Capacitor manufacturers use numbers rather than color codes to indicate capacitance, so you must know how to look at the printed numbers and interpret them. Unfortunately, some numbers can confuse us easily. To ensure you use the proper capacitance in a circuit, I recommend you purchase a DMM that includes capacitance measurements. For a helpful capacitance-marking table, visit: http://www.elecraft.com/Apps/caps.htm.

How Do I Interpret Capacitor Values? Many non-US schematic diagrams use capacitance values of nanofarad, or nF. I use microfarad (μF) throughout this book. To convert a nanofarad value to microfarad units, multiply it by 1000.

CAUTION: Electrolytic capacitors have a polarity market on their package. Always follow directions that explain how to connect electrolytic capacitors because reversing the leads can cause the capacitor to get hot, bulge, and perhaps explode. Always connect the positive lead (marked with a plus sign) to the most positive of the two voltages that connect to the capacitor. Some electrolytic caps mark the negative (minus sign) terminal rather than the positive terminal. And like the orientation of LEDs, the longer lead on an electrolytic capacitor indicates the positive lead.

Integrated Circuits. The integrated circuits used in this book's experiments plug into solderless breadboards, but to complete a circuit you must know how to orient an IC. To help, IC manufacturers include a mark, often a small dimple or mark close to pin number 1. As an alternate, a manufacturer might place a mark at the pin-1 end of ICs. First, all ICs, unless explained otherwise, should straddle the "valley" between the top and bottom halves of a solderless breadboard. Second, always orient an IC in a breadboard so you have pin 1 in the lower-left position as shown in the diagram below.



Courtesy of Electrical Engineering Stack Exchange.

Pin numbers continue in sequence around the outside of an IC in a counter-clockwise direction. Thus, a 14-pin IC has pins 1 through 7 along the lower edge, counter-clockwise from left to right. The numbers continue along the upper edge – again, counterclockwise – with pin 8 across the IC from pin 7, pin 9 across the IC from pin 6, and so on.

If you place an IC in a breadboard incorrectly, you could burn it out when you apply power. Check and recheck IC placement. (I know from experience, having burning out a $360 microprocessor IC!) If necessary, look for IC pin-out diagrams on the Internet through a Google search for the IC's data sheet.

Soldering iron. The experiments in this book require no soldering, so you won't need a soldering iron right away, if at all. I recommend a 60-watt iron with a fine tip, which you can buy for under $US 20. If within your budget, look for an iron that accepts replacement tips. As you get more involved with electronics consider a soldering station that lets you control the tip temperature. You can change this temperature depending on the type of work you do and the types of solder you use. Newer components use a lead-free solder, although you can use a lead-tin solder with them.

Solder. Concerns about lead as a poison have caused some manufacturers to switch to lead-free solders. I still prefer 60-percent tin, 40-percent lead (60-40) solder wire with a rosin core. The rosin helps prevent oxidation and serves as a flux that helps the molten solder "wet" another metal, usually copper, tin, brass, or silver in electronic equipment. To start, choose an ounce (30 grams) of small-diameter solder: 0.025 to 0.030 inches (0.6 to 0.8 mm). Even after constructing many, many circuits I still use the same 1-pound spool of solder purchased over 40 years ago. A small amount goes a long way.

Solder-removal braid. No matter what you solder, you'll eventually have to desolder too. I recommend solder wick that works like a sponge with molten solder. The wick – a braid of fine copper wire – includes rosin that helps pull molten solder off a circuit board, terminal, or contact. For more information, visit Chemtronics. If desoldering tasks involve large contacts, look at manual desoldering suckers or pumps. These spring-loaded devices work like a reverse syringe. Press a trigger and the molten solder gets sucked away. I have used an Edsyn Soldapullt for over 40 years. It still works well. For a less abrupt desoldering tool, look at desoldering rubber squeeze bulbs that give you more control over the amount of vacuum used to suck up solder. Electronics distributors carry a variety of desoldering tools and accessories.

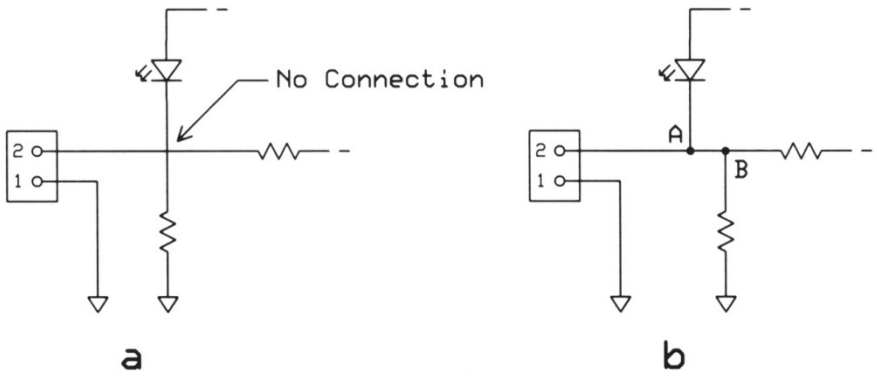
Practice desoldering on an old circuit board with through-hole components. A PC-recycling operation might give you an old motherboard or power supply – or even a complete PC! – that can keep you busy desoldering components for a while. Many municipal recycling operations have a "share" table that lets people pick up useful items for free. You might find an old PC there.

Schematic Diagrams

Just as builders use architectural drawings to build a house, electronic engineers and electronic enthusiasts use schematic diagrams to represent circuits and show connections between components. Special symbols represent LEDs, resistors, capacitors, switches, batteries, meters, and power supplies. Most diagrams in this book include component values to so you can quickly and correctly place and wire connections. Although the electronics industry has standardized symbols, not everyone draws them the same way. The best reference I've uncovered gives you a good overview, certainly sufficient for the experiments in this book as well as for general electronics work: https://learn.sparkfun.com/tutorials/how-to-read-a-schematic.

You can find other symbol charts on the Internet. Many European diagrams use slightly different symbols. When such a diagram includes a 1200-ohm, or 1.2 k-ohm resistor for example, the resistor will often appear with the nearby value 1k2. The k replaces the decimal point and indicates the multiplier (k = kilo = 1000).

To avoid confusion about which signals connect and which do not, diagrams identify connections of signals with a black dot. Nowhere do four or more signals connect at a dot. If you don't see a dot, the signals don't connect; they simply pass "over" or "under" each other. The diagram that follows shows the difference between a connection and a "pass over."



Protect Your Components from Static Electricity

Although many ICs protect themselves against damage caused by static electricity, a static discharge into or through an IC can destroy it. If you do not want to or cannot buy a static-dissipating work surface, always ground yourself when you work with sensitive ICs, transistors, and other semiconductor devices. I use an anti-static work surface that connects to ground through a nearby outlet (in the USA, the round hole in outlets). A similar mat costs about $70. Some mats include a wrist strap that helps reduce the buildup of static electricity on your body.

If you do not use an anti-static mat, at least ground yourself before you start your electronic work. You can buy wrist straps that connect to ground at a mains-power outlet or to another nearby ground connection. Straps cost under $10.

Work Safely

Safety must always take priority over anything else in your work area. You might not think much can happen when you work with tiny electronic components, but think again. My safety suggestions follow:

1. Turn off power supplies and equipment when you don't need them. A forgotten soldering iron could fall over and start a fire. Keep pets out of your work space. They can knock over a soldering iron or play with wires that cause a short circuit or destroy a carefully constructed circuit. I once had a cat jump on my lab table and knock a complicated breadboard circuit on the floor. I had to start over.

2. Protect your eyes. When you cut wires or component leads, wear eye protection, safety glasses, safety goggles, or something similar. You'll find it difficult to create circuits and perform experiments when you lose sight in one or both eyes.

3. Small children and pets can pick up and swallow small components. Keep your work area clean and free of loose components. Put away unneeded components. I used to find resistors, jumper wires, and clip leads on the floor. Now I use a small plastic box for miscellaneous resistors, LEDs and transistors I use again and again in breadboard circuits. My cats cannot get at them.

The Digital I/O Handbook: It's Free

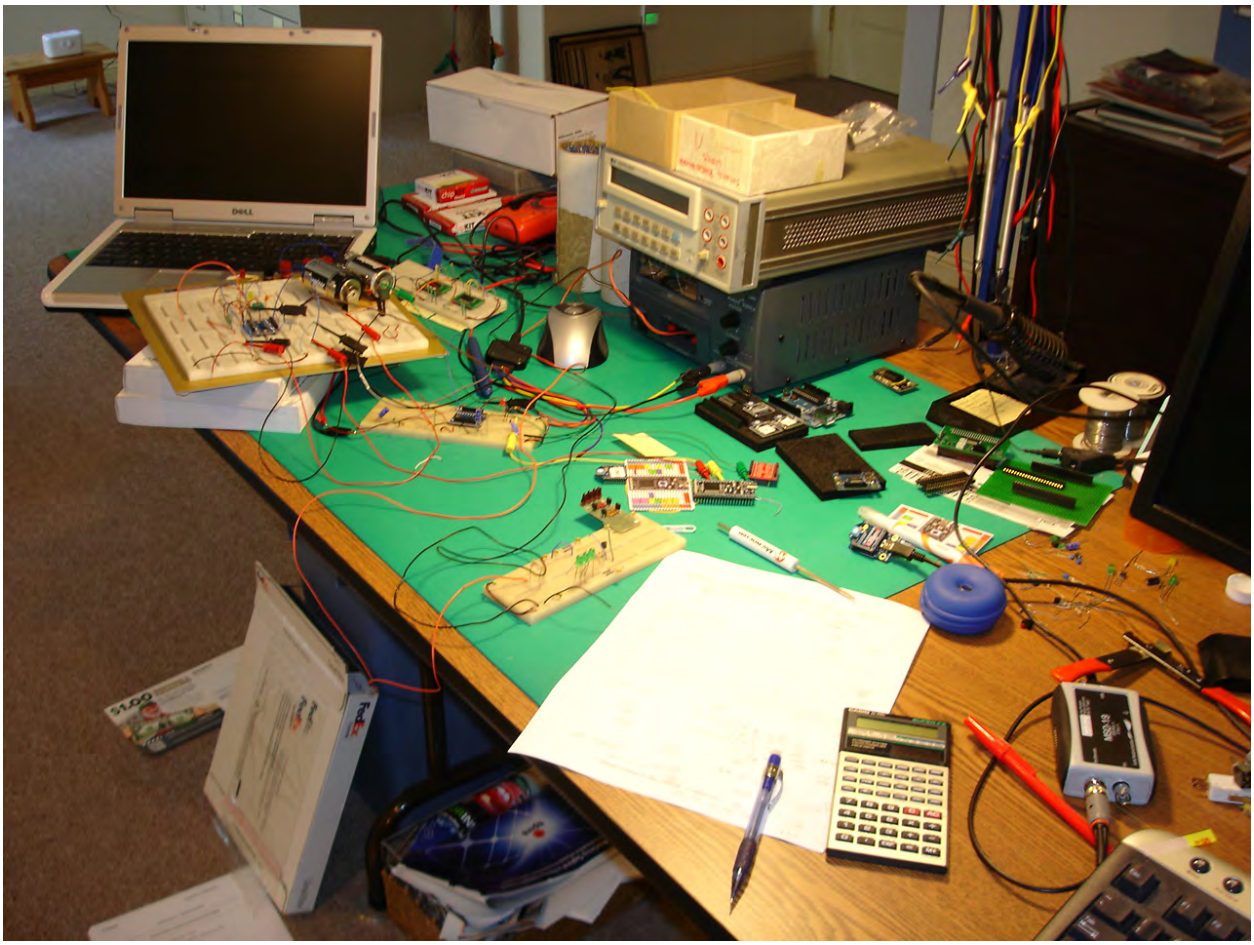
Download a free copy of "The Digital I/O Handbook" from Sealevel Systems at: http://www.sealevel.com/store/accessories/books-references.html. I wrote this book with Tom O'Hanlan, a friend and founder of Sealevel Systems in Liberty, SC. You'll learn more about how to control external devices and how computers recognize when external events occur. You can purchase a spiral-bound copy of the book from Sealevel for $19.95.

Questions or Comments?

I welcome comments and questions from readers via email. Connect with me by email at: jontitus@comcast.net. "Parallax Experiments" will appear in the email subject line. I can't guarantee to answer every message personally but will do my best to help you.

About the Author

During his career, Jon Titus has designed electronic equipment and written code in several languages. He created the Mark-8 Minicomputer, the first hobby computer, which *Radio-Electronics* magazine featured as a construction article in the July 1974 issue. Jon helped start and run several companies involved with educational hardware and teaching materials. For 18 years he worked as the chief editor at electrical-engineering magazines *EDN* and *Test & Measurement World*. He has written articles and columns for many other periodicals and enjoys working with young people who like science and engineering. Jon holds an Extra-class amateur-radio license, KZ1G, and operates regularly from his home in Utah's Salt Lake valley. Jon has a Ph.D. in chemistry from Virginia Polytechnic Institute. (Someday he'll clean up his lab area.)



Experiment No. 1 – How To Measure and Use LED Characteristics

Abstract

This experiment explains several key characteristics of light-emitting diodes and the need to limit current through them. You'll use Ohm's Law to determine the resistance needed to limit current through an LED. Voltage measurements in an LED circuit show how an LED's voltage drop changes as a supply voltage increases or decreases. Other steps let you experiment with LEDs in series and parallel circuits.

Keywords

Light-emitting diode, LED, current, voltage, resistor, resistance, Ohm's Law, parallel LED circuit, series-LED circuit, forward voltage

Requirements

(3 or 4) - LEDs of different colors (red, green, yellow, orange, blue, yellow-green, and so on). Choose LEDs with a 3- or 5-mm diameter and through-hole wire leads
(3) - LEDs, red
(3) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(1) - 1000-ohm, 1/4-watt resistors, 5% (brown-black-red)
(1) - 4700-ohm, 1/4-watt resistors, 5% (yellow-violet-red)
(1) - Solderless breadboard
(1) - Voltmeter and ammeter, a volt-ohm-milliammeter (VOM), or a digital multimeter (DMM)
(1) - Adjustable DC power supply, 0 to 12 volts output

Introduction

The brightness of an incandescent light bulb increases as the voltage applied to it increases. Increase the voltage above the manufacturer's specification, though, and the glowing filament will burn out, or its lifetime will decrease. The brightness of a light-emitting diode (LED), on the other hand, depends on the amount of current that flows through it. Increase the current beyond a specified maximum and the LED might burn out.

You don't usually think about the voltage applied to an incandescent bulb. We simply insert a 110-volt light bulb into a 110-volt lamp socket, or insert a 3-volt bulb into a two-cell flashlight (1.5 volts per cell), for example. But you must think of LEDs in a different way, because all LEDs require an additional component – usually a resistor – to properly limit current flow as specified by the LED manufacturer in a data sheet.

In this experiment you will learn about the information in a data sheet, you will use Ohm's Law to calculate resistance and you will observe how changes in current affect LED brightness. You also will learn how to determine a series-resistance value when you do not have a data sheet for a given LED.

Figures 1.1a and **1.1b** show pages 1 and 8 from a complete data sheet published by Avago Technologies for the HLMP-ELxx family of LEDs (Ref. 1).

HLMP-ELxx, HLMP-EHxx, HLMP-EJxx, HLMP-EGxx

T-1 3/4 (5 mm) Precision Optical Performance AlInGaP LED Lamps

AVAGO TECHNOLOGIES

Data Sheet

Lead (Pb) Free RoHS 6 fully compliant

Description

These Precision Optical Performance AlInGaP LEDs provide superior light output for excellent readability in sunlight and are extremely reliable. AlInGaP LED technology provides extremely stable light output over long periods of time. Precision Optical Performance lamps utilize the aluminum indium gallium phosphide (AlInGaP) technology.

These LED lamps are untinted, nondiffused, T-1 3/4 packages incorporating second generation optics producing well defined spatial radiation patterns at specific viewing cone angles.

These lamps are made with an advanced optical grade epoxy, offering superior high temperature and high moisture resistance performance in outdoor signal and sign applications. The high maximum LED junction temperature limit of +130°C enables high temperature operation in bright sunlight conditions. The package epoxy contains both uv-a and uv-b inhibitors to reduce the effects of long term exposure to direct sunlight.

These lamps are available in two package options to give the designer flexibility with device mounting.

Benefits

- Viewing angles match traffic management sign requirements
- Colors meet automotive and pedestrian signal specifications
- Superior performance in outdoor environments
- Suitable for autoinsertion onto PC boards

Features

- Well defined spatial radiation patterns
- Viewing angles: 8°, 15°, 23°, 30°
- High luminous output
- Colors:
  - 590 nm amber
  - 605 nm orange
  - 615 nm reddish-orange
  - 626 nm red
- High operating temperature: $T_{J\,LED}$ = +130°C
- Superior resistance to moisture
- Package options:
  With or without lead stand-offs

Applications

- Traffic management:
  - Traffic signals
  - Pedestrian signals
  - Work zone warning lights
  - Variable message signs
- Commercial outdoor advertising:
  - Signs
  - Marquees
- Automotive:
  - Exterior and interior lights

Figure 1.1a.
The first page of this Avago Technologies data sheet provides general information for several LED families. The T1¾ package type refers to an LED diameter of 5 mm. You will learn what this information means shortly.
(Courtesy of Avago Technologies. Copyright © 2005-2014 Avago Technologies.)

Absolute Maximum Ratings at $T_A = 25°C$

DC Forward Current [1,2,3]	50 mA
Peak Pulsed Forward Current [2,3]	100 mA
Average Forward Current [3]	30 mA
Reverse Voltage (I_R = 100 µA)	5 V
LED Junction Temperature	130 °C
Operating Temperature	-40 °C to +100 °C
Storage Temperature	-40 °C to +100 °C

Notes:
1. Derate linearly as shown in Figure 4.
2. For long term performance with minimal light output degradation, drive currents between 10 mA and 30 mA are recommended. For more information on recommended drive conditions, please refer to Application Brief I-024.
3. Operating at currents below 1 mA is not recommended. Please contact your local representative for further information.

Electrical/Optical Characteristics at $T_A = 25$ °C

Parameter	Symbol	Min.	Typ.	Max.	Units	Test Conditions
Forward Voltage	V_F					
Amber (λ_d = 590 nm)			2.02		V	I_F = 20 mA
Orange (λ_d = 605 nm)			1.98	2.4		
Red-Orange (λ_d = 615 nm)			1.94			
Red (λ_d = 626 nm)			1.90			
Reverse Voltage	V_R	5	20		V	I_R = 100 µA
Dominant Wavelength	λ_d					
Red		620.0	626.0	630.0	nm	I_F = 20 mA
Amber		584.5	590.0	594.5		
Orange		599.5	605.0	610.5		
Red Orange		612.0	615.0	621.7		
Peak Wavelength:	λ_{PEAK}					
Amber (λ_d = 590 nm)			592		nm	Peak of Wavelength of Spectral Distribution at I_F = 20 mA
Orange (λ_d = 605 nm)			609			
Red-Orange (λ_d = 615 nm)			621			
Red (λ_d = 626 nm)			635			
Spectral Halfwidth	$\Delta\lambda_{1/2}$		17		nm	Wavelength Width at Spectral Distribution $^1/_2$ Power Point at I_F = 20 mA
Speed of Response	τ_s		20		ns	Exponential Time Constant, $e^{-t/ts}$
Capacitance	C		40		pF	V_F = 0, f = 1 MHz
Thermal Resistance	$R\theta_{J\text{-}PIN}$		240		°C/W	LED Junction-to-Cathode Lead
Luminous Efficacy[1]	η_v					
Amber (λ_d = 590 nm)			480		lm/W	Emitted Luminous Power/ Emitted Radiant Power
Orange (λ_d = 605 nm)			370			
Red-Orange (λ_d = 615 nm)			260			
Red (λ_d = 626 nm)			150			
Luminous Flux	φ_V		500		mlm	I_F = 20 mA
Luminous Efficiency [2]	η_e					
Amber			12		lm/W	Emitted Luminous Flux/ Electrical Power
Orange			13			
Red-Orange			13			
Red			13			

Note:
1. The radiant intensity, I_e, in watts per steradian, may be found from the equation $I_e = I_v/\eta_v$ where I_v is the luminous intensity in candelas and η_v is the luminous efficacy in lumens/watt.
2. $\eta_e = \varphi_V / I_F \times V_F$, where φ_V is the emitted luminous flux, I_F is electrical forward current and V_F is the forward voltage.

Figure 1.1b.
Page eight from the LED data sheet includes electrical and optical specifications. Note that all measurements occurred at 25°C, or 77°F.
(Courtesy of Avago Technologies. Copyright © 2005-2014 Avago Technologies.)

The data-sheet page shown in **Figure 1.1a** lists several LED physical features that include:

a) Viewing angles (8, 15, 23, and 30 degrees)
These angles refer to the angle of one side of a cone that emanates from the front of an LED and contains most of the transmitted light. The axis of the cone has the same axis as the LED. A viewing angle of eight degrees corresponds to a subtended angle of 16 degrees, as shown in **Figure 1.2**.

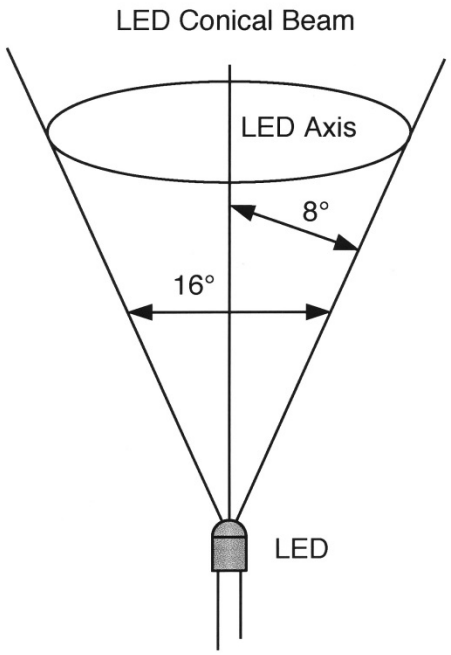


Figure 1.2.
Light from an LED usually remains in a cone-shaped beam with a side-to-side angle and an axis-to-side angle of half that value. Data sheets provide this information that helps designers choose LEDs with a specific beam angle for different applications. You might need a narrow beam in an entry-detection circuit but a wider beam in illumination equipment or signs.

b) Colors (590 nm amber, 605 nm orange, 615 nm reddish-orange, and 626 nm red)
The HLMP-ELxx family of LEDs offers four colors. The notation *nm* refers to nanometers and the number refers to the dominant wavelength produced by the LED. Some data sheets note the wavelength with the lowercase Greek letter lambda, followed by a subscript lowercase d; for example λ_d, where lambda stands for wavelength and the d stands for dominant.

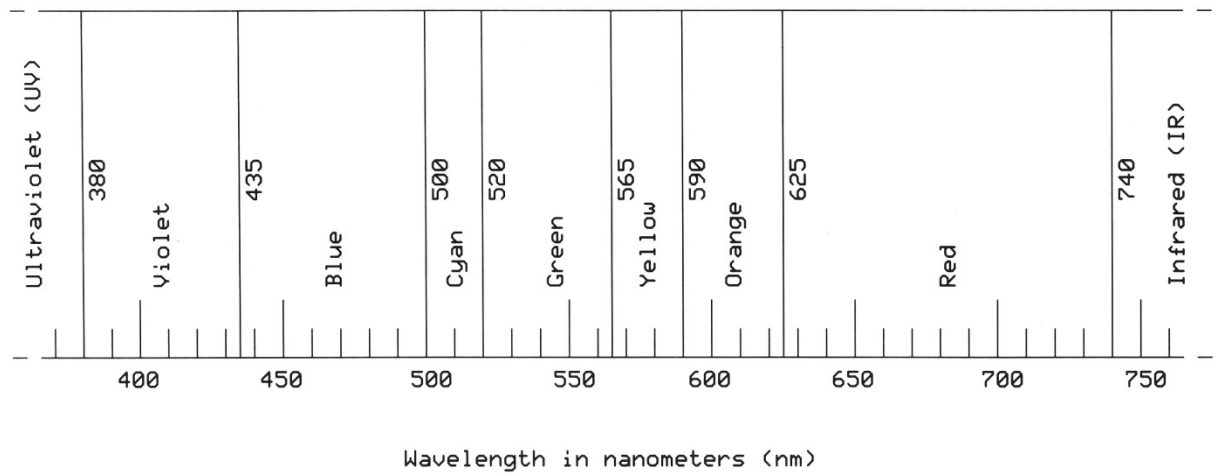


Figure 1.3.
This diagram illustrates the visible-light wavelengths and assigns a color name to each wavelength group. In general, manufacturers' LED descriptions use these or similar color designations.

c) LED junction temperature (T_{jLED} = 130°C). The T_{jLED} value specifies the maximum temperature at which you should operate the LED semiconductor device; that is, the semiconductor LED junction temperature. What types of products might require an LED to operating at a high temperature? Find my list at the end of this experiment.

d) The second page (**Figure 1.1b**) from the Avago Technologies data sheet for the HLMP-ELxx family of LEDs data sheet provides electrical characteristics. These values separate into Absolute Maximum Ratings at an operating temperature of 25°C, and Electrical/Optical Characteristics at the same temperature (25°C = 77°F). The key specifications here include:

Maximum DC forward current (I_F)	= 50 mA (0.050 A)
Average forward current (I_F)	= 30 mA (0.030 mA)
Forward voltage (V_F):	
a. Amber	= 2.02 volts (typical)
b. Orange	= 1.98 volts (typical)
c. Red-orange	= 1.94 volts (typical)
d. Red	= 1.90 volts (typical)

How Do We Power LEDs?

The semiconductor junction in an LED lets current flow in only one direction – from the anode (+) terminal through the LED and out the cathode (-) terminal. If you reverse the connection, current will not flow, at least at low voltages you'll work with in this and following experiments. LEDs with wire leads easily mount in solderless breadboards, but how can you tell the anode from the cathode? Manufacturers agreed on standard ways to identify wire terminals on a 2-lead LED package such as that shown in **Figure 1.4.** The longer of the two leads is the anode (+) terminal and the shorter lead is the cathode (-). Also, the flat spot on an LED package indicates the position of the cathode. When you use LEDs, the anode always goes to a voltage greater than the voltage connected to the cathode.

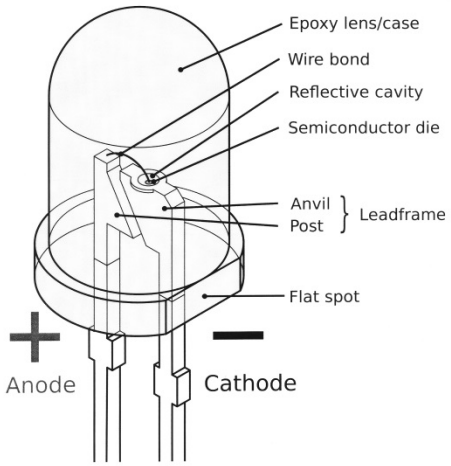


Figure 1.4.
This diagram shows the anode and cathode connections to a 2-wire LED in a plastic package. Some LEDs have a clear-plastic lens or case, while others employ a translucent plastic to disperse the light.
(Courtesy of Wikipedia.)

In a schematic diagram you will see an LED represented as shown in **Figure 1.5.** The anode terminal connects to the flat side of the triangle. The cathode connects to the flat line at the apex of the triangle. The triangle indicates the direction of current flow – anode to cathode.

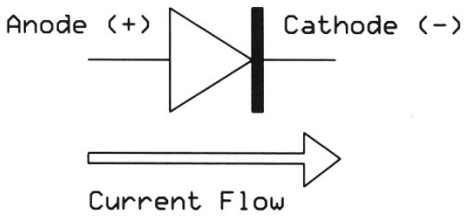


Figure 1.5.
This symbol represents a diode in a circuit diagram, or schematic diagram. Most diagrams do not label the cathode or anode. Nor do they use a plus or minus sign. You will quickly learn to associate the triangle with the anode and the flat terminal with the cathode.

As a young experimenter I remembered a cathode as a minus connection because I could mentally "straighten the letter "C" into a minus-like straight line. On the other hand, I could mentally push in the sides of the letter "A" in anode to create a plus sign. You can do the same thing for the straight line (cathode, minus) and the triangle (anode, plus) in the diode symbol. Or just draw a diode symbol, label the connections and keep it handy.

Step 1.

In this step you will use Ohm's Law to calculate a current-limiting resistance for a red LED in the Avago HLMP-ELxx family. The diagram in **Figure 1.6** shows a circuit that powers the LED from a 9-volt power supply or battery as current flows through a resistor, R. You will not build this circuit; it serves only as an example

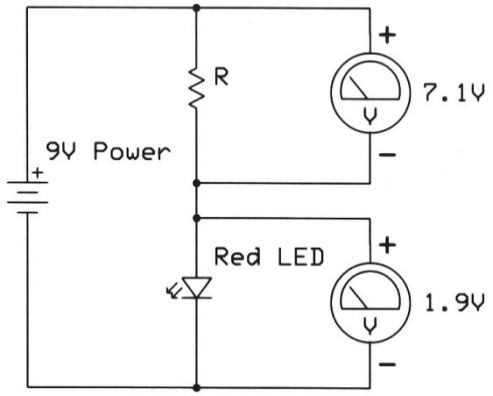


Figure 1.6.
This test circuit lets you measure the voltage across a resistor and an LED. Although the diagram shows two voltmeters, you could use one meter and move its leads from the resistor to the LED and vice versa.

The *forward voltage* (V_F), also called the *forward voltage drop*, refers to the voltage you would measure across the LED in a powered circuit, that is, from the LED anode (+) to the cathode (-). The type of semiconductor materials used to create an LED determine the forward-voltage drop. I'll use the term *forward voltage* from now on.

The information in the Avago data-sheet pages (**Figures 1.1a** and **1.1b**) Avago lists the forward voltage for the red LED shown in the **Figure 1.6** circuit as 1.90 volts. Thus the resistor must have a voltage difference across it of 9.0 volts minus the 1.90 volts across the LED, or 7.1 volts across the resistor. The LED and resistor act as a voltage divider that splits the 9-volt supply into two voltages that must add up to 9 volts; 1.90 volts plus 7.1 volts.

The LED data sheet lists a 50-milliampere (mA) maximum forward current for the LEDs, and a 20 mA forward current (I_F) used during LED tests. Let's assume we want a 30 mA forward current in our test circuit.

In Ohm's Law, $I = E/R$; that is, current (I) equals voltage (V) divided by resistance (R). You can use basic algebra and rearrange this equation to $R = E/I$.

We know the circuit will have 7.1 volts across the resistor and we need a 30 mA (0.030 ampere) forward current through the LED. Because the resistor and LED form a series circuit, the current passing through the resistor equals the current that passes through the LED.

Now, calculate the resistance value (in ohms) needed to limit the current to 0.030 A (30 mA) for the red LED. Ohm's Law uses units of amperes, volts, and ohms. You must use those units and not milliamps, microvolts, and so on, which can lead to errors.

$$R = E \,/\, I$$

$$R = (V + \, - V_F) \,/\, I_F$$

$$R = (9.0\,V \; - \; 1.9\,V) \,/\, 0.030\,A \quad \text{or} \quad R = 7.1V \,/\, 0.030\,A$$

$$R = 236 \text{ ohms}$$

A 236-ohm resistor in series with the red LED limits current to 0.030 amps, or 30 milliamps (mA). Unfortunately you cannot buy an inexpensive 236-ohm resistor, but you can get close enough with a 240-ohm resistor, or even a 220- or 270-ohm resistor.

Use Ohm's Law to calculate the current-limiting resistance needed in series for an amber LED (V_F = 2.02 volts) with a 12-volt power supply. Assume for this LED I_F = 25 mA (0.025 A). Can you locate a resistor with this value? Look on distributor Web sites:

- Digi-Key at www.digikey.com
- Mouser Electronics at www.mouser.com
- Jameco Electronics at www.jameco.com

Step 2.

Suppose you don't have a data sheet for an LED. Can you measure its characteristics? Set your power supply for 5.0 volts. Connect an LED of your choice and a 1000-ohm (brown-black-red) resistor in series on a solderless breadboard as shown in **Figure 1.7**.

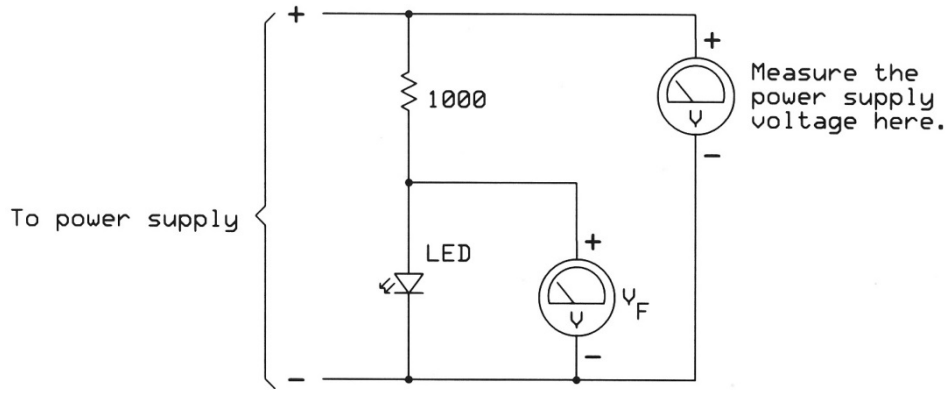


Figure 1.7.
A test circuit for a red LED with unknown characteristics.

I recommend you try this experiment with several types and colors of LEDs. The table for your experimental data on the next page includes spaces for measurements with as many as six LEDs and four test voltages, but you can test only two or three LEDs, if you choose.

Turn on power to your breadboard circuit. The LED should turn on. If it does not, recheck your connections and ensure you have the LED properly oriented so its cathode (-) connects to ground and its anode (+) connects to the 1000-ohm resistor.

Adjust your power supply so its voltage equals one of the voltages listed across the top of **Table 1.1**. You can check your supply voltage with a voltmeter connected between the power supply positive and negative (or ground) terminals. Then measure the voltage across the LED (as shown in **Figure 1.7**). Record your LED forward voltage (V_F) measurements in **Table 1.1** for several power-supply voltages and LED types.

I recommend you use at least two voltages from the following set: 5, 6, 9, and 12 volts. These voltages correspond to those commonly available from power supplies. Use other voltages if you wish. I found it easier to set a supply voltage, measure the voltage across an LED, and then substitute the next LED in the circuit and measure the voltage across it, and so on for all LEDs. Then change the power-supply voltage and repeat the process.

Table 1.1. LED forward voltages for several power-supply voltages.

		Voltage (volts)			
LED	**Color**	**5.00**	**6.00**	**9.00**	**12.00**
LED-1					
LED-2					
LED-3					
LED-4					
LED-5					
LED-6					

Step 3.
You should see the voltage you measure across each LED type remain fairly constant, regardless of the power-supply voltage. The information in **Table 1.2** provides the voltages I measured for six LEDs, listed by color.

Table 1.2. Measured LED voltage across LEDs for several power-supply voltages.

		Voltage (volts)			
LED	**Color**	**5.00**	**6.00**	**9.00**	**12.00**
LED-1	Red	1.99	2.04	2.18	2.32
LED-2	Green	2.00	2.05	2.17	2.27
LED-3	Yellow	1.93	1.97	2.07	2.17
LED-4	Orange	1.84	1.88	1.99	2.09
LED-5	Blue	2.90	2.98	3.17	3.31
LED-6	White	3.38	3.46	3.46	3.77

The data in **Table 1.2** shows an LED's forward voltage increases slightly in these tests as the voltage increases. Your data should show a similar change. As you increase the power-supply voltage, what happens in the circuit? For now, assume we use a green LED with a 2.00-volt forward voltage and this voltage does not change. For the 5-volt test, the 1000-ohm resistor has 3.00 volts across it and a current flow of:

$I = E/R$ or $I = 3.00 / 1000$ or $0.003\ amperes\ (3\ mA)$

For the 12-volt test, the 1000-ohm resistor has 9.00 volts across it, for a current of 0.009 amperes (9 mA). As the current increases through a diode, the diode's forward voltage drop (V_F) increases. A manufacturer's LED data sheets should include a graph that shows the relationship of forward current to forward voltage drop. **Figure 1.8** plots this information for the red and amber LEDs in the Avago Technologies' HLMP-ELxx LED family.

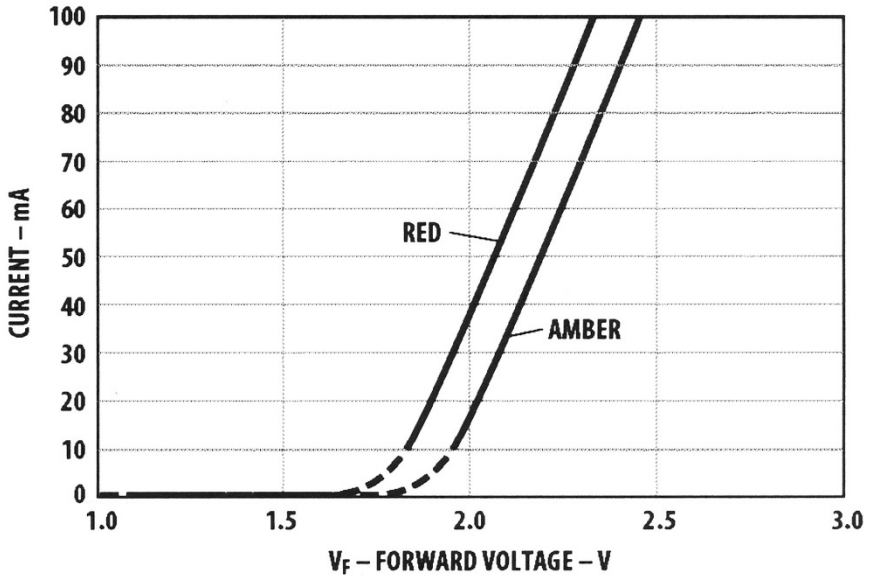


Figure 1.8.
This plot of test data shows the increase in LED forward voltage drop as forward current increases. Engineers who create commercial circuits must take this relationship into account to conserve power. Note the linear (straight-line) response when the forward current exceeds about 10 mA.
(Courtesy of Avago Technologies.)

Also, because the forward voltage varies with current, some LED data sheets provide a range of values for LED forward voltages as shown in **Table 1.3** for the LEDs I used to make the measurements shown earlier in **Table 1.2**.

Table 1.3. Specifications for the LEDs from their data sheets.

LED	Color	VF Range (volts)	IF (mA)
LED-1	Red	1.7 - 2.6	20
LED-2	Green	2.1 - 2.6	30
LED-3	Yellow	2.1 - 2.6	20
LED-4	Orange	1.7 - 2.6	20
LED-5	Blue	3.50	30
LED-6	White	3.6 - 4.0	30

Step 4.
Given the measured voltage (V_F) across an LED and the power-supply voltage (V+), you can calculate the series resistance needed to limit the current to a particular value. Without a data sheet, though, you must experiment to find a suitable resistance needed to limit current through an LED.

You can start with a high resistance in series with your "data-less" LED and see what happens. Most LEDs will "turn on" and create some light with only a few milliamperes, so even with a high series resistance you will see some light. Decreasing the resistance increases light intensity. So you want to find a resistance that provides maximum – or near maximum – light output for the LED. I recommend you use a resistor-substitution box with slide switches or rotary switches that let you quickly change the resistance in a circuit. I use an old EICO Model 1100 RTMA Resistance Box (**Figure 1.9**) that offers 36 values, from 15 ohms to 10 megohms. I start with a high resistance in series with an LED and decrease the resistance one step at a time. As the resistance decreases, the LED gets brighter until decreasing the resistance further causes no increase in

brightness. At that point I can use the selected resistance, or a slightly higher value, in a circuit with the LED. You reach a point at which decreasing the series resistance, and thus increasing the current through an LED, does not increase LED brightness; it only causes the LED to generate more heat. Too much current will damage an LED.

Figure 1.9.
A typical resistor-substitution box that offers 36 resistances between 15 ohms and 10 megohms.

You can buy an inexpensive resistance substitution box or kit, although many have only 24 resistances. Some substitution boxes use slide switches that create a sum of resistances, but it takes several switch changes to go from, say, 2200 ohms to 1000 ohms. As an alternative you can set up in a breadboard all 36 resistors – or only a few – with the values shown in **Table 1.4**.

Table 1.4. Standard resistor-substitution values in Ohms.

15	150	1500	15K	150K	1.5M
22	220	2200	22K	220K	2.2M
33	330	3300	33K	330K	3.3M
47	470	4700	47K	470K	4.7M
68	680	6800	68K	680K	6.8M
100	1000	10K	100K	1M	10M

K = x 1000
M = x 1,000,000

You would connect one side of each resistor to a common conductor that connects to your circuit. Then you can quickly move a wire from the unconnected end of one resistor to the unconnected end of another resistor to change the resistance value in your circuit. The schematic diagram in **Figure 1.10** shows this type of resistor arrangement as a circuit diagram, and the image in **Figure 1.11** shows the a range of resistors on a breadboard. You'll find this type of resistor-substitution arrangement, or a resistor-substitution box, helpful in many experiments.

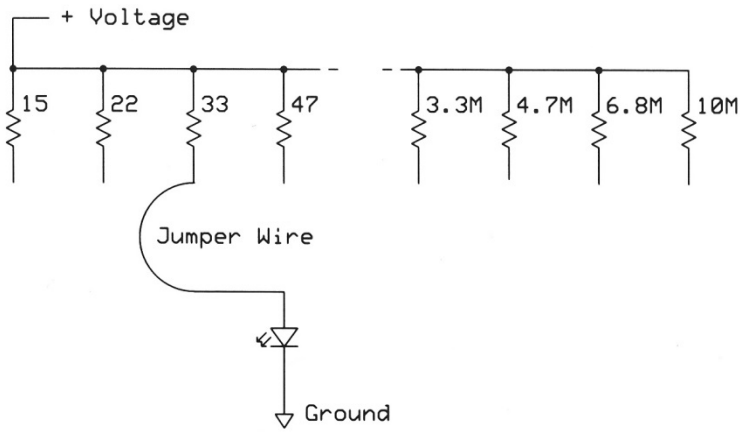


Figure 1.10.
Circuit diagram of a resistor-substitution circuit (all resistances in units of ohms). A jumper wire connects an LED or other device to a chosen resistance. In this example, the connection in common with all resistors goes to a +V power supply. It could go to other circuit points instead, as needed.

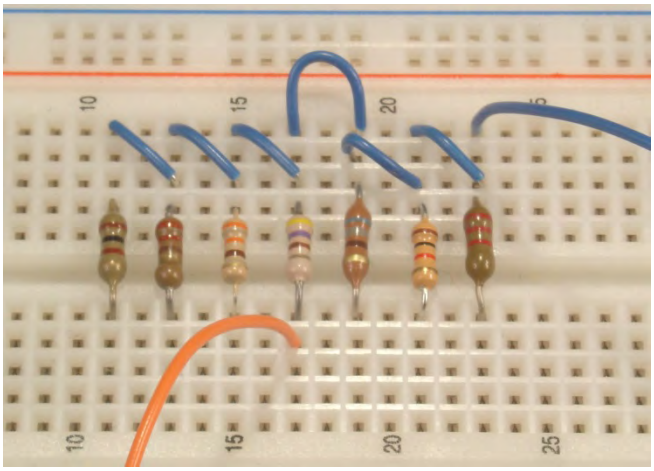


Figure 1.11.
Seven resistors of different values connected for use as a resistor-substitution circuit. The resistors have one terminal in common with each other (blue jumper wires). Their other terminal remains unconnected so I can choose a resistance as needed with the orange wire.

Step 5.

To show the effect of current on LED brightness, set your power supply for a 9-volt output. Place two red LEDs of the same type or model near each other in a breadboard. Use a 330-ohm resistor (orange-orange-brown) as the current-limiting resistor for one LED and use a 4700-ohm resistor (yellow-violet-red) for the second red LED, as shown in **Figure 1.12**. Assume the LED V_F equals 2.2 volts.

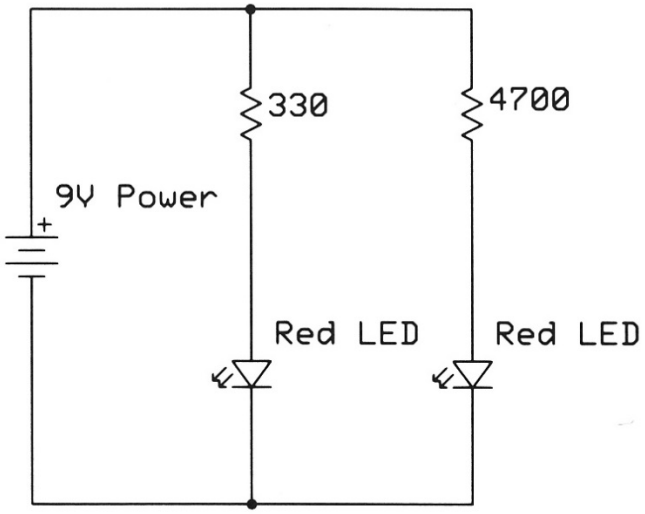


Figure 1.12.
A circuit that delivers a different current to two of the same type of LED.

Turn on power to the LED circuits. Do you see any intensity difference in the light from the two LEDs? How much current flows through the 4700-ohm resistor?

The 4700-ohm resistance decreased the current through its LED in series by more than 10 fold, but you can probably still see light from this LED. Not all circuits must drive an LED with the typical current specified in a data sheet, and a larger resistance helps save power. Keep your 2-LED circuit powered as you go to the next step.

Step 6.

When you calculate or determine a series resistance for use with an LED, you also must consider how much power the resistor dissipates. Continue to power the two LEDs for two or three minutes. Carefully feel the 4700-ohm resistor. Does it feel warm? Next, feel the 330-ohm resistor. Does it feel warmer than the 4700-ohm resistor?

The smaller resistance passes a higher current and thus it creates the heat you feel. This energy gets wasted and in a battery-powered circuit the batteries would require more frequent replacement or recharging. By powering the LEDs with a lower current, you reduce power use and create less heat. But you can still see the LEDs. Keep this trade off in mind when you create circuits.

Step 7.

Resistors also have a power rating, specified in watts, and the circuits in this experiment use 1/4-watt (W) resistors. That means they can dissipate 1/4 W of energy. Now you must determine whether a resistor chosen to limit LED current can handle the heat created as current flows through it. In an article, "Calculating Current Limiting Resistor Values for LED Circuits," author Mark Dobrosielski explains that many people forget to calculate how much power the resistor in an LED circuit will dissipate (Ref. 2). For the 330-ohm resistor used in Step 6, you have a current of 0.020 A, or 20 mA. The following equation lets you calculate the dissipated energy (watts, W) based on a given resistance (ohms, R) and the square of the current (amperes, I) that passes through it:

$W = I^2 * R$ or $W = I * I * R$

Working with a 330-ohm resistance and a 20-mA current flow, you have:

$$W = 0.020\ A^2 * 330\ Ohms = 0.13\ W$$

So when you examine resistor specifications you would look for a 330-ohm resistor with a power dissipation of greater than 0.13 watts. A 1/4-watt, or 0.25-W, resistor will do the job, even though it feels a bit warm. A resistor with a lower power rating would get too hot and fail. **Figure 1.13** shows several resistors with different power ratings.

Figure 1.13.
A variety of resistors with power ratings from 1/4 to 10 watts. All experiments in this book use 1/4-watt resistors similar to the four in the bottom of this photograph.

Step 8.

People who have not used LEDs in circuits often ask if several LEDs can share one resistor to limit current. This step will help you determine the answer. Take two LEDs of the same type and connect one of the LEDs in your breadboard as shown in **Figure 1.14** with a 4700-ohm resistor (yellow-violet-red) that goes to 9-V power.

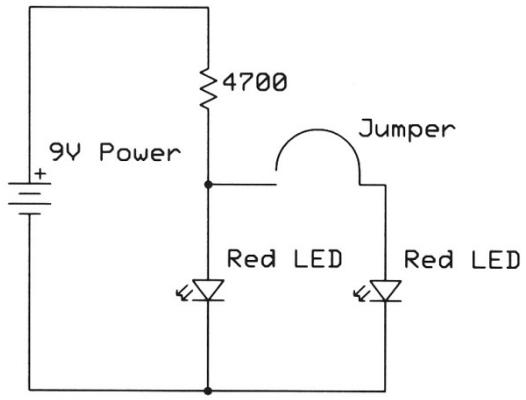


Figure 1.14.
Test circuit for LEDs that share a current-limiting resistor.

Connect the second LED cathode (-) to ground and connect a jumper wire to the LED anode (+). Leave the other end of the jumper fire unconnected for now. Given your measurements for this LED (see data in Table 1.1) about how much current will flow through the resistor and connected-LED circuit? Use Ohm's Law.

I calculated 0.0014 amperes, or 1.4 mA, and measured about 1.5 mA. Turn on power and the LED should light. Connect the free end of the jumper wire to the 4700-ohm resistor where it connects to the lit LED. Did you see any change in brightness in the powered LED when the second LED turned on? Disconnect and reconnect the end of the jumper several times. Did you observe a change in brightness of the always-on LED.

Both LEDs turned on and in a darkened room I saw a slight decrease in light from the already-lit LED. Because both LEDs have about the same forward voltage, they both turn on. But the 4700-ohm resistor continues to limit the *total current* through both LEDs to the 1.5 mA only one LED would draw from the power supply. Thus each LED receives only 0.75 mA, and each LED gets about half the current. So, one series resistor will work in this situation, although brightness decreased somewhat.

Step 9.

Now repeat the process in Step 8, but use a series resistance of 330 ohms (orange-orange-brown) and two LEDs that have different V_F voltages, as shown in **Figure 1.15**. Keep the power source set for 9 volts. Record your data in **Table 1.5**. What happened?

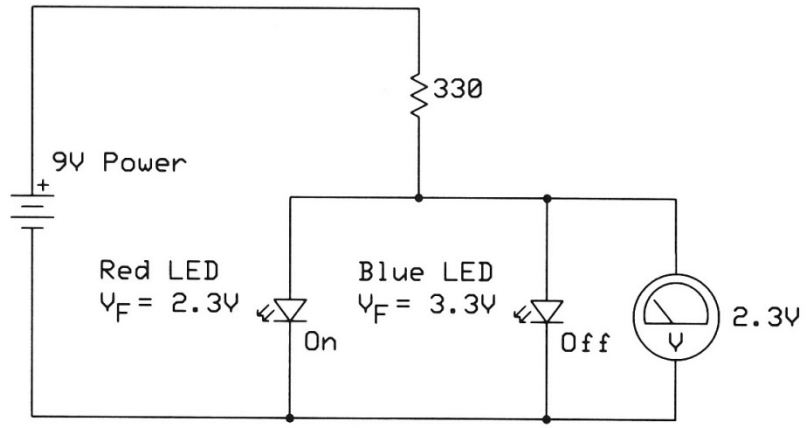


Figure 1.15.
Circuit used to measure the voltage across two LEDs, each with different forward-voltage (V_F) value.

Table 1.5. Experimental results for parallel LED test.

	Color	V_F	On of Off?
LED-1			
LED-2			

I used a red LED with V_F = 2.3 V in parallel with a blue LED with V_F = 3.3 volts and found:

	Color	V_F	On of Off?
LED-1	Red	2.3	On
LED-2	Blue	2.3	Off

The red LED with the lower V_F value "pulls" the connection between the LED anodes and the 330-ohm resistor down to 2.3 volts. This voltage falls below the 3.3 volts needed for the blue LED, so no current will flow through it.

Repeat this experiment with two LEDs that have similar V_F values. Now what do you observe?

	Color	V_F	On or Off?
LED-1			
LED-2			

I used a red and an orange-red LED and observed the results shown in the table that follows:

	Color	V_F	On or Off?
LED-1	Red	2.3	On
LED-2	Orange-Red	2.1	On

Even though these two LEDs have a slightly different V_F value, they both turned on because they can operate over a *range* of V_F values. The red LEDs I had on hand operate with V_F between 1.7 and 2.6 volts, and my orange-red LEDs had a similar range. I found the same results – both LEDs turned on – when I placed a red LED in parallel with a yellow or a green LED with a similar V_F range. This parallel configuration, though, makes it difficult to accurately control LED intensity. In almost all cases I recommend one current-limiting resistor per LED. These resistors do not cost much.

Step 10.

Could you use one current-limiting resistor with LEDs connected in series? Try an experiment to find out. Set up three of the same type of LED as shown in circuit diagram **Figure 1.16**. You have a single LED and a 330-ohm resistor between power and ground and a separate circuit with two LEDs and a 330-ohm resistor between power and ground.

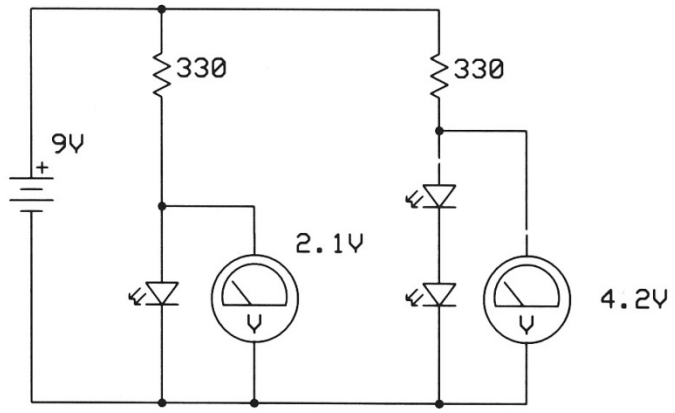


Figure 1.16.
Test circuit for two LEDs in series with a resistor and one LED with its own resistor. All three LEDs have the same V_F or 2.1 volts.

What happens when you turn on power? All three LEDs should turn on. How much current flows through the single LED?

$$I = E / R$$

$$I = (V + - V_F) / 330\,ohms = (9\,V - 2.1\,V) / 330\,ohms$$

$$I = 6.9\,V / 330\,ohms$$

$$I = 21\,mA$$

How much current flows through the two LEDs? Because you have them in series, you must add their V_F values. You would measure 4.2 volts across *both* LEDs and 4.8 volts across the resistor.

$I = E / R$

$I = [V + - (V_{F1} + V_{F2})] / 330\ ohms = (9\,V - 4.2\,V) / 330\ ohms$

$I = 4.8V / 330\ ohms$

$I = 14\ mA$

A smaller current now flows through the two LEDs in series. To have 20 mA flow through the two LEDs in series, calculate a new resistance value for the series circuit. Again, use Ohm's Law:

$I = E / R$

which rearranges to $R = E / I$.

Then:

$R = [V + - (V_{F1} + V_{F2})] / 0.020\,A = (9\,V - 4.2\,V) / 0.020\,A$

$R = 4.8V / 0.020\,A$

$R = 240\ ohms$

I didn't have a 240-ohm resistor handy, so I used a 220-ohm resistor. Current through the two LEDs in series measured 21 mA. Now the single LED and the two LEDs in series have about the same intensity. Remember, in a series circuit, the same current flows through each of the circuit components. And you need a voltage higher than the sum of the forward voltages for all LEDs in a series "string."

References

1. "HLMP-ELxx, HLMP-EHxx, HLMP-EJxx, HLMP-EGxx T-1¾ (5 mm) Precision Optical Performance AlInGaP LED Lamps," document AV02-0373EN.pdf, Avago Technologies: http://www.avagotech.com/docs/AV02-0373EN.

2. Dobrosielski, Mark, "Calculating Current Limiting Resistor Values for LED Circuits," *Nuts & Volts*, January 2005, pp. 50 – 52. www.nutsvolts.com.

Answers

Experiment 1, Introduction:

Answer: traffic signals, vehicle brake lights, instrument panels, toaster ovens, electric kettles, clothes dryer, hair dryer, and so on.

Experiment 1, Step 1:

$R = E / I$

$R = (V + - V_F) / I_F$

$R = (12.0\,V - 2.02\,V) / 0.025\,A \quad or \quad R = 9.08\,V / 0.025\,A$

$R = 363\,ohms$

You can buy a 360-ohm resistor.

Experiment 1, Step 5:

The LED connected to the 4700-ohm resistor should look a bit dimmer.

$I = E / R$

$I = (V + - V_F) / R$

$I = (9.0\,V - 2.2\,V) / 4700$

$I = (9.0\,V - 2.2\,V) / 4700\,ohms \quad or \quad I = 6.8V / 4700\,ohms$

$I = 0.0014\,amperes\,or\,1.4\,mA$

Experiment No. 2 – Flash LEDs with the 555 Timer

Abstract

In this experiment you will learn how to use an NE555 integrated circuit (IC) to flash an LED, and how to control the flashing frequency, or rate. You also will use a flip-flop IC to create a square wave and 4-bit binary counter to accumulate a total number of digital pulses produced by the 555-timer circuit.

Keywords

LED, flash, NE555, 555, timer, blink, counter, 7476, flip-flop, 74LS90, 74LS93, RC circuit, capacitor charging, capacitor discharging

Requirements

(1) - NE555 timer integrated circuit, 8-pin DIP integrated circuit
(1) - SN7476 (7476) flip-flop, 16-pin DIP integrated circuit
(1) - SN74LS93 (74LS93) binary counter, 14-pin DIP integrated circuit
(1) - SN74LS04 (74LS04) hex inverter, 14-pin DIP integrated circuit (optional)
(2) - LEDs, red
(2) - LEDs, green
(4) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(2) - 1000-ohm, 1/4-watt resistors, 5% (brown-black-red)
(1) - 36k-ohm, 1/4-watt resistors, 5% (orange-blue-orange)
(1) - 56k-ohm, 1/4-watt resistors, 5% (green-blue-orange)
(1) - 72k-ohm, 1/4-watt resistors, 5% (violet-red-orange)
(1) - 100k-ohm, 1/4-watt resistors, 5% (brown-black-yellow)
(1) - 0.1 µF disc-ceramic capacitor (50 volts or greater)
(1) - 10 µF electrolytic capacitor (16 volts or greater), radial leads
(1) - Voltmeter or digital multimeter with a voltage scale
(1) - Solderless breadboard
(1) - Variable DC power supply (or 9- and 5-volt power sources)
Hookup wires or jumpers

Introduction

Although you can turn one or more LEDs on or off with a simple switch, many devices need an LED that will flash or blink. Some projects might need several LEDs that flash at different rates. A small, inexpensive NE555-timer integrated circuit (IC) will do the job. Many companies manufacture the NE555-type timer ICs with part numbers such as NE555, SA555, SE555, TLC555, LM555, and so on. For simplicity I'll call it the 555 timer. This device comes in 8-pin plastic packages of various sizes and with several types of electrical leads. Experiments will use an 8-pin dual in-line package (DIP) that easily drops into a solderless breadboard or IC socket. According to industry sources, the semiconductor industry has sold more of the 555-timer ICs than any other IC.

This timer IC relies on a external capacitor and two resistors in series, which forms an resistor-capacitor (RC) network, or circuit, as shown in **Figure 2.1**. When someone closes the switch to connect the +9V power supply to the RC circuit, current flows through resistors Ra and Rb to charge the capacitor, C. At the instant you supply power to this circuit, the capacitor acts like a short circuit to ground. As a result, the voltage across the capacitor measures 0 volts to start. (Assume this capacitor does not already hold any charge.)

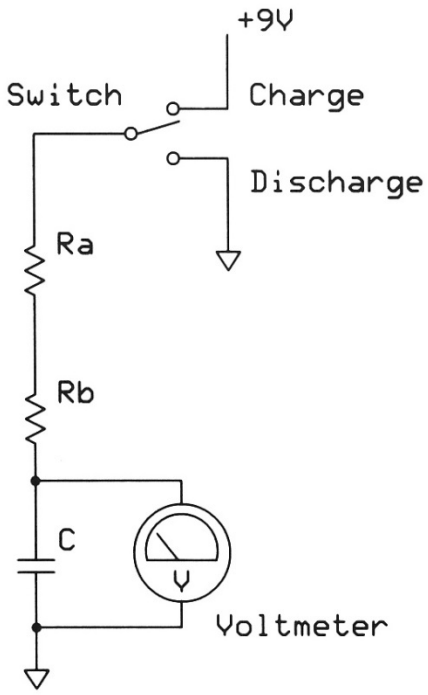


Figure 2.1.
A simple RC series circuit must include a resistance, here Ra + Rb, and a capacitance, C. You could use a single resistor in place of Ra and Rb. As you will learn, a 555 timer IC, though, requires two separate resistances.

As current flows through Ra and Rb, the capacitor accumulates charge. After a short period the capacitor has a "full" charge and will not accept any more current. At this time, the voltage across the capacitor has reached the supply voltage, +9V. As the current changes the capacitor, the voltage across the capacitor follows a characteristic RC curve, as shown in **Figure 2.2** for a 10 microfarad (μF) capacitor and a resistance of 680 ohms. By flipping the switch so it connects the capacitor to ground through Ra and Rb, the voltage across the capacitor produces a discharge curve as shown in **Figure 2.3**. A 555-timer IC takes advantage of capacitor charge and discharge cycles to create a timing signal that can flash an LED.

Figure 2.2.
This graph of time vs. voltage shows how the voltage across a capacitor increases as it charges in series with a resistor. All RC circuits exhibit the same characteristics. In this example, capacitor charging took about 25 milliseconds (msec).

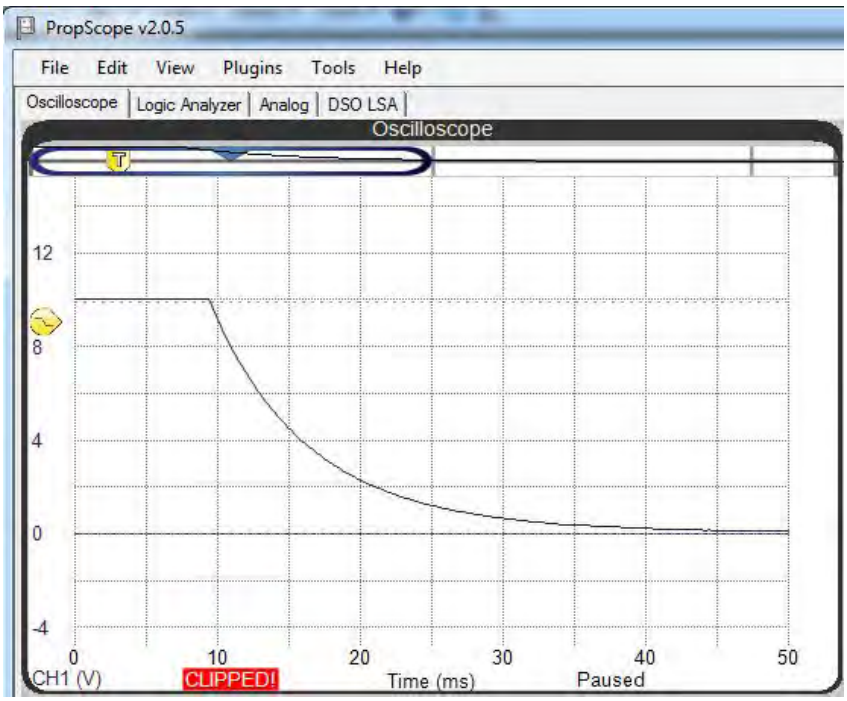


Figure 2.3.
Discharging a capacitor through a resistor produces a similar, though inverted, plot that shows how the voltage across the capacitor decreases as time goes on.

The circuit diagram in **Figure 2.4** shows the connection of an RC circuit (Ra, Rb, and C) to a 555 timer IC, which you'll learn more about shortly. When you apply power to this circuit, capacitor C starts to charge through Ra and Rb. The pin-2 and pin-6 inputs on the 555 timer monitor the voltage at the capacitor. When the

555 timer detects that this voltage has *increased* to about 2/3 of the supply voltage (+9V), it switches pin 7 to ground and discharges the capacitor through only Rb. Now the pin-2 and pin-6 inputs wait for the voltage across the capacitor to *decrease* to about 1/3 of the supply voltage. When the capacitor's voltage reaches this point, the 555 timer disconnects pin 7 from ground and the charging cycle starts again. And the cycles repeat continuously.

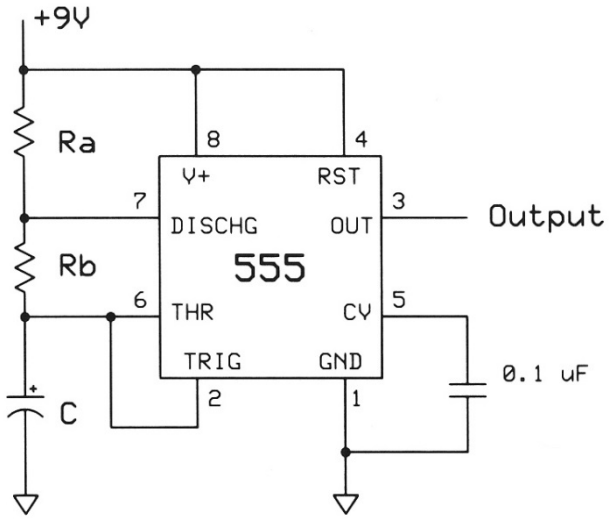


Figure 2.4.
The period or frequency of a signal created by a 555 timer IC depends solely on the values of Ra, Rb, and C.

Whenever the 555 timer reaches the 2/3-charged or 1/3-discharged state for the capacitor, it changes, or toggles, the state of the output at pin 3. This pin provides either a connection to your supply power (here +9V) or to ground. It has no "in-between" voltage. An LED connected to this point will turn on and off with the same frequency as the charge-discharge cycles of the capacitor. The diagram in **Figure 2.5** shows the charge-discharge voltages at the capacitor and the state of an LED connected between the supply voltage and pin 3, with a current-limiting resistor in series.

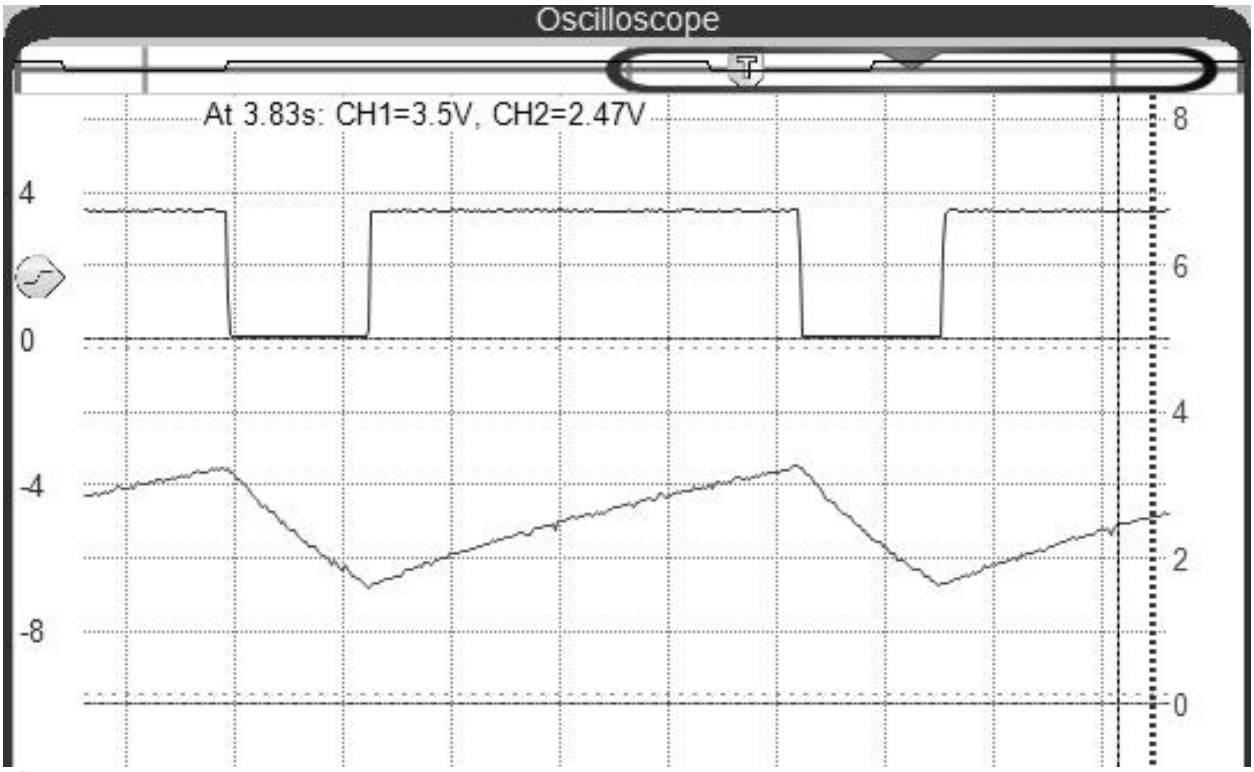


Figure 2.5.
This oscilloscope image shows the timing relationship between the voltage across the capacitor C (bottom trace), and the output at pin 3 (upper trace). A logic-0 at the output could turn on an LED. A logic-0 output connects the LED to ground.

Do you know why the capacitor-charge voltage trace does not look like the charge and discharge cycles shown in **Figures 2.2** and **2.3**? The voltage across the capacitor capacitor always remains between 1/3 and 2/3 the supply voltage. So, the capacitor does not fully charge or discharge. As a result, the trace looks like only a portion of the traces in **Figures 2.2** and **2.3**.

By changing the values of C, Ra, and Rb, you can cause an LED to flash at a specific rate and you can control the duty cycle, or the time the LED remains off and the time during which it turns on.

Step 1.

In this step you will create a 555-timer circuit to flash an LED at about once per second. Use the schematic diagram shown in **Figure 2.6** and connect the components to create your circuit. When experimenting with ICs, I recommend placing them in a solderless breadboard with pin 1 to your left. When you follow this "standard" orientation for ICs, you can easily find a pin by its number. Pin numbers increase counterclockwise from pin 1 around the IC edges when you view an IC from the top (pins down).

In this circuit capacitor C has a polarity, so ensure you have the capacitor's positive (+) wire connected to pins 2 and 6, and its negative (-) wire connected to ground. Most polarized capacitors have markings on the case to show either the + or - wire. The 0.1 µF capacitor between pin 5 and ground does not affect timing. It simply helps stabilize a voltage within the timer chip. This capacitor has no polarity, so you do not have a + and a - lead for it.

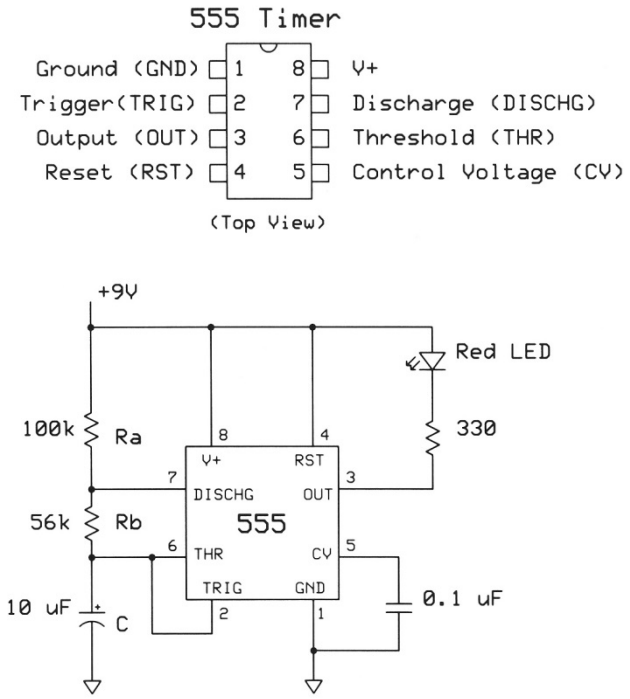


Figure 2.6.
This diagram shows a 555-timer circuit that will create an LED flash about once per second. The top view of the 555-timer IC shows pin locations and electrical functions. Diagrams such as this show pin numbers arranged according to circuit function rather than in numerical order as you see them around a chip package. The latter approach could lead to confusing diagrams.

After you make the connections, ensure your power supply is off. Then connect your power-supply ground and +9-volt outputs to your circuit and turn on power. Your LED should flash. How many times does it flash in one minute? My LED flashed 41 times. Your rate might vary somewhat from this value.

If your LED does not flash, ensure you have it properly connected with its anode (+) wire to +9V and the cathode (-) connected to the 330-ohm resistor. Check that you have made all the 555-timer connections and have nothing else connected to your circuit. Recheck the polarity of the 10 µF capacitor and ensure it matches that shown in **Figure 2.6**.

Step 2.

If you need to change the frequency at which the LED flashes, you must change the value of the capacitor, the resistors, or both. To increase the frequency, use smaller values for Ra or C. A smaller-value capacitor requires less time to charge. A smaller-value resistor lets more current pass, so the capacitor charges faster. To obtain a slower flash rate, you increase the values for Ra or C.

To achieve a specific flash rate you do not usually choose resistor and capacitor values at random until you find those that work. Instead, you use the two equations shown next to calculate the values for t1 (charge time) and t2 (discharge time) for a given a set of component values. Or, you rearrange the equations to determine the component values needed for t1 (logic-1) and t2 (logic-0) periods:

$$t1 = 0.693 * (Ra + Rb) * C$$

$$t2 = 0.693 * Rb * C$$

A quick look at these two equations shows that t1 will always exceed the time for t2 because the first equation adds resistances Ra and Rb. The equation for t2 uses only resistance Rb. Suppose you want to have a 1-second period for a 555-timer circuit. That means the LED's on time *plus* its off time will equal one second, and t1 + t2 = 1 sec. Because t2 represents the smaller time, choose components for it first.

In this example, for a 1-second period, I will start with a 0.25-second time for t2 and a 10 µF capacitor for C. The t2 time is somewhat arbitrary. I could have chosen 0.33, 0.19, 0.29 or some other fraction of a second.

Keep in mind these equations work only with units of ohms and farads, so 10 µF becomes $10*10^{-6}$ farads, or 0.000010 farads (F). Now calculate a value for the Rb resistance:

$$t2 = 0.693 * C * Rb$$

$$0.25\,sec = (0.693) * (0.000010\,F) * Rb, \quad \text{or} \quad (0.25\,sec) / (0.693 * 0.000010\,F) = Rb$$

$$Rb = 0.25\,sec / (0.00000693\,F) = 36075\,ohms,$$

or rounded to 36,000 ohms, or 36k ohms.

For a 1-second total period and t2 equal to 0.25 seconds, t1 must equal 0.75 seconds:

$$t1 = 0.75\,sec = 0.693 * C * (Ra + Rb),$$

so rearrange this equation to:

$$[0.75\,sec / (0.693 * C)] - Rb = Ra, \text{ or } [0.75\,sec / (0.693 * 0.000010\,F)] - 36000\,ohms = Ra$$

$$Ra = 108{,}225\,ohms - 36{,}000\,ohms = 72{,}225\,ohms,$$

rounded to 72,000,or 72k ohms.

You can buy standard 72k ohm and 36k ohm resistors.

Now use the following equation and your component values to calculate the overall period, t1 + t2. This equation gives you an extra check of the period t1 + t2. Again, use units of ohms and farads to find a period in seconds.

$$Period = 0.693 * (Ra + 2Rb) * C$$

What period did you calculate? Your total period should amount to the period given by t1 plus t2, or about one second.

OPTIONAL: Why does the unit of farads (F) multiplied by the unit of ohms (Ω) equal seconds? (If you're not interested, please skip to Step 3.) Many measurements rely on international standards that provide references for other measurements. The US National Institute of Standards and Technology (NIST) maintains a standard meter and kilogram. A special atomic clock defines a second in time. Units such as watts, volts, ohms, and farads all have equivalents made up of solely meters, kilograms, and seconds units (MKS). When scientists and engineers break down the unit of farad into MKS standard units they have:

$$f = m^{-2}kg^{-1}s^4\,A^2 \;\; or \;\; s^4\,A^2 / m^2kg$$

where m = meters, kg = kilograms, s = second, and A = amperes. (The ampere, A, is also a standard unit.)

For resistance:

$\Omega = m^2 kg\, s^{-3} A^{-2}$ or $m^2 kg / s^3 A^2$

Now when you multiply these MKS equivalents you have

$s^4\, A^2 \ / m^2\, kg * m^2 kg / s^3 A^2$ which simplifies to:

$$\frac{s^4\, A^2\, m^2\, kg}{m^2\, kg\, s^3\, A^2} = \frac{s^4\, \cancel{A^2\, m^2\, kg}}{\cancel{m^2\, kg}\, s^3\, \cancel{A^2}} = \frac{s^4}{s^3} = s$$

The shaded units appear in the numerator and denominator and thus cancel each other, leaving: s^4 / s^3 or s (seconds). When in doubt about why certain units multiplied or divided by each other yield a different unit, locate the definition of these units expressed in the MKS units, and use the latter to confirm your answer. We call the farad, ohm, watt, volt, joule, and pascal derived units because they come from fundamental MKS units.

Step 3.

Change your 555-timer circuit so it uses the chosen capacitance (10 μF) and the two resistor values calculated above for Ra (72k ohms, violet-red-orange) and Rb (36k ohms, orange-blue-orange). See **Figure 2.6** shown earlier for a complete schematic circuit diagram. After you complete and recheck your circuit, turn on power. The LED should flash on and off. Have someone time thirty flashes of the LED. Did your time come close to 30 seconds? I measured 30 flashes in 29.5 seconds, which seems reasonable. Thus, in my circuit the period of each flash – that is, the LED-on plus the LED-off time – amounts to:

$29.5\ seconds\ /\ 30\ flashes\ =\ 0.98\ seconds\ per\ flash$

If you need the frequency of the LED flashes rather than the period, just take the reciprocal of the period:

$Frequency\ =\ 1\ /\ Period$

Thus a period of 0.98 seconds per flash equals 1/0.98 seconds, or 1.02 Hertz (Hz), the unit of frequency.

OPTIONAL: Electronic components have some variations due to manufacturing. The resistors have a tolerance of ±5 percent and the capacitor I used has a tolerance of ±20 percent, so your LED flash period might differ somewhat from what I measured. Use the tolerances to determine the shortest and the longest period for the component values noted above. See answers at the end of this experiment.

Step 4.

In some cases, you might need two LEDs that alternate on and off conditions, much like a railway-crossing signal. You can use a 555 timer for this type of control. The 555 timer output at pin 3 will sink current (connection to ground) and source current (connection to +V), so it can control two LEDs as shown in **Figure 2.7.**

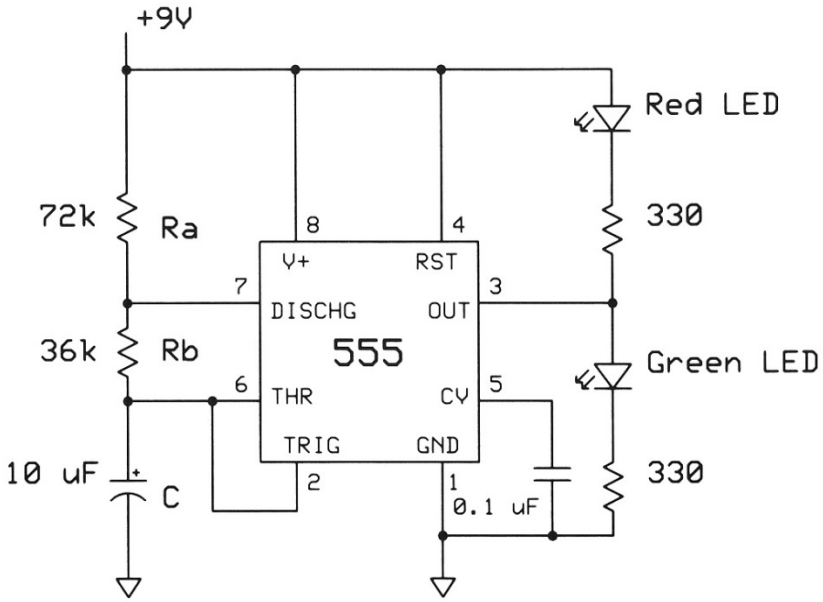


Figure 2.7.
In this circuit, the 555-timer output pin controls two LEDs. It sinks current through the red LED during t2 and sources current for the green LED during t1.

Turn off power to your circuit and add the green LED and 330-ohm resistor that connect between ground and pin 3 on the 555 timer IC. Turn on power. The LEDs should alternate on and off, back and forth. Do the LEDs each light for the same time?

Previously in Step 2 you calculated component values for a 1-second overall period, with a 0.25-second period for t2. So, during this period, the pin-3 output connects to ground and the red LED turns on. The pin-3 output connects to +V during t1 and causes the green LED to turn on. The oscilloscope display in **Figure 2.8** shows the voltages at pin 3 on my 555 timer. The period measures 1.04 seconds, with t1 = 0.77 seconds and t2 = 0.27 seconds. That gives a frequency of 1/1.06 seconds, or 0.94 Hz. Let the flashing-LED circuit continue to run.

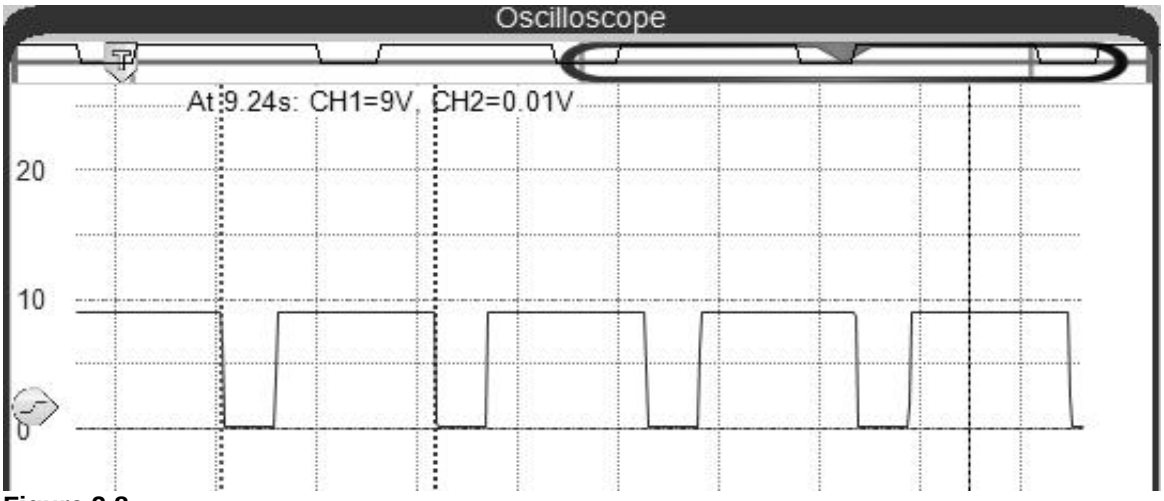


Figure 2.8.
This scope image shows the signal produced by a 555-timer circuit with component values chosen for a 1-second period. The circuit used power at 9 volts.

Step 5.
Because the timer circuit uses resistors Ra plus Rb to limit the current that charges the capacitor, C, and uses only the current limited by Rb to discharge the capacitor, the 555 timer cannot produce equal t1 and t2 periods, although it can get close. If you require a true square wave, a "flip-flop" integrated circuit connected to the 555-timer circuit will do the job. A square wave provides equal periods for the logic-1 and logic-0 outputs, and it has another capability you can use to create alternating flashing LEDs demonstrated in Step 4.

The output of a flip-flop changes state, 0-1-0-1-0, and so on, each time its clock input detects a pulse edge. The specifications for a given flip-flop IC – there are many types – indicate whether it responds to a positive-going signal edge (logic-0 to logic-1) or a negative-going (logic-1 to logic-0) edge. In the circuit that follows I use an SN7476 dual J-K flip-flop IC. Although the SN7476 J-K flip-flop dates back four decades, it remains available today, and provides a starting point for experiments.

The SN7476 J-K flip-flop IC requires a +5-volt power supply, but the 555-timer circuit uses a 9-volt power source. What would happen if you reduce the voltage for the 555-timer IC from +9V to +5V? Will this voltage change affect the LED-flashing times? If you have a variable power supply, place a voltmeter – or multimeter set for volts – across the power supply output and reduce the voltage to 5 volts.

If you have a fixed-output 5-volt power supply, disconnect your +9-volt power supply and use the 5-volt supply to power your 555-timer circuit. What did you see?

The t1, t2, and total-period times should remain the same. The introduction to this experiment explained the capacitor, C, will charge to 2/3 of the supply voltage and will then discharge to 1/3 the supply voltage. So although the capacitor will charge more slowly with a +5-volt supply than with a +9-volt supply, the voltage across the capacitor still only must reach 2/3 of the supply voltage. The scope displays shown in **Figures 2.9** and **2.10** show the signals acquired across the capacitor and from the pin-3 output for a 5-volt and a 9-volt supply respectively. In both images, the horizontal divisions represent 200 milliseconds (200 msec). See the section "Digging Deeper into the 555-Timer Circuit" later in this experiment.

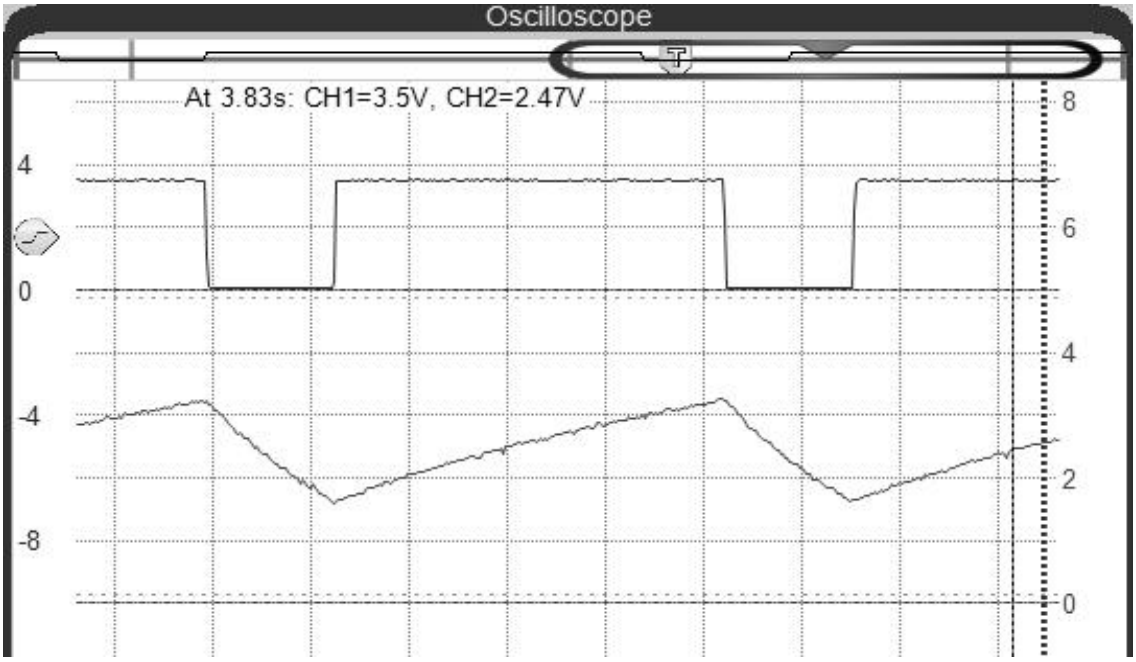


Figure 2.9.
The output signal for a 555 timer powered by 5 volts.

For the +5V supply, the maximum-voltage measurement I measured came to 3.25 volts and the minimum-voltage measurement came to 1.65 volts:

$3.25\,V\,/\,5.00\,V\;=\;0.65$ or 65% and $1.65\,V\,/\,5.00\,V\;=\;0.33$ or 33%

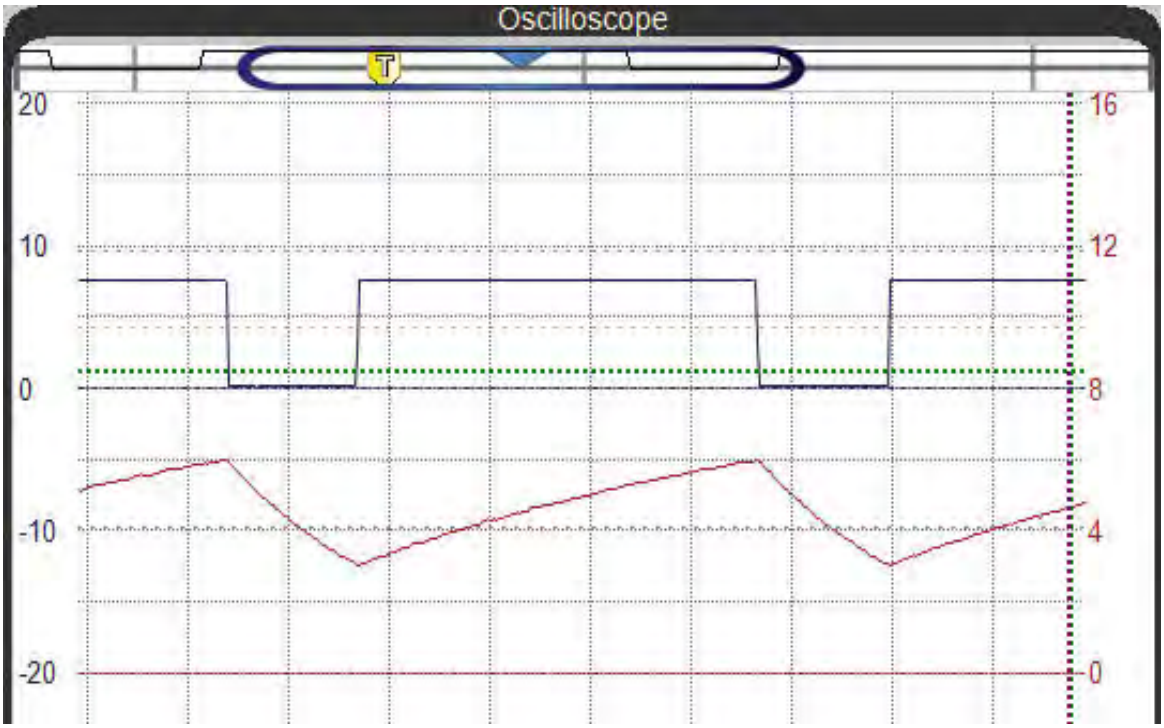


Figure 2.10.
The output signal for a 555 timer powered by 9 volts.

The timing equation used earlier:

$$Period\;=\;0.693\;*\;(Ra\;+\;2Rb)\;*\;C$$

does not include a voltage term. Regardless of the power-supply voltage, the resistances and capacitance alone determine frequency and period. Turn off the +5V power to your circuit.

CAUTION: An NE555-type timer can operate with a power-supply voltage between 4.5 and 16 volts, according to the Texas Instruments data sheet for the "NA555, NE555, SA555, and SE555" family of timers. These devices have a frequency limit of about 500 kHz. Texas Instruments also manufactures a 555-timer chip, TLC551 that operates with a recommended supply voltage between 1 and 15 volts. The TLC551 uses a complementary metal-oxide semiconductor (CMOS) process to create a low-power 555-timer equivalent. This timer has a maximum frequency of 1.8 MHz. Data sheets available on the Internet provide more information about how to use 555 timer ICs.

Step 6.

Remove the red and green LEDs and their associated 330-ohm resistors and set them aside. You will use these components later in this step. Insert an SN7476 IC in your breadboard and wire it as shown in **Figure 2.11**. (Note that a black dot represents a wire or component connection. When wires pass over or under each other, no dot appears in the circuit diagram.) The 1-CLK input to the flip-flop connects to the pin-3 output of the 555 timer IC. (Note: Each 7476 flip-flop IC comprises two independent J-K flop-flop circuits. Signals use a prefix numeral 1 or 2 to indicate which flip-flop they "belong" to.)

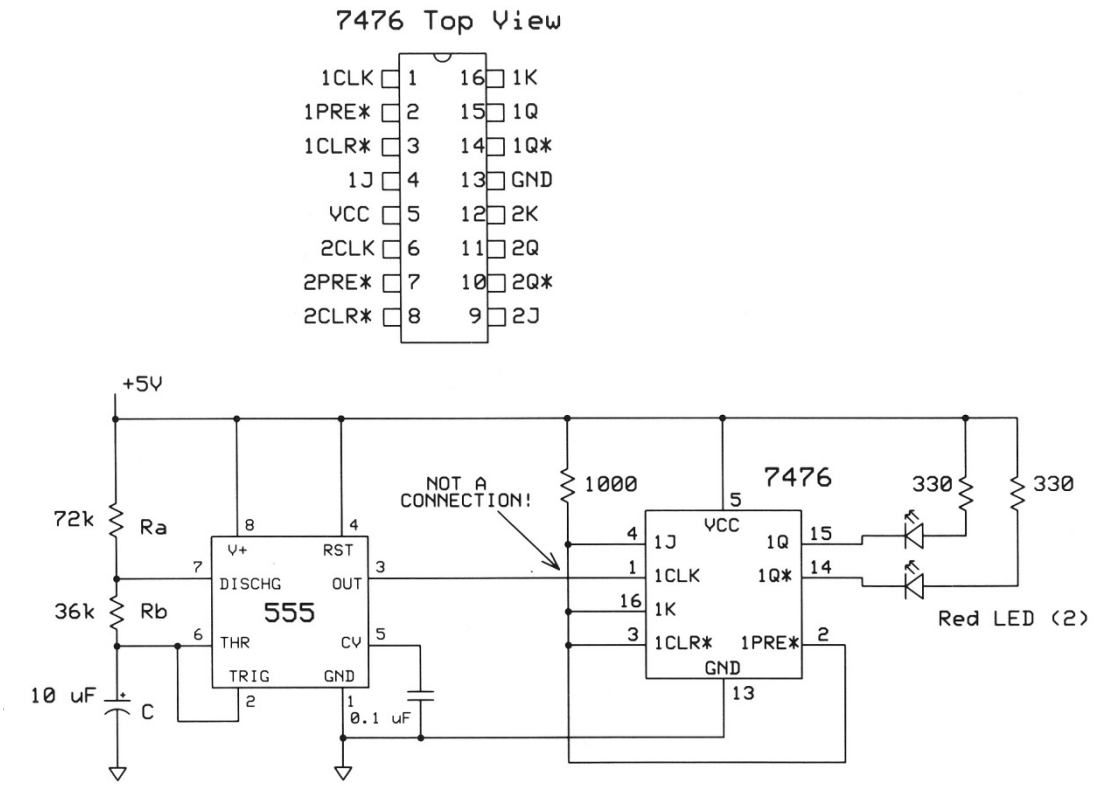


Figure 2.11.
A 555 timer connects to the number-1 J-K flip-flop in a 7476 IC to create a square wave at the 1Q and 1Q* outputs. The 1Q* output produces a logic 0 when the 1Q output produces a logic 1, and *vice versa*.

The flip-flop connections to the 1000-ohm resistor ensure a logic-1 state at these unused inputs. The voltage waveforms in **Figure 2.12** show the relationship between the output signal (pin 3) from the 555 timer in the upper trace and the 1Q output (pin 15) from the 7476 flip-flop in the bottom trace. What conclusion can you draw from this information?

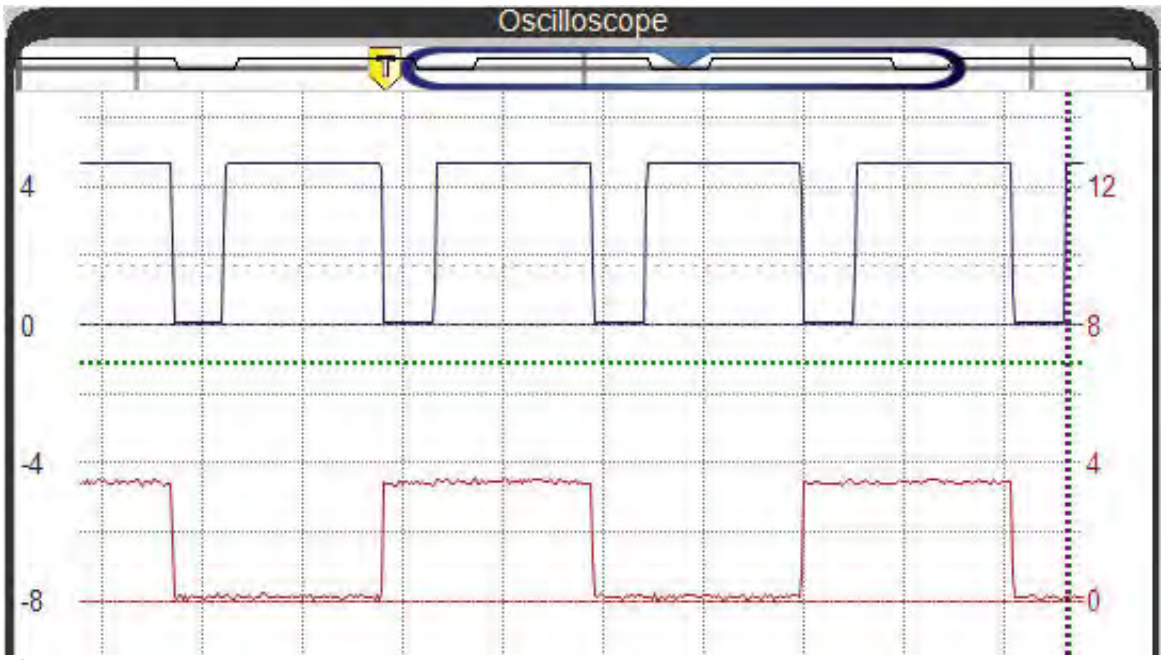


Figure 2.12.
This scope image shows the timing relationship between the 555-timer output (upper trace) and the 7476 flip-flop 1Q output (lower trace). The flip-flop "divides" the 555-timer output signal's frequency by two.

The output from the flip-flop changes state only on the negative-going edge (logic-1 to logic-0) of the 555-timer output. Note also that the output from the flip-flop only changes its output state once for each logic-1-to-logic-0 transition and logic-0-to-logic-1 change at the timer output. So, for every four logic-0 signals from the timer, the flip-flop produces two logic-0 pulses. Thus the flip-flop divided by two the output frequency from the 555 timer. As a result, each red LED will remain on (or off) for a complete *period* of the 555 timer. The flip-flop output also has an equal on and off period, and a total period of 2.1 seconds.

The 7476 IC provides a second J-K flip-flop circuit that will divide by a factor of two the output frequency of the flip-flop you now have wired. If effect, you "cascade" the pulse through another divide-by-two flip-flop circuit. Turn off power to your circuit and *add* the connection, resistors, and two green LEDs shown in **Figure 2.13** for the second flip-flop in the 7476 package. Leave all other connections in place. Now the two red LEDs will flash and the green LEDs will flash half as fast – and thus half as many times – as the red LEDs.

In this circuit, each LED had its own current-limiting resistor. Could you use only one current-limiting resistor for the two red LEDs and one for the two green LEDs? Find the answer at the end of this experiment.

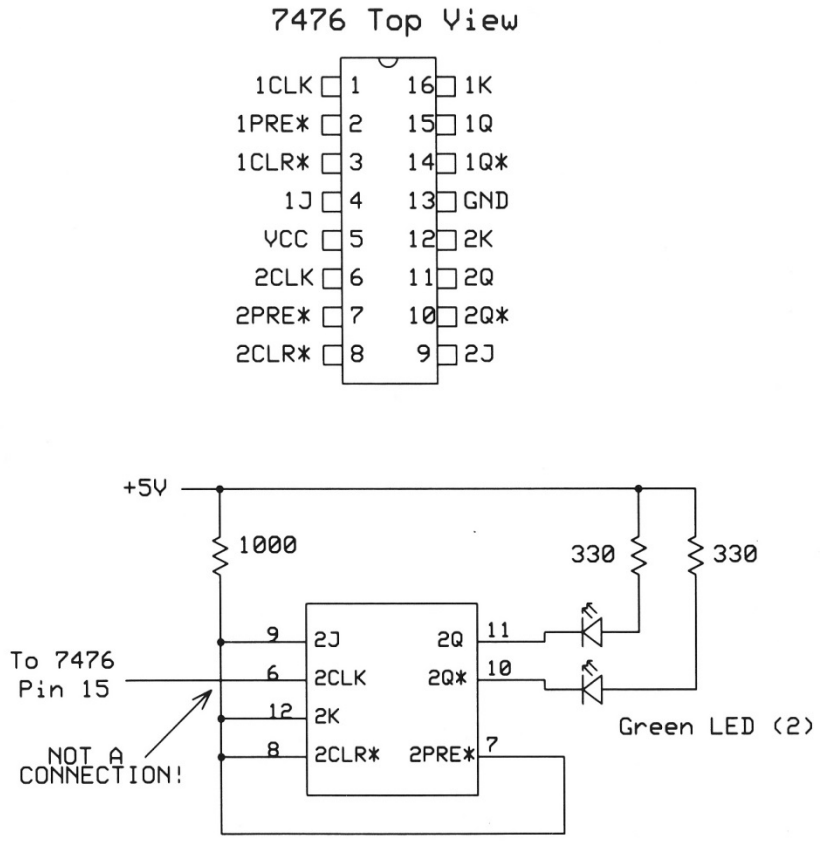


Figure 2.13.
Components added to your flip-flop circuit. The 7476 flip-flop exists within the same package as the flip-flop shown in **Figure 2.11**. DO NOT add a second 7476 16-pin IC.

Turn off power to your circuit and remove only the two LEDs that connect to the 1Q (pin 15) and 2Q (pin 11) flip-flop outputs. This change leaves an LED connected to the 1Q* (pin 14) and 2Q* (pin 10) outputs. Turn on power to your circuit. Congratulations, you have created a 2-bit binary counter. Keep in mind the flip-flop outputs sink current through the LED, so an LED-on condition indicates a logic-0 at the associated flip-flop pin. In this step, the LED-on indicates a logic-0 at the Q* output, and thus a logic-1 at the Q output.

Step 7.

You plugged a lot of wires into your breadboard to create a 2-bit binary counter. Thankfully, IC manufacturers provide complete counters in small packages, for example the SN74LS90 4-bit decimal counter and the SN74LS93 4-bit binary counter. The LS in the part number indicates ICs that incorporate low-power Schottky semiconductor technology. These devices provide high-speed operations at low power, not something you really worry about in most LED circuits.

Turn off power to your circuit and remove only the 7476 flip-flop IC and its associated components, LEDs, and wires. Leave your 555-timer IC wired for use in this step. Insert a 74LS93 counter IC in your breadboard and connect it as shown in **Figure 2.14**. Use red and green LEDs as you wish. The four LEDs will show the output of the 4-bit binary counter. (I recommend you place the LED for the QD output to your left on the breadboard and add the LEDs for QC, QB, and QA in that order to the right of the QD LED.) The LS-type devices cannot source much current – only about 1 mA – to drive LEDs, but they can sink about 16 mA, which can light an LED connected to +5V through a current-limiting resistor. Turn on your circuit and watch the LEDs. What do you observe?

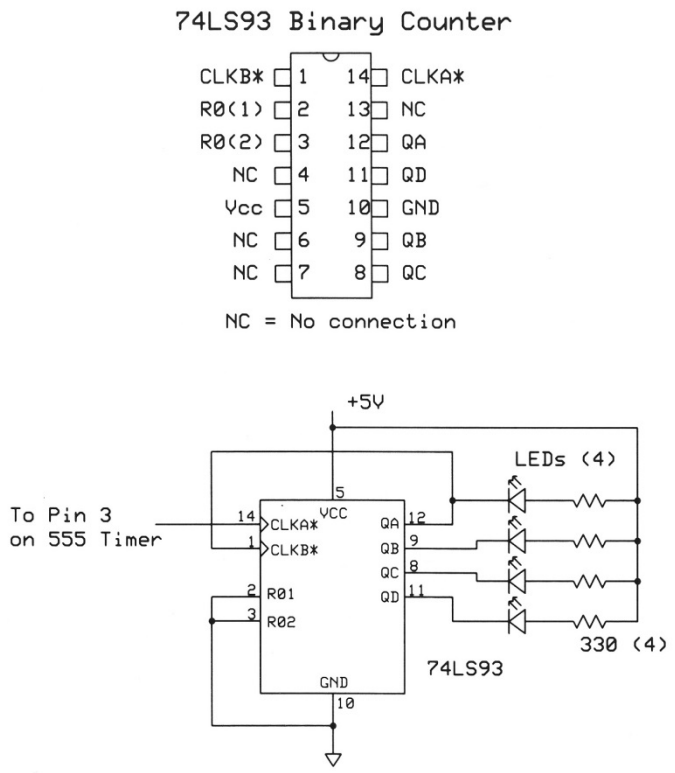


Figure 2.14.
The schematic diagram for a 74LS93 binary counter connected to your 555-timer output creates a 4-bit binary counter. The LED-on condition indicates a logic-0 at the corresponding output.

The 74LS93 operates as a 4-bit binary "up" counter that starts at 0000 and continues to 0001, 0010, 0011, and so on up to 1111. See **Table 2.1.**

Table 2.1. Decimal and binary values for a 4-bit counter.

Decimal Value	Binary Value	Decimal Value	Binary Value
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Note:
0 = LED off,
1 = LED on.

But in this circuit, the logic-0 outputs turn LEDs on, so you observe a count-down pattern instead, on-on-on-on, on-on-on-off, on-on-off-on, on-on-off-off, and so on to off-off-off-off. If you want to see the up-count pattern of LEDs, you could use an inverter IC, such as a 74LS04, and connect one inverter to each counter output and then use the inverter's outputs to drive LEDs, as shown in **Figure 2.15**. Each 14-pin 74LS04 package provides six independent inverters.

The 74LS93 has two reset inputs, R0(1) and R0(2). If either of these inputs goes to a logic 0, the counter operates properly and counts. If both of these inputs go to a logic-1, the counter resets to 0000. For your circuit, a reset turns all four LEDs on. The 74LS93 counter IC provides separate divide-by-2 and divide-by-8 circuits. The divide-by-2 circuit receives pulses from the 555 timer at input CLKA*. The output QA (pin 12) from the divide-by-2 circuit goes through an external wire and serves as the input for the divide-by-8 counter input CLKB* (pin 1). When you make this connection, you have a complete divide-by-16 circuit that forms a 4-bit counter: $2^4 = 16$.

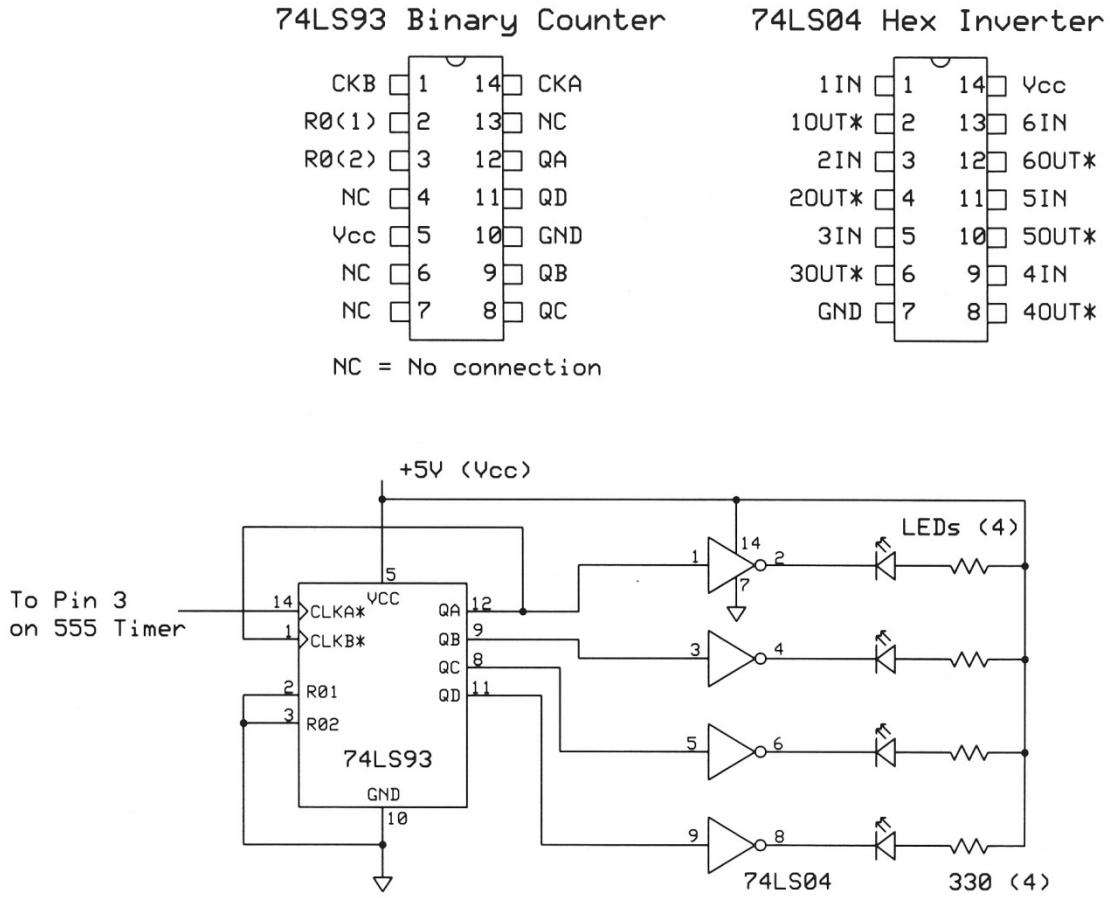


Figure 2.15.
Adding an inverter to each output from a 74LS93 counter creates a 4-bit binary "up" counter where LED-on equals a logic-1 at the corresponding counter output.

Can you figure out why the engineers who created the 74LS93 counter set it so the count increments by one on the negative-going edge of the clock signal at the CLKA* input? Hint: What happens to the counter's QD output when the count "rolls over" from 1111_2 to 0000_2? See answers at the end of this experiment.

Keep your 555-timer circuit wired in your solderless breadboard. You will use it in other experiments.

Digging Deeper into the 555 Timer Circuit

The diagram in **Figure 2.16** shows the primary components of the timing circuitry for the 555 timer chip. This diagram includes the external Ra, Rb, and C components described earlier in this experiment.

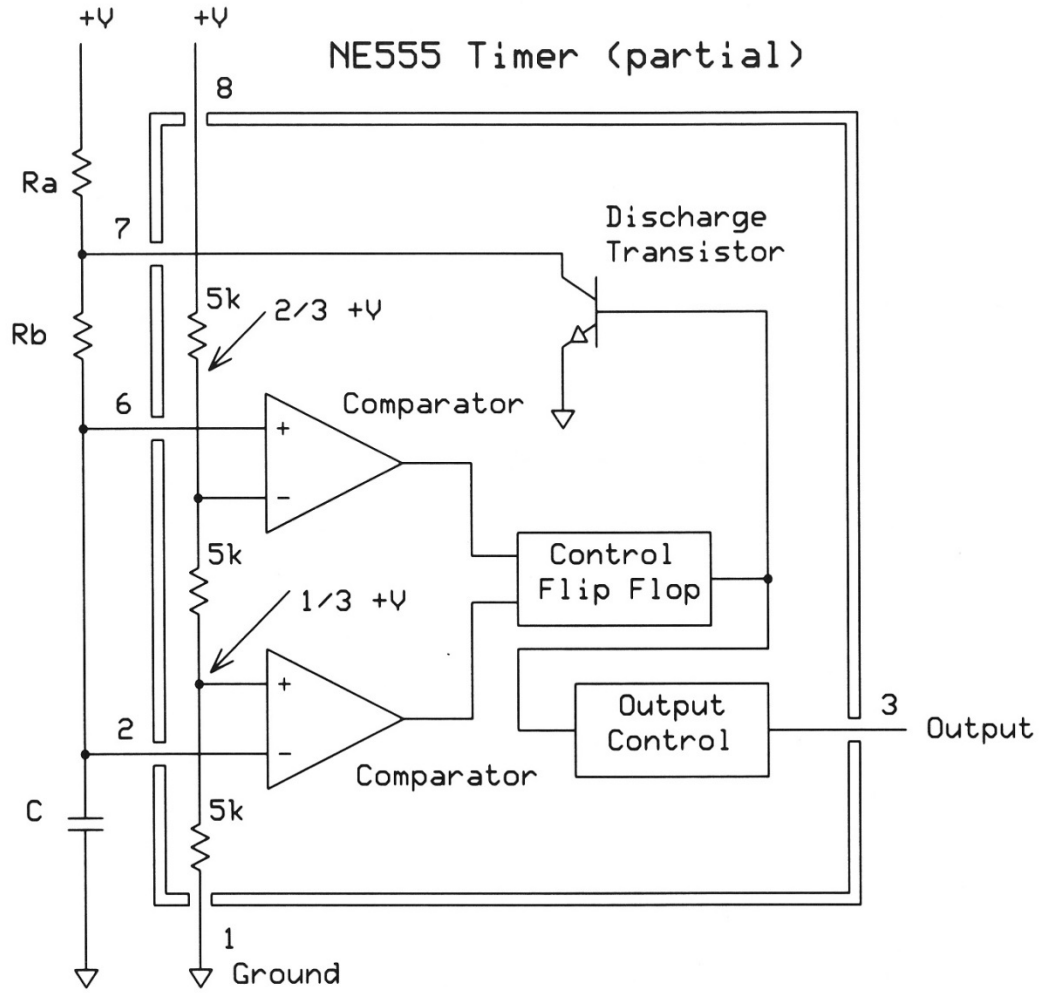


Figure 2.16.
This diagram shows the timing and output portions of a 555-timer integrated circuit.

You can see the three internal 5k-ohm resistors that form a voltage divider and the points at which the two comparators "tap off" 1/3 +V and 2/3 +V. Some circuits label the +V power input as Vcc. The upper comparator detects when the capacitor has charged to 2/3 of +V and then sets the Control Flip-Flop to activate the Discharge Transistor. This NPN transistor discharges the capacitor through resistor Rb. The voltage across the capacitor decreases until it reaches 1/3 +V. At that point, the bottom comparator resets the Control Flip-Flop, which then turns off the Discharge Transistor. With the transistor turned off, the capacitor will start to charge again and the cycle will repeat. You will learn about transistors in a later experiment.

The components shown in **Figure 2.16** represent only part of the 555-timer circuit, but they illustrate the basic timing portion of the 555 IC. The Output Control section switches between ground and +V for the pin-3 output, depending on the state of the Control Flip-Flop.

Hans R. Camenzind (1934 – 2012) invented the 555-timer circuit in 1970 while working with Signetics, a US semiconductor company. Signetics introduced the NE555 chip in 1972. According to some people, the 555 has proven more popular than any other integrated circuit.

References

Berlin, Howard M, "The 555 Timer Applications Sourcebook with Experiments," Howard W. Sams & Co., Indianapolis, IN. 1979. ISBN: 0-672-21538-1. A new edition appeared in 2008.

Answers

Experiment 2, Step 3:

For the fastest period you would need the smallest resistance values and the smallest capacitor so the capacitor would charge and discharge rapidly. Decrease the resistor and capacitor values accordingly:

$110\ kohms\ *\ 0.95\ =\ 104\ kohms$
$36\ kohms\ *\ 0.95\ =\ 34.2\ kohms$
$10 * 10-6\ farads\ *\ 0.80\ =\ 8.0 * 10-6\ farads$

Now calculate period from: $Period\ =\ 0.693\ *\ (Ra\ +\ 2Rb)\ *\ C$

$Period\ =\ 0.693\ *\ (104{,}000\ ohms\ +\ 2\ *\ 34{,}200\ ohms)\ *\ 8.0\ *\ 10-6\ farads$

$Period\ =\ 0.96\ seconds$

For the longest period, use the largest resistance and capacitance values:

$110\ kohms\ *\ 1.05\ =\ 116\ kohms$
$36\ kohms\ *\ 1.05\ =\ 37.8\ kohms$
$10 * 10-6\ farads\ *\ 1.20\ =\ 12.0 * 10-6\ farads$

$Period\ =\ 0.693\ *\ (116{,}000\ ohms\ +\ 2\ *\ 37{,}800\ ohms)\ *\ 12\ *\ 10-6\ farads$

$Period\ =\ 1.60\ seconds$

Engineers and circuit designers routinely do this type of "worst case" calculations to determine how a circuit might behave.

Experiment 2, Step 6:

Yes, you could use one current-limiting resistor for both red LEDs and a second resistor for both green LEDs. By definition, a flip-flop circuit can have only one output, Q or Q* at a logic 0 at any time, so only one LED will light and draw current. Try this with your flip-flop circuit.

Experiment 2, Step 7:

The 74LS93 counter increments the counter output by one on the negative-going (logic-1-to-logic-0) edge of the CLKA* input. When the count reaches 1111_2 and "rolls over" to 0000_2, the QD output provides a negative-going (logic-1-to-logic-0) edge. You could use that edge from the QD output to increment another 74LS93 counter IC to give you an 8-bit counter. Then, when the first 74LS93 counter goes from 1111_2 to 0000_2, the cascaded 74LS93 count would increment to 0000_2 to 0001_2. The circuit in **Figure 2.17** shows the connections between two 74LS93 ICs to create an 8-bit counter. You could add additional counters if you choose.

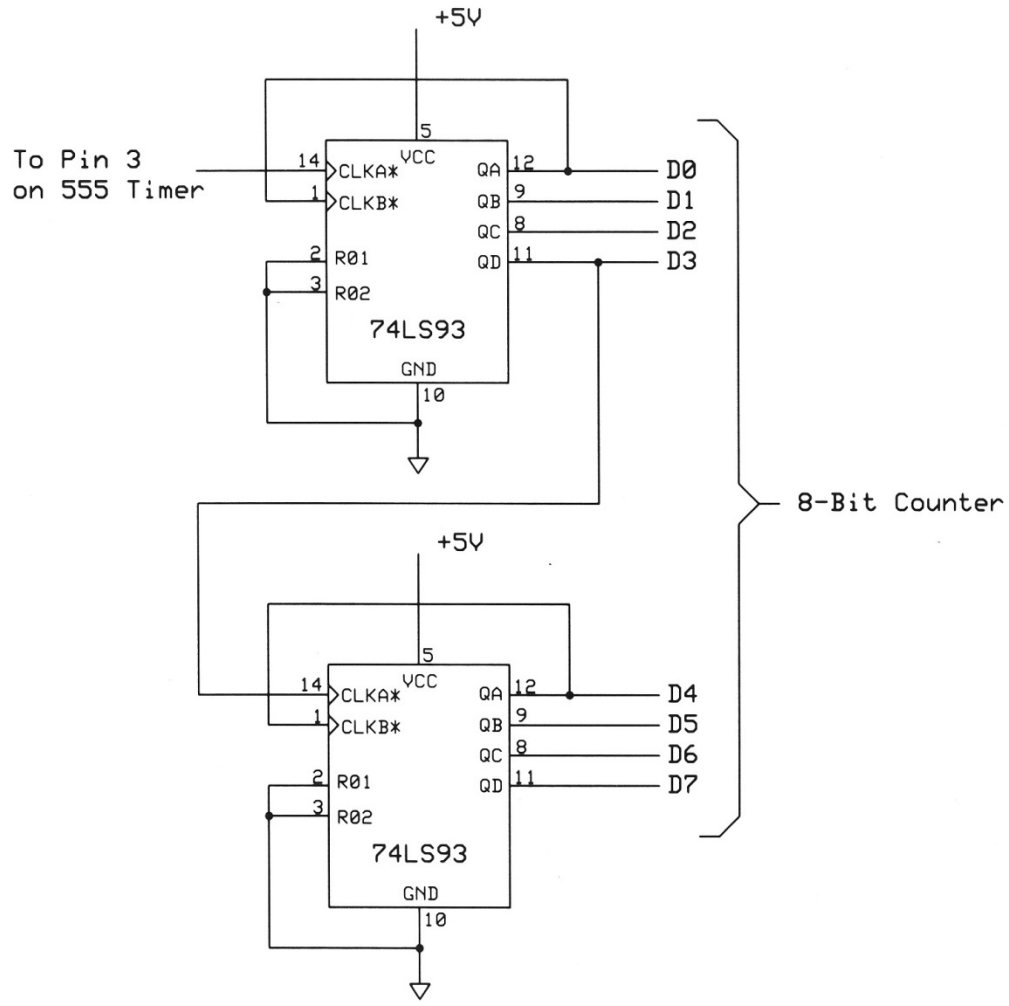


Figure 2.17.
In this circuit, the QD output from the the upper counter IC creates a clock edge (logic-1-to-logic-0) that increments the count in the lower counter IC. This circuit represents an 8-bit binary counter.

This type of counter cascading could use decade (74LS90 divide-by-10) or decade (74LS92 divide-by-12) counter ICs, too. The 74LS92 ICs also go by the name, divide-by-12 "decade" counters.

Experiment No. 3 – Counters and Numeric Displays

Abstract

In this experiment you will learn how to use a binary-coded-decimal-to-7-segment decoder integrated circuit (IC) to display numerals. A decimal and a binary counter will create those values. You also will learn about the reset capabilities of standard counter ICs, and how to "blank" leading zeros in a multi-digit display.

Keywords

LED, 7-segment display, counter, 74LS90, 74LS93, 7447, 74LS47, leading zeros, blanking, reset, decoder

Requirements

(1) - NE555 (555) timer, 8-pin DIP IC
(1) - SN74LS90 (74LS90) decimal counter, 14-pin DIP IC
(1) - SN74LS93 (74LS93) 4-bit binary counter, 14-pin DIP IC
(1) - SN7447 (7447) 7-segment decoder/driver, 16-pin DIP IC
(1) - 7-segment LED display, common anode (CA), through-hole pins
(1) - LED, any color
(10) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(1) - 36k-ohm, 1/4-watt resistors, 5% (orange-blue-orange)
(1) - 72k-ohm, 1/4-watt resistors, 5% (violet-red-orange)
(1) - 0.1 µF disc-ceramic capacitor (50 volts or greater)
(1) - 10 µF electrolytic capacitor (16 volts or greater)
(1) - Solderless breadboards
(1) - 5-Volt DC power supply
Hookup wires or jumpers

Introduction

Computers and logic circuits use binary (on or off) signals to convey information as individual bits. When circuits group eight bits, for example, they form a byte, such as 10011110_2. This binary value – indicated by the subscript 2 for base-2 values – represents the decimal value of 158_{10}. Here the subscript 10 indicates our everyday decimal numbering.

No one wants to see a phone number or email address expressed as binary numbers. Humans like digits 0 through 9 on clocks, microwave ovens, and smart phones. But digital computers understand only binary information. In Experiment 2, you learned how to use a 74LS93 IC as a 4-bit binary counter and you saw a circuit for an 8-bit binary counter. Now you will learn how to use a decoder/driver IC to convert 4-bit values into numerals and symbols shown on a 7-segment LED display.

Years ago, semiconductor manufacturers anticipated the need to convert binary information into decimal digits people could read, and they created decoder-driver ICs to do the job. A decoder-driver IC such as the 7447 or 74LS47 IC accepts binary-coded decimal information; that is, 0000_2 through 1001_2, which represent the values 0 through 9. The decoder-driver IC translate a 4-bit binary value into seven discrete signals that directly control the seven segments in a display. The 16-pin 7447 or 74LS47 decoder ICs have three special control inputs also worth exploring.

In this experiment, you will use one 7447 or 74LS47 decoder driver with one 7-segment display. Semiconductor manufacturers produce several varieties of digital ICs in the overarching 7400 family of

devices. The original 7400-family ICs used a lot of power, relatively speaking. The 74LS00-family ICs use less power. Experiments in this book use 7400-, 74LS00-, and 74HC00-family devices. For more information about these logic families, please visit: http://en.wikipedia.org/wiki/7400_series.

Step 1.

If you have not completed Experiment 2, please turn to it and perform the steps so you understand how the 555-timer IC and the 74LS93 counter IC work together to create a 4-bit binary counter. If you have disassembled your circuits from Experiment 2, assemble the circuit shown in **Figure 3.1** for this experiment. Turn on power to the circuit and use the free end of the TEST LED probe wire to test – one at a time – each output on the 74LS93 counter: QA (pin 12), QB (pin 9), QC (pin 8), and QD (pin 11). At each pin, the TEST LED should flash on and off, although it will take about 16 seconds for the QD output to go through a complete on-off-on cycle. The QA output changes state twice as fast as the QB output, the QB output changes twice as fast as the QC output, and so on.

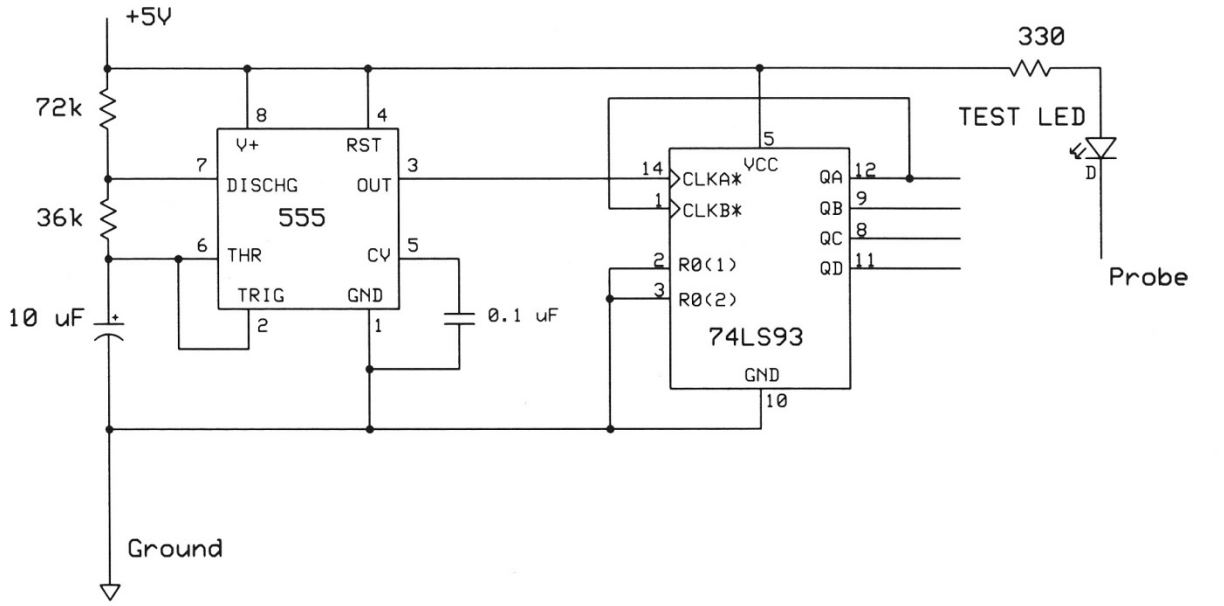


Figure 3.1.
This simple circuit provides a 555 timer IC as a pulse generator and a 74LS93 4-bit binary counter that produces a binary count from 0000_2 to 1111_2. The 555 timer IC creates a pulse about once per second (1 Hz) so you can easily observe counter and display operations.

Seven-segment and other types of displays that use several LEDs come in two versions, common-anode (CA) or common-cathode (CC). When you look at data sheets for displays, pay careful attention to the type of display you need. **Figure 3.2** shows the connections for the LEDs in each type of 7-segment display. If you want to control a common-anode display, for example, supply power to the anode terminal and sink current for a segment LED to ground through a current-limiting resistor. That segment will light. In most cases you will find it easier to use common-anode displays instead of common-cathode displays. A common-cathode display connects all of the segment LEDs to ground and you provide current from a positive-voltage power supply through a resistor to the segments you want to turn on.

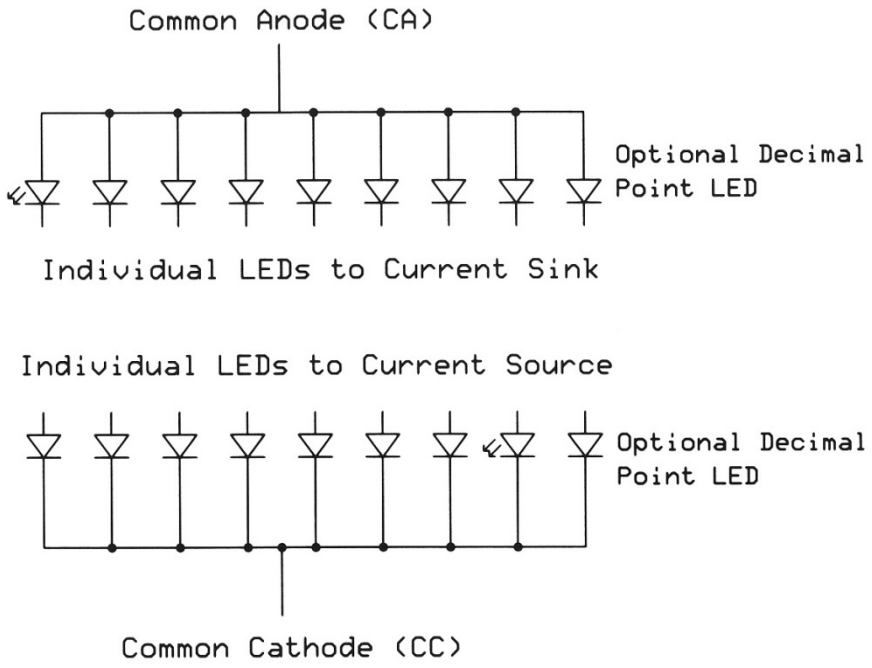


Figure 3.2.
Common-anode displays (top) supply a common connection to a positive current source. Connections through a resistor to ground turn on individual LEDs. Common-cathode displays (bottom) use a common connection to ground and require connections through a resistor to a positive current supply to turn on individual LEDs.

Step 2.

Insert a 74LS47 or 7447 decoder/driver IC in your breadboard and about 1.5 inches (4 cm) away, insert a common-anode (CA) 7-segment display in your breadboard, too. If necessary, you can use a separate breadboard for the decoder/driver IC and display. Ensure you connect +5V power and ground to both breadboards. The circuit will not work if you miss these connections. Again, orient ICs so pin 1 goes toward the bottom-left corner of your breadboard.

The diagram in **Figure 3.3** shows a front view of three types of common-anode 7-segment displays with labels for each signal. Some 7-segment displays include one or two decimal-point LEDs. Obtain the data sheet for the type of 7-segment display you have and compare the segment pin numbers, the decimal-point (DP) pin numbers (if any), and the anode (+V) input pin numbers to those shown in **Figure 3.3**. If your display's pin numbers vary, write them on **Figure 3.3** for reference or create your own diagram. The letter used for each display segment remains standard for all 7-segment displays. Thus "g" always refers to the middle segment that, when lit, would turn the numeral 0 into an 8. A 7-segment display might not have all the metal pins expected for a 14-pin IC. Most display manufacturers do not include pins that have no function, unless needed for mechanical mounting.

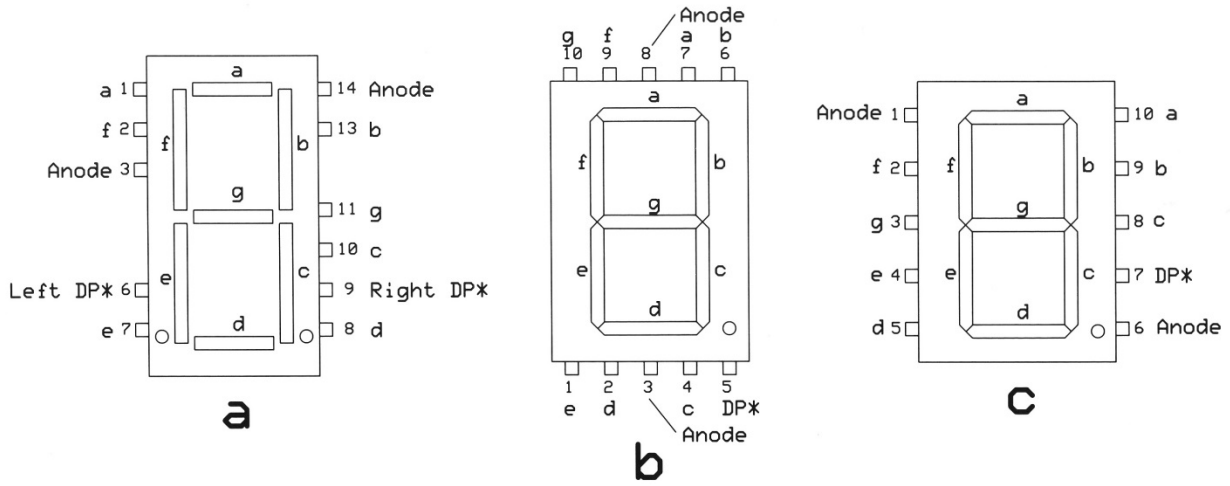


Figure 3.3.
Seven-segment displays usually follow the pin assignments shown here for segments and a decimal point or points. All three displays have a common anode. When a display has more than one anode pin, connect them all to +V. Common-cathode displays have the same segment-pin configurations, but cathode pins replace anode pins. The letters DP indicate a decimal-point LED connection, if any.

The displays I used needed about 10 mA per segment. I measured 8.8 mA per display segment with a 330-ohm resistor and used that resistance value in this experiment to limit current through the LEDs. If you don't know what resistance value to use with your display, refer to the manufacturer's data sheet (usually available online) and note the average current recommended for a segment. Experiment 1 explains how to determine a useful resistance value if you lack a display's data sheet.

Connect the common-anode pin or pins on your 7-segment display to +5V. Connect one end of a 330-ohm resistor (orange-orange-brown) to ground on the breadboard with the display on it. Connect the other end to an open column of breadboard contacts. Connect a long jumper wire to the end of the 330-ohm resistor not connected to ground. Turn on power to the breadboard and use the free end of the jumper wire to test the individual segment LEDs. Just touch the end of the jumper to the corresponding column of breadboard pins for each LED connection as shown in the data sheet for your display type. Avoid the connections to the display's +5V input. All LEDs and the decimal points (if any) should light as you perform the tests. If some segments do not turn on, ensure you have all common-anode pins connected to +5V. And ensure all of the display's pins have gone into their corresponding breadboard receptacles. Sometimes a pin can bend up under the IC body so the pin looks connected, but it's not. If necessary, remove the IC, straighten bent pins and reinsert it in the breadboard.

Step 3.

Add the seven 330-ohm resistors to the display circuit as shown in **Figure 3.4**. This experiment does not use decimal-point LEDs. To keep wiring simple I recommend you shorten the resistor leads and insert them flush with the breadboard and close together in a row that bridges the wide "channel" in the middle of your breadboard, as shown in **Figure 3.5**. I also recommend you connect the segments pins to the resistors in alphabetical order, a, b, c, and so on, left to right.

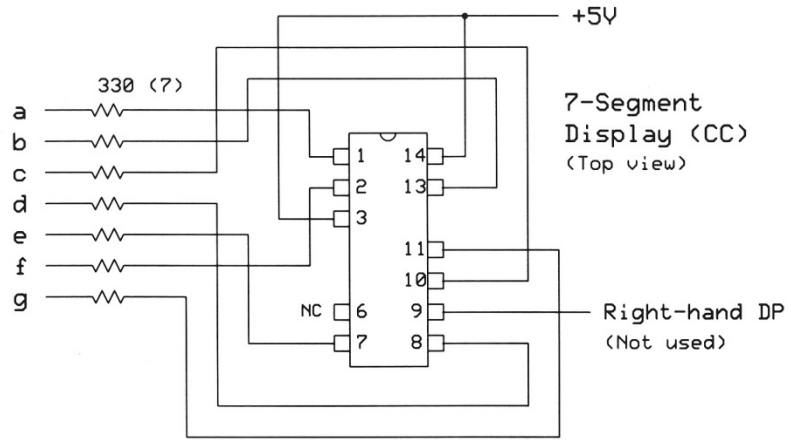


Figure 3.4.
Connections between 330-ohm resistors and a 7-segment display. Note the connection of *both* anode pins (pins 3 and 14) to +V. **Important:** Future diagrams will show the connections to the segments by letter and a single connections of power to the anodes, but without the top view of a 14-pin IC package.

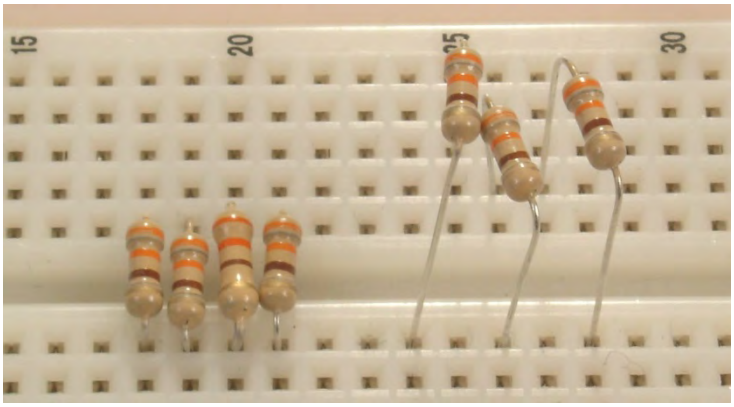


Figure 3.5.
The 330-ohm resistors on the left side of the breadboard mount across the channel in the center of a solderless breadboard. The short leads prevent accidental short circuits that can happen if you use resistors with long leads as seen on the right. Also, I find it easier to keep track of connections with resistors mounted close to a breadboard in an ordered fashion.

Instead of using a resistor for each segment and decimal point, could you use only one resistor between the common-anode pins and +5V and simply ground the pins for the segments you want to turn on? Find the answer at the end of this experiment.

After you have the seven resistors in place and the seven segments (a-g) wired to them, turn on power to the breadboard. Briefly connect the unconnected end of each resistor to ground to ensure all segment LEDs light. If not, recheck your connections. (If you connected a decimal point with a 330-ohm resistor, check it, too, although this experiment will not use it.)

Turn off breadboard power. What three segments would you connect to ground through their 330-ohm resistors to create a numeral 7 on the display? For those segments, connect ground to the "free" ends of the corresponding resistors.

IMPORTANT: DO NOT connect the display pins directly to ground. Without a current-limiting resistor you might burn out the LEDs in a segment.

Turn on power. Do you see the numeral 7 displayed? I connected the "free" end of the 330-ohm resistors for segments a, b, and c to ground and saw a 7 on my display. After you see the numeral 7, you can try other combinations of segments if you wish. When finished experimenting, turn off breadboard power and remove any wires between the 330-ohm resistors and ground. All other connections should remain in place.

Step 4.

Insert a 7447 or 74LS47 decoder/driver IC in your breadboard near the 7-segment display and the 330-ohm resistors, and connect it as shown in **Figure 3.6.** (I'll use the generic "7447" to indicate the decoder/driver IC.) This diagram does not show a display package. Instead it shows the anode connection and the segment LEDs by letter so you can connect the type of display you have to the 7447 IC. Ensure +5V power connects to Vcc (pin 16) on the 7447 IC and GND (pin 8) connects to a power-supply ground on your breadboard. The 7447 decoder/driver IC does not control a decimal-point LED. Another circuit must control this LED.

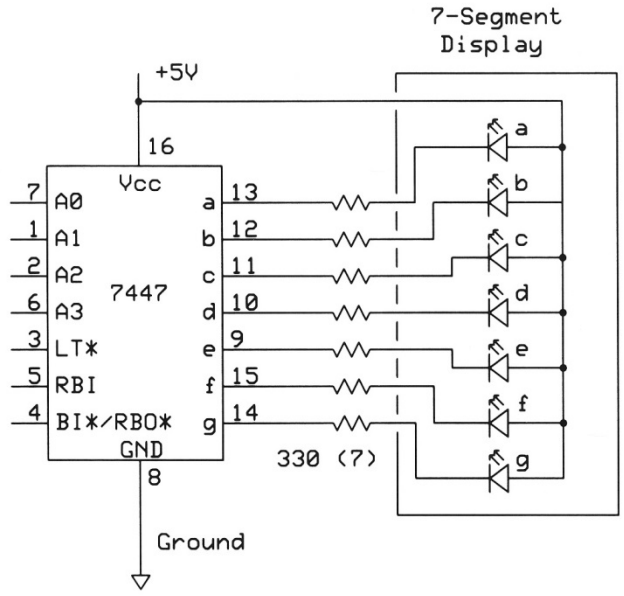


Figure 3.6.
Schematic diagram for the 7447 decoder/driver as it connects to a 7-segment common-anode display through current-limiting 330-ohm resistors. The display and the 7447 IC connect to +5. The display needs no ground connection.

Turn on power to your breadboard. Do any of the LEDs on the 7-segment display turn on? They should not. Connect a jumper wire between ground and LT* input (pin 3) on the 7447 decoder/driver. What happens on the display? A logic zero, or ground, at the LT* input causes all of the segment LEDs to turn on as an "LED test." A pushbutton or switch that grounds the LT* input lets people test the integrity of displays. No matter what digit a 7447 decoder/driver causes a 7-segment display to show, grounding the LT* input will force all segments to turn on.

Given the circuit you constructed, does the 7447 decoder/driver IC sink current through the display LEDs, or does it source current to the display? Find the answer at the end of this experiment.

Turn off power to your breadboards and remove the wire between ground and pin 3 on the 7447 IC. On the 7447 IC, connect pins 2 (A2) and 6 (A3) to ground. Turn on power. What numeral do you see on the display? You should see a "3." If you see the letter E, you have looked at the display upside down. Turn off power to your breadboard and remove the jumper wires that connect ground to pins 2 and 6 on the 7447 IC.

You did not have to connect pins 7 (A0) and 1 (A1) on the 7447 IC to +5V to create a logic-1 at these inputs. Unconnected *inputs* on this type of 7400-family IC "float" to a logic-1 state. But don't assume this condition

will always occur at unconnected pins in a real circuit. For testing, though, we can assume an unconnected input to a 7400-family IC presents the logic-1 state to internal circuits. More about this topic a bit later.

The information in **Table 3.1** shows all 16 combinations of 0's and 1's possible with four binary signals (A3-A0). You can see each Binary column represents a value, 8, 4, 2, or 1, and by adding the values across a row with a 1 in their columns you get the sum shown in the Decimal column. Ignore the columns in which your row has a 0. If a row shows 1001_2, for example, you have 8 + 1 = 9. Thus 1001_2 in binary equals 9 in decimal. Fill in the remaining Decimal numerals in the table.

Table 3.1. Binary inputs to a 7447 decoder/driver can create numerals and letters on a 7-segment display.

Binary				Decimal Value	Display Symbol
A3 = 8	**A2 = 4**	**A1 = 2**	**A0 = 1**		
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1	9	
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		
Pin 6	**Pin 2**	**Pin 1**	**Pin 7**	< – 7447 pin number	

Step 5.

The numbers at the bottom of the Binary columns refer to the pin numbers for the 8's, 4's, 2's, and 1's inputs on a 7447 decoder driver IC. Connect 7447 pin 2 (A2) and pin 1 (A1) to ground (logic-0) and leave pin 6 (A3) and pin 7 (A0) unconnected (logic-1). Turn on power and you should see a 9 appear on the display. The logic-0 and logic-1 settings correspond to the binary value for 9, or 1001_2. After you see the digit 9 on the display, leave power on and carefully remove the jumper wires to pins 1 and 2 on the 7447 IC. The display should go blank with all LEDs off.

Start with the first entry in **Table 3.1**, 0000_2, and ground the four inputs, A3, A2, A1, and A0 on the 7447 decoder/driver IC. Enter in **Table 3.1** the numeral or symbol you observe on the 7-segment display. Continue with the next value, 0001_2, for which you ground the A3, A2, and A1 inputs, but do not ground the A0 input. In turn, make connections for each of the 16 binary values and note the numerals or symbols you observe. The single-digit display cannot show a 10, 11, 12, and so on. Instead it shows symbols. The diagram in **Figure 3.7** shows what I observed for all 16 input conditions, 0000_2 through 1111_2 at the 7447 decoder/driver IC.

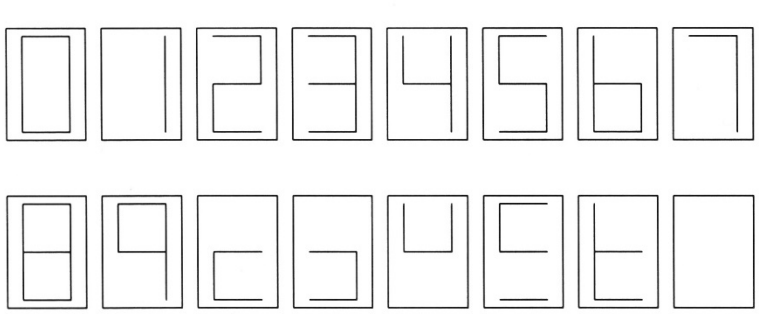


Figure 3.7.
The fifteen numerals and symbols created by a 7447 decoder/driver IC on a 7-segment common-anode display. The binary input 1111_2 "blanks" the LEDs.

Set the four inputs on the 7447 decoder/driver IC to display any numeral or symbol. Connect a jumper from the 7447 BI*/RBO* signal (pin 4) to ground. What happens to the display? My display turned off. The BI*/RBO* input overrides the 4-bit binary value and forces the display to become blank. Can you think of a situation in which you might want to "blank" a display of digits?

A weight scale in a grocery store might blank its display until the weight stabilizes and can show a final value. If a device counts objects, you might want to see only the final result. The counting circuit could blank the display as the counter increments, then turn on the display after it has counted all objects.

Step 6.

In this step, you will connect the 7447 decoder/driver IC to the 74LS93 binary counter circuit created in Experiment 2.

a) Turn off power to your breadboard and remove any jumper wires that connect to pins 1 through 7 on the 7447 decoder/driver.

b) Refer to **Figure 2.14** in Experiment 2. If you have LEDs and resistors connected to the 74LS93 counter circuit assembled in Experiment 2, remove them from your breadboard.

c) Refer to **Figure 2.15** in Experiment 2. If you used a 74LS04 hex inverter, carefully remove it, and any LEDs and resistors connected to it, from your breadboard.

d) Add the four wires highlighted with a large circle in **Figure 3.8** to connect your 74LS93 counter to the 7447 decoder/driver IC. Remember the 74LS93 counter and 555 timer circuit must have a shared +5V supply connection and a common ground connection or the circuit will not work properly. **Figure 3.9** shows the complete circuit for the 555 timer, 74LS93 counter, 7447 decoder/driver, and 7-segment display.

e) Recheck your wiring and then turn on power. What do you see on the display?

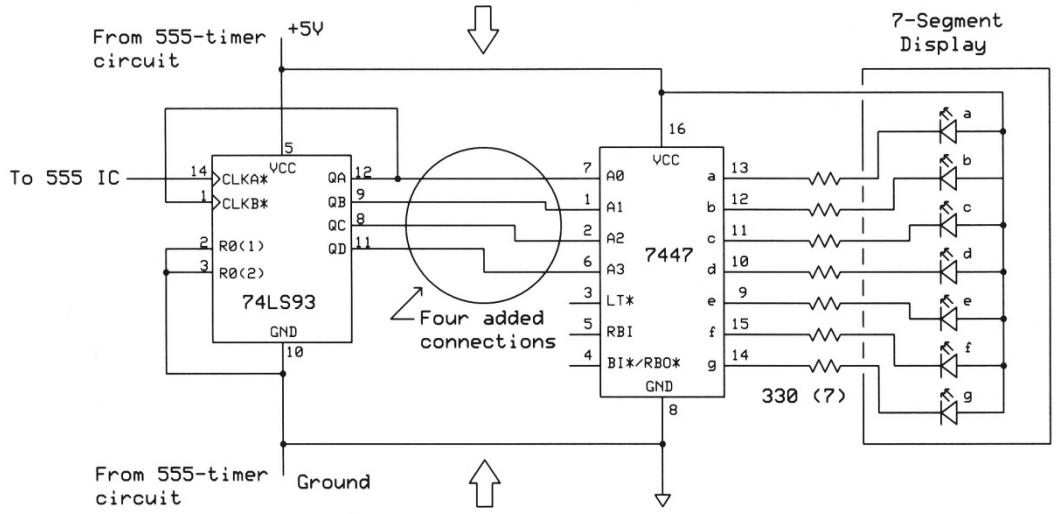


Figure 3.8.
This schematic diagram highlights the four new connections (large circle) between the counter circuit created in Experiment 2 and the 7474 decoder/driver assembled in this experiment. Ensure power and ground (large arrows) connect to the counter and the display circuits as shown.

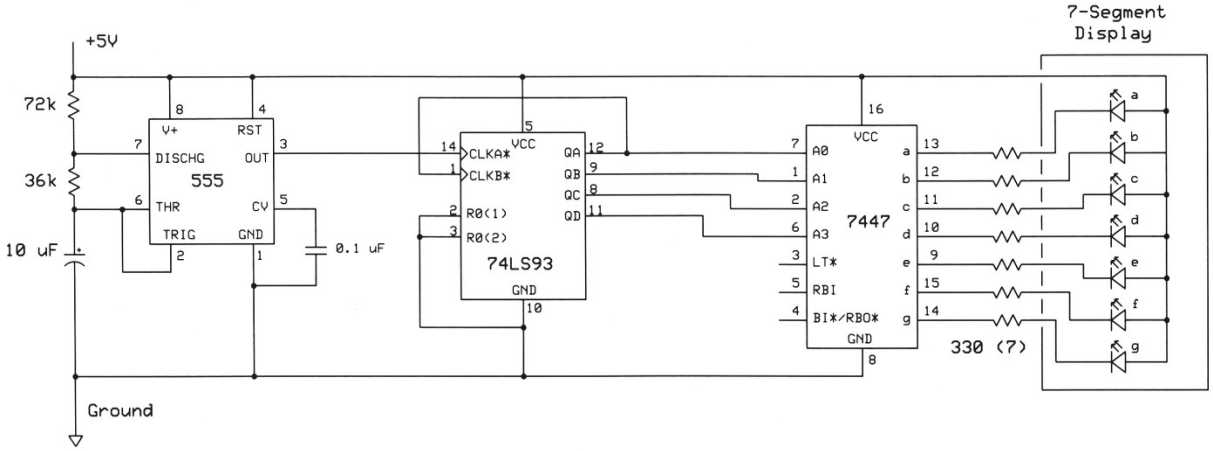


Figure 3.9.
The complete circuit for the 555 timer, 74LS93 counter, 7447 decoder/driver, and 7-segment display.

My display showed numerals 0 through 9 and then the five symbols shown earlier in **Figure 3.7**. After it displayed the last symbol, the LEDs turned off and then started to count again with the numeral 0. The five symbols and the "display off" conditions for binary inputs 1010_2 through 1111_2, don't have "real world" value and a practical circuit would never display them. We really want only the 0 through 9 count, which you can get in two ways. First, "convert" the 74LS93 binary counter (divide-by-16) into a decimal (divide-by-10) counter, or second, replace the 74LS93 *binary* counter with a 74LS90 *decimal* counter IC.

Step 7.

Let's not use a decimal-counter IC now. We'll modify the 74LS93 counter circuit instead. Turn off power to your breadboards. The 74LS93 counter provides two reset inputs, R0(1) and R0(2) at pins 2 and 3 respectively. If *both* reset inputs have a logic-1 signal applied to them, the counter will reset to 0000_2, and the display will show the numeral 0. To test the 74LS93 reset, remove the ground connections to pins 2 and 3 on the 74LS93 counter IC.

Turn on power. Does the display constantly show the numeral 0? It should because both the R0(1) and R0(2) reset inputs have floated to a logic-1 and caused the counter to reset. As long as *both* of these inputs remain at a logic-1 state, the counter remains reset.

Remember, an unconnected pin on a 7400-series IC "floats" to a logic-1 state, so you do not have to connect these inputs to +5 volts, although you might want to as a way to avoid circuit problems in real circuits. Professional circuit designers would add a pull-up resistor between the positive supply voltage (+5V in this experiment) and unconnected inputs that require a logic-1 signal to operate properly, as shown in **Figure 3.10**. If an unused input requires a logic-0 signal, you must connect it to ground.

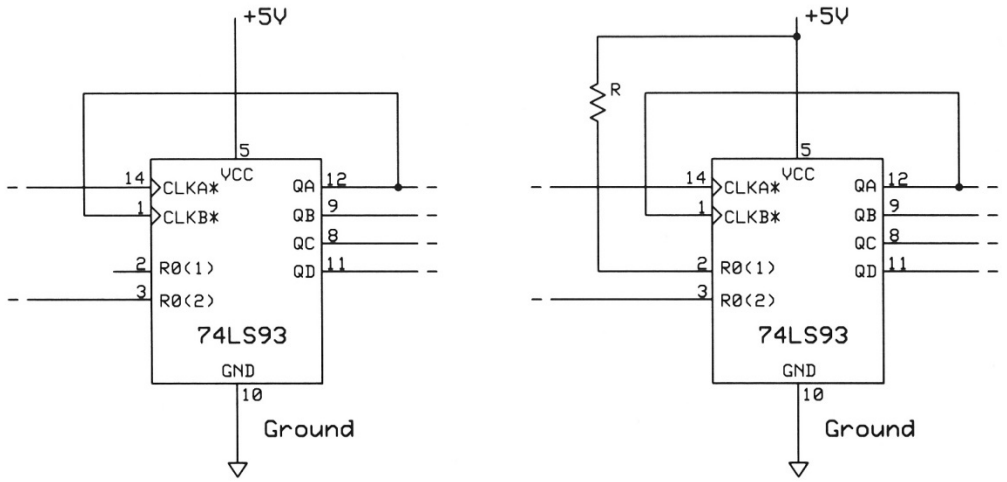


Figure 3.10.
Rather than assume an unused input, such as R0(1), will float to a logic-1 state (left), circuit designers use a pullup resistor between the 74LS93 R0(1) and the IC's power supply to ensures a logic-1 state (right).

Connect a jumper from ground to either pin 2 or pin 3 on the 74LS93 counter. What happens now? The count should resume, 0, 1, 2, and so on. Internal to the counter, the R0(1) and R0(2) inputs go to an AND circuit, also called an AND "gate" that requires a logic-1 at pin 2 AND a logic-1 at pin 3 to cause a reset. The timing diagram in **Figure 3.11** shows the relationship between the R0(1) and R0(2) reset inputs and the counter operation.

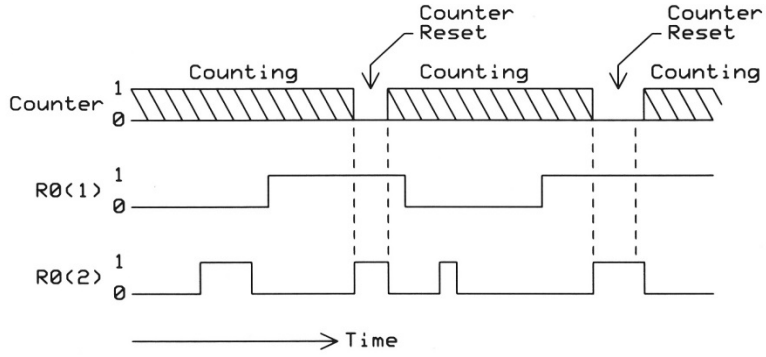


Figure 3.11.
This timing diagram, read from left to right, shows how the 74LS93 R0(1) and R0(2) reset inputs affect counting. External circuits must provide a logic-1 signal to both inputs to force the counter into its reset condition. The reset condition (0000_2) exists for as long as both reset inputs have a logic-1 applied to them.

Step 8.

You can use the binary outputs from the counter to cause the 74LS93 counter to reset. Turn off power to your breadboards and remove any connections to the reset inputs at pins 2 and 3 on the 74LS93 counter IC. Add a

jumper wire from the counter's QB output (pin 9) to the reset input R0(1) at pin 2 as shown in **Figure 3.12.** A logic-1 at the QB output will reset the counter back to 0000_2, and the display will show 0. The counter-reset input R0(2) at pin 3 remains unconnected and supplies a logic-1 to the counter.

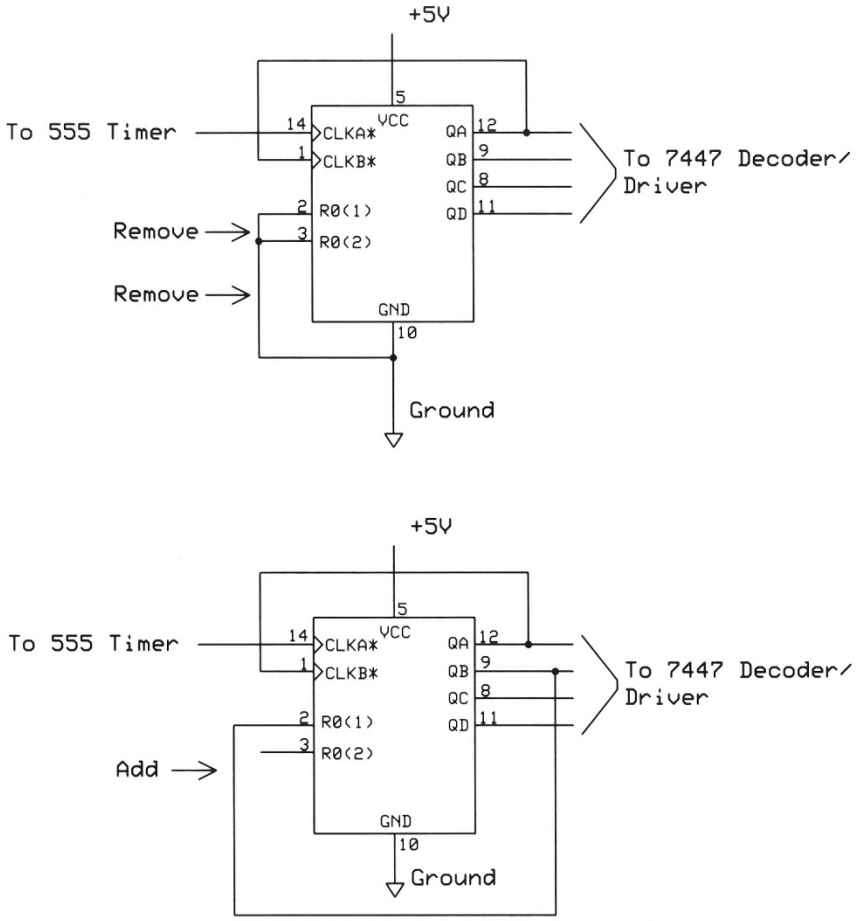


Figure 3.12.
Remove the jumper or jumpers between the 74LS93 reset inputs and ground (upper) and then connect a jumper between the counter QB output and the R0(1) input at pin 2.

Before you turn on power, look at the information in **Table 3.2**. When the counter reaches 0000_2 and starts to count up, 0001_2, 0010_2, 0011_2, and so on, when does the QB output first become a logic-1, which would reset the 74LS93 counter?

Table 3.2. Binary outputs from a 74LS93 counter and decimal equivalents.

Binary-Counter Outputs				Decimal
QD	QC	QB	QA	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5

(Continued on the next page.)

0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Turn on power to the breadboards. How does the display behave? You should see the numerals alternate 0-1-0-1-0 and so on. The QB output changes from a logic-0 to a logic-1 state as the counter outputs change from 0001_2 to 0010_2, or from 1 to 2. The QB output exists as a logic-1 only for an instant, which is enough to reset the counter back to 0000_2. **Figure 3.13** shows the timing relationship between the counter output and the reset inputs R0(1) and R0(2). The QB output remains a logic-1 for only a few nanoseconds (billionths of a second), so even though the counter reaches 0010_2, you never see a 2 on the display.

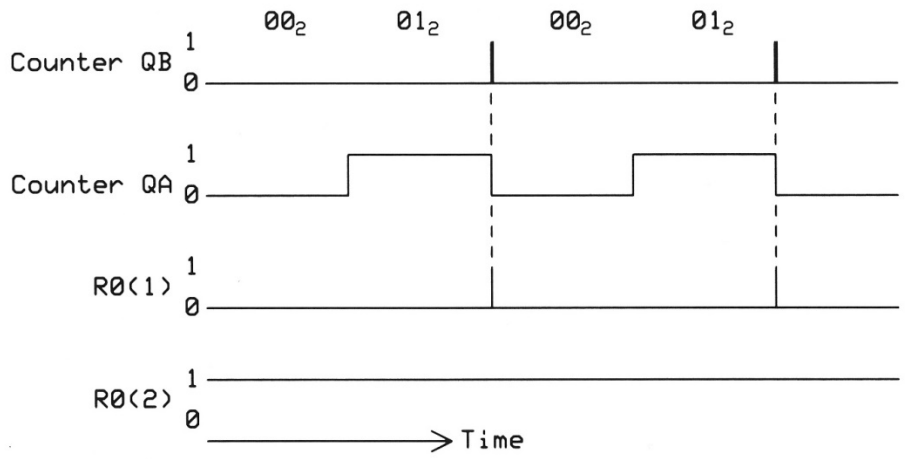


Figure 3.13.
This timing diagram shows counter outputs QA and QB. The binary numbers along the top show the binary value for these two outputs. As soon as the QB output becomes a logic-1, it causes the counter to reset to 00_2. The diagram artificially widens the short QB pulse for clarity.

Turn off power to your breadboards and remove the end of the reset wire from the QB output at pin 9 and connect it to the QC output at pin 8. The schematic diagram for the 74LS93 counter in **Figure 3.14** shows the connection to change. Again, look at the information in **Table 3.**2 and determine when the QC output first goes from a logic-0 to a logic-1 state. Assume the counter starts at 0000_2. Based on that information, when will the counter reset and what do you expect to see of the 7-segment display? Turn on power to your breadboards and note what you observed.

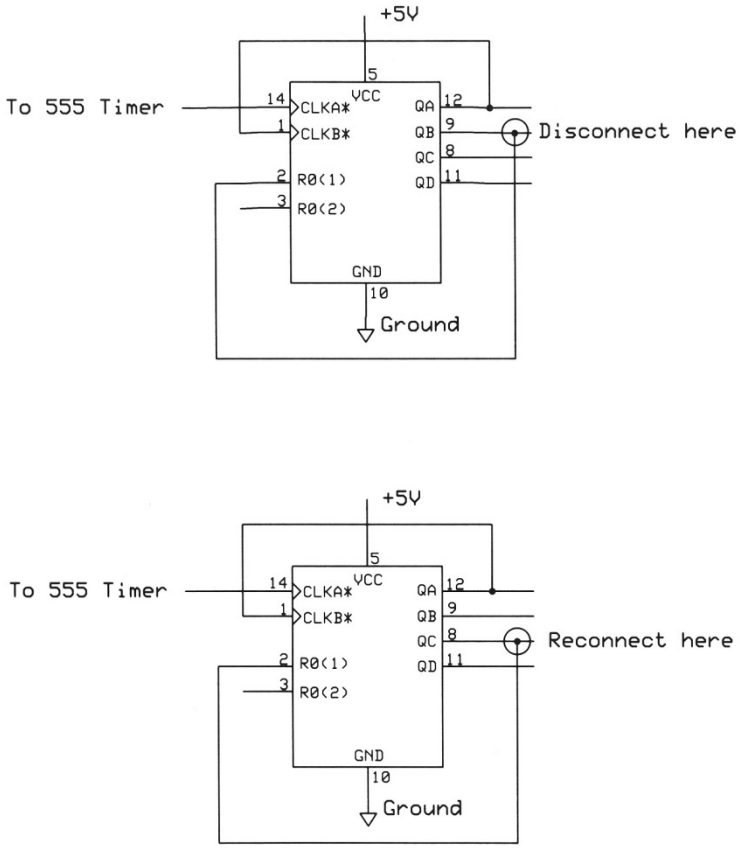


Figure 3.14.
Disconnect the end of the reset jumper at the counter QB output (pin 9) and reconnect it to the QC output at pin 9. The circles in the diagram highlight the changes. Now when the QC output goes to a logic-1, the counter will reset.

My display showed the digits 0, 1, 2, 3, 0, 1... When the counter reached the value 4, or 0100_2, the logic-1 signal from the QC output immediately reset the counter, which started to count again. **Figure 3.15** shows the timing information for this reset condition.

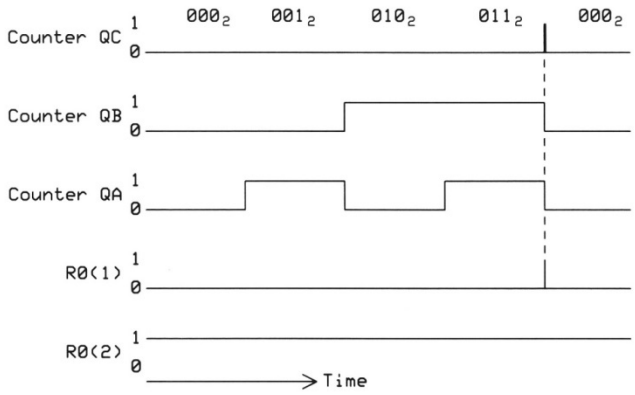


Figure 3.15.
This timing diagram shows counter outputs QA, QB, and QC. The binary numbers along the top show the binary values for these three outputs. As soon as the QC output becomes a logic-1, it causes the counter to reset to 000_2. This diagram artificially widens the QC pulse for clarity. Note the counter R0(2) reset input remains a logic-1.

Step 9.

Can you use the counter outputs to reset the count to 0 just as the counter output goes from 9 (1001_2) to 10 (1010_2)? Refer back to **Table 3.2**. What connections from the counter outputs would you make to the two reset inputs? Turn off power to the breadboards, make your connection(s) and turn on power. Did your circuit count only between 0 and 9?

On the 74LS93 counter:

a) I removed the connection from the counter QC output to R0(1),

b) Then I connected the QB output (pin 9) to the R0(2) reset input (pin 3),

c) And I connected the QD output (pin 11) to the R0(1) input (pin 2), as shown in **Figure 3.16**.

Now when the QB output AND the QD output both become logic-1 – the counter's outputs changes from 1001_2 to 1010_2 – the counter immediately resets back to 0000_2. If you didn't choose these connections, remove your connections to the reset inputs and make the connections shown in **Figure 3.16** and turn on power. Your counter should now cause the display to show, ...7, 8, 9, 0, 1, 2, and so on. You should not see any of the unusual symbols on the display.

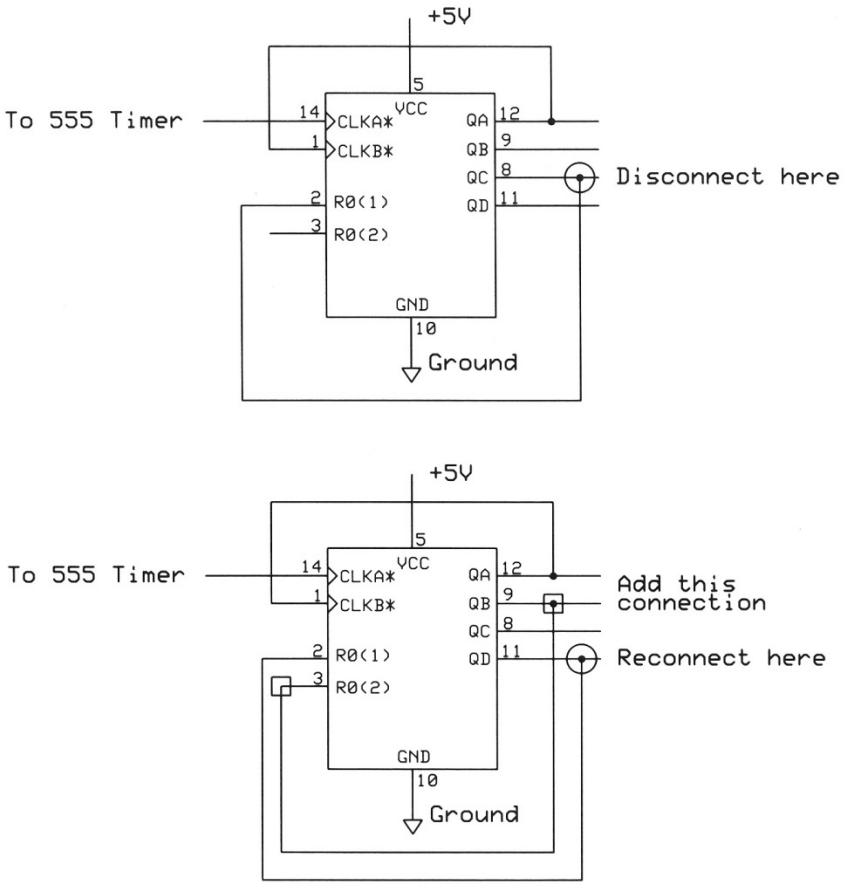


Figure 3.16.
Move the reset connection at the counter QC output (pin 8) to the QD output (pin 11). Circles in the diagrams highlight this change. Then add a jumper between the counter QB output (pin 9) and the R0(2) reset input. The squares highlight this added connection. Now the counter will reset as soon as the QB and the QD output both become logic-1.

The use of counter outputs to reset a counter makes for a good experiment, but circuit designers avoid this technique and use a *synchronous* reset instead. That type of circuit design goes beyond the scope of this book, though.

Step 10.

When you need a decade, or divide-by-10 counter, I recommend you choose an IC such as the 74LS90 that requires no connections between Q outputs and reset inputs. The 74LS90 ICs do have reset inputs, though.

In this step, you will substitute a 74LS90 decimal-counter IC for the 4-bit 74LS93 binary counter IC. The diagram shown in **Figure 3.17** provides the schematic symbol for a 74LS90 and a 74LS93. Note the 74LS90 decimal counter has the same pin configuration as the 74LS93 except for two additional reset inputs, R9(1) at pin 6 and R9(2) at pin 7. The reset inputs R0(1) at pin 2 and R0(2) at pin 3 remain the same as those on a 74LS93 binary counter.

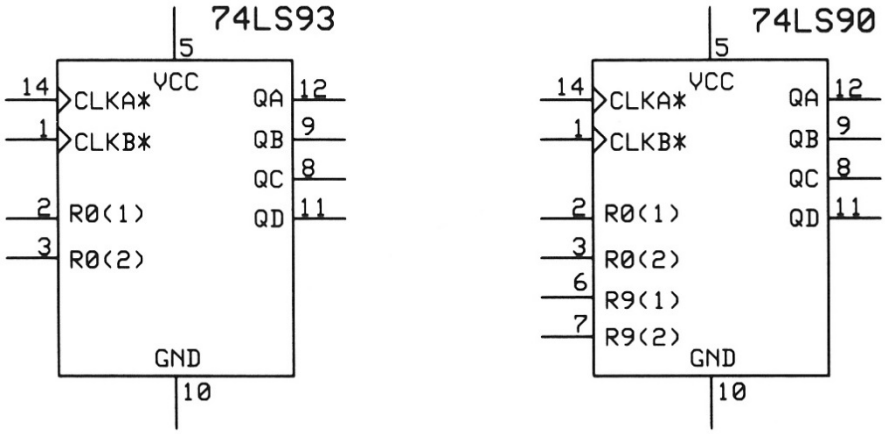


Figure 3.17.
The 74LS93 (left) and 74LS90 (right) counters provide pin-for-pin compatibility, although the 74LS90 decimal counter includes two additional reset inputs, R9(1) and R9(2).

The R9(1) and R9(2) reset inputs let circuit reset the counter to 9, or 1001_2. I cannot think of a practical reason to reset a decimal counter to 9, but obviously the IC designers did. As in the case of the 74LS93 used earlier in this experiment, the R9 inputs must *both* provide a logic-1 signal to the counter to force a reset to 9.

If you have power applied to your breadboards, turn it off now. Carefully note the pin-1 location of the 74LS93 counter IC in your breadboard. You can mark the position on the breadboard, or insert one end of a spare wire in the pin-1 column on the breadboard to note the position. You will need to know the pin-1 position when you replace the 74LS93 IC with a 74LS90 IC.

Carefully remove the 74LS93 IC from your breadboard and set it aside. Take a 74LS90 decimal-counter IC and insert it in your breadboard in place of the 74LS93 IC you just removed. Ensure you have pin 1 of the 74LS90 in the same position as you used for pin 1 on the 74LS93 counter. Although you have not changed any wiring, double check your connections to the 74LS90. The circuit diagram in **Figure 3.18** shows the connections you should have now.

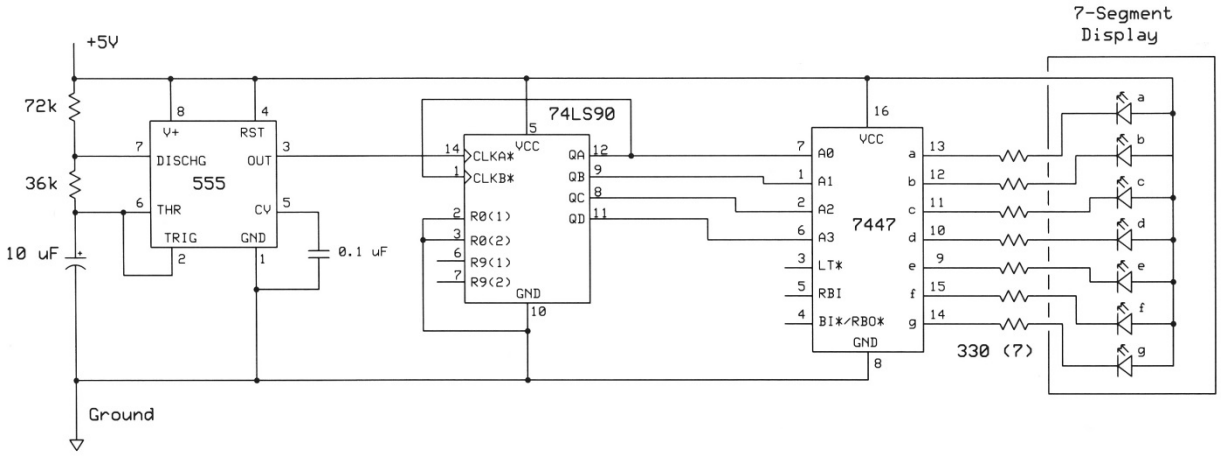


Figure 3.18.
This circuit diagram shows a 74LS90 decimal counter substituted for the 74LS93 binary counter used so far in this experiment. **CAUTION:** This circuit requires a minor modification to work properly as explained later.

After you confirm the connections to the 74LS90 counter, turn on power. What do you see? Does the counter count 0, 1, 2, and so on? If not, what digit does the 7-segment display show? Do you know why the display shows what it does?

For now, the circuit shown in **Figure 3.18** will not count. Both R9 reset inputs lack a connection to logic-0, so they "float" to a logic-1 and thus hold the counter in the reset-to-9 (1001_2) state. Thus the display constantly shows the numeral 9. Turn off power to your circuit, and on the 74LS90 IC connect the R9(1) pin (pin 6) to ground. Now turn on power again. Does the displayed count show: 0, 1, 2... ...8, 9, 0, 1, and so on? It should.

As shown in Experiment 2, **Figure 2.17**, you can take the QD output from one counter and use it as the input to another counter. With three 74LS90 counters, you could display values between 0 and 999. Add more counters and displays and you can count higher. As the QD output on a counter goes from a logic-1 to a logic-0 state, it "clocks" the next counter IC in the chain. Remember the 74LS90 and 74LS93 counters require a connection between the QA output and the CLKB* input. If you plan to go on to the next optional step, leave your circuit set up in the breadboards, but turn off power.

Step 11 - Optional.

The 7447 decoder/driver IC used with a 7-segment display has two signals, RBI and BI*/RBO* that serve a special purpose. Suppose you have a counter with a 4-digit numeric display. Instead of counting, 0001, 0002, 0003, and so on, you want to turn off, or disable, all leading 0's and have the display show only 1, 2, 3... You can use the RBI and BI*/RBO* inputs to turn off, or blank, the leading 0's. (In a calculator, for example, leading zeros would become a nuisance very quickly.)

For this step, assume you have created a 4-digit decimal counter (0000 to 9999) and the counter, decoder/driver, and 7-segment display on your breadboard represent the 10's digits. So your display would show the numeral 7 in the number 1374. On your 7447 decoder driver, connect the Ripple Blanking Input (RBI) at pin 5 to ground, or logic-0. This RBI signal indicates to the 7447 IC that higher decades – the 100's *and* the 1000's digits – have a value of 0 from their associated counter. That condition would exist for any count from 0000 to 0099.

Turn on power to your circuit and watch as the counter increments 1, 2, 3, and so on. What do you observe? With the RBI pin on the 7447 decoder connected to ground, or logic-0, the count proceeded ...7, 8, 9, blank, 1, 2... A zero never appears. Instead, the display goes blank (off). In our imaginary counter, the complete 4-digit

display would show blank, 1, 2... ...11, 12, 13... It would not show 0001, 0002, and 0011, 0012, 0013, so on. Thus, any leading 0's would prevent their associated display from turning on.

Turn off power to your breadboard and connect an LED and a 330-ohm resistor to the 7447 BI*/RBO* pin (pin 4), as shown in **Figure 3.19**. BI*/RBO* stands for Blanking Input/Ripple Blanking Output, so the pin can serve as an input or as an output. Turn on power to your circuit and note when the added LED turns on or off. What did you observe?

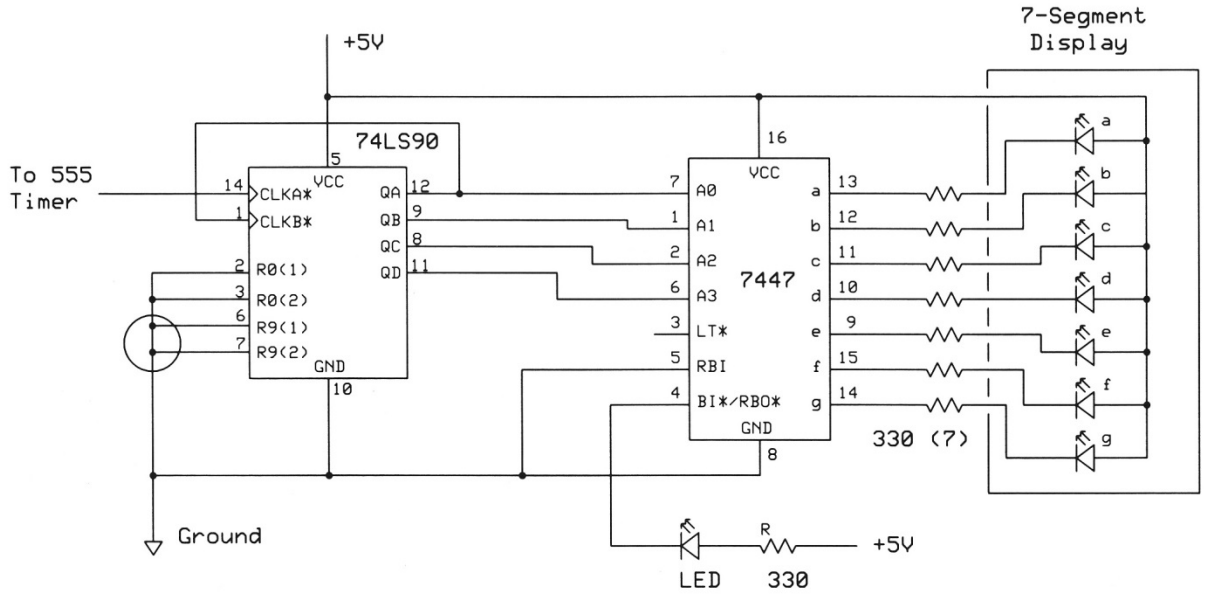


Figure 3.19.
Connections that illustrate operation of the Ripple Blanking Input and Ripple Blanking Output for a 7447 decoder/driver IC. Note the proper ground connections for the R9(1) and R9(2) reset inputs on the 74LS90 counter (circled).

The LED turns on when the display goes blank. In this circuit, a lit LED represents a logic-0 at the BI*/RBO* pin. In essence, the 7447 "reports" that all higher decades – hundreds, thousands, tens of thousands, and so on, including your digit – would display a 0, but have instead turned the 7-segment displays off to show only a blank.

This situation sounds a bit complicated, but the diagram in **Figure 3.20** shows how you would use the RBI and BI*/RBO* signals in a "daisy chain" to blank, or suppress leading zeros in a display. In this circuit, I have connected the LED Test (LT*, pin 3) input on each decoder/driver IC to one pushbutton so you could test all of the segment LEDs. For the sake of clarity, this circuit diagram does not show connections to 7447 inputs (A3 – A0), power connections, or connections a through g on 7-segment displays.

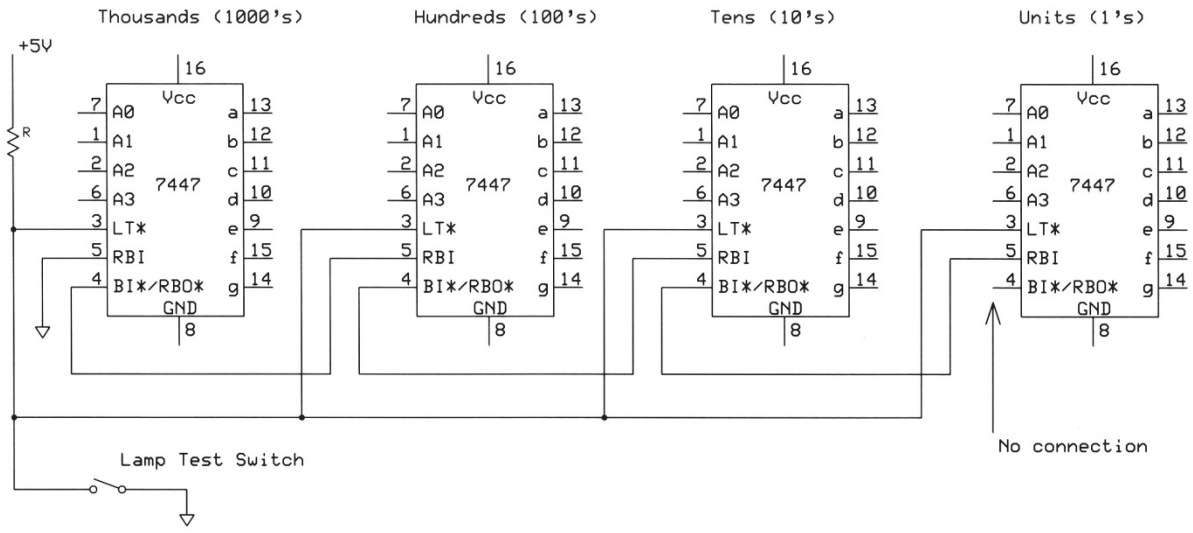


Figure 3.20.
This partial schematic diagram shows the connections between 7447 decoder/driver ICs when you must blank, or turn off, leading displays of leading zeros. This circuit also includes connections for a normally open (NO) LED-test switch.

In the next experiment, you will learn how a microcontroller can control LEDs and give you a lot of flexibility in how you display information. Software could determine when to blank leading zeros, for example. Circuits would not need extra hardware to translate 4-bit values into 7-segment codes.

Answers

Experiment 3, Step 3:

One current-limiting resistor would limit the total current to the entire display. As more segments turn on, the display would get dimmer. Depending on the 7-segment display in use, some segments might not turn on at all. Use a resistor for each LED.

Experiment 3, Step 4:

The 7447 decoder/driver IC sinks current. Current enters the display from the +5V power source, passes through an LED and its associated resistor, then into the corresponding segment pin on the 7447, then on to ground. The a through g inputs in the 7447 have what we call an "open-collector transistor." This transistor will create a path to ground (logic-0), but it will not go into a logic-1 state when it turns off an LED. You will learn more about transistors in another experiment.

Experiment No. 4 – Microcontrollers and LEDs – Individual LED Control

Abstract

In this experiment you will learn how a microcontroller integrated circuit can control individual LEDs. You will discover how a microcontroller gives you complete control over LEDs, without the need for a decoder/driver. Software provides the key element is such cases. This experiment emphasize basic on-off LED operations. After you understand how to control the LEDs with software commands, you can have the LEDs operate in almost any way you choose.

Keywords

Microcontroller, MCU, LED, Propeller P8X32A, Spin language, software

Requirements

(1) - Parallax Propeller P8X32A QuickStart module
(1) - USB cable

Introduction

The integrated circuits (ICs) you used in other experiments have a certain appeal. They give circuit designers basic functions they may connect in different ways to accomplish a given goal. Microcontrollers (MCUs), on the other hand, come unable to do anything useful until someone writes a step-by-step program. Then they can perform specific tasks, such as add two values, turn a machine on, count coins, and so forth. MCUs offer a lot of flexibility. Suppose you have "hard wired" a counter and decide to change it from a divide-by-10 counter to a divide-by-11 counter. That change involved a lot of redesign and rewiring, and perhaps a new printed-circuit board in a commercial product.

In an MCU, you would simply change part of a command from, say, `count_value / 10` to `count_value / 11`. So instead of dividing the `count_value` by 10, the program would divide it by 11. Likewise when you use an MCU as a timer, you do not change resistor or capacitor values as you did in Experiment 2 with a 555-timer IC. Instead, you change a value to increase or decrease timing periods. And MCUs do not need a 7-segment decoder/driver IC, either. A short program can easily control a 7-segment display to show numerals as well as a few letters and symbols.

MCUs rely on "ports," usually called input-output (I/O) ports, to communicate with external devices (**Figure 4.1**). An *analog* input port could measure a voltage from an external sensor, perhaps a thermometer. An analog output port could create a voltage to control a device. *Digital* ports work with logic-0 and logic-1 signals. A digital input port lets an MCU detect the state of one or more signals, and a digital output port lets the MCU send a logic-0 or -1 signal to external devices.

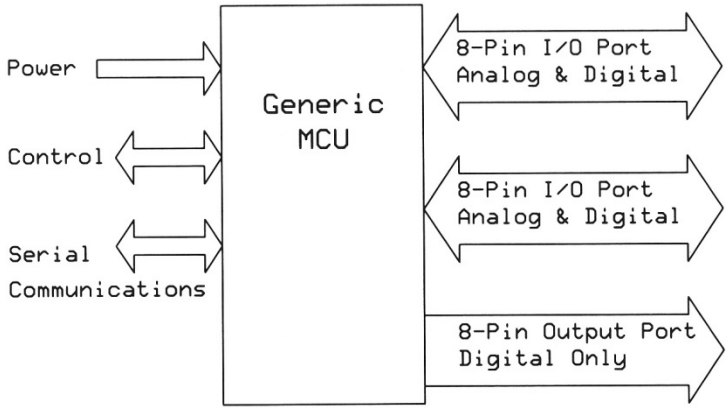


Figure 4.1.
A block diagram of a microcontroller with three 8-bit I/O ports. The arrows indicate two ports can operate as inputs, outputs, or a mix. The third port can operate only as an output port.

The I/O ports can comprise one or more digital signals. A small 8-pin MCU might have only three or four I/O signals, while the Propeller MCU offers as many as 32 digital I/O pins. Data sheets – these days more like books – provided by the MCU manufacturers give engineers and experimenters information about the arrangement of ports on a given MCU IC.

Experiments with MCUs

When I experiment with MCUs, I like to have a complete "computer-on-a-board" to work with rather than individual MCU ICs. These boards often provide LEDs, pushbuttons, a built-in power regulator, USB connection to a "host" PC, and other ready-to-use components. Thankfully, manufacturers give us many choices of such boards that include the Arduino Uno, the Digilent ChipKit Uno32, and the Atmel AVR XMEGA shown in **Figure 4.2.**

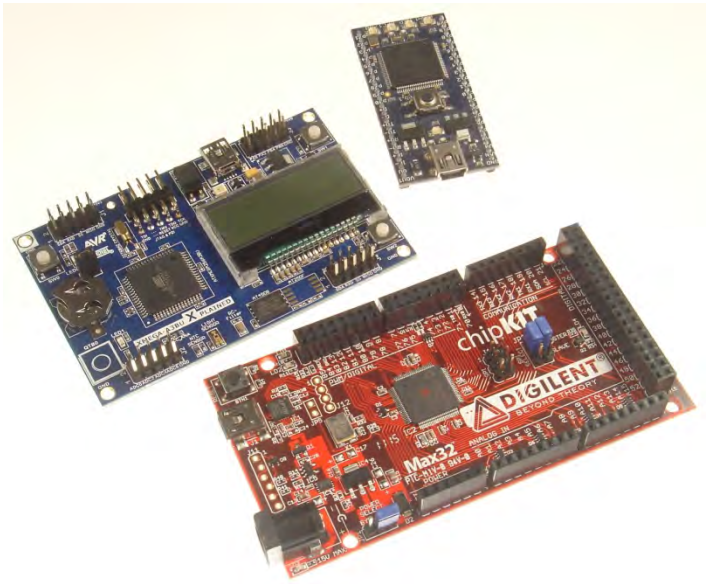


Figure 4.2.
A wide variety of MCU boards let engineers, experimenters, and students work with many types of MCUs and on-board devices such as LEDs, sensors, touch-sensitive buttons, and so on. Clockwise from the top, an ARM mbed board, a Digilent ChipKit MAX32 Arduino, and an Atmel AVR XMEGA-A3BU board.

This experiment and those that follow use an inexpensive Parallax Propeller P8X32A QuickStart board shown in **Figure 4.3**. Newcomers as well as experienced electronics enthusiasts will find The Propeller board easy to use and program. Parallax offers free software that simplifies creation of code, and the "Propeller Tool" software does not confuse people with dozens of "buttons" and menus. As people often say, "Try it; you'll like it." And by the way, the Propeller comes with eight MCU cores, or processors in one chip. So software can take advantage of one or more processors to handle complicated tasks. All processors share the 32 I/O lines.

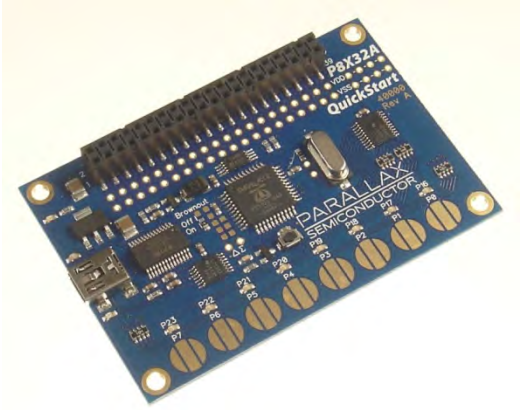


Figure 4.3.
The Parallax Propeller P8X32A QuickStart board includes eight LEDs and eight touch switches, as well as 12 unused signals for off-board experiments. All 32 I/O pins have receptacles in the 40-pin upright connector seen in the photo.

The Propeller MCU provides 32 general-purpose I/O signals, or "pins" labeled P0 through P32. Together, these signals form Port A. Newer members of the Propeller family might include an additional 32 signals labeled Port B. The P8X32A module includes eight LEDs and eight touch-switch controls that connect to 16 of the I/O pins. But you can still use these 16 I/O pins in experiments. Communication and programming signals require an additional four I/O pins. **Table 4.1** shows the functions of the 32 pins. In you don't want or need the touch switches and LEDs you could purchase a Propeller prototype board that comes without them. Visit the Parallax Web site for information about prototype boards and programming modules: www.parallax.com.

Table 4.1. Propeller signals on the P832A QuickStart board.

Pin Name	Function
P0	Touch Button P0
P1	Touch Button P1
P2	Touch Button P2
P3	Touch Button P3
P4	Touch Button P4
P5	Touch Button P5
P6	Touch Button P6
P7	Touch Button P7
P8	Available
P9	Available
P10	Available
P11	Available
P12	Available
P13	Available
P14	Available
P15	Available

Pin Name	Function
P16	LED P16
P17	LED P17
P18	LED P18
P19	LED P19
P20	LED P20
P21	LED P21
P22	LED P22
P23	LED P23
P24	Available
P25	Available
P26	Available
P27	Available
P28	Serial Clock
P29	Serial Data
P30	Transmit
P31	Receive

IMPORTANT: The Propeller P8X32A QuickStart board use 3.3-volt logic rather than 5-volt logic used in earlier experiments. Manufacturers sell logic-level converter ICs in surface-mount packages and you can use them with an adapter in solderless breadboards. Unless otherwise noted, experiments that involve a Propeller MCU will use 3.3-volt power.

Step 1.

Disconnect the power supply from all breadboard circuits created in other experiments. Connect a voltmeter or volt-ohm-meter (VOM) to your adjustable power supply. Turn on power and adjust the supply output to 3.3 volts. If you cannot "dial in" 3.3 volts exactly, a voltage close to, but under 3.3 volts should work. I have powered 3.3-volt logic with two 1.5-volt alkaline D-size cells connected in series. Turn off the power supply. You will need 3.3-V power later and I want to ensure you have your power source set for the correct voltage. You will not need a power supply in this experiment, though.

The P8X32A board obtains power via the USB cable that connects it to your host computer. Unless noted otherwise, you will not supply external power to the P8X32A board.

Step 2.

If you do not have the Propeller/Spin Tool Software installed on your PC, refer to instructions on the Parallax Web site at: http://www.parallax.com/downloads/propeller-tool-software. Parallax provides the Propeller Tool as a free download. It comes with the Parallax Serial Terminal software that many experiments use so you can communicate with the MCU or MCUs and see results on your computer's monitor.

Many experimenters and professionals use the C language for their programs. The Propeller can run C programs, but I prefer the Propeller's "Spin" language that greatly simplifies programming and use of the eight MCU processors in the Propeller IC. All experiments use the Spin language, commands, and functions. You'll learn more about the Propeller MCU as you go through experiments.

You also should download the "Experiments Software" folder from the Propeller Object Exchange: http://obex.parallax.com/. This folder contains all of the Propeller software used in the experiments. Place the contents of this folder in a new folder called "Propeller Workspace." You will refer to the contents of this folder often when experiments involve the Propeller MCU.

After you have loaded the Spin Tool software, connect your P8X32A board to your PC through a USB cable. Start the Propeller Tool, which should "find" the attached MCU board. Press the F7 keyboard key and the terminal should display a message such as "Propeller chip version 1 found on COM11," as seen in **Figure 4.4**. The communication-port number for your PC will likely differ from the 11 used by my computer. Click on "OK" to close the message window. Let the Propeller Tool continue to run. You will use it soon.

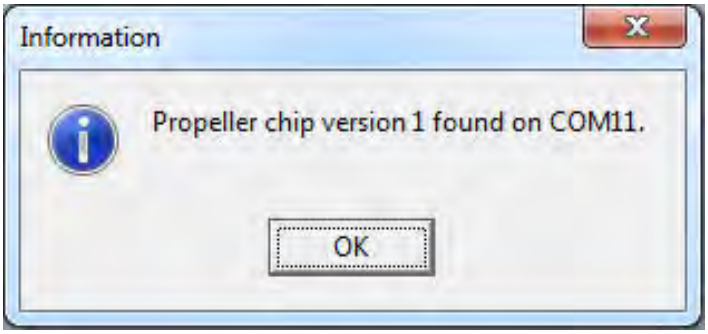


Figure 4.4.
Screen image from the Propeller Tool that shows the COM port the PC uses to connect with my Propeller board. Press F7 to get this information.

Step 3.

Because the Propeller QuickStart board includes eight LEDs, you do not need to put LEDs on a breadboard in this experiment. The eight LEDs connect to the P16 through P23 signals on the board. A logic-1 signal on a pin turns on the corresponding LED.

In this step you will learn how control the LEDs on the Propeller board and how the MCU can act as an 8-bit binary counter. The Propeller MCU uses a command, `DIRA`, to set the direction for the 32 I/O pins, P0 through P31. (As explained above, the A stands for Port A.) You can set individual pins, groups of pins, or both as either digital inputs or digital outputs. The command `DIRA[8] := 1`, for example, sets MCU I/O pin P8 as a digital output, while the command `DIRA[3..5] := %111` establishes pins P3, P4, and P5 as outputs. The `%111` notation indicates a binary value of 111, which sets a logic-1 in the `DIRA` operation for each of the three I/O pins. If you had used instead, `DIRA[3..5] := %101`, you would set P3 and P5 as outputs and P4 as an an input.

To control the eight on-board LEDs, a programmer would use the statement `DIRA[16..23] := %11111111`, which sets all eight I/O pins, P16 through P23, as outputs. You could use a hexadecimal value ($FF) in place of the binary information, but I prefer binary because you can immediately see the pin settings.

The `OUTA` command controls the state of individual or groups of pins set as outputs. In the case of the LEDs, you could use the command `OUTA[16..23] := %10101010` to turn four LEDs on and four LEDs off. When a program must display a changing value on the LEDs, a command such as: `OUTA[16..23] := Data_for_LEDs` will do the job. Here, the variable `Data_for_LEDs` – previously defined in a Propeller program – holds the value to display on the LEDs as an 8-bit binary number.

Step 4.

The flow chart in **Figure 4.5** shows how a simple 8-bit binary-counter program will work.

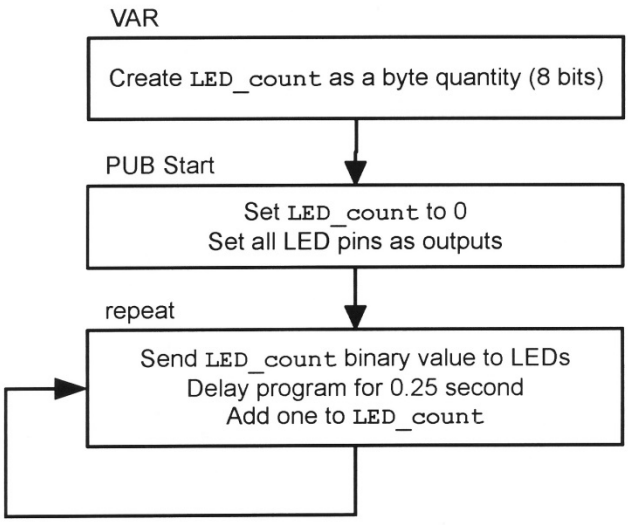


Figure 4.5.
Flow chart for a simple 8-bit counter program written in the Spin language.

Program 4.1 shows the Spin-language code that will display an incrementing 8-bit count. First, the program defines the byte variable, `LED_count`, so we have an eight-bit value for the LEDs to display. Next, the program defines a public object named Start. (The Propeller Spin language makes good use of objects that other software can use as needed.) The object portion of the program sets the `LED_count` value to zero and sets all eight of the MCU's connections to the LEDs as outputs. The repeat operation creates a loop that sends the 8-bit count to the LEDs, waits a quarter of a second, and lastly adds 1 to the `LED_count` value. The ++

operation simply adds 1 to the named variable; here labeled `LED_count`. You could use instead:
`LED_count := LED_count + 1.`

Program 4.1: A simple 8-bit counter in software.

```
{
LED Experiments, 10-30-2014
Jon Titus
EX 4.1, Rev. 1
}

VAR
  byte LED_count                'Set up a count to display
                                'as an 8-bit binary value
PUB Start                       'Name this object: Start
  LED_count := 0                'begin count at 0
  DIRA[16..23] := %11111111     'all LED pins set as outputs
  repeat                        'create an endless loop
    OUTA[16..23] := LED_count   'output the count to the LEDs
    waitcnt(clkfreq/4 + cnt)    'wait 0.25 seconds
    LED_count++                 'increase count value by one
```

As used in **Program 4.1**, the `waitcnt` operation provides a delay of a quarter of a second (0.25 sec) so you can see the count as it changes on the LEDs. The Propeller software predefines `clkfreq` as a 1-second period. When added to the Propeller's internal clock value, `cnt`, at the time the `waitcnt` operation starts, the program will not proceed until the internal clock reaches a time of `cnt` plus one second. If you want a delay of, say 1/12th of a second, the following command will do so:

```
waitcnt(clkfreq/12 + cnt)
```

Step 5.

Type **Program 4.1** into the Propeller Tool or use the Propeller Tool's File menu to open the 4.1.spin program. Go to the Propeller Workspace folder and locate within it the Experiment 4 folder. Open that folder and select the 4.1.spin program. The screen image in **Figure 4.6** shows what you can expect to see for when you open the Propeller Workspace folder and subfolders.

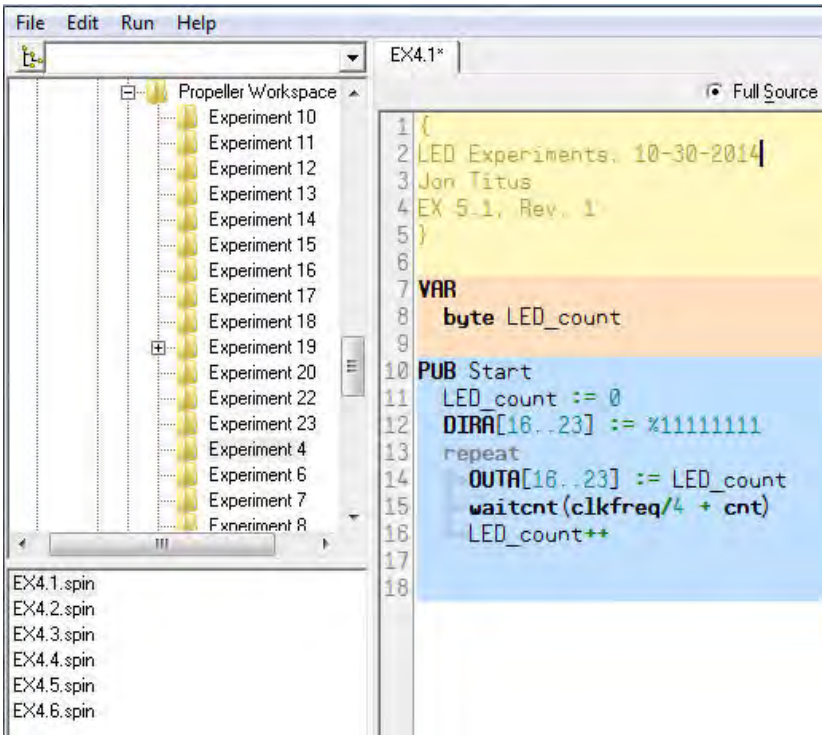


Figure 4.6.
A portion of the Propeller Tool screen shows the folder tree above the list of Spin programs in the Experiment 4 folder. The folders all belong in the Propeller Workspace folder.

Check your typing and the program listing. When you have corrected any errors – the provided software does not include any – go ahead and run the 4.1 program. Just press the F10 key and the code will get loaded into the Propeller's random-access memory and the MCU will run it. You should see the eight LEDs display an increasing binary count; 00000000, 00000001, 00000010, 00000011, and so on. What happened after the count reaches 11111111?

IMPORTANT: You may name objects and variables as you like, but you cannot use any of the Spin language reserved words such as `DIRA` or `OUTA`. If you inadvertently use a reserved word, when you try to run the program, an error message will appear (**Figure 4.7**). Choose a different object or variable name. When typing a program, pay careful attention to indentation and use of upper-case and lower-case characters. Improper typing can cause a program to fail to run. The extensive "Propeller Manual" provided for free on the Parallax Web site includes a list of the reserved words you may not use. Find the complete manual here: http://www.parallax.com/downloads/propeller-manual.

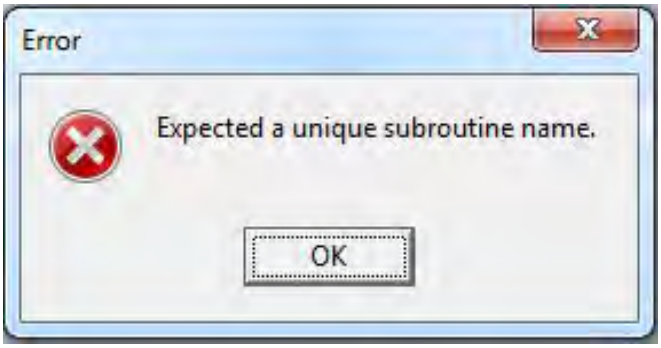


Figure 4.7.
This error message appears if you try to use a reserved word in a Propeller program.

Step 6.

You might remember how an inverter IC (Experiment 2, Figure 2.14) converted logic-1 signals to logic-0 signals, and vice versa. The inverter IC took outputs from a 74LS93 4-bit binary counter and changed their state so what appeared on the LEDs as a down counter (1000, 0111, 0110, and so on) became an up counter (0111, 1000, 1001, and so on). You could do something similar at the MCU outputs, but software can perform the inversion in one of two ways:

1. Instead of incrementing the count value with the instruction `LED_count++`, decrement the count with, `LED_count--`.

2. Perform a bitwise NOT operation (`!`) that reverses the state of each bit in a value. Thus, 11001010_2 would convert to 00110101_2.

   ```
   OUTA[16..23] := !LED_count
   ```

Make change #1 in the program and run it. What did you observe? Did the LEDs show a down count in which fewer and fewer of the LEDs remain on? Change **Program 4.1** to its initial state and try change #2. What happens now? Again you should see the count decrease. What would happen if you made both changes 1 and 2 above in your program?

You should observe a down count, whether you change the program to decrement the value of `LED_count`, or simply invert the state of the bits output to the LEDs. If you use both methods together, you will see the LEDs "count" up. Decrementing a value and inverting it has the same effect as incrementing the value.

A simple inversion of bits required no extra hardware, no added components, and no IC. And a change in software takes little time to enter and test. That's the beauty of using an MCU rather than adding hardware and complicating a circuit. I prefer the NOT operation (`!`) because it preserves the incrementing counter, which might get used elsewhere in a program for some other purpose. Later on I don't want to wonder, "Hmm, did I increment that value or decrement it?"

Step 7.

In this step you will turn on one LED and then "shift" it across all the LEDs. Of course the LED will not move, but the light will. What could you do to control the LEDs? A clear written statement of the problem and a similar description of a solution give people involved in a project a solid foundation to start with.

Here's my answer:

Problem: Turn on one LED and then "shift" it across the display.

Solution:

1. Set all LEDs to off.
2. Turn on 1st LED (at pin 16)
3. Wait so people can see the LED in this position.
4. Turn off 1st LED (at pin 16)
5. Turn on 2nd LED (at pin 17)
6. Wait so people can see the LED in this position.
7. Turn off 2nd LED (at pin 17)

   …and so on.

You could write a program to perform the steps just shown to turn each LED on, create a delay, and then turn the LED off. But steps 2, 3, and 4 resemble steps 5, 6, and 7 except for the pin number of LED turned on or off. Thus a program might use a repeat loop to combine these three steps and simply change the pin number of

the LED to control. The flow chart in **Figure 4.8** shows how the program would operate and **Program 4.2** shows the code for the process.

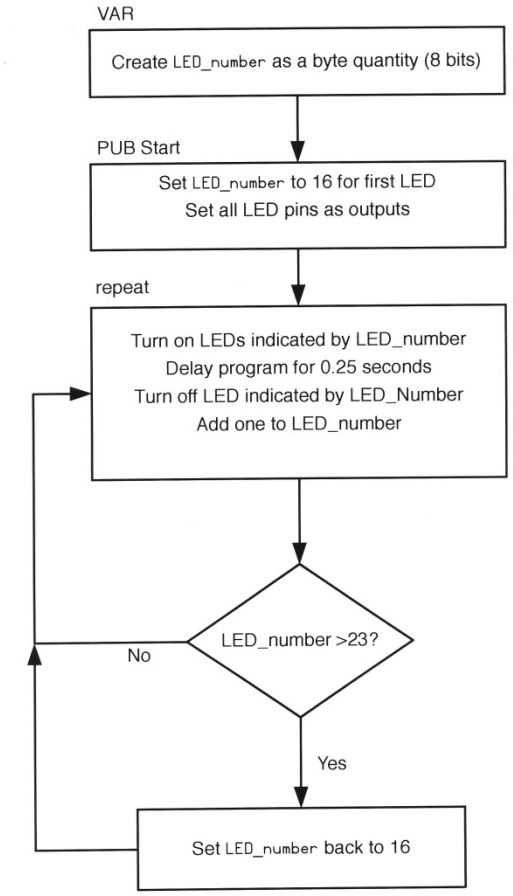


Figure 4.8.
Flow chart for a program to "move" a lit LED from one position to the next in one direction and then repeat the process. The LED_number variable will range from 16 through 23 to control the output at the I/O pin with the same number.

Go ahead and run **Program 4.2**. Does it operate as expected? It should.

Program 4.2.

```
{
LED Experiments, 10-30-2014
Jon Titus
EX 4.2, Rev. 1
}

VAR
  byte LED_number             'Set up a number to identify an LED

PUB Start                     'Name this object: Start
  LED_number := 16            'begin with 1st LED, P16
  DIRA[16..23] := %11111111   'all LED pins set as outputs
  repeat                      'create an endless loop
    OUTA[LED_number] := 1     'turn on LED at LED_number
    waitcnt(clkfreq/4 + cnt)  'wait 0.25 seconds
```

```
      OUTA[LED_number] := 0    'turn same LED off
      LED_number++
      if LED_number > 23       'ensure count doesn't go past
                               'last LED
      LED_number := 16         'if count > 23, reset count
                               'back to 16
```

The 4.2 program will shift a lit LED in the same direction again and again. The first part of this program looks familiar except for a new variable, `LED_number`, which begins with a value of 16 in the Start object. Within the repeat loop, instead of sending a varying value to the LEDs, the code uses `LED_number` to identify the LED to turn on and then off after a 1/4-second delay. After changing the state of the LED, the program increments the `LED_number` value to control the next LED in sequence. The `if` statement checks to ensure the `LED_number` doesn't exceed 23, the identifying value for the last LED, P23. Thus, when the `LED_number` value reaches 24, it gets reset immediately to 16 and the LED sequence starts again.

Could you write a program to shift the lit LED in the opposite direction once it reaches the LED at each end? Yes, but it takes a bit more code, as shown in **Program 4.3.**

Program 4.3.

```
{
LED Experiments, 10-30-2014
Jon Titus
EX 4.3, Rev. 1
Move lit LED back and forth
}

VAR
  byte LED_number                    'Set up a number to
                                     'identify an LED

PUB Start                            'Name this object: Start

  DIRA[16..23] := %11111111          'all LED pins set as
                                     'outputs
  repeat                             'create an endless loop

    repeat LED_number from 16 to 23  'loop for all 8 LED numbers
     OUTA[LED_number] := 1           'turn on identified LED
     waitcnt(clkfreq/4 + cnt)        'wait 0.25 seconds
     OUTA[LED_number] := 0           'turn off identified LED

    repeat LED_number from 22 to 17  'loop for 6 LED numbers
     OUTA[LED_number] := 1           'turn on identified LED
     waitcnt(clkfreq/4 + cnt)        'wait 0.25 seconds
     OUTA[LED_number] := 0           'turn off identified LED
```

Program 4.3 uses two short `repeat` loops; the first turns on LEDs at P16 through P23 in sequence as illustrated in **Program 4.2**. The second loop turns on the six LEDs at P22 through P17 in the opposite sequence. These two loops operate in an "outer" `repeat` loop that will run "forever." We call this type of "forever" loop an infinite loop. Load the program and run it.

Two important points to note here:

1. The two "inner" `repeat` statements now include the command `LED_number from x to y`. This information puts a condition on how many times the repeat loop will run. The first statement: `repeat LED_number from 16 to 23`, sets `LED_number` to the value 16 and runs through

the loop. Within the loop, the `LED_number` controls which LED gets turned on and off. At the end of the repeat loop, the MCU automatically increments `LED_number` by one and checks to determine whether it exceeds 23. If not, it runs the repeat loop again. When `LED_number` does exceed 23, the loop ends and operations move to the second repeat loop that turns the LED in the opposite direction. The capability to add conditions to a `repeat` loop gives you an easy way to limit how many times it will run.

2. The second "inner" loop operates only on LEDs at P22 through P17. Why doesn't it include the two "end" LEDs at P23 and P16? They get taken care of in the first loop and don't need to turn on and off a second time in the last repeat loop.

What would happen when you change the statement:

```
repeat LED_number from 22 to 17
```

to:

```
repeat LED_number from 23 to 16
```

Make the change and run the program. You should see the LEDs at P16 and P23 remain on twice as long as the other LEDs. Change the statement back to limit its range to 22 to 17 and run the program again. Did you see the difference? Subtle changes such as this can cause large programs to fail.

Step 8.

In **Program 4.3**, the MCU used a value to determine which LED to turn on and off. The MCU also could store patterns of on and off LEDs and then display them in sequence. Load and run **Program 4.4**.

Program 4.4.

```
{
LED Experiments, 10-30-2014
Jon Titus
EX 4.4, Rev. 1
Move two lit LEDs back and forth simultaneously
}

VAR
  byte Pattern_index     'Set up an index into LED-pattern array
  byte LED_pattern[8]    'Create array to hold LED patterns

PUB Start                                'Name this object: Start

  LED_pattern[0] := %00000000            'LED patterns
  LED_pattern[1] := %10000001            '1 = LED on, 0 = LED off
  LED_pattern[2] := %01000010
  LED_pattern[3] := %00100100
  LED_pattern[4] := %00011000
  LED_pattern[5] := %00100100
  LED_pattern[6] := %01000010
  LED_pattern[7] := %10000001

  DIRA[16..23] := %11111111              'all LED pins as outputs
  Pattern_index := 0                     'start at LED_pattern[0]
  repeat                                 'create an endless loop
    repeat Pattern_index from 0 to 7     'go through 8 patterns
```

```
      OUTA[16..23] := LED_pattern[Pattern_index]
      waitcnt(clkfreq/4 + cnt)          'wait 0.25 seconds
```

When you run **Program 4.4** the "bouncing" LEDs seem to either pass each other in the center, or they bounce off each other and ricochet back to their respective ends, depending on how you look at them. Patterns of LEDs can convey different types of information and prove useful in displays covered in other experiments.

You might notice that patterns 5 through 7 duplicated those from patterns 3 through 1. You could use only patterns 0 through 4 and then cycle through them backwards to "move" the LEDs in the opposite direction. If you don't want to do that, you could use statements:

```
LED_pattern[7] := LED_pattern[1]
LED_pattern[6] := LED_pattern[2]
LED_pattern[5] := LED_pattern[3]
```

to copy existing patterns rather than type them again – and perhaps make mistakes.

Step 9.

Use the framework of **Program 4.4** to create a bar-graph type display that starts with all LEDs off and starts to light LEDs one at a time from one end. After the graph has all LEDs on, the "graph" bar drops back so no LEDs are on. For a working program (4.5), go to the Answers section below. You can find **Program 4.5** in the Experiment 4 folder in the Propeller Workspace folder.

Answers

Experiment 4, Step 9: A Bar-graph Exercise.

Program 4.5 adds a second loop that decreases the number of lit LEDs in the graph bar until they all turn off. An outer infinite `repeat` loop causes the bar to continually go up and down. Patterns stored in an array cause LEDs to turn on or off.

OPTIONAL: The listing for **Program 4.6** illustrates the use of instructions that produce the increasing and decreasing bar. It does not rely on stored LED patterns. Here's a short explanation of how the software works:

The display starts with `LED_pattern = 0`, so all LEDs turn off. Within the second `repeat` loop, the statement:

```
LED_pattern := LED_pattern + (%00000001 << Pattern_index)
```

creates the increasing bar. First, the operation:

```
%00000001 << Pattern_index
```

takes the binary value 00000001 and shifts it to the left by the number of positions for the value of `Pattern_index`. Second, when `Pattern_index = 0`, no shifting occurs, so for the value of `LED_pattern = 0`:

```
   00000000  =  LED_pattern
+  00000001  =  %00000001 << 0
   00000001
```

The new `LED_pattern`, 00000001 gets sent to the LEDs. The next time through the loop:

```
   00000001  =  LED_pattern
```

```
+  00000010  =  %00000001 << 1
   00000011
```

The new `LED_pattern` now equals 00000011 and that value goes to the LEDs, which increases the displayed "bar" by one LED.

To decrease the bar, the process works in reverse and starts with the binary value 10000000, which gets shifted to the right and subtracted from the `LED_pattern` value to turn off LEDs from the top down.

Experiment No. 5 – LED Bar-Graph Driver ICs Supplement MCUs

Abstract

Although an MCU can create bar graphs by turning LEDs on or off, some equipment can take advantage of special integrated circuits that measure a voltage and control an LED bar-graph display. In this experiment you will learn how to use an LM3914 and an LM3915 integrated circuit. The LM3914 measures voltages on a linear scale – each LED represents an equal voltage increment from its nearest neighbors. The LM3915 uses "volume units" to create a VU display like the ones used in audio equipment to indicate sound output. A bit of simple algebra lets you calculate how to change the input-voltage range for the bar-graph circuits. In some situations, an MCU might not have enough extra output pins to create a bar graph, so you should know specialized ICs often can lend a hand. After this experiment we continue with MCU circuits and software.

Keywords

LM3914, LM3915, bar-graph, dot-graph, LED, driver, logarithm, decibel, linear, nonlinear, volume unit, VU meter

Requirements

(1) - 10-element bar-graph LED package *or* 10 small LEDs of one color, red or green
(1) - LM3914 Dot/Bar Display Driver, 18-pin DIP
(1) - LM3915 Dot/Bar Display Driver, 18-pin DIP
(1) - 220 ohm resistor, resistor, 1/4-watt, 5% (red-red-brown)
(1) - 470 ohm resistor, 1/4-watt, 5% (yellow-violet-brown)
(1) - 1000 ohm resistor, 1/4-watt, 5% (brown-black-red)
(1) - 1200 ohm resistor, 1/4-watt, 5% (brown-red-red)
(1) - 3300 ohm resistor, 1/4-watt, 5% (orange-orange-red)
(1) - 10 kohm resistor, 1/4-watt, 5% (brown-black-orange)
(1) - 47 kohm resistor, 1/4-watt, 5%, (yellow-violet-orange)
(1) - 10 kohm variable resistor, trimmer, 1-turn
(1) - Solderless breadboard
(1) - Voltmeter or digital multimeter (DMM)
(1) - 9-volt DC power supply
(1) - small flat-blade screwdriver
(2) - Sheets of graph paper

Introduction

Electronic bar graphs display information such as the amount of battery capacity remaining in an electronic device, the level of sound in audio equipment, or the time until a heating-system filter needs replacement. In this experiment you will learn how to use two types of integrated circuits to measure a voltage and display the results on a 10-LED bar-graph module. If you do not have this type of LED module, 10 small LEDs will work just as well. You also will learn how the two bar-graph integrated circuits (ICs) work.

The LM3914 and LM3915 ICs rely on circuits that compare two voltages and indicate whether one voltage exceeds the other. The LM3914 silicon chip, for example, contains 10 comparators, each of which has a non-inverting input (+) and an inverting input (-), and a single output. **Figure 5.1** shows a typical comparator circuit with a fixed 2.5-volt signal on the comparator's non-inverting input (pin 3) and a variable resistor connected to the inverting input (pin 4). The two 10-kohm resistors "divide the +5V to create a fixed voltage of 2.5V.

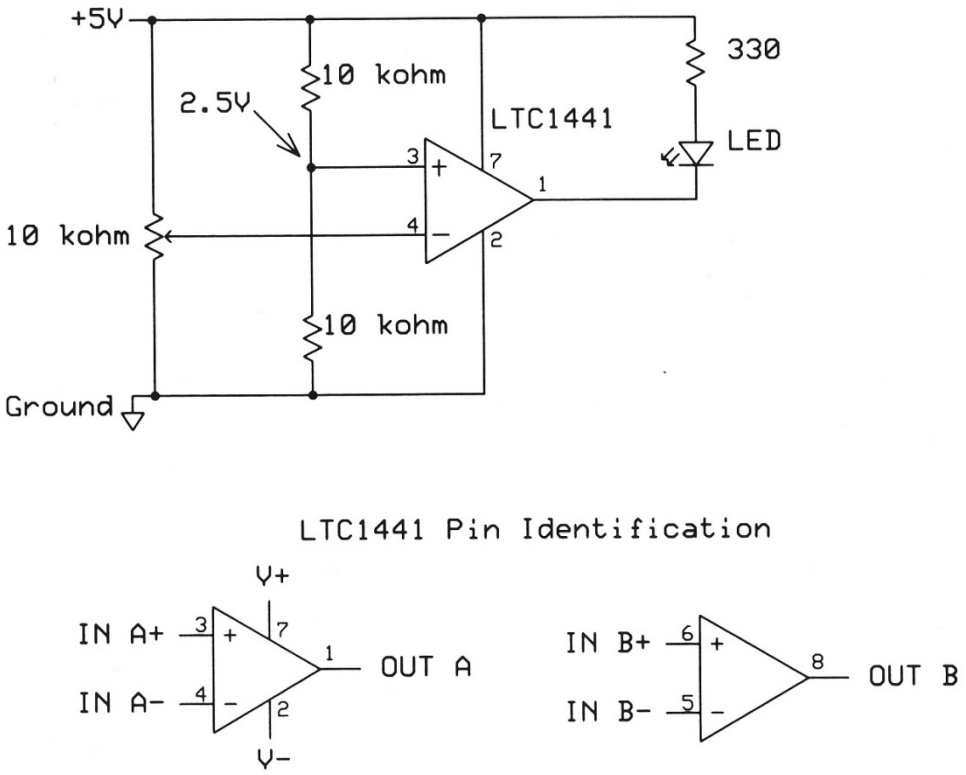


Figure 5.1.
A Linear Technology LTC1441 comparator with a constant-voltage input and a variable-voltage input. The output (OUT A) indicates whether the voltage on the non-inverting input (IN A+) exceeds that applied to the inverting input (IN A-), or whether the voltage on the inverting input exceeds the voltage on the non-inverting input. In this circuit, the LED turns on when the voltage at the IN A- pin exceeds the voltage applied to the IN A+ input. The LTC1441 includes a second, separate comparator, B.

If I turn the small control knob on the 10-kohm variable resistor (the arrow connection indicates a movable contact), the signal on the comparator's inverting input (IN A-) can vary between 0 and 5 volts. The timing diagram in **Figure 5.2** illustrates the state of the comparator output as the IN A- input voltage changes. In this circuit, the output changes from a logic-1 to a logic-0 when the varying voltage *exceeds* the fixed 2.5-volt signal. When the varying voltage *drops below* the fixed 2.5 volt input, the comparator's output changes from a logic-0 to a logic-1. I chose a 2.5-volt input for the comparator's IN A+ input simply to illustrate how a comparator operates. In other circuits, the voltage on this input might vary, too.

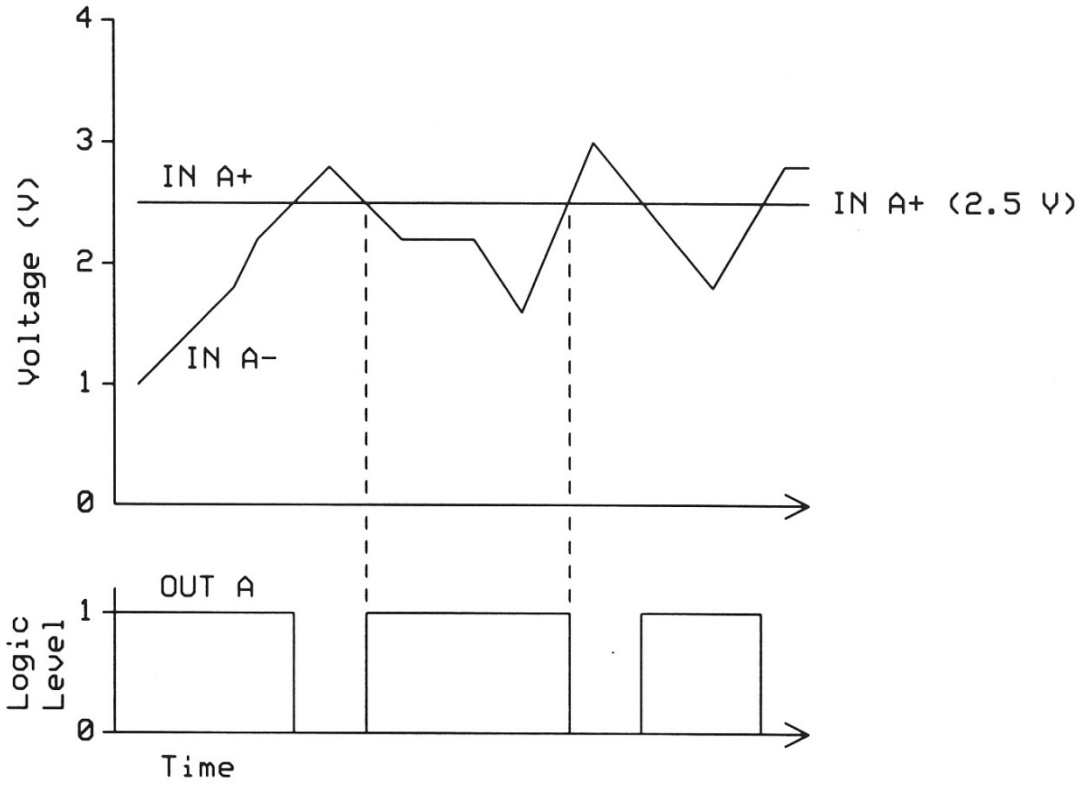


Figure 5.2.
Timing diagram for a comparator circuit with a constant voltage input (IN A+) and a varying-voltage input (IN A-). The lower section shows the logic state of the comparator output.

IMPORTANT: For the LTC1441 comparator, the voltage on the IN A+ input (pin 3) and on the IN A– input (pin 4) must stay between the most-negative power-supply voltage ,V– (pin 2) and the most positive voltage, V+, less 1.3 volts, or (V+ –1.3V). So for the circuit in **Figure 5.1**, the varying voltage should stay within the limits of ground and 3.7 volts. Other comparator ICs have similar limits on input voltages. Check their datasheet specifications and power-supply voltage limits before you put them in a circuit. You will see comparator circuits in a later experiment.

The LM3914 Bar-Graph Integrated Circuit

An LM3914 circuit applies a different fixed "reference" voltage to each of its 10 comparators' non-inverting input. The reference voltage for each comparator comes from a voltage-divider circuit that comprises 10 1000-ohm (1 kohm) resistors in series, as shown in **Figure 5.3.** The inverting input on all comparators connects to the unknown voltage signal (SIG IN). When this voltage exceeds the reference voltage for a given comparator, the associated LED turns on.

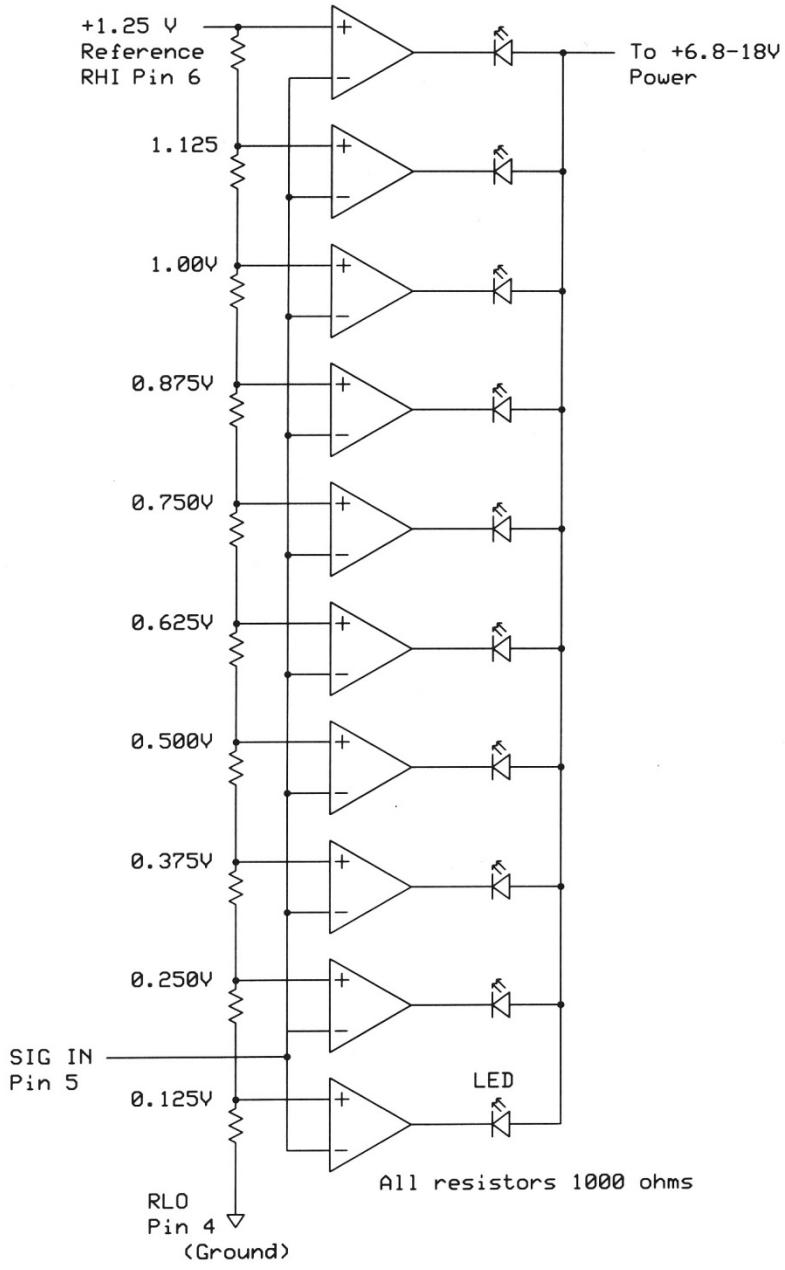


Figure 5.3.
A voltage-divider circuit of 10 resistors in series provides a different reference voltage for each comparator. In this diagram, the 10 resistors connect to ground at one end and to the LM3914 IC's internal 1.25-volt reference voltage at the other. The LM3914 provides equal voltage "steps" between adjacent comparators.

The LM3914 IC includes a 1.25-volt reference-voltage output you can connect to the IC's R_{HI} input (pin 6) at the "top" end of the voltage-divider resistor string. The "bottom" end of the resistor string connects to ground via the R_{LO} input (pin 4). The reference-voltage source also requires a connection between the REF ADJ pin (pin 8) and ground. In this configuration, the internal voltage-divider circuit provides voltage steps that change in increments of 0.125 volts – that is, the 1.25-volt reference divided by 10, the number of equal-value resistors. This arrangement creates a linear, or straight-line, relationship between the number of LEDs lit and the voltage measured.

Step 1.

In this step you will create a simple bar-graph display that uses an LM3914 dot/bar display driver IC and 10 LEDs. Because the LM3914 will regulate current through the LEDs, the circuit does not need a current-limiting resistor for each LED. If you do not have a 20-pin bar-graph LED package, use individual LEDs instead. **Figure 5.4** shows the complete circuit. Place the components in your solderless breadboard and connect them as shown. This circuit needs a 9 V power source. Recheck your wiring and turn on power.

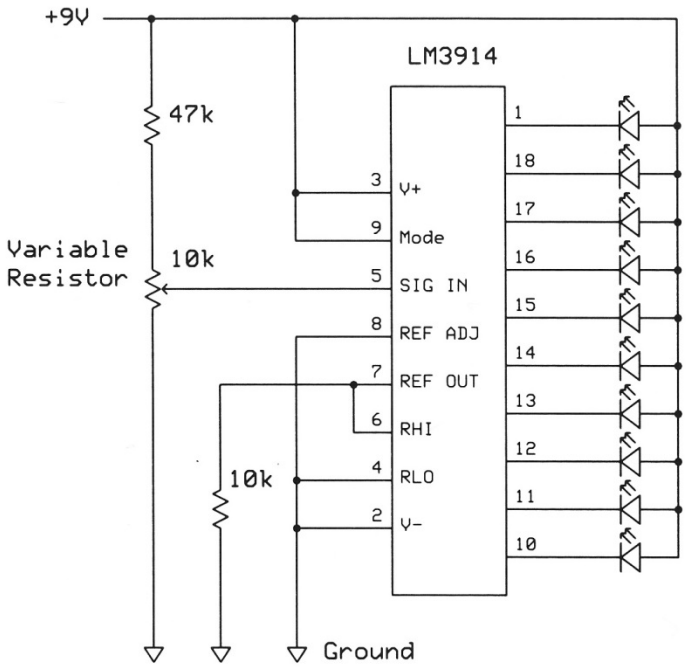


Figure 5.4
Schematic diagram for a bar-graph-display circuit controlled by an LM3914 dot/bar display-driver IC. This circuit has a measurement range from 0 to 1.25 volts. The 10 kohm variable resistor will produce a signal between 0 and about 1.6 volts for the LM3914 IC.

Use a small screwdriver to adjust the variable resistor and note any changes on the LEDs. Did you see the "bar" of LEDs go up and down (or back and forth) as you adjusted the variable resistor? If you did not see any LEDs turn on or off, recheck your wiring. Did you connect power to the LEDs? Did you connect the LED anodes (+) to the +9-volt power? Have you made all the ground connections?

If you want to see a single lit LED dot "move" instead of a bar, leave the MODE input (pin 9) disconnected, or "open circuit." This experiment describes display actions for a bar-type display, but you can use the dot mode if you choose. I prefer the bar mode because it gives me a quick indication of the "size" of the measurement, even at a distance.

The LM3914 IC uses an external resistance between pin 7 and ground to set the current that will flow through each LED, and thus LED brightness. In my experiments, I used a 10 kohm resistor as shown in **Figure 5.4**. Increase this resistance and the LED intensity decreases. Decrease the resistance and LED intensity increases.

Step 2.

As you changed the position of the variable-resistor control, did the LEDs turn on or off during a complete rotation of the control? You should see the number of LEDs increase or decrease for a bar display. If you use the dot mode, a lit LED will "move" back and forth from the top to the bottom of the LED array. If you did not see this result, recheck your wiring and ensure you have turned on power to your breadboard.

Step 3.

In this step, you will determine the voltage at which each LED turns on and off. Then you can plot the results to show how the LM3914 responds to voltage changes. Connect your voltmeter's positive lead to the output from the variable resistor or to pin 5 (SIG IN) on the LM3914 IC. Connect the voltmeter's negative lead to ground on the breadboard, as shown in **Figure 5.5**. This arrangement lets you measure the external voltage applied to the SIG IN pin. Set your meter for DC measurements on a scale that lets you measure no less than 2 volts full scale. I used the 2-volt scale on a Wavetek Meterman 25XT meter.

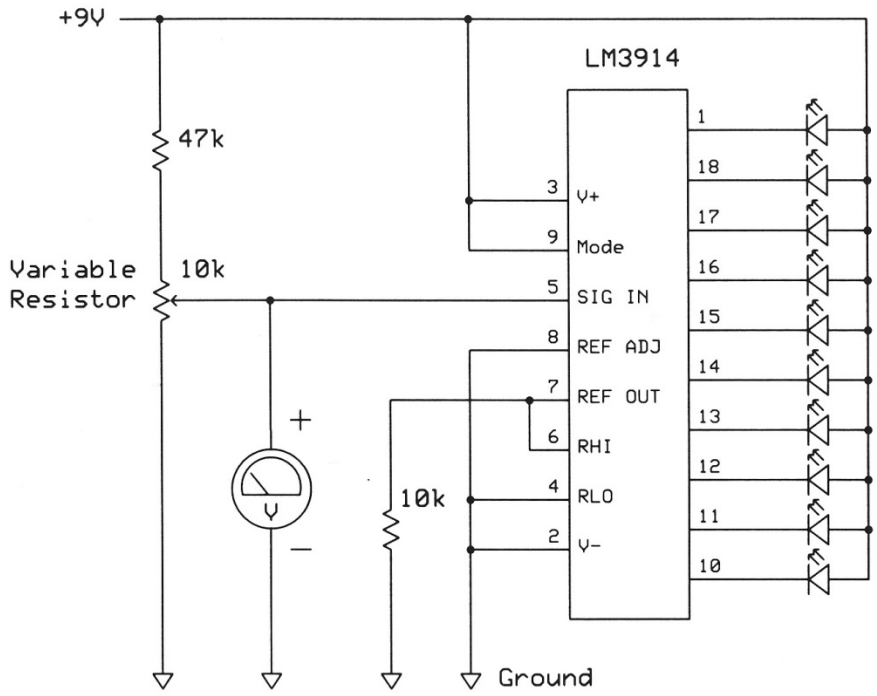


Figure 5.5.
A voltmeter between pin 5 on the LM3914 and ground lets you measure the voltage provided by the variable resistor.

Turn on your breadboard circuit and rotate the variable-resistor control so all LEDs have turned off and the control has reached the limit of its rotation.

Next, use the screwdriver to slowly adjust the variable resistor until the first LED turns on and record the turn-on voltage in **Table 5.1.** Then continue to slowly rotate the control and make the same measurement when each LED turns on. After you have all LEDs lit, reverse the process and decrease the voltage from the variable resistor and record the voltages at which each LED turns off. Use a piece of graph paper to plot those values. Place the LED number (1, 2, and so on) on the x-axis, and use the y-axis to represent voltage, from 0 to 1.5 volts.. Does your graph give you a somewhat straight line for the turn-on and turn-off voltages? It should. You may turn off power to your breadboard.

Table 5.1. Measurement Results for LED On-Off Tests with an LM3914.

LED Number	Turn-On Voltage	Turn-Off Voltage
1		
2		
3		
4		
5		

6		
7		
8		
9		
10		

Figure 5.6 shows the results I obtained. Another test with a 10-turn variable resistor gave me finer control over the voltage applied to the LM3914 IC. **Figure 5.7** shows those results and better illustrates the linear, or straight-line, relationship between the input voltage and the on-off behavior of the LEDs.

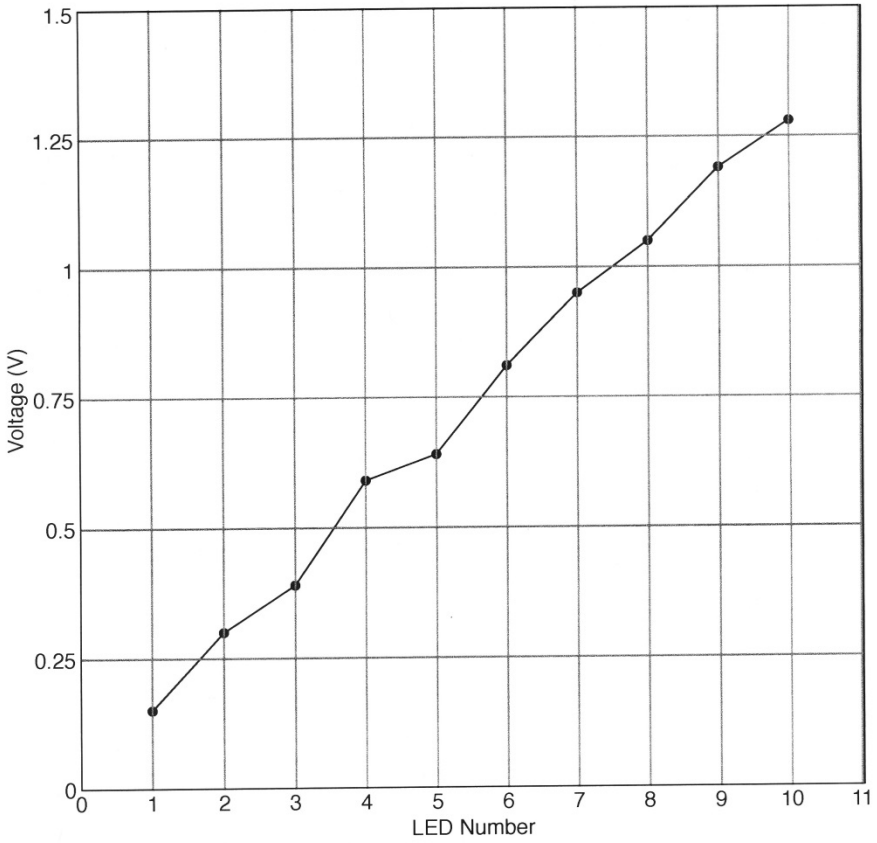


Figure 5.6.
A plot of voltages supplied by a small single-turn 10-kohm variable resistor as each LED turned on or off.

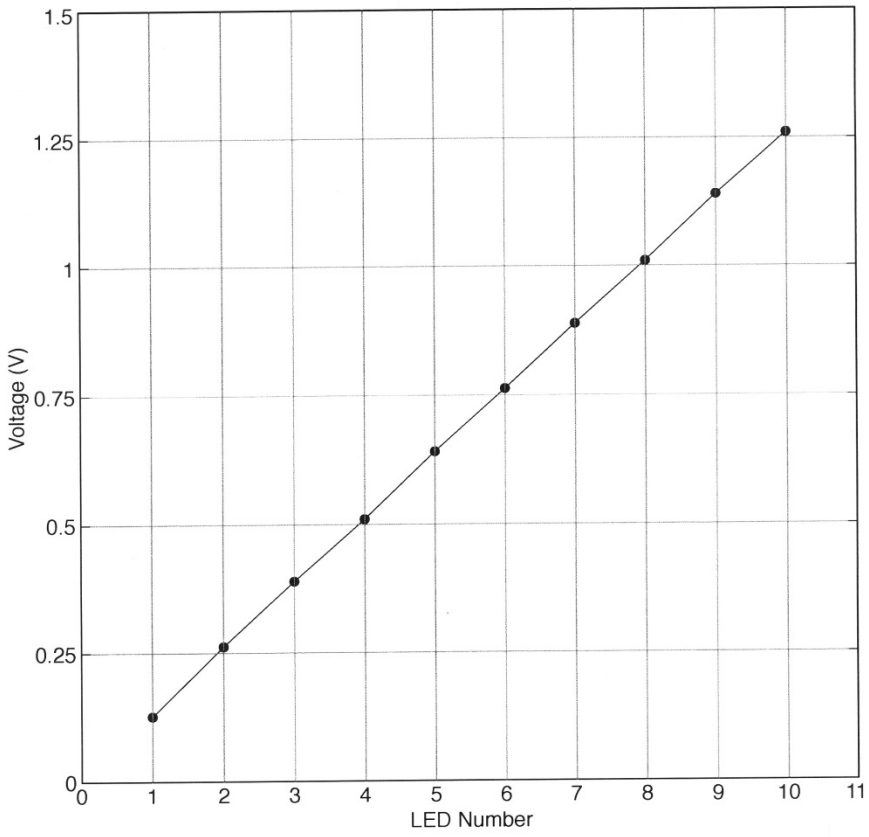


Figure 5.7.
A plot of voltage vs. LED number when a 10-turn 10-kohm variable resistor provided a voltage to the LM3914 IC.

Step 4.

The circuit you used in the previous step will turn bar-graph LEDs on or off over a small range: 0 to +1.25 volts. The Texas Instruments data sheet for the LM3914 IC suggests the circuit shown in **Figure 5.8** as a 0 to +5-volt bar-graph meter. You may construct this circuit and try it. I did not have a 1210-ohm or a 3830-ohm resistor in my lab, so I connected in series a 3300- and a 470-ohm resistor to get a 3770-ohm resistance. And I created another series connection with a 1000- and a 220-ohm resistor to give me a 1220-ohm resistance. When I tested my circuit, the top LED turned on for a 5.4-volt signal at the SIG IN pin. So, the bar-graph display had an accuracy of about 10 percent with my resistor substitutions.

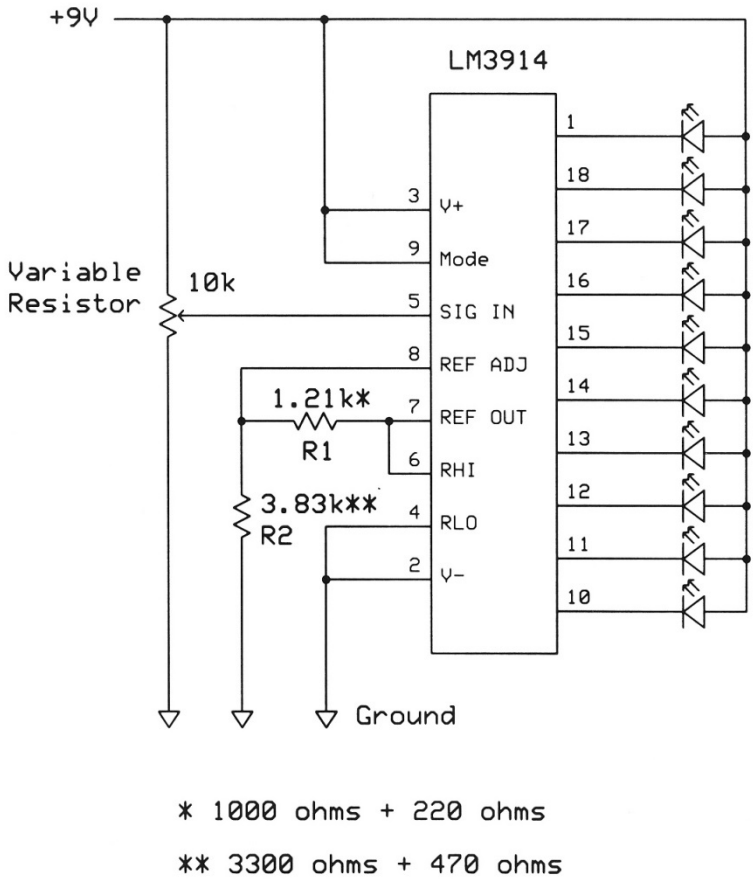


Figure 5.8.
Texas Instruments LM3914 bar-graph circuit for a 0-to-5-volt input signal.

If you need an LM3914 IC to accept an input (SIG IN) that covers a different voltage range, say 0 to 7 volts, you can use a formula from the data sheet to calculate the resistances you need for a circuit of the type shown in **Figure 5.8**:

$$V_{OUT} = V_{REF} * [1 + (R2 / R1)]$$

The V_{REF} in my circuit measured 1.28 volts between pins 7 and 8 on the LM3914:

$$7.0\,V = 1.28\,V * [1 + (R2 / R1)]$$

Algebra lets us rearrange this equation and solve for the ratio of R2 / R1:

$$(7.0\,V / 1.28\,V) - 1 = R2 / R1$$

or, $R2 / R1 = 4.5$

Now you can choose resistances with a ratio of 4.5-to-1 for R2/R1. In this case, you could choose 1000 ohms for R1 and 4500 ohms for R2. You won't find a 4500-ohm standard resistor, so connect a 3300- and a 1200-ohm resistor in series to produce a 4500-ohm resistance. I used R1 = 1000 ohms and R2 = 4500 ohms in the circuit shown earlier in **Figure 5.8**. The top LED turned on for an input of 7.5 volts.

IMPORTANT: An LM3914 bar-graph IC cannot produce a reference voltage greater than its own supply voltage. So, you could not set a reference of 15 volts and use only 9 volts to power the LM3914. Also, this IC requires a supply voltage of 6.8 to 18 volts. If you use it with, say, a 5-volt power supply you will get incorrect results.

Step 5.

In this step you will learn another way to use an LM3914 bar-graph circuit and a voltage divider to measure voltages.You take the basic 0-to-1.25-volt measurement circuit shown in Figure 5.4 and add two resistors to let the LM3914 IC light all 10 LEDs for an X-volt input and turn off for a 0-volt signal as long as X exceeds 1.25 volts. **Figure 5.9** shows a voltage divider circuit that comprises resistors R1 and R2. The resistors takes a 0-to-9-volt signal from the 10-kohm variable resistor and divide it so the LM3914 input received only a 0-to-1.25-volt input. (Keep in mind that voltage-divider circuits do not perform a mathematical division, they simply separate a given voltage into one or more parts.)

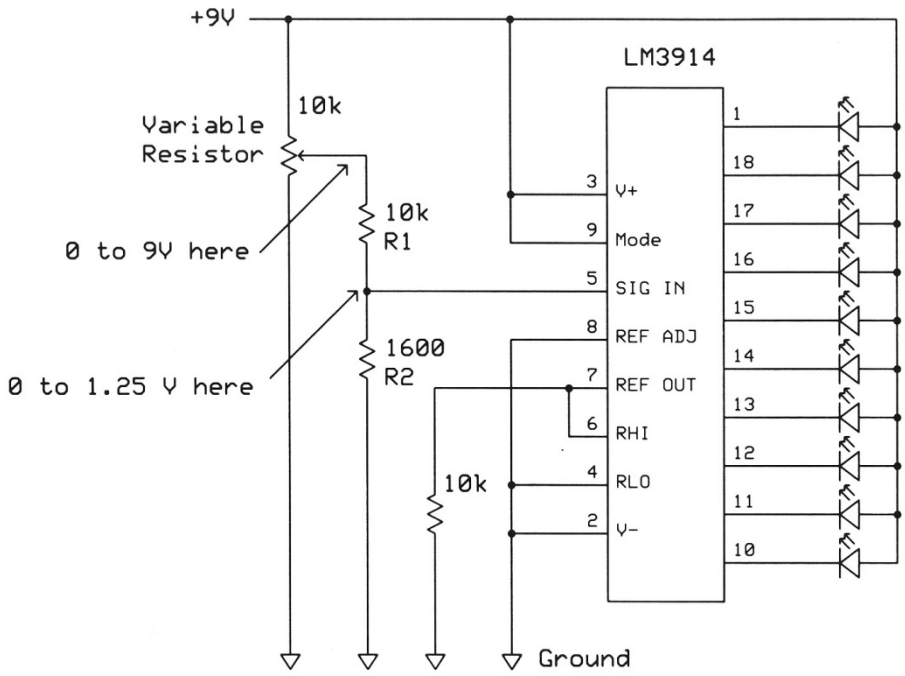


Figure 5.9.
This voltage-divider circuit (R1 and R2) produces a 0-to-1.25-volt output for a 0-to-9-volt input from the variable resistor.

Suppose you want to measure the voltage of a vehicle battery (14 volts maximum) with an LM3914 set up to operate from a 9-volt battery. In Step 4 you learned how two external resistors changed the LM3914 reference voltage so you could measure a 0-to-5-volt signal (see **Figure 5.8**). In that example the IC's reference voltage (REF OUT) had a lower voltage than the power supply for the circuit. The 14-volt signal from a vehicle battery exceeds the 9-volt power-supply voltage, so you cannot adjust ratios of R1 and R2 to create a 14-volt reference. You can't get 14 volts out of a 9-volt battery. Instead, you rely on a voltage divider. The vehicle-battery will produce a 0-to-14-volt output, but you need a 0-to-1.25-volt output. (I'll use an input voltage of 1.2 volts for calculations.) You can calculate the resistors values needed.

Figure 5.10 shows two resistors in a voltage-divider circuit and two voltmeters. From that diagram you know:

1. V4 for the LM3914 must equal 1.2 volts for a 14-volt input. For accurate measurements, the input to the LM3914 should not exceed 1.25 volts.

2. When V4 equals 1.2 volts, V3 must equal 12.8 volts, or 14V – 1.2V.

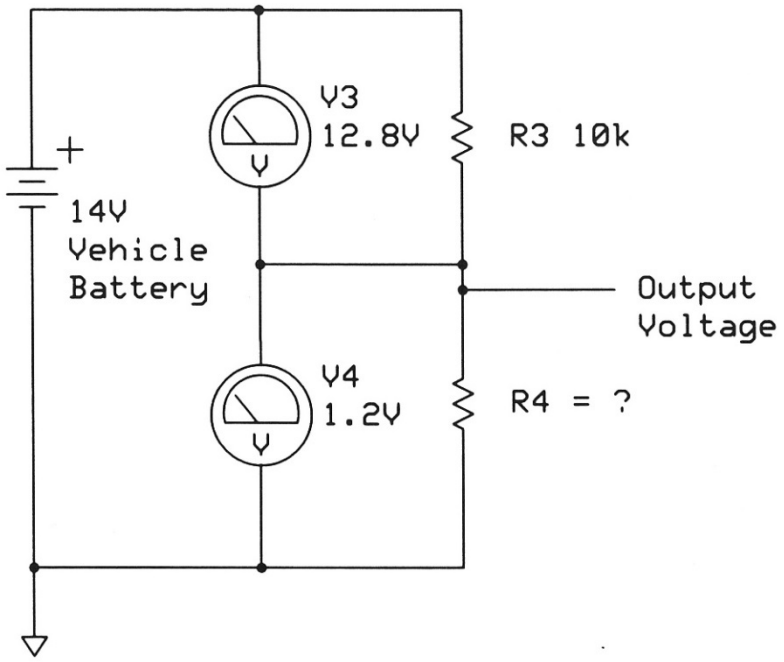


Figure 5.10.
The sum of voltages across individual components in series must equal the voltage applied across all those components. With the information shown in this diagram, you can calculate the needed value for resistor R4.

Now you need the values for R3 and R4. Based on experience, I start with a 10-kohm resistance for R3 and calculate the value for R4. By definition, the same amount of current must flow through R4 as through R3. (The LM3914 SIG IN requires an insignificant amount of current so we disregard it.) Ohm's Law provides the relationship: current (I) equals the voltage (E) across a resistor divided by the resistance (R), or $I = E/R$. For clarity I'll substitute the letter V for E in the equations that follow. Given equal currents through R3 and R4:

$I = V3 / R3 = V4 / R4$, so we can eliminate the current variable I. Then we have:

$V3 / R3 = V4 / R4$, which inverts to: $R3 / V3 = R4 / V4$.

Substitute the known values of V3 and V4, and 10,000 ohms for R3. Then calculate:

$R3 / V3 = R4 / V4$ or $10{,}000\ ohms / 12.8\ volts = R4 / 1.2\ volts$

Use algebra to rearrange this equation:

$(10{,}000\ ohms\ x\ 1.2\ volts) / 12.8\ volts = R4$

$R4 = 938\ ohms$ or 910 ohms + 30 ohms from my resistor bins.

The voltage-divider circuit shown **Figure 5.11** accepts a 0-to-14-volt signal and produces a 0-to-1.2-volt signal for the LM3914 SIG IN pin.

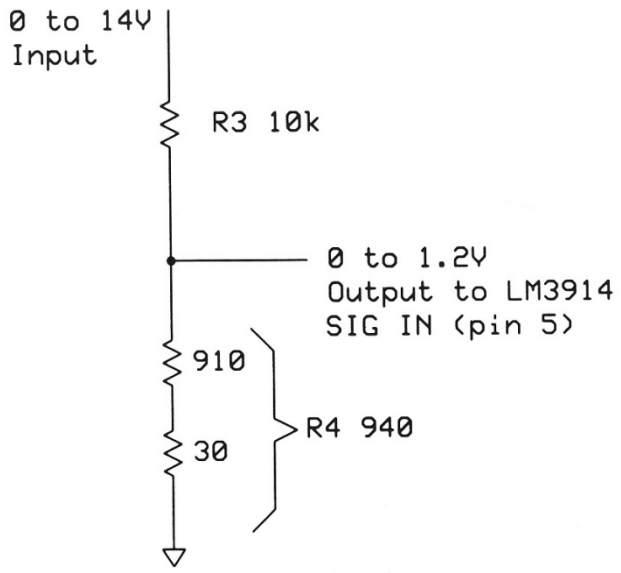


Figure 5.11.
This voltage-divider circuit comprises two resistances, R3 and R4, and simplifies circuit design when you must reduce the span of a voltage range. The input voltage must exceed the output voltage.

As a final calculation, find the current the divider circuit will draw from the 14-volt battery. Again, Ohm's Law provides the solution:

$$I = E / R \text{ or } I = 14\,volts\ /\ 10{,}940\,ohms = 0.0013\,amperes$$

or 1.3 milliamperes. If the vehicle battery can provide this small current without discharging over a long time, the divider will serve its purpose. Remember, though, the LEDs will draw much more current than the voltage divider. I measured 9 mA with all LEDs off, and 115 mA with all LEDs lit. In some circuits you might need to save power. A pushbutton could turn on the LEDs and LM3914 only as needed.

The circuit shown in **Figure 5.11** will work well. But because resistors can have a resistance in the range of ±5-percent from their marked value, you might need to slightly "tweak" or "trim" the resistance values to get the expected output. In a practical voltage divider I suggest you included a small variable resistor, or trimmer resistor, to allow minor adjustments that ensure an X-volt input produces a 1.2-volt input for the LM3914 IC. The resistance in my new battery-monitor circuit (**Figure 5.12a**) still equals about 11,000 ohms, but I used an 9500-ohm fixed resistor in series with a 1000-ohm 10-turn trimmer resistor and a 500-ohm fixed resistor. For a 14-volt input signal, at one limit the trimmer puts out about 1.9 volts and at the other limit, it puts out 0.6 volts. So in between the trimmer's limits, you have a point at which the output equals 1.2 volts. You can measure the output voltage as you adjust the trimmer. When the output gets to 1.2 volts, stop adjustments.

In this example I used a 1000-ohm trimmer resistor. So, the new R3 equals the old value, 10 kohms minus *half the trimmer's resistance* (500 ohms). So, for R3 10,000 ohms - 500 ohms, or 9500 ohms. Likewise, for R4, take the old R4 value (1000 ohms) and subtract 500 ohms from its value, too: 1000 ohms minus 500 ohms, so R4 equals 500 ohms.

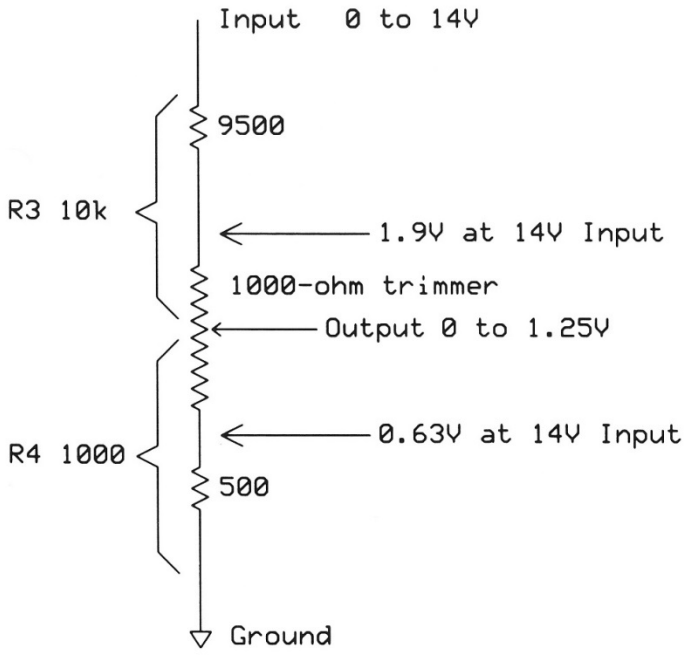


Figure 5.12a.
This voltage-divider circuit includes a small variable resistor called a trimmer so you can make a fine adjustment to ensure a 1.2-volt output when the circuit has a 14-volt input.

Step 5.5 - *Optional*.

How do you calculate a trimmer resistance? (If you do not want to go through this step, please skip ahead to Step 6.) Circuit designers have a small number of 10-turn trimmer resistors to choose from. They come in a 1, 2, 5 sequence; 100, 200, 500, 1000, 2000, 5000 ohms, and so on. You can't buy a 10-turn 300-ohm trimmer, for example. Instead a circuit designer would work with a 200- or 500-ohm trimmer. The trimmer has a resistive material and a wiper that moves back and forth as you turn a small screw head. Ten turns of the screw moves the wiper from one end of the resistive material to the other. The wiper provides the variable-resistance output.

Suppose you have calculated values for a voltage divider so R3 equals 10 kohms and R4 equals 3300 ohms (**Figure 5.12b**). The junction of these two resistances provides the output signal you need. You decide to add a trimmer resistor so you can get as close as possible to a 1.2-volt output for an LM3914 IC. In this case, a 2000-ohm trimmer represents a good choice. The smaller the trimmer resistance, the finer you can trim the output voltage. But the voltage trimming range decreases, too.

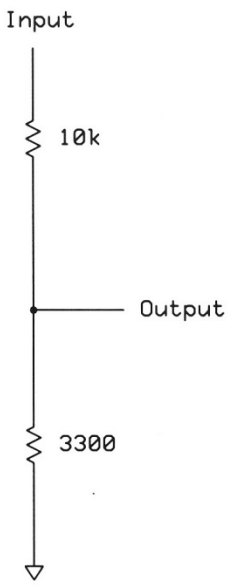


Figure 5.12b.
This circuit shows the fixed resistances of a voltage divider that could benefit from an added trimmer resistor. The trimmer would let you carefully adjust the output to overcome slight differences in actual component values from the values marked on them.

For R3, take the 10 kohm value and subtract *half* the resistance of the trimmer:

$$10{,}000\ ohms\ -\ 1000\ ohms\ =\ 9000\ ohms\ (R3)$$

Likewise, take the R4 resistance, 3300 ohms and subtract *half* of the trimmer resistance from it:

$$3300\ ohms\ -\ 1000\ ohms\ =\ 2300\ ohms\ (R4)$$

You can buy 9100-, 2200-, and 2400-ohm resistors with a ±5% tolerance, so you can create the circuit shown in **Figure 5.12c**.

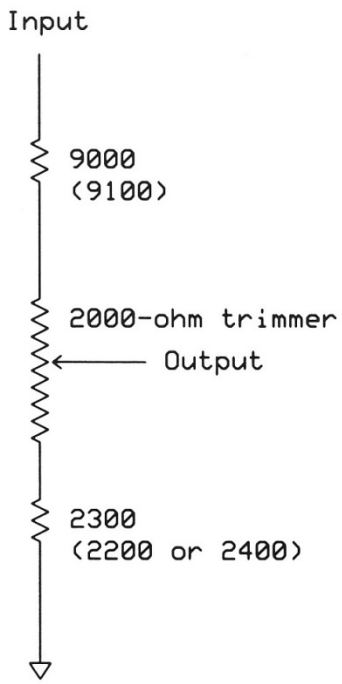


Figure 5.12c.
Schematic diagram equivalent of the circuit shown in **Figure 5.12b** but with an added 2000-ohm trimmer. This trimmer lets you easily adjust the output to the required voltage. Part values in parentheses represent readily available resistances.

A 2200- or 2400-ohm resistor will work well. Remember, resistor values can vary by ±5 percent from their marked value, so you get close enough to 2300 ohms for a practical circuit. The trimmer still has a wide enough voltage across it to provide the proper output after someone adjusts it. As an exercise, calculate the voltages at each end of the trimmer for a 5-volt input. Find the answers at the end of this experiment.

Non-Linear Measurements

Step 6.

Because the LM3914 IC has a linear response to a voltage input, it does not fill every need for a bar-graph display. Integrated-circuit manufacturers created a second bar-graph IC, the LM3915 that responds in a non-linear way. The LM3915 has the same pin configuration as the LM3914, so to experiment with it you can use the same circuits created for the LM3914.

You might wonder, "Why do we need nonlinear measurements?" They express *orders of magnitude* that cover a wide range of values, say 0.001 to 100 volts. That range covers five decades: 0.001 to 0.01 volts, 0.01 to 0.1 volts, 0.1 to 1.0 volts, and so on. Each decade covers a *multiple* of 10. **Table 5.2** lists some data from an experiment that did not involve LEDs or the LM3914 bar-graph driver IC. A plot of the data on a linear scale (**Figure 5.13**) doesn't tell us much about the measurements 1 through 6 that get "squashed" together at the bottom, though.

Table 5-2. Typical measurement values that span four orders of magnitude.

Measurement Number	Measured Value
1	0.003
2	0.009

3	0.027
4	0.783
5	0.981
6	1.03
7	2.9
8	3.17
9	7.65
10	9.09

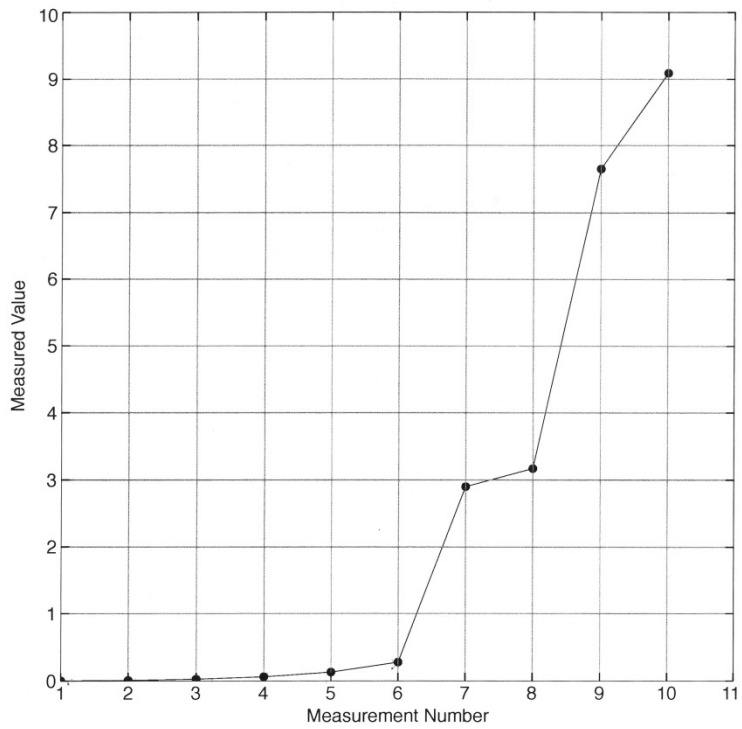


Figure 5.13.
A linear plot of values across four orders of magnitude. This type of plot gives a distorted view of data.

What happens when I plot of these values on a *logarithmic* scale that provides several decades on the y axis? **Figure 5.14** shows the smaller values become as easy to see as the larger values. The Richter scale, used to measure earthquake magnitudes, relies on a logarithmic scale from 2 to 10. Thus a magnitude-5 earthquake has 10 times the energy as one with magnitude 4. We also use logarithms to express the acidity or basicity of a chemical solution on the pH scale. An acid with a pH = 1 is 10 times stronger than an acid with pH = 2. The pH scale runs from 1 (most acidic) to 14 (most basic) and covers 13 orders of magnitude, or a span of 1 to 10 million million (that's not a typographical error).

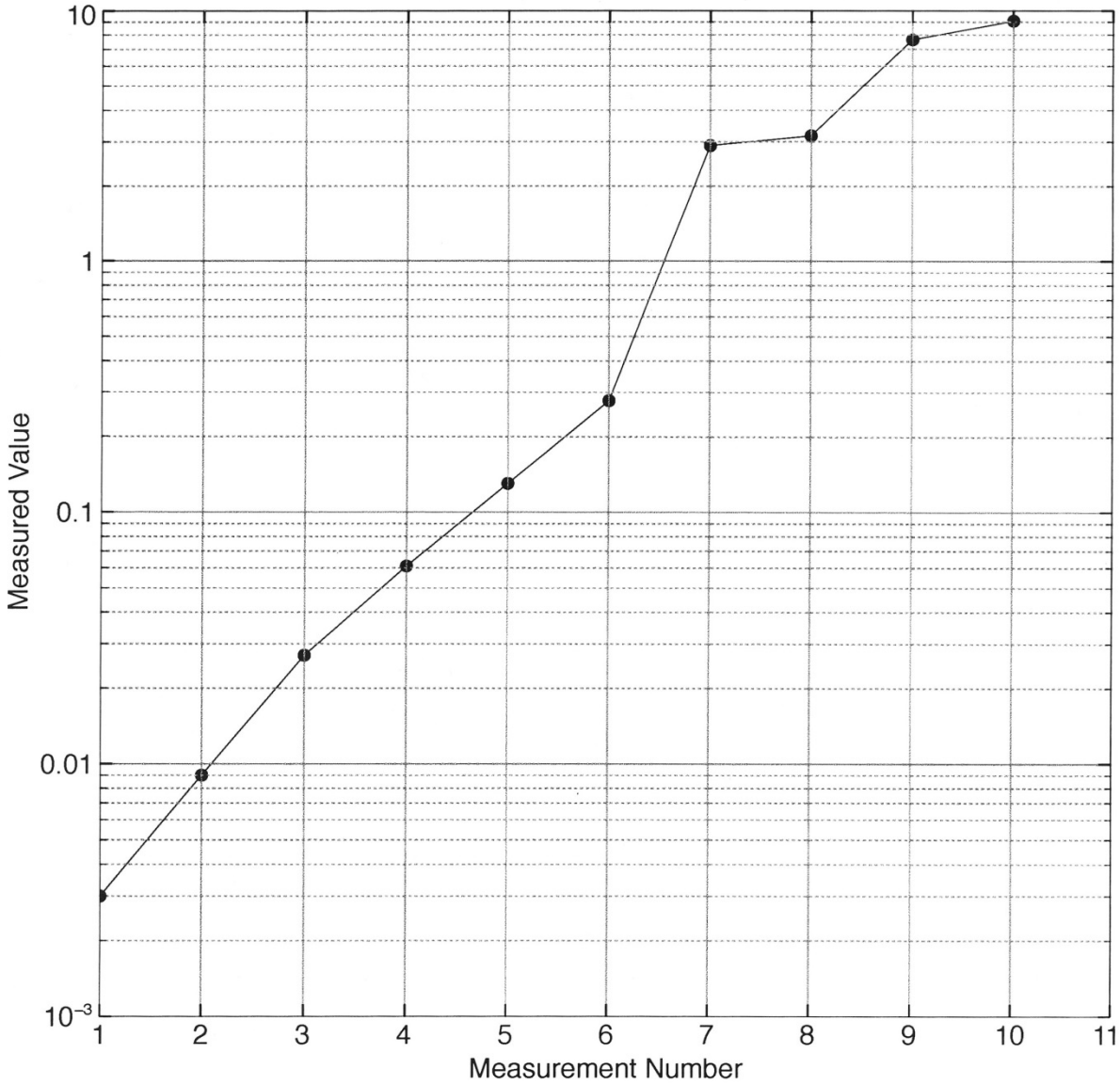


Figure 5.14.
People can better interpret values over several decade multiples, or orders of magnitude, when they use a logarithmic y axis. A linear plot of this data appears in **Figure 5.13**. My seventh measurement could result from an error on my part.

Engineers frequently use logarithmic information to graph and visualize data that has a large *dynamic range*, say from 1 to 100,000. Our ears respond to sounds across a large dynamic range that goes from an almost silent rustling of leaves in a light breeze to very loud noises from a nearby concert or fireworks display. Along with showing values on a logarithmic chart such as that in **Figure 5.14**, engineers employ units of decibels (dB), a ratio of two signal amplitudes that includes a logarithmic conversion. (If the next section seems too complicated, feel free to skip to Step 7.)

Suppose you have an audio amplifier. Without any input signals from a microphone or guitar, this amplifier produces a 1-millivolt (1 mV, or 0.001 volt) noise signal. (No electrical signal exists without some small amount of noise superimposed on it.) Engineers call this inherent noise from electrical components a *noise floor*. We cannot eliminate it. Assume the maximum output from the amplifier measures 6 volts. This amplifier

has a dynamic range of 0.001 to 6 volts. The following formula gives the dynamic range for the amplifier in units of decibels (dB) for the voltage measurements:

$$dB \;=\; 20 * log_{10}\,(V_{MAX} / V_{MIN}) \;=\; 20 * log_{10}\,(6\,volts / 0.001\,volts)$$

$$dB \;=\; 20 * log_{10}\,(6000) \;=\; 75.6\,dB\;dynamic\;range\;(voltage)$$

Engineers also call this value the *signal-to-noise ratio* for a device or circuit.

Note: Use the "log" key on a calculator to find the logarithm of a value. You cannot take the logarithm or 0 or of a negative number. If you have a logarithmic value and need to convert it to the original value, use the calculator's "10^X" key. (For more information about logarithms, see the References section at the end of this experiment.)

The bel (B) unit of magnitude honors Alexander Graham Bell who invented the telephone. The bel proved too large a measure of sound-pressure ratios, so engineers adopted the decibel (dB), or tenth of a bel, which we commonly use today to express a logarithmic ratio of values. For electrical circuits in which we measure voltages or currents, use the equation:

$$voltage\;gain_{dB} \;=\; 20\,log_{10} * (V1 / V2)$$

For power measurements we usually assume power approximates the square of a signal's amplitude. Thus the decibel equation becomes:

$$power\;gain_{dB} \;=\; 10\,log_{10} * (P1 / P2)$$

These two equations provide a decibel value of gain when V1 > V2, when P1 < P2, the $gain_{dB}$ becomes a negative number to indicates a power or voltage loss. Always specify whether you use voltage or power for your calculations because decibels represent a unit-less ratio.

Step 7.

Turn off power to your breadboard and replace the LM3914 IC in your breadboard circuit with an LM3915. This step uses the circuit shown earlier in **Figure 5.5**, and shown here in **Figure 5.15**. If you do not have this circuit in your breadboard, please construct it now. Recheck your connections.

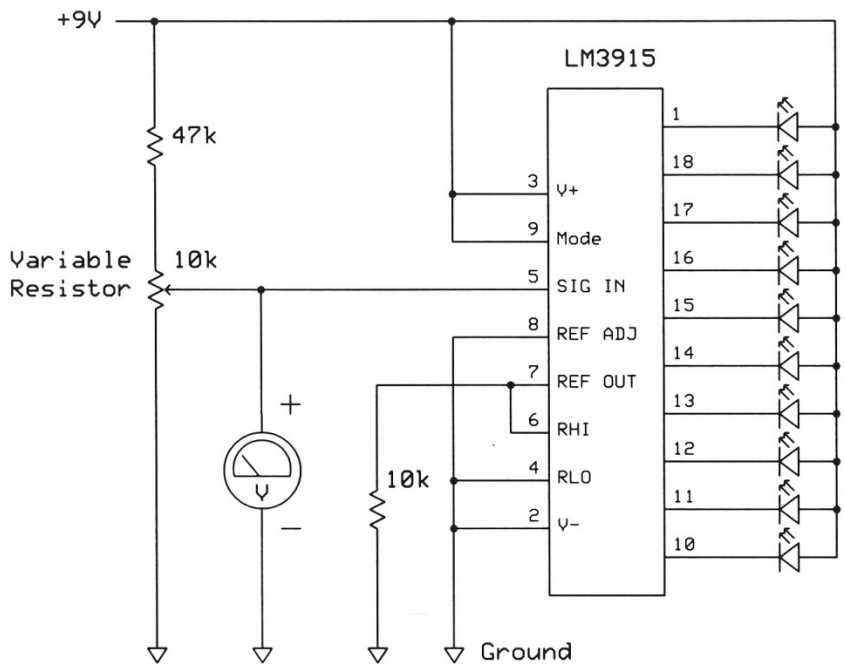


Figure 5.15.
Circuit used to measure the response of the LM3915 IC to a varying voltage between 0 and 1.25 volts.

Turn on power and note in **Table 5.3** the voltages at which each LED turns on as you change the trimmer-resistor setting. When you start with all LEDs off, it takes only a slight movement of the trimmer control to cause LEDs to turn on. As more LEDs light, though, it takes more turning of the control to make the last few LEDs turn on.

Table 5.3. Measurement Results for LED On-Off Tests with an LM3915.

LED Number	Turn-On Voltage
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Did you notice anything unusual about these measurements? Compare these voltages with those you wrote in **Table 5.1**. Values for the LM3915 IC show an increasing difference between the voltage steps as the voltage increases.

Use a piece of graph paper to plot the turn-on voltage for each LED, 1 through 10. The diagram in **Figure 5.16** shows my results for a large 10-turn variable resistor that gave me precise control over voltages supplied to the

LM3915 IC. Note how the plot's upward curve gets steeper for the higher-numbered LEDs. You can see a nonlinear relationship here: The points do not create a straight line.

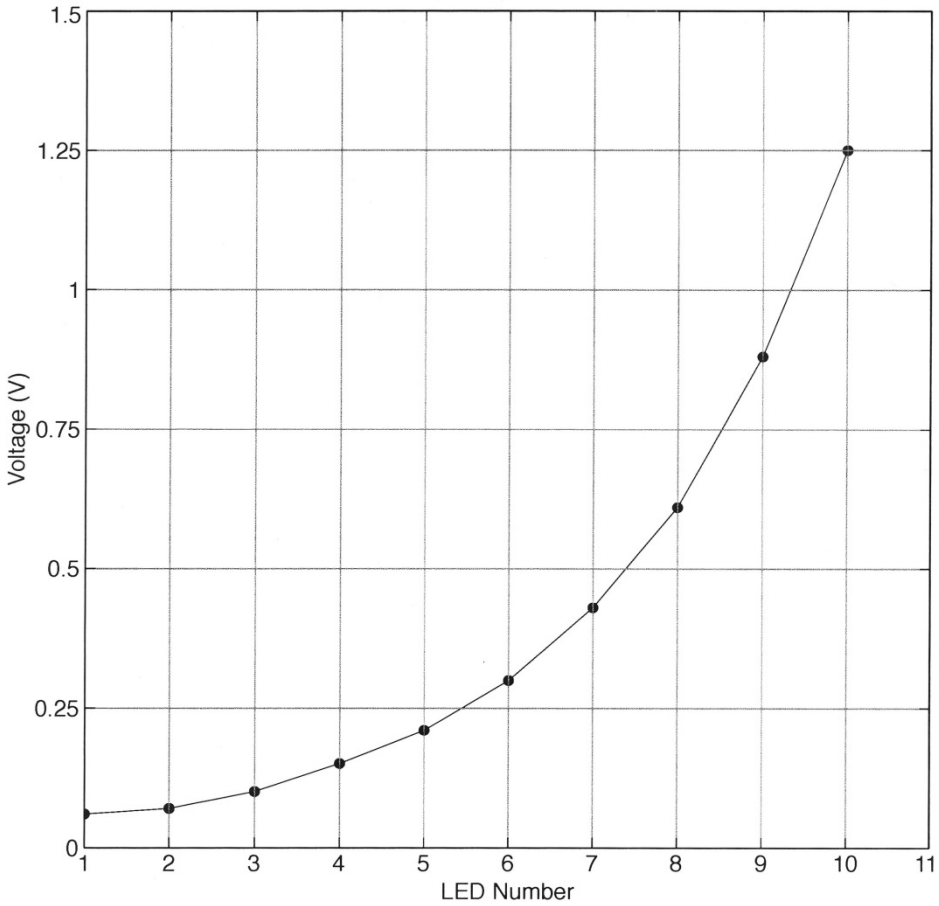


Figure 5.16.
This plot of voltages shows the nonlinear LED-turn-on values I measured for the LM3915 IC.

By changing the y axis from a linear scale to a *logarithmic scale* you can better see how an LM3915 IC turns on most of the LEDs to indicate the lower voltages (**Figure 5.17**). Eight LEDs turn on between 0 and 0.61 volts; about the halfway point to the 1.25-volts maximum. Only two LEDs turn on from 0.61 to 1.25 volts. On the logarithmic graph, each step from LED to LED approximates a 6-dB change, or a doubling of the voltage difference between the "steps."

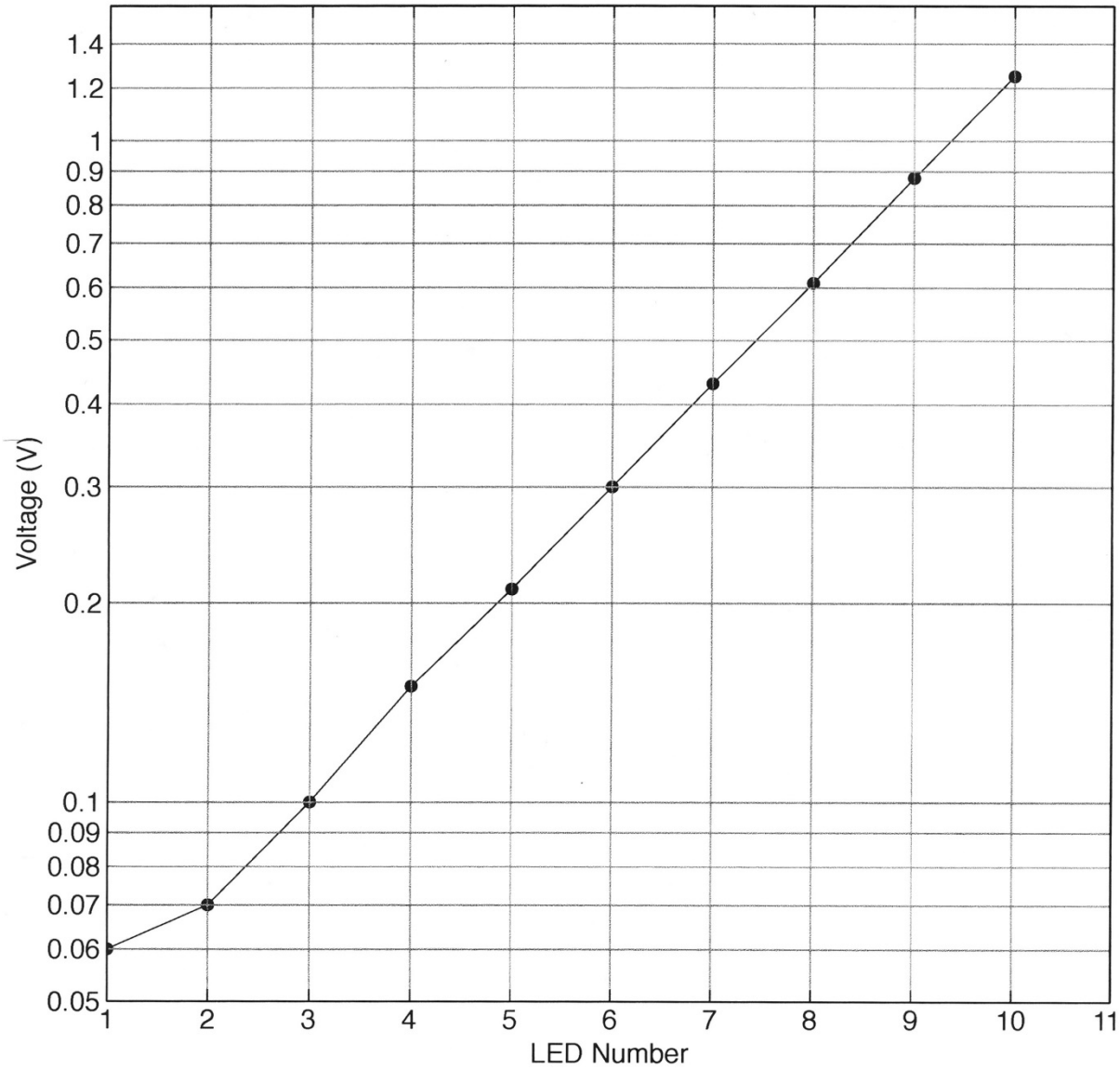


Figure 5.17.
This plot uses a logarithmic scale for the y axis, or voltage, to enhance the low-voltage readings and clearly show the magnitudes involved.

Step 8.

Please turn off power to your breadboard. You have completed this experiment.

The family of bar-graph ICs includes a third device, the LM3916 Dot/Bar Display Driver that has a volume-unit (VU) response common in the recording and broadcast industries. You might see VU, often pronounced "view," meters in audio equipment to show sound levels. A VU meter has a defined response to sound volume. The data sheet for the LM3916 IC does not define or explain the volume unit or how it relates to sound energy, although it shows several circuits you could use with the LM3916 IC. A detailed description of volume units and VU meters goes beyond the scope of this book. For a good article about audio metering, please visit: http://www.planetanalog.com/document.asp?doc_id=532491. Wikipedia provides a short description of VU meters and sound measurements at: http://en.wikipedia.org/wiki/VU_meter.

In the next experiment you will use a microcontroller (MCU) module to control a bar-graph-type array of LEDs. By using an MCU, you can control LEDs individually through software commands.

References

1. "LM3914 Dot/Bar Display Driver," data sheet. Texas Instruments. http://www.ti.com/lit/ds/symlink/lm3914.pdf.

2. "LM3915 Dot/Bar Display Driver," data sheet. Texas Instruments. http://www.ti.com/lit/ds/symlink/lm3915.pdf.

3. "LM3916 Dot/Bar Display Driver," data sheet. Texas Instruments. http://www.ti.com/lit/ds/symlink/lm3916.pdf.

4. "Introduction to Logarithms," http://www.mathsisfun.com/algebra/logarithms.html.

Answers

Experiment 5, Step 5:

You can have two answers for this problem: one when you use a 2200-ohm resistor and one when you use a 2400-ohm resistor. The 9100-ohm fixed resistor and the 2000-ohm trimmer remain the same in both cases. (See **Figure 5.18**.)

1. For the 2200-ohm resistor:

$9100\ ohms\ +\ 2000\ ohms\ +\ 2200\ ohms\ =\ 13{,}300\ ohms$

Divide the input voltage by the total resistance to get a volts-per-ohms answer:

$5\ volts\ /\ 13{,}300\ ohms\ =\ 3.8\ x\ 10^{-4}\ volts/ohm$

It makes sense to calculate voltages referenced to ground, so multiply:

$3.8\ x\ 10^{-4}\ volts/ohm\ *\ 2200\ ohms\ =\ 0.82\ volts$ across the 2400-ohm resistor.

$3.8\ x\ 10^{-4}\ volts/ohm\ *\ (2200\ ohms\ +\ 2000\ ohms)\ =\ 1.59\ volts$ at the top of the trimmer.

So the trimmer output has the range 0.82 to 1.59 volts.

2. For the 2400-ohm resistor:

$9100\ ohms\ +\ 2000\ ohms\ +\ 2400\ ohms\ =\ 13{,}500\ ohms$

Divide the input voltage by the resistance to get a volts-per-ohms answer:

$5\ volts\ /\ 13{,}500\ ohms\ =\ 3.7\ x\ 10^{-4}\ volts/ohm$

It makes sense to calculate voltages referenced to ground, so multiply:

$3.7\ x\ 10^{-4}\ volts/ohm\ *\ 2400\ ohms\ =\ 0.88\ volts$ across the 2400-ohm resistor.

$3.7\ x\ 10^{-4}\ volts/ohm\ *\ (2400\ ohms\ +\ 2000\ ohms)\ =\ 1.62\ volts$ at the top of the trimmer.

So the trimmer output has the range 0.88 to 1.62 volts.

For a 5-volt input, either the 2200- or 2400-ohm resistor will give good results.

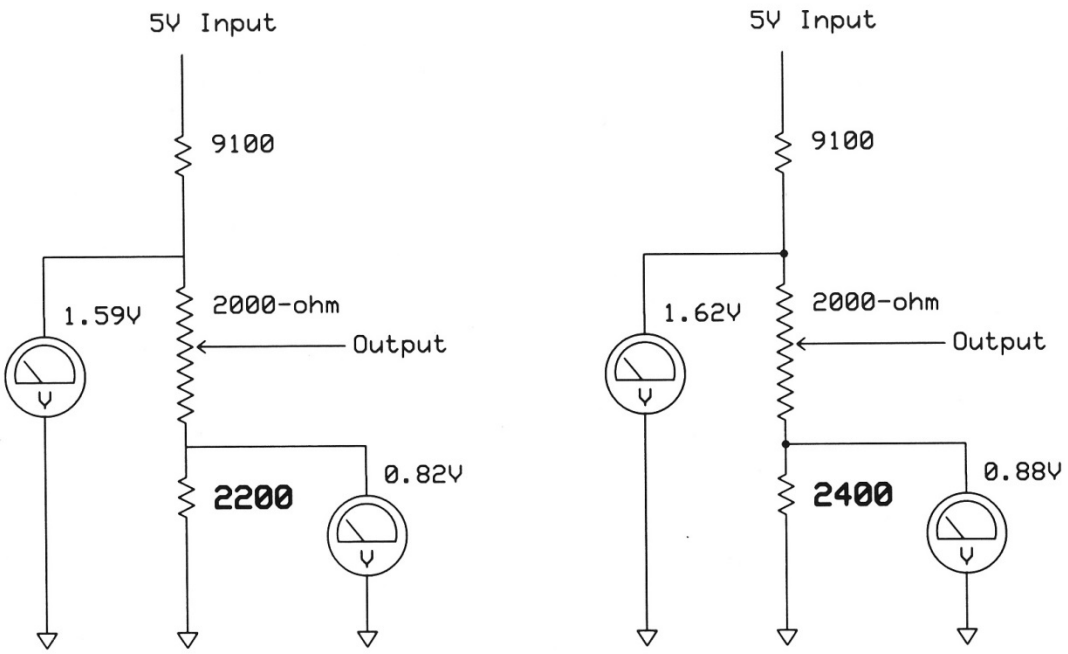


Figure 5.18.
These two divider circuits show the voltage measurements when the circuit uses a 2200- or a 2400-ohm resistor between the "bottom" end of the trimmer and ground.

Note: if you decide to calculate the voltage across the 9100-ohm resistor, the +5-volt signal becomes your reference, not ground (0 volts). You must subtract the voltage across the 9100-ohm resistor from 5 volts to get the voltage at the "top" of the trimmer.

Experiment No. 6 – Control the Brightness of LEDs with an MCU

Abstract

This experiment explains pulse-width modulation and how you can apply it to control the brightness of LEDs. You also will learn how to use software objects others have written and shared for the Parallax Propeller microcontroller.

Keywords

LED, pulse-width modulation, pulse-width modulator, PWM, intensity, objects, Propeller, software, duty cycle, microcontroller, MCU, Spin language, Parallax Serial Terminal, PST

Requirements

(1) - Parallax Propeller P8X32A QuickStart module
(1) - USB 2.0 A-Male to Mini-B cable

Introduction

In Experiment 4 you learned how to turn on or off individual LEDs on a Propeller P8X32A QuickStart board, but the LEDs always provided the same light intensity. In some situations you might want to turn an LED on and control its brightness for illumination, to indicate a condition, or possibly to save power. You already know a resistor in series with an LED limits the amount of current that flows through an LED and thus its intensity, or brightness. Increase this resistance and the LED brightness decreases. Decrease this resistance and brightness increases. A microcontroller (MCU) could change resistance values with some sort of external apparatus, but at the cost of a more complicated circuit and loss of flexibility.

An alternate MCU technique, pulse-width modulation, or PWM, controlled by software lets you easily change the *average* current through an LED, and thus affect its brightness. The timing diagram in **Figure 6.1** shows a signal that exists as a logic-1 for 20 percent of the PWM period. We say this signal has a 20-percent duty cycle. Thus an LED attached to this signal turns on for only 20 percent of the time. As a result, the average current through the LED amounts to 20 percent of the current when it has a constant logic-1 voltage applied to it. So LED appears dimly lit.

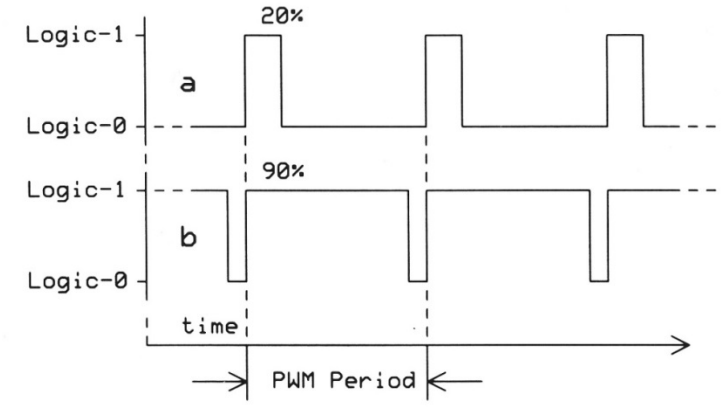


Figure 6.1.
PWM signals shown here represent a 20-percent (a), and 90-percent (b) duty cycle. The portion of the time the signal exists as a logic-1 divided by the period time yields the duty-cycle ratio. Multiply the answer by 100 to get a duty cycle as a percent.

In **Figure 6.1**, notice the period of the pulse-width modulated (PWM) signal remains constant as the duty cycle changes from 20 to 90 percent. Only the portion of time in the logic-0 and logic-1 state have varied. As a result you have a fixed-period signal for which the MCU "modulated" the width of a logic-1 pulse. Modulation simply means the MCU changes a characteristic of the signal. Radio stations on the AM and FM bands modulate amplitude or frequency, respectively.

Some MCUs have internal sections dedicated to specific PWM output pins. The Parallax Propeller does not, but it provides timers and counters that can do the job at any pin. Understanding these internal components can take quite a bit of study and experimentation, but not to worry. The Propeller community of users generously shares programming *objects* created to handle the "behind the scenes" details. Programmers have no reason to replicate operations others share with us for free. This experiment takes advantage of a PWM object created several years ago by Jev Kuznetsov and code in Parallax Semiconductor Application Note AN001, "Propeller P8X32A Counters." You can download this app note by visiting http://www.parallax.com/downloads and searching "AN001".

Step 1.

Download Jev Kuznetsov's software, "Dedicated pwm generator," from the Propeller Object Exchange Library: http://obex.parallax.com/objects/216. The download comprises a ZIP package of two files: pwmAsm.spin and pwmasm_test.spin. Unzip the two files and put them in your Parallax working directory, or find these files in the Experiment 6 folder. You only need the pwmAsm.spin file.

The pwmAsm.spin file provides four public objects other Propeller Spin-language programs may use:

`PUB Start (Pin)`	Start the PWM operation at a specified I/O pin.
`PUB stop`	Stop an object and free a Propeller cog (MCU core).
`PUB SetPeriod (counts)`	Set PWM period in terms of clock cycles.
`PUB SetDuty (counts)`	Set the duty cycle in terms of counts, 0 to 100.

The pwmASM.spin listing looks complicated because it uses assembly-language statements that look cryptic. Assembly-language instructions operate directly with an MCU's hardware. Thankfully, though, we just use the public objects

IMPORTANT: The Kuznetsov PWM program pwmAsm has a small problem as described in the Notes section at the end of this experiment. The problem does not affect this experiment; however, I want you to know about it. You will find other PWM objects and test programs in the Propeller Object Exchange Library: http://obex.parallax.com/. One of those object might better meet your requirements for a project that needs a PWM output.

Step 2.

Connect the P8X32A QuickStart board to your PC and start the Propeller Tool software. From within the Propeller Tool, go to the Experiment 6 folder and open **Program 6.1**, or enter it from the listing that follows. You also should have the pwmAsm.spin program open, which will give you a tab for each program at the top of your editing window (**Figure 6.2**). Click on the **Program 6.1** tab to see the test software. This program illustrates how two PWM outputs control the intensity of two LEDs.

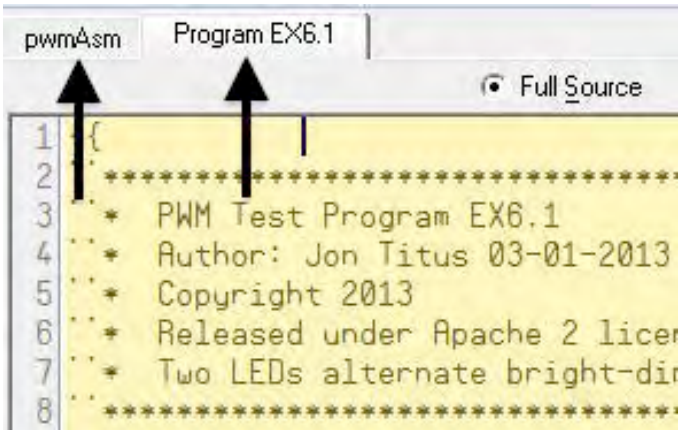


Figure 6.2.
Programs 6.1 and pwmAsm.spin should open with separate tabs to indicate they have loaded into the Propeller software editing window. (You will not see "EX" in front of names for released software for this book. That's my shorthand for experimental code.)

Program 6.1.

```
{{
''*********************************************
''*  PWM Test Program 6.1
''*  Author: Jon Titus 03-01-2013 Rev. 1
''*  Copyright 2013
''*  Released under Apache 2 license
''*  Two LEDs alternate bright-dim-bright cycles
''*********************************************
}}

CON _clkmode = xtal1 + pll16x   'Set MCU clock operation
    _xinfreq = 5_000_000        'Set for 5 MHz crystal

    max_duty = 100              'Maximum 100% duty cycle
    pwm1pin  = 22               'Control LED at P22
    pwm2pin  = 23               'Control LED at P23

OBJ
  pwm1  :  "pwmasm"             'Ensure you have pwmasm.spin
                                     'file in your working directory
  pwm2  :  "pwmasm"             '

PUB go | x                      'declare x as a local variable
  pwm1.start(pwm1pin)           'Startup for LED at P22
  pwm2.start(pwm2pin)           'Startup for LED at P23

  repeat                        'Advance duty cycle
                                'from 0 to 100
    repeat x from 0 to max_duty
      pwm1.SetDuty(x)           'LED P22 brightens
      pwm2.SetDuty(100-x)       'LED P23 dims
      waitcnt(1_000_000 + cnt)  'short delay between changes

    repeat x from max_duty to 0
      pwm1.SetDuty(x)           'LED P22 dims
      pwm2.SetDuty(100-x)       'LED P23 brightens
      waitcnt(1_000_000 + cnt)  'short delay between changes
```

Ensure you have **Program 6.1** displayed in the Propeller Tool editing window so you can see its statements. Run **Program 6.1**. What do you see? You should observe LEDs P23 and P22 get brighter and dimmer, back and forth again and again. Did you see the LEDs flash? You should not see the LEDs turn on or off in response to the PWM signal. Our eyes cannot respond to the high-frequency changes. Instead, we see only the average of the brightness changes.

Step 3.

The two LEDs at P22 and P23 go through brightening and dimming cycles, but suppose you want to set a specific brightness level and keep an LED lit at that level. You could change the statement `pwm1.SetDuty(x)` to `pwm1.SetDuty(45)` to get a 45-percent duty cycle for the `pwm1pin` output (P22 LED), but in most cases, you don't want to alter a program each time you need to change LED brightness. The Propeller Tool and the Propeller-chip software give you a way to send data from your PC to the Propeller board, and *vice versa*. These types of communications take place over the same USB cable that programs the Propeller chip on the P8X32A board. With a bit more programming, you could transmit the value 80 to the Propeller chip and cause one or both of the LEDs to turn on with an 80-percent duty cycle. Then you would have the flexibility to set any brightness level you want, from off to full-on.

The Parallax Serial Terminal (PST) available from within the Propeller Tool simplifies this type of back-and-forth communication and it serves as a helpful tool for testing and debugging programs. To use the Parallax Serial Terminal software, ensure you have a P8X32A board connected to your PC. Then start the Propeller Tool if you don't have it running now. The board's green power LED should turn on. After the Propeller Tool software has started, go to the top-left menu bar and click on "Run." Then select "Identify Hardware... F7," or press the F7 key. The PC will display information as shown in **Figure 6.3.** Write down the communication-port information. My PC used Com Port 11 (COM11) to communicate with the Propeller board.

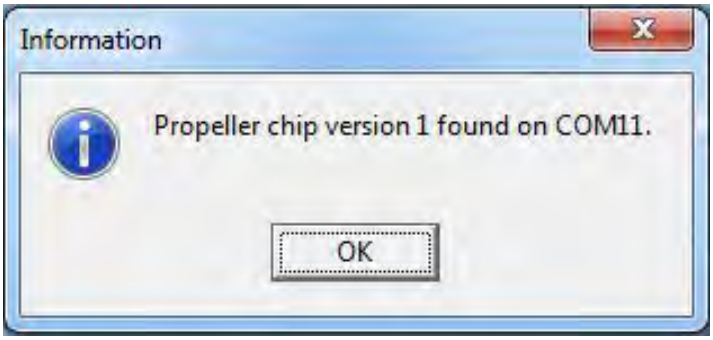


Figure 6.3.
The information in this window identifies the communication port the Propeller Tool and Parallax Serial Terminal use to communicate with your P8X32A board.

If you forgot to connect your P8X32A board to your USB cable, or disconnected the cable from your PC, the Propeller Tool will not "find" the board and you will see the message displayed in **Figure 6.4**. In this case, recheck your cable connections and ensure you have the P8X32A board connected to your PC. The board's power LED should turn on.

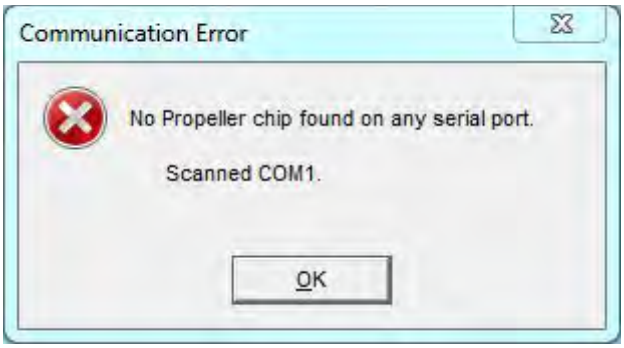


Figure 6.4.
This error message indicates the lack of a connection between your PC and a P8X32A Propeller board. If you see this message recheck your cable connections.

When you have the Com Port information, go back to the "Run" menu and click on "Parallax Serial Terminal... F12," or press the F12 key on your keyboard. You should see the terminal window as shown in **Figure 6.5**. For more Parallax Serial Terminal (PST) information, refer to the document, "Using the Parallax Serial Terminal," available at: http://learn.parallax.com/KickStart/ParallaxSerialTerminal.

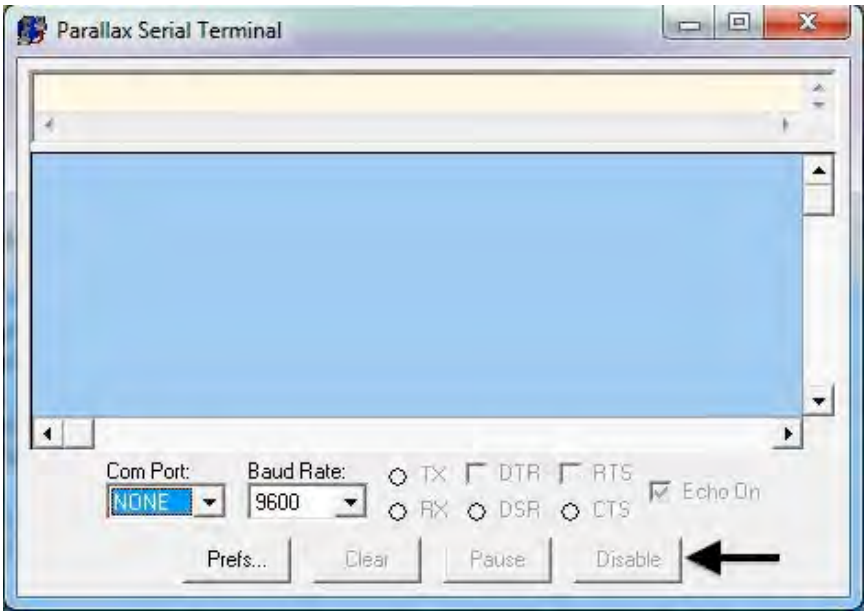


Figure 6.5.
The Parallax Serial Terminal (PST) provides a way to communicate with the Propeller. The Propeller IC also can send information back to the PST window for debugging or testing.

Within the PST window, look for the "Com Port:" selection area at the bottom left corner of the PST window. Select the Com Port you identified earlier. Select the 9600 "Baud Rate" because the programs that follow will set this rate for the Propeller IC. (You might see the "Enable" click button flash in the lower-right corner of the PST window, as indicated by the large arrow in **Figure 6.5**. You will need this control later, but ignore it for now.) Minimize the PST window, but do not close it.

Step 4.

Before your Propeller IC can transfer data to or from your PC, the MCU needs communication software. Again we rely on a set of objects created by a Propeller programmer, and shared with others in the Propeller Object Exchange Library (OBEX). You will need two sets of objects as noted below. You can locate these programs

in the Experiment 6 folder, too, along with all the software for this experiment. If you download the objects noted below, save them in your Propeller working directory:

a) "Full-Duplex Serial Driver" version 1.2 or higher. Download the Spin-language file from the Propeller Object Exchange at: http://obex.parallax.com/objects/54/

b) "Extended Full-Duplex Serial Object," version 1.1.0 or higher. Download the Spin-language file from the Propeller Object Exchange at: http://obex.parallax.com/objects/31/.

After you place these two files in your working directory, open them in the Parallax Tool editing window. The Extended Full-Duplex Serial Object puts a "wrapper" around the Full-Duplex Serial Driver object and adds helpful functions that simplify communications.

Step 5.

The PWM-test program that will follow shortly includes several communication operations I'll explain before you look at the complete program.

```
OBJ
      Serial  : "Extended_FDSerial"
```

This definition links the PWM-test program (**Program 6.2**, which you will see soon) to objects in the Extended Full-Duplex Serial file. Now you can use the prefix `"Serial"` to identify and run the communication objects in the SPIN-language file Extended_FDSerial.

```
Serial.start(31,30,0,9600)
```

This statement sets up the Propeller IC for serial communications through I/O pins P31 (receive) and P30 (transmit). These pins connect to the USB-interface IC on the P8X32A board that communicates with your host PC. The 0 in this statement resets any special communication options previous software might have used. Finally, the value 9600 establishes the baud rate – 9600 bits per second – that must match the baud rate you set in the PST window.

Next, software uses the statement:

```
Serial.RxFlush
```

to ensure any old information received via the USB IC gets "flushes out" so the Propeller IC's receiver program gets a fresh start.

```
ser_data  :=  Serial.RxDec
```

The Propeller receiver captures one byte at a time as you enter information in the PST window. Communications use an encoding scheme called the American Standard Code for Information Interchange, or ASCII, which people generally pronounce as "ask-ee." Each keyboard character has a unique code. The letter "A" equals hexadecimal 41_{16} ($41) and the letter "w" equals hexadecimal 77_{16} ($77), for example. When you enter the number 68, the Propeller receives the *code* for the "6" key followed by the code for the "8" key..

Martin Hebel, the primary creator of the Extended Full-Duplex Serial Object, included the .RxDec object that programmers can use to receive the ASCII keyboard values for individual numerals and convert them to the actual value. Suppose you use the PST to type 6, 8, and Enter. The Serial.RxDec object takes the ASCII information and converts it to 01100100_2 (%01100100 in Propeller code), or $100 in hexadecimal, both of which equal 68_{10}. So if you need to enter 68, or some other value, to get the Propeller to take an action, such as set a temperature, your Propeller software will use the Serial.RxDec to provide the proper value to work with.

You don't have to translate values from binary to hexadecimal, to ASCII-character values, and so on. Propeller objects do that work for you.

The statement `ser_data := Serial.RxDec` receives keyboard codes for each numeral and converts them to their actual value as soon as you press the Enter key. The .RxDec object cannot convert values that include a decimal fraction, such as 4.72.

Step 6.

The code shown in **Program 6.2** extends **Program 6.1** to include the communication objects explained in Step 5. Later, when you enter a value on the PC's keyboard and press Enter, the Propeller chip will receive the characters and convert them to the corresponding value. Then the code will use that value to change the PWM duty cycle of two LEDs, P22 and P23. You can make them dimmer or brighter.

Program 6.2.

```
{{
''*********************************************
''*  PWM Test Program 6.2
''*  Author: Jon Titus 11-17-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Two LEDs dim or brighten depending on the
''*  value entered from the PC keyboard via the
''*  Parallax Serial Terminal. Acceptable values
''*  range from 0 (off) to 100 (full on)
''*********************************************
}}

CON _clkmode = xtal1 + pll16x   'Set MCU clock operation
    _xinfreq = 5_000_000        'Set for 5 MHz crystal

    max_duty = 100              'Maximum 100% duty cycle
    pwm1pin  = 22               'Control LED at P22
    pwm2pin  = 23               'Control LED at P23

VAR byte ser_data               'Byte storage for PWM duty cycle

'You must have pwmasm.spin file 'in your working directory
'Serial communication object file also goes in
'working directory

OBJ
  pwm1  :  "pwmasm"
  pwm2  :  "pwmasm"
  Serial : "Extended_FDSerial"

PUB Start                       'Main program starts here
  pwm1.start(pwm1pin)           'Startup for LED at P22
  pwm2.start(pwm2pin)           'Startup for LED at P23
  pwm1.SetDuty(0)               'Start with both LEDs off
  pwm2.SetDuty(0)


'Set serial comms, P31 RX, Pin 30 TX. 9600 = baud rate. Must be
'same as Parallax Serial Terminal (PST)
  Serial.start(31,30,0,9600)
```

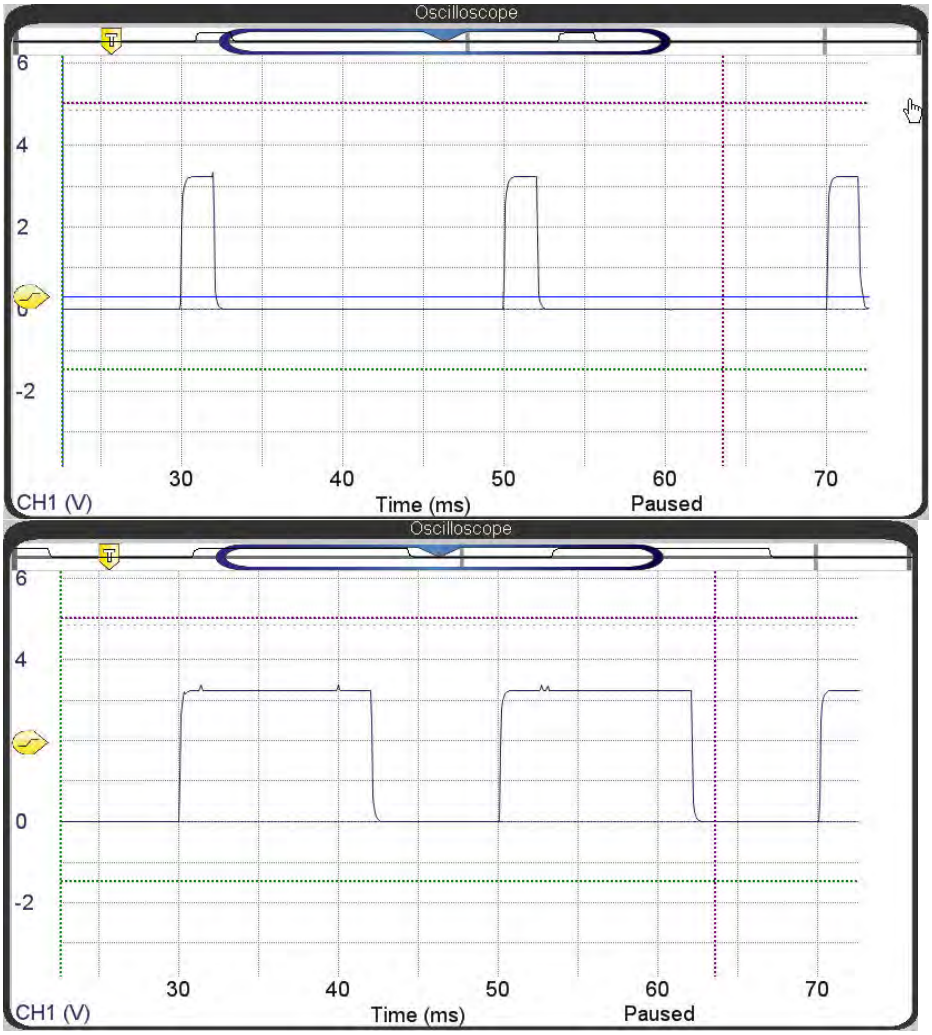
```
  Serial.RxFlush                'Clean out receiver buffer

  repeat                        'repeat loop "forever"
    ser_data := Serial.RxDec    'Get rcvd value as decimal
    pwm1.SetDuty(ser_data)      'Set PWM duty cycle
    pwm2.SetDuty(ser_data)      'for both PWM outputs
    waitcnt(1_000_000 + cnt)    'short delay between changes
```

Load this program and check it to catch any typing errors. Press F10 or click on Run – > Compile Current – > Load RAM. The LEDs will remain off. You should have the terminal software running on your PC. If necessary, expand the terminal window and click the button with the flashing "Enable" legend. If you need to start the terminal software, go to the Run menu and select "Parallax Serial Terminal," or press the F12 key.

Now type values between 0 and 100 (remember to press the Enter key after each value), and watch the LEDs. For a 0-percent duty cycle, the LEDs turn off, for a 100-percent duty cycle, they turn full on. Try in between values. Do the LEDs change brightness? They should.

In this step you changed the duty cycle of the pulses delivered to each LED. The diagrams in **Figure 6.6** show oscilloscope displays for 10-, 60-, and 95-percent duty-cycle signals from a Propeller PWM output.



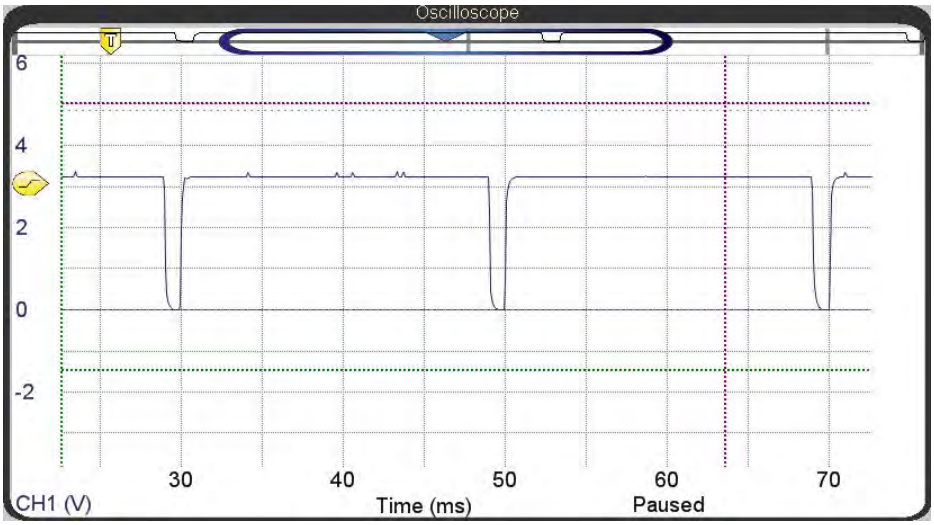


Figure 6.6.
These traces from an oscilloscope show the difference between PWM signals with a 10-percent (top), a 60-percent (middle), and a 95-percent (bottom) duty cycle. These signals drove one of the LEDs on my P8X32A board.

What would happen if you type a negative number, say, -35? First type in 10 [Enter] and the LEDs should dim. Then type in -35? How do the LEDs change? The LEDs got much brighter. Why?

To find out, either:

a) Insert the three statement shown below in the white area into **Program 6.2** and save the program in your Propeller workspace folder as **Program 6.3**.

b) Close **Program 6.2** and load **Program 6.3** from the set of programs in the Experiment 6 folder.

The first statement in the white area below prints the message between the quote marks. The second statement prints the decimal value of `ser_data,` the value you entered to control the PWM duty cycle. The third statement transmits the value 13_{10} to the PST to move the terminal cursor to a new line. The value "13" serves as a *command* code to the PST. You won't see the number 13 in the PST window.

```
repeat
      ser_data := Serial.RxDec
      Serial.str(String("Value of ser_data = "))
      Serial.dec(ser_data)
      Serial.tx(13)
      pwm1SetDuty(ser_data)
      etc...
```

Program 6.3

```
{{
''*********************************************
''*  PWM Test Program 6.3
''*  Author: Jon Titus 11-17-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Two LEDs dim or brighten depending on the
''*  value entered from the PC keyboard via the
''*  Parallax Serial Terminal. Acceptable values
''*  range from 0 (off) to 100 (full on)
''*  Added statements to put ser_data on
```

```
''*  the terminal so you can see its value.
''*****************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

    max_duty = 100                  'Maximum 100% duty cycle
    pwm1pin  = 22                   'Control LED at P22
    pwm2pin  = 23                   'Control LED at P23

VAR
  byte ser_data                   'Byte storage for PWM duty cycle

OBJ
  pwm1  :  "pwmasm"                 'Ensure you have pwmasm.spin file
  pwm2  :  "pwmasm"                 'in your working directory
  Serial : "Extended_FDSerial"      'Serial communication object file
                                    'also goes in working directory

PUB Start                           'Main program starts here
  pwm1.start(pwm1pin)               'Startup for LED at P22
  pwm2.start(pwm2pin)               'Startup for LED at P23
  pwm1.SetDuty(0)                   'Start with both LEDs off
  pwm2.SetDuty(0)

  Serial.start(31,30,0,9600)        'Set serial communications, P31 RX
                                    'Pin 30 TX, 0 = reset options
                                    '9600 = baud rate. Must be same as
                                    'Parallax Serial Terminal (PST)
  Serial.RxFlush                    'Clean out receiver buffer

  repeat                            'repeat loop "forever"
    ser_data := Serial.RxDec        'Get rcvd value as decimal
                                    'Print ser_data on PST

    Serial.str(String("Value of ser_date =  "))

    Serial.dec(ser_data)            'Now print the value of ser_data
    Serial.tx(13)                   'Send control value to PST

    pwm1.SetDuty(ser_data)          'Set PWM duty cycles
    pwm2.SetDuty(ser_data)          'for both PWM outputs
    waitcnt(1_000_000 + cnt)        'short delay between changes
```

Run **Program 6.3** and enter values from your PC's keyboard, but keep them between 0 and 100. The brightness of the LEDs should change as expected. If the PST window shows a check mark in the "Echo On" box you will see the value you entered appear on its own line in the PST window. And, you will see the `ser_data` value transmitted back from the Propeller IC displayed immediately after the text, "`Value of ser_data =`". The PST window on your PC should look similar to the illustration shown in **Figure 6.7**.

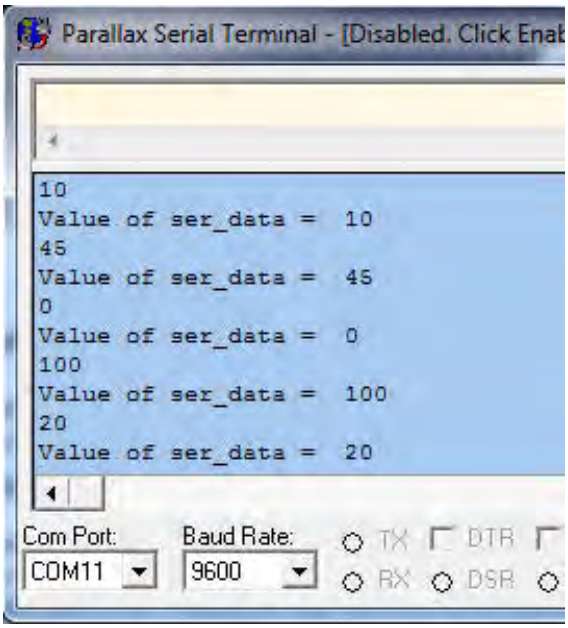


Figure 6.7.
The PST window displays a value you typed on a line and the value of the `ser_data` variable at the end of a text line.

Enter the value 10 to dim the LEDs. Then enter the value -35. Does the LED brightness increase or decrease? What do you see in the terminal window as the `ser_data` for the entered value -35? **Figure 6.8** shows what I observed.

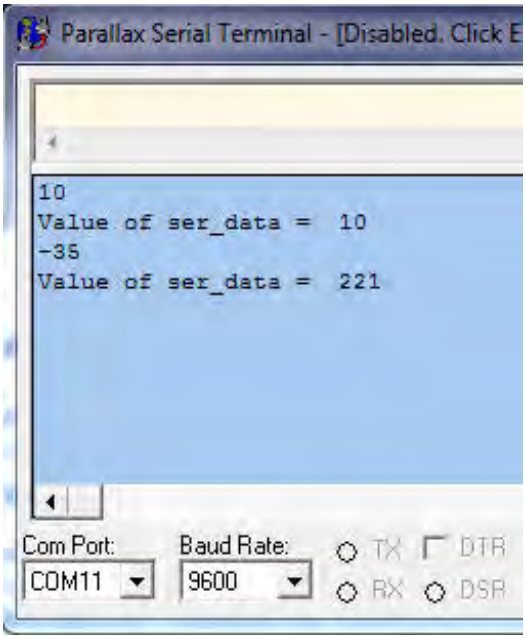


Figure 6.8.
Entering the value -35 for the PWM program yields a `ser-data` value of 221 in the Propeller IC.

If you entered -35, why does the Propeller report the value 221? This unexpected transformation arises from the way in which computers store positive and negative numbers. Because the PWM program uses values between 0 and 100, ideally the software should not let values outside this range – or decimal fractions – affect

the PWM settings. What tests or operations could you use in Propeller software to keep values received by the Propeller IC within these bounds? See my suggestions in the Answers section at the end of this experiment.

Step 7.

This step explains why you will get the value 221 when you enter -35. It involves the concept of negative binary numbers, so if you wish, you may skip ahead to Step 8.

Say you have a 8-bit byte. Given eight bits, you can save one of 256 different values, 00000000_2 through 11111111_2. But computers often use the most significant bit (MSB) to represent the sign of a number where a 0 indicates a positive number and 1 indicates a negative number. Thus positive numbers range from 00000000_2 to 01111111_2, or 0_{10} to 127_{10}. The value zero is neither positive nor negative. It is just zero.

Negative numbers can range from 11111111_2 to 10000000_2, or -1_{10} to -128_{10}. These negative binary numbers can seem tricky at first, but the information in **Table 6.1** helps clarify the numbering system.

Table 6.1. Representation of signed 8-bit binary numbers.

8-Bit Binary	Decimal
01111111	127
01111110	126
.....	.....
00000111	7
00000110	6
00000101	5
00000100	4
00000011	3
00000010	2
00000001	1
00000000	0
11111111	-1
11111110	-2
11111101	-3
11111100	-4
11111011	-5
11111010	-6
11111001	-7
.....	.....
10000001	-127
10000000	-128

If you add +5 and -5, you should get an answer of 0, so go to **Table 6.1** and take the equivalent binary codes for these two values. Then add them:

```
00000101  =  +5
11111011  =  -5
00000000  =  sum
```

As expected the binary sum also comes to zero. So how do you take a number and get its negative equivalent? First, take the positive value, say 75_{10}, or 1001011_2. Then take the *complement* of the binary value, which

simply means change each zero into a one, and change each one into a zero: $01001011_2 = 75_{10}$ so the complement comes to 10110100_2.

Now add one (1) to this binary value and get 10110101_2. The complement-and-add-one process is called a *two's-complement* and it gives you the negative binary number for a negative value.

When you entered -35 the Propeller reported the value 221. To discover why, convert -35_{10} to binary. I used a calculator and converted -35 to 11011101_2. The statement `Serial.dec(ser_data)`doesn't "know" about the use of the MSB to indicate the sign of the value. Instead, the `Serial.dec(ser_data)` statement takes the 11011101_2 and converts all eight bits to a decimal number. What value would 11011101_2 represent? 221.

Why does the -35_{10} value cause the two LEDs to turn on with full brightness? The `pwmAsm` object `SetDuty` examines an incoming duty-cycle value before using it. If the value exceeds 100 (100 percent), the `SetDuty` object forces the duty cycle to 100. Likewise, if the value goes below 0 (0 percent), the `SetDuty` object sets the duty cycle to 0 instead. In this case, the `SetDuty` object received the value 221 and forced the PWM output to a 100-percent duty cycle.

Step 8.

In each of the oscilloscope images shown earlier in **Figure 6.6**, the duty cycles, or on-off times, changed, but the period – that is, the time between the first negative edge (higher-to-lower-voltage change) and the second negative edge remained the same. A new PWM statements lets programmers change the *period* of PWM signals. The instruction immediately below sets a period of 37.6 microseconds (37.6 µsec) for the pwm2 output.

```
pwm2.SetPeriod(3000)
```

The instruction:

```
pwm2.SetPeriod(1_600_000)
```

creates the 20-msec period shown in **Figure 6.9**. This period works well when you must control a servo motor of the type used in model cars and airplanes, and in robots. (These "servos" require a duty cycle of 7.5 percent (1.5 msec) for the center position; 10 percent (2 msec) for a 90-degree position change; or 5 percent (1 msec) for a -90-degree position change.) I have preset this period in **Program 6.4**, which you can run if you have an oscilloscope and want to see the PWM timing.

In **Program 6.4**, the period remains set for 20 msec, but you can still enter the duty cycle. Note in this new program I have separated the set-up and configuration for `pwm1` and `pwm2` into two groups of statements. Please read the Note at the end of this experiment for an explanation and instructions about using the pwmAsm.spin objects.

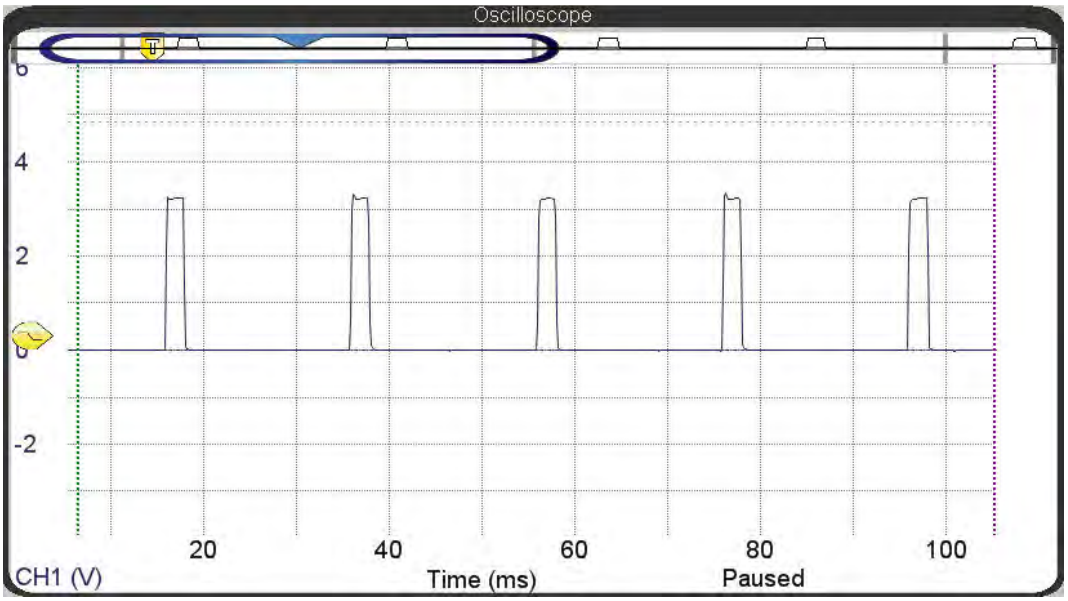


Figure 6.9.
This PWM signal with a 20-millisecond period will operate servo motors to control models or robots. The pulse width varies from about one to two milliseconds to cause the servo to turn left or right.

Program 6.4

```
{{
''************************************************
''*  PWM Test Program 6.4
''*  Author: Jon Titus 11-17-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Two LEDs dim or brighten depending on the
''*  value entered from the PC keyboard via the
''*  Parallax Serial Terminal. Acceptable values
''*  range from 0 (off) to 100 (full on)
''*  Added statements to put ser_data on
''*  the terminal so you can see its value.
''************************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

    max_duty = 100                  'Maximum 100% duty cycle
    pwm1pin  = 22                   'Control LED at P22
    pwm2pin  = 23                   'Control LED at P23

VAR byte ser_data                   'Byte storage for PWM duty cycle

OBJ
  pwm1  :  "pwmasm"                 'Ensure you have pwmasm.spin file
  pwm2  :  "pwmasm"                 'in your working directory
  Serial : "Extended_FDSerial"      'Serial communication object file
                                    'also goes in working directory

PUB Start                           'Main program starts here
  pwm1.start(pwm1pin)               'Startup for LED at P22
```

```
  pwm1.SetDuty(0)                   'Start duty cycle at 0 percent
  pwm1.SetPeriod(1_600_000)         'Set period for 20 msec.

  pwm2.start(pwm2pin)               'Startup for LED at P23
  pwm2.SetDuty(0)                   'Start duty cycle at 0 percent
  pwm2.SetPeriod(1_600_000)         'Set period for 20 msec.

  Serial.start(31,30,0,9600)        'Set serial communications, P31 RX
                                    'Pin 30 TX, 0 = reset options
                                    '9600 = baud rate. Must be same as
                                    'Parallax Serial Terminal (PST)
  Serial.RxFlush                    'Clean out receiver buffer

  repeat                            'repeat loop "forever"
    ser_data := Serial.RxDec        'Get rcvd value as decimal
    pwm1.SetDuty(ser_data)          'Set PWM duty cycles
    pwm2.SetDuty(ser_data)          'for both PWM outputs
    waitcnt(1_000_000 + cnt)        'short delay between changes
```

Step 9:
Whether you want to control a set of LEDs, a servo motor, or some other device, a simple equation lets you calculate the `period_value` to use in the statement: `pwm2.SetPeriod(period_value)`:

$period\_value \;=\; Period\ (in\ seconds)\ x\ 8.00\ x\ 107\ Hz$, or

$period\_value \;=\; Period\ (in\ seconds)\ x\ 80{,}000{,}000\ /\ sec$

Yes, that's 80 million. If you want a 12.5 msec (0.0125 seconds) period:

$period\_value \;=\; 0.0125\ seconds\ x\ 80{,}000{,}000\ /\ sec$

$period\_value \;=\; 1{,}000{,}000$, or 1_000_000 for a Propeller program, which makes it easier to read. Note that the units of seconds in the numerator and denominator cancel each other, so the answer has no units. It is just a number.

The value 80,000,000 represents the Propeller's internal clock frequency, 80 MHz, or 80,000,000 cycles per second. Although the Propeller uses a 5-MHz crystal to create a clock signal, an internal circuit, called a phase-lock loop (PLL), increases the internal frequency to 80 MHz.

The Propeller has a default `period_value` of 1000 for a PWM output. That means if you set up a PWM output and do not specify a `period_value`, the software uses the value 1000. Of course, you can change this value in your program. What do you calculate as the default *period* for the value 1000? Find the value in the Answer section at the end of this experiment.

Notes

The Kuznetsov PWM program pwmAsm has a small problem discovered when I created this experiment. The following statements from **Program 6.1** work properly:

```
pwm1.start(pwm1pin)
pwm2.start(pwm2pin
pwm2.SetPeriod(30000)
```

But if you want to set a different period for the pwm1 output and use the statements in your program in the following order, the program will run, but you will get incorrect results.

```
pwm1.start(pwm1pin)
pwm2.start(pwm2pin
pwm2.SetPeriod(30000)
pwm1.SetPeriod(70000)
```

As far as I can tell, the Kuznetsov PWM program causes an incorrect read of the period information from memory. To overcome this problem with the pwmAsm objects, write your program like this:

```
pwm1.start(pwm1pin)
pwm1.SetPeriod(70000)

pwm2.start(pwm2pin
pwm2.SetPeriod(30000)
```

and perform the operations for the pwm1 output consecutively and then perform the pwm2 output steps the same way. Do not mix the pwm1 and pwm2 statements. You can find other PWM objects in the OBEX Library that you might want to try.

Answers

Experiment 6, Step 6:

You can use an if statement so values outside the range 0-to-100 will not affect the PWM outputs. This statement also could display an error message in the PST window. I used **Program 6.4** as a starting point and added the instructions shown below.

```
repeat                          'repeat loop "forever"
    ser_data := Serial.RxDec    'Get received value, convert to
                                ' decimal
                                'Check for value 0 to 100

   if ((ser_data <0) OR (Ser_data >100))

       'Value outside range, so print error message and a new line
        Serial.str(String("Only values between 0 and 100, please."))
        Serial.tx(13)

       'Value is within range, so continue and change PWM duty cycle

   else
        pwm1.SetDuty(ser_data)  'Set PWM duty cycles to decimal value
        pwm2.SetDuty(ser_data)  'for both PWM outputs
        waitcnt(1_000_000 + cnt)    'short delay between changes
```

You will find the complete code for this new program (**Program 6.5**) in the program folder for Experiment 6. Review the operation of if-else statements in the Propeller Manual. But will the new program catch all values outside the range 0 to 100? Try 255, 256, and 350 through 360. What happens. Do you know why? The software allocated only a byte for the ser_data value. So when the Propeller receives the value 256, or:100000000_2 it saves only the eight least-significant bits 00000000_2 in ser_data. What could you do to ensure all values greater than 100 or less than 0 do not affect the LEDs? I leave it to you to find a solution. Look at the operations in the Extended_FDSerial code.

Experiment 6, Step 7:
Remember that 1 Hz = 1 / seconds. For a default `period_value` of 1000, you take the equation:

$period\_value \ = \ Period\ (seconds)\ x\ 80{,}000{,}000\ Hz$ and rearrange it to:

$period\_value\ /\ \ 80{,}000{,}000\ Hz \ = \ Period\ (seconds)$, so:

$1000\ /\ 80{,}000{,}000\ Hz \ = \ 12.5\ x\ 10-6\ seconds, or\ 12.5\ \mu sec.$

Experiment No. 7 – Control 7-Segment Multi-Digit Displays with an MCU

Abstract

In this experiment you will learn how a microcontroller can use a multiplexing technique to directly control more than one 7-segment display. Many displays use this technique because it requires fewer I/O pins than would direct control of each segment by an I/O pin. The multiplexing technique applies to other types of displays, too.

Keywords

LED, multiplex, multiplexing, driver, Darlington transistor array, multi-segment displays

Requirements

(1) - Parallax Propeller P8X32A QuickStart module
(1) - USB 2.0 A-Male to Mini-B cable
(2) - 7-Segment display, common cathode (Lumex LDS-C512RI or equivalent)
(7) - 39-ohm, 1/4-W, 5% resistors (orange-white-black)
(1) - ULN2003A Darlington transistor array, 16-pin DIP
(1) - 7-Segment display, common cathode
(1) - 4-Digit display module, common cathode (Lumex LDQ-N516RI or equivalent, optional)

Introduction

In Experiment 3 you learned how a 7-segment decoder/driver IC can accept a 4-bit binary value and control the seven LEDs in a 7-segment display. You also used a binary counter and a divide-by-10 counter that caused the display to show numbers and some graphic "characters." In this experiment you will learn how to eliminate the counter and decoder/driver and control a 7-segment display directly with a microcontroller (MCU) and software. The software will control which segments turn on or off.

Step 1.

Before you can think about control of a 7-segment display, you must describe the segments that turn on or off for a given number. Fill in the templates in **Figure 7.1** to help determine which segments you must turn on for each numeral, 0 through 9. Record your answers in **Table 7.1**. Use a "1" in the letter columns to indicate an "on" segment and use a "0" to indicate an "off" segment. This table will give you information to use in a short test program. The D7, or "X," column has a preset value of 0. The 7-segment display will not use this information.

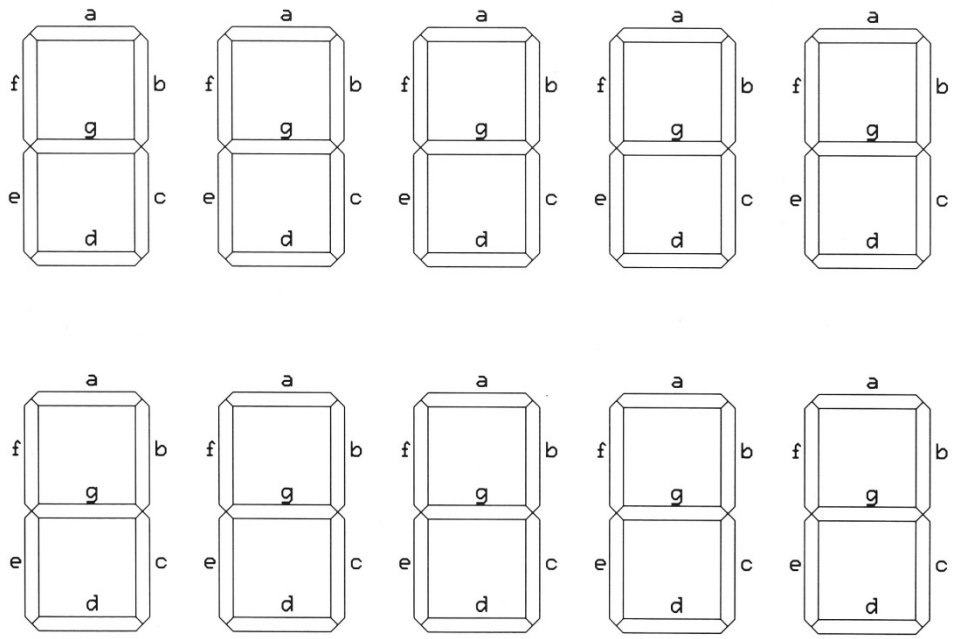


Figure 7.1.
Use this template to draw the numerals 0 through 9 so you can determine the on and off segments for the LEDs in a 7-segment display.

Table 7.1. Segment codes for a 7-segment display. (0 = off, 1 = on.)

	Segment							
	D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Numeral	**X**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0							
1	0							
2	0							
3	0							
4	0							
5	0							
6	0							
7	0							
8	0							
9	0							

Step 2.

Now that you have the code for each numeral, you will use a short test program to ensure the proper patterns appear on seven of the eight LEDs on the Propeller P8X32A board. LED P23 will remain off. **Program 7.1** gives you software you can run after you enter the code for each numeral in the "blank" array, named `segment_code[x]`, where the value of x indicates the corresponding display numeral. So after `segment_code[0] := %` you insert the eight binary bits for the numeral 0. I entered 01111110_2, so:

```
segment_code[0] := %01111110
```

Enter the remaining nine codes in the **Program 7.1** and run it. You should see the patterns appear in sequence on the seven active LEDs, P22 through P16. If you need help, find the proper settings at the end of this experiment.

Program 7.1.

```
{{
''*********************************************
''*  7-Segment Test Program 7.1
''*  Author: Jon Titus 11-17-2014 Rev. 1
''*  Copyright 2013
''*  Released under Apache 2 license
''*  7-Segment code-test program.
''*  Loop through all 10 codes and
''*  displays them on 7 LEDs on P8X32A board
''*  to ensure codes appear properly.
''*
''*********************************************
}}

CON _clkmode = xtal1 + pll16x        'Set MCU clock operation
    _xinfreq = 5_000_000             'Set for 5 MHz crystal

VAR
    byte segment_code[10]            'Storage for segment codes
    byte count                       'Variable for loop

PUB Start                            'Main program starts here

segment_code[0] := %01111110        'Pattern for "0"
segment_code[1] := %                 'Pattern for "1"
segment_code[2] := %                 'Pattern for "2"
segment_code[3] := %                 'Pattern for "3"
segment_code[4] := %                 'Pattern for "4"
segment_code[5] := %                 'Pattern for "5"
segment_code[6] := %                 'Pattern for "6"
segment_code[7] := %                 'Pattern for "7"
segment_code[8] := %                 'Pattern for "8"
segment_code[9] := %                 'Pattern for "9"

dira[23..16] := %11111111            'Set eight output pins
  repeat                             'Main loop
        repeat count from 0 to 9
                outa[23..16] := segment_code[count]
                waitcnt(clkfreq + cnt)
```

Step 3.

Program 7.1 set aside two byte variables in the variables section, `VAR`; `segment_code[10]` and `count`. The `count` variable gets used in a `repeat` loop that counts from 0 to 9 and then repeats. The statement `segment_code[10]` assigns ten byte-size memory locations to the `segment_code` variable. The value in brackets, or the index, lets software address individual bytes in the `segment_code` array, such as `segment_code[3]`.

IMPORTANT: Although the array assignment in the VAR block of **Program 7.1** sets aside 10 byte-storage locations, the index for those bytes starts at 0 and runs through 9. The maximum index in an array always points to one less than the index number used in the VAR block assignment. An array assignment such as `byte mytext[50]` would create 50 byte locations, `mytext[0]` through `mytext[49]`.

The software runs through a `repeat` loop and uses the loop counter to identify the 7-segment codes in order, 0 through 9, 0 through 9, and so on.

Step 4.

Now you will connect a 7-segment display to your Propeller board. Because a logic-1 will turn on a segment, you must use a *common-cathode* display that has inputs that power the seven-segment LEDs and a common-cathode connection to ground.

I used a Lumex-brand LDS-C512RI display that has pins along the top and bottom edges, similar to the display shown earlier in **Figure 3.3**, section b. This pin arrangements makes it easy to place displays next to each other in a breadboard without having wires get in the way. According to the LDS-C512RI data sheet, each segment LED has a forward-voltage drop of about 2.2 volts. A logic-1 output on the P8X32A Propeller board has a 3.2-volt output. The display data sheet notes a 25 mA (0.025 A) steady current for each segment.

I used Ohm's Law to calculate the current-limiting resistance to use in the display circuit. The voltage across the resistor comes to:

$$3.2\ volts\ from\ P8X32A\ output\ -\ 2.2\ volts\ across\ LED\ =\ 1.0\ volts\ across\ resistor.$$

$$I\ =\ E\ /\ R\ \text{ or }\ R\ =\ E\ /\ I$$

$$R\ =\ 1.0\ volt\ /\ 0.025\ A\ =\ 40\ ohms$$

I had 39-ohm resistors on hand and used them when I constructed the circuit. Your resistance value might differ, depending on the characteristics of the displays you use. The diagram in **Figure 7.2** shows the connections you must make and **Figure 7.3** shows the pin designations for the expansion connector on the P8X32A board. You can insert a wire into the corresponding receptacle on this connector to route a signal to a breadboard. You might make a copy of **Figure 7.3** because future experiments will need connections to these signals.

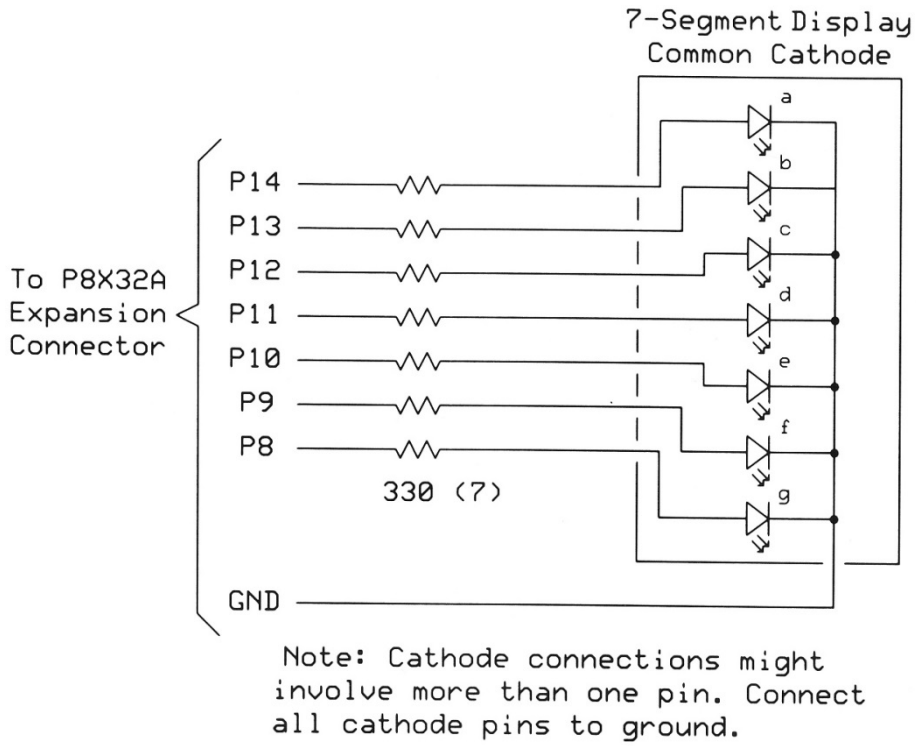


Figure 7.2.
Schematic diagram for connections to a common-cathode 7-segment display. For pin-number information refer to the data sheet for the display you will use. Displays often have two or more common-cathode connections. Ensure you connect all of them to ground.

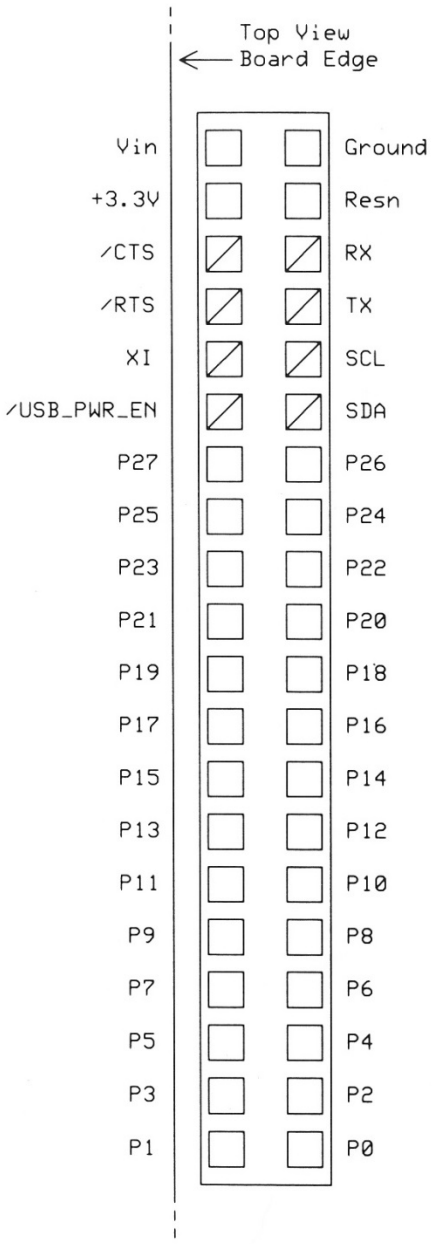


Figure 7.3.
This diagram shows the signal names for receptacles on the P8X32A Propeller board. Note the orientation of the board edge in the drawing. A diagonal line (/) indicates you should not use these signals for general-purpose I/O connections.

Step 5.

Before you run **Program 7.1**, should you change it in any way? What outputs did you connect to the 7-segment display? Do they match the `dira` and `outa` instructions in **Program 7.1**? The outputs in the **Program 7.1** listing control the on-board LEDs. To control the 7-segment display you must change:

`dira[23..16] := %1111111` to `dira[15..8] := %1111111,` and change

`outa[23..16] := segment_code[count]` to `outa[15..8] := segment_code[count]`

After you make those changes, recheck your wiring and run the program. What do you see on the display? Your display should cycle: 0, 1, 2, and so on. The Propeller MCU directly controls the seven segments.

Step 6.

In Experiment 3 you used a counter output to reset the counter when the outputs reached a specific binary count. In this way you could create a divide-by-5 counter, a divide-by-4 counter, and so on. Wiring the outputs to change the "divide-by" setting can involve adding ICs or other components. How would you change **Program 7.1** from the present divide-by-10 counter into a divide-by-7 counter? Remember, a divide-by-7 counter would display 0 through 6 again and again. Try your modification in the program and see what happens. Did it work?

By changing only the repeat statement from: `repeat count from 0 to 9` to `repeat count from 0 to 6`, you create a divide-by-seven counter. No extra hardware needed.

Step 7.

When I started this experiment, I found only *common-anode* 7-segment displays in my part bins. These displays would need a common connection to +3.3 volts and thus a logic 0 (ground) from the Propeller I/O pins to turn on segments. I wired a common-anode display as shown in **Figure 7.4** and made a small modification to **Program 7.1** to display a 0-to-9 count. Remember, instead on using logic-1 outputs to turn on an LED, I needed logic-0 outputs for the common-anode display. Without changing the pattern for each numeral, what could you do to make a common-anode display work?

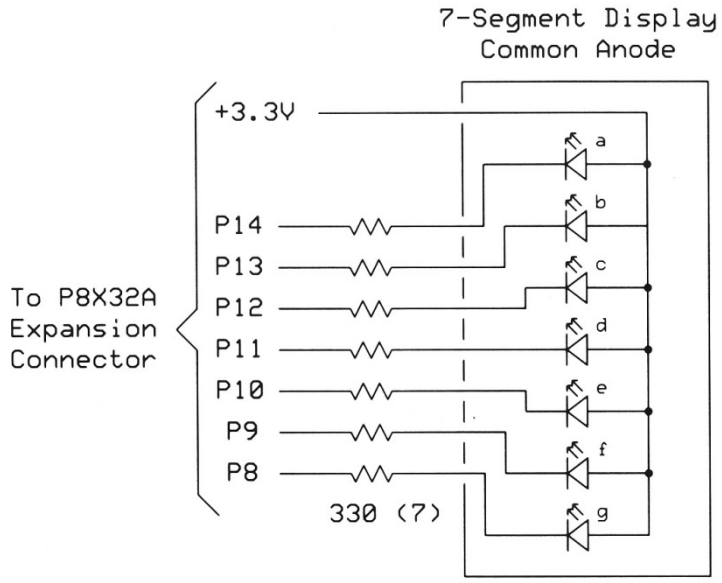


Figure 7.4.
Schematic diagram of a *common-anode* 7-segment display connected to a Propeller P8X32A board.

The Propeller's Spin language includes an command that performs a bitwise NOT, or bit-inversion, operation. A binary 1 becomes a 0, and a 0 becomes a 1. If you have the value 0110101_2 and use the bitwise NOT command, it yields the result: 1001010_2. In **Program 7.1**, I changed this statement:

```
outa[23..16] := segment_code[count]
```

to this:

```
outa[23..16] := !segment_code[count]
```

The exclamation point (!) just before the `segment_code[count]` command caused a bit-by-bit inversion of each segment pattern sent to the output pins that controlled the display. The inversion of the bits to control a common-anode display took only one extra *character* in the program. If you had to invert the output signals electronically, you would need additional ICs. Software gives us a lot of flexibility in how we control external devices without complicating the MCUs surrounding circuits. You can find the common-anode test code, **Program 7.1a**, in the Experiment 7 software folder.

If you make the connections to test a common-anode 7-segment display, please rewire your circuit for the common-cathode display shown earlier in ***Figure 7.2****. You will need this circuit in steps that follow.*

Step 8.

Most numeric or alphanumeric LED displays in weigh scales, digital thermometers, and electric meters need more than one digit. The 7-segment display used earlier in this experiment required seven outputs from the MCU, one per segment. You might think each added 7-segment display likewise would need its own seven MCU outputs. In fact, each added display requires only one extra output because we take advantage of a technique called *multiplexing*.

In a *multiplexed* 2-digit display, for example, all seven of the segment-control signals connect in parallel to both displays. That means all the *a* segments connect to one signal, all the *b* segments connect to a single wire, and so on. But each display has its own connection to ground, as shown in **Figure 7.5**. The switch connection to ground for each display simplifies the following circuit description.

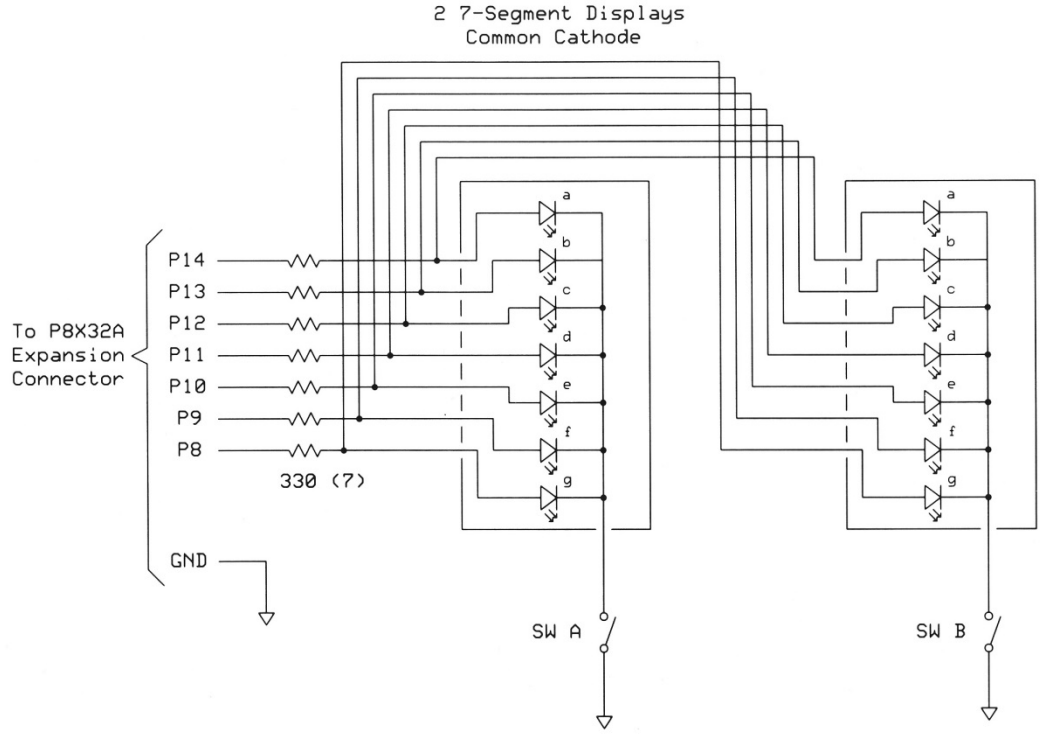


Figure 7.5.
A multiplexed 2-digit 7-segment display. The corresponding segments connect to each other; a to a, b to b, and so on. The common cathode for each display connects to ground through a switch. Only one switch at a time connects to ground.

Assume the MCU produces the pattern of logic-0 and logic-1 signals to display the numeral 6. When you close Switch A (SW A), current flows through Display A and you see it numeral 6. No current flows through Display B because it has no connection to ground. It remains dark. You turn off Switch A and have the

computer produce the bit pattern for the numeral 1. Then you close Switch B and Display B shows a 1. Because Display A now has no connection to ground, it appears dark.

Suppose the computer could control the two switches and synchronize their operation with the bit patterns for 6 and 1. If the switching occurs faster than about 60 times per second, the display appears to show 61. Below a switching frequency of 60 Hz, the human eye starts to detect a flicker in a display.

Obviously, the computer cannot reach out and operate switches, so we need an electronic "switch" to control the common-cathode connection to ground for each of the two displays. And we also need software to synchronize LED operations and switch operations to ensure the numerals appear on the proper display.

You might think an output on the Propeller IC could serve as a way to provide a ground for one 7-segment display. The MCU could switch this pin to a logic-1 to block current flow and change it to a logic-0 to let current flow to ground. That approach would work in theory, but not in practice. The amount of current for seven lit segments could easily exceed the amount an I/O pin could carry, and the MCU would suffer damage.

Instead of using a digital output on the Propeller MCU to sink current to ground, circuit designers use a device such as a 16-pin ULN2003A high-voltage, high-current Darlington transistor array. That sounds complicated, but for now just think of it as a switch that connects a "load," such as LEDs, to ground. Each of the seven switches in a ULN2003A IC can handle about half an ampere, or 500 mA; plenty of current for LEDs. **Figure 7.6** shows the internal circuits for a ULN2003A IC. The left-hand diagram (**a**) shows the complete circuit as you would see it in a data sheet. The right-hand diagram (**b**) shows a simplified view with only the symbols for seven inverters. When an input has a logic-1 applied to it, the corresponding output sinks current to ground. A logic-0 at an input causes the output to disconnect from ground. Experiments will use the simple diagram to show ULN2003A IC functions.

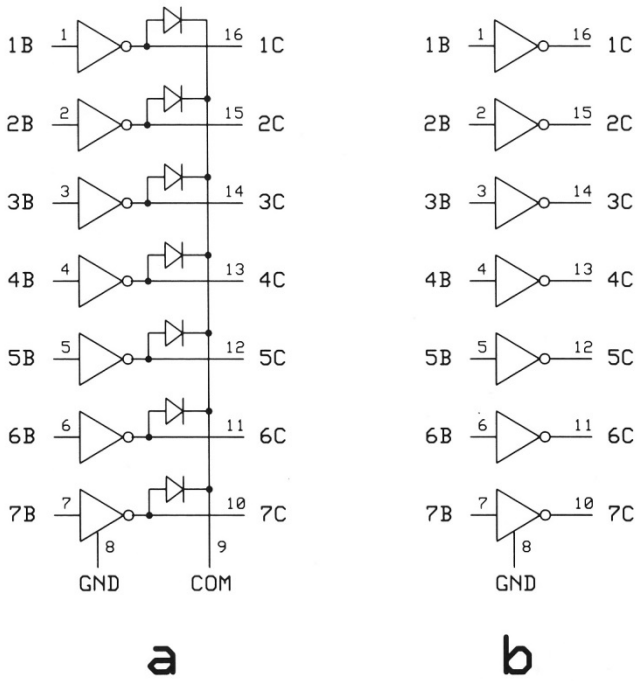


Figure 7.6.
This diagram shows a complete ULN2003A (**a**) as it appears in a manufacturer's data sheet. The simplified diagram (**b**) shows the primary functions that act as electronic switches that make a connection to ground for high-current loads.

Step 9.

In this step you will add a second common-cathode 7-segment display to your breadboard and use a ULN2003A IC to separately control each display. Wire your circuit as shown in **Figure 7.7**.

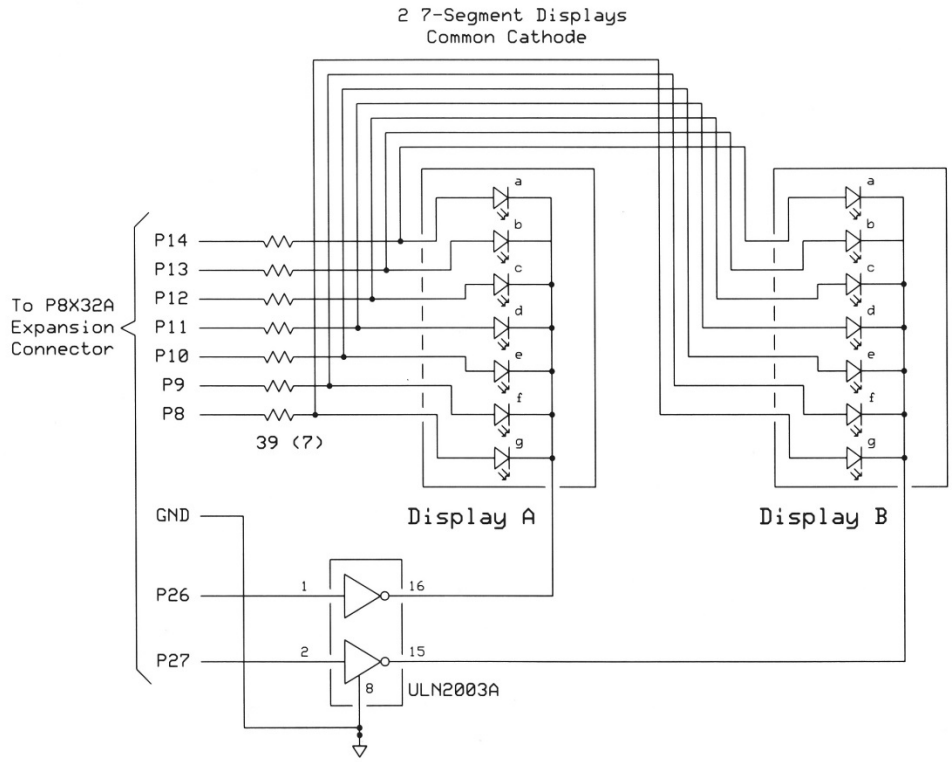


Figure 7.7.
This circuit provides two 7-segment displays with the *a* through *g* segments wired in parallel, but with a separate common-cathode control for each display. The ULN2003A IC requires only a ground connection. It does not connect to power.

The flow chart in **Figure 7.8** illustrates how a test program would multiplex two digits on two displays. The program would turn on each display and supply the appropriate digit.

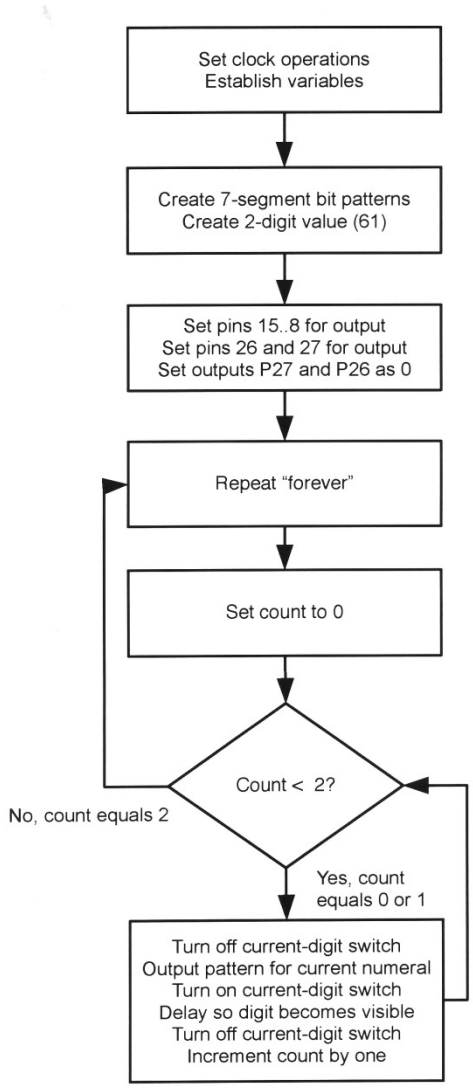


Figure 7.8.
Flow chart for a test program that would multiplex two digits, 6 and 1, on two 7-segment displays. The 7-segment pattern appears on the P15 through P8 outputs. The P26 and 27 signals control the two displays.

Program 7.2 uses the same 7-segment patterns shown in **Program 7.1**. Instead of showing all 10 numeric patterns, it displays two digits preset in array elements `digit[0]` and `digit[1]`. The software uses the values stored in those two elements to direct the Propeller to the pattern to display:

```
outa[15..8] := segment_code[digit[0]]
```

Program 7.2.

```
{{
''*********************************************
''*  Program 7.2
''*  Author: Jon Titus 11-03-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  7-Segment code-test program.
''*  Display two digits with 2 multiplexed
''*  common-cathode 7-segment displays.
```

```
''*  A ULN2003A controls the cathode on each display.
''*
''*********************************************
}}

CON _clkmode = xtal1 + pll16x        'Set MCU clock operation
    _xinfreq = 5_000_000             'Set for 5 MHz crystal

VAR
    byte segment_code[10]            'Storage for segment codes
    byte digit[2]                    'Storage for two digits

PUB Start                            'Main program starts here

segment_code[0] := %01111110         'Pattern for "0"
segment_code[1] := %00110000         'Pattern for "1"
segment_code[2] := %01101101         'Pattern for "2"
segment_code[3] := %01111001         'Pattern for "3"
segment_code[4] := %00110011         'Pattern for "4"
segment_code[5] := %01011011         'Pattern for "5"
segment_code[6] := %00011111         'Pattern for "6"
segment_code[7] := %01110000         'Pattern for "7"
segment_code[8] := %01111111         'Pattern for "8"
segment_code[9] := %01110011         'Pattern for "9"

digit[0] := 6                        'First digit to display
digit[1] := 1                        'Second digit to display

dira[15..8] := %11111111             'Set 8 output pins for
                                     ' segments
dira[27..26] := %11                  'Set 2 outputs to control
                                     ' cathodes
outa[26..27] := %00                  'Turn off cathodes to start

  repeat                                      'Main loop
    outa[15..8] := segment_code[digit[0]]     'Get segment pattern
                                              'for 1st digit
    outa[26] := %1                            'Turn on Display A
    waitcnt(clkfreq + cnt)                    'Delay
    outa[26] := %0                            'Turn off Display A
    outa[15..8] := segment_code[digit[1]]     'Get segment pattern
                                              'for 2nd digit
    outa[27] := %1                            'Turn on Display B
    waitcnt(clkfreq + cnt)                    'Delay
    outa[27] := %0                            'Turn off Display B
```

The program uses a single loop to display the 7-segment pattern for the `digit[0]` value and then for the `digit[1]` value. Run this program to control the displays as shown in **Figure 7.7**. What did you observe? If you did not see the "6" and "1" numerals on the displays, recheck your program and wiring. It's easy to have a wire "off by one position" in a breadboard. When I first wired and tested the circuit I had connected a Propeller output to the decimal-point connection instead of to segment B.

Now that you see the slow multiplexing operation, you can increase the multiplexing frequency so the two digits appear simultaneously. Make the following change to *both* `waitcnt...` statements in **Program 7.2** as shown here:

```
waitcnt(clkfreq + cnt)   to   waitcnt(clkfreq/60 + cnt)
```

Now the program causes the display to switch between the "6" and the "1" about 30 times a second. Increase the divisor by 10 and run the program again. Repeat until you increase the divisor enough so you observe a "flicker" free value. The flicker rate equals half of the divisor. So a divisor of 90 creates about a 45 Hz flicker. I found a value of 110 to 120 made the display look right. Also, change the value of the `digit[0]` and `digit[1]` variables to see your own displayed digits. Keep the value between 0 and 9 for each digit.

Did you notice the intensity of the displayed digits decreased when you increased the multiplex rate for flicker-free operation? Why would this happen? In the original test of two displays, each remained on for about a second, so you saw it at full brightness, but each display remained on for only half the total display time. When you increase the multiplex rate for flicker-free operation, each 7-segment display remains on again for only half the total display time, so it appears "half bright." Remember how the pulse-width-modulation operations introduced in Experiment 6 controlled LED brightness? The same thing happens here.

Optional: Program 7.2 uses one loop that assumes we always want to display two digits. The program also has two `waitcnt` statements that perform the same operation for each digit. Professional programmers aim to avoid duplication of code and "hard coding" a condition limited to, say, two digits. When the US price of gasoline rose above $US1 per gallon in 1980, gas-pump manufacturers had to go to each gas station and modify the mechanical displays of cost and total price to accommodate another digit. Let's avoid that sort of thing in our software!

Can you suggest a way to change **Program 7.2** to eliminate one of the delay statements and let the program handle as many as eight digits? Hint: Use two loops. Find one solution in the Answers section at the end of this experiment.

Step 10.

Program 7.2 works well as long as you know in advance what you want to display. But suppose we need a 2-digit counter that will show consecutive values from 00 to 99. How do we accomplish that? You can use two methods:

1. Set up an inner loop to count the units and an outer loop to count the 10's.

2. Create one loop that counts from 00 to 99 and add code that separates that value into individual and 10's and units characters.

The flow chart in **Figure 7.9** shows how this portion of a larger program operates. The box with the dashed lines represents the display delay loop. For the sake of clarity I excluded all of the display-control steps. You can run the complete program, saved as **Program 7.3.**

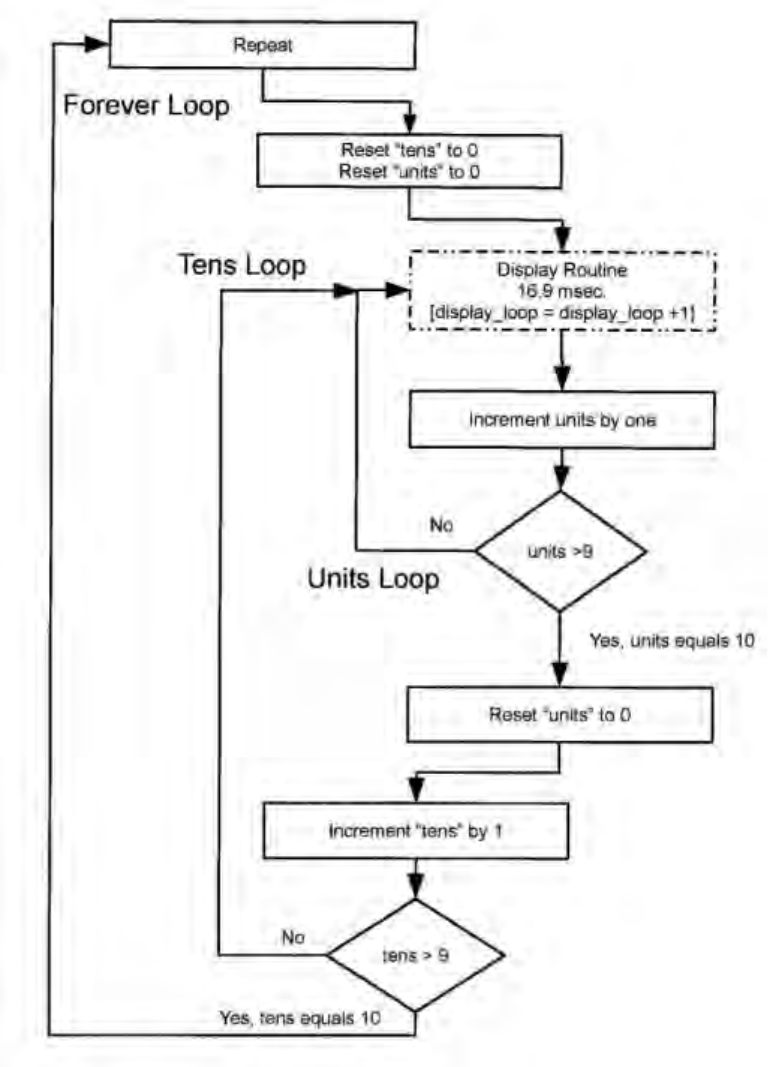


Figure 7.9.
This flow chart shows the "nested" arrangement of the units and tens repeat loops to create a counter that displays values from 00 to 99.

Program 7.3.

```
{{
''*********************************************
''*  Program 7.3
''*  Author: Jon Titus 11-03-2014 Rev. 2
''*  Copyright 2013
''*  Released under Apache 2 license
''*  Display counter demonstration that
''*  goes through count 00 to 99 on two
''*  7-segment displays. A logic-1 output on
''*  P26 or P27 turns on respective display.
''*  LDS-C512RI display with 39-ohm resistors
''*  ULN2003A driver for common cathodes
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
```

```
    _xinfreq = 5_000_000          'Set for 5 MHz crystal

VAR
    byte segment_code[10]         'Storage for segment codes
    byte count
    byte tens
    byte units
    byte display_loop
    byte digit[2]                 'Variable for loop, 2 digits

PUB Start                         'Main program starts here

segment_code[0] := %01111110      'Pattern for "0"
segment_code[1] := %00110000      'Pattern for "1"
segment_code[2] := %01101101      'Pattern for "2"
segment_code[3] := %01111001      'Pattern for "3"
segment_code[4] := %00110011      'Pattern for "4"
segment_code[5] := %01011011      'Pattern for "5"
segment_code[6] := %00011111      'Pattern for "6"
segment_code[7] := %01110000      'Pattern for "7"
segment_code[8] := %01111111      'Pattern for "8"
segment_code[9] := %01110011      'Pattern for "9"

digit[0] := 0                     'Storage for 10s digit
digit[1] := 0                     'Storage for 1s digit

dira[15..8] := %11111111          'Set eight output pins for
                                  'display
dira[27..26] :=%11                'Set display-switch outputs
outa[26] := 0                         'Clear display-switch outputs
outa[27] := 0
  repeat                              'Main loop
     repeat tens from 0 to 9          'Loop for 10's digit
         digit[0] := tens
         repeat units from 0 to 9     'Loop for units digit
             digit[1] := units

             'Timing and display loop
             repeat display_loop from 0 to 59
                 repeat count from 0 to 1
                    outa[26+count] := 0
                    outa[15..8] := segment_code[digit[count]]
                    outa[26+count] :=1
                    waitcnt(clkfreq/120 + cnt)
                    outa[26+count] := 0
```

Step 11.

The nested loops for units and tens explained in the previous step work well if you simply count in sequence and can place the ten's and unit's counters in separate loops. But you must add other loops to handle hundreds, thousands, and so on. That's not a lot of fun, and it produces inflexible software, too. In some cases you just cannot take that approach. You might have a single value called, say, `big_counter` that can hold values between 0 and 99999. In that case a program must separate, or parse, a value such as 5329 into separate digits before it can display them on multiplexed 7-segment displays.

For a given value, a few math operations let us obtain the separate numerals to display. You can apply this technique, or algorithm, to larger or smaller numbers. The following example assumes you already have a 4-digit value to work with: 5329.

To get the first digit, 5, you divide 5329 by 1000. The answer from a calculator comes to 5.329. But a Propeller MCU works with fixed-point values. That means it cannot handle a value such as 5.329 that includes a decimal fraction such as 0.329. The MCU operates on whole numbers only; for example, 4, 784, 1672, -7912, -3, and so on. So on the Propeller, dividing 5329 by 1000 yields the answer 5. The decimal fraction gets "dropped."

Now you have the first digit 5 and have saved it. Next, multiply 5 by 1000 to get 5000. Then subtract 5000 from 5329 to get 329. Now divide 329 by 100 to yield 3, which you save as the second digit. Multiply 3 by 100 to get 300, and subtract the 300 from 329. That leaves 29.

What would you do next to separate the last two digits? Find the answer at the end of this experiment.

For our display program, the lack of decimal fractions in the Propeller MCU works in our favor. If you must work with decimal fractions, use a set of floating-point-math objects, available in the Parallax Object Exchange library.

Step 12.

The code in **Program 7.4** shows how a repeat loop can take a 4-digit number and "separate" it into four digits that you can display. I'll explain the software one step at a time. In this explanation I set the `big_value` to 7286 and the `numb_of_digits` to 4. You could create a loop to determine the number of digits in a value, but that goes beyond the scope of this experiment. After you take a quick look at the code, read explanations of the three loops that follows.

Program 7.4.

```
{{
''*********************************************
''*  Program 7.4
''*  Author: Jon Titus 11-03-2014 Rev. 1
''*  Copyright 2013
''*  Released under Apache 2 license
''*  Display counter demonstration that
''*  separates a 4-digit value into separate
''*  numerals and displays the first two digits or
''*  the last two digits on 7-segment displays.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

VAR
    byte segment_code[10]          'Storage for segment codes
    byte count                     'Counter used in display loop
    byte display_loop              'Counter used for display-
                                   'loop delay
    byte digit[6]                  'Space for 6 digits
    long big_value                 'Variable for large value to
                                          'parse
    byte numb_of_digits            'Number of digits in
                                   'big_value
```

```
    long divisor                  'Divisor used to start
                                  'parsing
    byte index                    'Index to identify each digit
    byte decimal_counter          'Counter to track each
                                  'decimal digit

PUB Start                         'Main program starts here

segment_code[0] := %01111110      'Pattern for "0"
segment_code[1] := %00110000      'Pattern for "1"
segment_code[2] := %01101101      'Pattern for "2"
segment_code[3] := %01111001      'Pattern for "3"
segment_code[4] := %00110011      'Pattern for "4"
segment_code[5] := %01011011      'Pattern for "5"
segment_code[6] := %00011111      'Pattern for "6"
segment_code[7] := %01110000      'Pattern for "7"
segment_code[8] := %01111111      'Pattern for "8"
segment_code[9] := %01110011      'Pattern for "9"

digit[0] := 0                     'Storage for 10s
digit[1] := 0                     'Storage for 1s

dira[15..8] := %11111111          'Set 8 pins for display
dira[27..26] :=%11                'Set display-switch outputs
outa[26] := 0                     'Clear display-switch outputs
outa[27] := 0                     'for control of 2 7-segment
                                  'displays
big_value := 7286                 'Large value to work with
numb_of_digits := 4               'Number of digits in
                                  'big_value
divisor := 1                      'Starting value of divisor

'LOOP 1: Given the number of digits, calculate the
'largest divisor.
'For four digits, start with 1000, for example.

  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10


'LOOP 2: Use divisor to separate the numerals. Use '(numb_of_digits - 1) so
loop can go from digit[0]
'to digit[3] for a 4-digit value.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10

 'LOOP 3: OK, big-value separated, display the first 2 digits
 '"forever." To display the last 2 individual digits,
 'add two to the count in this statement:
 '  outa[15..8] := segment_code[digit[count+2]] in the code.

  repeat
    repeat display_loop from 0 to 59
        repeat count from 0 to 1
```

```
            outa[26+count] := 0
            outa[15..8] := segment_code[digit[count]]
            outa[26+count] :=1
            waitcnt(clkfreq/120 + cnt)
            outa[26+count] := 0
```

Loop 1: This sequence calculates the divisor value to use when software starts to parse `big_value`.

```
numb_of_digits := 4
divisor := 1

  repeat decimal_counter from 1 to (numb_of_digits - 1)
       divisor := divisor * 10
```

When this loop starts, it subtracts 1 from the number of digits `(numb_of_digits - 1)` to determine how many times it should repeat. For our 4-digit value, the loop runs three times and because it starts with a `divisor` of 1, the final `divisor` value comes to 1000.

Loop 2: This short loop parses the 4-digit number into four separate values and saves each value separately in the `digit[x]` array.

```
repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10
```

The `index` value increments 0, 1, 2, 3, so the loop operates four times; once per digit in the value 7286. During the first pass through the loop, `big_value` (7286) gets divided by 1000 and the result (7) goes into `digit[0]`. Next, `digit[0]` gets multiplied by the `divisor` (7 * 1000) and subtracted from `big_value` (7286 - 7000) to make `big_value` now equal 286. At the end of the loop, the `divisor` value (1000) gets divided by to 10 yield 100. As the loop runs, the index array becomes:

```
index[0]  =  7, index[1]  =  2, index[2]  =  8, and index[3]   =  6.
```

Loop 3: This final set of three `repeat` loops displays the first two digits, 7 and 2. If you want to display the last two digits, replace the statement:

```
outa[15..8] := segment_code[digit[count]]
```

with:

```
outa[15..8] := segment_code[digit[count+2]]
```

in the following section of code:

```
repeat
    repeat display_loop from 0 to 59
        repeat count from 0 to 1
            outa[26+count] := 0
            outa[15..8] := segment_code[digit[count]]
            outa[26+count] :=1
            waitcnt(clkfreq/120 + cnt)
            outa[26+count] := 0
```

Could you modify the Loop 3 code and your display hardware to show all four digits?

Step 13 - *Optional*.

Instead of connecting the corresponding segment anodes in parallel for four separate displays, which could create quite a wiring mess, you can buy 4-digit modules with those connections made internally. The LUMEX LDQ-N516RI provides a good example. This module (**Figure 7.10**) requires only 12 pins: eight for the display segments and decimal point, and four for the cathode from each numeral. The schematic diagram in **Figure 7.11** shows the connections between this module and a P8X32A board. The circuit still needs current-limiting resistors – one per segment. The ULN2003A switches the four cathodes.

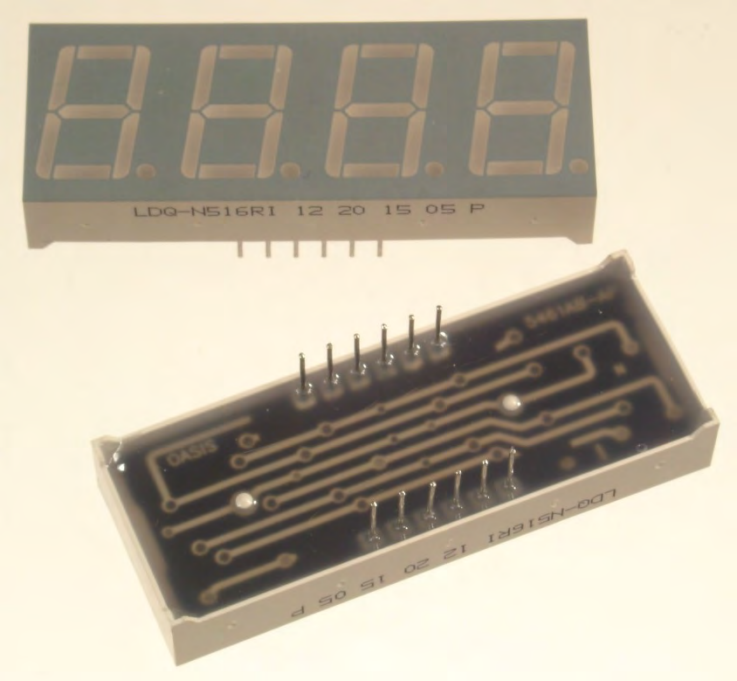


Figure 7.10.
A 4-digit 7-segment display simplifies wiring to eight LED-control inputs and four common-cathode connections. This display includes decimal points. The module dimensions make it easy to place modules horizontally, end-to-end when a design needs more digits. The upside-down display reveals internal connections between segment LEDs. This photo shows a LUMEX LDQ-N516RI module.

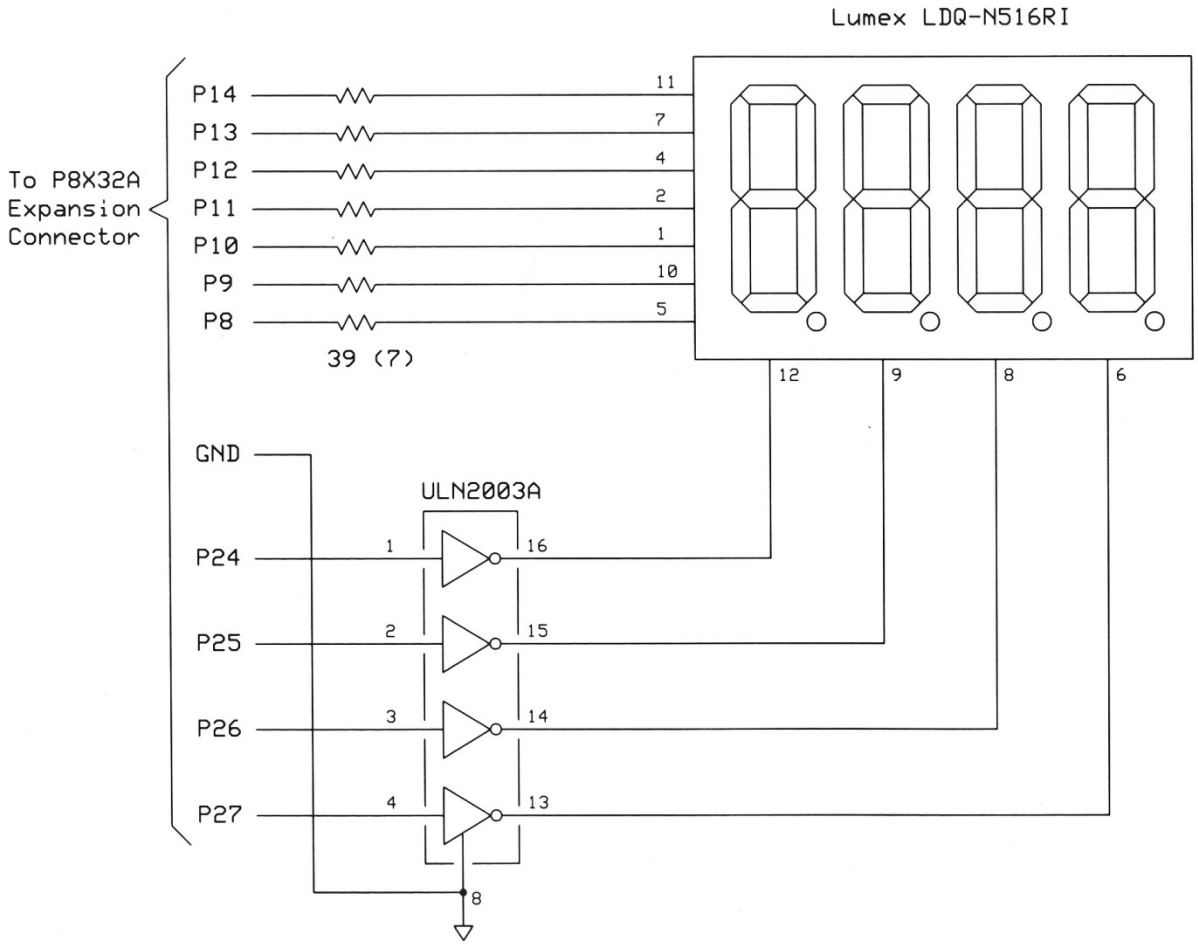


Figure 7.11.
Schematic diagram for a 4-digit display module that shows the simplified wiring. Internal connections link the same segment in each numeral to a single pin. Each numeral requires a switch that connects to ground. This circuit does not use the decimal points in the display module.

Program 7.5 shows a modified version of **Program 7.4** that controls the display and show four digits. You will find this program in the Experiment 7 folder.

Program 7.5.

```
{{
''*********************************************
''*  Multiplexed Display Program 7.5
''*  Author: Jon Titus 11-03-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display counter demonstration that
''*  separates a 4-digit value into separate
''*  numerals and displays four digits on a
''*  LUMEX LDQ-N516RI 4-digit display module.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x    'Set MCU clock operation
    _xinfreq = 5_000_000         'Set for 5 MHz crystal
```

```
VAR
    byte segment_code[10]        'Storage for segment codes
    byte count                   'Counter used in display loop
    byte display_loop            'Counter used for display-loop
                                       ' delay
    byte digit[6]                'Space for 6 digits
    long big_value               'Variable for large value to
                                 ' parse
    byte numb_of_digits          'Number of digits in big_value
    long divisor                 'Divisor used to start parsing
    byte index                   'Index to identify each digit
    byte decimal_counter         'Counter to track each decimal
                                 ' digit

PUB Start                        'Main program starts here

segment_code[0] := %01111110     'Pattern for "0"
segment_code[1] := %00110000     'Pattern for "1"
segment_code[2] := %01101101     'Pattern for "2"
segment_code[3] := %01111001     'Pattern for "3"
segment_code[4] := %00110011     'Pattern for "4"
segment_code[5] := %01011011     'Pattern for "5"
segment_code[6] := %00011111     'Pattern for "6"
segment_code[7] := %01110000     'Pattern for "7"
segment_code[8] := %01111111     'Pattern for "8"
segment_code[9] := %01110011     'Pattern for "9"

digit[0] := 0                    'Storage for left-most digit
digit[1] := 0
digit[2] := 0
digit[3] := 0                    'Storage for right-most digit

dira[15..8] := %11111111         'Set eight output pins for
                                 'the display
dira[27..24] := %1111            'Set display-switch outputs
outa[27..24] := %0000            'Clear display-switch outputs
                                 'for control of 4 7-segment
                                 'displays
big_value := 7286                'Large value to work with
numb_of_digits := 4              'Number of digits in big_value
divisor := 1                     'Starting value of divisor

'LOOP 1: Given the number of digits, calculate the largest
'  divisor. For four digits, start with 1000, for example.

  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10


'LOOP 2: Use divisor to separate the numerals. Use '  (numb_of_digits - 1)
'  so loop can go from digit[0] to digit[3] for a 4-digit
'  value, which saves four values.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
```

```
    divisor := divisor / 10

 'LOOP 3: OK, big-value separated, display the first 2 digits
 '  "forever." To display the last 2 individual digits,
 '  add two to the count in:
 '  outa[15..8] := segment_code[digit[count+2]] statement
 '  in the code.

  repeat
    repeat display_loop from 0 to 59
        repeat count from 0 to 3
            outa[24+count] := 0
            outa[15..8] := segment_code[digit[count]]
            outa[24+count] :=1
            waitcnt(clkfreq/240 + cnt)
            outa[24+count] := 0
```

Note the `waitcnt` statement causes a shorter delay. If you use 120 instead of 240, the display appears to flicker.

Answers

Experiment 7, Step 2:

The following codes will work in **Program 7.1** for a 7-segment common-cathode display properly wired as shown in **Figure 7.2**.

```
segment_code[0] := %01111110        'Pattern for "0"
segment_code[1] := %00110000        'Pattern for "1"
segment_code[2] := %01101101        'Pattern for "2"
segment_code[3] := %01111001        'Pattern for "3"
segment_code[4] := %00110011        'Pattern for "4"
segment_code[5] := %01011011        'Pattern for "5"
segment_code[6] := %00011111        'Pattern for "6"
segment_code[7] := %01110000        'Pattern for "7"
segment_code[8] := %01111111        'Pattern for "8"
segment_code[9] := %01110011        'Pattern for "9"
```

Experiment 7, Step 9:

You can simplify the software, make it easier to modify, and make it easier for other programmers to understand. **Program 7.2X** in the Experiment 7 folder shows you what I did:

Program 7.2X

```
{{
''*********************************************
''*  Program 7.2X
''*  Author: Jon Titus 11-03-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  7-Segment code-test program.
''*  Display two digits with 2 multiplexed
''*  common-cathode 7-segment displays.
''*  A ULN2003A controls the cathode on each display.
''*
''*********************************************
}}
```

```
CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

VAR
    byte segment_code[10]          'Storage for segment codes
    byte digit[8]                  'Storage for as many as 8
                                   'digits
    byte digit_count               'Number of digits to display
    byte counter                   'Loop counter for digit
                                   'display
    byte time_factor               'Timing value for
                                   'multiplexing

PUB Start                          'Main program starts here

segment_code[0] := %01111110       'Pattern for "0"
segment_code[1] := %00110000       'Pattern for "1"
segment_code[2] := %01101101       'Pattern for "2"
segment_code[3] := %01111001       'Pattern for "3"
segment_code[4] := %00110011       'Pattern for "4"
segment_code[5] := %01011011       'Pattern for "5"
segment_code[6] := %00011111       'Pattern for "6"
segment_code[7] := %01110000       'Pattern for "7"
segment_code[8] := %01111111       'Pattern for "8"
segment_code[9] := %01110011       'Pattern for "9"

digit[0] := 6                      'First digit to display
digit[1] := 1                      'Second digit to display
                                   'Eight digits maximum

digit_count := 2                   'Number of digits to display,
                                   '1 to 8 only

time_factor := 1                   'Change as needed, 1 to
                                   '255 only

dira[15..8] := %11111111           'Set output pins for segments
dira[27..26] := %11                'Set 2 outputs to control
                                   'cathodes
outa[26..27] := %00                'Turn off cathodes to start

  repeat                           'Main loop

     'Set upper digit count
     repeat counter from 0 to digit_count-1
        'Get segment pattern for digit
        outa[15..8] := segment_code[digit[counter]]
        outa[26 + counter] := %1              'Turn on display
        waitcnt(clkfreq/time_factor + cnt)    'Delay
        outa[26 + counter] := %0              'Turn off display
```

The program includes new or changed variables:

```
byte digit[8]         'Storage for as many as 8 digits
byte digit_count      'Number of digits to display
byte counter          'Loop counter for digit display
```

```
byte time_factor      'Timing value for multiplexing
```

The `digit[8]` variable sets up an array with as many as eight bytes, `digit[0]` through `digit[7]`, the maximum we expect to display. So a programmer could adjust the code to accommodate more or fewer as needed.

The `digit_count` establishes the number of digits to display. In our case, two.

The `counter` variable will keep track of the number of times our code passes through a loop.

The `time_factor` variable lets us set a specific value that determines the time each digit in our display will remain lit.

Now on to the main `repeat` loop:

```
repeat counter from 0 to digit_count-1    'Set upper digit count
    'Get segment pattern for digit
    outa[15..8] := segment_code[digit[counter]]
    outa[26 + counter] := %1               'Turn on display
    waitcnt(clkfreq/time_factor + cnt)     'Delay
    outa[26 + counter] := %0               'Turn off display
```

The first statement sets the starting value for the `digit[0]` array element to display. The loop counter, `counter`, will go from 0 through the value `digit_count - 1`. Why subtract 1 from the `digit_count`? The count ranges from 1 through 8, but the array elements range from `digit[0]` through `digit[7]`. The subtraction properly offsets the `counter` value to correspond to the proper array element.

The second statement places the segment code for the value saved in `digit[counter]` on the output pins. In this case the software uses the `count` value to identify successive elements in the `digit[x]` array

The third and fifth statements control the cathode connection to ground through the ULN2003A IC to turn a display on or off.

The fourth statement:

```
waitcnt(clkfreq/time_factor + cnt)
```

includes a divisor, `time_factor`, which lets us set a value for timing in one place in the code. To start, the program uses a value of 1 for `time_factor`, but you can change it as noted in the comments.

Experiment 7, Step 11:

When you have separated the value 3, and subtract 300 form 329, you have 29. Divide this by 10 to yield the answer 2. Save this as the third digit. Then multiply 3 by 10 to get 30. Subtract 30 from 39 and you have the fourth digit, 9.

Experiment 7, Step 12:

You would need to add two 7-segment displays with parallel connections to the segment pins on the two displays already in your breadboard. You also must add two more switches to your circuit (already in the ULN2003A IC) to switch the cathode of each added display to ground.

To keep the code roughly similar to that used in **Program 7.4**, I recommend you use outputs P24 to control the display for the 1000's, P25 for the 100's, P26 for the 10's, and P27 for the 1's. Then your `display-loop` code would look like:

```
repeat
    repeat display_loop from 0 to 59
        repeat count from 0 to 3
            outa[24+count] := 0
            outa[15..8] := segment_code[digit[count]]
            outa[24+count] :=1
            waitcnt(clkfreq/120 + cnt)
            outa[24+count] := 0
```

Experiment No. 8 – How to Use Serial Communications to Control LED Displays

Abstract

In Experiment 7 you learned how to use a multiplexing circuit to control 7-segment displays. To control four displays you needed 11 output pins on the Parallax Propeller P8X32A microcontroller board. In this experiment you will learn how to use serial communications and external integrated circuits to control LEDs. In some cases, you can control four 7-segment displays with only three wires.

Keywords

shift register, 74HC595, serial, LED, display, flip-flop, ULN2003A, multiplex, transistor array, MCU

Requirements

(1) - 74HC595 8-bit serial-in, parallel-out shift register
(1) - ULN2003A high-current transistor array, 16-pin DIP
(8) - 39-ohm, 1/4W, 5% resistors (orange-white-black)
(1) - 1000-ohm, 1/4W, 5% resistor (brown-black-red)
(8) - LEDs, any color or a 10-segment LED bar-graph 20-pin DIP
(1) - Lumex LDQ-N516RI 4-digit module, common cathode, or equivalent
(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - USB cable

Introduction

Circuit designers want to have many microcontroller I/O pins available to control external devices, to obtain information from sensors, and to transmit and receive information from other equipment. In some cases, you can make more I/O pins available at the cost of adding a few external integrated circuits (ICs) to a project. Manufacturers have understood this need for many years and they manufacture chips that provide a standard interface, or interfaces, that simplify software and hardware in a system based on a microcontroller (MCU).

Step 1.

This experiment will show you how an IC called a *shift register* can reduce the number of connections between an MCU and LED displays. To control the LEDs in a 7-segment display, for example, a shift register needs only two or three connections with an MCU. Before you start to wire a circuit, you must understand how a shift register works. The 74HC595 8-bit shift register provides a good example. You can find the Texas Instruments data sheet for this IC at: http://www.ti.com/lit/ds/symlink/sn74hc595.pdf.

Basically a shift register accepts binary information, one bit at a time and moves the bits left or right, one at a time. Imagine a bucket brigade of eight people each with a clear-plastic pail. You put a red ball in the bucket for the first person in the line. When you blow a whistle, the first person puts the ball in the second person's bucket. Each time you blow the whistle, the ball moves along the line, one bucket at a time. At any time you can look at all eight buckets and see the position of the red ball.

Suppose you start the same way, with a red ball in the first bucket. When you blow the whistle and the red ball gets moved to the next bucket, you drop a blue ball into the empty first bucket. Blow the whistle again and both balls move one bucket down the line. Again, you can see the positions of the balls at any time.

A shift-register IC does the same thing but with logic 0's and logic 1's. The diagram in **Figure 8.1** shows a simple 8-bit shift register that has one serial input and eight parallel outputs. (For the sake of clarity, the diagram does not show the output connections, but you can see the bit in each position.)

A clock pulse, analogous to the whistle, causes the binary information to shift into the shift register one bit at a time. The clock also causes each bit already in the shift register to move one position to the right.

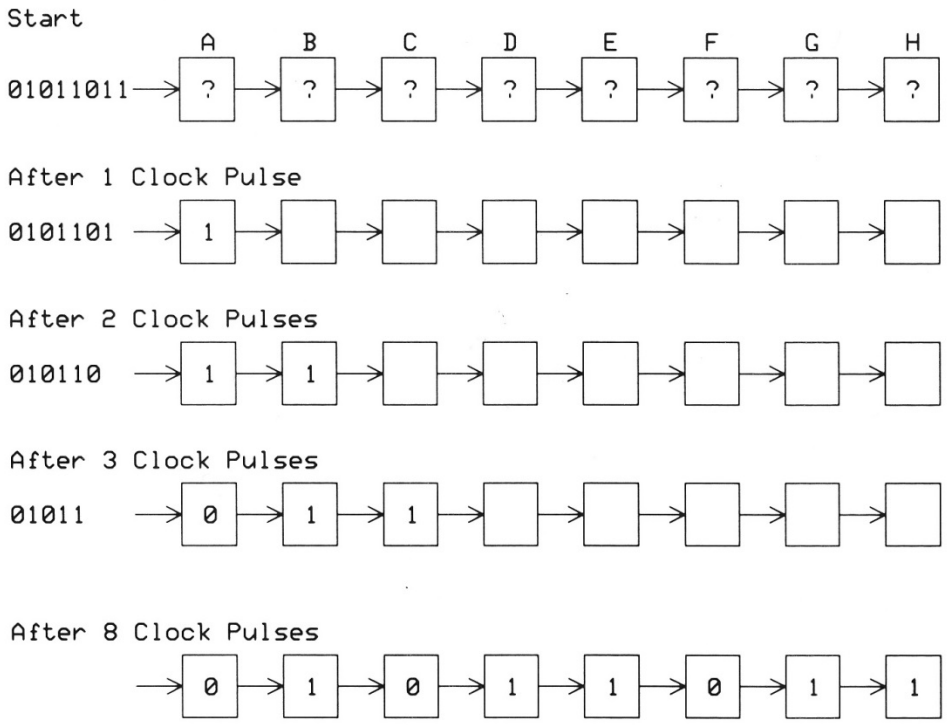


Figure 8.1.
A simple serial-in, parallel-out shift register with the state shown after 1, 2, 3, and 8 clock pulses. Each of the eight temporary-storage locations has an electrical output (not shown for clarity).

With only two wires from an MCU – a clock signal, and a series of logic-0 and logic-1 bits on another signal – the shift register will provide eight outputs in parallel. You could use these eight outputs to control the *a* through *g* segments and a decimal point in a 7-segment display. Instead of eight outputs from an MCU, you need only two.

You might wonder how a shift register holds the 0's and 1's. It uses a flip-flop that serves as a 1-bit "memory" circuit. Once you clock a bit into a flip-flop, it holds it until you shift in a new bit or until you remove power. **Figure 8.2** shows a block diagram for a simple flip-flop used in a 74HC174 IC, which contains six of these independent flip-flop circuits.

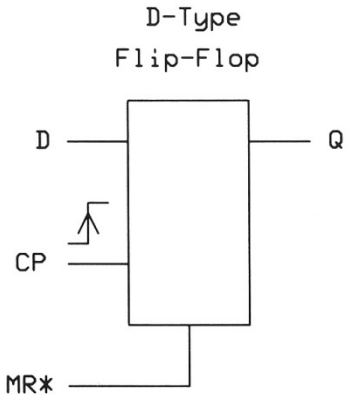


Figure 8.2.
Block diagram for a D-type flip-flop used in integrated circuits to temporarily hold one bit of information. The 74HC174 IC uses this type of flip-flop circuit. I added a symbol to indicate a positive-going pulse on the CP input causes data transfer from D to Q.

The flip-flop has three inputs, clock pulse (CP), data (D), and master reset (MR*), along with one output, Q. The asterisk after a signal name means a logic-0 causes the action the label refers to. Some diagrams might note MR* as MR/ or /MR where the slash also indicates a logic-0 causes an action to occur. The timing diagram in **Figure 8.3** shows how the flip-flop Q output responds to changes at the three inputs. This diagram might look complicated, but the explanation that follows points out important actions that occur at points labeled A through G.

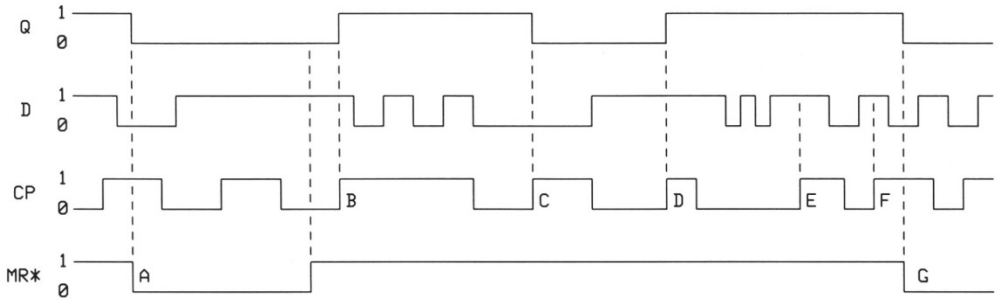


Figure 8.3.
Timing diagram for a positive-edge-triggered D-type flip-flop circuit.

1. **A -** When the master reset input (MR*) becomes a logic-0, it forces the Q output to a logic 0. During this reset time, actions on the data (D) and clock-pules (CP) inputs do not affect the Q output.

2. **B -** Here the MR* input has already become a logic-1, which has "released" the flip-flop from its reset mode. The positive-going edge (logic-0 to logic-1) on the CP input transfers the D-input state (logic-1) through to the Q output, which becomes a logic-1.

3. **C -** The positive-going edge of the clock signal transfers the logic-0 signal on the D input through to the Q output.

4. **D -** The positive-going edge of the clock signal transfers the logic-1 signal on the D input through to the Q output.

5. **E** and **F** - The positive-going edge of the clock signal transfers the logic-1 signal on the D input through to the Q output. But in both cases, the Q output already exists in a logic-1 state, so no change occurs on the Q output. It remains a logic-1.

6. **G** - Regardless of changes on the D or CP inputs, the MR* in a logic-0 state always overrides them and always resets the flip-flop. When reset, the flip-flop holds the Q output in the logic-0 state for as long as the MR* input remains at a logic-0.

To summarize:

1. When the CP input experiences a positive-going signal edge, it transfers the state of the D input to the Q output.

2. A logic-0 on the MR* input forces the Q output to a logic-0 and the flip-flop ignores the CP and D inputs.

A shift register such as the 74HC595 uses eight flip-flops connected "head to tail," and the clock-pulse input causes a new bit to shift into the first flip-flop as "old" data shifts one flip-flop down the "bucket brigade."

Step 2.

In this step you will connect eight LEDs to a 74HC595 shift register and control it with software. The two letters, HC, in the IC part number indicate a logic device that uses a semiconductor technology that lets it operate over a wide voltage range. The data sheet for a 74HC595 shows an operating-voltage range from 2 to 6 volts. This IC will work well with the 3.3-volt signals on a Propeller P8X32A board.

The schematic diagram in **Figure 8.4** shows the 74HC595 shift-register, eight LEDs, and current-limiting resistors. Wire this circuit on a solderless breadboard. Instead of eight LEDs, you could use eight LEDs in a bar-graph LED module. Remember to check the polarity of the LEDs as you place them in a breadboard. The signal names for the 74HC595 come from the Texas Instruments data sheet for this IC. Other manufacturers might use different signal names, but the function at each pin number remains the same. You will learn about the third signal, RCLK, in another step.

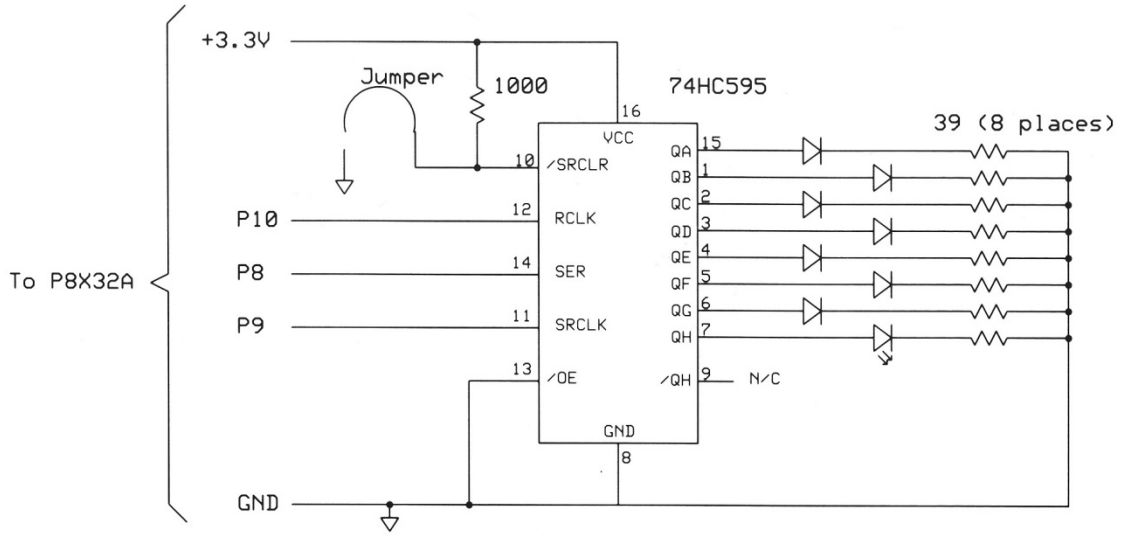


Figure 8.4.
Wiring diagram for a 74HC595 shift register IC that controls eight LEDs. The Propeller P8X32A board will supply the serial data and a clock signal created by software.

The 74HC174 flip-flop circuit described earlier transfers data from its D input to the Q output on a positive-going edge at CP input. The 74HC595 shown in **Figure 8.4** operates in a similar fashion. It transfers the logic state at the data input (SER, pin 14) into the shift register and to the corresponding Q output on the positive-going edge of the clock signal (SRCLK, pin 11). Software that controls the shift register must supply a logic-0 or a logic-1 bit at the SER pin and then provide a logic-1 pulse at the SRCLK input. This action shifts a bit into the first position in the shift register. Don't confuse the SRCLK input with the /SRCLR input.

Step 3.

After a short explanation of software operations, you will run a program and see how the shift register works. The code in **Program 8.1** takes eight bits of information, saved as variable `ser_data`, and determines the state (0 or 1) of the left-most, or most-significant bit (MSB). The Propeller SPIN instructions cannot test a specific bit, but you can use an `if` statement to determine if the `ser_data` value equals or exceeds 128_{10}, which means it has a value of 10000000_2 through 11111111_2. As long as the MSB equals 1, the software changes the state at the Propeller `data_port` output pin (pin P8) to a logic-1.

If `ser_data` has a value less than 128_{10}, or from 00000000_2 to 01111111_2, the MSB must be a 0. So the code sets the `data_port` output pin (pin P8) to a logic-0. Finally, the code briefly changes the `serial_clock` output pin (pin P9) to a logic 1 and back to a logic 0 to clock the data on the `data_port` pin into the shift register.

The following example shows the first few steps for the 8-bit binary value 01101011_2 in which I underline the most-significant bit. The decimal value in parentheses will help you interpret what happens. Note the shift operation moves the MSB to the left and inserts a zero on the right:

1. Does 0 1101011 (107) exceed 128? No. Send a zero to the shift register.
2. Shift 0 1101011 one bit position left: 1 1010110.
3. Does 1 1010110 (214) exceed 128? Yes. Send a one to the shift register.
4. Shift 1 1010110 one bit position left: 1 0101100.
5. Does 1 0101100 (172) exceed 128? Yes. Send a one to the shift register.
6. Shift 1 0101100 one bit position left: 0 1011000.
7. Does 0 1011000 (88) exceed 128? No. Send a zero to the shift register.

...and so on for the remaining four bits. At the end, `ser_data` equals 00000000_2.

Program 8.1 uses the statement:

```
ser_data := ser_data << 1
```

to shift all the bits in the `ser_data` variable one bit to the left. An `if` statement test this bit the next time through the loop.

Program 8.1.

```
{{
''*********************************************
''*  Shift-Register Program 8.1
''*  Author: Jon Titus 11-04-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Shift register control program uses
''*  74HC595 chip. P8 for data, P9 for clock
''*  P10 for register clock. This program shows
''*  how bits get shifted into the IC and displayed
''*  on eight LEDs.
''*********************************************
```

```
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

VAR byte ser_data                   'Byte storage
    byte data_port                  'Identify data output pin
    byte serial_clock               'Identify serial clock pin
    byte register_clock             'Identify register-clock pin

PUB Start                           'Main program starts here
  ser_data := %11101001             'Data to transmit
  dira[8..10] := %111               'Set output pins for shift-reg
  outa[8..10] := %000               'Clear output pins to logic-0
  data_port := 8                    'Pin P8
  serial_clock := 9                 'Pin P9
  register_clock := 10              'Pin P10

  outa[register_clock] := 0         'Strobe register clock to clear
  outa[register_clock] := 1         'the register
  outa[register_clock] := 0
  waitcnt(cnt + clkfreq)            '1-sec delay

  repeat 8                          'Go through this loop 8 times
    if (ser_data => 128)            'Test MS bit (MSB)
        outa[data_port] := 1        'If MSB = 1, so put it on
                                    'ser_data pin
    else
        outa[data_port] :=0         'If not, put 0 on ser_data pin
    ser_data := ser_data << 1       'Bit shift left ser_data bits
    outa[serial_clock]  := 1        'Shift into shift register
    outa[serial_clock]  := 0
    outa[register_clock] := 1       'Pass data to output register
    outa[register_clock] := 0
    waitcnt(cnt + clkfreq)          'wait for 1 sec.

  repeat                            'An infinite "do-nothing" loop
     ser_data := ser_data           'that stops the MCU until you
                                    'rerun the program
```

You may use the `ser_data` value already in the program, or substitute your own 8-bit binary value. After you have power attached to your breadboard, briefly connect the jumper at pin 10 on the 74HC595 to ground. This brief connection resets all eight positions in the shift-register IC to logic-0. When you run the program, all the LEDs will turn off to show this reset condition and after a short delay you should see your `ser_data` pattern appear one bit at a time on the LEDs. Try several patterns, but remember to use the jumper to reset the shift register before you re-run the program. Did you see the patterns you expected? If not, recheck your wiring and the program you ran.

Step 4.

You probably noticed the 74HC595 requires three connections to the Propeller MCU instead of the two described in Step 1. The shift-register electronics provides *two* flip-flops for each of the eight bit positions. The simple shift register shown previously in **Figure 8.1** lets you see the state of each register location as the shifting takes place. If the output from each shift-register location directly controlled a 7-segment display, you would see segments light up in odd patterns as the segment-control bits shifted into position. **Figure 8.5** shows the patterns you would see as the bits for the numeral 5 on a 7-segment display shifts into position. (Depending

on the frequency of the shift-register clock signal, SRCLK, your eye might not see each pattern, but some segments might appear turned on at a low light level.)

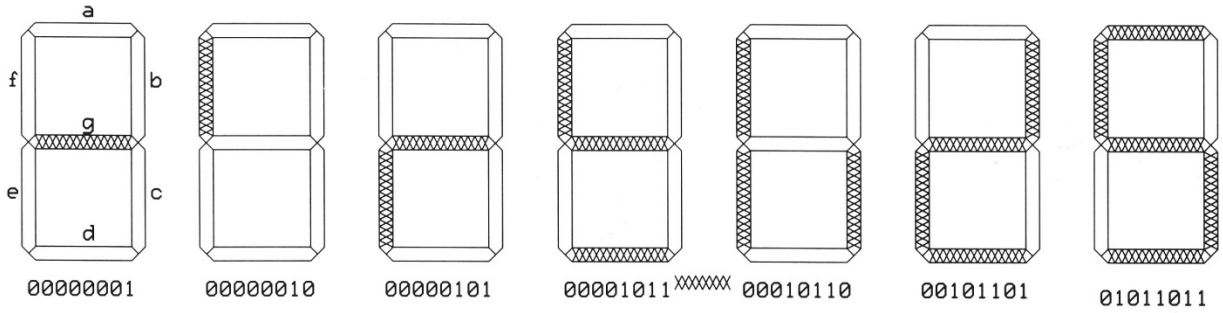


Figure 8.5.
Patterns you might see as segment-control bits for the numeral 5 shift slowly into position. The cross-hatch marks indicate a lit segment. Note the segment pattern "enters" at segment g and then shifts through the segments f, e, d, c, b, and a.

To prevent this type of visible action, after software shifts out the eight bits it creates a short logic-1 pulse for the RCLK input to the 74HC595 IC. This pulse causes a simultaneous transfers of all eight bits from the shift register to an output register. **Figure 8.6** illustrates the two flip-flops used for each shift-register position, or stage. By using two sets of flip-flops, you can shift eight bits into the shift register and then assert the RCLK signal to move all eight bits simultaneously to the outputs. As a result, the outputs on the 74HC595 IC do not show a shifting or blurred pattern, only the end result. Shifting occurs entirely within the 74HC595 and doesn't affect the outputs until you give the RCLK input a short logic-1 pulse.

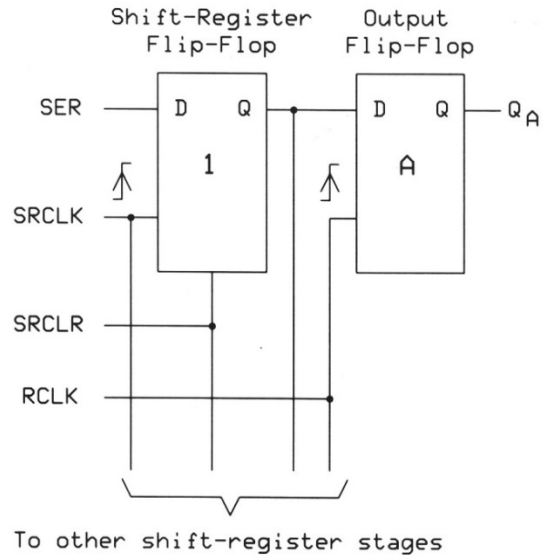


Figure 8.6.
The 74HC595 shift register transfers data from the SER input to the Q output on flip-flop 1 on the positive edge of the SRCLK signal. This Q output goes to the next flip-flop in the shift register. After all eight bits move into the shift-register flip-flops, a pulse on the RCLK input transfers all shift-register data into the output flip-flops.

In **Program 8.1**, you see these three statements just after variables have values assigned to them:

```
outa[register_clock] := 0
outa[register_clock] := 1
outa[register_clock] := 0
```

When you touch the jumper wire to ground, you clear the shift-register flip-flops, but not the output flip-flops. The three instructions above create a short logic-1 pulse on the Propeller's P10 pin. That signal goes to the 74HC595 RCLK pin and causes the reset shift-register bits (logic-0s) to transfer to the output pins. In this case, those outputs remain at a logic-0 until the code sends another RCLK signal to the 74HC595 IC.

The main repeat loop runs eight times and determines whether to send a logic-0 or a logic-1 to the 74HC595 IC. It sets or clears the Propeller P8 pin accordingly and then pulses the SRCLK pin on the shift-register IC. I included the statements below in **Program 8.1** to clear the outputs on the 74HC595 to logic-0:

```
outa[register_clock] := 1   'Set register clock pin to logic-1
outa[register_clock] := 0   'Reset this pin to logic-0
waitcnt(cnt + clkfreq)     '1-sec delay
```

As a result, the shift-register information gets transferred to the output flip-flops after each shift occurs. The last statement, a 1-second delay, lets you see the bits shift on the LEDs.

Step 5.

This step shows you how the shift register will accept eight bits and then simultaneous transfer all bits simultaneously to the eight outputs. This type of operation eliminates seeing bits shift on the LEDs.

The code in **Program 8.2**, which you'll see shortly, uses the same instructions as those in **Program 8.1**, but in a slightly different order. **Code Snippet 8.1** shows seven of the original steps:

Code Snippet 8.1.

```
outa[serial_clock]  := 1
outa[serial_clock]  := 0
outa[register_clock] := 1
outa[register_clock] := 0
waitcnt(cnt + clkfreq)

  repeat
    ser_data := ser_data
```

Code Snippet 8.2 places the register-clock statements *outside* the shift-and-output loop. Before you run **Program 8.2**, briefly touch the jumper from the 74HC595 SRCLR pin (pin 10) to ground to reset the shift register flip-flops. When you run the program, what do you see on the LEDs? It takes about eight seconds to see results because the program still includes a 1-second delay between each shift operation.

Code Snippet 8.2.

```
outa[serial_clock]  := 1
outa[serial_clock]  := 0
waitcnt(cnt + clkfreq)

outa[register_clock] := 1
outa[register_clock] := 0

repeat
  ser_data := ser_data
```

Change the pattern of bits assigned to the `ser_data` variable, but **do not** reset the shift register. (Do not touch the jumper from the 74HC595 SRCLR pin (pin 10) to ground to reset the shift register.)

Run the program again. Did you see any bits shift through the LEDs? You should see the LEDs change only after the shift-register flip-flops have received a complete 8-bit pattern. By waiting to update the 74HC595 outputs until all eight bits have moved into the shift register you prevent any disruption to an external device due to patterns that change during shift operations. In fact, the shifting of bits takes place "behind the scenes." Change the bit patterns and run the program again, but without resetting the 74HC595 shift register. You may remove the `waitcnt` statement to eliminate the time delay so your patterns will appear almost instantaneously. Or you can "comment out" the wait statement by placing an apostrophe in front of it. The statement turns a light blue to indicate it will no longer get executed:

```
'waitcnt(cnt + clkfreq)
```

Program 8.2.

```
{{
''*********************************************
''*  Shift-Register Program 8.2
''*  Author: Jon Titus 11-03-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Shift register control program uses
''*  74HC595 chip. P8 for data, P9 for clock
''*  P10 for register clock. This program shows
''*  how bits get shifted into the IC and displayed
''*  on eight LEDs.
''*  Briefly ground the SRCLR pin on the 74HC595
''*  chip before you run this program. This action
''*  clears the shift register for a clean start.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

VAR byte ser_data                  '8-bit serial data
    byte data_port                 'Data-output pin to shift reg
    byte serial_clock              'Pulse for shift reg clock
    byte register_clock            'Pulse for register-transfer clock

PUB Start                          'Main program starts here
  ser_data := %10111101            'Save 8 serial bits here
  dira[8..10] := %111              'Select 3 output pins
  outa[8..10] := %000              'Clear the output pins
  data_port := 8                   'Pin P8
  serial_clock := 9                'Pin P9
  register_clock := 10             'Pin P10

  outa[register_clock] := 0        'Generate a short register-
  outa[register_clock] := 1        'transfer pulse to clear
  outa[register_clock] := 0        'shift-register outputs
  waitcnt(cnt + clkfreq)           'Delay so you can see cleared
                                   ' bits

  repeat 8                         'Repeat loop 8 times for
                                   'a byte
    if (ser_data => 128)           'Test MSB for 1 or 0
        outa[data_port] := 1       'MSB = 1, output 1 on
                                   'data_port
```

```
    else
        outa[data_port] :=0         'MSB = 0, output 0 on
                                    ' data_port
    ser_data := ser_data << 1       'shift ser data 1 bit to left
    outa[serial_clock]  := 1        'Pulse shift register to
                                    'input one bit
    outa[serial_clock]  := 0
    waitcnt(cnt + clkfreq)          'Delay so you wait to see the
                                    'parallel transfer of bits.
  outa[register_clock] := 1         'Pulse RCLK input on
                                    '74HC595 IC
  outa[register_clock] := 0
  repeat
    ser_data := ser_data            'Do-nothing loop. MCU "waits"
                                    'here.
```

Step 6.

The shift-register should work well to control the LEDs in a 7-segment display. But a design on paper doesn't always turn into a practical circuit, so in this step you will test it with hardware and software. The circuit diagram in **Figure 8.7** shows how the 74HC595 shift register connects to a 4-digit multiplexed display module. I used a Lumex LDQ-N516RI common-cathode module that simplified wiring on a solderless breadboard. You can find similar 4-digit displays from other manufacturers. Go ahead and wire this circuit. The connections might look similar to those in **Figure 7.12**, but the pin numbers differ and the new circuit uses a shift-register IC. Ensure ground connections actually go to ground. At times they can prove easy to overlook.

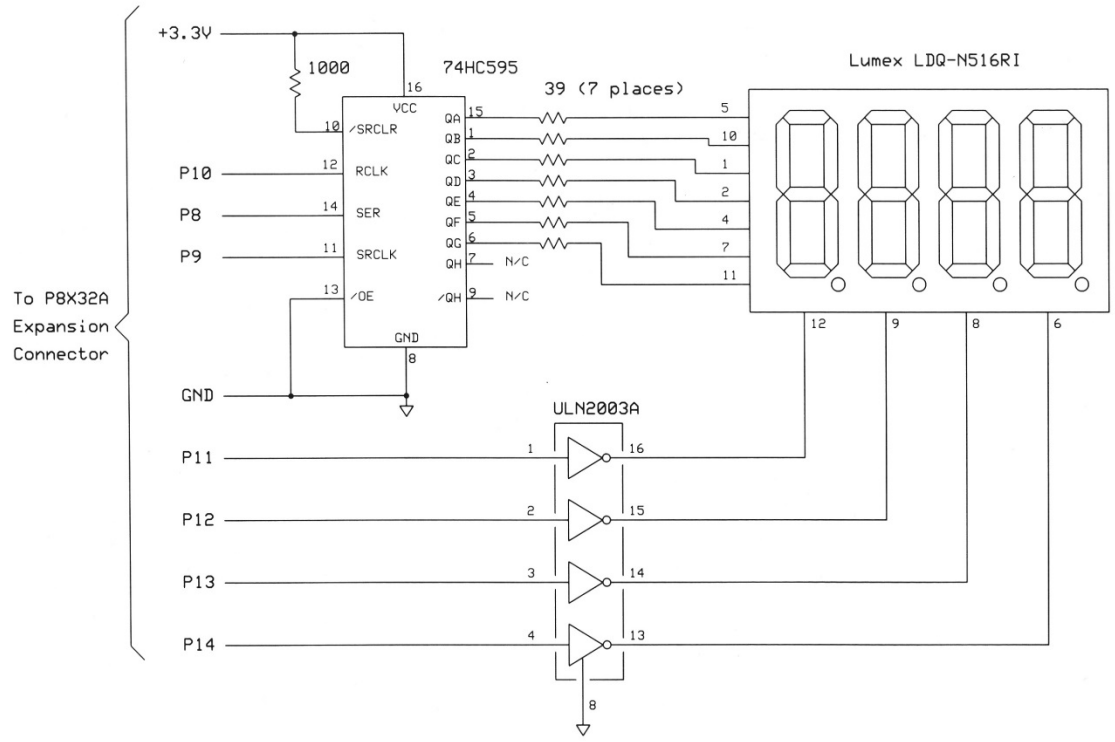


Figure 8.7.

A multiplexed 4-digit module can use a shift register IC to control individual segment LEDs. This circuit uses three connections between the 74HC595 shift register and the P8X32A MCU board rather than seven needed when an MCU directly controls the LED segments. (See **Figure 7.7**). Four other connections switch each digit to ground one at a time through a ULN2003A IC.

Program 8.3 will control the multiplexed display through the shift register. You may load this program and run it. Time-delay statements slow the display so you can see how the digits change as the program runs. I set the `big_value` variable to 3716 to test the program, but you may use any 4-digit positive integer (0000 to 9999) you choose.

Program 8.3.

```
{{
''*********************************************
''*  Program 8.3
''*  Author: Jon Titus 11-03-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration that separates a
''*  4-digit value into separate numerals and
''*  uses 74HC595 shift register to control
''*  display segments.
''*  LUMEX LDQ-N516RI 4-digit display module.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

VAR
    byte segment_code[10]          'Storage for segment codes
    byte count                     'Counter used in display loop
    byte display_loop              'Counter used for display-
                                   'loop delay
    byte digit[6]                  'Space for 6 digits
    long big_value                 'Variable for large value
                                   'to parse
    byte numb_of_digits            'Number of digits in
                                   'big_value
    long divisor                   'Divisor used to start
                                   'parsing
    byte index                     'Index to identify each digit
    byte decimal_counter           'Counter to track each
                                   'decimal digit
    byte ser_data                  '8-bit serial data
    byte data_port                 'Data-output pin to shift reg
    byte serial_clock              'Pulse for shift reg clock
    byte register_clock            'Pulse for register-transfer
                                   'clock

PUB Start                          'Main program starts here

segment_code[0] := %01111110       'Pattern for "0"
segment_code[1] := %00110000       'Pattern for "1"
segment_code[2] := %01101101       'Pattern for "2"
segment_code[3] := %01111001       'Pattern for "3"
segment_code[4] := %00110011       'Pattern for "4"
segment_code[5] := %01011011       'Pattern for "5"
segment_code[6] := %00011111       'Pattern for "6"
segment_code[7] := %01110000       'Pattern for "7"
segment_code[8] := %01111111       'Pattern for "8"
```

```
segment_code[9] := %01110011       'Pattern for "9"


dira[8..10] := %111                'Output pins for shift
                                   'register
outa[8..10] := %000                'Set shift-register outputs
                                   'to 0
dira[11..14] :=%1111               'Set I/O outputs to ULN2003A
outa[11..14] := %0000              'Clear ULN2003A outputs

data_port := 8                 'Use P8 for shift-register data
serial_clock := 9              'P9 for shift-register clock
register_clock := 10           'P10 for register flip-flop
                               'outputs

big_value := 3716              'Large value to work with
numb_of_digits := 4            'Number of digits in big_value
divisor := 1                   'Starting value of divisor

'LOOP 1: Given the number of digits, calculate the largest divisor. For four
digits, start with 1000, for example.

  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10


'LOOP 2: Use divisor to separate the numerals. Use '(numb_of_digits - 1) so
loop can go from digit[0]
' to digit[3] for a 4-digit value, which saves four values.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10

'LOOP 3: Get the segment data for a numeral and sent to the
'shift register. After all bits sent, transfer to output flip-
'flops to drive segments, control ULN2003A to turn on
'corresponding numeral.

'Repeat this loop forever
repeat
   repeat display_loop from 0 to 59
      repeat count from 0 to 3             'Four digits
         'Get segment codes
         ser_data := segment_code[digit[count]]
            'Repeat loop 8 times for a byte
            repeat 8
               if (ser_data => 128)       'Test MSB for 1 or 0
                  outa[data_port] := 1
               else
                  outa[data_port] :=0
               ser_data := ser_data << 1
               outa[serial_clock]  := 1
               outa[serial_clock]  := 0
               'waitcnt(cnt + clkfreq)       'Commented out
         outa[register_clock] := 1        'Pulse RCLK input
```

```
        outa[register_clock] := 0    'on 74HC595 IC
        outa[11+count] :=1           'turn on digit
        waitcnt(clkfreq/200 + cnt)   'time delay for 1 sec
        outa[11+count] := 0          'turn off the digit
```

The 8.3 program displays each numeral from left to right. The thousands digit turns on for about a second, turns off for about a second, and the process repeats for the other three digits. To eliminate the "off" period between the display of each digit, remove or comment-out the delay statement in `LOOP 3`:

```
waitcnt(cnt + clkfreq)        'Delay so you wait to see parallel
```

Now the digits turn on one after the other, but not fast enough so you see all four simultaneously on the display module. To increase the frequency at which the digits turn on and off, change the delay statement in `LOOP 3` so the `clkfreq` has a divisor of 120:

```
waitcnt(clkfreq/120 + cnt)    'time delay for 1 sec
```

Can you see all four digits simultaneously? Does your display flicker? If so, increase the divisor. I found a divisor of 180 just about eliminated the flicker, although I could still see some with my peripheral vision. A divisor of 200 worked well.

Step 7.

By using a 74HC595 shift register to drive the display segments, you used three I/O pins instead of seven – a saving of four I/O pins for other uses. The four cathode switches in the ULN2003A IC still requires four I/O pins. Can you think of a way to reduce the number of I/O pins needed to drive the ULN2003A IC? How about another shift register?

You might think that approach would still need three I/O pins pins for the second shift register instead of four for the ULN2003A IC, a savings of only one I/O pin. The 74HC595 shift-register chips have the capability to operate in series, so you can connect two, head to tail as shown in **Figure 8.8.** Integrated circuits A and B have connections that let circuit designers create a 16-bit shift register. Adding a third 74HC595 would give you a 24-bit shift register. As you shift bits into IC A, any bits already in the shift-register flip-flops shift out the QH' pin (pin 9) and into IC B on its SER input (pin 14). **Do not build this circuit now.** Leave the circuit shown in **Figure 8.7** in your breadboard. You will need it in Experiment 9.

In this arrangement (**Figure 8.8**), you could shift 16 bits out of the P8X32A; the first eight bits to control the ULN2003A IC and the following eight bits to control the display segments. The ULN2003A IC needs only four of the eight outputs, though.

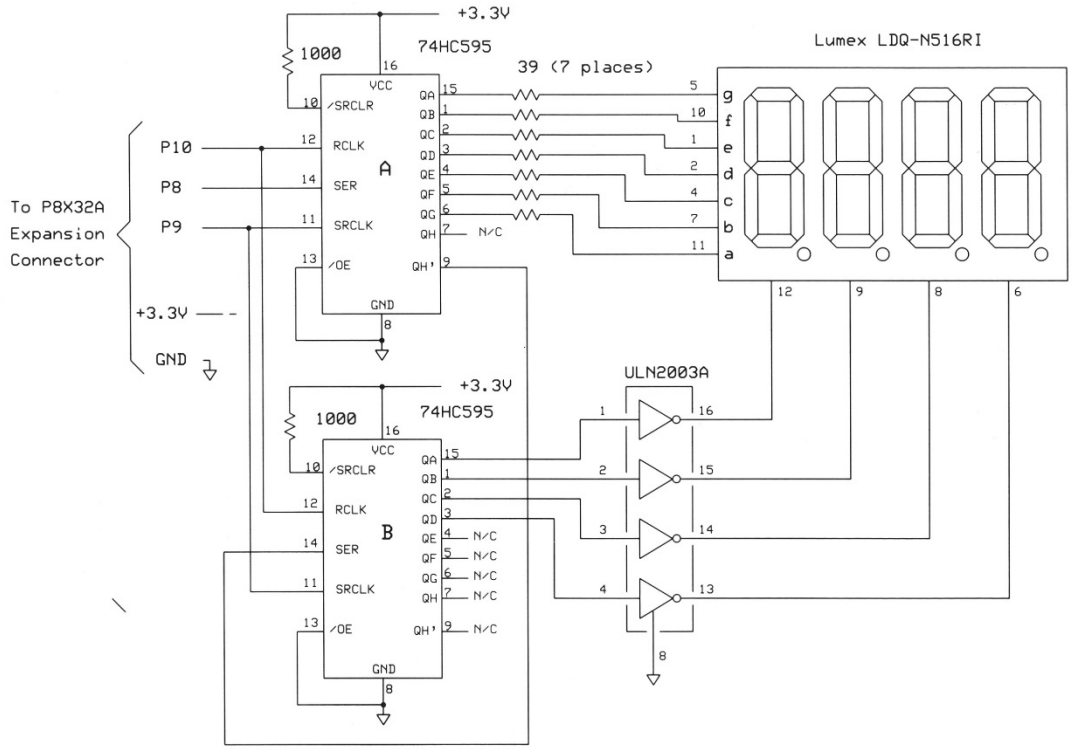


Figure 8.8.
This circuit connects shift registers A and B in series so a 12-bit value can control segments and cathodes for four digits.

The circuit shown in **Figure 8.8** eliminates the four Propeller pins needed for the ULN2003A IC. Note these key points:

a. Both shift-register ICs use the same SRCLK and RCLK signals.

b. The QH' output from shift-register A connects to the SER input on shift-register B. This connection "cascades" serial data from IC A to IC B.

c. This arrangement still needs only the three signals used for one 74HC595 shift-register IC. The MCU circuit shown previously in **Figure 7.12** needed 11 I/O pins on the MCU.

d. The two shift registers have a total of five unused outputs you can use for other purposes: QH on shift-register A, and QE through QH on shift register B.

e. Software can transfer 16 bits to the shift-register flip-flops and then assert the RCLK signal to simultaneously set the seven segment-control bits and the four digit-control bits.

The bit arrangement in **Figure 8.9** shows which bits control segments and digits. Because the circuit only requires 12 bits, software need not shift bits into shift-register B for the QE through QH outputs. You might use these outputs later for additional digits, individual LEDs, an audio alarm, and so on.

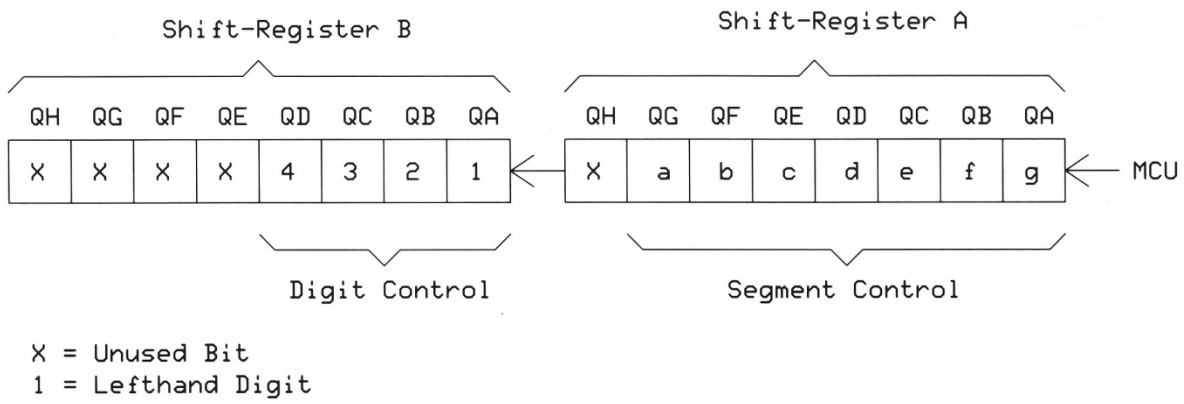


Figure 8.9.
Arrangement of bits as sent to the two 74HC595 shift-register ICs. The value 00000100 01110000_2 would show the numeral 7 at the 10's digit. The four leading, or most-significant bits (Shift Register B, bits QH-QE) have no use, so sending only the needed 12 bits: 0100 01110000_2 would have the same effect.

Using software to combine the bits for the numerals and the bits for the segments can get complicated when you must test bits in the segment code and convert a value into binary information. The next experiment introduces the Serial Peripheral Interface (SPI) hardware and software that simplifies communications with shift registers.

Reference

Bauer, Michael J., "The 7-uP Alarm Clock/Time-Switch, Part 2" *Elektor* magazine, March 2013, pp. 24 – 29. www.elektor-magazine.com.

Experiment No. 9. – Better Serial Communications for LED-Display Control

Abstract

The hardware and software included in Experiment 8 showed how serial communications from an MCU can simplify control of multi-digit LED displays. The serial technique also uses fewer I/O pins that a brute-force parallel-signal approach so you have more I/O pins on an MCU for other input and output tasks. In this experiment you will learn about the Serial Peripheral Interface (SPI) that further simplifies control of a variety of integrated circuits. A set of Propeller objects provides the software needed to implement SPI communications.

Keywords

shift register, 74HC595, serial, LED, display, flip-flop, ULN2003A, multiplex, transistor array, serial-peripheral interface, SPI, Propeller, object, cog

Requirements

(2) - 74HC595 8-bit serial-in, parallel-out shift register
(1) - ULN2003A high-current transistor array, 16-pin DIP
(7) - 39-ohm, 1/4W, 5% resistors (orange-white-black)
(2) - 1000-ohm, 1/4W, 5% resistor (brown-black-red)
(1) - Lumex LDQ-N516RI 4-digit module, common cathode, or equivalent
(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - USB cable

Introduction

In Experiment 8, program statements separated a byte into bits that the Propeller sent to an external shift register to control individual segments in a 4-digit display module. Each time the program updates the segment information for a digit, the program must performed the same steps to transmit bits to the shift register. The following code snippet shows these steps:

```
repeat 8                        'Repeat loop 8 times for a byte
  if (ser_data => 128)          'test MSB for 1 or 0
     outa[data_port] := 1       'MSB = 1, output 1 on data_port
  else
     outa[data_port] :=0        'MSB = 0, output 0 on data_port
  ser_data := ser_data << 1     'shift serial data 1 bit to left
  outa[serial_clock]  := 1      'Pulse shift register to input
```

The timing diagram in **Figure 9.1** shows the data transmitted (upper trace), and the shift-register clock, SRCLK (lower trace). The time scale at the bottom of the display runs from "time zero" out to 500 milliseconds on the right. The overall transmission time takes about 375 µsec. That might not seem like a lot of time, but when the Propeller must do other things, the 375 µsec periods add up. Remember, the MCU goes through this routine every time it changes the segment information. It would help to have software that can send bits to external devices but without the time "overhead" of the `repeat 8` loop explained above.

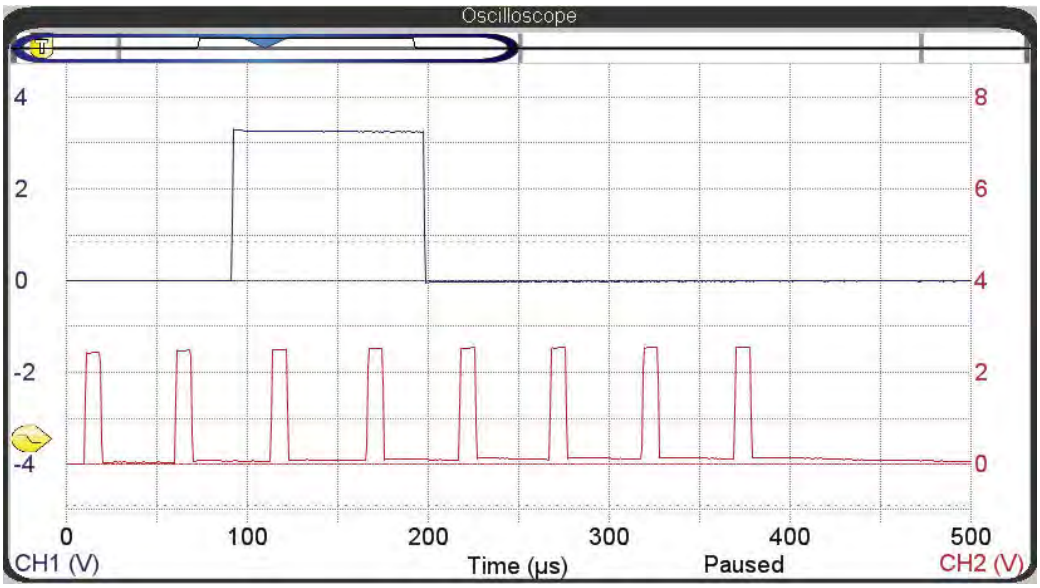


Figure 9.1.
Timing of the data (top trace) for the numeral 1 (00110000_2) and the SRCLK clock signal (bottom trace). The software transmits the most-significant bit first. The period between adjacent positive-going edges of the SRCLK signal amounts to about 50 µsec. The LED displays do not use the most-significant bit (MSB).

In the late 1970's, Motorola's semiconductor group (now Freescale Semiconductor) produced a microcontroller with a Serial Peripheral Interface, also known as SPI and pronounced "spy." The designers aimed to create a simple "bus" for chip-to-chip communications on a circuit board. The SPI bus operates with one *master* device – usually an MCU – and one or more *slave* devices. The slaves can receive information from a master and transmit information to a master, but they cannot start an exchange with a master or with another slave.

The master controls all SPI communications, and the SPI signals comprise:

1. A serial-clock signal (SCLK) that synchronized data communications,
2. A master-out, serial-in signal (MOSI),
3. A master-in, serial-out signal signal (MISO), and
4. An optional fourth signal, Slave Select (SS) that lets a master choose a slave device if several such devices exist.

In **Figure 9.2**, the SCLK, MOSI, and MISO signals connect in parallel between the master and slave devices. A slave-select signal (SSx*) from the master to each slave determines which slave the master will communicate with. For only one slave device, you might not need an SS signal.

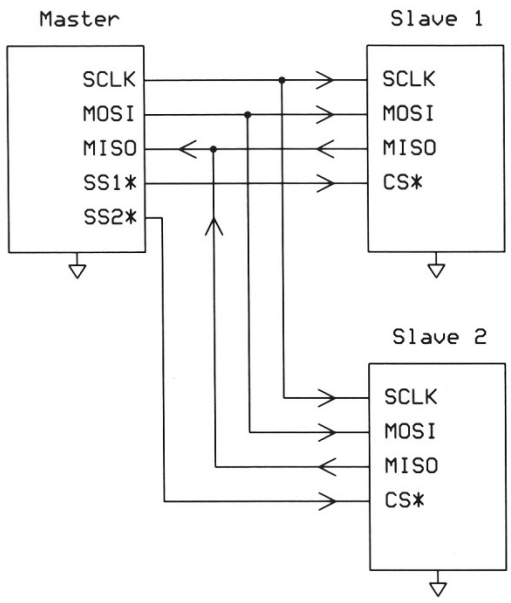


Figure 9.2.
A typical SPI-bus circuit that includes two peripheral devices. All three devices must have a common ground connection as shown here.

The diagram in **Figure 9.3** shows shows the SPI signals and internal registers in more detail for a master and a slave device. A transmission starts when the master transmits a bit on its MOSI output, synchronized with the SCLK clock signal. The slave transmits a bit to the master on the MISO line, also synchronized with the SCLK clock. A shift register in the master and a shift register in the slave accepts the serial information.

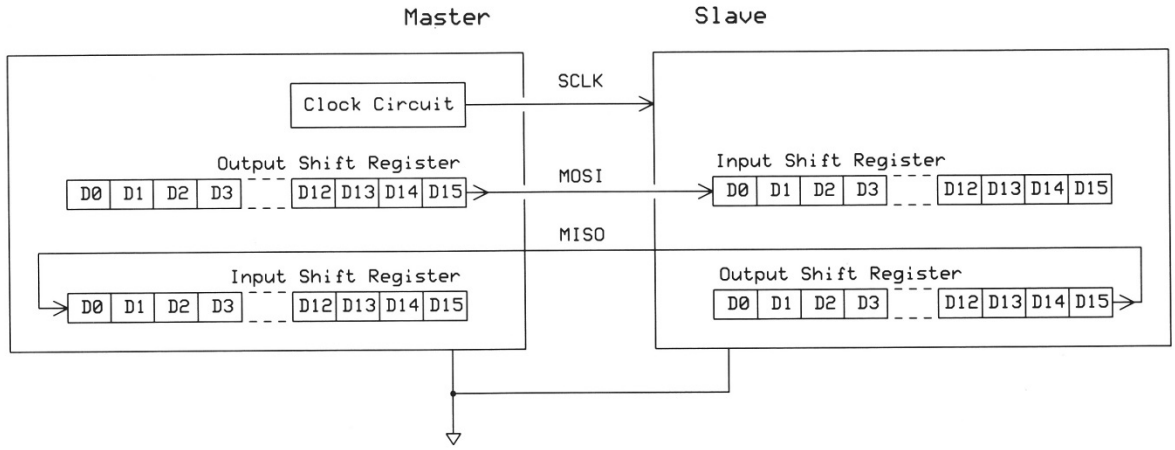


Figure 9.3.
An internal view of the SPI signal paths and shift registers. Note the ground connection between both devices.

Information sent from a master to a slave might include a request for a temperature or a command for the slave to go into a low-power standby mode. Information transmitted from a slave to a master could include measurement data, IC identification number, a status update, and so on. The Microchip Technology TC77 digital temperature sensor, for example, includes three 16-bit registers: configuration, temperature, and manufacturer's ID information. An MCU can configure the sensor to shut down or to update the temperature continuously. When a TC77 sensor receives a "get temperature" command it sends a 13-bit temperature value to the MCU in a 16-bit word. (Three bits go unused.) A complete temperature measurement takes about 300

msec. The SPI protocol, which means the types of data communicated and how devices interpret it, lacks a way to identify a specific slave device in a communication. Thus the need for a chip-select or slave-select input (CS*) on SPI-compatible ICs. The TC77 IC includes a chip-select signal, CS*, which the master must set to a logic-0 to start communications. For the TC77 data sheet, please visit: http://ww1.microchip.com/downloads/en/devicedoc/20092a.pdf.

Unlike some forms of serial communications, the SPI protocol does not limit the number of bits slaves and a master can transmit or receive. A master device could transmit as few as eight bits or as many as you require. The types of registers in a slave device usually govern the number of bits in an SPI communication. When the master starts an SPI communication, it transmits information on the MOSI pin and simultaneously generates a clock signal that indicates to a slave device when it should sample each bit. The diagram in **Figure 9.4** shows this timing relationship.

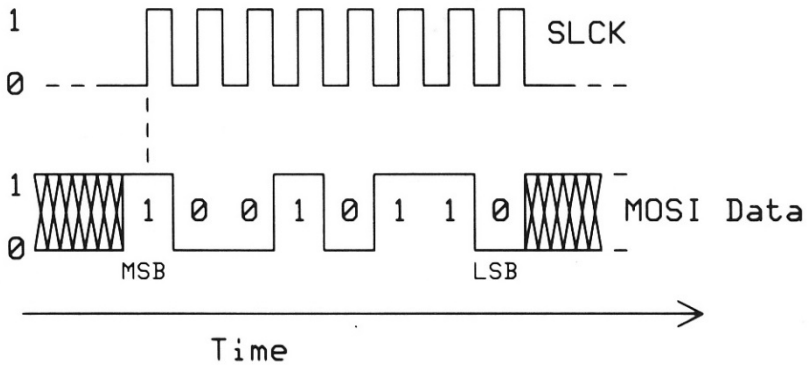


Figure 9.4.
Timing relationship between the SCLK clock output from a master and the MOSI data output to a slave device. The cross-hatch areas indicate unknown, or "don't care," signal conditions.

Unfortunately, SPI-chip manufacturers have created confusion about the clock signals so circuit designers have four choices! You can find more information about clock polarity and phase in the Wikipedia article, "Serial Peripheral Interface Bus," at: http://www.en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus. This experiment uses clock polarity = 0 and clock phase = 0, as seen in **Figure 9.4**.

In this experiment you will use only two SPI signals, SCLK and MOSI, because the programs only send data to one device – the cascaded 74HC595 shift registers. So the shift registers do not need a Slave-Select signal. The MCU needs a master-out-serial-in (MOSI) output to send information to the shift registers. But the Propeller MCU will not receive any information from them, so it doesn't need a master-in-slave-out (MISO) input. Although the 74HC595 shift-register ICs aren't strictly SPI devices, but they can receive information via SPI signals.

Step 1.

Unlike many MCUs, the Parallax Propeller MCU does not have an internal circuit devoted to SPI communications. But the Propeller Object Exchange (OBEX) – a collection of software developed by Propeller enthusiasts – includes several object libraries that handle SPI communications. For the OBEX Web site, please visit: http://obex.parallax.com. In this experiment, you will use the "Propeller SPI Engine," object library created by Beau Schwabe (Parallax) in 2009. You can download a ZIP file that contains the SPI software and a demonstration program, or you can find it in the Experiment 9 software folder. You will not need the demonstration program.

So what are these "objects" anyway? As explained briefly in previous experiments, in Spin and other programming languages, objects refer to code that performs a specific function. The structure of the language lets programmers use objects written by others to perform specific actions. In the case of the SPI objects, we take advantage of software by linking to it and using the functions within it, as you'll see when I explain SPI-control software in a later step. (Objects written in languages such as Java, C++, and Python have other capabilities that go beyond the scope of this book.)

If you downloaded SPI_Asm.spin, put it in your Propeller working directory. Or, open the Experiment 9 folder that already contains it. When I ran a short program to test the `SPI_Asm.spin` object with an 8-bit value, my oscilloscope displayed the signals shown in **Figure 9.5**.

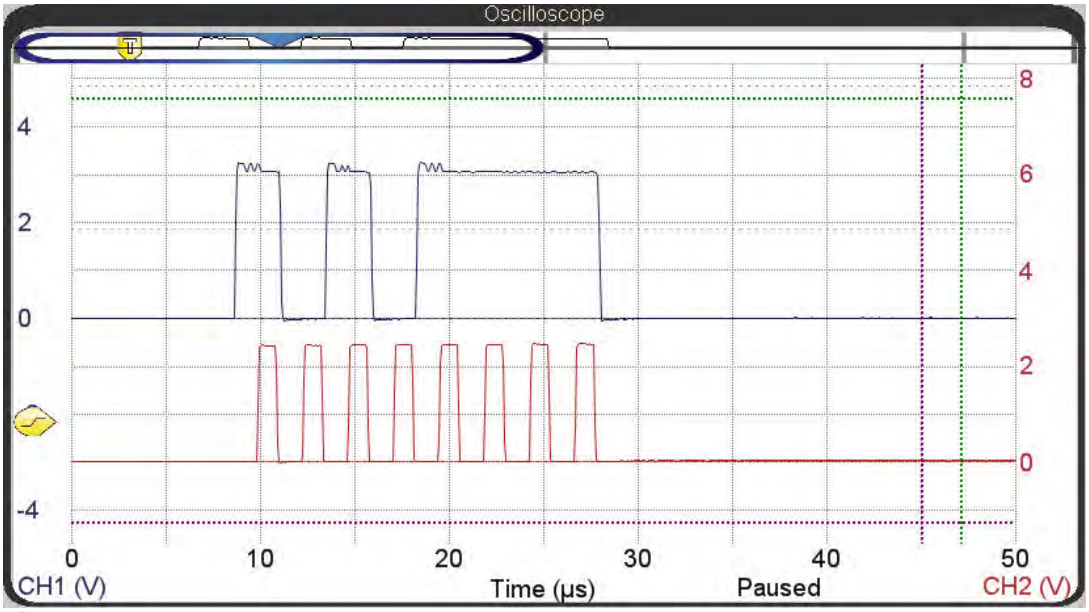


Figure 9.5.
Timing of the data (top trace, 10101111_2) and the SPI clock signal (SCLK, bottom trace). The period between adjacent positive-going edges of the SCLK signal amounts to about 2.35 µsec. Thus objects in the `SPI_Asm` object library has almost a 20-fold time advantage over the bit-by-bit approach used in Experiment 8.

Step 2.

Now you will use the shift-register circuit you built in Experiment 8 and shown again in **Figure 9.6**. If you have that circuit ready, please jump to Step 3. If you do not have this circuit in a solderless breadboard, build it now and use **Program 8.3** to test it. After the program runs and the four 7-segment displays show your `big_value` digits, continue with Step 3.

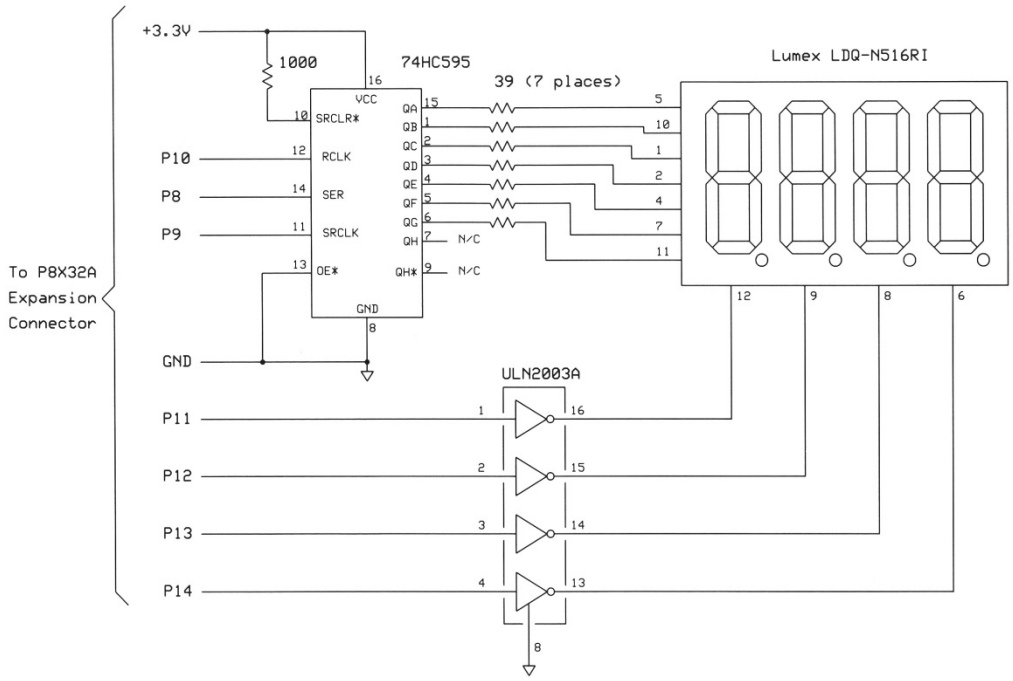


Figure 9.6.
Schematic diagram for a circuit that uses a 74HC595 shift register to control the seven segments in a display. Four I/O pins on the P8X32A board control the common cathodes via a ULN2003A IC.

Step 3.

Instead of separating the 7-segment patterns into 0s and 1s, and shifting each bit into the 74HC595 shift register, the SPI object library will handle the details. The following statement shows information needed to perform the `SHIFTOUT` operation in the SPI object library:

```
SHIFTOUT(Dpin, Cpin, Mode, Bits, Value)
```

`Dpin` = Pin used to transmit data to an SPI device (MOSI)
`Cpin` = Pin used as the output clock (SCLK)
`Mode` = Selects whether MSB or LSB shifts out first. Defaults to MSB first.
`Bits` = Number of bits in a transmission
`Value` = Value to shift out

To review, before you can use the SPI object library, a program needs several other statements:

1.
```
OBJ
SPI     :     "SPI_Asm"
```

   In the OBJ section of a Spin-language program, your code must declare it wants to use the SPI objects in the file SPI_Asm. The "SPI" prefix gives us a way to uniquely identify the SPI objects in the library. You could choose another identifier such as `SPI_LEDs` if you wish.

2.
```
SPI.start(15,0)
```

   Before you use any objects in the SPI library, the program must initialize the SPI settings with a clock-delay time and a value that configures the clock output. The first value, 15, sets a 1 µsec clock delay according to the formula:

$300\,nsec + [(n - 1) * 50\,nsec]$ and $300\,nsec + [(15 - 1) * 50\,nsec] = 1\,\mu sec$

The second value, a 0 or a 1, sets the clock output for either positive clock pulses or negative clock pulses, respectively. The programs that follow all uses positive pulses (see **Figure 9.5**).

Note the `SPI.start(15,0)` statement used the `SPI.` prefix to indicate we want to use the `start` object in the SPI_asm library. Many objects need information – placed in parentheses after the `SPI.`"command" – so they know what to do. We call this passing parameters to an object. So, when we use the command `SPI.start(15,0)`, the Spin software stores the values 15 and 0 and lets the `start` object know where to find them.

3. `SPI.stop`

   This command stops SPI communications. The program in the next step does not use the `stop` command because it continues to use SPI communications to update the 4-digit display.

At this point, you probably wonder, "How do I know what's in an object library? How did you know about the `start` and `SHIFTOUT` operations?" Well-written object libraries should include comments about each object, how it works, the information it needs, and so on. You can look at the code in an object library for statements such as: `PUB SHIFTIN`, `PUB stop`, and `PUB squareroot`. The `PUB` prefix defines an object as available for "public" use. That means other programs can use the object when they need its function. You might also see "private" objects labeled as `PRI objectname`. These objects perform functions only for other objects in the library and aren't available for your use in a program.

Unless you look closely at the code in the `SPI_asm.spin` file you won't know this object library relies on a second processor in the Propeller chip. Remember, a Propeller chip has eight separate processors, which Parallax calls cogs. So after a program finishes SPI communications the `SPI.stop` command stops the SPI software and makes the processor it used available for other programs. For more information about Propeller cogs, please visit: http://www.parallax.com/propeller/qna/Content/QnaTopics/QnaCogs.htm.

Step 4.

Program 9.1 includes the SPI statements explained above. Make sure you have the SPI_Asm.spin program in your Propeller workspace folder and open it in the Propeller Tool. Next, open the **Program 9.1** file and run it. You should see your 4-digit `big_value` appear on the four-digit display. Further explanations of the code follow the **Program 9.1** listing.

Program 9.1.

```
{{
''*********************************************
''*  Program 9.1
''*  Author: Jon Titus 11-04-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration that separates a
''*  4-digit value into separate numerals and
''*  uses 74HC595 shift register to control
''*  display segments.
''*  SPI_Asm object library used for SPI-type
''*  output of 8-bit 7-segment patterns
''*  LUMEX LDQ-N516RI 4-digit display module.
''*********************************************
}}
```

```
CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000               'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"        'Invoke the SPI_Asm.spin file
                                   'saved in your Propeller
                                   'working folder.
VAR
    byte segment_code[10]          'Storage for segment codes
    byte count                     'Counter used in display loop
    byte display_loop              'Counter used for display-
                                   'loop delay
    byte digit[6]                  'Space for 6 digits
    long big_value                 'Variable for value to parse
    byte numb_of_digits            'Number of digits in
                                   'big_value
    long divisor                   'Divisor for parsing
    byte index                     'Index to identify each digit
    byte decimal_counter           'Counter to track each digit
    byte ser_data                  '8-bit serial data
    byte data_port                 'Data-output pin to shift reg
    byte serial_clock              'Pulse for shift reg clock
    byte register_clock            'Pulse for reg-transfer clock

PUB Start                          'Main program starts here

segment_code[0] := %01111110       'Pattern for "0"
segment_code[1] := %00110000       'Pattern for "1"
segment_code[2] := %01101101       'Pattern for "2"
segment_code[3] := %01111001       'Pattern for "3"
segment_code[4] := %00110011       'Pattern for "4"
segment_code[5] := %01011011       'Pattern for "5"
segment_code[6] := %00011111       'Pattern for "6"
segment_code[7] := %01110000       'Pattern for "7"
segment_code[8] := %01111111       'Pattern for "8"
segment_code[9] := %01110011       'Pattern for "9"

''Start SPI for 1 uSec delay and positive-going clock pulses
SPI.start(15,0)

dira[8..10]  := %111               'Output for shift register
outa[8..10]  := %000               'Clear shift-register outputs
dira[11..14] :=%1111               'Set display-switch outputs
outa[11..14] := %0000              'Clear display-switch outputs

data_port := 8                     'P8 for shift-register data
serial_clock := 9                  'P9 for shift-register clock
register_clock := 10               'P10 for reg flip-flops
big_value := 3716                  'Large value to work with
numb_of_digits := 4                'Number of digits in
                                   'big_value
divisor := 1                       'Starting value of divisor

'LOOP 1: Given the number of digits, calculate the largest
'divisor for the digits.
```

```
  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10


'LOOP 2: Use divisor to separate the numerals. Use '(numb_of_digits - 1) so
loop can go from digit[0] to digit[3] 'for a 4-digitvalue. Loop stores four
values.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10

'LOOP 3: Get the segment data for a numeral and send to shift
'register via SPI object. Transfer data to output flip-flops
'to drive segments & control ULN2003A to turn on
'corresponding numeral.

'Repeat this loop "forever"

  repeat
     repeat display_loop from 0 to 59
        repeat count from 0 to 3          'Loop for four digits
           'Get segment codes
           ser_data := segment_code[digit[count]]
           SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, ser_data)

           outa[register_clock] := 1          'Pulse RCLK input
           outa[register_clock] := 0          'on 74HC595 IC
           outa[11+count] :=1                 'turn on digit
           waitcnt(clkfreq/200 + cnt)         'time delay
           outa[11+count] := 0                'turn off digit
 ' –  – -end –  – -
```

The code snippet that follows (without comments) comes from **Program 8.3**, and it shows the original way in which the program had to separate a segment pattern into individual bits and shift each bit into the external 74HC595 shift register.

Code Snippet 9.1.

```
ser_data := segment_code[digit[count]]
   repeat 8
     if (ser_data => 128)
        outa[data_port] := 1
     else
        outa[data_port] :=0
     ser_data := ser_data << 1
     outa[serial_clock]  := 1
     outa[serial_clock]  := 0
```

The next snippet shows the same routine, but with a single SPI statement. Often you can find objects that perform a task you need, so look in the Propeller Object Exchange (OBEX) before you try to recreate software that already exists.

Code Snippet 9.2.

```
ser_data := segment_code[digit[count]]
SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, ser_data)
```

In the last statement above, the notation `SPI#MSBFIRST` might look unfamiliar because it includes a number sign, #. That sign indicates the `SPI.SHIFTOUT` object should use the constant, `MSBFIRST`, already defined in the SPI_asm.spin file and given a useful name. Without the capability to use a constant already defined in an object, a programmer would have to dig into the object-library code to determine what values cause specific actions to occur. Likewise when someone inspects your code and sees `SPI#MSBFIRST`, they better understand what that variable identifies and how the `SPI.SHIFTOUT` command works.

Before you include any object library in a program, write a short program to test the objects you plan to use. I found – and corrected – a few minor errors in objects that led to unexpected behavior.

Step 5.

In this step you will use a second 74HC595 shift register in place of the four digit-control outputs (P11 through P14) from the Propeller board to the ULN2003A IC. The circuit you have on your breadboard now corresponds to the one shown earlier in **Figure 9.6**.

Before you add a second 74HC595 IC to the circuit you first must **remove** four connections between the Propeller P8X32A board and your solderless breadboard:

1. Remove the connection from the ULN2003A pin 1 to the Propeller board P11 expansion connector.
2. Remove the connection from the ULN2003A pin 2 to the Propeller board P12 expansion connector.
3. Remove the connection from the ULN2003A pin 3 to the Propeller board P13 expansion connector.
4. Remove the connection from the ULN2003A pin 4 to the Propeller board P14 expansion connector.

The circuit diagram in **Figure 9.7** shows only the components and connections to *add* to your circuit. Shift register "A" already forms part of the circuit and it appears in **Figure 9.7** only for its pin-number references. **Figure 9.7** does not show all connections to this shift register.

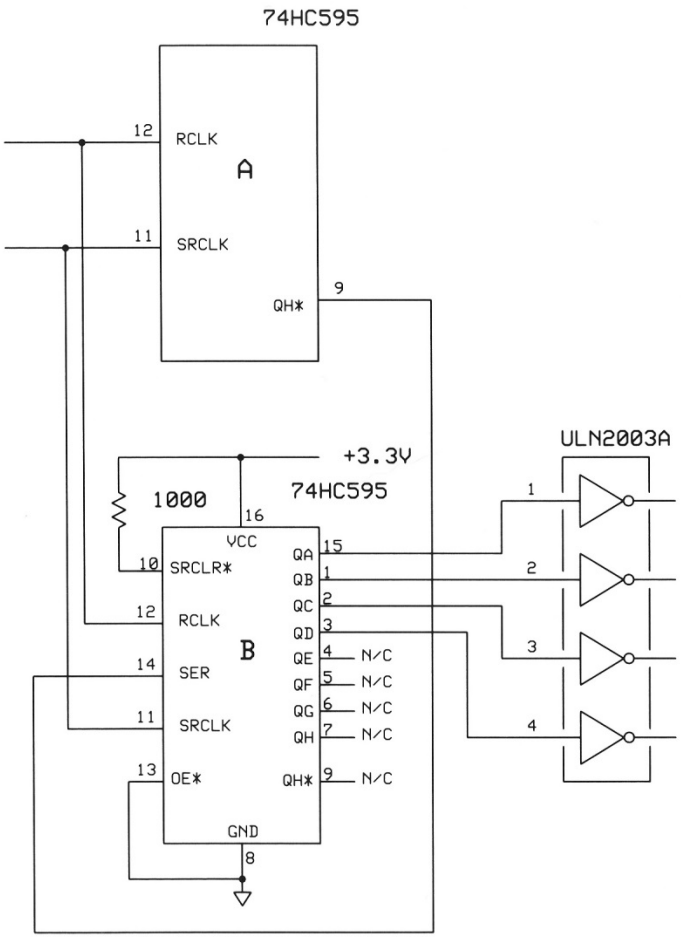


Figure 9.7.
Diagram of the circuit components and connections to add so you can control the 4-digit display with two shift registers. Shift register A shows only the connections to add. This IC will have other wires already connected to it.

As a reference, **Figure 9.8** shows the complete circuit you should have after you add the second 74HC595 shift register, the 1000-ohm resistor, and the connections.

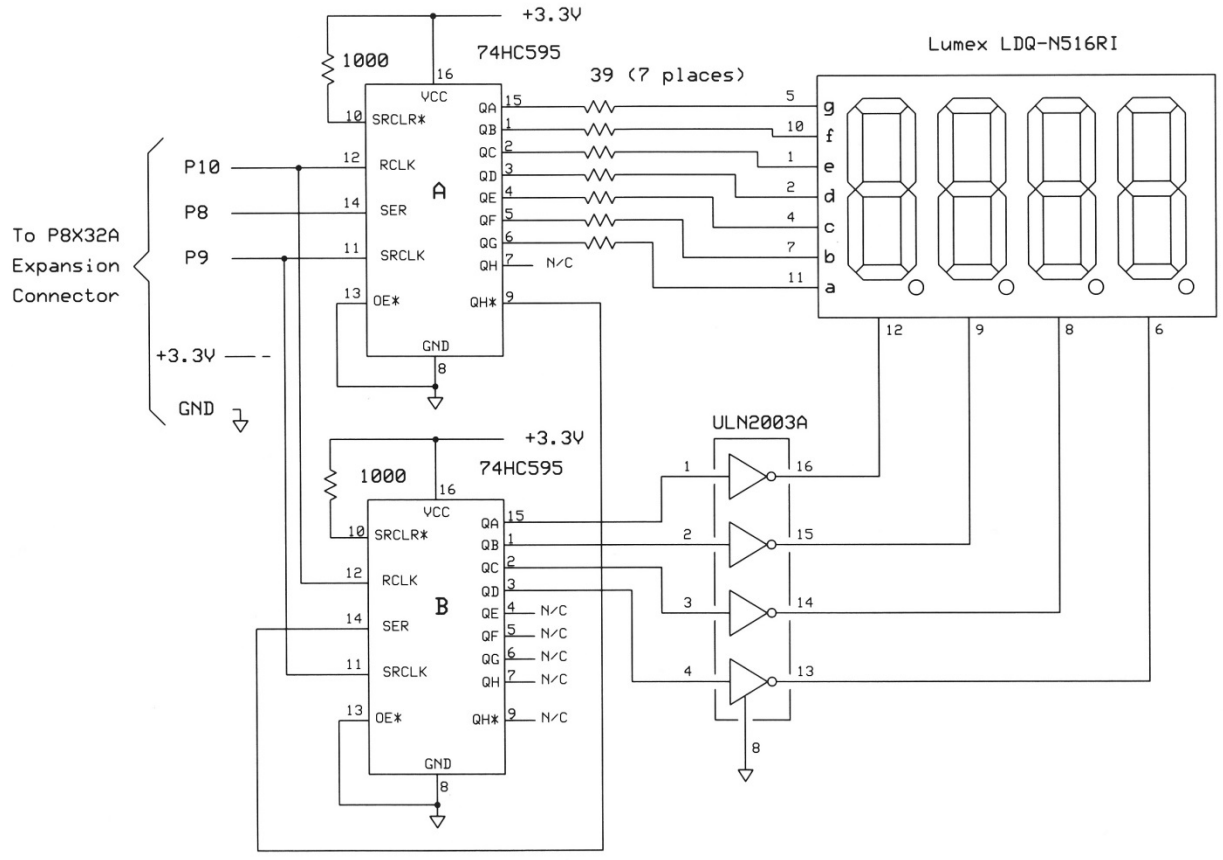


Figure 9.8.
The complete schematic diagram for a 4-digit display that uses two 74HC595 shift registers to control both the display segments and the ULN2003A switches that connect to the common cathode for each digit.

Step 6.

Now you need software to control the 4-digit display. The diagram in **Figure 9.9** shows the two shift registers as blocks and it shows the outputs that control segments and digits. The software could transmit a 16-bit value that contains the digit *and* segment bits to the two shift registers. Or, it could first send an 8-bit value to turn on one *digit* to shift register A. (Keep in mind the new bits will not affect the display because we haven't created an RCLK signal to transfer the newly received data to the outputs.)

Then the program could send another 8-bit value for the *segment* pattern. Shifting in the second eight bits moves the *digit* bits from shift register A into shift register B. Although software transferred two 8-bit values in sequence, the effect is the same as sending all 16 bits at once. The following descriptions relate to the shift registers shown in **Figure 9.9** and explains the 8-bit shift operations used to display numeral 6 on 7-segment digit 2.

a) The MCU creates the pattern to turn on digit 2 in the 4-digit display.

b) The SPI `SHIFTOUT` statement transmits the digit-2 control bits to shift register A.

c) The MCU obtains the segment pattern for the numeral 6.

d) The SPI `SHIFTOUT` statement transmits the segment pattern to shift register A. The digit-2 control bits move from shift-register A to shift-register B.

e) The MCU creates an RCLK pulse for both shift registers, which simultaneously output their bits to the display. (The digit-control bits go to the ULN2003A IC.)

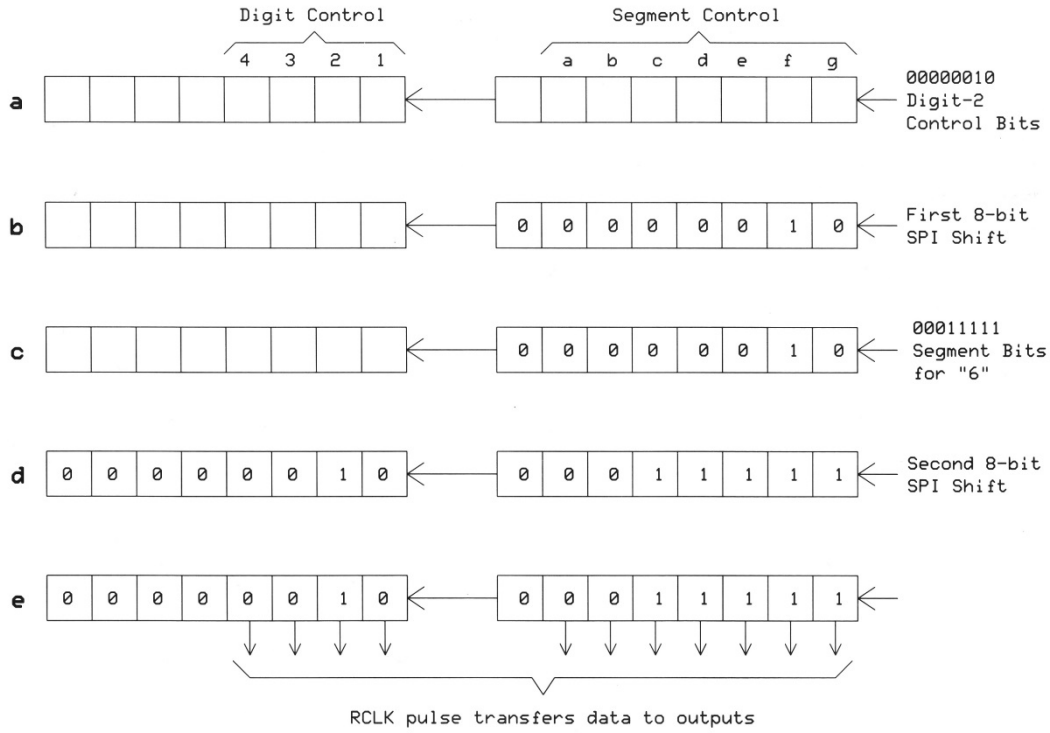


Figure 9.9.
These five steps illustrate how two 8-bit SPI transmissions to the shift registers connected in series properly set up information to select a digit in the display and to control the segments to display the numeral 6.

You can load **Program 9.2** and run it. The results might not look exciting because the display does the same thing as it did when you ran **Program 9.1**. The main change occurred in `LOOP 3` as explained in Step 7. By using only serial communications with the two 74HC595 ICs, you have four "extra" I/O pins on the Propeller P8X32A board. Shift registers A and B have five unused output pins. If necessary, you could use four outputs to select a decimal point, and one to turn a minus sign on or off. If you need only three digits, say, for a temperature reading, the "g" segment in the left-hand digit could serve as a minus sign.

Program 9.2.

```
{{
''*********************************************
''*  Program 9.2
''*  Author: Jon Titus 11-18-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration that separates a
''*  4-digit value into separate numerals and
''*  uses two 74HC595 shift registers to control
''*  display segments.
''*  SPI_Asm object library used for SPI-type
''*  8-bit 7-segment patterns and digit selection.
''*  LUMEX LDQ-N516RI 4-digit display module.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
```

```
    _xinfreq = 5_000_000          'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"       'Invoke the SPI_Asm.spin file
                                  'saved in Propeller working
                                  'folder.
VAR
    byte segment_code[10]         'Storage for segment codes
    byte count                    'Counter used in display loop
    byte display_loop             'Counter used for display-
                                  'loop delay
    byte digit[6]                 'Space for 6 digits
    long big_value                'Variable for large value
    byte numb_of_digits           'Number of digits in
                                  'big_value
    long divisor                  'Divisor used to start
                                  'parsing
    byte index                    'Index to identify each digit
    byte decimal_counter          'Counter to track each
                                  'decimal digit
    byte ser_data                 '8-bit serial data
    byte data_port                'Data-output pin to shift reg
    byte serial_clock             'Pulse for shift reg clock
    byte register_clock           'Pulse for register-transfer
                                  'clock
    byte digit_enable             'Select digits 1 to 4

PUB Start                         'Main program starts here

segment_code[0] := %01111110      'Pattern for "0"
segment_code[1] := %00110000      'Pattern for "1"
segment_code[2] := %01101101      'Pattern for "2"
segment_code[3] := %01111001      'Pattern for "3"
segment_code[4] := %00110011      'Pattern for "4"
segment_code[5] := %01011011      'Pattern for "5"
segment_code[6] := %00011111      'Pattern for "6"
segment_code[7] := %01110000      'Pattern for "7"
segment_code[8] := %01111111      'Pattern for "8"
segment_code[9] := %01110011      'Pattern for "9"

''SPI Setup
SPI.start(15,0)
dira[8..10] := %111               'Output for shift register
outa[8..10] := %000               'Clear shift-register outputs
dira[11..14] :=%1111              'Set display-switch outputs
outa[11..14] := %0000             'Clear display-switch outputs

data_port := 8                    'P8 for shift-register data
serial_clock := 9                 'P9 for shift-register clock
register_clock := 10              'P10 for register outputs

big_value := 3716                 'Large value to work with
numb_of_digits := 4               'Number of digits in
                                  'big_value
divisor := 1                      'Starting value of divisor
```

```
'LOOP 1: Given the number of digits, calculate the largest divisor for four
digits, start with 1000, for example.

  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10

'LOOP 2: Use divisor to separate the numerals. Use '(numb_of_digits - 1) so
loop can go from digit[0] to digit[3] 'for a 4-digit value. Store four
values.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10

'LOOP 3: Get the segment data for a numeral and send
'to shift register via SPI object. Transfer data to
'output flip-flops to drive segments, and control ULN2003A
'to turn on corresponding numeral.

'Repeat this loop forever
repeat
   repeat display_loop from 0 to 59
      digit_enable := 1
      repeat count from 0 to 3           'Loop for four digits

         SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, digit_enable)
         digit_enable := digit_enable << 1
         ser_data := segment_code[digit[count]]
         SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, ser_data)

         outa[register_clock] := 1           'Pulse RCLK input
         outa[register_clock] := 0           'on 74HC595 IC
         waitcnt(clkfreq/200 + cnt)          'time delay
' –  – -end –  – -
```

Step 7.

The statements in **Code Snippet 9.3** shows the commands used to control the shift registers.

Code Snippet 9.3.

```
digit_enable := 1
  repeat count from 0 to 3
    SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, digit_enable)
    digit_enable := digit_enable << 1
    ser_data := segment_code[digit[count]]
    SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 8, ser_data)

    outa[register_clock] := 1
    outa[register_clock] := 0
    waitcnt(clkfreq/200 + cnt)
```

In this short section of code, the `repeat count` loop controls the four digits in sequence. The first `SPI.SHIFTOUT` statement sets up the code for the first digit `(digit_enable)` and sends it to the shift register input. The statement:

```
digit_enable := digit_enable << 1
```

shifts the `digit_enable` bit one position to the left so the next time through the loop, the bit for digit 2 will go to the shift register. The second `SPI.SHIFTOUT` statement transfers the segment pattern to the shift register, and the `outa[register_clock]` statements generate an RCLK pulse that puts out all 16 bits from the two shift registers for the display.

Step 8.

At the start of Step 6 I noted the two 74HC595 shift registers connected in series create a 16-bit shift register you could load all at once with a 16-bit value. Other operations in a program must combine the eight bits for shift-register B with those for shift-register A to create a 16-bit value for the `SPI.SHIFTOUT` command. To combine the two values, I defined a word (16 bits), `data16_out`, for the result. I also redefined `digit_enable` as a word variable. **Code Snippet 9.4** shows *only* the SPI portion of my program. (I'll show a complete program shortly.)

Code Snippet 9.4.

```
repeat count from 0 to 3
  ser_data := segment_code[digit[count]]
  data16_out := 0
  data16_out := (digit_enable << 8) | ser_data

  SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16,  data16_out)

  digit_enable := digit_enable << 1
  outa[register_clock] := 1
  outa[register_clock] := 0
  waitcnt(clkfreq/200 + cnt)
```

After I made these changes in **Program 9.2** and ran it the display showed... nothing! It was dark. Why would combining two 8-bit SPI-output operations into one 16-bit SPI-output operating cause the program to not work?

Time for troubleshooting. Often the behavior of hardware provides a clue about software problems, so I used a dual-trace PropScope to view the SPI MOSI output and the SPI SCLK signals. **Figure 9.10** shows the results the scope captured for digit 2 (second from the left on the display) and the numeral 7. That information looked correct, given my test number 3716 and the location of the 7 in it. (Remember, digit positions in the display go 3, 2, 1, and 0, left to right.)

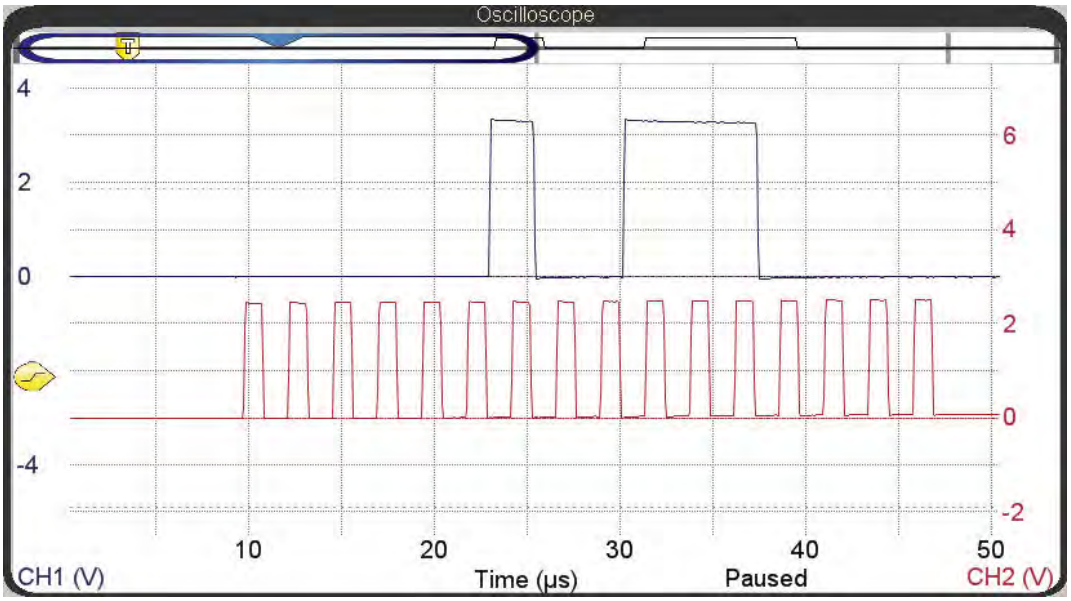


Figure 9.10.
Oscilloscope traces for a 16-bit output on the SPI MOSI pin (upper) and the serial clock, SCLK (lower). This information appears correct for the numeral 7 at position 2 on the display.

The 16-bit data looked fine, so next I looked at the RCLK pulse that controls the shift-register output flip-flops, and the SPI SCLK signal. **Figure 9.11** shows the results. You'll see the RCLK signal goes to a logic-1 *before* the end of the SPI transmission, but it should go to a logic-1 only *after* the SPI transmission ends. Why would the RCLK signal go to a logic-1 state prematurely?

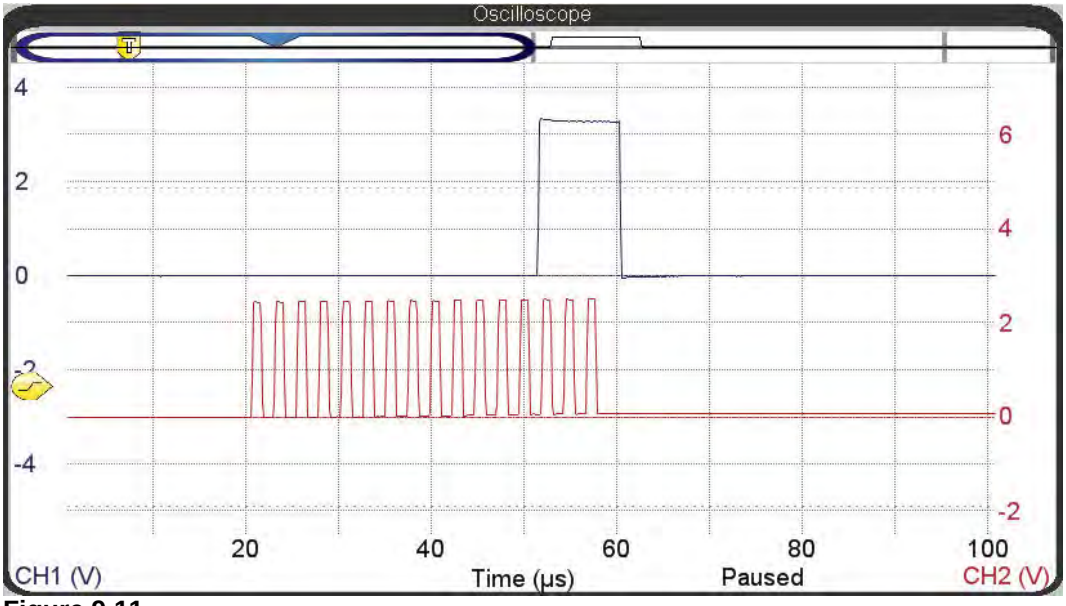


Figure 9.11.
This oscilloscope image shows traces for the RCLK signal (upper) and the serial clock, SCLK (lower). The RCLK signal has incorrect timing and might cause the problem.

When you look at the SPI_Asm.spin program you will see the following comments at about line 75:

```
'' Start SPI Engine - starts a cog
'' returns false if no cog available
```

This information lets us know the SPI operations start another cog (another processor) that handles the SPI tasks. When an `SPI.SHIFTOUT` operation starts, the object software runs *on the second cog*, which goes about its tasks and sends out the 16 bits of data. My main program continues to run; *it does not wait until the second cog completes the SPI transmission*.

I inserted a short delay – about 20 μsec – after the `SPI.SHIFTOUT` statement to give the 16-bit transmission time to finish in the second cog, and that solved the problem.

```
waitcnt(clkfreq/100000 + cnt)
outa[register_clock] := 1
outa[register_clock] := 0
```

If you wish, run **Program 9.3**, which corrects the timing problem. I have abbreviated many comments that you can find in the **Programs 9.1** and **9.2**.

Program 9.3

```
{{
''*********************************************
''*  Program 9.3
''*  Author: Jon Titus 11-04-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration that separates a
''*  4-digit value into separate numerals and
''*  uses 74HC595 shift register to control
''*  display segments.
''*  SPI_Asm object library used for SPI-type
''*  of 8-bit 7-segment patterns
''*  LUMEX LDQ-N516RI 4-digit display module.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"        'Invoke the SPI_Asm.spin file

VAR
    byte segment_code[10]          'Storage for segment codes
    byte count                     'Counter used in display loop
    byte display_loop              'Counter for delay
    byte digit[6]                  'Space for 6 digits
    long big_value
    long divisor
    byte index
    byte decimal_counter
    byte ser_data
    byte data_port
    byte serial_clock
    byte register_clock
    word digit_enable
    word data16_out

PUB Start                          'Main program starts here
```

```
segment_code[0] := %01111110       'Pattern for "0"
segment_code[1] := %00110000       'Pattern for "1"
segment_code[2] := %01101101       'Pattern for "2"
segment_code[3] := %01111001       'Pattern for "3"
segment_code[4] := %00110011       'Pattern for "4"
segment_code[5] := %01011011       'Pattern for "5"
segment_code[6] := %00011111       'Pattern for "6"
segment_code[7] := %01110000       'Pattern for "7"
segment_code[8] := %01111111       'Pattern for "8"
segment_code[9] := %01110011       'Pattern for "9"

''SPI Setup
  SPI.start(15,0)

dira[8..10] := %111                'Output pins for shift reg
outa[8..10] := %000                'Clear shift register
dira[11..14] :=%1111               'Set display-switch outputs
outa[11..14] := %0000              'Clear display-switch outputs

data_port := 8
serial_clock := 9
register_clock := 10

big_value := 3716                  'Large value to work with
numb_of_digits := 4                'Number of digits in
                                         'big_value
divisor := 1                       'Starting value of divisor

'LOOP 1: Given the number of digits, calculate divisor.
  repeat decimal_counter from 1 to (numb_of_digits - 1)
    divisor := divisor * 10


'LOOP 2: Use divisor to separate the numerals.

  repeat index from 0 to (numb_of_digits - 1)
    digit[index] := big_value / divisor
    big_value := big_value - (digit[index] * divisor)
    divisor := divisor / 10

'LOOP 3: Get the segment data for a numeral and send
'to shift register.

repeat
    repeat display_loop from 0 to 59
        digit_enable := 1
        repeat count from 0 to 3
            ser_data := segment_code[digit[count]]
            data16_out := 0
            data16_out := (digit_enable << 8) | ser_data

            SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16,
data16_out)

            digit_enable := digit_enable << 1
            waitcnt(clkfreq/100000 + cnt)     'Delay, for SPI to
```

```
                                        'complete
            outa[register_clock] := 1   'Pulse RCLK input
            outa[register_clock] := 0   'on 74HC595 IC
            waitcnt(clkfreq/200 + cnt)  'time delay

 ' -  - -End -  - -
```

Step 9.

Please leave your circuit set up for use in the next experiment. The information that follows explains how the program puts together the 16-bit word to transmit to the shift registers. Skip this step if you have no interest in the details.

Here's how **Program 9.3** operates to display four digits for a 16-bit SPI transmission:

a. `digit_enable = 1` to start.

b. `ser_data` holds the bit pattern for the first digit.

c. `data16_out` gets set to 0.

d. The operation `data16_out := (digit_enable << 8) | ser_data` shifts the `digit_enable` bit 8 positions to the left and the bitwise OR operation (`|`) "drops" the `ser_data` into the low byte of `data16_out`.

e. The `SPI.SHIFTOUT` command sends all 16 bits to the 74HC595 shift registers.

Step **d** might seem confusing at first, so the example below should clarity the operations. All the numbers use binary notation, and the underscore in these values serves as a visual separator to make numbers easier to read. It has no effect on a Spin program.

```
digit_enable = 00000000_00000001
ser_data =              01010110
data16_out =   00000000_00000000

digit_enable << 8 = 00000001_00000000

bitwise OR of 00000001_0000000 and 010101100:

       00000001_00000000
OR              01010110
Result 00000001_01010110
```

`SPI.SHIFTOUT` sends: `00000001_01010110` to the shift registers.

Reference

"SPI/I²C Bus Lines Control Multiple Peripherals," Application Note 4024, Maxim Integrated, 2007. http://www.maximintegrated.com/app-notes/index.mvp/id/4024.

Experiment No. 10 – How an MCU Controls an LED Matrix

Abstract

This experiment lets you work with 35 LEDs arranged in a 5-by-7 matrix and controlled by the Propeller MCU. You have seen LED-matrix signs in stores, airline terminals, movie theaters, and on billboards and kiosks. So you probably have wondered how they worked and if you could create something similar. You will learn how this type of display works and you will have the software needed to control it.

Keywords

shift register, 74HC595, serial, LED matrix display, flip-flop, ULN2003A, multiplex, transistor array, serial-peripheral interface, SPI, Propeller, object, cog, arrays, software

Requirements

(2) - 74HC595 8-bit serial-in, parallel-out shift registers, 16-pin DIP
(1) - ULN2003A high-current transistor array, 16-pin DIP
(7) - 39-ohm, 1/4W, 5% resistors (orange-white-black)
(7) - 22-ohm, 1/4W, 5% resistors (red-red-black)
(2) - 1000-ohm, 1/4W, 5% resistor (brown-black-red)
(1) - LiteOn 5-by-7 LED matrix, part no. LTP-757G, or equivalent (see text)
(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - USB cable

Introduction

An LED matrix comprises individual LEDs positioned in a 2-dimensional array. LED matrices have equal spaces between the LEDs, but you can place LEDs in any sort of array you want. To simplify the design, assembly, and maintenance of alphanumeric displays, manufacturers place LEDs in ready-to-use fixed arrays. They place LED die – the individual tiny LED semiconductors – on a substrate material that connects LEDs electrically in x-and-y planes. A plastic frame placed onto the substrate isolates the light from each LED and concentrates the light so it radiates forward to a viewer's eyes. This type of LED arrangement provides the capability to display numerals, letters, symbols, and any pattern or design you can fit in the display area. In return for that flexibility, it takes more software and circuits to control an LED matrix than to control a 7-segment display.

Matrix displays come in many formats such as 5-by-7, 5-by-8, and 8-by-8, and costs start at just over $US 2 and increase with the size of the matrix, the number of "dots," the dot color, and whether the matrix can display more than one color. Full-color matrices have a red, blue, and green LED at each position so signs can show color images. Control of this type of LED matrix goes beyond the scope of this experiment.

A 7-segment display requires seven signals – one for each segment – and one for power or ground. All seven segments (and an optional decimal point) turn on simultaneously. In Experiment 9 you saw how to control a 4-digit 7-segment display module with only 3 wires.

When you have a 5-by-7 LED matrix, things get more complicated. A control circuit must drive the seven horizontal rows and five columns of LEDs. The diagram in **Figure 10.1** shows the connections for each display type. You might remember that 7-segment LED displays come in a common-cathode or a common-anode form. LED matrices have similar configuration types: common-cathode rows and common-anode

columns, or common-anode rows and common-cathode columns. The schematic diagram in **Figure 10.2** shows both configurations. This experiment requires an LED matrix with *common-anode rows* and *common-cathode columns*. I used a LiteOn LTP-757G LED matrix with green LEDs, but you may choose an equivalent 5-by-7 matrix.

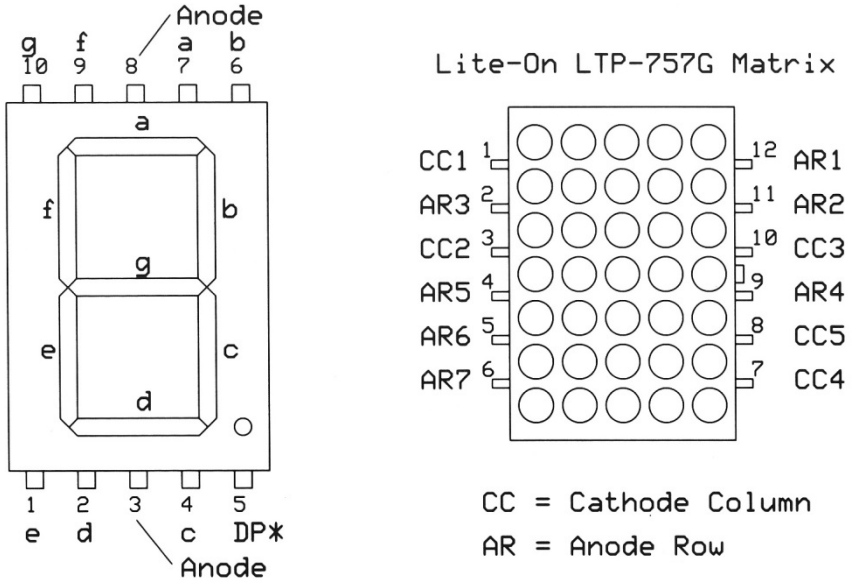


Figure 10.1.
A 7-segment display (left) requires fewer connections than a 5-by-7 LED matrix, but the matrix can display any letter, number, or symbol you can create on the 35 discrete LEDs. Note the small plastic tab between pins 9 and 10 on the right side of the matrix package. This type of "indicator" lets you determine the proper orientation of the device.

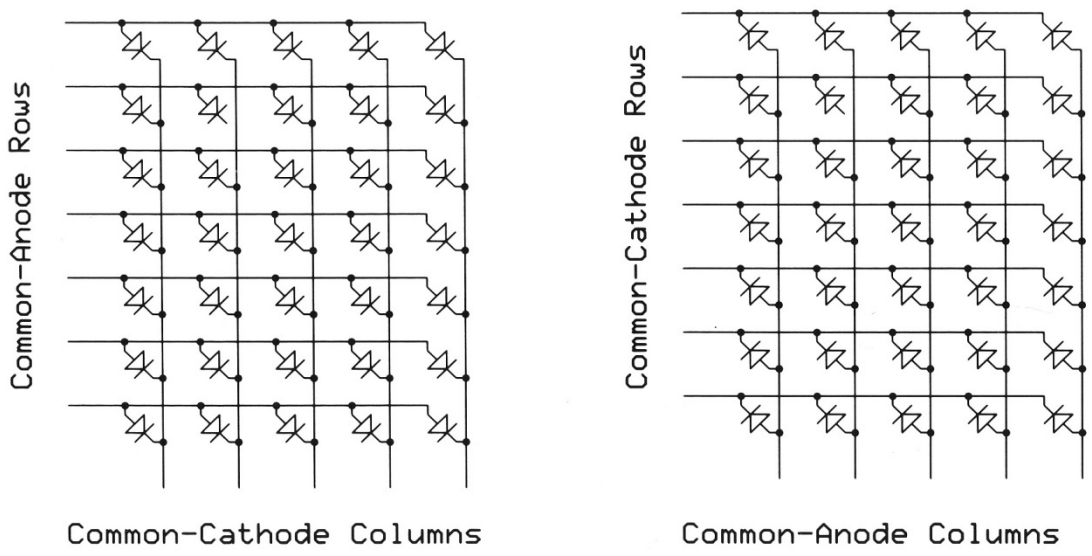


Figure 10.2.
This diagram shows the connections for two arrangements of LEDs in a 5-by-7 matrix. Circuit designers can supply current to the rows and connect the columns to ground (left). Or they can supply current to the columns and ground the rows (right).

Regardless of the anode and cathode arrangement, each 5-by-7 LED matrix requires seven row connections and five column connections. Likewise, an 8-by-8 matrix requires eight row and eight column connections. As a result, it takes more external circuitry to drive an LED matrix than to drive a 7-segment display. Also,

because you must create a large table of 1's and 0's the numerals, symbols, and letters, an LED matrix-control program requires a large amount of memory. It takes one byte to store a 7-segment pattern, but it takes eight bytes for *one* 8-by-8 symbol or character.

This experiment builds on the last circuit created in Experiment 9 and shown in **Figure 9.8**. If you have not yet run that experiment, please do so now so you understand how the circuit works, and have the circuit tested and on your breadboards.

Step 1.

After you completed Experiment 9, your breadboards should have contained the circuit shown in **Figure 10.3.** To prepare this circuit for a matrix display you must make several changes.

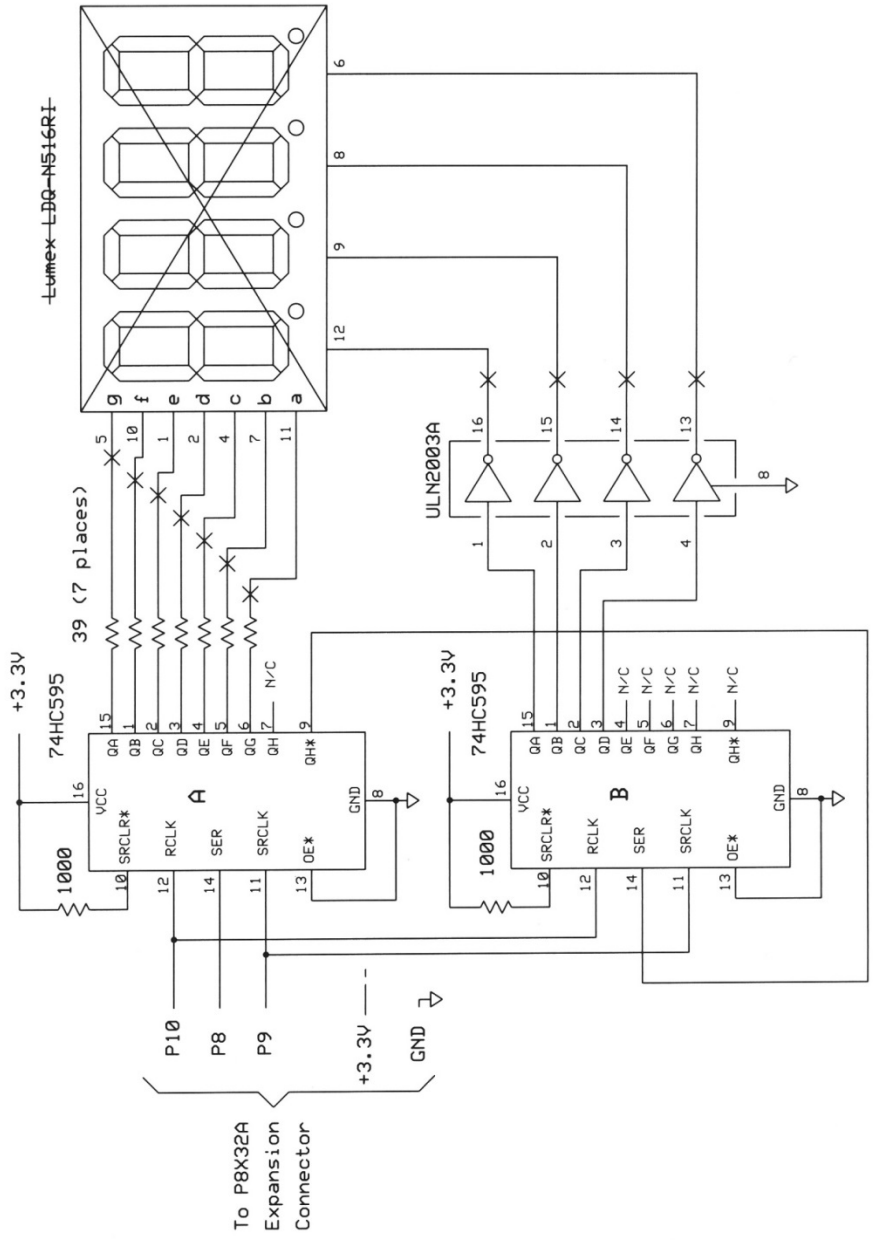


Figure 10.3.

This diagram shows modifications needed on the numeric-display-module to ready it for a 5-by-7 LED matrix. Before you make any changes, read the remaining text in Step 1.

First, the LED matrix might need different resistances to limit LED current. I used a Lite-On LTP-757G device that has an 11-mA *average* forward current (I_f) and a 2.1-V drop across the LED (V_f). The Propeller P8X32A board supplies 3.3 volts to my breadboards, so I used Ohm's Law to calculate the needed resistance; 109 ohms.

But because the software turns on only one of the five columns at a time, individual LEDs remain on only for a short period. For the 5-by-7 LED matrix, you can assume about a 20-percent duty cycle because each column will connect to ground for about 1/5th of the time.

The LiteOn LTP-757G data sheet provides a graph that plots peak forward current (IP) vs. duty cycle. For a 20-percent duty cycle at 1000 Hz, the maximum LED current equals 60 mA. Ohm's Law calculates the series resistance at 20 ohms. I had 39-ohm resistors on hand and used them. You may use 20- or 22-ohm resistors instead.

Second, several components and wires shown in **Figure 10.3** have an "X" marked through them to indicate you will remove or replace them. Follow the lettered directions below:

a) Remove the 4-digit display module and set it aside. You will not use it in this experiment.

b) Leave the seven 39-ohm resistors in your breadboard or replace them with seven 22-ohm (red-red-black) resistors. (If you will use a different display type, check its data sheet for the peak-current value for a 20-percent duty cycle and calculate the current-limiting resistance you need in place of the 22- or 39-ohm resistors .)

c) Remove the seven wires that connect the resistors to the display-module inputs, *a* through *g*. You should still have the wires between the 74HC595 shift register and the seven resistors.

d) Remove the four wires that connect the ULN2003A IC to the display-module cathodes.

Step 2.

Insert in your breadboard a 5-by-7 LED matrix display with *common-anode rows* and *common-cathode columns*. The LiteOn green LTP-757G matrix uses a 12-pin dual inline package (DIP) that can straddle the mid-breadboard channel as would an IC. The data sheet for this display lacks clear information about pin numbers and display orientation, so refer to **Figure 10.1** for this information. The circuit diagram in **Figure 10.4** shows the necessary connections for this experiment. Note that the circuit uses a fifth driver in the ULN2003A IC.

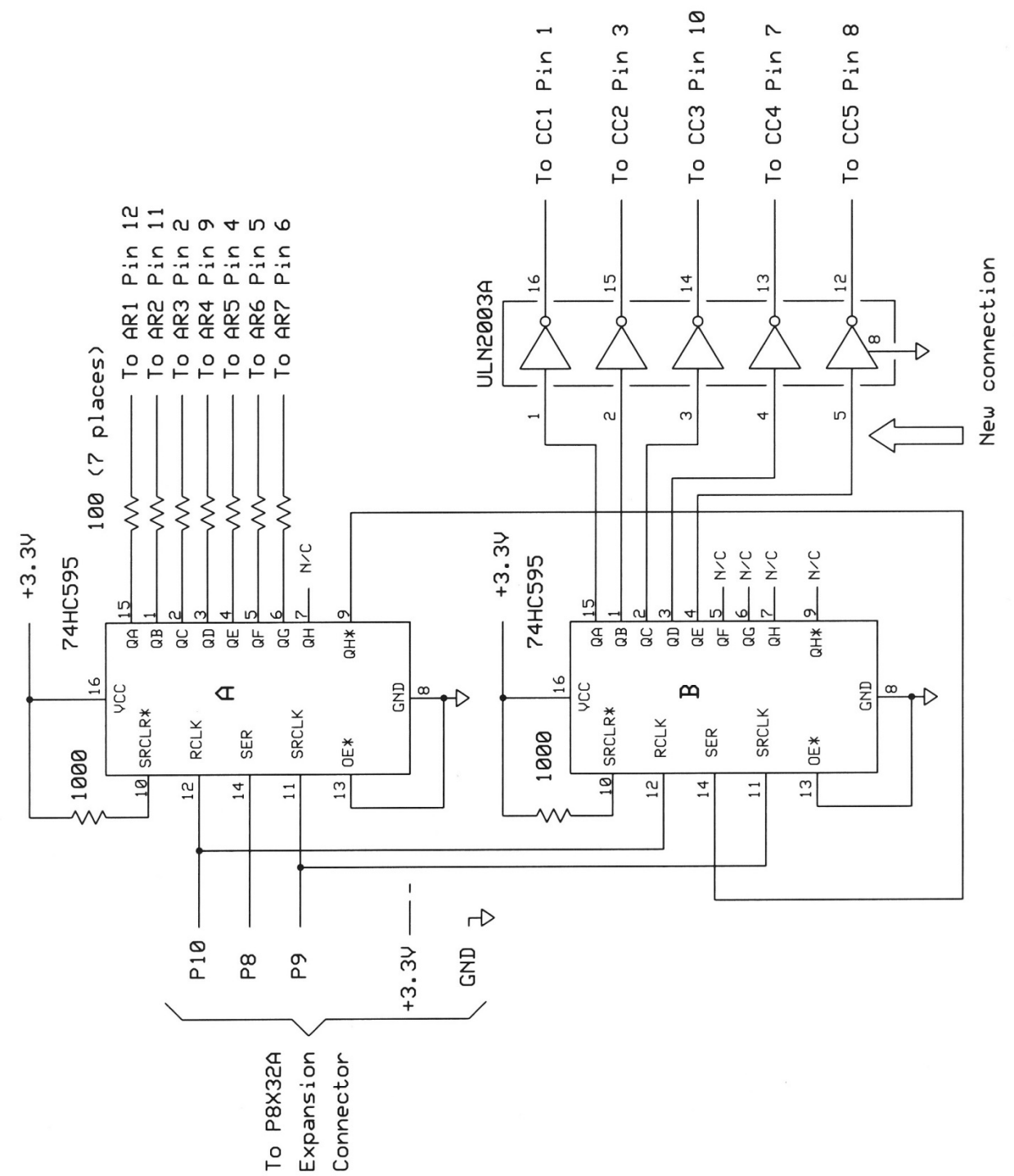


Figure 10.4.
Schematic diagram for Propeller MCU control of one 5-by-7 LED matrix with common-anode rows and common-cathode columns. Note the new connection (large arrow) between 74HC595 shift register B and the ULN2003A IC.

Follow steps a through c below to wire the LED matrix. Check off the connections in **Figure 10.4** as you make them.

a. Connect the QE output (pin 4) on the lower 74HC595 shift register (B) to the 5B input (pin 5) on the ULN2003A.

b. Connect the unconnected ends of the seven resistors attached to shift-register A to the corresponding row-input pins on the LED matrix. These connections need seven wires.

c. Connect the five outputs on the ULN2003A IC to the corresponding column pins on the LED matrix.

Recheck your wiring. You should have 12 wires from the control ICs to the LED matrix. Confirm you have the connection explained in step **a** above between the 74HC595 and the ULN2003A ICs.

Step 3.

Now run **Program 10.1** to test the LEDs. All seven LEDs in a column will turn on and "move" from left to right. At first, it will appear that all 35 LEDs turn on simultaneously. If you see a row of unlit LEDs, check connections from the 74HC595 shift register (A) through the resistors to the AR (anode-row) pins on the LED matrix. If you see a column of unlit LEDs, check the connections from the ULN2003A IC to the CC (cathode-column) pins on the matrix. Remember: Connect a wire from pin 4 on the 74HC595 shift-register IC (B) to pin 5 on the ULN2003A IC. It's easy to overlook this connection.

Program 10.1.

```
{{
''*********************************************
''*  Program 10.1
''*  Author: Jon Titus 11-18-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration for a single 5-by-7
''*  LED matrix and 74HC595 shift registers that
''*  control columns and rows of LEDs.
''*  Test program to check LED connections.
''*  All LEDs should turn on.
''*  SPI_Asm object library used for SPI-type
''*  communications with the 74HC595 shift registers.
''*  LiteOn LTP-757G LED matrix module.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"         'Invoke SPI_Asm.spin

VAR
    byte pattern[10]            'Storage for character patterns
    byte count                  'Counter used in display loop
    byte display_loop           'Counter used for display-loop
                                'delay
    byte ser_data               '8-bit serial data for LED
                                'columns
    byte data_port              'Data-output pin to shift reg
    byte serial_clock           'Pulse for shift reg clock
    byte register_clock         'Pulse for register-transfer
                                'clock
    word digit_enable           'Value to select column
```

```
    word data16_out             '16-bit SPI transmission

PUB Start                       'Main program starts here
pattern[0] := %11111111         'Test pattern to light all LEDs

''SPI Setup
  SPI.start(15,0)

dira[8..10] := %111             'Pins for shift register
outa[8..10] := %000             'Clear shift-register outputs
data_port := 8                  'P8 for shift-register data
serial_clock := 9               'P9 for shift-register clock
register_clock := 10            'P10 register flip-flop outputs

'LOOP: Get pattern data for 5-by-7 LED matrix and send to shift
'register via SPI object. Transfer data to output flip-flops to 'drive LED
rows, and control ULN2003A to turn on corresponding 'LED column.

'Repeat this loop forever
 repeat
    repeat display_loop from 0 to 59
       digit_enable := 1
       repeat count from 0 to 4          'Loop for five columns
            ser_data := pattern[0]       'Get test pattern
            data16_out := 0              'Clear output word

            data16_out := (digit_enable << 8) | ser_data

            SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16,
data16_out)

            digit_enable := digit_enable << 1

            waitcnt(clkfreq/100000 + cnt)
            outa[register_clock] := 1
            outa[register_clock] := 0
            waitcnt(clkfreq/2000 + cnt)    'Refresh time delay

 ' –  – -End –  – -
```

In **Program 10.1**, you again use the SPI_Asm.spin objects to transfer data from the Propeller MCU to the shift registers. This program repeats the pattern 11111111_2 for each column, so all LEDs appear on. The 7-row display does not use the MSB from shift register A.

Change the 8-bit binary value assigned to `pattern[0]` and rerun the program to observe the effect of the 1's and 0's on the LED rows. The rows of lit LEDs correspond to the 1's in the `pattern[0]` value.

Set `pattern[0]` so it contains a mix of 1's and 0's. Then decrease the value 2000 to 200 in the statement:

```
waitcnt(clkfreq/2000 + cnt)
```

What happens to the "refresh rate" on the display when you again run the program? The refresh rate decreases and your peripheral vision might let you observe the LEDs flicker slightly. Run the program again but with a value of 2 instead of 200. Now what happens on the display? The slower refresh rate lets you see the LEDs

turn on or off one column at a time. Did you notice anything else about the LEDs? The LEDs might appear brighter because they remain on for a longer time.

Step 4.

The 5-by-7 LED matrix must do more than displays the same pattern in each column. To display a digit, letter, or symbol, we need an array of bytes that provide the on-or-off representations for the 35 LEDs. The uppercase "A," shown in **Figure 10.5**, requires five binary or hexadecimal values, one value per column.

LED Matrix	Column						Column				
Row	1	2	3	4	5		1	2	3	4	5
a		■	■	■			0	1	1	1	0
b	■				■		1	0	0	0	1
c	■				■		1	0	0	0	1
d	■	■	■	■	■		1	1	1	1	1
e	■				■		1	0	0	0	1
f	■				■		1	0	0	0	1
g	■				■		1	0	0	0	1

Figure 10.5.
The arrangement of lit LEDs that display an uppercase letter A. Each character, number, and symbol requires its own set of five values. A dark square represents a 1, while an open square represents a 0. In hexadecimal notation, the values equal 7E, 09, 09, 09, and 7E for the letter A, from the left to the right column.

Find hexadecimal values for the uppercase letters at: http://www.edaboard.com/thread225655.html . I found no source of such information for 5-by-7 lowercase characters or symbols. The data sheet for a Holtek HT16523 5x7 Dot Character VFD [vacuum fluorescent display] Controller & Driver shows the on-off LED patterns, but you'll have to extract the binary information on your own: www.holtek.com/pdf/Display_Driver/ht16523v100.pdf.

Step 5.

Program 10.2 will display the letter A to further test the 5-by-7 LED matrix. The statements in this program vary slightly from those used in **Program 10.1**. Instead of using a one-column test pattern, **Program 10.2** uses five locations, `pattern[0]` through `pattern[4],` which hold the column values for an uppercase A as shown above. The program includes a loop, `repeat count from 0 to 4`, that sends the first pattern to the display and turns on the first column. The loop then turns off the first column, sends the second pattern, turns on the second column, and so on. Load and run **Program 10.2**. The LED matrix should show an A.

Program 10.2.

```
{{
''*********************************************
''*  Program 10.2
''*  Author: Jon Titus 11-18-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration for a single alpha-
''*  numeric character on a 5-by-7 LED matrix.
''*  74HC595 shift registers control columns
''*  and rows of LEDs.
''*  SPI_Asm object library used for SPI-type
''*  communications with the 74HC595 shift registers.
''*  LiteOn LTP-757G LED matrix module.
''*********************************************
}}
```

```
CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"         'Invoke SPI_Asm.spin file

VAR
    byte pattern[10]                'Storage for character
                                            'patterns
    byte count                   'Counter used in display loop
    byte display_loop            'Counter for display-loop delay
    byte ser_data                '8-bit serial data for LED
                                 'columns
    byte data_port               'Data-output pin to shift reg
    byte serial_clock            'Pulse for shift reg clock
    byte register_clock          'Pulse for register-transfer
                                 'clock
    word column_enable           'Value to select column
    word data16_out              '16-bit value for SPI

PUB Start                        'Main program starts here

pattern[0] := $7E                '5-byte pattern for letter A
pattern[1] := $9
pattern[2] := $9
pattern[3] := $9
pattern[4] := $7E


''SPI Setup
  SPI.start(15,0)

dira[8..10] := %111               'Outputs for shift register
outa[8..10] := %000               'Clear shift-register

data_port := 8                    'P8 for shift-register data
serial_clock := 9                 'P9 for shift-reg clock
register_clock := 10              'P10 register flip-flops

'Get the column data for a character and send it to the shift
'registers via SPI object. Transfer data to output flip-flops
'to drive rows and control ULN2003A to turn on corresponding 'column.

'Repeat this loop forever
repeat
   repeat display_loop from 0 to 59
      column_enable := 1
      repeat count from 0 to 4            'Loop for five columns
         ser_data := pattern[count]       'Get pattern for each
                                          'column
         data16_out := 0                  'Clear output word
         data16_out := (column_enable << 8) | ser_data

         SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16, data16_out)
         column_enable := column_enable << 1   'Shift enable bit
```

```
        waitcnt(clkfreq/100000 + cnt)    'Delay, wait for SPI
                                         'to end
        outa[register_clock] := 1        'Pulse RCLK input
        outa[register_clock] := 0        'on 74HC595 IC

        waitcnt(clkfreq/2000 + cnt)      'Refresh time delay

 ' –  – -End –  – -
```

Step 6.

The Propeller SPIN language lets programmers save information in many ways. When you have ordered information, such as the patterns for numbers, letters, and symbols, a linear array offers a good structure. The American Standard Code for Information Interchange (ASCII) used to represent printing characters and control codes comprises 128 values, from 0 to 127. The first 32 values correspond to non-printing codes, so for a display, you can ignore them. The remaining 96 codes (32 to 127) represent standard symbols, numbers, and upper- and lower case letters. For the sake of clarity, and to save memory, I limit the display characters to two symbols (`!` and space) and four letters (`A`, `B`, `C`, and `K`). You may insert the 5-byte codes for other characters as you wish.

The array that saves the 5-byte codes for each of the 96 characters requires 96 * 5, or 480 bytes of storage, defined as shown below and followed by the bytes for six characters, "space," `!`, `A`, `B`, `C`, and `K`. Note the "jump" between the index numbers used for the `char_patterns` array as it goes from `!` to `A`, and again from `C` to `K`. Other characters would "fill in" those spaces in the array.:

```
VAR
      byte char_patterns[480]

PUB Start
      char_patterns[0]  :=  $00     'Column 1, "space"
      char_patterns[1]  :=  $00     'Column 2
      char_patterns[2]  :=  $00     'Column 3
      char_patterns[3]  :=  $00     'Column 4
      char_patterns[4]  :=  $00     'Column 5

      char_patterns[5]  :=  $00     'Column 1, "!"
      char_patterns[6]  :=  $00
      char_patterns[7]  :=  $4F
      char_patterns[8]  :=  $00
      char_patterns[9]  :=  $00

      char_patterns[165]  :=  $7E   'Column 1, "A"
      char_patterns[166]  :=  $09
      char_patterns[167]  :=  $09
      char_patterns[168]  :=  $09
      char_patterns[169]  :=  $7E

      char_patterns[170]  :=  $7F   'Column 1, "B"
      char_patterns[171]  :=  $49
      char_patterns[172]  :=  $49
      char_patterns[173]  :=  $49
      char_patterns[174]  :=  $36

      char_patterns[175]  :=  $3E   'Column 1, "C"
      char_patterns[176]  :=  $41
      char_patterns[177]  :=  $41
```

```
      char_patterns[178]  :=  $41
      char_patterns[179]  :=  $22

      ....

      char_patterns[215]  :=  $7F   'Column 1, "K"
      char_patterns[216]  :=  $08
      char_patterns[217]  :=  $14
      char_patterns[218]  :=  $22
      char_patterns[219]  :=  $41
```

Given this type of array, how can a program locate the proper five bytes to display for a character? A bit of simple math does the job based on the ASCII value for each character. The letter C has an ASCII value of 67. Because we don't store the first 32 ASCII codes that won't print anything, take the ASCII value 67 and subtract 32 from it. Then because each character requires five bytes for column bits, multiply the result by 5:

$$(67 - 32) * 5 = 175$$

You can do this math to locate any character's location for its column-1 display data. (Although this experiment uses only the characters defined above, you can calculate where to add others as you wish.) The code section shown next illustrates how to take an ASCII value for a character and locate its five bytes so a loop can display them:

```
'Letter C
ASCII_value := 67

'Calculate an array index for a letter's first column
letter_code := (ASCII_value - 32) * 5

'Loop to display all five columns of character bits for a "C"
repeat count from (letter_code) to (letter_code + 4)
  etc...
```

In the example just shown, the repeat operation must calculate the value `letter_code + 4` each time through the loop. To save MCU processing time, calculate the end value for the loop before it starts:

```
ASCII_value := 67
letter_code := (ASCII_value - 32) * 5

letter_code_end := letter_code + 4

repeat count from (letter_code) to (letter_code_end)
  etc...
```

The use of ASCII values helps others who might review or modify your code. They can quickly interpret the ASCII information as it relates to individual characters. Of course, well-commented code helps, too!

Program 10.3 shows how to use ASCII values to create a message. Of course, with one LED matrix, the letters appear one after another. With additional shift-registers, ULN2003A drivers, and 5-by-7 LED matrices, a program could create a message you could see all at once. **Program 10.3** has a longer display time for each character so you can see them individually on the LED matrix.

Program 10.3.

```
{{
''*********************************************
```

```
''*  Program 10.3
''*  Author: Jon Titus 11-18-2014 Rev. 3
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration that shows how you can
''*  save ASCII characters in 5-byte sections
''*  in the array char_patterns[x] and display
''*  them on a 5-by-7 LED matrix.
''*  SPI_Asm object library used for SPI-type
''*  communications with shift registers
''*  LiteOn LTP-757G LED matrix display used here
''*********************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

OBJ
  SPI     :       "SPI_Asm"
VAR
    byte char_patterns[320]
    byte message[10]                'Message ASCII values go
                                    'here
    byte message_length             'Length of message in bytes
    byte message_loop               'Counter for loop that
                                    'displays characters
    byte char_pointer               'Pointer to character bytes
    byte count                      'Counter for display loop
    byte display_loop               'Counter for display-loop
                                    'delay
    byte ser_data                   '8-bit ser data for columns
    byte data_port                  'Data-output to shift reg
    byte serial_clock               'Pulse for shift reg clock
    byte register_clock             'Pulse for reg-transfer clk
    word column_enable              'Value to select columns
    word data16_out                 '16-bit value for SPI

PUB Start                           'Main program starts here

        char_patterns[165]  :=  $7E   'Column 1, "A"
        char_patterns[166]  :=  $09
        char_patterns[167]  :=  $09
        char_patterns[168]  :=  $09
        char_patterns[169]  :=  $7E

        char_patterns[170]  :=  $7F   'Column 1, "B"
        char_patterns[171]  :=  $49
        char_patterns[172]  :=  $49
        char_patterns[173]  :=  $49
        char_patterns[175]  :=  $36

        char_patterns[175]  :=  $3E   'Column 1, "C"
        char_patterns[176]  :=  $41
        char_patterns[177]  :=  $41
        char_patterns[178]  :=  $41
        char_patterns[179]  :=  $22
```

```
        char_patterns[215]  :=  $7F   'Column 1, "K"
        char_patterns[216]  :=  $08
        char_patterns[217]  :=  $14
        char_patterns[218]  :=  $22
        char_patterns[219]  :=  $41


''SPI Setup
SPI.start(15,0)
dira[8..10] := %111              'Outputs for shift reg
outa[8..10] := %000              'Clear shift-register outputs
data_port := 8                   'P8 for shift-reg data
serial_clock := 9                'P9 for shift-reg clock
register_clock := 10             'P10 for reg flip-flop outputs

message[0] := 65                  ' ASCII "A"
message[1] := 67                  ' ASCII "C"
message[2] := 75                  ' ASCII "K"
message_length := 3               ' 3 characters in message

'Get the column data for a character and send to the shift
'register via SPI object. Transfer data to output flip-flops
'to drive rows and control ULN2003A to turn on
'corresponding column.

repeat
   repeat message_loop from 0 to message_length - 1
            char_pointer := 5 * (message[message_loop] - 32)
      repeat display_loop from 0 to 159   'Long character delay
            column_enable := 1
            repeat count from 0 to 4        'Loop for 5 columns

               'Get pattern for each column
               ser_data := char_patterns[count + char_pointer]

               'Clear output word
               data16_out := 0

               data16_out := (column_enable << 8) | ser_data

               SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16,
data16_out)
               'Shift column-enable bit
               column_enable := column_enable << 1
               waitcnt(clkfreq/100000 + cnt)     'Delay for SPI
               outa[register_clock] := 1         'Pulse RCLK
               outa[register_clock] := 0         'on 74HC595 IC
               waitcnt(clkfreq/2000 + cnt)       'Refresh delay

 ' –  – -End –  – -
```

Step 7.
After the code in **Program 10.3** displays ACK, can you force it to "blank" the display for a short time before it repeats the ACK output again? Try your modification. Then, modify the 3-character message to display CAB.

Step 8.
An LED matrix that flashes one character at a time might suffice in some cases, perhaps to provide an error-code number or letter in equipment. But many displays scroll characters across a series of LED matrices. You can do the same sort of thing even with only one 5-by-7 LED matrix. Software gives you the flexibility to try several options.

So far, the information used to create each character has come from bit patterns, saved in the `char_patterns[x]` array. That technique works well, but it makes more sense to place character patterns in a "buffer" and constantly display the buffer's contents. A buffer, or buffer memory, provides a set of memory locations or registers that temporarily store information. When you print a large document, for example, your computer places printer information in a section of memory defined as a "printer buffer." The computer then sends the buffered information to your printer as it becomes ready for additional pages.

For the 5-by-7 LED matrix, the buffer will comprise five 1-byte memory locations defined by `display_buffer[5]` in the `VAR` section of a program, and addressed as `display_buffer[0]` through `display_buffer[4]`. Before software begins displaying characters, it should clear the buffer so the LED matrix appears blank to start:

```
repeat buffer_counter from 0 to 4
    display_buffer[buffer_counter]  :=  0
```

The display-update portion of a Propeller program follows, but without comments for the sake of clarity. The code listing for the entire program includes comments.

```
repeat display_loop from 0 to 159
    digit_enable  := 1

    repeat buffer_counter from 0 to 4

        ser_data := display_buffer[buffer_counter]
        data16_out := 0
        data16_out := (digit_enable << 8) | ser_data

        SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16,
        data16_out)

        digit_enable := digit_enable << 1

        waitcnt(clkfreq/100000 + cnt)

        outa[register_clock] := 1
        outa[register_clock] := 0

        waitcnt(clkfreq/2000 + cnt)
```

The internal loop, `repeat buffer_counter..`, obtains from the display buffer array a byte for a column of LEDs. It sends the proper serial bits to the 74HC595 shift registers, waits a short period for the SPI transmission to end, and then transfers the data to the output flip-flops on the shift-register ICs.

The outer loop, `repeat display_loop..`, causes the MCU to run through the 5-column display sequence 160 times; enough time so you can see the lit LEDs. Whatever bit pattern you have in the display buffer you see on the 5-by-7 LED matrix. The last statement in the code above creates a delay that governs how long the LEDs in a given column remain lit.

The `repeat buffer_counter` loop runs five times, once per column, during each of the 160 passes through the `repeat display_loop` portion of the software. For a flow chart of this software, see **Figure 10.6.**

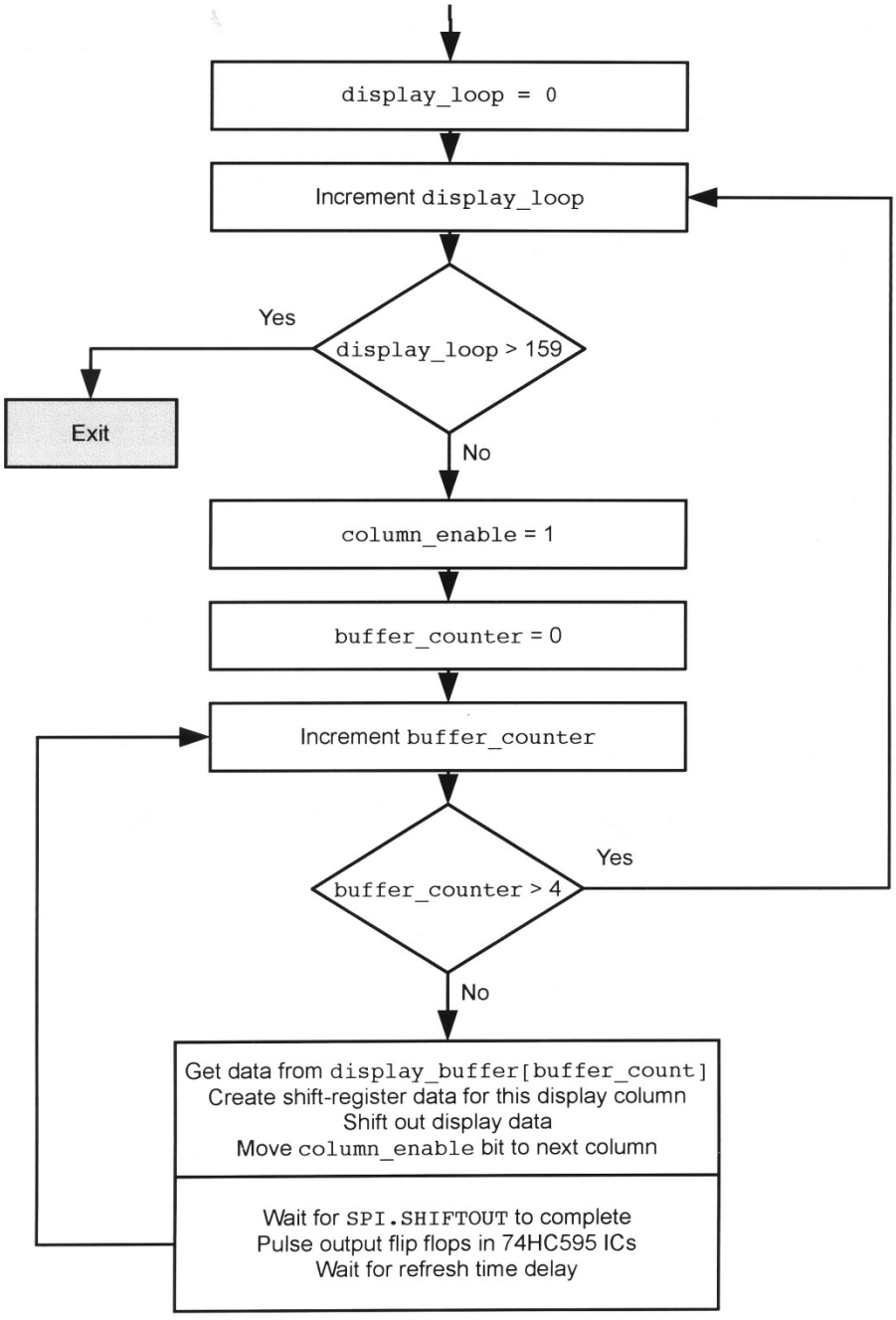


Figure 10.6.
Flow chart for software that will display the bit pattern saved in a display buffer on a 5-by-7 LED matrix.

Now you probably want to know how a display buffer helps create a scrolling display. The diagram in **Figure 10.7** shows how a program would shift bytes into the buffer array to make the letter "A" appear to shift from right to left. Remember, the overall program displays the bit patterns in the buffer again and again. **Figure 10.7a** shows the state of the buffer prior to any new information going in. The letters P, Q, R, S, and T serve only to show how information "moves" in the five buffer locations. In **Figure 10.7b** the Q, R, S, and T

information has moved one location to the left in the buffer. That is, the Q information moved from `display_buffer[1]` into `display_buffer[0]`. The left-most column of bits for the letter "A" has moved into the right-most column, or `display_buffer[4]`. **Figures 10.7c** and **10.7d** show two more shifts of information into the buffer locations. These shifts occur slowly relative to the continuous scan rate of the buffer information and its transfer to the display. As a result, the letter “A” appears to shift into the display from the right.

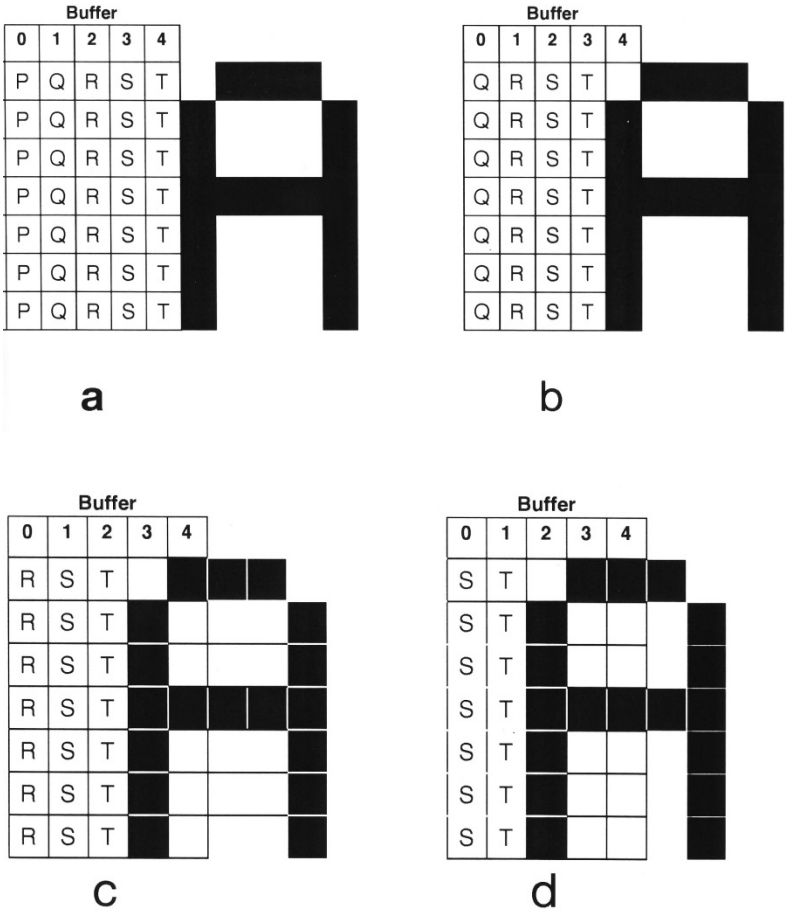


Figure 10.7.
A graphic representation of data moved into the display buffer one column at a time.

The program steps that shift information into the display buffer one column at a time requires only a few statements in a loop:

```
repeat buffer_counter from 0 to 3

  display_buffer[buffer_counter] := display_buffer[buffer_counter + 1]
  display_buffer[4] := char_patterns[char_pointer]
  char_pointer++
  waitcnt(clkfreq/2000 + cnt)
```

The `repeat` statement creates a loop that runs four times and moves data from columns 1, 2, 3, and 4 into columns 0, 1, 2, and 3, essentially a left-shift for these four columns.

The next statement, `display_buffer[4] := char_patterns[char_pointer]`, moves a column of data from the `char_patterns` array into `display_buffer[4].` The `char_pointer` value determines where the program starts to retrieve data for a specific character. In the overall program, I set `char_pointer` to 65 to display the letter "A."

The next-to-last statement in this piece of code increments `char_pointer` so the next time through the loop, the second column of data for the letter "A" moves into `display_buffer[4]`. Remember, before this action occurs, the contents of `display_buffer[4]` got moved into `display_buffer[3]`. So after the second pass through this code, you should see the two left-hand columns of the letter "A" displayed on the 5-by-7 LED array.

In **Program 10.4**, you can change the `ASCII_char` value to 66 to display the letter "B", or to 67 to display the letter "C." What value would you use to display the letter "K"? Run **Program 10.4.** What do you observe? Do you know why you get those results? In the interest of space and to avoid repeating the same code again and again, print **Program 10.4** from the Propeller Tool window if you want a hard copy.

Step 9.

Program 10.4 properly displays the letter "A" again and again, but without any space between characters. That action reveals a flaw. After all, whowantstoreadwithoutspaces? We must modify the program to insert a blank column between characters. You can do this in two ways: First, change all the `char_patterns[x]` data for each character to include a blank column. Each character would require an extra byte, which wastes memory. Second, have the software automatically insert a space after each character. Let's try the second approach because it doesn't use memory to save a space for each character..

The code in **Program 10.5** changes the `message_loop` count from:

```
repeat message_loop from 0 to 4
```

to:

```
repeat message_loop from 0 to 5
```

Program 10.4 limited each character to five columns, 0 through 4. Now we want *six* columns per character; five for the character itself and one "blank" column to separate characters.

In **Program 10.4**, the following statements shift information in the display buffer and inserts the next byte for a character into array location `display_buffer[4]`.

```
repeat buffer_counter from 0 to 4
  display_buffer[buffer_counter] := display_buffer[buffer_counter + 1]
  display_buffer[4] := char_patterns[char_pointer]
```

Program 10.5 includes statements that test the value of `message_loop`, which now runs from 0 to 5. While the `message_loop` variable has a value of 0 through 4, the display gets updated with columns of data for the selected character, just as it did in **Program 10.4**. But, when `message_loop` exceeds four (the same as equalling 5 in this loop), the program will insert a blank column.

I used `if-else` statements to decide what statement to execute based on the value of `message_loop:`

```
if message_loop < 5
    display_buffer[4] := char_patterns[char_pointer]
else
    display_buffer[4] := 0
```

If the `message_loop` variable has a value less than five, the statement right after `if message...` places the next column of information for a character into `display_buffer[4]`, the last location in the buffer.

When the `message_loop` value *equals or exceeds* 5, the software has displayed the given character and has shifted it one column to the left. So the statement after `else` executes and loads `display_buffer[4]` with 0 to create a "blank" column with all LEDs off. Now you should see a blank column between characters when you run **Program 10.5**. Imagine if you had to create a circuit to insert a blank column!

Again, print **Program 10.4** from the Propeller Tool window if you want a hard copy.

Step 10.

Now the program and external circuit display a single scrolling character so you can read it. Can you change the program to display a complete message? In **Programs 10.3** and **10.4**, the variable `ASCII_char` determined the starting location for a character to display. Now you will use an array similar to the one introduced in **Program 10.3**:

```
message[0] := 65                    'A
message[1] := 67                    'C
message[2] := 75                    'K
message[3] := 32                    '"space"
message_length  := 4
```

Program 10.6 combines elements of the previous programs in this experiment. I added the "space" character to the `char_patterns[x]` array so you have a space, or blank display, after the "ACK" message scrolls through the LED matrix. Run the program and see what happens.

Step 11.

Could you add more 5-by-7 LED matrices to the circuit you have? Yes. You must add another 74HC595 shift register and ULN2003A transistor switch to control the column connections on another 5-by-7 LED matrix. And you must change the software to scroll characters across two matrices. Keep in mind that when you control 10 columns instead of five, the duty cycle per column decreases from 20 to 10 percent. As a result, the LEDs will seem dimmer.

The LiteOn data sheet for the LTP-757G LED matrix shows a maximum peak current of 100 mA for a 10% duty cycle. Given a 3.3 volt power supply and an average drop of 3.0 volts across an LED, you can calculate the needed resistance for each row:

$I = E / R$ or $R = E / I$

$R = 0.3\ volts / 0.100\ A = 3\ ohms$!

That's a very small resistance, but you can buy 3-ohm resistors (orange-black-gold). I suggest you experiment with several low-value resistances, say 10, 8, and 5 ohms, before you choose a value to use in a practical multi-character circuit.

Experiment No. 11 – Drive 7-Segment Display Modules with the MAX7219

Abstract

In Experiments 8, 9, and 10 you learned how a microcontroller can control several types of displays with only a few signals. But that type of repeat-loop control can consume much of the microcontroller's time, which prevents it from doing other things. Just look at the long code listings in the previous experiments. This experiment lets you work with an integrated circuit, the Maxim Integrated Products MAX7219, which can handle all of the display-control requirements for an eight-digit 7-segment display. An MCU communicates with a MAX7219 LED display driver IC via an SPI interface, which makes it popular among engineers as well as among hobbyists and experimenters. The MAX7219 controls *common-cathode* 7-segment displays, and this experiment uses two 4-digit modules to simplify wiring.

You also will learn about circuits that convert logic levels so devices that operate from a 3.3V supply can communicate with those that operate at 5 volts, and *vice versa*.

Keywords

MAX7219, serial, LED, SN74LVC4245, multiplex, transistor array, serial-peripheral interface, SPI, Propeller, object, software, logic-level conversion

Requirements

(1) - MAX7219 8-Digit Display Driver, 24-pin DIP
(1) - 74HC04 inverter, 14-pin DIP (see Logic-Level Conversion section)
(1) - 10k-ohm, 1/4W, 5% resistor (brown-black-orange)
(2) - Lumex LDQ-N516RI, or LiteOn LTC-5723HR, 4-digit common-cathode display module
(1) - 100 µF aluminum electrolytic capacitor, 16V, axial wire leads
(1) - 0.1 µF disc-ceramic capacitor, 50V, wire leads
(1) - 5-volt power supply
(2) - Solderless breadboards
(1) - Propeller P8X32A microcontroller board
(1) - USB cable
(1) - Piece of stiff cardboard, 8.5-by-11 inches (22-by-28 cm) or slightly larger.

Logic-Level Conversion

Before you start to put components in breadboards, you should know about logic levels and logic-level conversion. The Parallax Propeller P8X32A board produces and accepts 3.3-volt logic signals. On the other hand, the MAX7219 LED display-driver IC requires 5-volt logic signals. You cannot simply connect Propeller outputs to MAX7219 inputs. If you do, circuits will not work. You *must* use an IC or module to perform the logic-level conversion. I'll give you information about four approaches; simple to more complicated. You must decide on one and obtain the parts before you get to the first experiment step. I'll recommend a simple way to get started with this experiment.

1. The 74HC and 74HCT logic families can convert the Propeller 3.3V-logic signals into 5V-logic signals. I have used a 74HC04 inverter – six inverter circuits per IC – between a Propeller MCU and a MAX7219 display driver. Unfortunately, the conversion only works one way, so it will not convert 5V-logic signals to 3.3V-logic signals. A 74HC04 or 74HCT04 IC in a 14-pin DIP will work in this experiment.

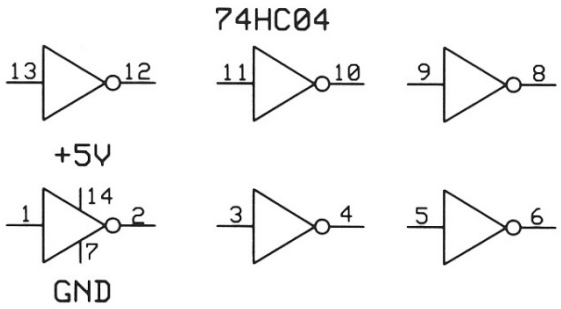


Figure 11.1.
Pin configuration for a 74HC04 inverter IC in a 14-pin DIP. Pin 14 provides 5V power to the six inverter circuits while pin 7 connects the IC to ground.

2. You can purchase a logic-level converter module from Sparkfun Electronics as part no. BOB-12009 ($US 2.95). A module gives you two 3.3V-to-5V and two 5V-to-3.3V logic-level converters (**Figure 11.2**). This experiment would need two of these modules to communicate with a MAX7219. You also must buy some break-away pin headers, too, and solder them to the modules so they plug into your solderless breadboards. Sparkfun sells these pins in strips of 40, part no. PRT-00117 ($US 2.95). For more information, please visit: www.sparkfun.com. I have not used these modules.

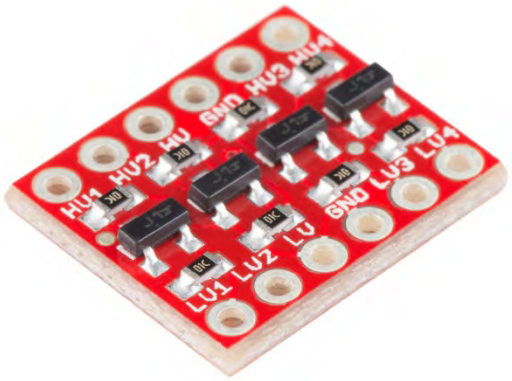


Figure 11.2.
Sparkfun Electronics BOB-12009 board two 3.3-to-5-volt converters and two 5-to-3.3-volt logic-level converters. *Courtesy of Sparkfun Electronics.*

3. Adafruit sells an 8-channel bi-directional logic-level converter module, part no. 395 $US 8.00, shown in **Figure 11.3**. The bidirectional capability comes in handy when you must communicate in both directions between a 3.3V and a 5V logic device. For more information, visit: http://www.adafruit.com/product/395. The module comes with break-away pin headers. The IC used on this module cannot directly drive "heavy" loads such as LEDs. Adafruit recommends against using it for I2C communications, another form of serial signaling, but the company has an I2C-compatible module, too. Adafruit also sells the Texas Instrument SN74LVC245AN IC in a 20-pin DIP, part no. 735 ($US 1.50). Find the data sheet here: http://www.ti.com/lit/ds/symlink/sn74lvc245a.pdf. I have not used this module.

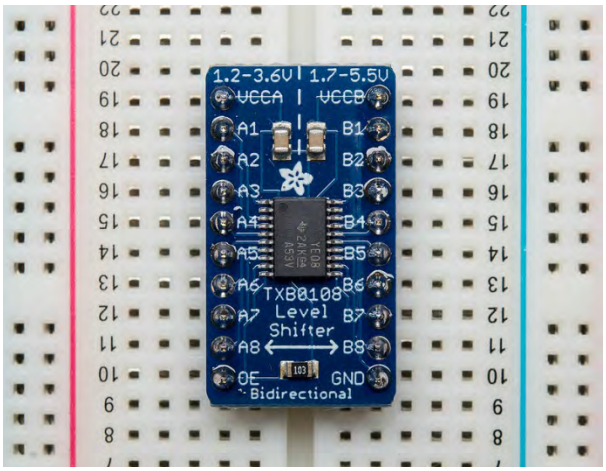


Figure 11.3.
Adafruit 8-channel bi-directional logic-level converter module.
Courtesy of Adafruit.

4. In my lab, I use the SN74LVC4245A "Octal Bus Transceiver and 3.3V to 5V Shifter with 3-State Outputs" IC that contains eight independent logic-level-conversion "channels." You can easily set all eight for 3.3-to-5-volt or 5-to-3.3-volt conversions (**Figure 11.4**). This IC comes in a 24-pin surface-mount package called a small-outline integrated circuit, or SOIC, so it *cannot* plug into a solderless breadboard. An inexpensive adapter solves this problem. SchmartBoard sells an SOIC-to-DIP adapter as part no. 204-0004-01 ($US 6.00), which comes with pins. For more information, please visit: http://www.schmartboard.com/index.asp?page=products_smttodip. The company has a video that demonstrates how to use the SchmartBoard soldering technique. I have used these boards many times without any difficulties. You can purchase the SN74LVC4245 ICs from DigiKey and other distributors for about $US 1.10. Buy several. For technical details, please visit: http://www.ti.com/product/sn74lvc4245a.

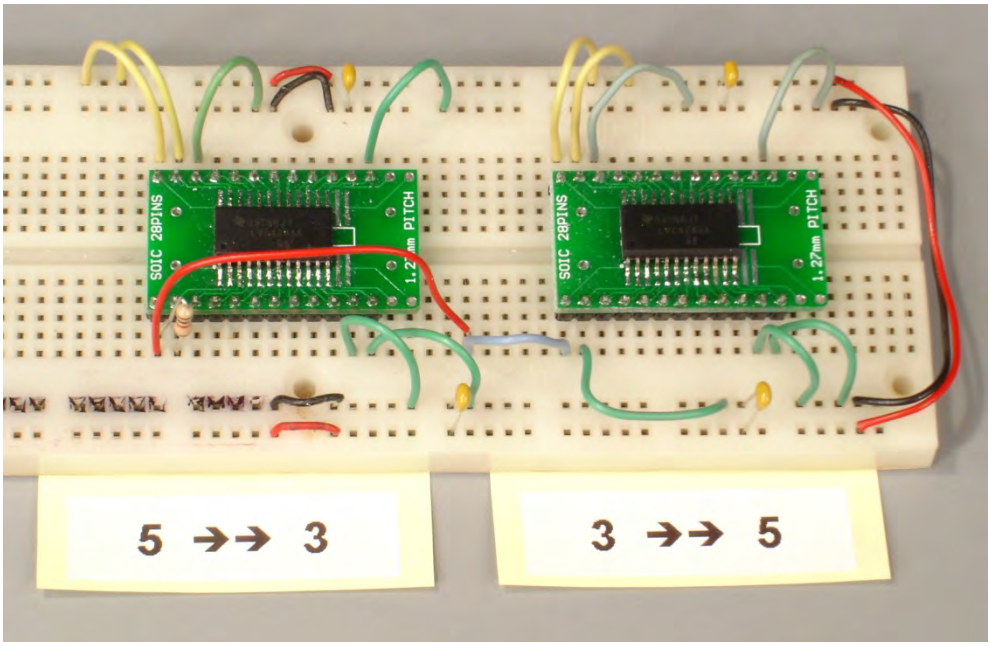


Figure 11.4.
An SN74LVC4245A bus-transceiver performs logic-level conversions for eight signals per IC. I have mounted two of these ICs on Schmartboard surface-mount-IC adapters for 28-pin small-outline ICs (SOICs). The labels indicate the type of conversion I wired for each IC.

The variety of integrated circuits in use today means you will likely run up against logic-level conversion requirements again. Texas Instruments offers a helpful document, "Selecting the Right Level-Translation Solution," for people interested in more technical details (Ref. 1).

Introduction

Displays controlled directly by an MCU require constant "refreshing" or multiplexing to control LEDs so our eyes see them as constantly lit. The almost-constant serial communications with external shift registers or other circuits can take a lot of an MCU's time. So we aim to "off load" the display-control functions to an external device designed specifically for that role. This experiment introduces you to the MAX7219 display-driver IC.

Before you can use a MAX7219 display-driver IC, you must understand how it works. **Figure 11.5** shows a simplified block diagram of a MAX7219 circuit. You will find more details within the Functional Diagram in the complete data sheet for the MAX7219 on the Maxim Integrated Products Web site (Ref. 2).

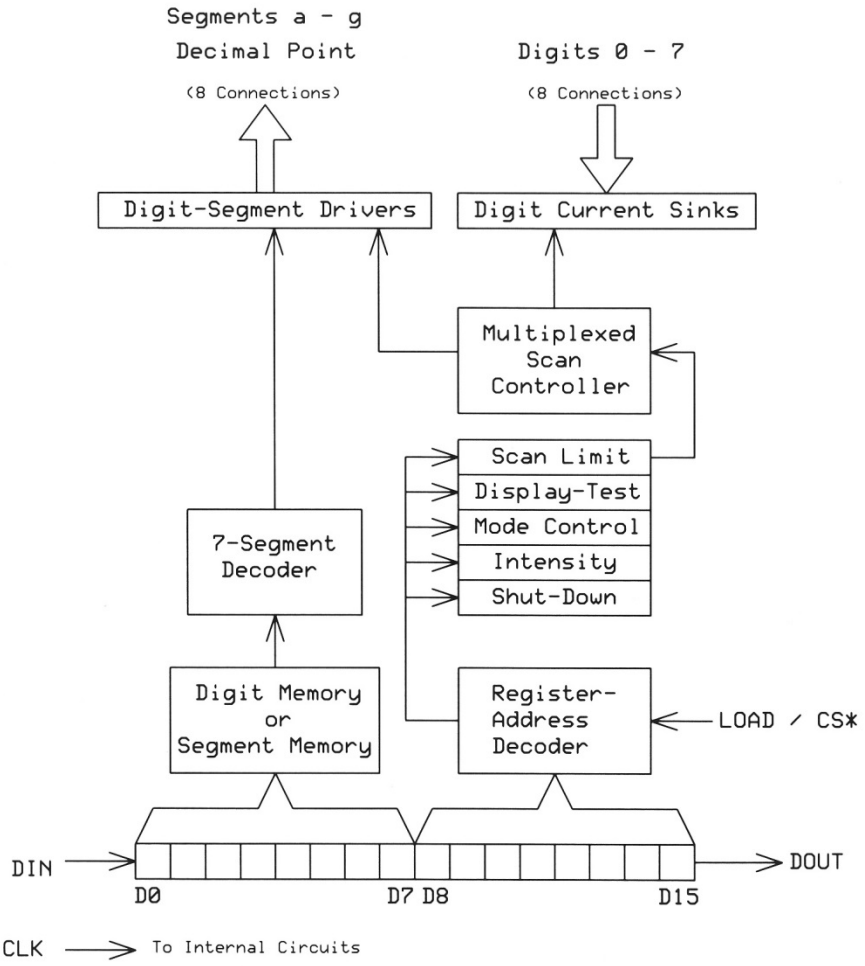


Figure 11.5.
Block diagram for the MAX7219 display-driver IC. This device supplies eight outputs for segments (seven numeric segments and one decimal point) and eight pins that sink current from each digit, one at a time. Internal circuits continue to multiplex the digits or symbols you select without interaction from an MCU.

In Experiment 8 you used two shift-register ICs, and one ULN2003A transistor-switch IC to control four 7-segment displays. The MAX7219 performs the same functions, but with as many as eight digits. This IC also lets you use software to control LED brightness, test display LEDs, set the number of digits to display, blank the segments, and even create your own 7-segment characters. It does all these operations with only three signals from an MCU.

Figure 11.6 shows the pin configuration and corresponding signal names for a MAX7219 in a 24-pin dual inline package (DIP). The IC controls common-cathode displays, either individually, or in modules such as the Lumex LDQ-N516RI. I recommend you use two 4-digit display modules so you can experiment with all eight digits yet keep wiring simple.

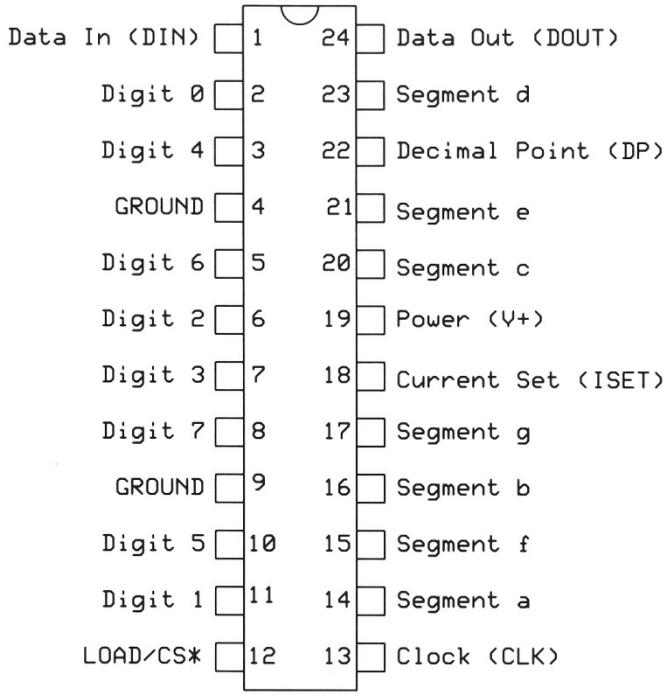


Figure 11.6.
Pinout diagram for a MAX7219 IC. Note the 5-volt power input (V+) at pin 19 and the *two* ground connections (GND) at pins 4 and 9. You must connect both GROUND pins to ground on your breadboard.The MAX7219 IC requires an external resistor connected between the V+ input (pin 19) and the Current Set input (ISET, pin 18). This resistor sets the current level for the displays.

In addition to the Segment outputs and the Digit inputs, the MAX7219 IC has three pins for signals that originate at an MCU: Data In (DIN, pin 1), Clock (CLK, pin 13), and Load/CS* (pin 12). The Propeller P8X32A board, or another MCU board can create these signals, which conform to the Serial Peripheral Interface (SPI) specifications. In previous experiments, you used objects in the SPI_Asm.spin software, and they get used again in the steps that follow.

The 74HC595 shift registers used in previous experiments relied on a 16-bit value transmitted serially from the Propeller P8X32A board. The MAX7219 IC also uses 16 bits of information, but separated into an 8-bit address and eight bits of data, as illustrated in **Figure 11.7**. The address byte identifies a register within the IC that will receive the 8 bits of information that follow. The 13 MAX7219 registers comprise two groups: eight for the displays (one per digit), and five for internal controls.

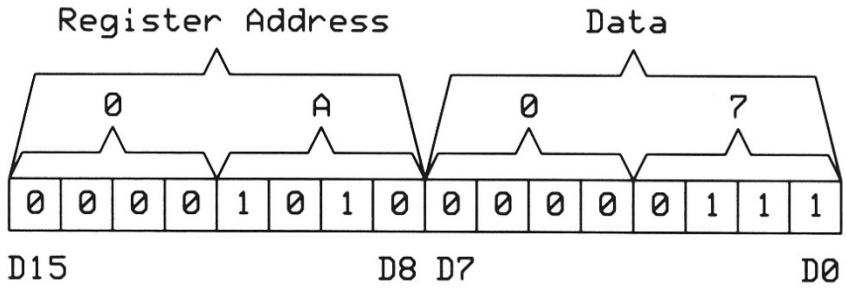


Figure 11.7.
Interpretation of a 16-bit SPI transmission to a MAX7219 IC. This transmission selects the Intensity register ($0A) and sets it for about half brightness ($07).

Table 11.1 lists the 13 registers, their function, their address, and the range of valid data for each. The data sheet for the MAX7219 IC lists a 14th register, labeled No-Op, or no-operation, but it serves no function, so I haven't listed it. The registers act like individual memories and hold information until you change their contents or turn off power. The section that follows **Table 11.1** explains what each register controls. (Spin programs use a dollar sign, $, prefix to identify hexadecimal values. DP indicates a decimal point.)

Table 11.1. MAX7219 Internal Register and Control Information.

Register Name	Register Address	Allowed Values	Action
Digit 0	$01	Any	Segment & DP Control
Digit 1	$02	Any	Segment & DP Control
Digit 2	$03	Any	Segment & DP Control
Digit 3	$04	Any	Segment & DP Control
Digit 4	$05	Any	Segment & DP Control
Digit 5	$06	Any	Segment & DP Control
Digit 6	$07	Any	Segment & DP Control
Digit 7	$08	Any	Segment & DP Control
Decode Mode	$09	$00	No decoding for digits 0-7. Free-form segment use.
Decode Mode	$09	$01	Code B for digit 0, no code for other digits.
Decode Mode	$09	$0F	Code B for digits 0-3, no code for digits 4-7.
Decode Mode	$09	$FF	Code B for all digits.
Intensity	$0A	$00-$0F	Intensity control in 1/32nd steps. Start at 1/32nd.
Scan Limit	$0B	$00-$07	Enables ranges of digits, from digit 0 ($00) to all digits ($0F).

Shutdown	$0C	$00	Shutdown
Shutdown	$0C	$01	Normal Operation
Display Test	$0F	$00	Normal Operation
Display Test	$0F	$01	Display test. All segments and DP forced on.

1. **Decode Mode:** The contents of this register (address $09) lets you choose how to "decode," or display, information sent for each of the eight 7-segment displays. Choices include a "B Code," which uses numerals 0 through 9 as well as the letters “H, E, L, P”, a dash, and a "blank." No-Code, the alternative to B Code, lets you create your own 7-segment characters. Whether you use a B-Code or a No-Code setting, you always can independently control the decimal point for each digit.

   The Decode Mode register can have one of four values. To start, use the B-Code setting, $FF:

   $00 = No Decode for digits 0 through 7, you define symbols and characters..
   $01 = B Code for digit 0, no code for digits 1 through 7.
   $0F = B Code for digits 0 through 3, but No Code for digits 4 through 7.
   $FF = B Code for all digits.

2. **Intensity:** The four least-significant bits (D3 – D0) in this register (address $0A) set the display intensity from minimum ($00) to maximum ($0F) in 16 steps. Each increase or decrease represents a change in brightness of 1/16th of full-on brightness. Thus $07 sets brightness at about half the maximum brightness. For experiments, half brightness will work well.

3. **Scan Limit:** The 3-bit binary value saved in this register (address $0B) determines which digits the MAX7219 IC will use to display information. Allowable values range from $00 for digit-0 alone to $07 for all eight digits. (The MAX2719 data sheet cautions against changing the Scan-Limit value to turn off leading zeros because doing so can increase current excessively through displays. Instead, use the B Code for a blank display ($0F) to turn off a digit. Assume an 8-digit display and digit 0 represents the most-significant digit (10's of millions), and digit 7 the least significant (1's).

4. **Shut-Down:** The single bit in this register (address $0C) lets you turn off the display but maintain the contents of the digit and control registers. You could use the shut-down mode to save power in portable devices, or to display a value or message only when needed. The value $00 puts the display in shut-down mode. A $01 establishes normal operations.

5. **Display-Test:** At times you might want to ensure that all LEDs in a display work properly. This 1-bit register lets you over-ride displayed information and turn on all segments and decimal points. Use $00 for normal display operation or $01 to test all LEDs. In aircraft navigation equipment or medical devices, for example, an operator might want to confirm a displayed 0 really means 0 and not 8 with a burned out LED segment. While in the test mode, the MAX7219 still operates and can accept new register values. But you will not see any changes until you remove the IC from the test mode.

6. **Digit:** The eight digit registers correspond to the eight 7-segment displays the MAX7219 can control. Addresses start at $01 for digit-0 (left-most digit) and increase to $08 for digit-7 (right-most digit). When in B-Code mode, you store the *binary value* for the digit to display, $05 to display the numeral 5, and so on. In the No-Code mode you store the *segment pattern* you want to display. In both cases, the most-significant bit (D7) controls the digit's decimal point. A logic-1 turns a decimal point on, a logic-0 turns it off. More about segment patterns later in this experiment.

With this basic information about registers, their contents, and the operations they control, you can see how a MAX7219 works when controlled by an MCU. As described in the following steps, the IC operates from a 5-volt power source, while the Propeller P8X32A board operates from a 3.3-volt supply. Thus, circuits require conversion of 3.3-volt logic levels to 5-volt logic levels as explained earlier in this experiment.

Step 1.

For this experiment, I suggest you place your breadboards on a piece of cardboard listed in the Requirements section. You will use the display circuit in other experiments, but you might choose to disconnect it temporarily from your P8X32A board. The cardboard "base" makes it easy to move the breadboards and their connections out of the way without disturbing them.

I recommend you use two 4-digit displays to see how a MAX7219 IC operates. I used two Lumex LDQ-N516RI 4-digit common-cathode displays because I had them in my lab. The LiteOn LTC-5723HR displays have the same pin connections and should work just as well, and they cost less than the Lumex displays. If you have only one 4-digit display, you can still run this experiment.

You can place two of the 4-digit displays on one solderless breadboard and still have room for the MAX7219 IC, as shown in **Figure 11.8**. I placed a 74HC04 IC, used for logic-level conversion, on a separate breadboard, as described later. You can place it near the MAX7219 IC, though.

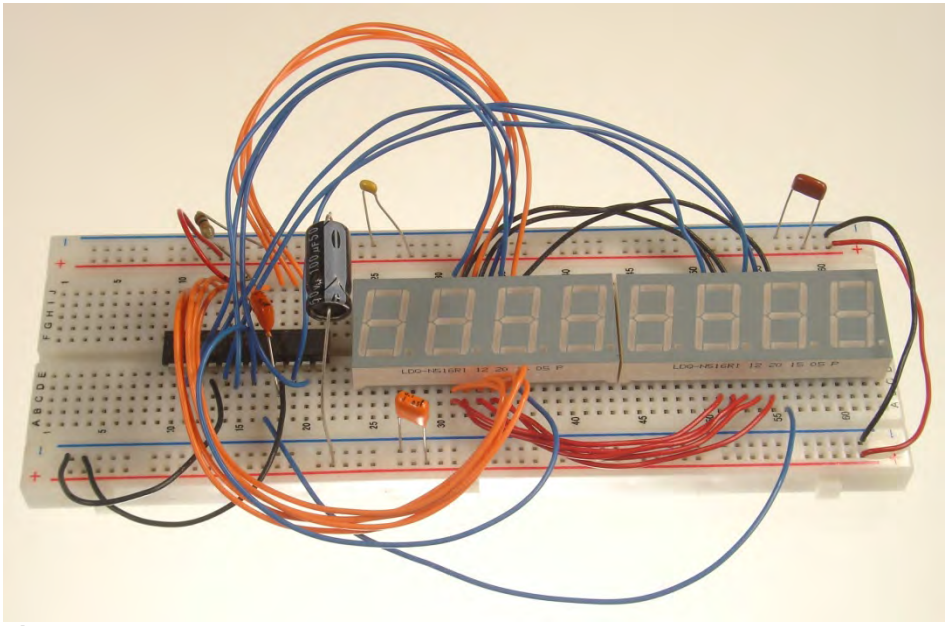


Figure 11.8.
Photograph of the MAX7219 IC and two 4-digit displays wired in a breadboard. Use long wires so you can move them out of the way of the displays.

Step 2.

After you insert the MAX7219 IC and the display module or modules in a breadboard, connect a 100 microfarad (µF) electrolytic capacitor directly between ground (pin 4) and +5V (pin 19) on the MAX7219. Observe the polarity marks on this capacitor and ensure its positive terminal connects to pin 19, and the minus terminal connects to pin 4. Next, connect a 0.1 µF disc-ceramic capacitor in parallel with the electrolytic capacitor. This capacitor reduces momentary voltage changes, or transients, on the power buses caused as the LEDs turn on or off. Transients can have unpredictable effects on circuits, which we don't want.

Connect MAX7219 pin 19 to the nearest +5V power bus on your breadboard. Connect pins 4 and 9 to the nearest ground bus.

Step 3.

Figure 11.9 shows a schematic diagram for the complete display circuit. The Experiment 11 software folder includes a larger version of this figure. I recommend you now add the 10k-ohm resistor (brown-black-orange) between the ISET input (pin 18) and the +5V bus. Cut the leads short so they will not interfere with wire connections to the IC. Next, make the eight parallel connections between the two displays. Then make the connections between the MAX7219 and the displays. Finally, make the connections for the 74HC04 inverter IC and connect the inverter inputs to the Propeller-board expansion connector. Ensure you have a common ground connection between the +5V-supply ground and the ground pin on the expansion-board connector (pin 39).

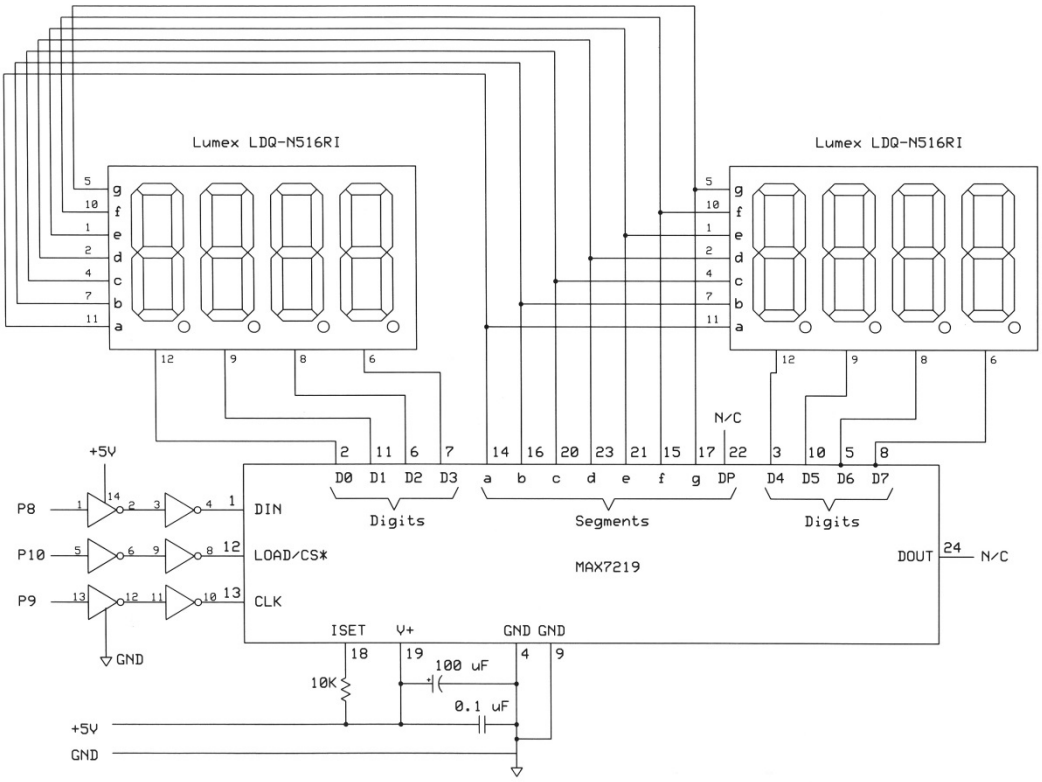


Figure 11.9.

Schematic diagram for the MAX7219 and display circuits. This circuit requires a common ground with the ground pin on the Propeller P8X32A board. Note the absence of current-limiting resistors for the display LEDs. The MAX7219 limits the current internally based on the resistance between +5V and the ISET input.

The 74HC04 inverter IC takes the 3.3V control signals from the Propeller MCU and converts them to 5V logic signals for the MAX7219. The head-to-tail connection for the six inverters simply passes the control signals through to the MAX7219 with the proper voltage and polarity. If the circuit used only one inverter on each of the three signals, logic-0 signals would become logic-1s, and *vice versa*. If you choose an IC other than a 74HC04 to perform a logic-level conversion, ensure you know whether it will or will not invert signals.

Step 4.

Now that you have your circuit wired, you use knowledge of the MAX2719 to set it up for a test. Keep in mind all SPI communications with a MAX7219 require 16 bits; an 8-bit register address followed by an 8-bit value.

Given the set of registers described previously, determine the registers addresses and values needed to initialize the MAX7219 IC so it can display useful information. Given the five registers that do not control specific digits, what values would you use to initialize the MAX7219 IC for the 7-segment B Code, set half-

bright intensity, perform normal operations, and do a display test for all eight digits? Find the registers and addresses below. You supply the data each needs. (The MAX7219 IC does not have registers D or E.)

```
Decode Mode:          09__________
Intensity:            0A__________
Scan Limit            0B__________
Shutdown:             0C__________
Display Test:         0F__________
```

After you create the five hex values, open **Program 11.1** and place your values in the `MAX7219_init[x]` array. Remember: In the Spin language a hexadecimal value has a dollar-sign prefix; $5FF2, for example. Now run the program. What do you see on the display? If you have problems getting the proper hexadecimal values, you will find them in the Answers section at the end of this program.

Program 11.1.

```
{{
''*********************************************
''*  Program 11.1
''*  Author: Jon Titus 11-05-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display-test demonstration for an 8-digit
''*  7-segment display (Lumex LDQ-N516RI)
''*  by a Maxim MAX7219 display controller IC.
''*  This test relies on 3.3-V-to-5-V logic-level
''*  conversion via a 74HC04 inverter IC.
''*  SPI_Asm object library used for SPI-type
''*  communications with the MAX7219 IC.
''*  Set for all-segments-on test.
''*********************************************
}}
'Set MCU clock operation for 5 MHz crystal
CON _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000

'Invoke the SPI_Asm.spin file in your Propeller Workspace
OBJ
  SPI     :       "SPI_Asm"
                                        '

VAR
'Storage for initialization data in an array
    word MAX7219_init[10]

    word display_loop         'Counter used for display loops
    byte data_port            'Data-output pin to shift register
    byte serial_clock         'Pulse for shift register clock
    byte chip_select          'Pulse for LOAD pin
    word data16_out           '16-bit value for SPI transmission

'Main program starts here
PUB Start

''SPI Setup
 SPI.start(15,0)

'Output pins for SPI and LOAD signals
```

```
dira[8..10] := %111
outa[8..10] := %000 'Set outputs to 0

data_port := 8                'Use P8 for SPI data
serial_clock := 9             'Use P9 for SPI clock
chip_select := 10             'Use P10 for MAX7219 LOAD signal

'Insert your hex values in the spaces provided
MAX7219_init[0] :=  $09__     'Decode Mode for "B Code"
MAX7219_init[1] :=  $0A__     'Intensity for half-bright LEDs
MAX7219_init[2] :=  $0B__     'Scan limit: Use all 8 digits
MAX7219_init[3] :=  $0C__     'Normal Operation = $0C01
MAX7219_init[4] :=  $0F__     'Test? $0F00 = no; $0F01 = yes.

'Initialize with MAX7219_init values
outa[chip_select] := 0       'Reset LOAD signal for MAX7219

'Set loop for five parameters and send them to MAX7219
repeat display_loop from 0 to 4
      data16_out := (MAX7219_init[display_loop])
      MAX7219_SPI_out
      'See object definition below

'SPI_output object defined here
PUB  MAX7219_SPI_out
     SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16, data16_out)

     'Delay, wait for SPI transmission to end
     waitcnt(clkfreq/100000 + cnt)

      'strobe MAX7219 LOAD pin after transmission
      outa[chip_select] := 1          'Set LOAD pin to logic-1
     outa[chip_select] := 0          'Reset LOAD pin to logic-0

 ' –  – -End –  – -
```

You should see all segments and decimal points turn on. Change the Display Test register value to $00 and run the program again to switch to normal operation. Now what do you see? It depends... You might see a blank display or some numbers, depending on the values already in the digit memory within the MAX7219 IC.

Step 5.

The values for the digit displays, 0 through 7, get saved in registers with an address one greater than the digit position. Thus register 01 controls digit 0, register 02 controls digit 1, and so on to 08 for digit 7. Create a short `repeat` loop that puts each digit's position value in the corresponding register location so digit-0 displays a 0, digit-1 displays a 1, and so on. The display should show 01234567 after you run the modified program. Use the `MAX7219_SPI_out` object to send the 16-bit register-and-data values to the IC. If you get stuck, **Program 11.2** shows how I added a loop, shown in boldface type, that accomplished the task.

Program 11.2

```
{{
''*******************************************
''*  Program 11.2
''*  Author: Jon Titus 11-05-2014 Rev. 2
''*  Copyright 2013
''*  Released under Apache 2 license
```

```
''*  Display-test demonstration for an 8-digit
''*  7-segment display (Lumex LDQ-N516RI)
''*  by a Maxim MAX7219 display controller IC.
''*  This test relies on 3.3-V-to-5-V logic-level
''*  conversion via a 74HC04 inverter IC.
''*  SPI_Asm object library used for SPI-type
''*  communications with the MAX7219 IC.
''*  Display values 7 through 0.
''*******************************************
}}
'Set MCU clock operation for 5 MHz crystal
CON _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000

'Invoke the SPI_Asm.spin file in your Propeller Workspace
OBJ
  SPI     :       "SPI_Asm"
                                        '

VAR
'Storage for initialization data in an array
    word MAX7219_init[10]

    word display_loop         'Counter used for display loops
    byte data_port                 'Data-output pin to shift register
    byte serial_clock         'Pulse for shift register clock
    byte chip_select          'Pulse for LOAD pin
    word data16_out           '16-bit value for SPI transmission

'Main program starts here
PUB Start

''SPI Setup
 SPI.start(15,0)

'Output pins for SPI and LOAD signals
dira[8..10] := %111
outa[8..10] := %000 'Set outputs to 0

data_port := 8                'Use P8 for SPI data
serial_clock := 9             'Use P9 for SPI clock
chip_select := 10             'Use P10 for MAX7219 LOAD signal

'Insert your hex values in the spaces provided
MAX7219_init[0] :=  $09__     'Decode Mode for "B Code"
MAX7219_init[1] :=  $0A__     'Intensity for half-bright LEDs
MAX7219_init[2] :=  $0B__     'Scan limit: Use all 8 digits
MAX7219_init[3] :=  $0C__     'Normal Operation = $0C01
MAX7219_init[4] :=  $0F__     'Test? $0F00 = no; $0F01 = yes.

'Initialize with MAX7219_init values
outa[chip_select] := 0       'Reset LOAD signal for MAX7219

'Set loop for five parameters and send them to MAX7219
repeat display_loop from 0 to 4
      data16_out := (MAX7219_init[display_loop])
      MAX7219_SPI_out
      'See object definition below
```

```
repeat display_loop from 0 to 7
   data16_out := ((display_loop + 1) <<8) + display_loop
   MAX7219_SPI_out

'SPI_output object defined here
PUB  MAX7219_SPI_out
     SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16, data16_out)

     'Delay, wait for SPI transmission to end
     waitcnt(clkfreq/100000 + cnt)

      'strobe MAX7219 LOAD pin after transmission
      outa[chip_select] := 1          'Set LOAD pin to logic-1
     outa[chip_select] := 0          'Reset LOAD pin to logic-0

 ' –  – -End –  – -
```

The added loop runs eight times, 0 through 7, and the statement:

```
data16_out := ((display_loop + 1) << 8) + display_loop
```

takes the `display_loop` value and adds one to it to create the register address for that digit. For the left-most digit, digit 0, controlled by register 1, `display_loop + 1` equals 1. This value gets shifted eight bit positions to the left to create the register address for the digit. Then the `display_loop` value gets added to the shifted result to create values such as $0100 to display a 0 for digit 0, and so on.

Program 11.2 works but it contains a "problem" that could cause future problems for programmers. Can you find it? If you give up, look in the Answers section at the end of this experiment.

Can you change the program so the display reverses the digits to show 76543210? Also, how would you "clear" the display so all LEDs turn off? Find the answers at the end of this experiment.

The Code B setting lets the MAX7219 display the letters “H”, “E”, “L”, and “P”. Write another loop to display “HELP” one letter at a time. Use these hex values for the letters:

```
H = $0C
E = $0B
L = $0D
P = $0E
```

After you get the letters to turn on one at a time, how would you flash "HELP" on and off? Find the answer in the Spin program **Program 11HELP** and at the end of this experiment. The Experiment 11 software file includes the complete program you can run.

Step 6.

So far, this experiment has used the "B Code" setting, but the purpose of the Decode-Mode register (address $09) requires more explanation. With this register set to $00, you can create your own patterns for the 7-segment displays, and you can control the decimal points. Instead of sending the MAX7219 the *value* you want it to display at a given position, you send it a segment *pattern*.

The standard segmented numeral pattern and MAX7219 bit arrangement appear in **Figure 11.10**, and **Table 11.2** shows the format of segment bits you use to create any 7-segment symbol. A logic-1 for a segment turns it on. A logic-0 turns it off. To create an upper-case J, for example, I must turn on segments b, c, d, and e. All

other segments – and the decimal point – remain off. The segment pattern equals 00111100_2, or $3C. To display the letter J at position 2, I must send the pattern to register $03. Combining the address and register contents gives me: $033C.

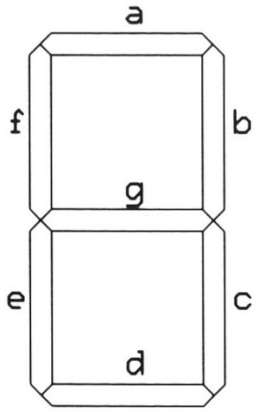


Figure 11.10.
The standard layout for LEDs in a 7-segment numeric display.

Table 11.2. Register data needed to control individual segments with a MAX7219 IC. Codes go in registers 0 through 7 to create characters and symbols.

	Segment Data for Registers 0 through 7							
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Segment	DP	a	b	c	d	e	f	g

Previous programs placed a "blank" character ($0F) in each digit's register to clear the eight digits. But that situation applies only when we used the Code-B format. Now we must place the *segment pattern* $00 in a digit register to turn off the corresponding segments and decimal point. (If you try to clear a digit register with $0F, or 00001111_2, you will not see a blank digit position. Instead, you will see the display turn on *segments* for the 1's in the 00001111_2 pattern. It took me a few minutes to realize I could no longer use a "blank" value to clear the registers for the eight digits.

Step 7.

The Experiment 11 folder includes **Program 11Jon.spin** that demonstrates how to use the "No Code" mode and control individual LED segments. This program displays a short message from me to you.

What's next? Please keep your display circuit wired for use in Experiment 13 as a temperature display. If you placed your breadboards on a piece of cardboard, you can disconnect them from the Propeller P8X32A board and set them aside. Or carefully slide the cardboard under the breadboards now, disconnect them from the MCU board, and set them aside. Experiment 12 lets you investigate how the Propeller MCU can make measurements in the "real world" and in Experiment 13 you learn how to display them with a MAX7219.

Reference

1.Dhond, Prasad, "Selecting the Right Level-Translation Solution," Application Report SCEA035A, Texas Instruments. June 2004. http://www.ti.com/lit/an/scea035a/scea035a.pdf. Helpful information about logic families and the types of signals they use as inputs and outputs.

2. Data sheet for the Maxim Integrated "Serially Interfaced, 8-Digit, LED Display Drivers," http://www.maximintegrated.com/datasheet/index.mvp/id/1339.

Answers

Experiment 11, Step 4:

Given the five registers that do not control specific digits, what values would you use to initialize the MAX7219 IC for the 7-segment B Code, about half-bright intensity, normal operations, and display test for all eight digits?

```
Decode Mode:            $09FF
Intensity:              $0A07
Scan Limit              $0B07
Shutdown:               $0C01
Display Test:           $0F01
```

Experiment 11, Step 5:

To display 76543210, change the loop statement:

```
data16_out := ((display_loop + 1) << 8) + display_loop
```

to

```
data16_out := ((8 - display_loop) << 8) + display_loop
```

To blank the display, use a loop that sends the "blank" code, $0F to each display register:

```
data16_out := ((display_loop + 1) << 8) + $0F
```

Program 11.2 has a "problem" in the use of the variable `display_loop` because I have used the same variable in two separate loops. In the code below I knew I could use `display_loop` as the variable that counts the initialization bytes and use it again in the loop that displays digits:

```
repeat display_loop from 0 to 4
      data16_out := (MAX7219_init[display_loop])
      MAX7219_SPI_out
      'See object definition below

repeat display_loop from 0 to 7
   data16_out := ((display_loop + 1) <<8) + display_loop
   MAX7219_SPI_out
```

But programmers should not reuse variables for different purposes! To solve the problem, I should have defined and used a different and descriptive variable for each loop:

```
repeat init_display_loop from 0 to 4
      data16_out := (MAX7219_init[display_loop])
      MAX7219_SPI_out
      'See object definition below

repeat data_display_loop from 0 to 7
   data16_out := ((display_loop + 1) <<8) + display_loop
   MAX7219_SPI_out
```

Program 11.HELP

In this program I defined a 4-element array, `byte HELP[4]` and loaded the array with the *codes* for the letters “H”, “E”, “L”, and “P” in that order. The "Code B" setting gives you these four letters. You do not have to create these letters from individual segments.

Separate loops initialize the MAC7219, clear the display, display HELP one letter at a time, and then switch back and forth between the Normal Operation and Shutdown modes. Note that the Shutdown mode does not alter any register contents, it simply turns off the LEDs and goes into a low-power standby mode. Find the complete **Program 11HELP.spin** in the Experiment 11 folder.

Experiment No. 12 – Have an MCU Take Real-World Temperature Measurements

Abstract

This experiment explains and demonstrates how a Propeller P8X32A microcontroller (MCU) can obtain temperature information from a sensor; the Maxim Integrated Products DS1620 IC. The Digital Thermometer and Thermostat IC uses serial communications, much like those used in previous experiments. Back and forth communications between the Propeller MCU and the sensor establish the DS1620's operating conditions. In this experiment you also will learn how to handle negative binary values that the DS1620 reports for temperatures below 0 °C (32 °F).

Keywords

DS1620, serial communications, temperature measurements, temperature sensor, serial-peripheral interface, SPI, Propeller, object, software, one's complement, two's complement, negative numbers

Requirements

(1) - DS1620 Digital Thermometer and Thermostat integrated circuit, 8-pin DIP
(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - Can, freeze spray (MCM Electronics, part no: 20-3000), or an ice cube (see text)
(1) - USB cable
(1) - Small piece of flat metal (see text)
Heat-sink compound (optional)

Introduction

Microcontrollers can perform many tasks that involve measuring real-world characteristics such as temperature, pressure, light, voltage, acidity, fluid levels, and so on. In most situations engineers can find sensors designed specifically to measure one of these characteristics. The Maxim Integrated DS1620, for example, measures temperatures between -55 and 125 degrees Celsius (-67 to 257 degrees Fahrenheit). This 8-pin sensor IC uses an internal semiconductor to measure a temperature that gets reported to an MCU as a 9-bit binary value. The MCU transmits information to the DS1620 to control its operations.

The MAX7219 display controller used in Experiment 11 includes several registers that control LED brightness, individual digits, and so on. Likewise, the DS1620 contains registers that control internal operations and hold temperature information an MCU can read via serial communications. (The DS1620 can operate as a thermostat with a high- and a low-temperature limit, and high- and low-temperature alarm outputs. This experiment uses only the simplest temperature-measurement capabilities of a DS1620.)

Communications between a DS1620 and an MCU require three signal lines, along with a 3.3-volt power input and a common ground. The IC operates with a supply voltage between 2.7 and 5.5 volts, so you can obtain power directly from a P8X32A MCU board.

The three signal lines on a DS1620 perform the following operations:

- DQ (pin 1): Input and output of serial data; bidirectional signals.
- CLK (pin2): Clock signal for serial communications.

- RST* (pin 3): A reset input on the DS1620 IC (logic-0 causes the reset).

The DQ signal can transmit or receive information, depending on the type of communication requested. The MCU will transmit data to configure the DS1620 and start its temperature measurements. Immediately afterward, the DS1620 "reverses" the use of the DQ pin and transmits temperature information to the MCU. The DS1620 can provide a new temperature reading every 750 milliseconds (750 msec), or every three-quarters of a second. Most temperatures do not change rapidly, so for practical purposes, that measurement rate will suffice. **Figure 12.1** shows the pin numbers and signal names for a DS1620 8-pin DIP IC.

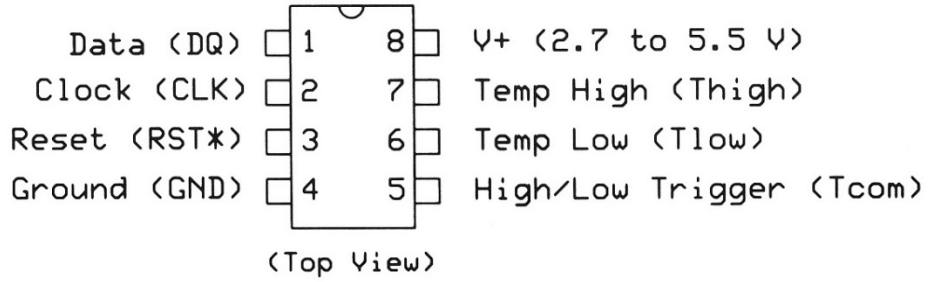


Figure 12.1. Top view of a DS1620 temperature sensor. Temperature measurements in this experiment do not use pins 5, 6, or 7. These outputs can indicate a temperature has exceeded a high or low limit you set.

In this experiment you will not get involved with the serial-communication formats because I modified and simplified several objects that handle those details. If you want to look into the details, find the modified program in ds1620_JT2.spin. The programs now include better variable names and comments so you can understand what the statements do. A tip of the hat for John Williams at Parallax. John created the original DS1620 software I used to get started. No sense starting from scratch. This experiment also uses the Parallax Serial Terminal (PST) program so you can see temperature values in the PST window on your PC.

You can read the DS1620 data sheet to learn more about how the IC operates and the other types of information you can send to it and receive from it. For a link to the online data sheet, see the References section at the end of this experiment. Experiment 13 provides more information about communications between a DS1620 sensor IC and the Propeller MCU.

Step 1.

If you still have your display circuit set up from Experiment 11, you can disconnect it from the P8X32A board and temporarily set it aside for use in Experiment 13. You may choose to keep the display circuit connected to the MCU board and use a separate breadboard for the DS1620. I find a less-cluttered lab bench makes it easier to run experiments and test circuits.

Carefully plug a DS1620 IC into a solderless breadboard and make the connections shown in **Figure 12.2**. Pins 5, 6, and 7 on the DS1620 should remain unconnected.

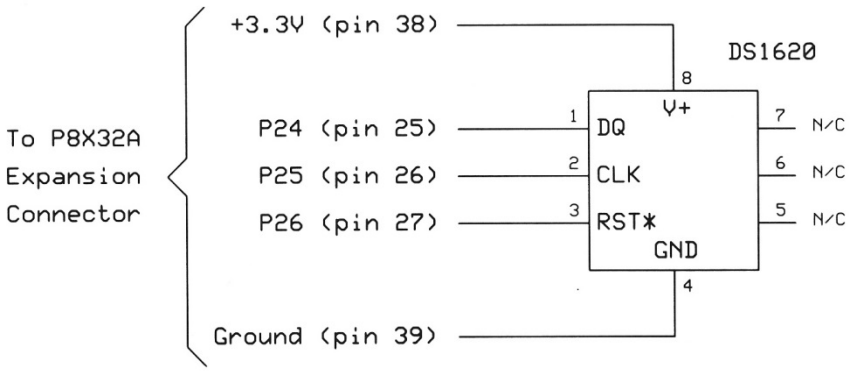


Figure 12.2.
Schematic diagram for the DS1620-to-P8X32A MCU board connections that will let software measure temperature.

Step 2.

Connect your P8X32A MCU board to a PC and if not already running, start the Parallax Propeller Tool software. Ensure you have no program open (look for any tabs at the top of the programming-and-editing window). If you have Propeller programs open, please close them.

Use the upper folder window on the left side of the Propeller Tool to locate the Experiment 12 software folder. Then open this folder. Ensure you have the following six programs available in that folder:

- Program 12.1.spin
- Program 12.2.spin
- DS1620_JT2.spin
- Timing.spin
- FullDuplexSerial.spin
- ShiftIO.spin

From within the Propeller Tool, open **Program 12.1**. Run this program. You must have the programs listed above in your Propeller Workspace folder.

Program 12.1

```
{{
''*********************************************
''*  Program 12.1.spin
''*  Author: Jon Titus 11-11-2014 Rev. 9
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Program used to obtain a 9-bit value
''*  from a DS1620 Digital Thermometer and Thermostat.
''*  This program does not use the thermostat functions.
''*  Display temp on PST as a raw 3-digit hex value.
''*  Based on program DS1620_Demo, written by John
''*  Williams, Parallax, 28 March 2006.
''*  Uses P8X32A Propeller-I board.
''*********************************************
}}

CON

  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal
```

```
VAR
  ''Create variables for DS1620 control pins and serial output
  ''for the Parallax Serial Terminal (PST) for testing DS1620
  ''operation.

  byte PropPin_DQ               'Variable for DS1620 DQ, pin 1
  byte PropPin_CLK              'Variable for DS1620 CLK, pin 2
  byte PropPin_RST              'Variable for DS1620 RST*, pin 3
  byte PropPin_SerOut           'Variable for P8X32A pin P30
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for serial comm format
  word SerPort_BaudRate         'Variable for P8X32A serial comm
                                'bit rate

OBJ
  temp  :  "ds1620_JT2"         'ds1620_JT2,spin file
  delay :  "timing"             'timing.spin file
  pst   :  "FullDuplexSerial"   'Serial library for testing

'Main program starts here. The tc defines a local variable
'available for use only in the "main" object.
PUB main | tc

 PropPin_DQ := 24             'P24 for DS1620 DQ pin1
  PropPin_CLK := 25                 'P25 for DS1620 CLK pin 2
  PropPin_RST := 26           'P26 for DS1620 RST pin 3
  PropPin_SerOut := 30              'P30 for USB to host PC
  PropPin_SerIn := 31               'P31 for USB from host PC
  PropPin_SerMode := 0              'Invert RX mode
  SerPort_BaudRate  := 9600         'Baud rate at 9600 per sec


'Configure and start DS1620 temperature conversions
  temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Configure Propeller serial port to communicate with host PC
'via Parallax Serial Terminal (PST)
  pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

'Loop to get temp & display it as hex value every 3 sec
    repeat
      delay.pause1ms(3000)              '3 second delay
      tc := temp.gettempc               'get temp via
                                        'DS1620_JT2.spin
      pst.hex(tc, 3)                    'display temp (tc) as
                                        'hex value on PST
      pst.str (String("  Raw HEX..."))  'HEX reminder text
      pst.Tx(13)                        'move cursor to start a
                                        'new line
                                        'run loop again...
' –  – -End –  – -
```

Step 3.

Program 12.1 offers a simple test of a DS1620 IC. The software will display hexadecimal temperature values as received from the sensor. If you do not have the Parallax Serial Terminal (PST) program already running,

press F12 to start it. Look in the bottom-left corner of the PST window and check that you have the proper COM-port assigned for the P8X32A board and that you have the correct Baud Rate value. I use 9600 baud as a default. If you have doubts about the proper COM port, go back to the Propeller Tool, click on Run, and then on Identify Hardware... You will see a small window open with the COM-port designation for your attached P8X32A board. My PC displayed the message shown in **Figure 12.3**. You will probably see a different COM-port number. Write your COM-port number here in case you need it again:

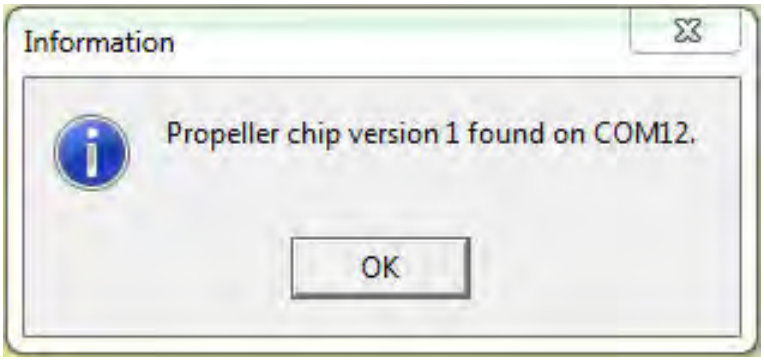


Figure 12.3.
Response displayed by the Propeller Tool for a P8X32A board attached to my host PC through a USB cable.

Use the COMxx setting shown in this window on your PC to choose the "Com Port" in the PST window. If not already set, choose 9600 for the Baud Rate.

Step 4.

Switch back to the Propeller Tool window and press the F10 key on your PC keyboard to load **Program 12.1** into your Propeller MCU. When the program runs, switch to the PST window. Click on the Clear control and then click on the flashing Enable control in the lower-right corner of the PST window.

The software in **Program 12.1** will get a temperature value from the DS1620 sensor every three seconds, and **Figure 12.4** shows what I saw on my PC.

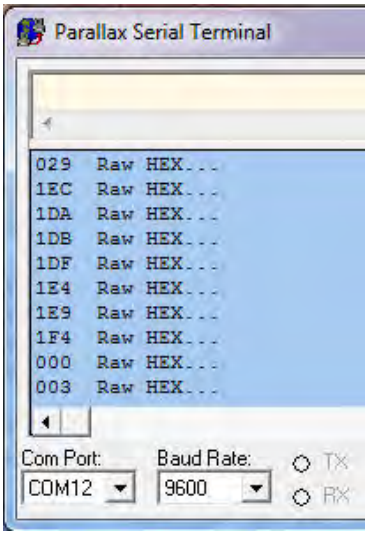


Figure 12.4.
Hexadecimal values obtained at 3-second intervals from a DS1620 sensor, as seen on my PC. I included the "Raw HEX..." text to remind me about the type of temperature values displayed.

If you do not see values appear in the PST window, watch the red LED next to the USB connector on the P8X32A board. It will flash each time the MCU transmits information to your PC via the USB cable. This flashing LED indicates whether or not **Program 12.1** continues to run in the Propeller MCU and transmit temperature information. (LEDs provide excellent tools when you test software.)

If **Program 12.1** runs properly – as shown by the flashing red LED – and you still cannot see information in the PST window, check that you have the PST properly configured with the correct "Com Port" and Baud Rate values. Within the PST window you should see the blue RX indicator briefly flash white for each temperature "message" received. If do not see this indicator flash every three seconds, the PST has not received any data. Recheck your PST settings.

Step 5.

What happens when you use a finger or a hair dryer (briefly, and at a low setting!) to warm the DS1620 IC? Do the raw hex values increase? Write down some of your values. You can click the Pause control in the PST window to suspend operation so you can scroll back through values received earlier.

Write your temperature values here:

You can cool the DS1620 sensor in two ways so you can see what happens to the reported hexadecimal values.

Use either use a can of spray coolant (**a**) or an ice cube (**b**):

a. **Spray coolant**. Place a small piece of metal on top of the DS1620. A small amount of heat-sink compound grease between the metal and the top of the DS1620 will fill imperfections between the two surfaces and help hold the metal in place. The metal shields the breadboard contacts from the cold spray that can cause water to condense on and short circuit the DS1620 pins and breadboard contacts. When I "shot" the spray coolant directly on a DS1620 IC, I saw errors in temperature readings. I used a piece of 0.6-mm-thick brass that measured about 1.5-by-3 cm. A metal coin will work, too. Just ensure a good contact between the metal and the top of the DS1620 IC.

b. **Ice cube**. Place an ice cube in two zipper-type plastic bags. You do not need a metal hat when you use ice because it will only affect the temperature of the DS1620 IC. You won't see condensation. An ice cube from a home refrigerator will reduce the DS1620 temperature only to about -5 °C. If you want to get to lower temperatures, a nearby grocery store might sell dry ice (-78 °C). (Handle dry ice only with gloves and observe other safety measures described at: http://www.dryiceinfo.com/safe.htm.)

As the DS1620 cools, does the PST show changing values? Give the metal a longer burst of freeze spray or longer contact with the ice, and see how the temperature values change. As the IC gets colder, do the hex values get smaller, too? Record some of your hex values below as you cool the IC:

Step 6.

When I ran this experiment, the hexadecimal values for temperatures started at 02A, ambient room temperature here, and increased to 035 with a finger on the IC. A hair dryer increased the temperature to 04E, at which point I turned off the heat.

After the DS1620 sensor cooled to about room temperature from my hair-dryer burst of hot air, I sprayed its metal "hat" with coolant spray. The hex values decreased to 005 as shown in **Figure 12.5**. Then the temperature suddenly "jumped" to 1F6, even though the spray continued to cool the IC. I saw a similar change in temperature values when I cooled the DS1620 with ice. The abrupt change in hexadecimal temperature values results from the way a DS1620 IC reports temperatures.

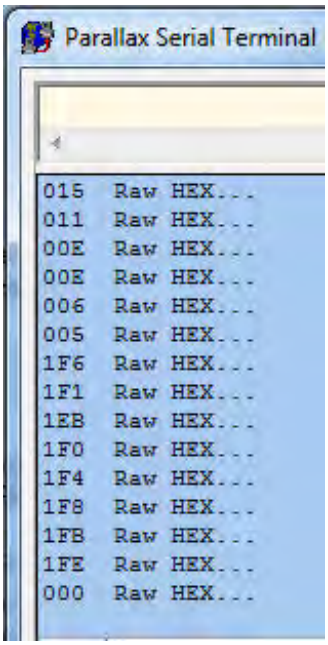


Figure 12.5.
Data from an experiment cooling the DS1620 temperature sensor. The temperature data decreases from 015 to 005 and then increases to 1F6.

Step 7.

The information in **Table 12.1** lists seven temperatures and the corresponding 9-bit binary value from a DS1620 sensor. The table also includes the hexadecimal and decimal equivalent for each binary value. Temperature values from a DS1620 IC change in half-degree steps, so the IC will produce 360 unique values between -55.0 °C and 125.0 °C in half-degree steps: ; 10.0, 10.5, 11.0, 11.5, 12.0 °C, and so on.

Table 12.1. Seven Representative Temperatures and Equivalent Data.

Temp (°C)	Binary Data	Hexadecimal Data	Decimal Data
+125	011111010	00FA	250
+25	000110010	0032	50
+0.5	000000001	0001	1
0.0	000000000	000	0
-0.5	111111111	01FF	511
-25.0	111001110	01CE	462
-55.0	110010010	0192	402

The DS1620 IC does not start with the value of 00000000_2 at its lowest temperature, -55 °C, and increase this value in half-degree steps up to 125 °C. Instead, the sensor reports *2's-complement* temperature values that allow for positive and negative binary numbers that correspond to positive and negative temperatures.

How Do We Handle 2's Complement Numbers?

In the 2's-complement numbering system, positive values range from 0 to larger values as shown with binary numbers 0000, 0001, 0010, 0011, and so on for 0, 1, 2, and 3... As expected, the binary values increase.

But negative numbers such as -1, -2, -3, and -4 get handled in a different way. They *decrease* from 0 (0000_2), as in binary numbers 1111, 1110, 1101, 1100, and so on for -1, -2, -3, and -4. Those binary numbers look like the decimal equivalents, 15, 14, 13, and 12. So how do we interpret negative binary values? In a 2's-complement value, the most-significant bit (MSB) represents the sign (+ or -) of the value; 0 for positive binary numbers and 1 for negative binary numbers. The binary numbers 0011101 and 01010000100 both have an MSB equal to zero, so they represent positive 2's complement values. Those binary numbers have the decimal equivalents 29 and 644 respectively.

On the other hand, binary numbers 100010 and 11100000101 both have an MSB equal to one, so they represent negative 2's complement values. Those binary numbers have the decimal equivalents -14 and -254 respectively. But you cannot "convert" a positive value to an equivalent negative value simply by changing the MSB from a 0 to a 1. Thus the 4-bit number 0111_2 represents +7, but 1111_2, *does not* represent -7.

When you use 2's-complement numbers in a byte of data, positive values range from 0 to +127 (01111111_2), and negative values can go down to -128 (10000000_2). Zero is just 0; we cannot have a -0.

IMPORTANT: According to the Propeller Manual, (ver. 1.2), "Since the Spin language performs all mathematic operations using 32-bit signed math, any byte-sized values will be internally treated as positive long-sized values." That means the Propeller treats all long (16-bit) and byte (8-bit) values as positive. If we want to use the MSB of a value to indicate the sign of a value, it's up to us to write programs that interpret it that way. A computer does not "know" the difference between regular and 2's-complement binary values, so it treats all values as binary unless "told" otherwise within a program.

The Propeller Spin language treats all 32-bit values as signed. In the C language programmers use the data type `char` to define a byte. They can narrow the definition – `unsigned char` or `signed char` – to identify how they want to use the byte

How to Interpret 2's Complement Values

How can we "create" a negative binary value? The following step-by-step example in **Table 12.2**, illustrates the process. In Step 1 you have the binary value for +13 that you want to convert to a 2's-complement negative number. In Step 2 you create a 1's complement value, which simply inverts the state of each bit. A 0 in the starting value becomes a 1, and a 1 in the starting value becomes a 0, bit by bit. Step 3 introduces the value 1 and by adding it to the value shown in Step 2, you get the 2's-complement value for -13, shown in Step 4: 11110011_2.

Table 12.2. Steps Used to Create Negative Binary Values in 2's-Complement Format.

	MSB							LSB	
Step Number	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**Data Actions**
1	0	0	0	0	1	1	0	1	Binary value for decimal 13.
2	1	1	1	1	0	0	1	0	Invert each bit to create a 1's complement binary value.
3	0	0	0	0	0	0	0	1	Add 1 to one's complement value of 13 in the row above.
4	1	1	1	1	0	0	1	1	Sum of 1 plus 1's complement of binary 13, which equals -13.

To generalize, invert the bits in the number you need to convert to a negative value. Add 1 to get the 2's complement negative value. In the Spin language, the NOT operator (!) performs the bit-by-bit inversion for a value given here (86):

`Test := !%01010110` yields the value `%10101001`

Addition of 1 creates the 2's complement value; the negative of the original value:

`Test := !%01010110 + %1` so `Test = %10101010` (-86)

How can we prove 10101010_2 actually equals -86_{10}? If you add $+86_{10}$ and -86_{10}, the *sum* should total 0, and the same must hold true for binary numbers, so:

```
   86  =  01010110
  -86  = +10101010
result =     ?
```

What result did you get?

The answer comes to 00000000_2. The 8-bit addition produces a carry from the most-significant-bit column, but because we have added two 8-bit values, we get an 8-bit result. The carry gets ignored. If you haven't done binary math, see the rules for addition in the Notes section at the end of this experiment.

Follow the steps shown in **Table 12.2** and create *negative* binary values in a single byte for decimal numbers 37, 23, and 94. Find the results in the Answers section at the end of this experiment.

Step 8.

Now we need a way to convert the negative 2's-complement values for temperatures below 0°C into values people can understand. **Table 12.3** illustrates how to take the 2's-complement value for -13 and convert it into a positive value "flagged" with a minus sign. A "flag" uses a byte or bit to indicate a true or a false condition. The Propeller Spin language defines FALSE as 0 and TRUE as not zero (actually -1).

Table 12.3. Steps to Convert a Negative 2's-Complement Byte to a Useful Value for a Display.

	MSB							LSB	
Step Number	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**Data Actions**
1	1	1	1	1	0	0	1	1	Two's-complement value for decimal -13.
2	1	1	1	1	0	0	1	1	MSB = 1, so treat this as a negative 2's-complement number.
2	0	0	0	0	1	1	0	0	Invert each bit to create a 1's complement binary value.
3	0	0	0	0	0	0	0	1	Add 1 to one's complement value of 13 in the row above.
4	0	0	0	0	1	1	0	1	Sum of 1 plus 1's complement of -13, which equals 13.
5	0	0	0	0	1	1	0	1	Negative number (see Step 2), so print a minus sign and this value, 13, for -13.

The shaded box in **Table 12.3**, Step 2, holds the MSB for an 8-bit 2's-complement value, so a 1 here indicates a negative value. Software can detect this 1 and then set a flag, which I called `Minus_Flag`. Later the software can test the `Minus_Flag` for a TRUE or FALSE state. If TRUE, the software would display a minus sign. If FALSE, the software could print a plus sign or a blank space. As an alternative, the software could avoid using a flag and simply print a minus sign as soon as it detected a negative number.

Step 9.

Now you will learn how to apply the 2's-complement operations to the negative-temperatures shown previously in **Table 12.1**. Those values from the DS1620 IC have nine bits of data rather than eight used in the previous examples. But the same rules still apply, and the MSB represents the sign of the value. Let's start with a temperature of -25°C, which the DS1620 reports as 111001110_2. The MSB indicates we have a negative number, so our program set the `Minus_Flag` to TRUE.

Now, take the 1's complement of 111001110_2, which equals 000110001_2 and then add 000000001_2 to it. You should get the result: 000110010_2. What decimal value does that binary number equal? (32 + 16 + 2 = 50)

Because you started with the binary equivalent of -25 °C, you probably think the answer should equal -25. But remember, the DS1620 reports temperatures in *half-degree* steps, so 50 half-degree steps takes you from 0 °C down to -25 °C. To ignore the half-degree steps simply divide the result by 2. In this case, 50/2 = 25. And because `Minus_Flag` equals TRUE, you put a minus sign in front of the answer: -25.

How does a computer quickly perform a divide-by-2 operation? It takes a binary value and shifts it one place to the right. If you take 50 in binary (00110010_2) and shift it one place to the right, you get 00011001_2, which equals 25_{10}. Go through the 2's-complement conversion steps shown in **Table 12.3** with the nine-bit value for -55 °C. Did you get the proper result?

Step 10.

The flow chart shown in **Figure 12.6** provides an algorithm that converts 9-bit binary values from a DS1620 sensor into whole-number Celsius temperature values. This algorithm divides the temperature data by 2 to ignore half-degree values. You can run **Program 12.2** to see temperatures in degrees Celsius displayed in the PST window.

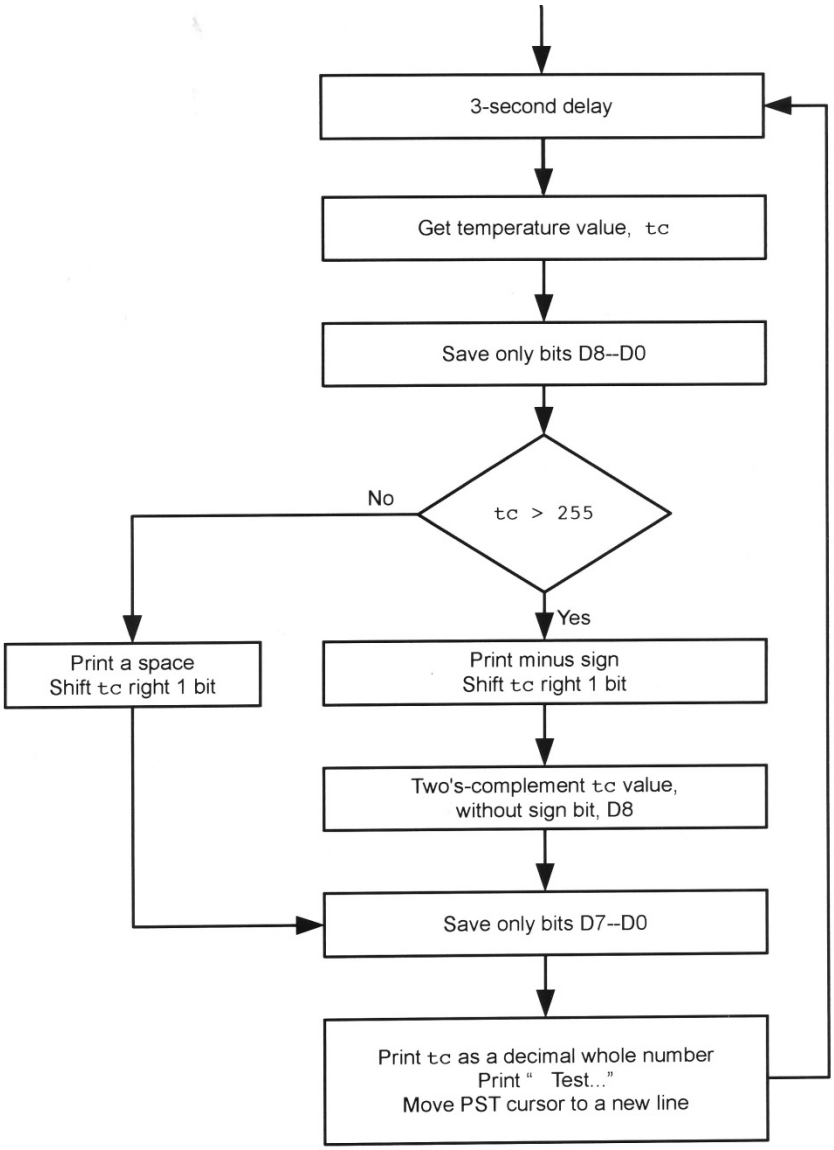


Figure 12.6.
This flow chart shows the steps needed to produce useful temperature values from the DS1620 temperature sensor's 9-bit binary values. These steps correspond to those contained within the `repeat` loop used in **Program 12.2**.

Program 12.2

```
{{
''*********************************************
''*  Program 12.2.spin
''*  Author: Jon Titus 11-11-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Program used to obtain a 9-bit temperature value
''*  from a DS1620 Digital Thermometer and Thermostat.
''*  This program does not use the thermostat functions.
''*  Display temp on PST as temp degrees C with sign.
''*  Based on program DS1620_Demo, written by John
''*  Williams, Parallax, 28 March 2006.
''*  Uses P8X32A Propeller-I board.
```

```
''*********************************************
}}

CON

  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal

''Create variables for DS1620 control pins and serial output for ''the
Parallax Serial Terminal (PST).
VAR
  byte PropPin_DQ               'Variable for DS1620 DQ, pin 1
  byte PropPin_CLK              'Variable for DS1620 CLK, pin 2
  byte PropPin_RST              'Variable for DS1620 RST*, pin 3
  byte PropPin_SerOut           'Variable for P8X32A pin P30
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for serial comm format
  word SerPort_BaudRate         'Variable for serial bit rate

OBJ

  temp  :  "ds1620_JT2"         'ds1620_JT2,spin file
  delay :  "timing"             'timing.spin file
  pst   :  "FullDuplexSerial"   'Serial library for testing

'Main program starts here. The tc defines a local variable
'available for use only in the "main" object.

PUB main | tc
  PropPin_DQ := 24                  'P24 for DS1620 DQ pin1
  PropPin_CLK := 25                 'P25 for DS1620 CLK pin 2
  PropPin_RST := 26                 'P26 for DS1620 RST pin 3
  PropPin_SerOut := 30              'P30 for USB to host PC
  PropPin_SerIn := 31               'P31 for USB from host PC
  PropPin_SerMode := 0              'Invert RX mode
  SerPort_BaudRate  := 9600         'Baud rate at 9600 per sec

'Configure and start DS1620 temperature conversions
  temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Configure Propeller serial port to communicate with host PC
'via Parallax Serial Terminal (PST)
  pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

'Loop to get temperature and display it as a signed decimal 'value in degrees
C every 3 seconds.
 repeat
   delay.pause1ms(3000)                 '3 second delay
   tc := temp.gettempc                  'Get temp via
                                        'DS1620_JT2.spin
        tc := tc & $1FF                 'Clear all bits except
                                        'those from DS1620
        if tc > 255                     'Check MSB of temp data
            pst.str(String("-"))        'If MSB = 1, print minus
            tc := tc >> 1               'Shift temp 1 bit to
                                        'right to div by 2
            tc := 1 + !(tc)             'Get 2's complement of
```

```
                                    'minus temp value

        else                        'OK, temp is positive
            pst.str(String(" "))    'print a space
            tc := tc >> 1           'Shift temp 1 bit to
                                    'right to div by 2
            tc := tc & $FF          'Clear all bits except
                                    'D7-D0
            pst.dec(tc)             'Display temp (tc) as
                                    'decimal on PST
            pst.str (String("  Test..."))
            pst.Tx(13)              'Move cursor to new line
                                    'Run loop forever
' -  - -End -  - -
```

When you run this program, use the coolant spray or an ice cube inside two sealable plastic sandwich bags to decrease the temperature of the DS1620 IC below 0 °C (32 °F). Use the PST window to watch the temperature change as the IC warms. **Figure 12.7** shows some of my results.

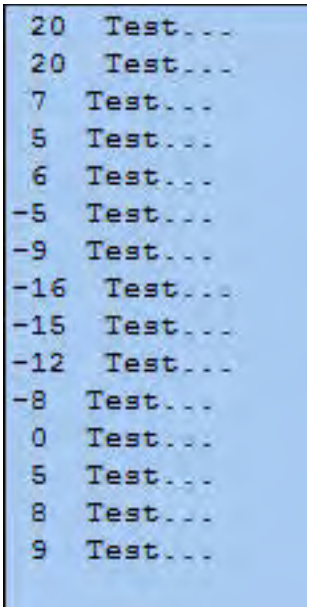


Figure 12.7.
Results created by **Program 12.2** as it takes temperature values from a DS1620 sensor every three seconds and reports them as degrees Celsius.

Now you have software that reports a temperature every three seconds. Most temperatures do not change much in such a short time, so you could increase the time delay if you so choose. For lab work, a measurement period might cover several minutes. A real-world instrument might need to track temperatures for months and display the highest-, lowest-, and average-temperature values.

Step 11.

For each 9-bit temperature value, the code in **Program 12.2** uses a comparison:

```
if tc > 255
```

to determine whether the DS1620 has reported a positive or a negative temperature. How does that statement do its job? In a 9-bit binary number from the DS1620 IC:

D8 D7 D6 D5 D4 D3 D2 D1 D0

the sign bit, D8 in the 2's-complement data, contributes either 0 or 256 to the overall value. Bit D7 contributes 0 or 128, bit D6 contributes 0 or 64, and so on. Refer back to **Table 12.1** for examples of raw data in decimal format. Whenever bit D8 equals 1 for a negative temperature, the 9-bit raw temperature value, tc in the code section that follows, must equal or exceed 256. Thus, any raw 9-bit value greater than 255 must represent a negative temperature, and that condition causes the program to print a minus sign.

```
repeat
  delay.pause1ms(3000)       '3 second delay (3000 msec.)
  tc := temp.gettempc        'Get temp via DS1620_JT2.spin
  tc := tc & $1FF            'Clear bits not from DS1620
  if tc > 255                'Check MSB of temp data
     pst.str(String("-"))    'If MSB = 1, print minus
     tc := tc >> 1           'Shift temp 1 bit to right
     tc := 1 + !(tc)         'Get 2's complement of minus temp
  else
     pst.str(String(" "))    'If positive temp, print a space
     tc := tc >> 1           'Shift temp 1 bit to right
     tc := tc & $FF          'Clear all bits except D7-D0
     pst.dec(tc)             'Display temp (tc) on PST

  pst.str (String("  Test..."))  'Some text to display
  pst.Tx(13)                     'Move cursor to new line
```

Step 12.

Many instruments must record a maximum and a minimum temperature measured over a given period. Can you modify **Program 12.2** to display the minimum and maximum temperatures as well as the current temperature in the PST window? Hint, you must define two additional long variables: `Temp_Max` and `Temp_Min`.

Probably your program would compare the `Current_Temp` value to the `Temp_Max` value, and if `Current_Temp` exceeds `Temp_Max`, then `Current_Temp` becomes the new `Temp_Max.` A similar comparison of `Current_Temp` and `Temp_Min` will let you update the lowest temperature. But what values should you assign to `Temp_Min` and `Temp_Max` to start? It might seem counterintuitive, but for the DS1620 sensor, I set `Temp_Min` to 125 °C and set `Temp_Max` to -55 °C. When you run **Program 12.3**, these values ensure the first maximum and minimum temperature equals the current temperature.

Program 12.3.

```
{{
''*********************************************
''*  Program 12.3.spin
''*  Author: Jon Titus 11-11-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Program used to obtain a 9-bit temperature value
''*  from a DS1620 Digital Thermometer sensor and display
''*  the current temperature along with the minimum and
''*  maximum temperatures in the PST window.
''*  For the PST program, press F12 within Propeller Tool.
''*  Based on program DS1620_Demo, written by John
''*  Williams, Parallax, 28 March 2006.
''*  Uses P8X32A Propeller-I board.
```

```
''*********************************************
}}

CON

  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal


''Create variables for DS1620 control pins and serial output for ''the
Parallax Serial Terminal (PST). Added variables for min
'' and max temperatures.
VAR

  byte PropPin_DQ               'Variable for DS1620 DQ, pin 1
  byte PropPin_CLK              'Variable for DS1620 CLK, pin 2
  byte PropPin_RST              'Variable for DS1620 RST*, pin 3
  byte PropPin_SerOut           'Variable for P8X32A pin P30
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for serial comm format
  word SerPort_BaudRate         'Variable for serial bit rate

  long Current_Temp             'Long variable for temperature
  long Temp_Max                 'Long variable for max temp
  long Temp_Min                 'Long variable for min temp

OBJ

  temp  :  "ds1620_JT2"         'ds1620_JT2,spin file
  delay :  "timing"             'timing.spin file
  pst   :  "FullDuplexSerial"   'Serial library for testing

'Main program starts here. The tc defines a local variable
'available for use only in the "main" object.
PUB main | tc

PropPin_DQ := 24              'P24 for DS1620 DQ pin1
PropPin_CLK := 25             'P25 for DS1620 CLK pin 2
PropPin_RST := 26             'P26 for DS1620 RST pin 3
PropPin_SerOut := 30          'P30 for USB to host PC
PropPin_SerIn := 31           'P31 for USB from host PC
PropPin_SerMode := 0          'Invert RX mode
SerPort_BaudRate  := 9600     'Baud rate at 9600 per sec


'Configure and start DS1620 temperature conversions
temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Configure Propeller serial port to communicate with host PC
'via Parallax Serial Terminal (PST)
pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

'Loop to get temperatures and track maximum and minimum and 'display them and
ambient temperature every 3 seconds.
    Temp_Max := -55             'Set max temp to lowest
    Temp_Min := 125             'Set min temp to highest
    repeat
```

```
        delay.pause1ms(3000)    '3 second delay
        tc := temp.gettempc     'Get temp via DS1620_JT2.spin
        tc := $FF & (tc >> 1)   'Shift tc one bit to the left
                                'and clear all bits except D7-D0
        Current_Temp := ~tc     'Sign-extend tc into
                                'Current_Temp
        pst.dec(Current_Temp)   'Print Current_temp as decimal
        pst.str (String("  Your text..."))

        if (Current_Temp > Temp_Max)    'Compare current temp
                                        'to highest, if higher
             Temp_Max := Current_Temp   'save it as highest

        if (Current_Temp < Temp_Min)    'Compare current temp
                                        'to lowest, if lower
                                               'save it as lowest
            Temp_Min := Current_Temp

        pst.dec(Temp_Max)               'Print max temp
        pst.str (String("  Max    "))
        pst.dec(Temp_Min)               'Print min temp
        pst.str (String("  Min"))
        pst.Tx(13)                      'New line
                                        'Run loop forever
' –  – -End –  – -
```

Step 13.
If you plan to run the next experiment, keep the DS1620 sensor connected to your P8X32A MCU board. You may disconnect the P8X32A Propeller board from its USB cable to turn off power.

Reference

"DS1620 Digital Thermometer and Thermostat" data sheet, Maxim Integrated.
http://datasheets.maximintegrated.com/en/ds/DS1620.pdf.

Notes

Binary addition follows the rules below for two values in a column:

```
 0     0     1     1
+0    +1    +0    +1
 0     1     1     0  and a carry of 1 for the column to the left
```

When you have a carry bit (highlighted below), addition in a column follows these rules:

```
 1     1           1           1
 0     0           1           1
+0    +1          +0          +1
 1     0  and a carry  0  and a carry  1  and a carry
```

Answers

Experiment 12, Step 7:

$37_{10} = 0100101_2$.

Two's complement of $0100101_2 = 1011010_2$

Add 1 to 1011010_2 and get 11011011_2 for -37.

$23_{10} = 00010111_2$

Two's complement of $00010111_2 = 11101000_2$

Add 1 to 11101000_2 and get 11101001_2 for -23.

$94_{10} = 01011110_2$

Two's complement of $01011110_2 = 10100001_2$

Add 1 to 10100001_2 and get 10100010_2 for -94

Experiment No. 13 – Create A Thermometer with a Digital Display

Abstract

In Experiments 11 and 12 you learned about the MAX7219 display-driver and the DS1620 temperature-sensor integrated circuits. This experiment combines the those ICs to create a digital thermometer. Software from past experiments gives you the foundation for this experiment. In addition to creating a digital thermometer, you will learn how Propeller objects can share information saved in an array. This knowledge increases your ability to have many objects use the same information. If you skipped Experiments 11 and 12, I recommend you do them before you run this one.

Keywords

DS1620, MAX7219, thermometer, serial communications, temperature measurements, temperature sensor, Propeller, object, software, arrays, argument passing

Requirements

Please see Experiment 11 and Experiment 12

Introduction

In this short experiment you will use the display circuit from Experiment 11 and the temperature-sensor circuit from Experiment 12 to create a digital thermometer. Hopefully you still have both circuits set up and ready to use. If not, please run Experiments 11 and 12 so you have the circuits and know they work. This experiment also requires software from these experiments, as explained in the steps that follow. The final digital thermometer will display temperatures in degrees Celsius with a minus sign for negative temperatures.

Step 1.

You should have the DS1620 sensor IC already connected to your Propeller board as shown in **Figure 13.1**. This circuit takes power from the Propeller board. Run **Program 12.2** from Experiment 12 to confirm the DS1620 operates properly. You should see Celsius temperatures appear in the Parallax Serial Terminal (PST) window.

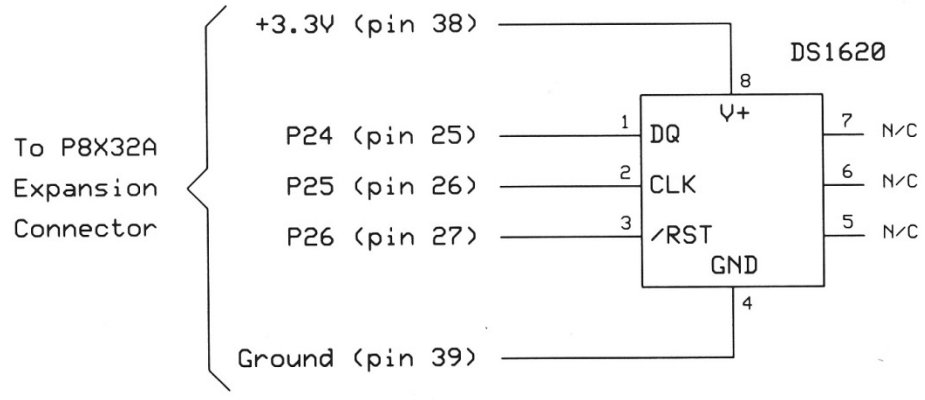


Figure 13.1.
Connections between a DS1620 temperature-sensor IC and a Propeller P8X32A board. For more information, please review Experiment 12.

Step 2.

Reconnect the multi-digit display circuit (see **Figure 11.9** in Experiment 11) to your Propeller MCU. Find a larger version of this figure in the Experiment 11 software folder. The display circuit, which uses a Maxim Integrated Products MAX7219 display-driver IC, requires only one ground and three signal connections to the Propeller board. Remember: The display circuit uses a 74HC04 inverter IC so the 3.3-volt logic levels from the Propeller board can drive the 5-volt logic inputs on the MAX7219 IC. You also need a 5-volt power supply for the display circuits. Ensure you have a common ground that connects ALL circuit and power-supply grounds. **Figure 13.2** illustrates *only* the power and ground connections between the circuits and the Propeller board.

Keep ground wires short and try to place them close together on your breadboards. A single-point ground will help reduce electrical noise on circuit signals. I recommend you distribute several 0.1 µF disc ceramic capacitors around your breadboards between the power and ground buses. These capacitors will help "absorb" short power changes due to switching and control signals elsewhere in the circuits. Engineers use the term decoupling capacitors to describe them.

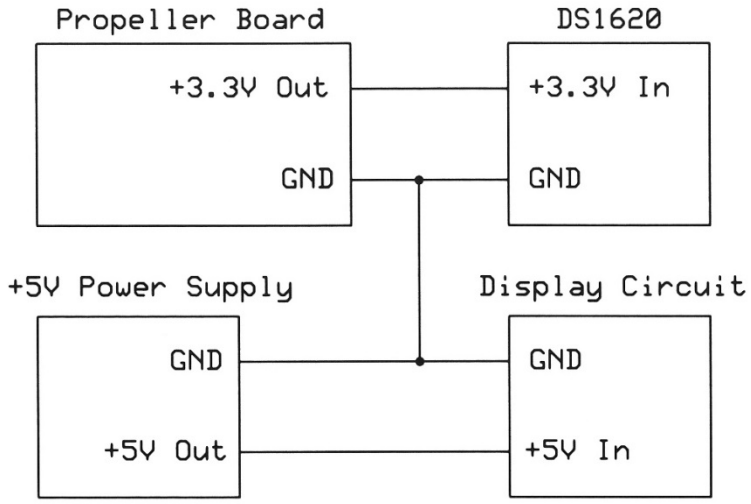


Figure 13.2.
This diagram shows only the power and ground connections among the circuits.

The photograph in **Figure 13.3** shows my lab setup for the digital thermometer. I put the DS1620 on a separate breadboard and obtained 3.3V power from the Propeller P8X32A board. A separate power supply provided the 5 volts for the display circuit.

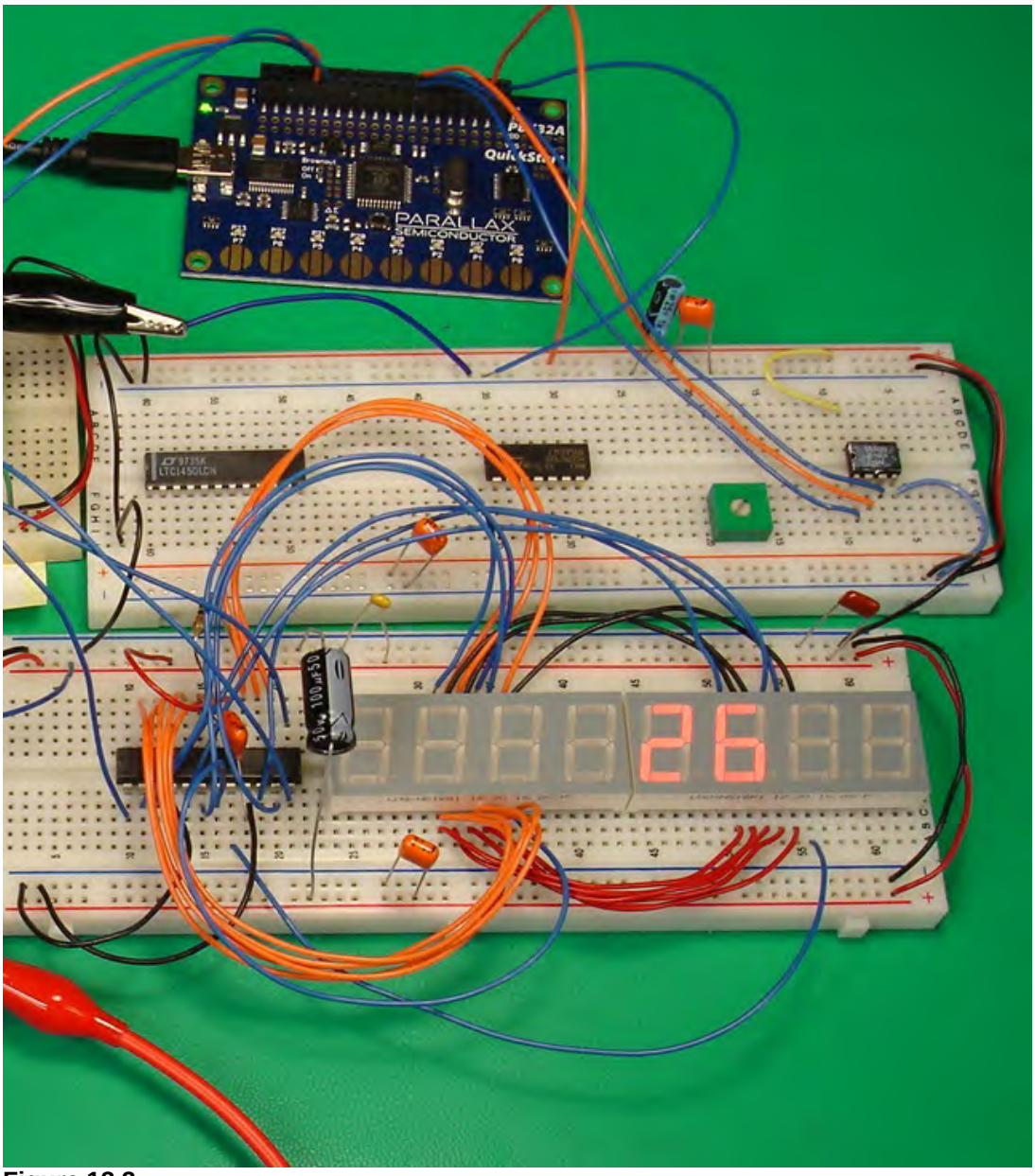


Figure 13.3.
The lab setup for the digital-thermometer experiment. I used a different logic-level-conversion circuit (not shown) for this test. Under the photography lamps, the DS1620 sensor measured a temperature of 26°C (78°F).The upper breadboard holds the sensor and components used in other experiments.

Step 3.

After you have the display circuit and the DS1620 sensor IC connected to your Propeller board, run **Program 11.3** from Experiment 11 to confirm your display works properly. This program will display 01234567 on the eight-digit display module, or modules, depending on how you wired the display circuit. (Of course if you have fewer than eight 7-segment displays, you'll see fewer digits.)

If you do not see the digits light up, recheck the wiring of the display circuit, or rerun Experiment 11. Many problems come down to improper or incomplete wiring of a common ground circuit that connects all circuits, breadboard power buses, and power sources.

Step 4.

Now you must "link" the display software with the software that obtains temperatures from the DS1620 sensor. In Experiment 11 you ran **Program 11.4** that took *segment patterns* saved in the array named `digit[x]` and displayed them as a message. The following statement from **Program 11.4** set the MAX7219 so we could control individual segments:

```
MAX7219_init[0] :=  $0900   'No characters, segment control
```

To display only digits (along with a minus sign, a blank, and the letters “H”, “E”, “L”, and “P”) we must set MAX7219 register $09 for "Code B":

```
MAX7219_init[0] :=  $09FF   'Code-B numbers and symbols
```

Do not change **Program 11.4**. I have saved a copy for this experiment as **Program 13.1**.

Step 5.

If not already running, start the Propeller Tool program and then open **Program 13.1**. Run it on your Propeller MCU board. You should see my message to you "HI-Jon". Find the statement:

```
MAX7219_init[0] :=  $0900   'No characters, segment control
```

and change it to:

```
MAX7219_init[0] :=  $09FF   'Code-B numbers and symbols
```

Go to the upper-left corner of the Propeller Tool window and click on File. Select Save-As... and locate the Experiment 13 folder in the window that opens. Give this program the name: **Program 13.2** and save it. (If the Experiment 13 folder already includes **Program 13.2**, you may overwrite it, or save the file as Program 13.2X.spin.) Now you have a copy of the program set for Code-B numerals and characters. Run the program. What do you see?

On my 7-segment displays I saw `701HL500`. Do you know why?

Within **Program 13.2**, find the array `digit[0]` through `digit[5]`. This array contains six bytes that controlled individual segments that formed the message "HI-Jon."

```
digit[0] := $37   'Segments for letter H
digit[1] := $30   'Segments for letter I
digit[2] := $01   'Segments for a dash
digit[3] := $3C   'Segments for letter J
digit[4] := $1D   'Segments for letter o
digit[5] := $15   'Segments for letter n
```

After you change the program command to: `MAX7219_init[0] := $09FF`, the MAX7219 display driver interprets the bytes in the array as codes for *digits* you want to display. In the Code-B mode, the MAX7219 recognizes only the four bits D3 through D0 (**Table 13.1**). Thus \$37 (00110111_2) becomes 0111_2 for the display driver, so you see a "7" on the first 7-segment display. Likewise, \$30 (00110000_2) displays a "0" and \$01 (00000001_2) displays a "1," and so on. So your *segment codes* get interpreted by the MAX7219 as *digit codes*.

Table 13.1. Code-B Characters and Symbols.

7-Segment	Register Data							
Character	D7	D6	D5	D4	D3	D2	D1	D0
0	DP	X	X	X	0	0	0	0
1	DP	X	X	X	0	0	0	1
2	DP	X	X	X	0	0	1	0
3	DP	X	X	X	0	0	1	1
4	DP	X	X	X	0	1	0	0
5	DP	X	X	X	0	1	0	1
6	DP	X	X	X	0	1	1	0
7	DP	X	X	X	0	1	1	1
8	DP	X	X	X	1	0	0	0
9	DP	X	X	X	1	0	0	1
- (Minus)	DP	X	X	X	1	0	1	0
E	DP	X	X	X	1	0	1	1
H	DP	X	X	X	1	1	0	0
L	DP	X	X	X	1	1	0	1
P	DP	X	X	X	1	1	1	0
(Blank)	DP	X	X	X	1	1	1	1
Notes:	X = Don't care.							
	DP = Decimal Point 0= off; 1= on							

Step 6.

Change the value for `digit[0]` to $87 and run **Program 13.2** again. Do you see any difference in the displayed information? Look closely. You should see a decimal point turned on.

When set to display Code-B characters, the MAX7219 uses the most-significant bit (D7) in each value to determine whether or not to light the associated decimal point. The hexadecimal value $87 ($\underline{1}0000111_2$) has a "1" as its MSB, so the decimal point for the first digit gets turned on. Change one of the other digit values to turn on its decimal point.

Step 7.

You probably noticed the `digit[x]` array provides only six digits to display, so why do the two right-most digits display a 0 when the program includes no data for them? When you define an array, the Propeller software automatically presets all values in the array to 0.

Put an apostrophe at the beginning of the six statements that set the values for the 7-segment displays:

```
'digit[0] := $87
'digit[1] := $30
'digit[2] := $01
'digit[3] := $3C
'digit[4] := $1D
```

```
'digit[5] := $15
```

We call this action "commenting out" code. The apostrophe "converts" these statements to comments and the Propeller software ignores them. Now run **Program 13.2** again. What do you see on the display? You should see 00000000 because the Propeller software preset all values in an array to zero when it created the array: `byte digit[8]`. This statement defines *eight* members of the array and presets them all to zero, so you see eight 0's on the display.

Step 8.

Program 13.2 goes through the display steps only once. Unlike previous programs that controlled shift registers, **Program 13.2** does not continue to update or refresh the display with any new information. The data in the `digit[x]` array gets sent only once to the MAX7219 controller. This IC multiplexes the displays without further interaction with the Propeller MCU.

To make the MAX7219 control operations useful to other programs, I modified the **Program 13.2** code to create three separate objects:

```
Start_MAX7219           'Initialize the MAX7219
Clear_MAX7219           'Clear the display
Display_MAX7219         'Send new data to the MAX7219
```

You will use the three objects named above to display and update temperature information. To save time and avoid typing errors, find my modified code in the **Program 13.3** file. You can open it, and examine it, and print a listing. But do not try to run it. **Program 13.3** includes another change I'll explain shortly. By the way, any program can use the MAX7219 objects in **Program 13.3**. If you want to display other information, you need not start with a blank sheet of paper. What other objects might you need to control a MAX7219? Find suggestions in the Answers section at the end of this experiment.

Program 13.3.

```
{{
''*********************************************
''*  Program 13.3 Display_MAX7219 objects
''*  Author: Jon Titus 11-12-2014 Rev. 3
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Spin objects to start and clear a MAX7219
''*  display-driver IC and eight 7-segment displays.
''*  The Display_MAX7219 object takes values from
''*  an array and displays them.
''*  SPI_Asm object library used for SPI-type
''*  communication.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x      'Set MCU clock operation
    _xinfreq = 5_000_000           'Set for 5 MHz crystal

OBJ
  SPI  :  "SPI_Asm"                'Invoke the SPI_Asm.spin file

VAR
    word MAX7219_init[10]          'Storage for init data
    word display_loop              'Counter for display loops
    byte data_port                 'Data-output pin to shift reg
    byte serial_clock              'Pulse for shift reg clock
```

```
    byte chip_select               'Pulse for LOAD pin
    word data16_out                '16-bit value for SPI
                                   'transmission

'=====================
'Start object to set up preset pin assignments, and
'initialization data.

PUB Start_MAX7219                  'SPI Setup
  SPI.start(15,0)
  dira[8..10] := %111              'Output pins for SPI and LOAD
                                   'signals
  outa[8..10] := %000                'Set outputs to 0

  data_port := 8                     'Use P8 for SPI data
  serial_clock := 9                  'Use P9 for SPI clock
  chip_select := 10                  'Use P10 for MAX7219 LOAD

'MAX7219 initialization data saved here:
MAX7219_init[0] :=  $09FF          'B Code mode
MAX7219_init[1] :=  $0A07          'Half intensity for LEDs
MAX7219_init[2] :=  $0B07          'Use all 8 digits
MAX7219_init[3] :=  $0C01          'Normal Operation
MAX7219_init[4] :=  $0F00          'Not a Test

'Initialize MAX7219 IC with MAX7219_init values
    outa[chip_select] := 0         'Reset MAX7219 LOAD signal

    repeat display_loop from 0 to 4     'Send MAX7219 five
                                        'parameters
       data16_out := (MAX7219_init[display_loop])
       MAX7219_SPI_out

'=====================
'Clear all 8 digits controlled by MAX7219.
'Send MAX7219 register number and value $0F to blank all
'digit positions.

PUB Clear_MAX7219
    repeat display_loop from 0 to 7
        data16_out := ((display_loop + 1)  <<8) + $0F
        MAX7219_SPI_out

'=====================
'Display values saved in an array. Use array pointer to access
'values.

PUB Display_MAX7219(ptr)
      repeat display_loop from 2 to 5
        data16_out := ((display_loop + 1)  <<8) + byte[ptr][display_loop]
        MAX7219_SPI_out

'=====================
'SPI-output object, uses SPI_asm.spin file
'strobes MAX7219 LOAD pin after transmission

PUB  MAX7219_SPI_out
```

```
      SPI.SHIFTOUT(data_port, serial_clock, SPI#MSBFIRST, 16, data16_out)
      waitcnt(clkfreq/100000 + cnt)   'Delay for SPI to end

      outa[chip_select] := 1          'LOAD pin to logic-1
      outa[chip_select] := 0          'LOAD pin to logic-0

' –  – -End –  – -
```

Step 9.

Now we'll look at the main program that gets a 9-bit temperature value from the DS1620 sensor and uses the MAX7219 objects to display it. Locate the complete program, **Program 13.4**, in the Experiment 13 software folder. You do not need to modify any code to have a working digital thermometer. I will take you through portions that start after the program listing.

Program 13.4.

```
{{
''*********************************************
''*  Program 13.4 Digital Thermometer Program
''*  Author: Jon Titus 11-12-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Display demonstration for a digital thermometer
''*  Used DS1620 IC as a temperature sensor and a
''*  MAX7219 display-driver IC to display Celsius
''*  temperatures
''*  SPI_Asm object library used for SPI-type
''*  communication
''*********************************************
}}

CON _clkmode = xtal1 + pll16x     'Set MCU clock operation
    _xinfreq = 5_000_000          'Set for 5 MHz crystal

OBJ
  MAX7219   :  "Program 13.3"   'Objects here control MAX7219
  temp      :  "ds1620_JT2"     'ds1620_JT2,spin file for sensor
  delay     :  "timing"         'timing.spin file


VAR
  byte digit[6]                 'Array for temperature digits
  byte PropPin_DQ               'Variable for DS1620 DQ, pin 1
  byte PropPin_CLK              'Variable for DS1620 CLK, pin 2
  byte PropPin_RST              'Variable for DS1620 RST*, pin 3
  long Current_Temp             'Long variable for signed
                                'temperature value
  byte hundreds                 'Hundreds value storage
  byte tens                     'Tens value storage

'Main program starts here with tc as a local variable
PUB main | tc

  MAX7219.Start_MAX7219          'Initialize the MAX7219
  MAX7219.Clear_MAX7219          'Turn all digits off
  PropPin_DQ   := 24             'P24 for DS1620 DQ
  PropPin_CLK  := 25             'P25 for DS1620 CLK
```

```
  PropPin_RST  := 26              'P26 for DS1620 RST

  'Configure and start DS1620 temperature conversions
  temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

  'This loop gets a new temperature and displays it
  'as a decimal value every four seconds.

    repeat
        delay.pause1ms(2000)         '2 second delay
        tc := temp.gettempc          'Get temp "tc" via
                                     'DS1620_JT2.spin
        delay.pause1ms(2000)         '2 second delay
        tc := $FF & (tc >> 1)        'Shift tc one bit left
                                     'to divide by 2
        if (tc > 127)                'Look at MSB for temp sign
            digit[2] := 10           'Negative temperature, so
                                     'set digit 2 for minus
            tc := 1 + !(tc)          'Get 2's complement of
                                     'minus temp value
        else                         'If temperature is
                                     'positive, blank digit
            digit[2] := 15           '2 on display.

        tc := tc & %01111111         'Set sign bit to 0
        hundreds := tc/100           'Divide temp value by 100'
        if (hundreds == 0)           'Test for temp of 100 or
                                     'greater
            digit[3] := 15           'No 100's, so blank digit 3
        else
            digit[3] := 1            'Can only have a 1 in 100s
                                     'so set digit 3 to "1"
                tc := tc - 100       'Then subtract 100 from
                                     'temperature value
        tens := tc/10                'Check for no 10s and no
                                     '100s in temp
        if (tens == 0) AND (hundreds == 0)
            digit[4] := $0F          'No tens AND no hundreds,
                                     'so blank ten's digit
        else
            digit[4] := tens        'Have some 10s, so display
                                     'ten's digit

        tc := tc - (tens * 10)       'Get 1s value
        digit[5] := tc               'Set digit 5 to 1s value

        MAX7219.Clear_MAX7219            'Clear the display
        MAX7219.Display_MAX7219(@digit)  'Display latest temp

' –  – -End –  – -
```

a. The program indicates it will require Propeller objects in **Program 13.3**. The statement below invokes those objects and assigns them the prefix MAX7219 for use in this program:

```
OBJ
MAX7219   :  "Program 13.3"    'Objects here control MAX7219
```

When you need to use one of these three objects (see Step 8), add the prefix `MAX7219` to the object name, as in:

`MAX7219.Clear_MAX7219`.

b. The program defines an array of bytes in which it will save the values of the individual digits to display:

```
VAR
byte digit[6]             'Array to save temperature digits
```

c. A series of statements that start with the line `if (tc > 127)` determines whether the program has received a positive or a negative temperature, sets a minus sign for negative temperatures, and then separates the temperature value into 100's, 10's, and 1's digits to display. These values go into the `digit[x]` array as shown below:

```
digit[0]:  not used
digit[1]:  not used
digit[2]:  minus sign or blank
digit[3]:  100's digit
digit[4]:  10's digit
digit[5]:  1's
```

Program 13.4 differs slightly from **Program 12.2** in that it shifts the 9-bit temperature value one bit to the right to divide by 2. Then it examines the D7 bit for a negative 2's-complement value. **Program 12.2** examined bit D8 to determine the sign of a temperature and then divided it by two.

d. After the program has the sign and the various digits ready to display, the statement below clears the display to make it ready for a new value.:

```
MAX7219.Clear_MAX7219
```

e. Finally, the program uses the statement:

```
MAX7219.Display_MAX7219(@digit)
```

to cause the `Display_MAX7219` object in **Program 13.3** to display the digits. But the `@digit` notation for the array doesn't look at all like `digit[x]` notation used to put values in the array. What happened?

In previous experiments, a "call" to use an object looked like this:

```
temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)
```

and it transferred three values to the object `start` which exists in another piece of software. That approach works well for a few values, but suppose you want to transfer an array of 150 values to an object; what then? Perhaps we would need a statement such as:

`abc.average(data[1], data[2], data[3],..` and so on up to `...data[150])`

That's a lot of values to pass to the averaging object! (And a lot of typing, too.) In fact, you cannot handle an array's data that way. But because an array saves information in an ordered fashion, we simply tell a program

where to find the first element in the MCU's memory and the internal Spin software takes care of the rest automatically. That means we give the object `MAX7219.Display_MAX7219` the *memory address* of the first element, noted as `@digit`. We don't need to know that address – the MCU and the Spin language keep track of it. Programmers call the address a *pointer*.

Start **Program 13.4**. What do you see on the displays? After about four seconds, you should see a Celsius temperature. In my lab I usually saw 22 or 23. Those temperatures match what I saw on a commercial digital thermometer.

Step 10.

A Bit More About Pointers

At first glance, the diagram in **Figure 13.4** might look confusing, but the text below explains what happens, step by step. This example uses two programs, start_up.spin and do_something.spin. These program don't include all the code needed to make them work; they simply help illustrate how pointers can simplify addressing members of an array. The numbered explanations refer to the lines and arrows in **Figure 13.4**.

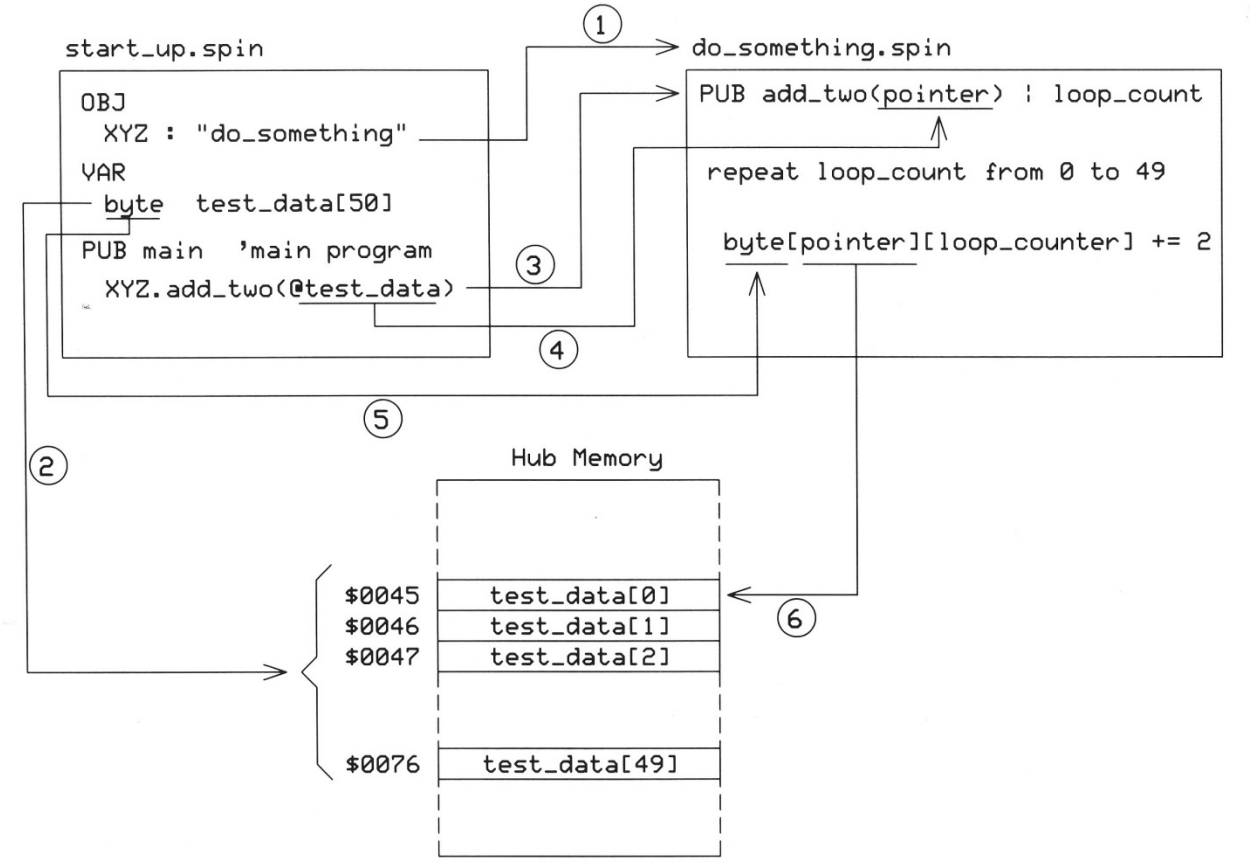


Figure 13.4.
This diagram shows the relationships between the start_up.spin and the do_something.spin programs and the way the start_up program passes an array pointer to the do_something program. The text explains the numbered connections between these two programs.

Operation 1. This portion of the start_up.spin program:

```
OBJ
   XYZ : "do_something"
```

indicates the start_up.spin program will use objects within the do_something.spin program. The prefix XYZ in the start_up.spin program simply gives us an identifier prefix that identifies objects in the do_something.spin program; for example XYZ.add_two, XYZ.subtract, XYZ.toggle_pin5, and so on. The XYZ prefix applies only within the start_up.spin program. (When you write code, use a name more descriptive than XYZ!)

Operation 2. The start_up program creates an array of 50 bytes in the Propeller's memory:

```
VAR
  byte  test_data[50]
```

The Propeller Tool assigns array members to specific memory locations based on the unused memory available. We don't know the addresses of those locations, not do we care. In this example, I chose the hexadecimal addresses $0045 through $0076 simply to illustrate memory operations.

Operation 3. Within the public (PUB) portion of the start_up.spin program, the statement:

```
XYZ.add_two(@test_data)
```

tells the Propeller to use the add_two object in the program identified here with the prefix XYZ. The XYZ, defined previously in Operation 1, links the add_two object to the do_something.spin code.

Operation 4. Now control passes from the start_up.spin code to the do_something code. This transfer also involves @test_data. The "at" sign, @, indicates we want the memory address for the start of the test_data array. In this case, @test_data equals $0045. The add_two object receives only this address from the start_up.spin code. (Again, we don't need to know the actual address.)

In the statement:

```
add_two(pointer) | loop_counter
```

The add_two object receives the array pointer and names it "pointer," although you can use any name except a reserved word. The notation "| loop_counter" defines a local variable for use only within this object.

The add_two object includes a repeat statement that runs the following instruction:

```
byte[pointer][loop_counter] += 2
```

fifty times, once per member of the array. The += notation has the Propeller take the current value saved in an array location, add two to it, and put the result back in the same array location. This operation is equivalent to:

```
byte[pointer][loop_counter] := 2 + byte[pointer][loop_counter]
```

The Propeller knows where the array starts (@test_data) as well as the member of the array you want to use, identified by the loop_counter value, so it can determine where to find the array data you need for this math operation.

Operation 5. Suppose a program defines an array of bytes and includes objects that will use an *address* to locate information in that array. The objects that will work on the array data do not know the type data – byte, word, or long – the array contains. Thus, whenever an object will use a pointer to locate array members, you must "tell" the object the type of data the array contains. The word byte in this statement:

```
byte[pointer][loop_counter] += 2
```

lets the Propeller know it has a pointer to an array of bytes.

Operation 6. In the statement immediately above, the combination of the `pointer` and `loop_counter` values lets the Propeller find each of the 50 array elements in the `repeat` loop. The value of `pointer` does not change. Don't worry about how the Propeller stores bytes, words, or longs in memory. It handles the details "behind the scenes" to keep everything in the proper order.

Step 11.

Next, a quick overview of how the digital thermometer software uses a pointer. In **Program 13.4**, the last statement includes the notation, `@digit`:

```
MAX7219.Display_MAX7219(@digit)
```

When the Propeller executes this instruction, it passes control to the `Display_MAX7219` object along with the pointer, `@digit`, to the first member of the `digit[x]` array.

Program 13.3 contains the `Display_MAX7219` object:

```
'Display values saved in an array
PUB Display_MAX7219(ptr)             'use array pointer...
   repeat display_loop from 2 to 5   'values in an array
     data16_out := ((display_loop + 1)  <<8) + byte[ptr][display_loop]
     MAX7219_SPI_out
```

The name in parentheses at the end of the statement:

```
PUB Display_MAX7219(ptr)
```

indicates this object will receive a value – also called an argument – in this case an address. But for now the object only "sees" this as a value. Only when the Propeller gets to the statement:

```
data16_out := ((display_loop + 1)  <<8) + byte[ptr][display_loop]
```

does it interpret the `ptr` value as an address pointer. The portion of the statement shown in boldface characters translates into, "use the value of `ptr` to locate an array of bytes and then use `display_loop` to locate a specific array member."

Step 12.

The next experiment provides more information about the DS1620 temperature sensor and how it can determine when a temperature goes above or below a temperature you program into the sensor. If you wish, you may skip this experiment, and remove the DS1620 sensor IC from your breadboard.

In either case, you can disconnect the 7-segment-display circuit from the Propeller P8X32A board. And you can disconnect its +5V power supply.

Reference

"DS1620 Digital Thermometer and Thermostat" data sheet, Maxim Integrated.
http://datasheets.maximintegrated.com/en/ds/DS1620.pdf.

"MAX7219/MAX7221 Serially Interfaced, 8-Digit LED Display Drivers" data sheet, Maxim Integrated. http://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf.

Answers

Experiment 13, Step 8:

It might help to have an object that would let you set the operating parameters such as decode mode, intensity, scan limit, and shutdown. **Program 13.3** includes these parameters, but as "hard coded" values that might not meet your requirements. Another object could control the test condition that lights all segments and decimal points.

Experiment No. 14 – Explore the DS1620 Sensor-Alarm Operations

Abstract

A DS1620 temperature sensor IC can report a value for the ambient temperature and it has other capabilities worth exploring. It also operates as a thermostat with an upper and a lower limit set with software, and three output pins let it control external devices based on temperatures. Because you have a DS1620 in a breadboard it's worth the time to examine how the three "alarm" outputs work. You'll also learn more about the single-wire communication technique the sensor uses. This knowledge will help you write code to control other single-wire serial-communication devices such as memories, clocks, and battery monitors.

Keywords

DS1620, serial communications, temperature measurements, temperature sensor, Propeller, object, software, single-wire, one-wire, hysteresis, flag, half duplex

Requirements

(1) - DS1620 Digital Thermometer and Thermostat integrated circuit, 8-pin DIP
(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - Can, freeze spray (MCM Electronics, part no: 20-3000), or an ice cube (see text)
(1) - USB cable
(1) - LED, red
(1) - LED, yellow
(1) - LED, green
(3) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)

Introduction

In addition to reporting a 9-bit temperature value, the Maxim Integrated DS1620 sensor IC can store a low- and a high temperature limit. When temperature reaches a limit, the DS1620 will change the state of an output pin to indicate an under- or over-temperature condition.

Before you can use the set-point capabilities, though, you should know a bit more about how a DS1620 sensor IC communicates with an MCU. The sensor uses a single wire to receive communications from an MCU and to send data to an MCU. A separate input on a DS1620 receives a clock signal created by the MCU to synchronize communications. These single-wire communications fall into three categories:

a) Send only a command to the DS1620 sensor,

b) Send a command and data sent to the DS1620,

c) Send a command to the DS1620 sensor, receive data from the DS1620.

I'll give a brief explanation of each operation and show how to use them in a program. All following examples require objects within the ds1620_JT2.spin program, so projects must include this file within their working directory. (The Experiment 14 software folder contains all needed files and references.) First, a program must identify the file that contains the DS1620 communication objects:

```
OBJ
      temp  :  "ds1620_JT2"
```

Now you can employ all public (PUB) objects in the ds1620_JT2.spin program by using their name with the `temp.` prefix. Don't forget the period after the prefix. Of course, a programmer may define his or her own prefix, such as:

```
OBJ
      Jane_sensor1  :  "ds1620_JT2"
```

and use the `Jane_sensor1.` prefix for objects in the ds1620_JT2.spin program.

Send a Command to a DS1620

Only two commands, `Start Convert ($EE)` and `Stop Convert ($22)`, fall in the command-only category. The ds1620_JT2.spin program includes a `CommandWrite` object that will transmit either of these two commands. The line below shows that object used in its general form:

```
temp.CommandWrite(command_goes_here)
```

The following specific example starts temperature measurements in a DS1620 sensor IC:

```
temp.CommandWrite($EE)
```

To avoid problems when people work with your programs, define the command as a byte and then use the command name with the `CommandWrite` code as shown here:

```
Start_Convert := $EE                /Define the DS1620 command
temp.CommandWrite(Start_Convert)    /Start temp convert
```

Figure 14.1 shows the timing for the three signals, DQ, CLK, and RST* for a Stop Convert command ($22, or 00100010_2). Remember, the DS1620 IC requires an MCU to transmit or receive the least-significant bit (LSB) first, so it appears to the left in the diagram.

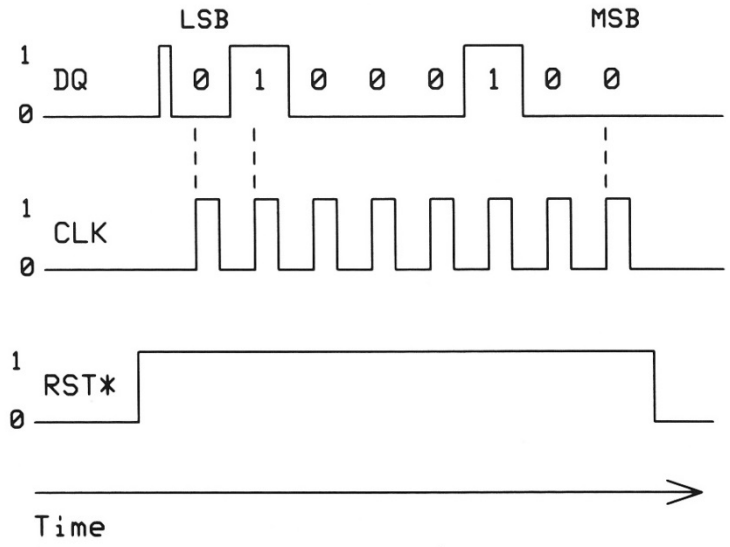


Figure 14.1.
Timing diagram for transmission of the $22 ($00100010_2$) command from an MCU to a DS1620 temperature sensor.

Send a Command and Data

When you send a command followed by data, things get a more complicated. A command always requires eight bits, The data portion of this type of transmission, however, requires either eight or nine bits. Eight bits of data follow a `Write_Config ($0C)` command, for example. But nine bits of data must follow a `Write_TH ($01)` command. (More about this command shortly.)

To simplify a command-and-data transmission, the ds1620_JT2.spin software includes the object, `SendDS1620`, as shown next:

```
temp.SendDS1620(command_goes_here, data_goes_here, total_bitcount_goes_here)
```

The following example illustrates how to load the nine least-significant bits of the long-type value $0050 ($00000000\_01010000_2$) into the TH register (command $01). A long variable holds 16 bits:

```
temp.SendDS1620($01, $0050, 17)
```

The data sheet for a DS1620 indicates the `Write_TH ($01)` command requires nine bits of data. In the example above, the value 17 arises from eight bits for the command *plus* nine bits for the transmitted data.

The statement above could confuse people. Who knows what the three values represent? (It could confuse people who know what the values mean, too!) Well-written code would define the DS1620 sensor IC commands and other constants as shown next. First the programmer defines constants for the commands. When the program runs, these values become unchangeable:

```
CON
  Write_Config    =     $0C   'Write to Configuration register
  Write_TH        =     $01   'Write to High Temp register
  Write_TL        =     $02   'Write to Low Temp register
  Start_Convert   =     $EE   'Start temperature conversions
  Nine_bit_data   =     17    '8-bit command + 9-bit data
  Eight_bit_data  =     16    '8-bit command + 8-bit data
  ...etc...
```

Second, the definition of variables defines space needed for data:

```
VAR
   byte TH_Data               'High-temp limit
   long Config_Data           'DS1620 configuration data
   ...etc...
```

Third, the variables get "loaded" with values:

```
TH_Data     := $050           'High-temp limit of 40 degrees C
Config_Data := $02            'Config bit D1 for MCU control
...etc...
```

Then the statement below makes more sense:

```
temp.SendDS1620(Write_TH, TH_Data, Nine_bit_data)
```

To send eight bits of data after a command, use the same object but with different information to configure a DS1620 sensor:

```
temp.SendDS1620(Write_Config, Config_Data, Eight_bit_data)
```

The `SendDS1620` object in the ds1620_JT2.spin program handles the operations that "splice" the command and data and transmit them to a DS1620 sensor. The timing diagram in **Figure 14.2** shows a transmission of the Write Configuration command $0C ($00001100_2$), followed by the 8-bit data $02, ($00000010_2$).

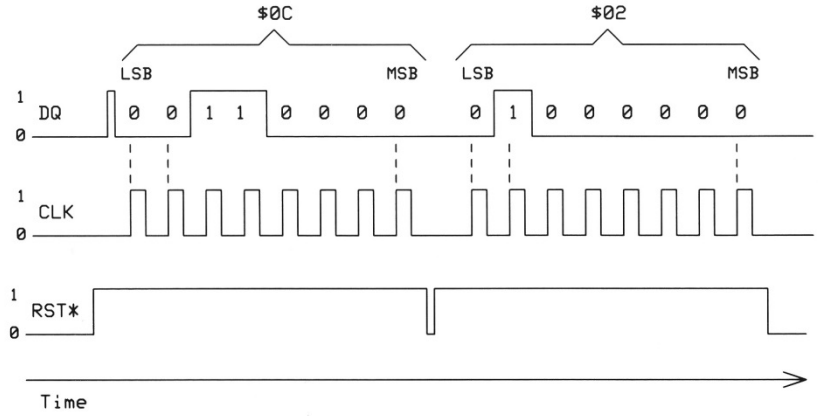


Figure 14.2.
Timing diagram for the command $0C that will put the data $02 in the DS1620 configuration register. This register controls how the DS1620 operates and it provides information about the status of several internal circuits. The vertical dashed lines show when the DS1620 IC accepts the serial DQ bits as the CLK signal makes a transition from logic-0 to logic-1.

Send a Command and Receive Data

Several previous experiments used the `gettemp` object in the ds1620_JT2.spin file, as shown in the example here:

```
tc := temp.gettemp
```

The `gettemp` object transmits the command Read Temperature ($AA) to the DS1620 sensor IC. Then the Propeller program switches the serial-output pin so it serves as a serial-input pin. The MCU continues to generate clock pulses for the DS1620, which in turn transmits the 9-bit temperature value to the MCU. We call this single-wire type of communication *half duplex*. Duplex indicates 2-way communications and the word "half" lets us know communications take place only in one direction at a time.

The `gettemp` object serves a specific purpose; it obtains a 9-bit temperature value. To use other commands that can get 8- or 9-bit data from a DS1620 sensor IC, we need a different object. I have added the object, `GetDS1620_data`, to the ds1620_JT2.spin program. Use this object when you must send a command to a one-wire device and expect to receive data from it. The `GetDS1620_data` object requires two pieces of information, as shown next, and it returns with the value requested:

```
GetDS1620_data (command_goes_here, number_of_bits_to_receive)
```

The example that follows will read a 9-bit temperature and then read the 8-bit Configuration Register data (this example is not a complete program, though):

```
CON
  Read_Temp       =     $AA   /Command to read 9-bit temperature
  Read_Config     =     $AC   /Command to read configuration byte
  RCV_Nine        =     9     /Receive nine bits of data
  RCV_Eight       =     8     /Receive eight bits of data
```

```
VAR
  word      TempC       /Variable for 9-bit temperature
  byte      Config_Byte /Variable for 8-bit configuration data

PUB

TempC := GetDS1620_data (Read_Temp, RCV_Nine)

Config_Byte := GetDS1620_data (Read_Config, RCV_Eight)
```

The `TempC` variable will hold the 9-bit temperature information and the `Config_Byte` will hold the 8-bit contents of the sensor's configuration register.

The timing diagram in **Figure 14.3** illustrates the transmission of a Read Temperature command ($AA) from an MCU, followed by transmission of the 9-bit temperature value, $030 from the sensor to the MCU. These communications use the same DQ connection. The MCU always supplied the needed number of pulses on the CLK connection.

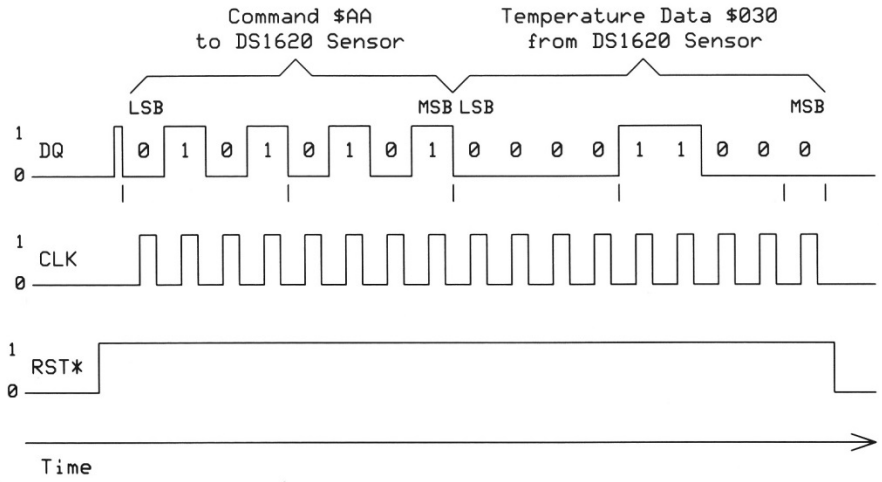


Figure 14.3.
After the Propeller sends a Read Temperature command ($AA), the DS1620 sensor responds with a 9-bit temperature value. Because the Propeller MCU can quickly switch a pin from a digital output to a digital input, the MCU needs only one pin to transmit or receive information.

Step 1.

A DS1620 sensor IC can act like a thermostat; a device that controls heating and cooling elements to keep a temperature within a small range. The 9-bit TH and TL read-write registers hold the temperature for the high and low limits, respectfully. The term read-write, or R/W, means you can put a value in the register and read the value from the register. Four commands control these operations:

```
Write TH  =  $01        Read TH   =  $A1
Write TL  =  $02        Read TL   =  $A2
```

When a temperature equals or exceeds the TH value, the DS1620 THIGH output (pin 7) goes from a logic-0 to a logic-1. This output returns to the logic-0 state only when the sensor temperature goes below the TH value.

Likewise, if the temperature equals or drops below the value in the TL register, the TLOW output (pin 6) goes from a logic-0 to a logic-1. This output returns to a logic-0 state only when the temperature climbs above the TL value.

The TH and TL settings let the DS1620 sensor IC operate like a mechanical thermostat. In cold weather, when the temperature goes below that set in the TL register, the DS1620 TLOW output could control a heater.

The DS1620 has a third output, TCOM (pin 5). This output goes from a logic-0 to a logic-1 when the temperature reaches or exceeds the TH-register value. But this output returns to a logic-0 only when the temperature *falls below* the TL value, as shown in **Figure 14.4**. You might use the TCOM output to control a fan in electronic equipment. As soon as the equipment temperature reaches TH, the fan turns on and it stays on until the temperature goes below TL.

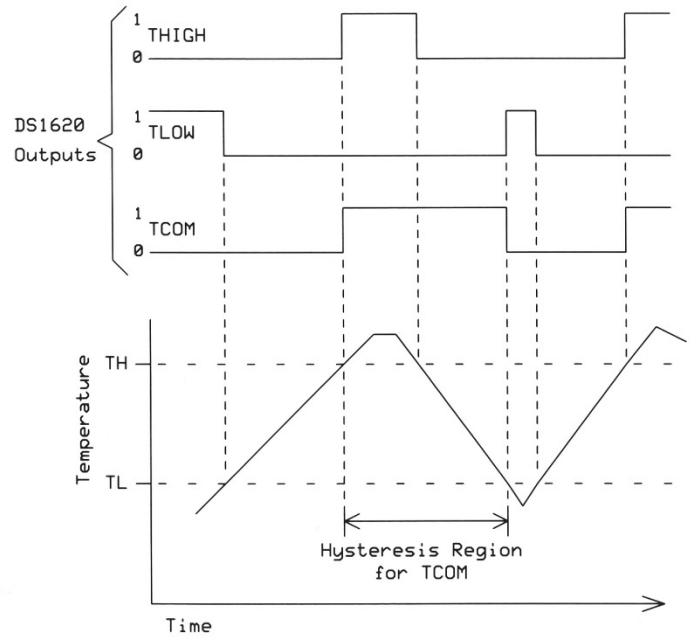


Figure 14.4.
Behavior of the THIGH, TLOW, and TCOM outputs as they react to a changing temperature. You will learn about hysteresis in this experiment.

How would you use the information in the Send a Command and Data section of this experiment to set the 9-bit TH and TL values? Find a solution in the Answers section at the end of this experiment.

Step 2.

Ensure you have a DS1620 sensor IC connected to your Propeller P8X32A MCU board as shown in **Figure 14.5**. The Propeller board must connect to your PC via a USB cable. If you have other components or displays connected to the breadboard or to the MCU board, please disconnect them now.

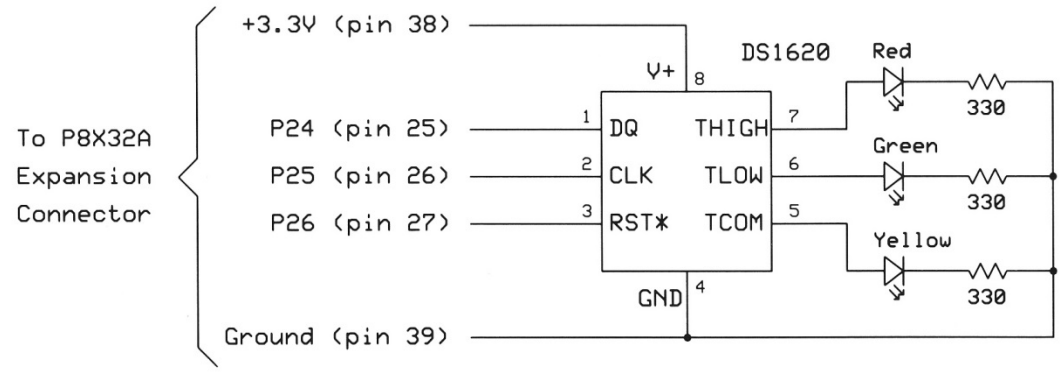


Figure 14.5.
Circuit diagram for connections between a DS1620 temperature-sensor IC and a Propeller P8X32A MCU board. This experiment also requires three LEDs that indicate alarm conditions.

Open the Experiment 14 software folder and ensure you have the following files in it.

Program 14.1.spin	ds1620_JT2.spin	timing.spin
Program 14.2.spin	FullDuplexSerial.spin	shiftio.spin

Program 14.1 lets you load the TH and TL registers with any 9-bit value you choose. Keep in mind the DS1620 provides accurate temperature measurements only between +125 °C and -55 °C, or from $0000 up to $00FA and from $01FE down to $0192. My ambient room temperature measured about 23 °C ($002E), so I set an upper limit of 30 °C ($003C) and a lower limit of 15 °C ($001E). The DS1620 measures temperature in half-degree increments, so 15 °C equals 30 half-degree steps, and 30_{10} = $001E hex.

Choose 9-bit hexadecimal values for the TH and TL registers. Make the TH value a few degrees higher than your ambient temperature, and make the TL value a few degrees lower. Put your values in **Program 14.1** to replace the ones already in the code:

```
TH_value   :=  $003C    'Substitute your TH value here
TL_value   :=  $001E    'Substitute your TL value here
```

Program 14.1.

```
{{
''*********************************************
''*  Program 14.1 Load DS1620 TH and TL registers
''*  Author: Jon Titus 11-13-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Demonstration of the DS1620 as a thermostat
''*  Three LEDs indicate over-temperature,
''*  under-temperature, and range "excursion"
''*********************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

    Write_TH        = $01           'Write 9-bit TH register
    Write_TL        = $02           'Write 9-bit TL register
    Read_TH         = $A1           'Read from 9-bit TH register
    Read_TL         = $A2           'Read from 9-bit TL register
```

```
    Nine_bit_data   = 17            '17 bits, command plus data
    Eight_bit_data  = 16            '16 bits, command plus data
OBJ

  temp      :  "DS1620_JT2"         'ds1620_JT2,spin file
  delay     :  "timing"             'timing.spin file
  pst       :  "FullDuplexSerial"   'Serial library for testing

VAR

  byte PropPin_DQ               'Variable for DS1620 DQ
  byte PropPin_CLK              'Variable for DS1620 CLK
  byte PropPin_RST              'Variable for DS1620 RST*
  byte PropPin_SerOut           'Variable for P8X32A pin P30
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for P8X32A serial mode
  word SerPort_BaudRate         'Variable for serial bit rate
  long TH_value                 'TH register value
  long TL_value                 'TL register value

PUB main | tc                         'tc is a local variable

 PropPin_DQ        := 24        'P24 for DS1620 DQ pin1
 PropPin_CLK       := 25        'P25 for DS1620 CLK pin 2
 PropPin_RST       := 26        'P26 for DS1620 RST* pin 3

 PropPin_SerOut    := 30        'P30 for serial transmit
 PropPin_SerIn     := 31        'P31 for serial receive
 PropPin_SerMode   := 0         'Invert RX mode
 SerPort_BaudRate  := 9600      '9600 bps baud rate

 'Put TH and TL values here:
 TH_value          := $003C     'Upper-limit temp, 9 bits
 TL_value          := $001E     'Lower-limit temp, 9 bits

'Configure and start DS1620 temperature conversions
    temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Configure Propeller serial port to communicate with host PC via
'Parallax Serial Terminal (PST)
    pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode,
SerPort_BaudRate)

'Write data to TH and TL registers
    temp.SendDS1620(Write_TH, TH_value, Nine_bit_data)
    temp.SendDS1620(Write_TL, TL_value, Nine_bit_data)

'Loop to get temperature and display it as a hex value every 'four seconds.
    repeat
        delay.pause1ms(4000)    '4 second delay
        tc := temp.gettempc     'Get temp via DS1620_JT2.spin
        pst.hex(tc, 3)          'Display 3-digit hex temperature
        pst.Tx(13)              'Force a new line on PST

'  –  – -End –  – -
```

Step 3.

Run the program. It will display temperatures as hexadecimal values in the Parallax Serial Terminal (PST) window with an update every four seconds. (Press F12 on your keyboard to start the PST software.)

Use a hair dryer, or other flameless heat source to slowly warm the DS1620 sensor IC. Watch the LEDs. Do you see any LEDs turn on as the temperature rises? The red LED should turn on when the DS1620 temperature reaches the TH value you put in **Program 14.1**.

Let the DS1620 IC cool and watch the LEDs again. Does the red LED turn off? It should turn off as soon as the temperature of the DS1620 sensor IC drops below the TH value you set. After you experiment with the DS1620 a bit more, remove the 3.3-volt connection between your breadboard and the Propeller board and reconnect it. Or briefly remove the USB connector from the Propeller board and then reconnect it. This action resets the DS1620. Restart **Program 14.1**.

Read this paragraph and then go through the instructions in this step once more. When the red LED turns on, note the last two or three temperatures shown in the PST window. Do these temperatures exceed your TH value? You'll probably find several temperatures that do. What temperatures do you observe when the red LED turns off? Again, you'll probably see several temperatures below the TH value. Remember you can click the Disable button in the PST window to pause the display so you can write down values.

Step 4.

Before you go any farther, try to answer the following questions. What happens to the yellow LED when the DS1620 sensor IC temperature exceeds the TH value? What happens to this LED when the red LED turns off as the sensor's temperature drops below the TH value? Does the response of the yellow LED match the behavior illustrated in **Figure 14.4**? Find answers at the end of this experiment.

Step 5.

Briefly turn off power to your breadboard to reset the DS1620 IC again. Restart **Program 14.1**. Gently warm the DS1620 IC until the red and yellow LEDs turn on.

Now use an ice cube in two plastic bags, or coolant spray, to cool the DS1620 sensor IC below the temperature you set in the TL register. What happens to the green LED? The green LED should turn on as soon as the sensor temperature equals or goes below the TL value you set. Did you see the hex values in the PST window dip below your TL value? If you set the TL value too far below room temperature, your fingers might get cold holding an ice cube in place!

Experiment with different values for the TH and TL registers. Then note the effects on the three LEDs during heating and cooling.

Step 6.

The yellow LED should turn on when the red LED turns on at temperature TH, and the yellow LED should turn off only when the green LED turns on at temperature TL. The operation of the yellow LED, and thus the DS1620 TCOM output exhibits what engineers call *hysteresis*. In short, a system with hysteresis exhibits "memory" in the sense that it "remembers" past conditions and takes action accordingly. **Figure 14.6** graphs the hysteresis effect for the TCOM output

Temperature can increase or decrease along line A, but when it reaches TH, the TCOM output immediately switches to a logic-1 (line B). The temperature can then vary along line C, but as soon at it drops to TL, the TCOM output switches to a logic-0 (line D). The arrows on lines A through D indicate the allowed temperature "paths." An increasing temperature always follows the path A-B-C. A decreasing temperature always follows the path C-D-A.

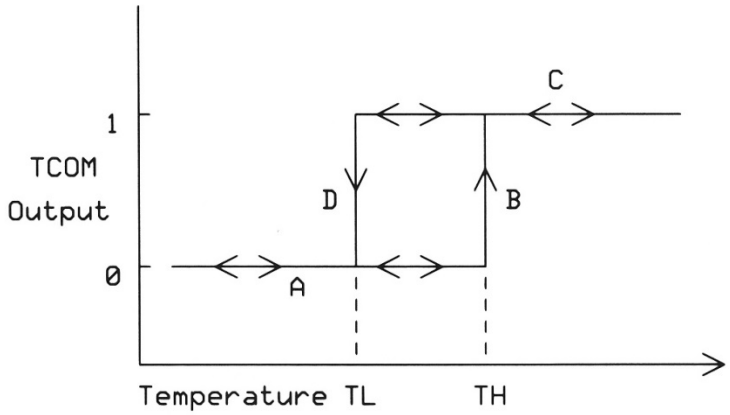


Figure 14.6.
Hysteresis diagram for the TCOM output on a DS1620 sensor IC vs. temperature.

Suppose engineers have an electronic assembly that can get hot when turned on. To avoid thermal problems, they add a DS1620 sensor IC to the assembly's circuit and use the TCOM output to control a fan. They set the TH register to the equivalent of 40 °C (104 °F) and the TL register to the equivalent of 25 °C (77 °F). The fan turns on when the equipment temperature exceeds 40 °C and turns off when the temperature drops to 25 °C.

Step 7.

While working on another project, an engineer decided to use the THIGH output instead of the TCOM output to control a fan to cool electronic equipment. She sets TH to the equivalent of 40 °C (104 °F) and tests her equipment. What happens?

As soon as the temperature reaches the TH value (40 °C), the fan turns on and cools the equipment. But the fan turns off when the temperature drops half a degree to 39.5 °C. Then the temperature rises half a degree to 40 °C and the fan turns on again. The small temperature difference between the fan-on and fan-off conditions causes the fan to cycle on or off for short periods, which will wear it out quickly. Also, the constant on-off cycles would annoy anyone near the fan. Why would the fan turn off at 39.5 °C? The DS1620 works with half-degree values.

On the other hand, the hysteresis effect on the TCOM output introduces a longer delay between fan-on and fan-off times because the temperature must cycle between TH and TL. You can make this temperature span (TH minus TL) as wide or as narrow as necessary (within the range of the DS1620 sensor, of course).

Use the THIGH and TLOW outputs as alarms to indicate a temperature has exceeded or fallen below the TH and TL settings. Take advantage of the TCOM hysteresis behavior to control cooling equipment. In a different situation could you use the TCOM output to control a heater to keep a liquid within a given temperature range?

Step 8.

The THIGH and TLOW outputs can turn on LEDs, activate buzzers, or trigger other warning devices. But changes at these outputs do not automatically alert an MCU to a changed condition. The DS1620 sensor IC does provides this type of information, however, in the Configuration Register **Figure 14.7**. Bit D6 represents the state of the TH Flag (THF), and thus the THIGH output. Likewise, bit D5 holds for the TL Flag (TLF) and lets an MCU know the state of the TLOW output. (For information about the other status and configuration bits, please refer to the DS1620 data sheet. See the Reference section at the end of this experiment.)

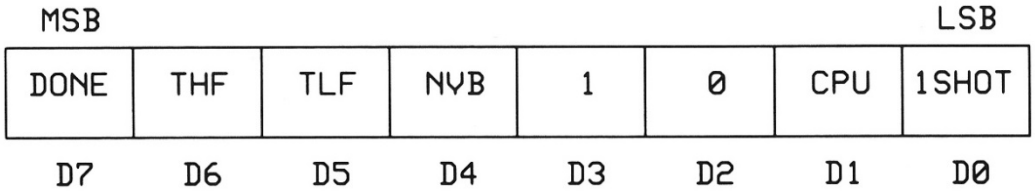


Figure 14.7.
Configuration Register bits for a DS1620 temperature-sensor IC.

In the introduction to this experiment, the "Send a Command and Receive Data" section showed how to use the `GetDS1620_data` object in the DS1620_JT2.spin file to read data from the Configuration Register:

```
Config_Byte := GetDS1620_data (Read_Config, RCV_Eight)
```

The THIGH and TLOW outputs change their state as temperatures change. The Configuration Register TH and TL flags "latch." That means the sensor IC sets the Temperature High Flag bit (THF) to logic-1 when the temperature equals or exceeds the TH value. This bit *remains set to a logic-1* until you reset it by writing a logic-0 into the THF bit (D6) in the Configuration Register.

The TLF bit, D5, works the same way. It *remains set at a logic-1* when the temperature equals or goes below the TL value. The THF and TLF bits let software determine if the temperature of the DS1620 has *ever* reached the TH value, or has *ever* fallen to the TL value since the sensor last received power. If you remove and reapply power to the DS1620, the TLF and THF bits reset to logic-0. But we don't want to do that whenever the flag bits need a reset. The IC designers let an MCU read the THF and TLF bits and then reset them with another communication.

Step 9.

Program 14.2 examines the THF and TLF bits in the Configuration Register and reports their state in messages you can see in the PST window. If the yellow light in your circuit is on now, briefly disconnect power to the DS1620 to reset the TCOM output.

Program 14.2.

```
{{
''*********************************************
''*  Program 14.2 Load DS1620 TH and TL registers
''*  Author: Jon Titus 11-13-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Demonstration of the DS1620 THF and TLF
''*  flags. Messages printer when flags set. After
''*  printing messages, flags get reset.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

    'DS1620 commands
    Read_Temp         = $AA         ' read temperature
    Read_Counter      = $A0         ' read counter
    Read_Slope        = $A9         ' read slope
    Start_Convert     = $EE         ' start conversion
```

```
    Stop_Convert      = $22         ' stop conversion
    Write_Config      = $0C         ' write config register
    Read_Config       = $AC         ' read config register
    Write_TH          = $01         ' write command for TH reg
    Write_TL          = $02         ' write command for TL reg
    Read_TH           = $A1         ' read from 9-bit TH reg
    Read_TL           = $A2         ' read from 9-bit TL reg

    Nine_bit_data     = 17          '17 bits, command plus data
    Eight_bit_data    = 16          '16 bits, command plus data

OBJ
  temp      :  "DS1620_JT3"         'ds1620_JT2,spin file
  delay     :  "timing"             'timing.spin file
  pst       :  "FullDuplexSerial"   'Serial library for testing

VAR
  byte PropPin_DQ               'Variable for DS1620 DQ
  byte PropPin_CLK              'Variable for DS1620 CLK
  byte PropPin_RST              'Variable for DS1620 RST*
  byte PropPin_SerOut           'Variable for P8X32A pin P30
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for P8X32A serial mode
  word SerPort_BaudRate         'Variable for serial bit rate
  long TH_value                 'TH register value
  long TL_value                 'TL register value
  byte DS1620_flags             'Variable for DS1620 Config Reg
  byte Config_Reset                   'Config-reset value

PUB main | tc

 PropPin_DQ        := 24        'P24: DS1620 DQ pin1
 PropPin_CLK       := 25        'P25: DS1620 CLK pin 2
 PropPin_RST       := 26        'P26  DS1620 RST* pin 3

 PropPin_SerOut    := 30        'P30 for serial transmit
 PropPin_SerIn     := 31        'P31 for serial receive
 PropPin_SerMode   := 0         'Invert RX mode
 SerPort_BaudRate  := 9600      '9600 bps baud rate

 'Put TH and TL values here:
 TH_value          := $003C     'Upper-limit temp, 9 bits
 TL_value          := $001E     'Lower-limit temp, 9 bits
 Config_Reset      := %00000010 'Config reg reset bits

'Configure and start DS1620 temperature conversions
    temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Configure Propeller serial port to communicate with host PC via
'Parallax Serial Terminal (PST)
    pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode,
SerPort_BaudRate)

'Write data to TH and TL registers
    temp.SendDS1620(Write_TH, TH_value, Nine_bit_data)
    temp.SendDS1620(Write_TL, TL_value, Nine_bit_data)
```

```
'Loop to get temperature and display it as a hex value every 'four seconds.
    repeat
        delay.pause1ms(4000)        '4 second delay
        tc := temp.gettempc         'Get temp via DS1620_JT2.spin
        pst.hex(tc, 3)              'Display 3-digit hex temp
        pst.str(String("   "))      'insert some spaces

       'Read Config Register
        DS1620_flags := temp.GetDS1620_data (Read_Config, 8)
       DS1620_flags &= %01100000    'Mask all but D6 and D5

        if DS1620_flags > 32        'If D6 = 1 (64)
            DS1620_flags -= 64            'Subtract 64
            pst.str(String("THF Set"))    'Print message

        if DS1620_flags == 32       'If D5 = 1 (32)
            pst.str(String("TLF Set"))    'Print message

        pst.Tx(13)                              'New line for PST

        'Reset Configuration Register
        temp.SendDS1620(Write_Config, Config_Reset, Eight_bit_data)

'  –  – -End –  – -
```

Run **Program 14.2** and gently warm the DS1620 sensor IC until the red LED turns on. Turn off your heat source. Do you see the message "THF Set" in the PST window?

The red and yellow LEDs should behave as shown earlier. The "THF Set" message appears in the PST window when the sensor temperature reaches your TH value. Let the sensor cool. When the red LED turns off, you should no longer see the "THF Set" message next to the new hexadecimal temperature values. Although the flags latch, or maintain their state, **Program 14.2** resets them after each temperature measurement.

Cool the sensor with a bagged ice cube or with coolant spray. What happens now? You should see the green LED turn on and the "TLF Set" message appears next to each temperature measurement. Let the sensor warm until the green LED turns off. At this time, you should no longer see the message "TLF Set" next to new temperatures in the PST window.

Step 10.

The section of code shown next performs the bit-testing operations and displays the results. Earlier in the program the code set: `Config_Reset := %00000010.`

```
    repeat
        delay.pause1ms(4000)   '4 second delay
        tc := temp.gettempc    'Get temp via DS1620_JT2.spin
        pst.hex(tc, 3)         'Display 3-digit hex temperature
        pst.str(String("   ")) 'insert some spaces

        'Read Config Register
        DS1620_flags := temp.GetDS1620_data (Read_Config, RCV_Eight)
        DS1620_flags &= %01100000       'Mask out all but D6 and D5

        if DS1620_flags > 32            'If D6 = 1 (64)
            DS1620_flags -= 64          'Subtract 64, D6 goes to 0
            pst.str(String("THF Set"))  'Print message
```

```
        if DS1620_flags == 32           'If D5 = 1 (32)
            pst.str(String("TLF Set"))  'Print message
        pst.Tx(13)                      'New line for PST

        'Reset Configuration Register
        temp.SendDS1620(Write_Config, Config_Reset, Eight_bit_data)
```

The following two statements first use the `GetDS1620_data` object to send the sensor the `Read_Config` command ($AC) defined earlier in the program:

```
DS1620_flags := temp.GetDS1620_data (Read_Config, RCV_Eight)
DS1620_flags &= %01100000  'Mask all but D6 and D5 bits
```

The second statement performs a bitwise AND between the `DS1620_flags` data and the mask 01100000_2. The "shortened" statement:

```
DS1620_flags &= %01100000
```

performs the equivalent of:

```
DS1620_flags := %01100000 & DS1620_flags
```

The bitwise AND operation ensures DS1620_flags contains only logic-1 bits for the THF and TLF bits (assuming they are set to a logic-1). All other bits get forced to a logic-0. The `DS1620_flags` variable may have only one of four states:

00000000_2 (0_{10})	No temperature flags set
00100000_2 (32_{10})	TLF set
01000000_2 (64_{10})	THF set
01100000_2 (96_{10})	Both THF and TLF set

The `if` sections of **Program 14.2** show how the software evaluates the state of the D6 and D5 bits. The first `if` statement:

```
if DS1620_flags > 32           'If D6 = 1 (64)
    DS1620_flags -= 64         'Subtract 64
    pst.str(String("THF Set")) 'Print message
```

determines whether the `DS1620_flags` value exceeds 32. In other words, does bit D6 equal a 1, which would give `DS1620_flags` a value of 64 or 96. If so, the DS1620 sensor IC has set the THF flag to a logic-1 and the program displays "THF Set" in the PST window.

Note when the value of `DS1620_flags` exceed 32, this section of the `if` code subtracts 64 from `DS1620_flags`, which leaves a value of either 0 or 32:

```
  01000000₂ (64₁₀)          01100000₂ (96₁₀)
- 01000000₂ (64₁₀)        - 01000000₂ (64₁₀)
  00000000₂ (0₁₀)           00100000₂ (32₁₀)
```

In this way, the program "isolates" the TFL flag bit in the `DS1620_flags` value. Then the program uses another `if` statement to test the resulting value. If `DS1620_flags` equals 32, the program prints TLF Set. Otherwise, it continues in the `repeat` loop and makes another temperature measurement.

```
if DS1620_flags == 32          'If D5 = 1 (32)
    pst.str(String("TLF Set")) 'Print message
```

At the end of the program, the statement:

```
temp.SendDS1620(Write_Config, Config_Reset, Eight_bit_data)
```

resets the THF and TLF flags without disturbing the sensor's operation.

IMPORTANT: The Maxim data sheet for the DS1620 explains the use of the Status Register bits D1 and D0 that control how the sensor operates with an MCU. This experiment uses CPU (central-processing unit) mode, probably better named MCU mode. To use this mode, D1 = 1 and D0 = 0 get written into the Status Register (see **Figure 14.7**). The `Config_Reset := %00000010` statement shows these bits ready to go into the Status Register.

The CPU and 1SHOT bits get written into the DS1620's Electrically Erasable Programmable Read-Only Memory (EEPROM), so even if the sensor IC loses power, it will power-up in the mode last selected. The "nonvolatile" EEPROM memory does not lose any data when it loses power.

In effect, each time our software resets the THF and TLF bits, the `Config_Reset` value rewrites the CPU and 1SHOT bits into EEPROM. The data sheet explains the EEPROM can handle a minimum of 50,000 write operations. **Program 14.2** sends the `Config_Reset` value to the sensor – and thus into the EEPROM – every 4 seconds. At this rate, the EEPROM reaches 50,000 write operations after about 55 hours, or just over two days. You could decrease the measurement period to several minutes, but that delays the response to an over- or under-temperature condition. If you need the THF and TLF information, I suggest you NOT reset the internal Configuration Register and connect the THIGH and TLOW pins directly to input pins on an MCU. Sampling these pins provides the same information as the internal flag bits and they don't require any reset operations.

Step 11.

You might have noticed that clearing the THF and TLF flags in the Configuration Register did not affect the TCOM output that controls the yellow LED. This output operates independent of the THF and TLF settings. The TH and TL outputs also operate independently, although the flags let you determine their state; logic-0 or logic-1 via software.

Step 12.

You may disconnect the DS1620 sensor from your Propeller P8X32A board.

Reference

"DS1620 Digital Thermometer and Thermostat" data sheet, Maxim Integrated.
http://datasheets.maximintegrated.com/en/ds/DS1620.pdf.

Answers

Experiment 14, Step 1:

Use the following statements in a complete program to set the TH and TL values.

```
OBJ
      temp  :  "ds1620_JT2"
```

```
CON
    Write_TH       = $01    'Write command for 9-bit TH register
    Write_TL       = $02    'Write command for 9-bit TL register
    Nine_bit_data  = 17     '17 bits for command plus data

VAR
  long TH_value              'TH register value
  long TL_value              'TL register value

PUB main
 'Put TH and TL values here:
 TH_value     := $003C     'Your upper-limit temperature, 9 bits
 TL_value     := $001E     'Your lower-limit temperature, 9 bits

'Configure and start DS1620 temperature conversions
    temp.start(PropPin_DQ, PropPin_CLK, PropPin_RST)

'Write data to TH and TL registers
    temp.SendDS1620(Write_TH, TH_value, Nine_bit_data)
    temp.SendDS1620(Write_TL, TL_value, Nine_bit_data)
```

You might have forgotten to include the information shown in boldface type. You should always initialize an external IC before you try to communicate with it.

Experiment 14, Step 4:

What happens to the yellow LED when the DS1620 sensor IC temperature exceeds the TH value? If not already lit, the yellow LED turns on.

What happens to this LED when the red LED turns off as the sensor's temperature drops below the TH value? The yellow LED should remain on.

Does the response of the yellow LED match what you saw in **Figure 14.4**? It should. The yellow LED will not turn off until the DS1620 sensor temperature reaches the TL value.

Experiment No. 15 – How To Use Digital-to-Analog Converters

Abstract

A digital-to-analog converter (DAC) – usually offered as a complete integrated circuit – lets a microcontroller use digital values to create a proportional analog voltage. With a bit of software, experimenters and engineers can create almost an infinite range of signals for many purposes. A DAC can provide a signal to control motor speed, drive an x-y plotter to chart experiment results, deliver signals to medical devices, generate sounds and tones, and so on. And DACs take digital data in MP3 files and create audio signals for portable players and home-entertainment equipment. This versatile component also serves as a key part of analog-to-digital converters, covered in the next experiment.

Keywords

Digital-to-analog-converter, DAC, LTC1450L, voltage output, reference voltage, discrete voltage steps, ramp, triangle wave, continuous signal, discrete signal

Requirements

(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - LTC1450L digital-to-analog converter, 24-pin DIP, Linear Technology
(1) - Digital voltmeter or digital multimeter (DMM)
(1) - USB cable
(1) - 1000-ohm, 1/4-watt resistor, 5% (brown-black-red)
(13) - 10-kohm, 1/4-watt resistors, 5% (brown-black-orange)
(1) - Power supply for 3.3 volts (optional)

Introduction

In Experiments 13 and 14 you used a DS1620 sensor IC that provided discrete temperature values; 15, 15.5, 16, 16.5 and 17 °C, and so on. Although the surrounding temperature could change by small fractions of a degree, the sensor can only provide values in half-degree steps. Temperature undergoes what we call a *continuous* change. Here the word continuous doesn't mean the temperature constantly changes. In terms of signals and measurements, continuously means a "smooth" change from one temperature to another. If you look at a liquid thermometer you see a smooth change in temperature from, say, 22 to 25 °C. The liquid moves through every temperature between those two. On the other hand, a digital thermometer "jumps" from, 22.0 to 22.5 °C, then to 23 °C... The thermometer shows a continuous change – no "missing" temperatures. The digital sensor detects changes only in half-degree steps.

Think of the temperature changes much like positions on a ramp. You can place an object anywhere along the ramp's slope and move it up or down any amount you choose. In effect, it can have an infinite number of locations, or heights above ground. Now consider a set of stairs. The object can sit only on one of the stair treads. You cannot place an object between steps. The diagram in **Figure 15.1** illustrates the difference between positions on a *continuous* ramp and the *discrete* positions on the stairs. We live in a world of continuous changes, or analog signals. Digital computers, though, work with discrete information that comprises 1s and 0s.

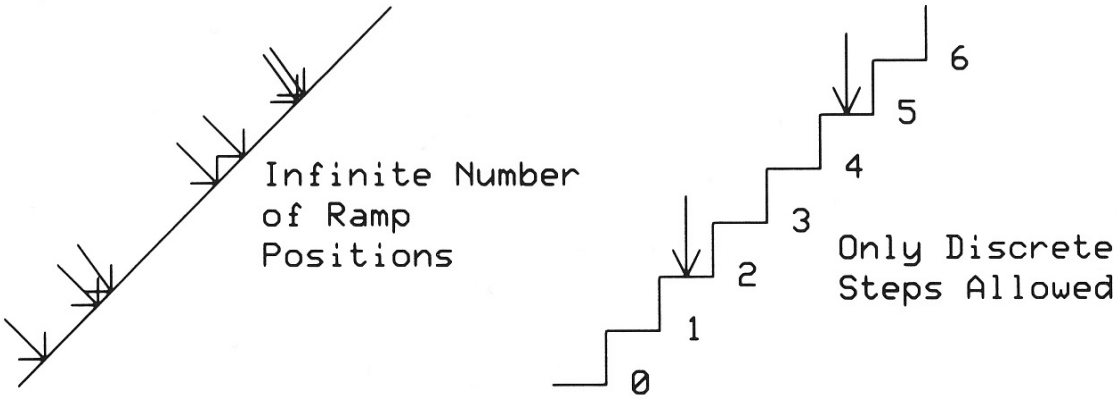


Figure 15.1.
You can put an object anywhere along an angled ramp. A set of equidistant stairs offers fixed treads for an object. You cannot place something between Step 2 and Step 3, for example. There's no in-between position.

To make the transition between the realms of analog signals and discrete values, engineers use two types of circuits; analog-to-digital converters (ADCs) that measure analog signals, and digital-to-analog converters (DACs) that create analog signals. In this experiment you will learn how a DAC works.

The voltage produced by a DAC can control the speed of a motor, the amount of energy generated by an electric heater, the brightness of LEDs, the volume of audio signals, and other devices that depend on a control voltage. As you learn more about electronics you will find other uses for a voltage signal produced under control of an MCU.

How Does a Digital-to-Analog Converter Work?

A digital-to-analog converter accepts a binary value of *n* bits and produces a proportional voltage or current in 2^n increments. A 4-bit DAC, for example, produces 2^4 or 16 voltages with equal voltage steps between them. Those voltages correspond to the 4-bit binary values from 0000 to 1111_2 . At this point, though, we don't know the value of those 16 voltages. Commercial DAC ICs provide an internal voltage reference or require an external reference voltage to establish a voltage-output range. A Linear Technology LTC1450L DAC IC lets circuit designers select either an internal 2.50- or 1.22-volt reference.

For the hypothetical 4-bit DAC, assume a 2.5-volt reference, which corresponds to a maximum output voltage of 2.5 volts. This DAC will produce 16 (2^4) different voltages, with 15 (2^4 - 1) increments between them. Thus, by simple math: 2.5 volts divided into 15 steps:

$$2.5\,V\ /\ 15\,steps\ =\ 0.1666\,volts/step\ (V/step)$$

Thus, the binary value 0000_2 produces a 0.00-volt output, the value 0001_2 produces 0.166 (0.17) volts, the value 0010_2 produces 0.333 (0.33) volts, and so on. (I rounded results to two decimal places.) Given the voltage per step, fill in the voltage for each binary value shown in **Table 15.1**.

Table 15.1. Binary inputs and voltage outputs for a 4-bit DAC with a 2.5-volt reference.

DAC Binary Input	DAC Voltage
0000	0.00
0001	0.17

0010	0.33
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

So far, a DAC seems like a magic black box. To help you better understand what a DAC does you will create a 4-bit DAC circuit on a solderless breadboard and test it.

Step 1.

Engineers have used many techniques to create a DAC circuit. The most popular of these uses a set of resistors in an R-2R "ladder" arrangement to produce a current or a voltage. **Figure 15.2** shows an R-2R-ladder circuit for a 4-bit DAC with a voltage output. Four switches independently connect to either ground or to a reference voltage.

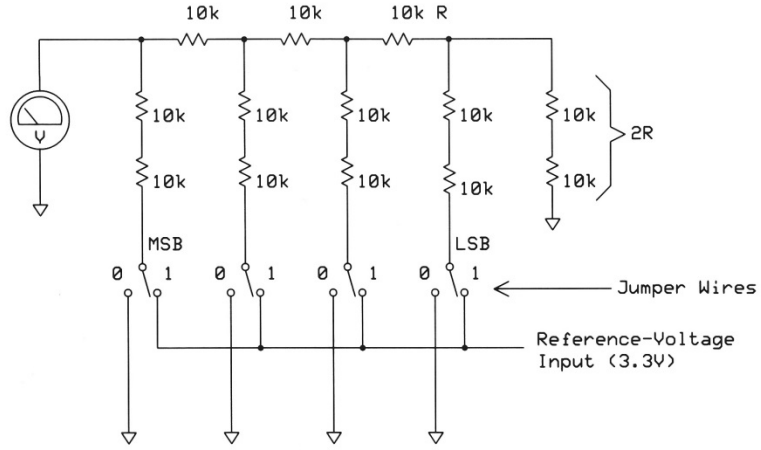


Figure 15.2.
An R-2R DAC schematic diagram. All 10-kohm resistors have a 5% tolerance. The Reference-Voltage Input connects to a 3.3-volt power supply.

In a breadboard circuit, a 10-kohm resistor represents R and two 10-kohm resistors in series provide the 2R value. Go ahead and construct this circuit on your solderless breadboard. Use jumper wires instead of switches. **Figure 15.3** shows the arrangement of resistors I used to simplify connections and make the circuit easy to check. The left ends of two resistors (R and 2R) connect to the voltmeter and the left end of the bottom resistor will connect to either the ground or the +3.3V bus. Refer to **Figure 15.2** for all component connections.

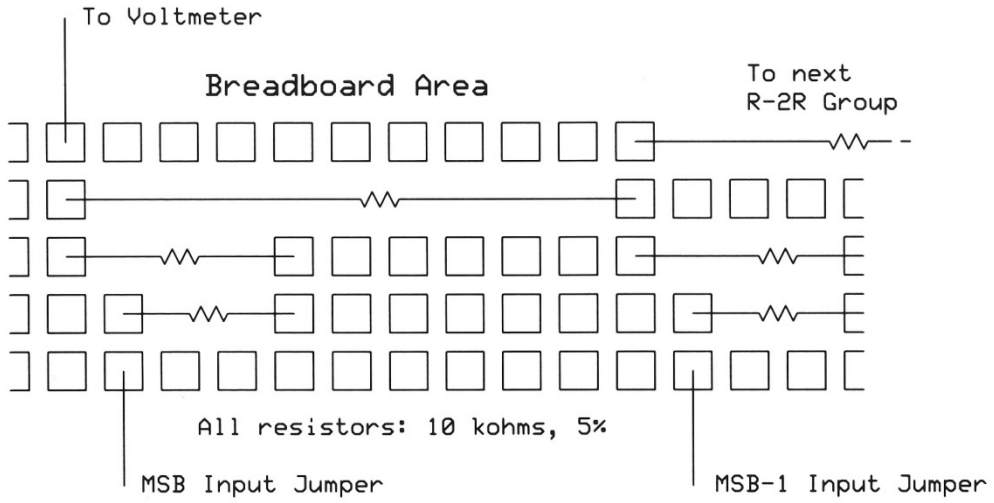


Figure 15.3.
The author's arrangement of resistors on a breadboard shows the connections for the MSB section of the R-2R DAC circuit and the connection to an adjoining section. The vertical columns of five breadboard receptacles connect to one another. For the sake of clarity, this diagram does not show all receptacles in a breadboard.

To start, connect the free end of each jumper to ground. If you have an adjustable power supply, set it as close as possible to 3.3 volts and use it as your reference. If you don't have a separate power supply, use two 1.5V batteries or connect to the 3.3-volt output (pin 38 on the I/O connector) on the P8X32A Propeller board attached to your PC. You also need a ground connection between your power supply or P8X32A board and the DAC circuit. To simplify jumper changes, use the power and ground buses on the breadboard for, well, power and ground.

Connect your voltmeter to the breadboard as shown in **Figure 15.2**. Switch the meter to the *voltage* range that comes closest to and *exceeds* 3.3 volts so you can measure the DAC-circuit output. Ensure you have the leads properly connected to the DVM to make voltage measurements. Recheck your wiring.

Step 2.

I oriented my breadboard so the left-most jumper controls the most-significant bit (MSB). This arrangement simplified keeping track of the connections for each binary value.

Turn on power to your circuit and measure the power-bus voltage.

Power-supply voltage equals: _____ volts.

Given this voltage, calculate the voltage *per step* for your 4-bit DAC circuit. Remember, you will have 16 voltages, but only 15 steps between them.

Voltage per step equals: _____ volts / step.

Step 3.

Disconnect the voltmeter from the power supply and connect it between ground and the R-2R output as shown in **Figures 15.2** and **15.3**. Ensure all four jumpers connect to ground. What voltage do you measure now? With all jumpers connected to ground and power applied to your circuit, you should see a 0-volt output (or close to it).

Now you will measure the voltage for each of the 16 possible states that the four jumpers allow. To start, move the least-significant-bit (LSB) jumper from ground to the voltage bus on your breadboard. Measure the voltage

and enter it in **Table 15.2** on the row that corresponds to the binary value 0001_2. A logic-0 in the 4-bit values represents a jumper connected to ground for a given bit, and a logic-1 represents a jumper connected to the power-bus voltage. (Enter voltages as volts; don't mix volt and millivolt values.)

Table 15.2. Binary values and voltages you measure.

DAC Binary Input	Your Voltage	Jon's Voltage
0000		0.000
0001		0.208
0010		0.415
0011		0.624
0100		0.831
0101		1.04
0110		1.25
0111		1.45
1000		1.67
1001		1.88
1010		2.09
1011		2.30
1100		2.50
1101		2.71
1110		2.91
1111		3.12

I have included my voltages in **Table 15.2**. Although they will not match yours, you can see the 16 measurements increase in discrete steps. I used a 3.33-volt reference.

Figure 15.4 shows a plot of my voltages versus the decimal equivalent of the 4-bit binary value applied to the DAC circuit. The small diamonds represent the measured voltages. The plot reveals the DAC circuit provides a fairly straight line, which gives us a sense of the *linearity* of the DAC output. A linearity specification describes how well the DAC output voltages match the calculated voltage outputs based on a given reference voltage. In a perfect DAC (which unfortunately doesn't exist), the voltage difference between each sequential measurement would have exactly the same value. Manufacturers' specifications list a DAC's *differential nonlinearity* (DNL) to describe the differences between ideal output and actual output for a given *n*-bit input.

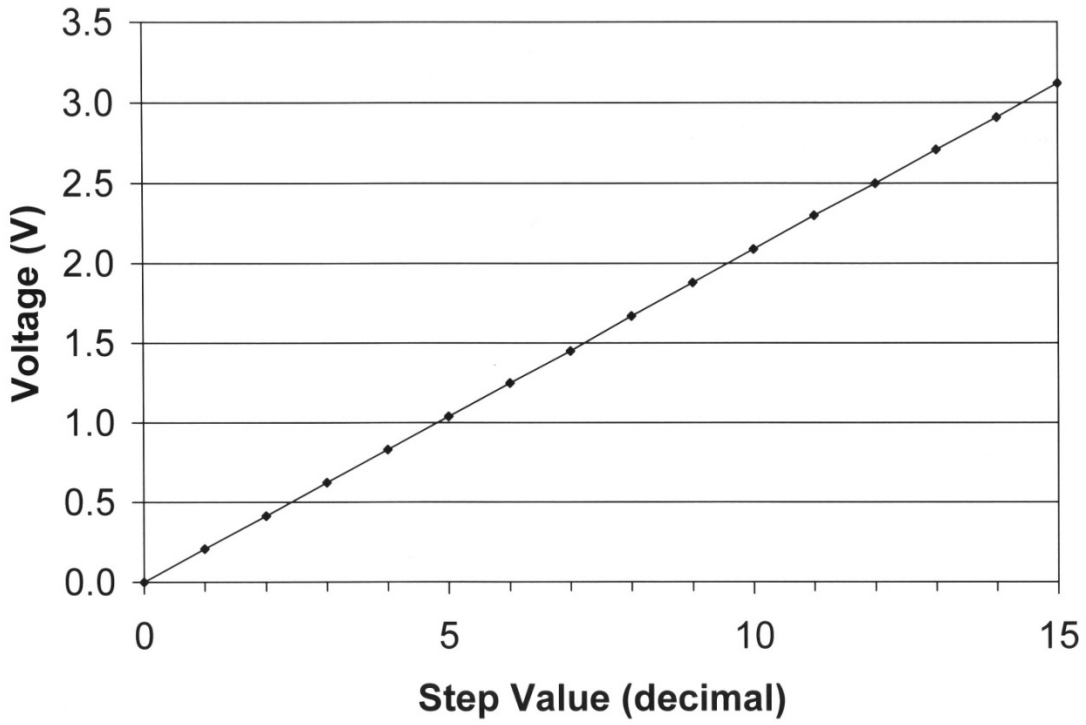


Figure 15.4.
Plot of voltages from a 4-bit R-2R-resistor ladder configured as a digital-to-analog converter vs. the corresponding binary value expressed as a decimal number. The DAC produces only the voltages represented by the small diamonds. The line between them just gives us a way to look at the linearity of the 16 DAC voltages.

The graphed results surprised me. I thought the use of resistors with a 5% tolerance might cause results to vary more from a straight line. If you plot the voltages from your measurements, do you see a fairly straight line through the voltage points? The DAC output voltages exist only at the *points* in **Figure 15.4**, not everywhere along the line. We simply use the line to indicate illustrate linearity and any variations from it for a given DAC.

The mathematics of how the R-2R-resistor circuit works go beyond the scope of this book. If you want to learn more, see the information in the Reference section at the end of this experiment.

Step 4.

To test the 4-bit DAC circuit further, you will connect it to the Parallax Propeller P8X32A board and use software to control the four inputs. A Propeller MCU output can produce either a logic-1 or a logic-0 level, which I measured as 3.0 and 0.32 volts, respectively. These voltages come close enough to those on the breadboard that we can use them instead of the jumpers to ground or power buses. Then Propeller software can control the DAC output voltage. Turn off power to the DAC circuit.

Figure 15.5 shows the necessary connections between the Propeller P8X32A board and the R-2R-resistor circuit. Remember to remove the jumpers between the DAC inputs and either the ground or the 3.3V buses. Keep your voltmeter connected to the circuit and make a ground connection between the MCU board and the resistor circuit. Please make the necessary circuit changes now and recheck the connections.

The circuit shown in **Figure 15.5** does not include a reference voltage. Instead, the Propeller's digital outputs will produce either a logic-0 (0.32 volts) or a logic-1 (3.0 volts), which will approximate ground and a 3-volt reference in the following steps. The Propeller MCU serves as the four switches and connects each 2R input as directed by software.

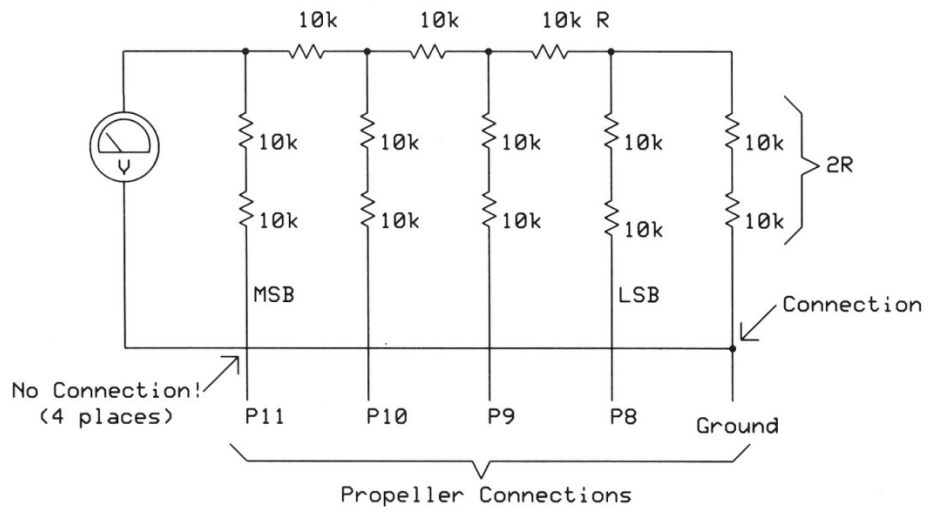


Figure 15.5.
Schematic diagram for the connections between the R-2R DAC circuit on a breadboard and the Parallax P8X32A MCU board. This circuit also requires a ground connection to the MCU board. It does not need a power connection, though.

Step 5.

Program 15.1 lets you test the DAC as the Propeller MCU controls it. Open the folder for Experiment 15 and ensure you have **Program 15.1** available. This folder also should contain the files: ShiftIO.spin and Timing.spin. Within the Propeller Tool window, open **Program 15.1** and run it. Next, start the Parallax Serial Terminal program, or press F12 on your computer keyboard. Each time you press the keyboard spacebar, the binary value shown on four Propeller-board LEDs will increase by one. Press the spacebar several times. Does the binary value increase? If not, ensure you have opened **Program 15.1** and look for any error messages within the Propeller Tool window. Correct any errors and try again.

Program 15.1.

```
{{
''*********************************************
''*  Program 15.1 4-Bit R-2R DAC Test
''*  Author: Jon Titus 11-15-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Output a 4-bit value to a 4-bit R-2R resistor-
''*  ladder DAC circuit and increment the count by
''*  one each time space bar pressed on keyboard.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

OBJ

  delay     :  "timing"             'timing.spin file
  pst       :  "FullDuplexSerial"   'Serial library for testing

VAR

  byte PropPin_SerOut           'Variable for P8X32A pin P30
```

```
  byte PropPin_SerIn            'Variable for P8X32A pin P31
  byte PropPin_SerMode          'Variable for serial comm format
  word SerPort_BaudRate         'Variable for serial bit rate
  byte DAC_value                'Variable for output to DAC
  byte PST_value                'Variable for data received data

PUB main

 PropPin_SerOut    := 30       'P30 for USB transmit
 PropPin_SerIn     := 31       'P31 for USB receive
 PropPin_SerMode   := 0        'Invert RX mode
 SerPort_BaudRate  := 9600     'Set baud rate 9600 bps

 DIRA[23..16] := $0F           'Set four LED pins as outputs
 DIRA[11..8] := %1111          'Set four outputs to DAC

'Configure Propeller serial port to communicate with host PC
 pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

 pst.RxFlush                   'Clear serial-receiver buffer
 DAC_value := %0000            'Set DAC output to zero

 'Loop accept a "space" character and increments DAC_value

 repeat
    PST_value := pst.rx         'Get serial data (if any)
        if PST_value == $20     'If data = "space bar"
            DAC_value := 1 + DAC_value  'increment DAC_value
            pst.RxFlush         'clear serial buffer

    OUTA[23..16] := DAC_value   'Output DAC_value to LEDs
    OUTA[11..8] := DAC_value    'Output DAC_value to DAC

'  –  – -End –  – -
```

As the value shown on the LEDs increases, the four output pins, P8 through P11, send the corresponding logic levels (LED on = logic-1, LED off = logic-0) to the four DAC inputs. Each time you press the spacebar, you should see the DAC output voltage increase, too. When the LED value reaches 1111_2, press the spacebar again. The value changes to 0000_2 and the voltage return to its minimum.

Press the spacebar until you have all the LEDs off (0000_2) and record your measured voltage in **Table 15.3**. Increment the binary value and record each corresponding voltage in the table.

Table 15.3. Binary values and measured voltages.

DAC Binary Input	Your Voltage	Jon's Voltage
0000		0.000
0001		0.203
0010		0.406
0011		0.609
0100		0.811
0101		1.01
0110		1.22

0111		1.42
1000		1.63
1001		1.83
1010		2.04
1011		2.24
1100		2.44
1101		2.65
1110		2.85
1111		3.05

The chart in **Figure 15.6** shows my measurements as diamonds and the straight line shows good linearity for the R-2R resistor circuit.

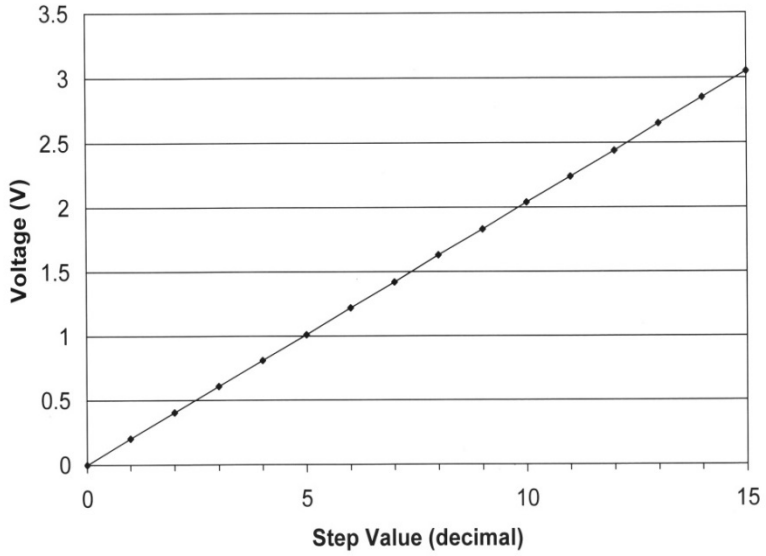


Figure 15.6.
Plot of voltages from an 4-bit R-2R DAC vs. the corresponding binary value expressed as a decimal number. This plot comes from measurements made when Propeller software controlled the DAC circuit.

Step 6.

The code in **Program 15.1** used a keyboard input to signal the Propeller MCU to increment the value for the DAC. Suppose we remove – or comment-out – the statements that wait for a keyboard value and instead let the `repeat` loop run continuously. What might you see on the voltmeter? Run **Program 15.2** to find out.

Program 15.2

```
{{
''*********************************************
''*  Program 15.2 4-Bit R-2R DAC Test
''*  Author: Jon Titus 11-15-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Output a 4-bit value to a 4-bit R-2R resistor-
''*  ladder DAC circuit and increment the count by
''*  one each time through a repeat loop.
''*********************************************
```

```
}}

CON _clkmode = xtal1 + pll16x       'Set MCU clock operation
    _xinfreq = 5_000_000            'Set for 5 MHz crystal

VAR
  byte DAC_value                    'Variable for output to DAC

PUB main

 DIRA[23..16] := $0F                'Set 4 LED pins as outputs
 DIRA[11..8] := %1111               'Set 4 outputs for DAC
 DAC_value := %0000                 'Set DAC output to zero

 'Repeat this loop forever
 repeat
    DAC_value := 1 + DAC_value      'Increment DAC_value by one

    OUTA[23..16] := DAC_value       'Output DAC_value to LEDs
    OUTA[11..8] := DAC_value        'Output DAC_value to DAC

'  -  - -End -  - -
```

My DVM read 1.52 volts. The PropScope captured the output signal from the resistor ladder, as shown in **Figure 15.7**. You can see the 16 voltage steps as the 4-bit input to the R-2R-resistor ladder increases, followed by the reset of the count to 0000_2. Then the count begins again. Given the signal in **Figure 15.7**, why would the DVM display 1.52 volts? My DVM, an old Hewlett-Packard Model 3478A instrument, showed an average voltage; or just about half of the 3-volt "reference voltage" provided by the Propeller's digital outputs.

It took the Propeller about 580 µsec (1724-Hz frequency) to complete a 16-step ramp. When I removed the LED-update statement from **Program 15.2**, a complete cycle required only 385 µsec (2597 Hz).

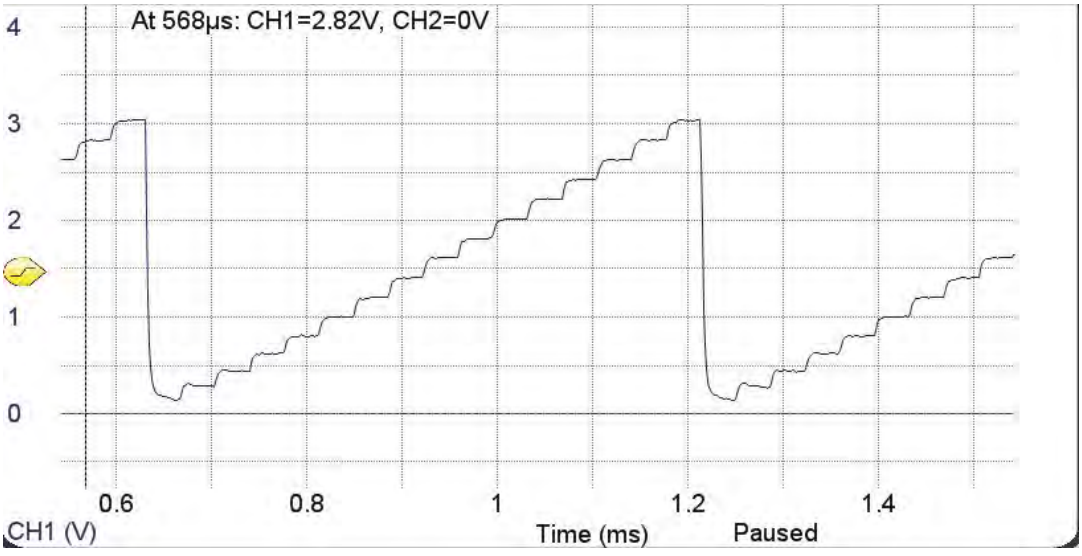


Figure 15.7.
The voltage output from the R-2R resistor circuit as captured by the PropScope and plotted against time. The y axis has a 0.5-volt/division scale. In this example, the voltage does not drop to 0 volts because the Propeller outputs to the DAC have a logic-0 voltage of about 0.3 volts. A commercial DAC would have a 0-volt output for the corresponding digital input.

Step 7.

Because the 4-bit DAC can produce only 16 voltages, it has little use in real applications. Most of the time, circuit designers buy a DAC IC that includes a reference-voltage source, an output amplifier, and latches that can "grab" and hold a binary value. These ICs can offer a resolution from 8, 10, 12, or more bits. How many voltages would you expect from an 8-bit DAC. Find the numbers in the Answers section at the end of this experiment.

Suppose a 12-bit DAC has a 2.5-volt reference, which means the output could produce voltages between 0 and 2.5 volts. A step between adjacent 12-bit binary values would represent what change in voltage?

The 12-bit converter can produce 4096 different voltages (2^{12}), but only has 4095 (2^{12} - 1) steps between them. Calculate the voltage per step:

$$2.5\ volts\ /\ 4095\ steps\ \ =\ \ 0.00617\ volts/step\ \ or\ \ 6.17\ millivolts/step$$

For a DAC with an *n*-bit input, use the general formula:

$$V_{REF}\ /\ (2n\ -\ 1)\ \ =\ \ voltage\ per\ step$$

The voltage per step also represents the voltage contributed by the least-significant bit (LSB) on the DAC. Consider an 8-bit DAC with a 9.8 mV/step output. If we apply 00000000_2 to the eight inputs, we get 0 volts out. Change the LSB (D0) to a 1, 00000001_2, and the output voltage should equal 9.8 mV. When bit D1 changes state, it contributes either 0 or 19.6 mV (2 * 9.8 mV). The MSB (D7) contributes 1.25 volts (128 * 9.8 mV). To determine the voltage a given bit provides, multiply the bit position's "weight," by the voltage per step.

Step 8.

Instead of continuing to experiment with the 4-bit DAC-resistor circuit, you will use a Linear Technology LTC1450L 12-bit DAC IC. This device operates from a 2.7-to-5.5-volt power supply, which means it can connect directly to the Propeller MCU I/O pins. The LTC1450L includes a 1.22- and a 2.50-volt reference, so you can choose either one. **Figure 15.7** provides a block diagram for this DAC. You can download a complete 16-page data sheet for this IC at: http://cds.linear.com/docs/en/datasheet/14500lf.pdf.

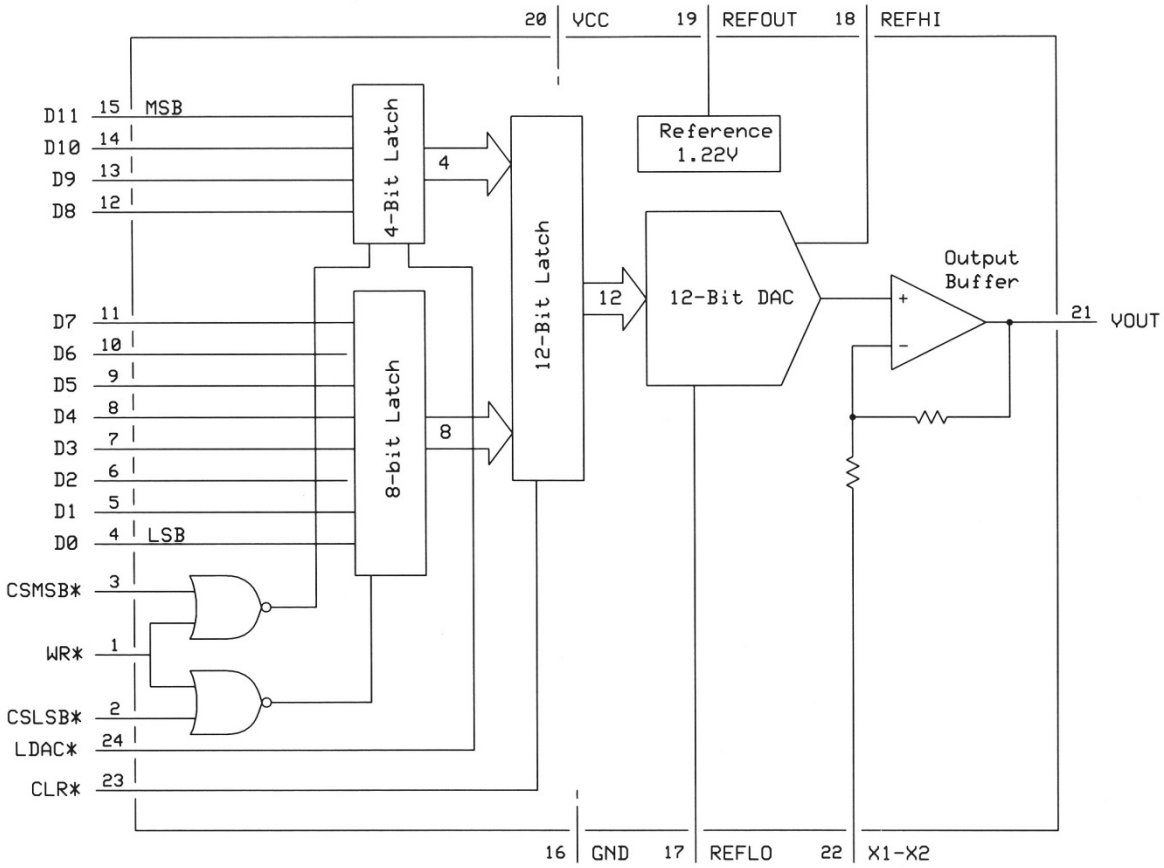


Figure 15.8.
Block diagram for a Linear Technology LTC1450L 12-bit DAC with voltage output. This IC has an operating voltage range from 2.7 to 5.5 volts (VCC). Pin numbers shown here correspond to those for a 24-pin dual in-line package (DIP) IC.

Before you can use this DAC IC, you must understand how its 4-bit, 8-bit, and 12-bit latches work. A latch circuit takes the value at its digital input and transfers it to its output when it receives a control signal. The state of the output will not change until the latch receives another input and another control signal. Think of a latch as an n-bit memory with n inputs, n outputs, and one control signal. The LTC1450L IC offers several ways to input new 12-bit information.

Option A. If a circuit can provide the DAC with all 12 bits (D11-D0) simultaneously, we can "open" the 4-bit and 8-bit latches so data can always flow through them. These two latches become "transparent" to the 12-bit data. Then the 12-bit latch can capture all 12 bits at once and the DAC output will change accordingly. This approach requires 12 output pins for data and one pins for a control signal.

Option B. The circuit shown in **Figure 15.9** shares four output pins with the D3-D0 and D11-D8 data signals. This arrangement saves four MCU output pins, but as you'll see, it requires two additional output pins to control the DAC. The MCU would put the eight least-significant bits (LSBs) on the eight data lines and load them into the 8-bit latch (see **Figure 15.8**). Next, the MCU would put the four most-significant bits (MSBs) on the four shared data lines and load them into the 4-bit latch. Finally, the MCU would transfer the 8-bit and 4-bit latch bits simultaneously to the 12-bit latch. The DAC output voltage would change accordingly. **Figure 15.10** shows the timing diagram for this type of transfer. The circled numbers along the time axis correspond to the detailed Option B explanations that follow the figure.

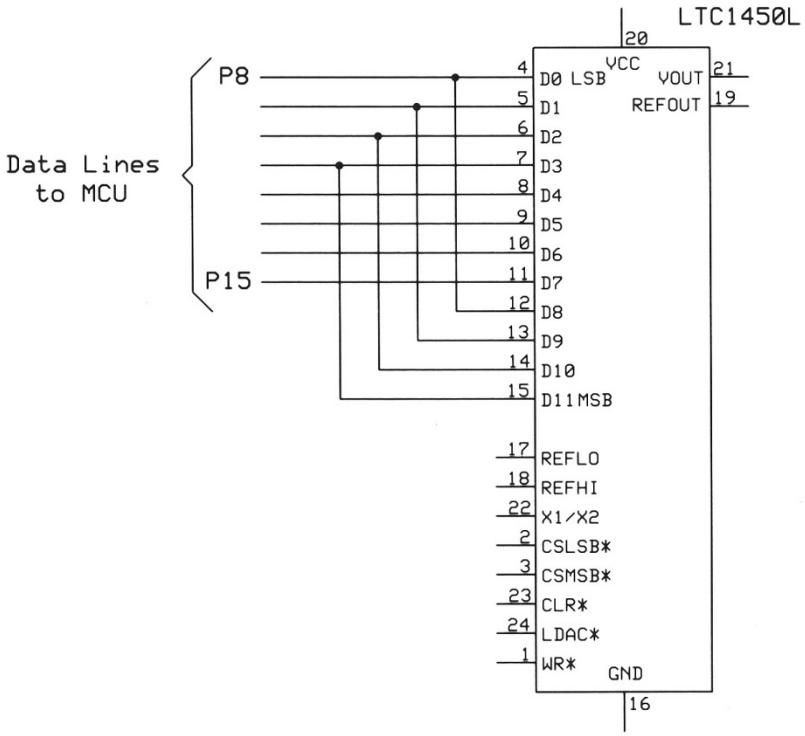


Figure 15.9.
In this circuit, the D3-D0 inputs share MCU connections with the D11-D8 inputs. As a result, the 12-bit DAC requires only eight data lines. A later figure will provide a complete circuit diagram.

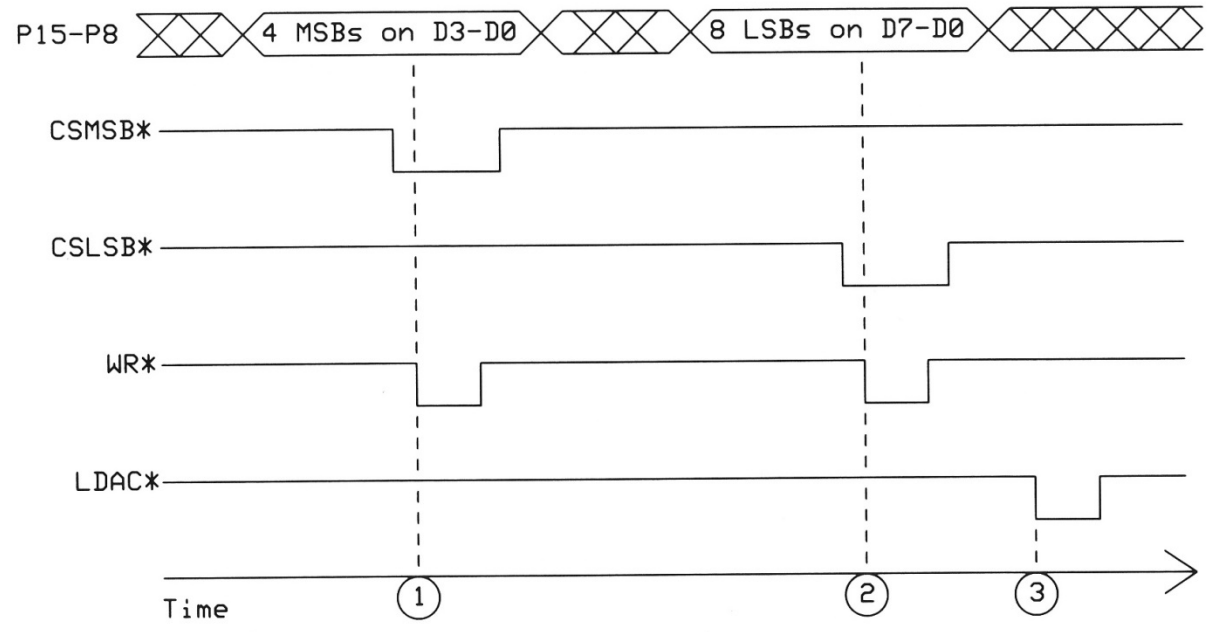


Figure 15.10.
The timing diagram for the circuit shown in **Figure 15.9**. It doesn't matter whether you transfer the 8 LSBs and then the 4 MSBs, or vice versa. The cross-hatch marks for the MCU-board signals indicate we do not care what the MCU puts on these signals during these periods.

- **Time 1.** An MCU places the four MSBs for the DAC on the four shared data connections that go to D11-D8 *and* to D3-D0 pins on the DAC. The MCU places a logic-0 on the CSMSB* input and on the WR* input. The combination of these two signals latches the four MSBs into the 4-bit latch in the LTC1450L.

- **Time 2.** The MCU next places the eight LSBs on the eight data connections. The MCU places a logic-0 on the CSLSB* input and on the WR* input. The eight LSBs go into the 8-bit latch.

- **Time 3.** After the MCU has loaded data into both the 4-bit and the 8-bit latch, it places a logic-0 on the DAC's LDAC* input. This signal simultaneously transfers all 12 bits into the 12-bit latch and holds them. The 12-bit DAC produces the equivalent voltage at its output.

Within a Spin program that controls an LTC1450L DAC, you can handle the 12-bit data in two ways. First, use two byte-size variables, one for the eight LSBs and another for the four MSBs. Second, use a word-size variable and output the MSBs followed by the LSBs, or *vice versa*. I prefer the second because if I increment, decrement, or update information for the DAC, my program needs only one variable. As you'll see shortly, program statements can use a shift operation to easily "splits" the MSBs from the LSBs in a word variable.

Step 9.

Now you will work with an LTC1450L DAC. Place a 24-pin LTC1450L IC in a solderless breadboard and connect it to the P8X32A board as shown in **Figure 15.12**. Recheck your wiring. You will measure the voltage between the DAC's VOUT pin (pin 21) and ground. Ensure you have a common-ground connection between your breadboard circuit, meter, and the P8X32A board.

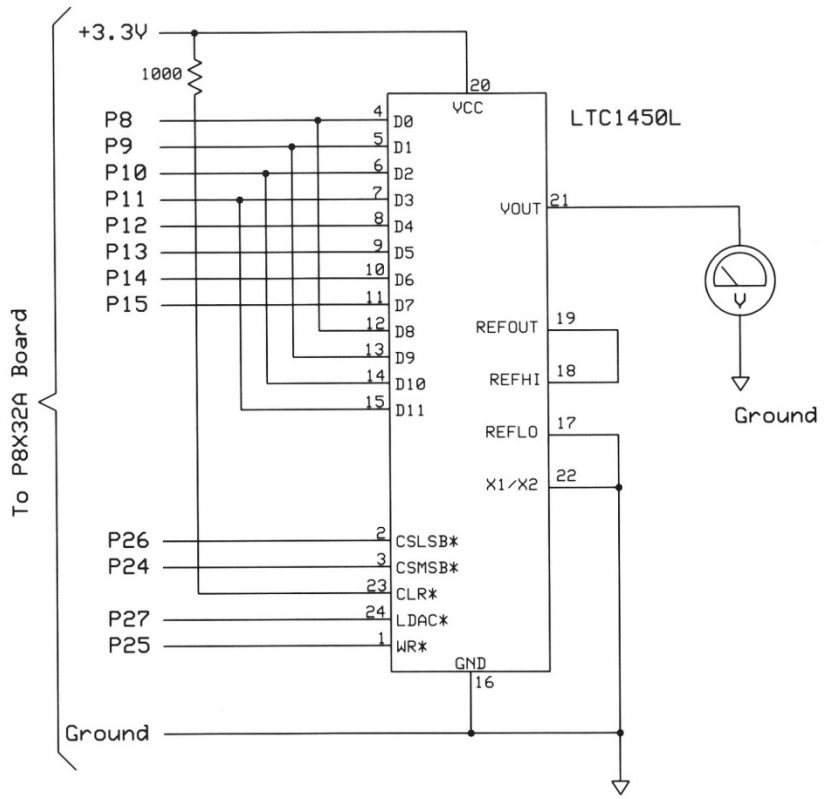


Figure 15.12.
Schematic diagram for the LTC1450L-to-P8X32A MCU board. Ensure you have a common ground between the breadboard and the Propeller board.

The code in **Program 15.3** will cause the DAC output to alternate between the minimum and the maximum voltage every 10 seconds. Connect your voltmeter as shown in **Figure 15.12** and note below the voltages you measured.

Highest voltage = _____ Lowest voltage = _____

Did you see the voltage change between about 0 and 2.5 volts? If not, recheck your wiring. You might have mis-wired some of the data or signal connections to the LTC1450L IC. I found a high voltage of 2.47 volts and a low voltage of 0.489 millivolts, or 0.000489 volts.

Program 15.3.

```
{{
''*********************************************
''*  Program 15.3 Test of LTC1450L 12-bit DAC
''*  Author: Jon Titus 11-19-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Output alternating highest and lowest
''*  12-bit value to LTC1450L DAC and
''*  delay for 10 seconds so user can measure
''*  DAC voltage output on a meter.
''*********************************************
}}

CON _clkmode = xtal1 + pll16x   'Set MCU clock operation
    _xinfreq = 5_000_000        'Set for 5 MHz crystal

    Hi = 1                      'Setting for logic-1
    Lo = 0                      'Setting for logic-0

OBJ
  delay     :  "timing"         'timing.spin file

VAR
 word DAC_value                 'Variable for 12-bit DAC circuit

PUB main
 DIRA[15..8]  := %11111111      'Set eight outputs to drive DAC
 'P8 to D0 and D8 on DAC
 'P9 to D1 and D9
 'P10 to D2 and D10
 'P11 to D3 and D11
 'P12 to D4
 'P13 to D5
 'P14 to D6
 'P15 ro D7

 DIRA[27..24] := $F            'Set four control pins as outputs
 OUTA[27..24] := %1111         'Clear four control lines,
                                     'set to logic-1
 'P24 to /CSMSB
 'P25 to /WR
 'P26 to /CSLSB
 'P27 to /LDAC
```

```
 DAC_value  := 0               'Set DAC output to zero

 'Repeat this loop forever
 repeat
    OUTA[11..8] := DAC_value >> 8   'Output 4 MSBs to DAC
    OUTA[24] := Lo                  'CSMSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[24] := Hi                  'CSMSB* to logic-1

    OUTA[15..8] := DAC_value        'Output 8 LSBs to DAC
    OUTA[26] := Lo                  'CSLSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[26] := Hi                  'CSLSB* to logic-1

    OUTA[27] := Lo                  'Pulse the LDAC* pin
    OUTA[27] := Hi

    'switch DAC from highest to lowest value and
    'vice versa so you get highest and lowest voltage
    'output to measure.

    if (DAC_value == 0)             'if DAC_value = 0, add 4095
        DAC_value += 4095
    else                            'if DAC_value not = 0, make
        DAC_value := 0              'it zero
    delay.pause1ms(10000)           '10-second delay

''  –  – -End –  – -
```

A simple loop in **Program 15.4** increases the 12-bit value by 1 and transfers the new value to the LTC1450L DAC via the type of circuit shown in **Figure 15.12**. This loop will run "forever" so the DAC output will increase up to the maximum voltage and then start again from 0 volts. You can run this program, but unless you have an oscilloscope you can't easily observe the voltages from the DAC.

Program 15.4.

```
{{
''*********************************************
''*  Program 15.4 Test of LTC1450L 12-bit DAC
''*  Author: Jon Titus 11-16-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Output a 12-bit value to LTC1450L DAC and
''*  increment the 12-bit DAC value by one each
''*  pass through the repeat loop. No delay.
''*********************************************
}}
CON _clkmode = xtal1 + pll16x   'Set MCU clock operation
    _xinfreq = 5_000_000        'Set for 5 MHz crystal

    Hi = 1                      'Setting for logic-1
    Lo = 0                      'Setting for logic-0

VAR
 word DAC_value                 'Variable for output to 12-bit
```

```
                                'DAC circuit
PUB main
  DIRA[15..8]  := %11111111      'Set eight outputs to drive DAC
  DIRA[27..24] := $F             'Set four pins as outputs
  OUTA[27..24] := %1111          'Clear these outputs

  DAC_value  := 0                'Initialize DAC output to zero

  'Repeat this loop forever
  repeat
    DAC_value := 1 + DAC_value      'Increment DAC_value by one

    OUTA[11..8] := DAC_value >> 8   'Output 4 MSBs to DAC
    OUTA[24] := Lo                  'CSMSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[24] := Hi                  'CSMSB* to logic-1


    OUTA[15..8] := DAC_value        'Output 8 LSBs to DAC
    OUTA[26] := Lo                  'CSLSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[26] := Hi                  'CSLSB* to logic-1

    OUTA[27] := Lo                  'Pulse LDAC* pin to 0
    OUTA[27] := Hi                  'LDAC* back to logic-1

'  –  – -End –  – -
```

Figure 15.11 shows my results as captured by a PropScope. The voltage increases to 2.5 volts and then drops back to 0 volts. The ramp – also called a sawtooth wave – represents 2^{12}, or 4096 voltages. A close look at the ramp seems to reveal voltage steps, but they relate to the PropScope sampling techniques. You'll also see three small voltage glitches probably caused by other equipment. The software takes about half a second to produce one ramp.

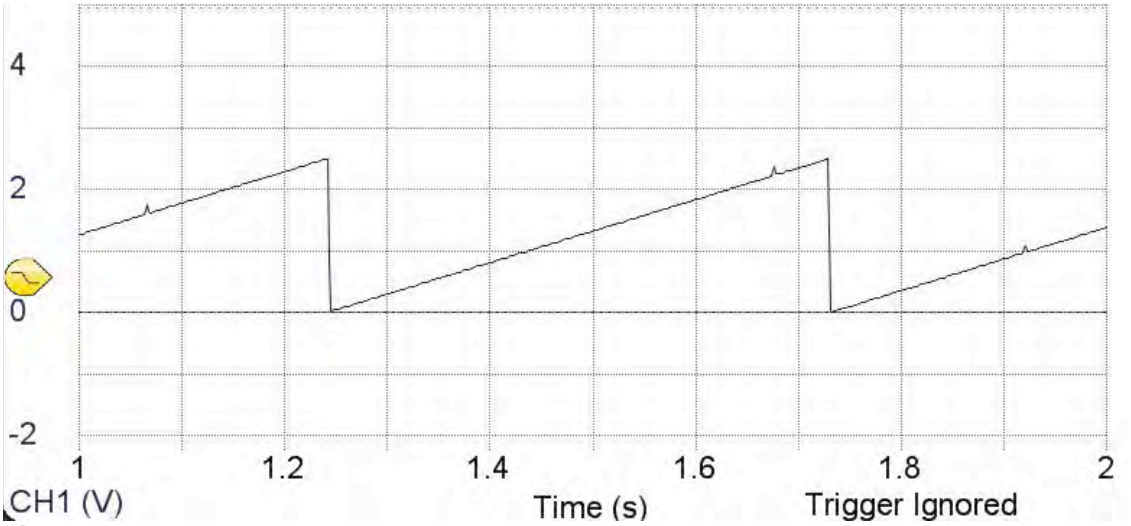


Figure 15.11.
I obtained the voltage ramp shown here when I ran **Program 15.3** to test the 12-bit DAC. The voltages on the y axis indicate the DAC's output has a range from 0 to 2.5 volts.

Step 10.

Program 15.4 increases the 12-bit DAC value by one. Let's see what happens when we increase the step size to eight. **Figure 15.13** shows the new voltage signal. Now the Propeller MCU takes less time to create a complete ramp because it needs only 512 steps (4096 / 8). I measured a 58.9-msec ramp period.

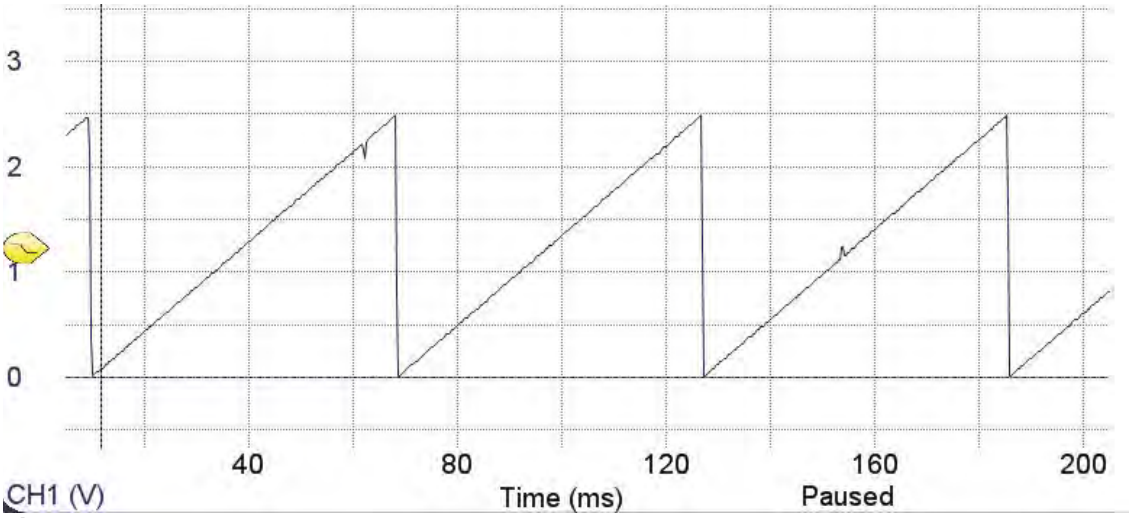


Figure 15.13.
Adding eight to the 12-bit DAC value each time through the `repeat` loop creates fewer voltage steps, although you cannot see them here. In this diagram I increased the voltage scale so you can better see the ramp.

When I changed **Program 15.4** to add 64 to the DAC value each time through the `repeat` loop I could see individual steps on an oscilloscope trace. But by adding 64, I reduced the number of voltage steps to only 64 (4096 / 64 = 64). In this case, the period for the voltage ramp measured only 7.33 msec.

Step 11.

Look closely at the voltages produced by the LTC1450L DAC when you add a value greater than one to the 12-bit digital-input value. You will notice an interesting effect. Start with the binary value: 0000_00000000. (I'll use the Propeller-type underline to separate bytes.) Then add 64 ($0000\_01000000_2$) to increase the DAC output voltage in large steps. The binary values for the 12-bit DAC appear in sequence as:

```
0000_00000000  =  0
0000_01000000  =  64
0000_10000000  =  128
0000_11000000  =  192
0001_00000000  =  256
0001_01000000  =  320
```

and so on...

The final value comes to:

`1111_11000000 = 4032,` or $63 \, x \, 64 = 4032$

That value corresponds to a 2.46-volt output from the DAC, not 2.5 volts. The larger the step size used to increment the 12-bit DAC value, the larger the voltage difference from the final DAC value and the maximum value for a given reference voltage.

Circuit designers overcome this problem by using a different reference voltage and/or an operational amplifier (op-amp) to scale a DAC's output to cover almost any voltage or current range for a given number of bits. In

my opinion, op amps provide the most versatile of all *analog IC* components, and they require their own books for thorough explanations and experiments. For more information, see the References section at the end of this experiment.

Step 12.

By incrementing a count and sending it to the LTC1450L DAC software produced a sawtooth wave. Could you modify **Program 15.4** to create a triangle wave? The voltage in this type of signal increases stepwise to a maximum and then decreases stepwise to a minimum, and then increases again, and so on. Go ahead and change the code.

If you get stuck or can't get a program to work, **Program 15.5** shows the steps needed to increase a voltage, then decrease the voltage and continue this routine indefinitely. **Figure 15.14** shows the resulting signal. In **Program 15.5** I created a public object, `Output_DAC_data` that handles the word-length data and the control signals. You can use that object in programs of your own to control an LTC1450L DAC IC.

Program 15.5.

```
{{
''*********************************************
''*  Program 15.5 Test of LTC1450L 12-bit DAC
''*  Author: Jon Titus 11-16-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Output values to LTC1450L DAC to create a triangle
''*  waveform on the DAC output. Each "side"
''*  of the triangle wave has 64 DAC voltage steps.
''*********************************************
}}
CON _clkmode = xtal1 + pll16x   'Set MCU clock operation
    _xinfreq = 5_000_000        'Set for 5 MHz crystal

    Hi = 1                      'Setting for logic-1
    Lo = 0                      'Setting for logic-0

VAR
 word DAC_value                 'Variable output to 12-bit DAC

PUB main

 DIRA[15..8]  := %11111111      'Set eight outputs for DAC
 DIRA[27..24] := $F             'Set four control outputs
 OUTA[27..24] := %1111          'Clear four control lines

 DAC_value  := 0                'Initialize DAC output to zero

 'Repeat this loop forever
 repeat
    repeat 63                       'go through loop 63 times
        Output_DAC_data(DAC_value)  'output DAC_value
        DAC_value += 64             'add 64 to DAC_value

    repeat 63                       'go through loop 63 times
        Output_DAC_data(DAC_value)  'output DAC_value
        DAC_value -= 64             'subtract 64 from DAC_value

'Object to output a 12-bit value to the LTC1450L 12-bit DAC
```

```
PUB Output_DAC_data (DAC_out)

    OUTA[11..8] := DAC_out >> 8     'Output 4 MSBs to DAC
    OUTA[24] := Lo                  'CSMSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[24] := Hi                  'CSMSB* to logic-1

    OUTA[15..8] := DAC_out          'Output 8 LSBs to DAC
    OUTA[26] := Lo                  'CSLSB* to logic-0
    OUTA[25] := Lo                  'WR* to logic-0
    OUTA[25] := Hi                  'WR* to logic-1
    OUTA[26] := Hi                  'CSLSB* to logic-1

    OUTA[27] := Lo                  'LDAC* pin to logic-0
    OUTA[27] := Hi                  'LDAC* back to logic-1

' –  – -End –  – -
```

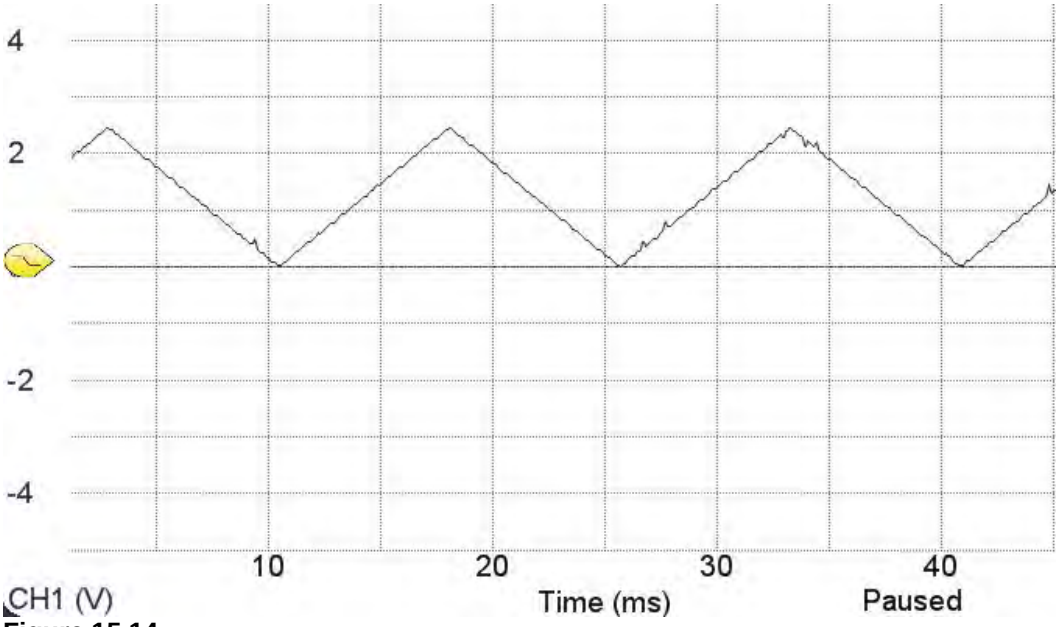


Figure 15.14.
Triangle-wave signal produced by **Program 15.5**.

If you have a way to observe the voltage-versus-time waveforms created under software control, you can experiment with software to produce many types of signals, from square waves or pulses to sine waves. Many school labs have an oscilloscope, a plug-in scope module for a PC, or a USB scope pod for a PC that acquires a signal and displays it vs. time on a computer monitor. And you might find software and accessories for a tablet computer so it will operate like a scope. A Google search will find a variety of products.

If you want to experiment with sound signals, the Propeller Object Exchange lists many interesting programs you can try. Many do not need a DAC.

Step 13.

If you expect to go on to Experiment 16, leave your LTC1450L IC in the solderless breadboard, but remove the connections to it. You will use the DAC in a different configuration.

Before you leave this experiment I'll give you a real-world example of how an analog output from a computer can help solve problems. When engineers design a satellite they must test it under many conditions. Suppose the satellite has a sensor that will produce an analog voltage to let the satellite's computer know the temperature inside sensitive equipment has gotten too hot. The engineers must thoroughly test the computer software to ensure it operates correctly as it receives the sensor's signal. (The satellite computer might use the temperature information to determine how to point the equipment away from the sun.) Unfortunately the electronics and its sensor got delayed and won't arrive for another two months! What can the engineers do in the meantime?

Instead of waiting for the sensor, the engineers decide to use a microcontroller with a DAC connected to it. They program the MCU so the DAC will produce a signal that *simulates*, or mimics, the type of signal the sensor would provide. This approach lets the engineers watch how the computer system responds to the DAC signal. The engineers can program the MCU to produce a steady voltage, slowly changing signals, or signals that change rapidly. The satellite's computer doesn't "know" the sensor signals come from a DAC and not the sensor itself.

You will find engineers routinely use this type of test system to produce signals that simulate the output from a wide variety of sensors. After all, even if the satellite engineers had a temperature sensor, it might take a lot of other equipment and heat sources to provide the satellite's computer with realistic information.

A final note: You can buy DACs with more than one output and some DACs provide a serial rather than a parallel interface. Companies such as Linear Technology, Microchip Technology, Analog Devices, Silicon Laboratories, and Texas Instruments manufacture a variety of DACs.

Advanced Information

The Propeller MCU includes a built-in lookup table of sine values that you may use to create a sine-wave signal. For more information, see "Appendix B: Math Samples and Function Tables," in the Parallax Propeller manual available online. The lookup table covers one quadrant, 0 to 90°, but some math operations let you get produce values for all four quadrants.

Reference

For more information about the math behind the R-2R DAC resistor ladder, see the tutorial, "Digital to Analog Conversion – The R-2R DAC," available on the Tektronix Web site at: http://www.tek.com/blog/tutorial-digital-analog-conversion-–-r-2r-dac.

Carter, Bruce, and Ron Mancini, "Op Amps for Everyone," 3rd ed., Elsevier Newnes, 2009. ISBN: 978-1-85617-505-0.

Mims, Forrest M., "Timer, Op Amp, and Optoelectronic Circuits & Projects," Master Publishing, Inc. 2004. ISBN: 978-0945053293.

Answers

Experiment 15, Step 7:

a) An 8-bit DAC would produce 2^8 distinct voltages (256) and a 12-bit DAC would produce 2^{12} distinct voltages (4096).

b) If you have an 8-bit DAC with a 2.5-volt reference, what voltage would the MSB contribute to the DAC output?

The MSB in an 8-bit value has the value of 0 or 128: $10000000_2 = 128$. Because the LSB contributes 9.8 mV to the output, you simply multiply:

$128\ steps\ x\ 9.8\ mV\ /\ step\ =\ 1254.4\ mV\ or\ 1.25\ volts$.

You can approach the problem in another way. If the LSB contributes 9.8 mV, the next bit to the left contributes twice as much, or 19.6 mV. You can continue to double the voltage for each bit to the left:

```
LSB   D0     9.8 mV
   a. D1    19.6
   b. D2     39.2
   c. D3     78.4
   d. D4    157
   e. D5    314
   f. D6    627
MSB   D7  1250 mV
```

I rounded the results above to three significant digits.

Experiment No. 16 – Analog-to-Digital Conversion and How It Works, Part 1

Abstract

This experiment demonstrates two analog-to-digital conversion techniques and explains how they work. This type of conversion produces a digital value proportional to an applied analog signal. Software in a Propeller microcontroller will manage the processes and provide a binary value. This type of software-controlled conversion works well for demonstrations, but most circuit designers rely on ICs that handle the process on their own. You also will learn how to set up an analog-to-digital converter (ADC) IC with two inputs for analog signals. Please run Experiment 15 before you tackle this one. In Experiment 17 you will work with two sensors that connect to an ADC

Keywords

Analog-to-digital-converter, ADC, digital-to-analog converter, DAC, ramp, successive approximation, comparator, analog converter, LTC1450L, MCP3202, voltage measurements

Requirements

(1) - Propeller P8X32A microcontroller board
(1) - LTC1450L digital-to-analog converter, 24-pin DIP, Linear Technology
(1) - MCP3202 2-input analog-to-digital converter, 8-pin DIP, Microchip Technology
(1) - LM339 comparator, 14-pin DIP (many sources)
(1) - Digital voltmeter or digital multimeter (DMM)
(1) - USB cable
(1) - 220-ohm, 1/4-watt resistor, 5% (red-red-brown)
(1) - 1000-ohm, 1/4-watt resistor, 5% (brown-black-red)
(1) - 3300-ohm, 1/4-watt resistor, 5% (orange-orange-red)
(2) - 10-kohm, 1/4-watt resistor, 5% (brown, black, orange)
(2) - 10-kohm variable resistors, single turn trimmer
(1) - LED (red or green)
(1) - Power supply for 3.3 volts (optional)
(1) - Solderless breadboard

Introduction

Sensors for temperature, pressure, humidity, weight, acceleration, and other physical quantities usually produce an analog voltage you can measure. Years ago, scientists and engineers would take voltage information from a electromechanical meter equipped with a small pointer and a scale, from which they could read a voltage. They would watch the meter and at regular intervals write voltage values in a lab notebook. Or they might use a strip-chart recorder that moves a pen across a moving roll of paper to record information.Today, an analog-to-digital converter (ADC) does the job and when joined with a microcontroller, it can record digital values over long periods at precise intervals.

An ADC accepts a voltage input and then provides a digital value proportional to the applied signal. Digital multimeters (DMMs) or digital voltmeters (DVMs), for example, use an ADC to measure an unknown voltage. An internal microcontroller then uses that measurement information to display a voltage with decimal digits.

In this experiment you will get an introduction to analog-to-digital conversion processes and see two examples that use an MCU for control. The experiment ends with the use of an ADC integrated circuit. References point

you to more information about how to use ADCs, and Experiment 17 gives you an opportunity to connect two sensors – one for light and the other for temperature – to an ADC IC and display results.

The previous experiment introduced you to conversion of digital information to an analog voltage. This experiment investigates how a digital-to-analog converter (DAC) finds use in the analog-to-digital conversion process.

A Bit of History

The analog-to-digital converter didn't just appear in electronics at a "eureka" moment. It arose from the need to communicate more voice signals over longer telephone circuits with less noise. Almost as soon as companies started to offer long-distance telephone service they encountered a problem – noise. As an analog telephone signal went from, say, Boston to Philadelphia, it passed through electronic amplifiers, each of which added some electrical noise to the amplified analog signal. Engineers can reduce this type of noise somewhat with filter circuits, but electronic signals always have some amount of noise associated with them.

To overcome problems with amplifiers and noise, engineers investigated replacing the analog voice signals with digital signals; essentially logic-0s and logic-1s. A logic-1 or a logic-0 pulse might have a bit of noise superimposed on it, but the receiving circuits would still recognize the proper logic levels. Also, engineers could more easily design amplifiers for digital bits. As a result, a conversation converted to digital form and transmitted as a series of bits would not sound noisy when reconstructed for the receiver.

Converting an analog voice signal into a series of pulses presented a challenge because engineers couldn't go to an electronics-supply store and buy an analog-to-digital converter, they had to invent one. Alec Harley Reeves, an engineer in Paris, France created a pulse-code-modulation (PCM) technique that converted an analog signal into a series of pulses that faithfully represented that analog signal.

Reeves' design took an analog signal, sampled it, and created an equivalent electrical pulse. The length, or width, of the pulse corresponded to the amplitude (voltage) of the analog signal at the time it got sampled. A higher voltage created a long pulse and a low-voltage signal created a short pulse. We call this technique pulse-width modulation (PWM). The word "modulation" means one signal alters another. In this case, the incoming analog signal modulates the width of the digital pulse, as shown in **Figure 16.1**.

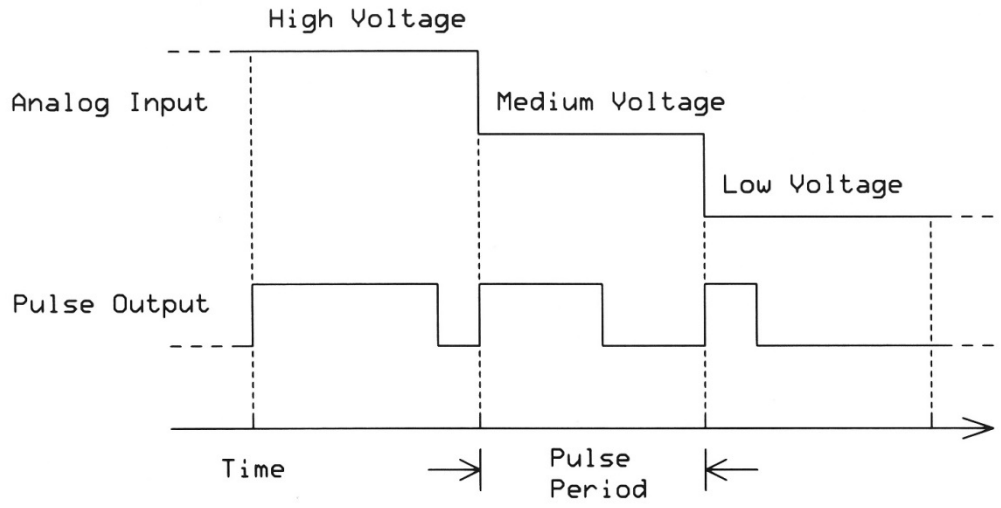


Figure 16.1.
This diagram shows how a modulator circuit will change the width of a pulse based on an input voltage. The higher the voltage, the longer the logic-1 pulse. The tick marks on the time axis indicate the start and end of the period allowed for each pulse.

Instead of transmitting pulses of varying widths, Reeves used the pulses to turn on or off an accurate clock signal applied to a counter, shown in **Figure 16.2**. The longer the PWM pulse, the more clock pulses pass through the AND gate to the counter. A short pulse lets only a few clock pulses pass through to the counter. A binary counter produced a 5-bit binary code for each pulse.

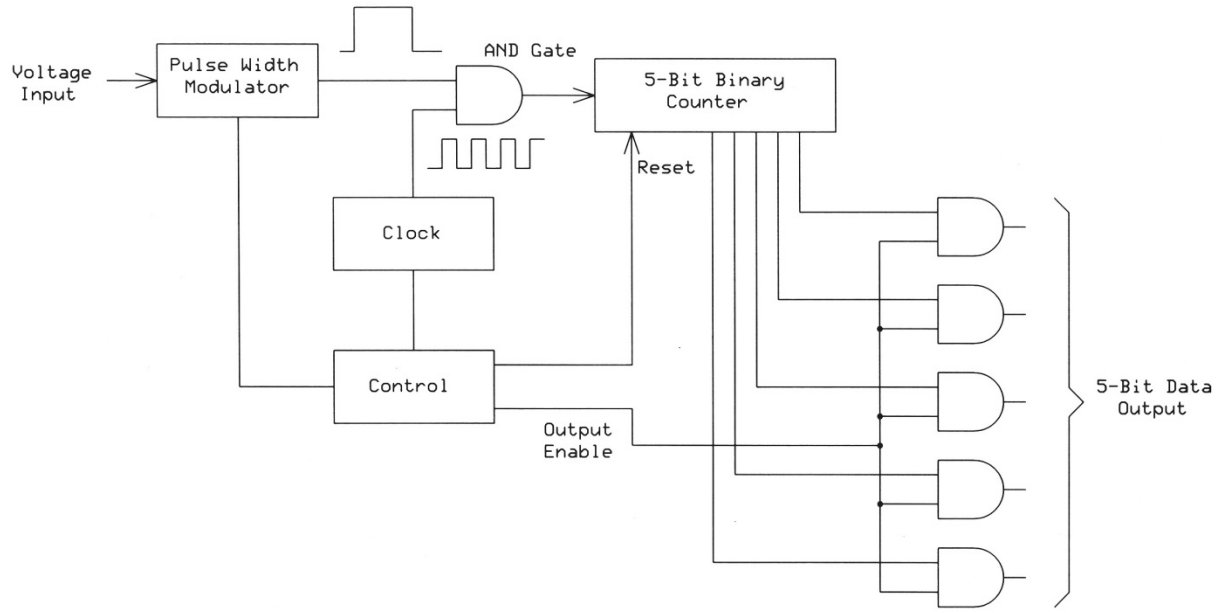


Figure 16.2.
This block diagram represents the type of analog-to-digital converter created by Alec Harley Reeves as a way to reduce noise in long-distance communications. The circuit receives a varying voltage created by a voice and converts it into a series of 5-bit binary codes. Communications sent as digital bits help reduce the effects of electrical noise.

After each pulse, additional circuits (not shown in **Figure 16.2**) took the 5-bit code and transmitted it in a string of logic-0 and logic-1s as a pulse-code modulation (PCM) signal seen in **Figure 16.3**. Thus, each of the original three analog samples created a corresponding 5-bit code. At the receiving end of the PCM signal, digital-to-analog converter circuits translated each 5-bit code into a voltage to reproduce the voice of the caller. For more information about analog-to-digital-converter history and developments, see the references at the end of this experiment.

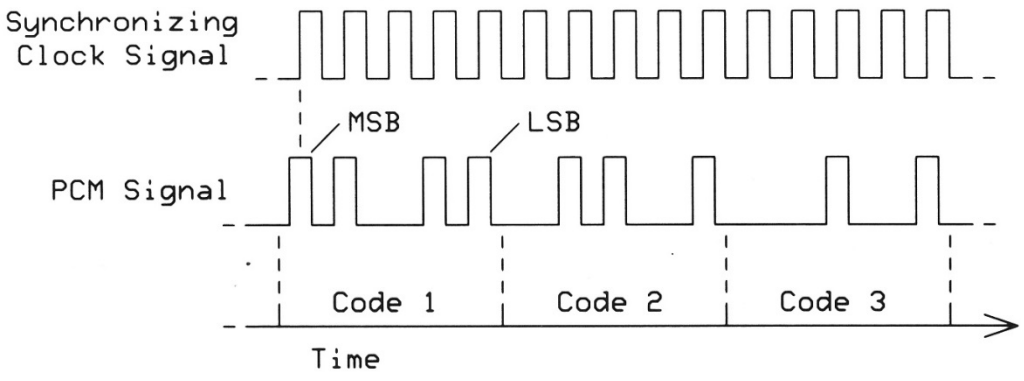


Figure 16.3.
This example shows three pulse-code modulation (PCM) values as communication equipment might transmit them. The synchronizing clock signal shows how electronic circuits at a receiver would sample the logic state of the PCM signal on the rising edge of the clock signal. Code 1 represents the value 110112, or 2710. Code 2 represents 13, and Code 3 represents 5.

Since Reeves designed his ADC circuit, converter technologies have advanced greatly. Today, you can buy an ADC IC for a few dollars. Many MCUs include an ADC, so you might not one in many applications. The two techniques you'll investigate in this experiment rely on the DAC circuit you assembled with the LTC1450L IC in Experiment 15. You should run that experiment before you proceed with the following steps.

Step 1.

Experiment 15 illustrated how to use the LTC1450L as a 12-bit DAC. This experiment also uses the LTC1450L IC, but as an 8-bit DAC. The new configuration simplifies hardware connections and software but still illustrates key concepts of analog-to-digital conversion techniques. If you still have the circuit connected as shown in Experiment 15, **Figure 15.12**, please remove all the connections to the LTC1450L IC so you can start with a "clean" breadboard.

Now connect the LTC1450L IC as shown in **Figure 16.4**. Recheck your wiring and ensure you have a ground connection between your breadboard circuit and the P8X32A board. You may use the 3.3-volt output on the P8X32A board at pin 38 on the I/O connector to power your breadboard circuit, or use a separate 3.3-volt power supply.

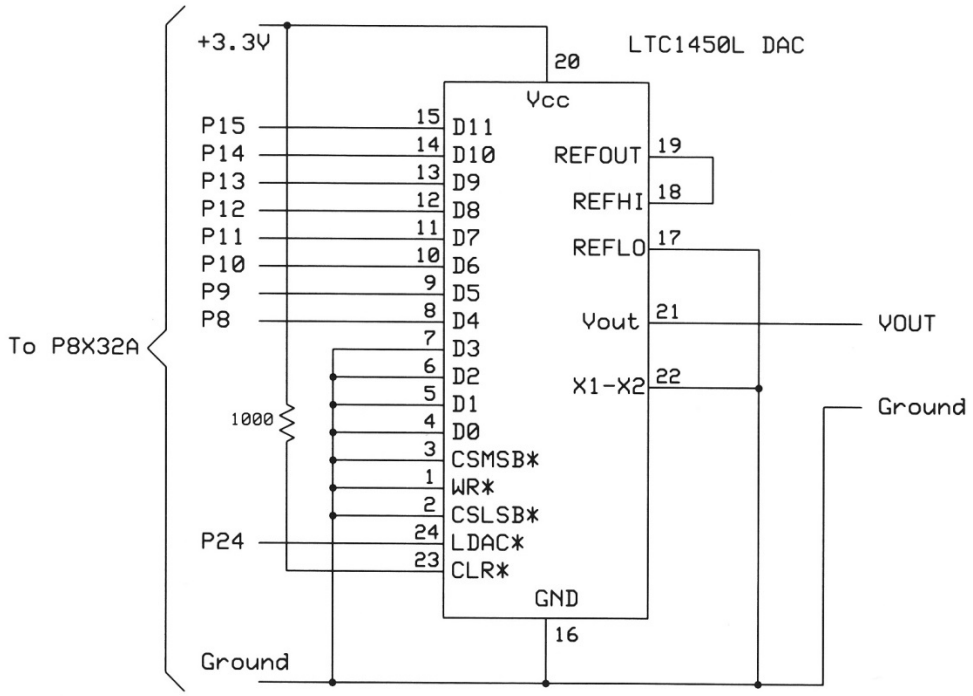


Figure 16.4.
Schematic diagram of a12-bit LTC1450L DAC connected to a Propeller P8X32A. In this circuit the DAC operates as an 8-bit device. The four LSBs, D3-D0, connect directly to ground, or logic-0. This circuit requires a ground connection with the MCU.

Open the software folder for Experiment 16 and ensure you have **Program 16.1** available. This folder also should contain the files: ShiftIO.spin and Timing.spin. You will see other files we'll use later. Within the Propeller Tool window, open **Program 16.1** and run it. This program generates the same type of voltage ramp you saw in Experiment 15, but it takes about 25 seconds per ramp so you can use a voltmeter to watch the voltage increase. The program also uses only the eight most-significant bits (MSBs) on the LTC1450L DAC IC. The four least-significant bits (LSBs) connect to ground (logic-0) and only slightly affect the DAC voltage steps.

Program 16.1.

```
{{
''*********************************************
''*  Program 16.1 DAC Ramp Test
''*  Author: Jon Titus 11-20-2014 Rev. 2
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Uses Parallax P8X32A Propeller board.
''*  Use LTC1450L DAC IC as an 8-bit DAC to
''*  create a positive-going ramp as a test.
''*  8 bits connect to 8 MSBs. 4 LSBs connect
''*  ground (logic-0).
''*********************************************
}}

CON
  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal

VAR
byte Counter                        'Variable for loop counter
                                    'and DAC output

OBJ
  delay :   "timing"                'Use the objects in the
                                    'timing.spin code
'Main program starts here
PUB main
 dira[15..8] := %11111111           'P15 – P8 as output pins
 outa[15..8] := %00000000           'Clear DAC output to 0V
 dira[24] := %1                     'P24 latch DAC data
 outa[24] := %1                     'P24 = logic 1

'Code to initialize the DAC
 Counter     := 0                   'Set Counter to zero to
                                    'start DAC at 0 volts
 outa[15..8] := Counter             'Reset DAC for 0 volts out
 outa[24]    := %0                  'Load DAC with data
 outa[24]    := %1

repeat
   outa[15..8] := Counter          'Send Counter value to DAC
   outa[24]    := 0                'Load DAC with value
   outa[24]    := 1
   Counter++                       'Add 1 to Counter value
   delay.pause1ms(100)             'Wait 0.1 second (100 msec.)

 ' –  – -End –  – -
```

To increase or decrease the delay between the voltage steps produced by the LTC1450L DAC, change the value in the statement:

```
delay.pause1ms(100)
```

which now creates a delay of 100 msec, or one tenth of a second.

Because this circuit uses only the eight most-significant bits (MSBs) of the LTC1450L DAC, how many discrete voltages will the DAC produce? Assume the LTC1450L uses its 2.50-volt internal reference. What voltage difference will you observe between adjacent voltage steps? Find the answers at the end of this experiment.

Step 2.

How can a DAC help convert an analog signal into a digital value? We use a circuit that compare the DAC's voltage-ramp output with an unknown voltage and wait until they match. And a voltage-comparator IC does just that.

An LM339 comparator IC comprises four independent comparator circuits that operate with a single power-supply voltage between 2 and 36 volts. (Some comparators use a bipolar power supply of ± X volts.) I chose the LM339 because many semiconductor companies manufacture it, it works with only one power supply, it comes in a 14-pin dual inline package (DIP), and it costs about 40 cents (US).

Each comparator has two signal inputs, IN+ and IN- and one output, OUT. Do not confuse these *signal inputs* with a comparator's power-supply connections. A comparator output changes state when the IN+ and IN- inputs have the same voltage.

To verify the comparator's operation, I ran an experiment with the circuit shown in **Figure 16.5**. Two 10-kohm resistors formed a voltage divider that held the comparator's IN- signal input at about 1.66 volts. The 10-kohm variable resistor let me change the voltage applied to the comparator's IN+ pin. I watched the LED to determine the comparator's output state; LED on = logic-0, LED off = logic-1. **Table 16.1** summarizes the results.

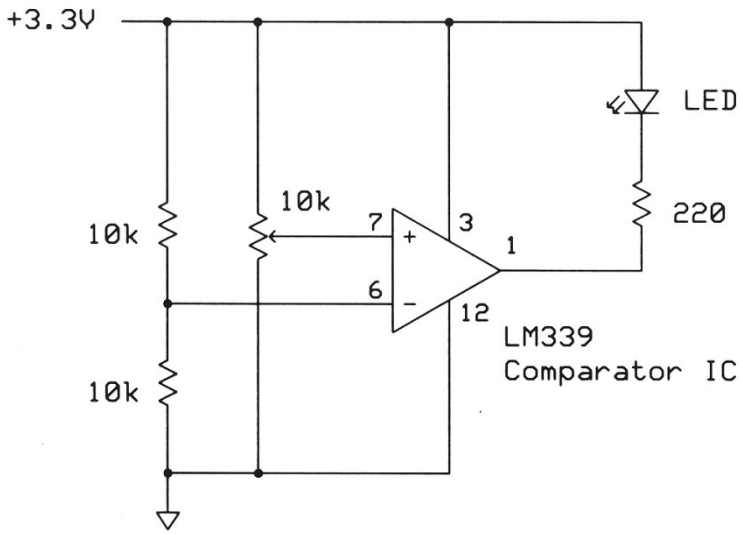


Figure 16.5.
A simple comparator test circuit let me test how the device operates with a fixed input of about 1.66 volts on its IN- input, and a varying voltage applied to its IN+ input. See **Table 16.1** for the results.

Table 16.1. LM339 Comparator outputs for voltage-input conditions.

Comparator Input Voltages	LED State	Logic Output
IN+ > IN-	OFF	1
IN+ < IN-	ON	0

The LM339 comparator provides an open-collector output, which means the output connects to ground to provide a logic-0 output, but for a logic-1, the output looks like an open circuit or disconnected,. To deliver a true logic-1 signal, the comparator output requires a *pull-up resistor* that connects to the positive power supply. You will see such a resistor used in the steps that follow.

Step 3.

Turn off power to your breadboard circuit. Disconnect the USB cable to your PC at the Propeller P8X32A board. Next, place a 14-pin LM339 comparator in your breadboard. Orient the IC so you have pin 1 in the bottom-left corner and refer to the pin-out diagram in **Figure 16.6** for the locations of the power (pin 3) and ground (pin 12) connections.

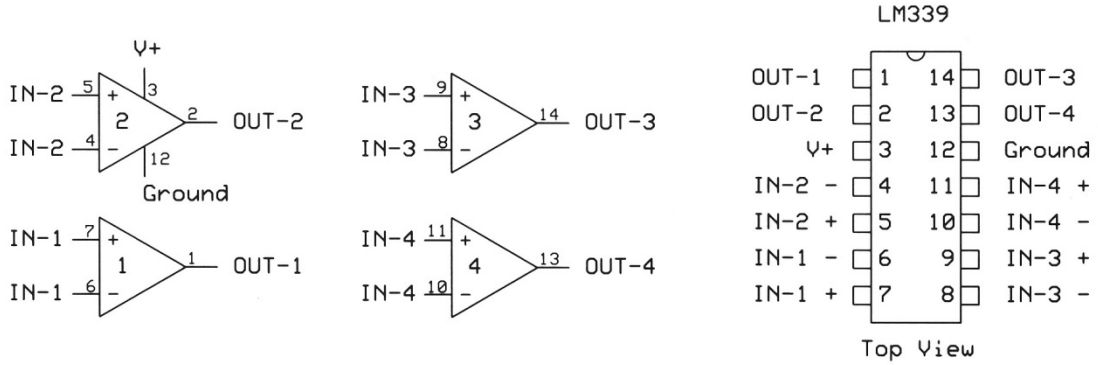


Figure 16.6.
Top view of the pin configuration for a 14-pin LM339 comparator IC and the schematic-diagram of the four independent comparators. The V+ and Ground connections power all four comparators.

Make the connections to the LM339 comparator as shown in **Figure 16.7**. This schematic diagram includes the connections you made earlier to the LTC1450L DAC IC. Do not change or remove any of these connections. The 10-kohm variable resistor and 3300-ohm fixed resistor (orange-orange-red) will provide an "unknown" voltage to the comparator's IN+ input (pin 7) and the voltage produced by the DAC IC will connect to the comparator's IN- input (pin 6). The 3300-ohm resistor keeps the variable-resistor output voltage between 0 and about 2.5 volts. In this configuration, what conditions will the LED indicate?

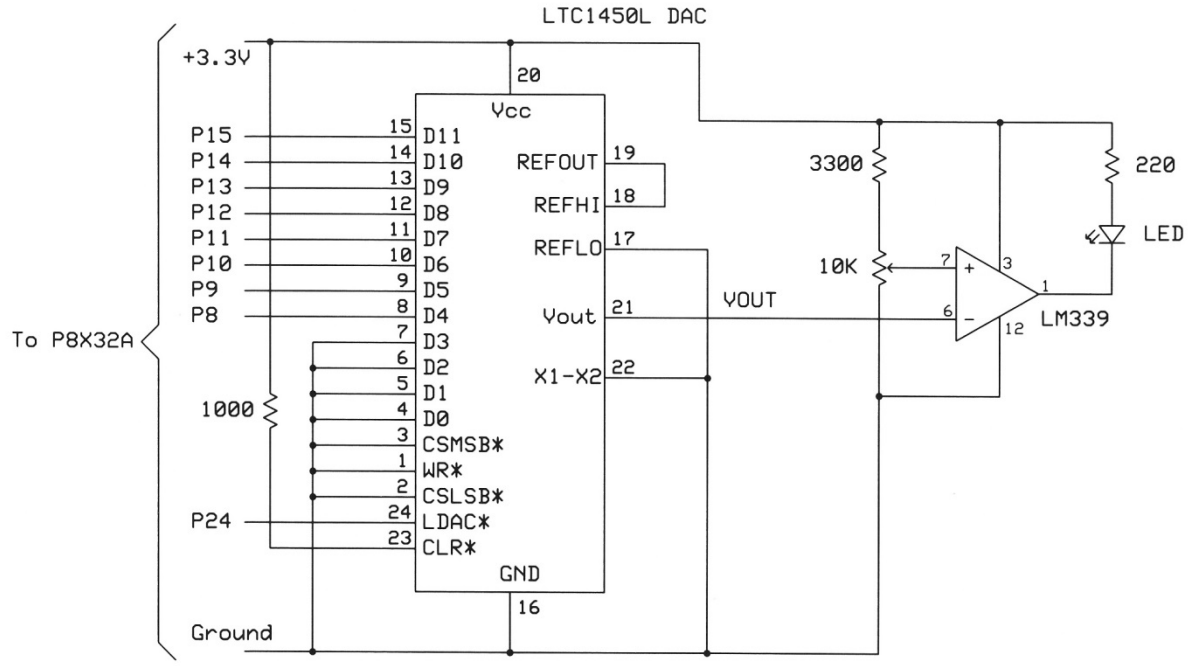


Figure 16.7.
Schematic diagram of an LTC1450L DAC connected to an LM339 comparator. The Propeller MCU will cause the DAC to produce a ramp and the LED will indicate the state of the comparator output. As noted for **Figure 16.5**, the LED-on condition indicates a logic-0 output, and the LED-off state indicates a logic-1 output.

According to the information in **Table 16.1**, the LED will turn on when the voltage from the DAC *equals or exceeds* the unknown voltage from the variable resistor. The LED turns off when the DAC output has a lower voltage than the unknown signal.

Step 4.

Now that the Propeller MCU controls the DAC, a short program can increase the DAC output voltage as you watch the LED. I recommend you connect the voltmeter between the IN+ pin (pin 7) on the comparator and ground. Then you can monitor the unknown voltage from the 10-kohm variable resistor. Adjust this resistor so you have zero volts at the comparator's IN+ input.

Reconnect the P8X32A board to your USB cable, and if you have a separate 3.3V power supply for the breadboard turn it on now. Open the Propeller Tool window and look for the statement `delay.pause1ms(100)` in **Program 16.1**. Change this delay value to 50 for a 50-msec delay. Then run the program. What happens to the LED?

The LED should remain on (logic-0). The variable resistor applies a signal close to 0 volts to the comparator and the DAC output almost immediately exceeds this voltage; IN- > IN+. Adjust the variable resistor so it provides a voltage between 0.2 and 0.3 volts. What happens to the LED now?

You should see the LED turn off for a short period and then turn on for a longer period, as shown in **Figure 16.8** for three voltages. In the V1 case, the LED turns off (logic-1) for the period during which the DAC output has a voltage *lower* than that from the variable resistor; IN- < IN+. **Figure 16.8** also shows the result when the variable resistor provides an output of 1.25 to 1.3 volts (V2). If you adjust the variable resistor to provide a voltage slightly below 2.5 volts (V3), you should see the LED turn on for a short period. Go ahead and adjust the variable resistor, note the voltmeter reading, and watch the LED. Do you see the LED-on period get shorter as you increase the "unknown" voltage from the variable resistor?

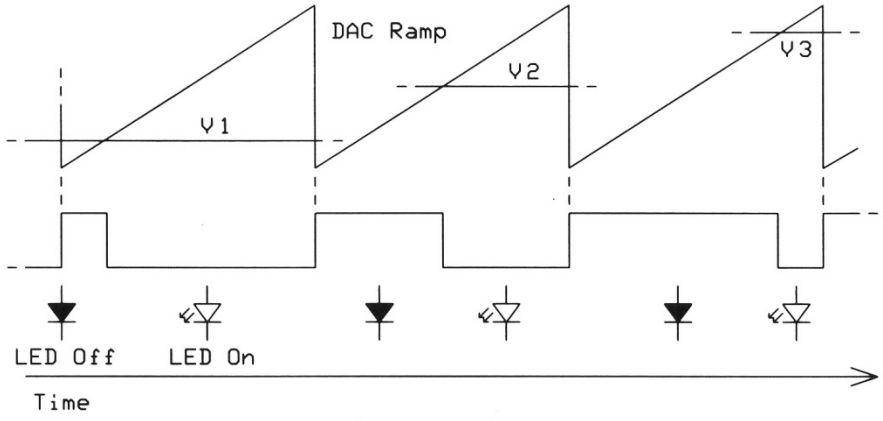


Figure 16.8.
As the unknown voltage from the variable resistor increases it takes longer for the DAV ramp-voltage output to reach the point at which the comparator causes the LED to turn on. At that instant, the ramp voltage from the DAC equals or slightly exceeds the voltage from the variable resistor.

The instant the LED turned on, the 8-bit code at the DAC *inputs,* D11-D4, matched or slightly exceeded the voltage from the variable resistor. So you have an 8-bit code that corresponds to the unknown voltage. In the next steps you will get that code and display it as a value. You might remember from the "A Bit of History" section how Alec Reeves used a circuit to create a pulse with a width that corresponded to an applied voltage. The circuit and software you just tested performs the same function. The LED-off period corresponds to the voltage level from the variable resistor. (The LED-off period represents a logic-1 at the comparator's output.)

Step 5.

The software you ran in Step 4 wastes time. After the comparator signaled – via the LED – that the DAC voltage matched the voltage from the variable resistor, the program continued to generate a complete ramp up to the maximum output from the DAC. Then the ramp began again from 0 volts. If you have a small unknown voltage, the DAC voltage will quickly exceed it, but the software will continue to produce all of the remaining ramp-voltage steps. This time gets "wasted" creating DAC voltages you don't need. You can see this effect in **Figure 16.9**.

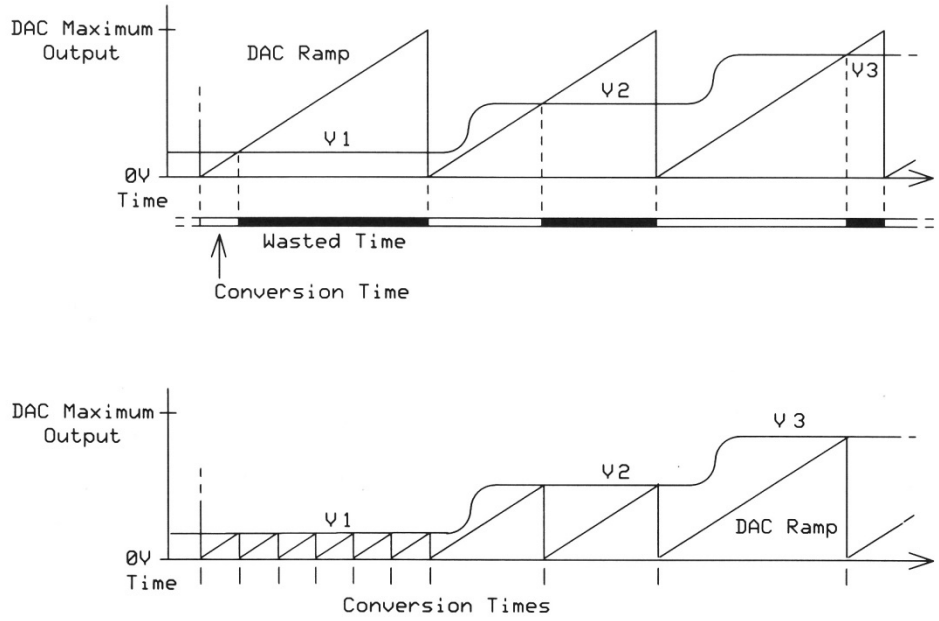


Figure 16.9.
The upper part of this diagram shows how creating a complete ramp to test voltages V1, V2, and V3 wastes time. After the comparator indicates a voltage match, the MCU continues increasing the voltage from the DAC. Continuing the ramp gives us no useful information and simply wastes time. The lower portion shows how conversion times shorten when the DAC ramp ends as soon as the DAC and the unknown voltage match.

So, we need a way to let the MCU know when the comparator output changes from the LED-off to the LED-on state. Instead of using the comparator output to control an LED, we use it to cause the Propeller MCU to stop the ramp and give us the 8-bit value applied to the DAC. As soon as the MCU displays this information, it can reset the DAC to a 0-volt output (0000000_2) and start a new ramp for the next analog-to-digital conversion.

The 8-bit result multiplied by the voltage/step calculated for the DAC gives us the value of the unknown signal in volts.

Step 6.

Turn off power to the Propeller board and the breadboard. Remove the 220-ohm resistor (red-red-brown) and the LED now connected to the LM339 comparator output (pin 1). Connect a 1000-ohm (brown-black-red) resistor between the comparator output and +3.3 volts. Also, connect the comparator output to the Propeller MCU's P25 pin, pin number 26 on the I/O connector. The circled portion of the diagram in **Figure 16.10** shows the circuit modifications.

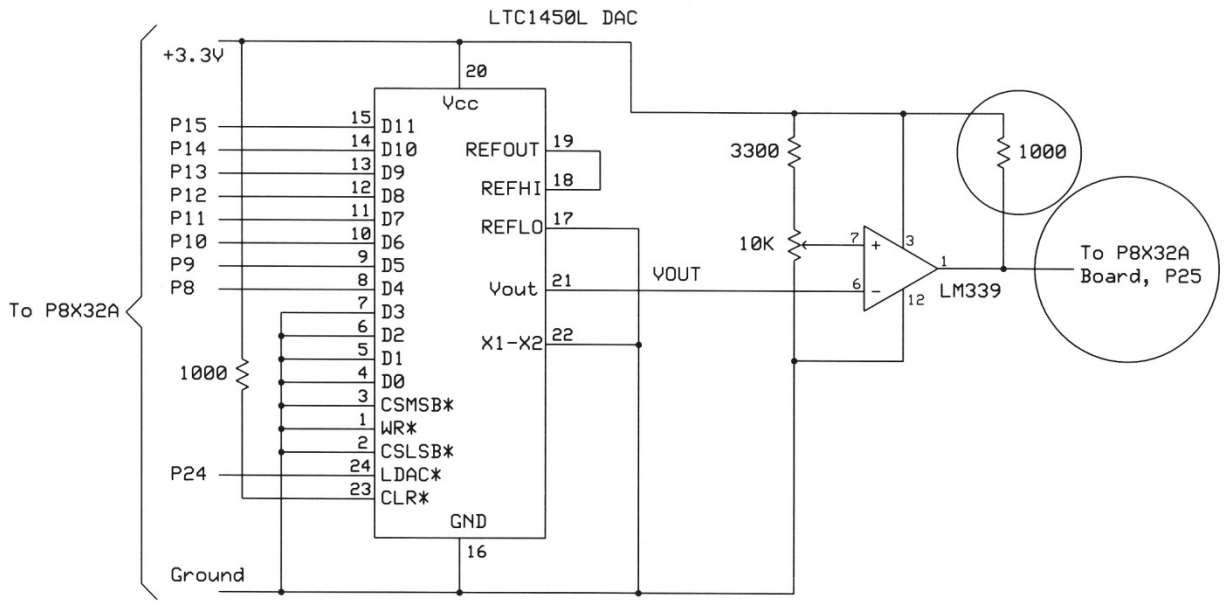


Figure 16.10.
Circuit additions within the circles let the P8X32A MCU determine when the DAC's voltage ramp equals or exceeds the signal from the 10-kohm variable resistor. Remember to remove the 220-ohm resistor and LED shown earlier in **Figure 16.7**.

The new connection between the breadboard circuit and the MCU lets software determine the state of the comparator's output. When the output changes from a logic-1 to a logic-0, the software will stop the ramp, display the 8-bit value applied to the DAC, and restart the ramp back at 0 volts. **Figure 16.11** provides a flow chart that described the operations of the `repeat` loop in **Program 16.2**. That program sends a value to the LTC1450L DAC and then checks the comparator's output signal. As soon as the MCU detects a logic-0 from the comparator, the DAC output voltage equals or just exceeds the unknown voltage. The program transfers the `Counter` value to the eight LEDs, and the process starts again.

Run **Program 16.2**. The eight LEDs on the P8X32A board will display the eight-bit value on the DAC's inputs when the comparator output goes to a logic-0. Adjust the variable resistor and observe the LEDs. You should see a minimum value close to 00000000_2 and a maximum value near 11111111_2. (My LEDs showed 00000000_2 up to 11101000_2. If I replaced the 3300-ohm resistor with a slightly smaller resistance, the unknown voltage from the 10-kohm resistor might reach 11111111_2.)

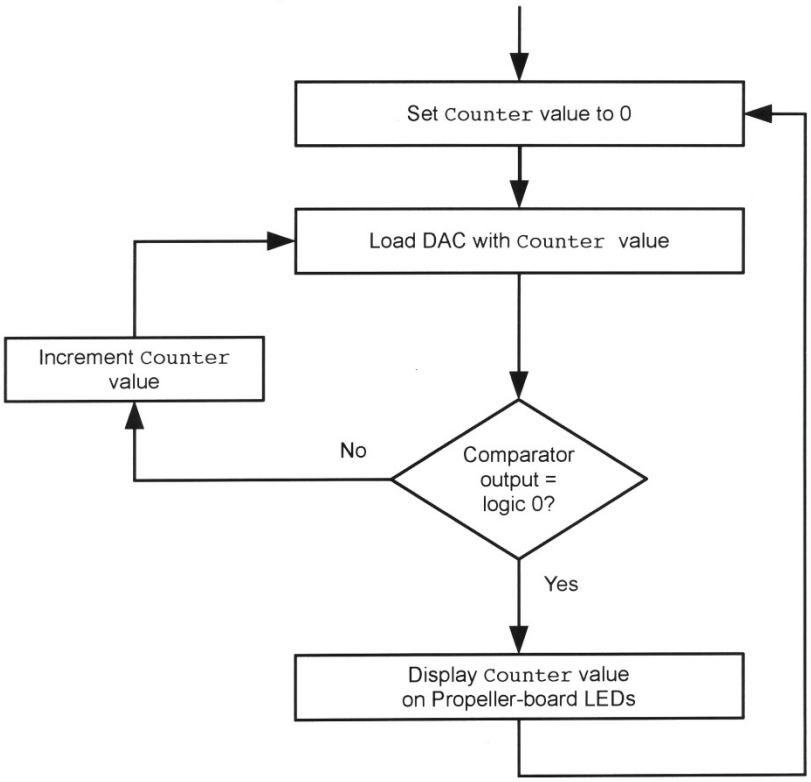


Figure 16.11.
Flowchart for an analog-to-digital conversion that uses a ramp of increasing voltages and a comparator. See **Figure 16.7** for the corresponding circuit.

Program 16.2.

```
{{
''*********************************************
'''*  Program 16.2 Ramp Analog-to-Digital Converter
'''*  Author: Jon Titus 11-20-2014 Rev. 1
'''*  Copyright 2014
'''*  The Propeller created an increasing voltage
'''*  that a comparator IC compares with an unknown
'''*  voltage from a variable resistor. When a match
'''*  occurs, the ramp stops and repeats the
'''*  process again and again.
'''*  Released under Apache 2 license
'''*  Uses Parallax P8X32A Propeller board.
''*********************************************
}}

CON
  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal

VAR
  byte Counter                      'Counter & DAC variable

OBJ
  delay :   "timing"
```

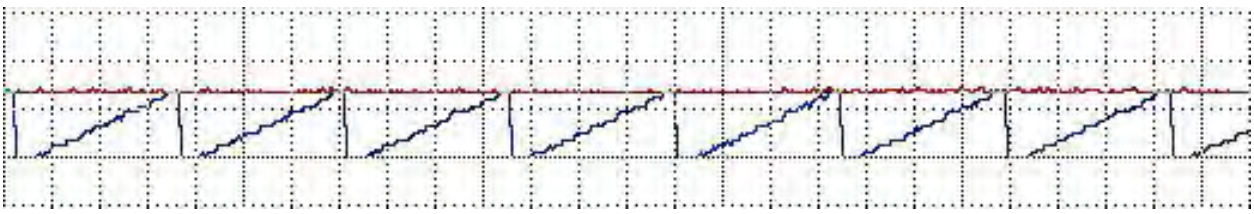
```
'Main program starts here
PUB main
 dira[15..8] := %11111111           'P15 – P8 as output pins
 outa[15..8] := %00000000           'Clear DAC output to 0V
 dira[24] := %1                     'P24 latch DAC data
 outa[24] := %1                     'P24 = logic 1
 dira[25] := %0                     'Set pin P25 as an input pin
                                    'to test comparator output.
 dira[23..16] := %11111111          'Set LED pins as outputs
 dira[26] := 1                      'Optional output to trigger
                                    'a scope trace
'Code to initialize the DAC
 Counter     := 0                   'Set Counter to zero
 outa[15..8] := Counter             'Reset DAC for 0V out
 outa[24]    := %0                  'Load DAC with data
 outa[24]    := %1

'Loop that performs continuous analog-to-digital conversions
    repeat
        if  ina[25] <> 0            'Check comparator for no
                                    'voltage match.
            outa[15..8] := Counter  'No match, so send Counter
                                    'value to DAC P15 – P8
            outa[24]    := 0        'Load DAC
            outa[24]    := 1
            Counter++               'Increment Counter by one

        else                        'OK, equal voltages
                                    'detected, end of
                                    'conversion!
            outa[23..16] := Counter 'Display DAC value on LEDs
            Counter     := 0        'Reset Counter to 0
            outa[15..8] := Counter  'Reset DAC to 0 volts out
            outa[24]    := %0       'Load DAC
            outa[24]    := %1
            outa[26]    := %0       'Create a pulse to trigger
            outa[26]    := %1       'an oscilloscope (optional)

' –  – -End –  – -
```

To see how well the ramp software worked, I had a a Dataq Instruments DI-145 "Data Acquisition Starter Kit" module handy and used it to gather voltage information. The Dataq software has a slow acquisition rate, so I put a 100 msec delay in **Program 16.2**. **Figure 16.12** shows plots of the ramp voltage superimposed on three voltages from the variable resistor. You could use a PropScope or another type of scope to observe the results from your circuit.



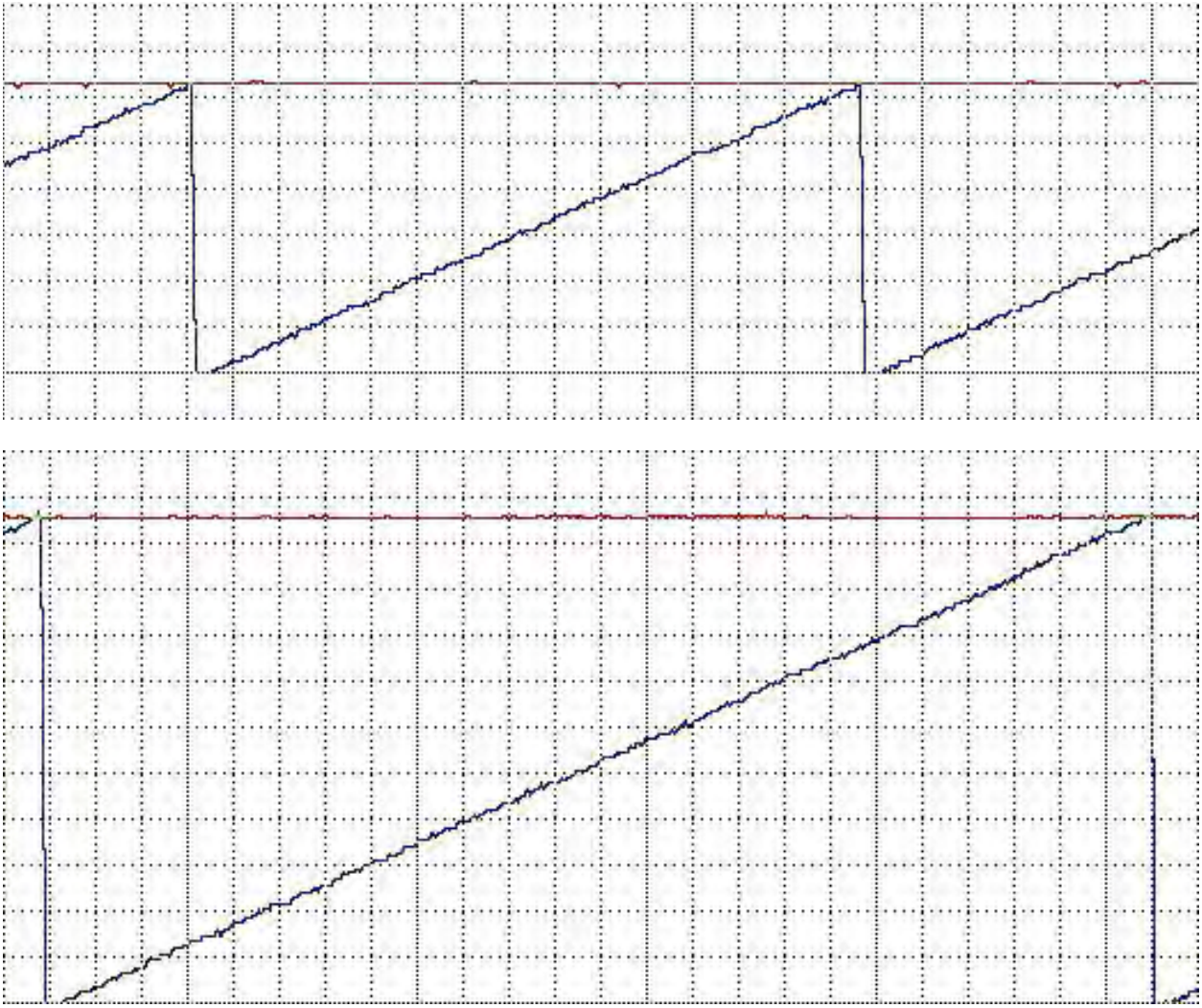


Figure 16.12.
The top graph shows the ramp created by the software for a 0.20-volt input signal. The middle graph shows the ramp created for a 0.90-volt input, and the bottom graph illustrates the ramp for a 1.58-volt input.

To see the DAC value as decimal numbers, run **Program 16.3** and open the PST window. Adjust the variable resistor to produce different voltages. The LEDs should change, as should the value shown in the PST window. What happens to the PST display for low voltages? What about for higher voltages? The low voltages appear in the PST window almost immediately, but as the voltages go up, the analog-to-digital conversions take more time. Why? Find out in the text that follows the **Program 16.3** listing.

Program 16.3.

```
{{
''*********************************************
''*  Program 16.3 Ramp Analog-to-Digital Converter
''*  Author: Jon Titus 11-20-2014 Rev. 1
''*  Copyright 2014
''*  The Propeller creates an increasing voltage
''*  that a comparator IC compares with an unknown
''*  voltage from a variable resistor. When a match
''*  occurs, the ramp stops and the program displays
''*  the DAC's 8-bit value on eight LEDs and puts
''*  the decimal value in the PST window. Then the
```

```
''*  process repeats again and again.
''*  Released under Apache 2 license
''*  Uses Parallax P8X32A Propeller board.
''*********************************************
}}

CON
  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal

VAR
  byte Counter                      'Variable for loop counter
  byte PropPin_SerOut               'Serial out on P30
  byte PropPin_SerIn                'Serial in on P31
  byte PropPin_SerMode              'Serial comm format
  word SerPort_BaudRate             'Serial comm bit rate

OBJ
  delay :   "timing"
  pst   :   "FullDuplexSerial"

'Main program starts here
PUB main
 dira[15..8] := %11111111      'Set P15 – P8 as outputs
 outa[15..8] := %00000000      'Clear DAC
 dira[24] := %1                'DAC latch output
 outa[24] := %1                'Clear DAC
 dira[25] := %0                'P25 = input for comparator
 dira[23..16] := %11111111     'Set LED pins as outputs
 outa[23..16] := 0             'Clear LEDs, turn them off

'Configuration information for serial comms on P8X32A board
 PropPin_SerOut    := 30
 PropPin_SerIn     := 31
 PropPin_SerMode   := 0
 SerPort_BaudRate  := 9600

'Configure Propeller serial port
 pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

'Initialize the DAC
 Counter     := 0                 'Set Counter to zero
 outa[15..8] := Counter           'Reset Counter to 0
 outa[24]    := %0                'Load DAC with data
 outa[24]    := %1

 'Loop that performs continuous analog-to-digital conversions
    repeat
        delay.pause1ms(10)         'Short delay
        if  ina[25] <> 0           'Check comparator for no
                                   'voltage match.
            outa[15..8] := Counter  'No match, so send Counter
                                   'value to DAC P15 – P8
            outa[24]    := 0       'Load DAC
            outa[24]    := 1
            Counter++              'Increment Counter by one
```

```
        else                          'OK, equal voltages
                                      'detected, end of
                                      'conversion!
            outa[23..16] := Counter 'Send Counter to LEDs
            pst.str(String("ADC = "))   'Print text to PST
            pst.dec(Counter)       'Print ADC value as decimal
            pst.tx(13)             'Create a new line in PST
            Counter     := 0       'Reset Counter to 0
            outa[15..8] := Counter 'Reset DAC to 0 volts out
            outa[24]    := %0      'Load DAC
            outa[24]    := %1

' –  – -End –  – -
```

The time difference between a low-voltage and a high-voltage conversion causes a problem seen previously in **Figure 16.12**. If you have an unknown signal and use the ramp ADC-conversion technique you cannot tell the time at which the ramp stops. Also, because the ramp can take a long time to match an unknown voltage, that voltage could have changed from the time the ramp voltage started until it equaled the unknown signal. In science and engineering fields, people need accurate measurements taken at specific and constant intervals. We need a better way to approach analog-to-digital conversions to meet the timing requirements just described. The next steps explain one approach used widely in industrial and consumer products.

Step 7.

Suppose you must find a word in a dictionary with 1600 pages. This dictionary lacks the tabs with letters on them, so you can't immediately turn to a section for words that start with a particular letter. Then how can you quickly find the word "stilted?" You could take the "ramp" approach and go through the dictionary page by page. No one does that!

Here's a faster way to find a word: open the dictionary at the middle. Does the word "stilted" appear in the first half or the last half? OK, the last half. Now split the "last-half" section in half. Does the word appear in the first half or the last half? Then repeat the divisions into halves. It took me 10 of these halving steps to find the page that listed "stilted."

Engineers have applied the same technique, called *successive approximations*, to analog-to-digital conversions. Although this technique differs from the ramp approach, it uses the same hardware. Only the software changes. As an example of a successive-approximation ADC operation, assume you have an 8-bit DAC that has a maximum range of 2.55 volts. That range means the bits shown in **Table 16.2** have the corresponding voltages.

Table 16.2. Volts contributed by individual bits for an 8-bit DAC output.

Bit	Voltage Contributed
D7 (MSB)	1.28
D6	0.64
D5	0.32
D4	0.16
D3	0.08
D2	0.04
D1	0.02
D0 (LSB)	0.01

Given those voltages, the following example takes you through the successive-approximation steps for a 1.54-volt signal connected to the comparator's IN+ pin. You may read the steps below and follow the flowchart shown in **Figure 16.13**. I used boldface-italic letters to indicate the bits that remain fixed as the steps occur:

1. For D7: Apply the value 10000000_2 to the DAC inputs for a 1.28-volt output. This value does not exceed the 1.54-volt unknown signal, so leave the D7 bit as a 1.

2. For D6: Apply the value $\boldsymbol{1}1000000_2$ to the DAC inputs for a 1.92-volt output (1.28 + 0.64 volts). This value exceeds the 1.54-volt unknown signal, return bit D6 to 0.

3. For D5: Apply the value $\boldsymbol{10}100000_2$ to the DAC inputs for a 1.60-volt output (1.28 + 0.32 volts) which also exceeds 1.54 volts, so return bit D5 to a 0.

4. For D4: Apply the value $\boldsymbol{100}10000_2$ to the DAC inputs for a 1.44-volt output (1.28 + 0.16 volts). This value does not exceed the 1.54-volt unknown signal, so leave the D4 bit as a 1.

5. For D3: Apply the value $\boldsymbol{1001}1000_2$ to the DAC inputs for a 1.52-volt output (1.28 + 0.16 + 0.08 volts). This value does not exceed the 1.54-volt unknown signal, so leave the D3 bit as a 1.

6. For D2: Apply the value $\boldsymbol{10011}100_2$ to the DAC inputs for a 1.56-volt output (1.28 + 0.16 + 0.08 + 0.04 volts). This value exceeds the 1.54-volt unknown signal, so return bit D2 to a 0.

7. For D1: Apply the value $\boldsymbol{100110}10_2$ to the DAC inputs for a 1.54-volt output (1.28 + 0.16 + 0.08 + 0.02 volts). This value does not exceed the 1.54-volt unknown signal, so leave the D1 bit as a 1.

8. For D0: Apply the value $\boldsymbol{1001101}1_2$ to the DAC inputs for a 1.55-volt output (1.28 + 0.16 + 0.08 + 0.02 + 0.01 volts). This value exceeds the 1.54-volt unknown signal, so return bit D0 to a 0. The final result equals $\boldsymbol{10011010}_2$. That binary value equals 154_{10}, which nicely corresponds to 1.54 volts.

The eight steps above boil down to this: the software moves a 1 from left to right and checks the comparator output. If the comparator output hasn't changed, a 1 remains at the bit position and the software moves the 1 bit one place toward the LSB.

In real circuits an ADC *does not* produce a direct voltage value. Math in software takes the ADC value and calculates the equivalent voltage based on the voltage contributed by the LSB, or the smallest voltage step. You will learn more about this type of math operation in Experiment 17.

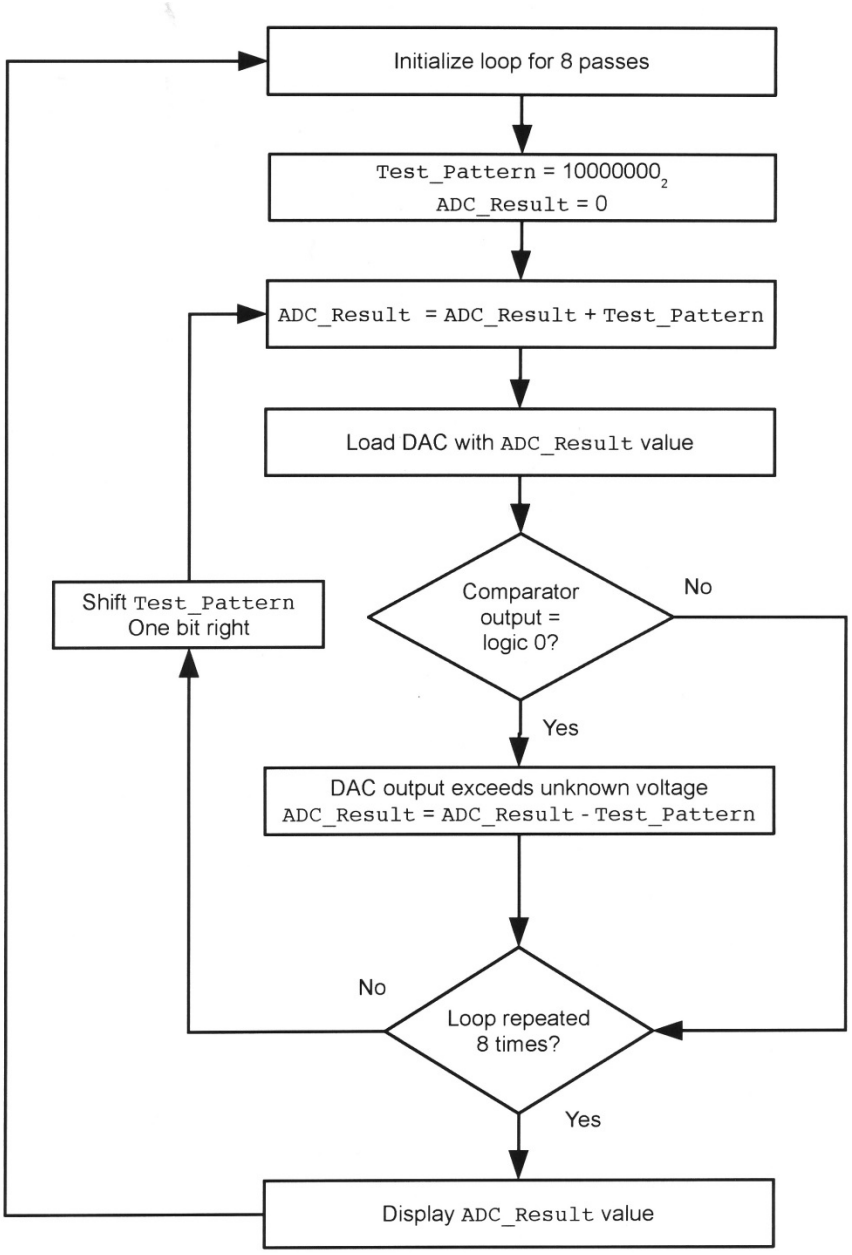


Figure 16.13.
Flowchart for an 8-bit successive-approximation analog-to-digital conversion that uses an 8-bit DAC and a comparator. See the circuit diagram in **Figure 16.10**.

The successive-approximation technique for a *n*-bit converter always takes only *n* periods, and thus all *n*-bit conversions take exactly the same time to complete. (The time of a each test in a successive-approximation depends either on software speed or the speed of the converter's circuits.) If you had used the ramp method with the 2.55-volt DAC and the 1.54-volt unknown signal, the ramp would have required 154 steps. The successive-approximation software needed only eight voltage tests for any voltage.

As an exercise, choose a voltage (0 to 2.55 volts) and fill in the binary bits in **Table 16.3** as you perform a successive-approximation conversion on paper.

Table 16.3. Blank table for successive-approximation exercises.

Bit	D7	D6	D5	D4	D3	D2	D1	D0	
Voltage	1.28	0.64	0.32	0.16	0.08	0.04	0.02	0.01	**Total Volts**
Test 1									
Test 2									
Test 3									
Test 4									
Test 5									
Test 6									
Test 7									
Test 8									

Step 8.

If you have an oscilloscope, you can run **Program 16.4** and observe the successive-approximation tests for a voltage from the 10-kohm variable resistor. This program also displays the 8-bit result on the P8X32A board LEDs. The three plots in **Figure 16.14** shows conversions for 0.210, 0.910, and 1.57 volts, from top to bottom. No matter what the input voltage, all conversions required only eight steps.

Program 16.4.

```
{{
''*********************************************
''*  Program 16.4
''*  Author: Jon Titus 11-20-2014 Rev. 1
''*  Copyright 2014
''*  Successive-approximation analog-to-
''*  conversion. Uses LTC1450L as an 8-bit DAC
''*  with only 8 MSBs used. LSBs grounded.
''*  Results shown on LEDs.
''*  Released under Apache 2 license
''*  Uses Parallax P8X32A Propeller board.
''*********************************************
}}

CON
  _clkmode      = xtal1 + pll16x    'Set MCU clock operation
  _xinfreq      = 5_000_000         'Set for 5 MHz crystal

VAR
  byte    Test_Pattern              'Test pattern for DAC
  byte    ADC_Result                'Result of conversion
byte    Numb_of_Bits                'Number of bits on DAC

OBJ
  delay :   "timing"

'Main program starts here
PUB main
  dira[15..8] := %11111111           'P15 – P8 as output pins
 outa[15..8] := %00000000           'Clear DAC output to 0V
 dira[24] := %1                     'P24 latch DAC data
 outa[24] := %1                     'P24 = logic 1
 dira[25] := %0                     'Set pin P25 as an input pin
```

```
                                    'to test comparator output.
 dira[23..16] := %11111111          'Set LED pins as outputs
 dira[26] := 1                      'Optional output to trigger
                                    'a scope trace
 Numb_of_Bits := 8                  '8-bit DAC

 'Code to initialize the DAC for each new conversion
 repeat
    Test_Pattern     := %10000000       'Initial test pattern
    ADC_Result  := 0                    'Reset start value for
                                        'the DAC
    outa[15..8] := 0                    'Clear DAC to 0V output
    outa[24]    := %0                   'Load DAC
    outa[24]    := %1
        repeat Numb_of_Bits             'One pass through loop
                                        'for each bit position
            delay.pause1ms(1000)          '1 sec delay

            'Add the test pattern to the ADC_Result value here.
            'This action sets up a DAC value to test against the
            'unknown voltage at the comparator.

            ADC_Result := ADC_Result + Test_Pattern

            outa[15..8] := ADC_Result   'Send DAC ADC_result
            outa[24]    := %0           'Load DAC
            outa[24]    := %1

            'Now, test comparator output for a voltage match or
            'for a test voltage that exceeds the unknown voltage
            'at the comparator input. If DAC voltage is too
            'high, subtract the test pattern to remove the bit
            'just tested and reset the tested bit back to 0.
            'No need to send this new ADC_Result to the DAC.
            'It gets updated as the next pass through the loop
            'starts.

            if ina[25] <> 1
                ADC_Result := ADC_Result - Test_Pattern

            'Shift the test bit one position toward the LSB so
            'we can add the next smallest voltage increment to
            'the DAC's test voltage output.

            Test_Pattern := Test_Pattern >> 1

        'Conversion done, display result on LEDs

        outa[23..16] := ADC_Result

' –  – -end –  – -
```

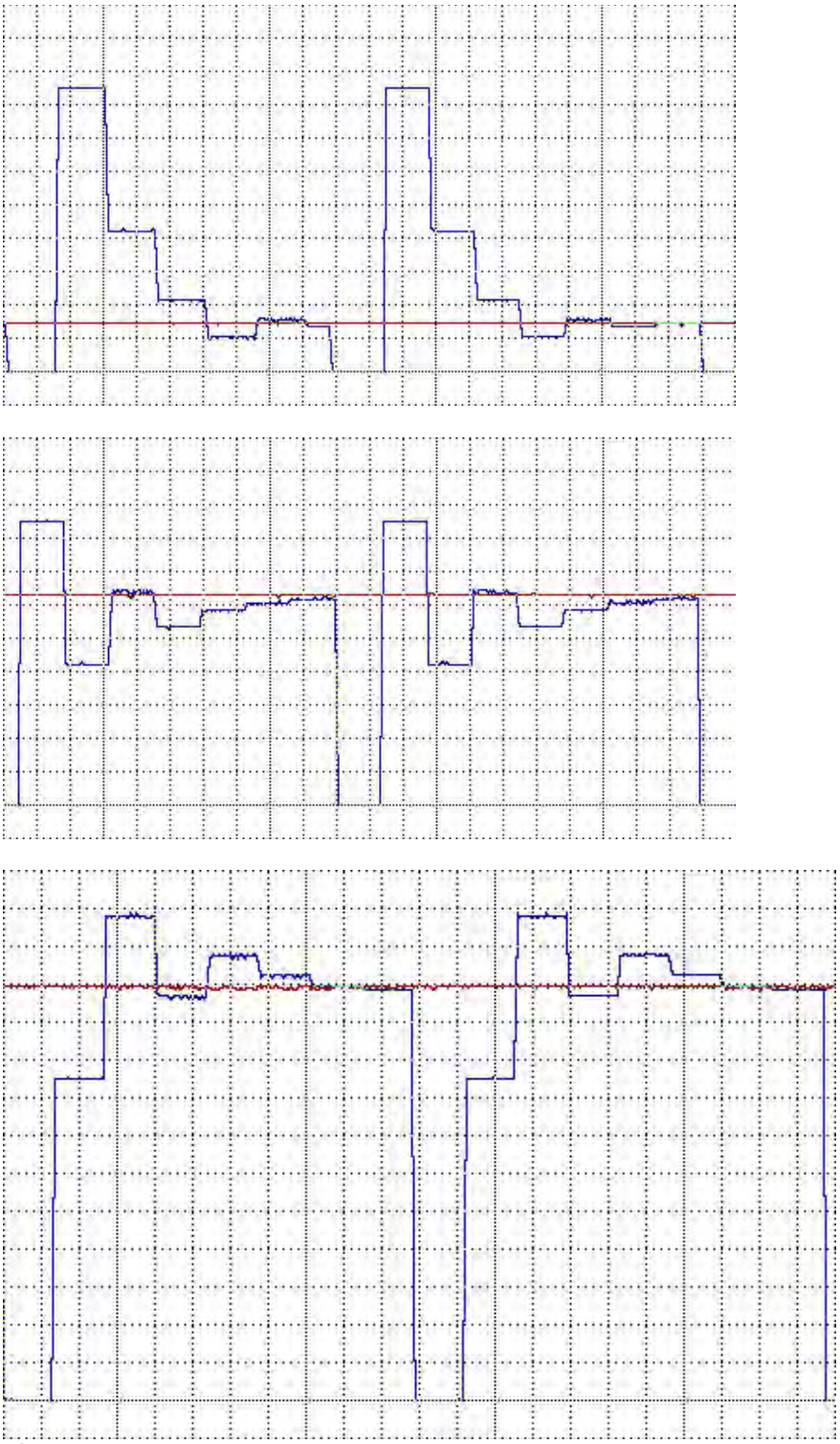


Figure 16.14.
Successive-approximation conversions for 0.210, 0.910, and 1.57 volts, from top to bottom. The solid line shows the constant voltage from the 10-kohm variable resistor. Note how the DAC voltage "homes in" on the stable voltage. All conversions take the same time, regardless of voltage. A commercial 8-bit successive-approximation ADC will perform a conversion in a few microseconds.

People often call a successive-approximation ADC a "SAR" ADC. Years ago, companies manufactured ICs called successive-approximation registers, or SARs, that performed the comparator tests and bit shifts. The name has stuck, so an SAR ADC is simply one that uses the successive-approximation technique. Designers like SAR ADCs because they make conversions quickly, are easy to work with, and don't cost much. Many MCUs come with an SAR ADC built in. (Early successive-approximation registers required an external DAC, comparator, and voltage reference.)

Step 9.

Please turn off power to the breadboard and remove the USB cable from the P8X32A board. Carefully remove the LTC1450L and LM339 ICs, their connecting wires, and associated components from the breadboard.

Commercial ADC ICs offer many advantages over the do-it-yourself circuit constructed and tested in previous steps. Examples of commercial ADCs include the Microchip Technology MCP3202 and the Texas Instruments ADC0831. Both devices use serial communications to receive commands from, and send data to, MCUs. The ADC0831 has an 8-bit resolution and the MCP3203 has a 12-bit resolution.

The MCP3202 diagram in **Figure 16.15** shows the ADC's internal circuit functions in block-diagram form, and a pin diagram for the 8-pin dual inline package. The block diagram includes the familiar comparator, DAC, and 12-bit successive-approximation register (SAR), all built on one silicon chip.

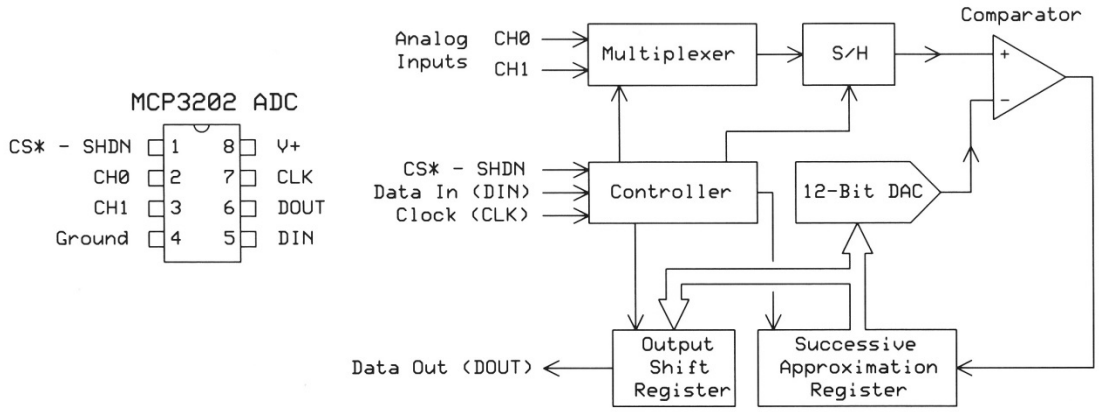


Figure 16.15.
Pin configuration for an MCP3202 12-bit ADC (left) and the block diagram of the IC's internal functional blocks. The S/H block represents a sample-and-hold circuit.

The block diagram includes an input-channel multiplexer, so you can measure two separate analog signals – one at a time – selected with software commands. The sample-and-hold block has an important function. It samples the CH0 or CH1 input and takes a "snapshot" of the analog signal so the ADC can convert a steady signal at a specific time. An internal capacitor briefly stores enough charge so the ADC can perform an accurate conversion. For more information about sample-and-hold circuits and how they work, see the references at the end of this experiment.

Carefully place an MCP3202 ADC IC in your breadboard and make the connections shown in **Figure 16.16**. You will use two 10-kohm variable resistors on the breadboard to provide signals for the two ADC analog inputs, CH0 (pin 2) and CH1 (pin 3). The MCP3202 IC can operate with a voltage from 2.7 to as high as 5.5 volts, so the 3.3-volt power from the Propeller P8X32A board will suffice. The voltage applied to the MCP3202 power input also sets the reference voltage for the ADC. This experiment uses the 3.3-volt output on a Propeller P8X32A board to power an MCP3202 ADC, so the two analog inputs can accept and properly convert a voltage between 0 and 3.3 volts. As always, ensure you have a connection between the breadboard ground bus and the ground pin on the P8X32A board I/O connector.

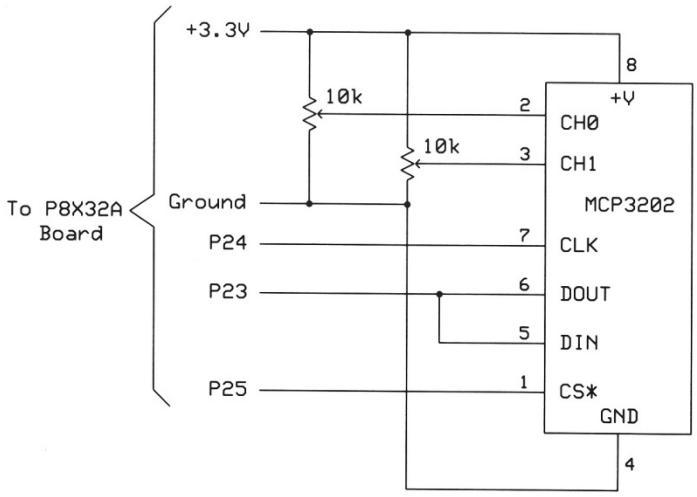


Figure 16.16.
Circuit diagram for an MCP3202 2-input ADC connected to a Propeller P8X32A MCU board. You may use the 3.3 volts provided on the MCU board or a separate 3.3-volt power supply. Either case requires a ground between a power source (if any), this circuit, and the P8X32A board.

Step 10.

An object contributed to the Propeller Object Exchange Library, OBEX, provides an easy way to test the MCP3202 ADC. The community of Propeller programmers, engineers, hobbyists, and experimenters graciously contributes Spin-object files others can use and modify. The Experiment 16 program folder includes the object, MCP3202.spin, created by Chip Gracey at Parallax, with modifications and updates from John Abshier and "Kuroneko." This object provides the code necessary to communicate with an MCP3202 ADC.

John Abshier also contributed an MCP3202 demonstration program that uses two variable resistors – one on channel 0 and the second on channel 1 – to provide varying voltages to the ADC. I modified this object slightly to use the FullDuplexSerial.spin library of serial-communication operations, and I included a short delay to make the information easier to see in the PST window. New comments and new variable names help clarify software operations.

From within the Propeller Tool window, open the **Program 16.5** file that will communicate with the MCP2303 and display the decimal equivalent of the binary value for the CH0 and CH1 voltage inputs.

Program 16.5

```
''*****************************************************
''*  Program 16.5, Analog-to-Digital Converter Test
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*  with different pin numbers and potentiometers
''*  instead of joystick inputs.
''*  Modified by Jon Titus, 09-28-2013 Rev. 1
''*  Copyright 2014
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays the decimal value for Channel 0
''*  and Channel 1.
''*****************************************************
OBJ
  pst       : "FullDuplexSerial"    'Serial comm objects here
  MCP3202   : "MCP3202"             'MCP3202 driver objects
  delay     : "timing"              'timing file
```

```
CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

clock_pin         = 24              'MCP3202 pin assignments  data_pin
= 23
chip_select_pin   = 25

VAR
  long  CH0                    'ADC Channel-0 data
  long  CH1                    'ADC Channel-1 data
  byte PropPin_SerOut          'Serial out
  byte PropPin_SerIn           'Serial in
  byte PropPin_SerMode         'Serial comm format
  word SerPort_BaudRate        'Serial comm bit rate

'Main program starts here
PUB Main
  PropPin_SerOut    := 30      'P30 for transmit to host PC
  PropPin_SerIn     := 31       'P31 for receive from host PC
  PropPin_SerMode   := 0        'Invert RX mode
  SerPort_BaudRate  := 9600     'Baud rate

  'Serial-communication start-up operation
  pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

  'MCP3202 ADC start-up configuration
  'Enable Channels 0 and 1 for single-ended analog inputs
  'See MCP3202 datasheet for configuration options

  MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

repeat
  pst.tx(1)                           'Move PST cursor to home
  pst.str(string("MCP3202 12-bit A/D"))   'Display this string
  pst.Tx(13)                          'New line
  pst.str(string("=================="))'Display this string
  pst.Tx(13)                          'New line

  'Get digital value for analog signal on Channel 0
  CH0 := MCP3202.in(0)
  pst.str(string("Channel 0: "))      'Display this string
  pst.tx(11)                          'Clear previous Channel 0
                                      'value from PST window
  pst.dec(CH0)                        'Display Channel-0 data
                                      'as a decimal value
  pst.Tx(13)                          'New line

  'Get digital value for analog signal on Channel 1
  CH1 := MCP3202.in(1)
  pst.str(string("Channel 1: "))      'Display this string
  pst.tx(11)                          'Clear previous Channel 1
                                      'value from PST window
  pst.dec(CH1)                        'Display Channel-1 data
                                      'as a decimal value
  pst.Tx(13)                          'New line
```

```
  delay.pause1ms(100)                   'Short delay

  ' –  – -end –  – -
```

Set both variable resistors about midway between their end points so they provide about 1.6 volts for the ADC CH0 and CH1 inputs. Run **Program 16.5** and then open the Parallax Serial Terminal window (F12). Clear the terminal window and click the Enable button in the bottom-right corner of this window. What do you see? The program should display the following information in the PST window, although my values will differ from yours. Some people use the abbreviation A/D for analog-to-digital converter. I prefer ADC.

```
MPC3202 12-bit A/D
==================
Channel 0: 2222
Channel 1: 2195
```

Program 16.5 causes the MCP3202 ADC to send new measurements frequently, so adjust the variable resistors and watch the values change. Remember you will see the *decimal* equivalent of the 12-bit binary ADC results. What maximum and minimum value do you get from each variable resistor? My readings ranged from 2 to 4091 on Channel 0, and from 2 to 4095 on Channel 1. I swapped the variable resistors and got the same results, so these resistances vary slightly from one another – a common occurrence for electronic components.

For a 12-bit ADC, the output values should range from 0 to 2^n - 1, or 0 to 4095. Those values correspond to input voltages of 0 and 3.3 volts respectively. What would you expect from the ADC when you turn their control about half way between its end points? Does that value come close to the values I observed in the PST window and shown above? Find the answers at the end of this experiment.

Experiment with the variable-resistor settings. Again, the PST window displays decimal values, not voltages. How would you convert these values to voltages? Find the answers at the end of this experiment.

Step 11.

The next experiment introduces a temperature sensor and a light sensor and shows you how to connect them to an MCP3202 ADC IC and how to take the 12-bit values and convert them to useful information. Leave the breadboard circuit set up. If you plan to take a break for more than a few minutes, disconnect the P8X32A Propeller board from its USB cable. If you used a separate power supply for the MCP3202 ADC circuits, turn it off.

Step 12.

Before this experiment ends, I want you to understand the difference between *accuracy* and *resolution* in DACs and ADCs. Descriptions of the 12-bit Microchip Technology MCP3202 and the 8-bit Texas Instruments ADC0831 ADCs note their *resolution*, or the number of bits they can provide for an unknown signal. So you will always get a 12-bit result from an MCP3202 and an 8-bit result from an ADC0831.

But the resolution number does not indicate how accurately the 8- or 12-bit values represents an input signal. The data sheet for an MCP3202 specifies an accuracy within about ±1 LSB. The accuracy for an ADC0831 goes from ±0.5 LSB to ±1 LSB, depending on the model selected. So, although an ADC might have a 12-bit resolution, expect accuracy only within ±1 LSB *at best*. Remember, the LSB corresponds to the smallest voltage step a converter can make. For a 12-bit ADC with a 2.5-volt reference (2.5-volt full-scale input), the LSB comes to 2.5 volts/4095 steps, or 0.611 mV.

The design of ADC circuits takes a lot of time, knowledge, and experience. For more information about accuracy and resolution in converters, see the references that follow.

Please continue to Experiment 17.

References

– -, "Analog to Digital Conversion," Measurement Computing. http://www.mccdaq.com/PDFs/specs/Analog-to-Digital.pdf. Good section on accuracy and resolution as well as helpful information about different ADC techniques.

– -, "AN-74 LM139/LM239/LM339 A Quad of Independently Functioning Comparators," Texas Instruments. 2013. http://www.ti.com/lit/an/snoa654a/snoa654a.pdf.

– -, LM339 comparator data sheet, Texas Instruments. http://www.ti.com/lit/ds/symlink/lm339.pdf.

– -, "Sample and Hold," at Wikipedia. http://en.wikipedia.org/wiki/Sample_and_hold.

– -, "Sample-and-Hold Amplifiers," MT-009 Tutorial, Analog Devices. 2008. http://www.analog.com/static/imported-files/tutorials/MT-090.pdf.

Bryant, James, "Just How Accurate was William Tell, Anyway?" Analog Devices, 2007. http://www.analog.com/static/imported-files/rarely_asked_questions/RAQ_ADCAccuracy.pdf. More information about accuracy and resolution.

Kester, Walt, "Data Converter History," Analog Devices. http://www.analog.com/library/analogdialogue/archives/39-06/chapter%201%20data%20converter%20history%20f.pdf.

Answers

Experiment 16, Step 1:

Because this circuit uses only the eight most-significant bits (MSBs) of the LTC1450L DAC, how many discrete voltages will the DAC produce?

An 8-bit DAC will produce 2^n voltages (one at a time), where *n* equals the number of bits. Thus, $2^8 = 256$.

Assume the LTC1450L uses its 2.50-volt internal reference. What voltage difference will you observe between adjacent voltage steps?

By using the 2.50-volt internal reference, the LTC1450L DAC has a full-scale range of output voltages from 0.00 to 2.50 volts.

The 12-bit DAC can produce 2^n voltages, where *n* equals the number of bits. In this case, 2^{12}, or 4096, voltages. But the 12-bit DAC has only 2^n - 1, or 4095, steps over its output range. Assume you have set the LTC1450L with a 2.5-volt reference. The voltage per step amounts to:

$2.50\ volts\ /\ 4095\ steps\ =\ 0.0006105\ V,$ or $0.611\ mV\ per\ step.$

But because the four LSBs always supply a logic-0 to the DAC circuit, the 12-bit values applied to the DAC change increase by increments of 16, or 10000_2 from ***0000_0000***0000 to ***0000_0001***0000 to ***0000_0010***0000, and so on. The four LSBs do not change, so the larger voltage steps for the LTC1450L when used as an 8-bit DAC amount to:

$0.611\ mV\ per\ step\ *\ 16\ steps\ =\ 9.77\ mV/step$ for 8-bit DAC, D11-D4 data inputs, with D3-D0 equal to logic-0.

When you operate the LTC1450L ADC as an 8-bit device and ground the D3-D0 inputs, the maximum binary value you can apply to the DAC comes to: ***1111_1111***0000_2. It cannot reach $1111\_11111111_2$, the maximum possible code for a 12-bit input. As a result, the maximum output voltage cannot reach 2.5 volts. Instead, it reaches the voltage for the 12-bit code ***1111_1111***0000_2 (and produces an output of 2.49 volts. For this experiment, that value comes close enough to 2.50 volts to avoid any concerns.

Experiment 16, Step 9:

What maximum and minimum value do you get from each variable resistor?

A voltage half way between the 3.3-volt reference and ground should cause the ADC to produce the value $0111\_11111111_2$, or 2047_{10}. The values I measured, 2222 and 2195, come close enough for "eyeball" adjustments with a small screwdriver.

Keep in mind all electronic components have a tolerance, and specifications for the 10-kohm variable resistors I use list a ±10% tolerance. That means even if you carefully adjust a 10-kohm variable-resistor control exactly half way between end points (5 kohms), the resistance there could vary by ±500 ohms.

How would you convert a decimal value to an input-voltage value for this ADC?

To convert a 12-bit value – displayed as a decimal value in the PST window – to a voltage, take the ADC's full-scale voltage and divide it by 2^n-1. Then multiply the result by the decimal value from the ADC. (I assume the full-scale voltage equals the reference voltage for the ADC.)

For a decimal reading of 1490 from the MCP3202 ADC, the equivalent voltage comes to:

For a 12-bit ADC, 2^{12} - 1 = 4095

$$Voltage\ =\ 1490\ *\ (3.3\ volts\ /\ 4095)$$

$$Voltage\ =\ 1490\ *\ 0.000805\ volts$$

$$Voltage\ =\ 1.20\ volts$$

Keep in mind the Propeller would need additional software to perform math operations with values that have decimal fractions.

Experiment No. 17 – Analog-to-Digital Conversion, Part 2

Abstract

In this experiment you will connect a temperature sensor and a photoresistor to the MCP3202 analog-to-digital converter (ADC) introduced and used in Experiment 16. You will learn how to convert the 12-bit values from the ADC into useful information. A sensor such as a photoresistor has nonlinear characteristics that challenge engineers and programmers. That means they don't produce a resistance directly proportional to illumination intensity. I'll explain several ways to address the problems of nonlinear sensors when an MCU has limited built-in math capabilities so you'll have ideas to pursue on your own. To completely understand the photoresistor portion of this experiment, you should have some familiarity with logarithms, although you can work through most parts with basic algebra.

Keywords

Analog-to-digital-converter, ADC, MCP3202, photoresistor, temperature sensor, cadmium-sulfide, logarithms, LM335, Celsius, Kelvin, lux, spreadsheet, log-log plot, array, look-up table, Zener diode

Requirements

(1) - Solderless breadboard
(1) - Solderless breadboard (optional)
(1) - Propeller P8X32A microcontroller board
(1) - Digital voltmeter or digital multimeter (DMM)
(1) - USB cable
(1) - 10-kohm, 1/4-watt resistor, 5% (brown-black-orange)
(1) - 100-kohm, 1/4-watt resistor, 5% (brown-black-yellow)
(1) - 10-kohm variable resistor
(1) - LM335 Temperature Sensor IC, TO-92 package, Texas Instruments
(1) - MCP3202 12-bit ADC, Microchip Technology
(1) - Photoresistor, cadmium-sulfide, Advanced Photonix PDV-P9003-1 (see text)
(1) - Thermometer, approximate range; -10° to +100°F (-20° to +35°C)
(1) - 9-Volt battery
(1) - 9-Volt battery clip with wire leads (Jameco 216452 or equivalent)
(1) - Power supply for 3.3 volts (optional)

Introduction

After you have adjusted the two variable resistors in Experiment 16 and watched the Channel-0 and Channel-1 values change in the PST window, you probably wondered, "What comes next?" You can use an ADC such as the Microchip MCP3202 to measure the output from almost any sensor or device that produces a voltage. In this experiment you will learn how to use an inexpensive LM335 temperature sensor and a small cadmium sulfide (CdS) photoresistor to measure temperatures and illumination intensity respectively. You should have completed Experiment 16 and have an MCP3202 ADC in a breadboard and connected to a P8X32A microcontroller board.

Measurements with an LM335 Temperature Sensor IC

In Experiment 12 you worked with a self-contained temperature sensor, the DS1620, which used SPI communications to transfer information to and from an MCU. Here you will work with a different type of temperature sensor, the LM335 IC, that produces a voltage proportional to ambient temperature. Instead of

using the Celsius or Fahrenheit scales, this sensor reports temperatures in Kelvin (not degrees Kelvin). The Kelvin scale starts at absolute 0 and increases from there in increments equal to those on the Celsius scale. A Kelvin thermometer reads 273 K at 0 °C (32 °F) and 293 K at 20°C (68 °F), for example.

The voltage produced by an LM335 sensor increases or decreases linearly at 10 mV per K. Thus at 20 °C (about room temperature), the sensor would produce an output of 2.93 volts (293 K * 10 mV/K) . And at 0 °C you would measure an output of 2.73 volts (273 K * 10 mV/K).

The LM335 sensor belongs to a family of similarly numbered sensors that have slightly different temperature ranges (**Table 17.1**), but these sensors have the same electrical connections, so you can substitute one for another as you like. According to the LM335 data sheet, operating a sensor in the Intermittent Range for long periods might decrease its life. The LM335 has an accuracy of about 1 or 2 K.

Table 17.1. Continuous and intermittent temperature ranges for LM335-type temperature sensors.

	Continuous Range		Intermittent Range	
Device Types	Min (°C)	Max (°C)	Min (°C)	Max (°C)
LM135, LM135A	-55	150	150	200
LM235, LM235A	-40	125	125	150
LM335, LM335A	-40	100	100	125

The internal sensor circuit operates like a *zener diode* that passes current in one direction, much like an LED. But zener diodes also have a "breakdown voltage," above which they also will conduct current in the *opposite* direction. In an LM335-type sensors, that breakdown voltage changes in direct proportion to temperature (10 mV/K), which makes for a simple sensor.

The diagram in **Figure 17.1** shows a *top view* of the pin connections for an LM335 in an 8-pin DIP and in a TO-92 plastic transistor package. I used the latter 3-lead device in this experiment. The 8-pin DIP will work just as well. The LM335 provides a positive and a negative connection and an adjustment input. The latter lets you calibrate the sensor for accurate temperature vs. voltage results.

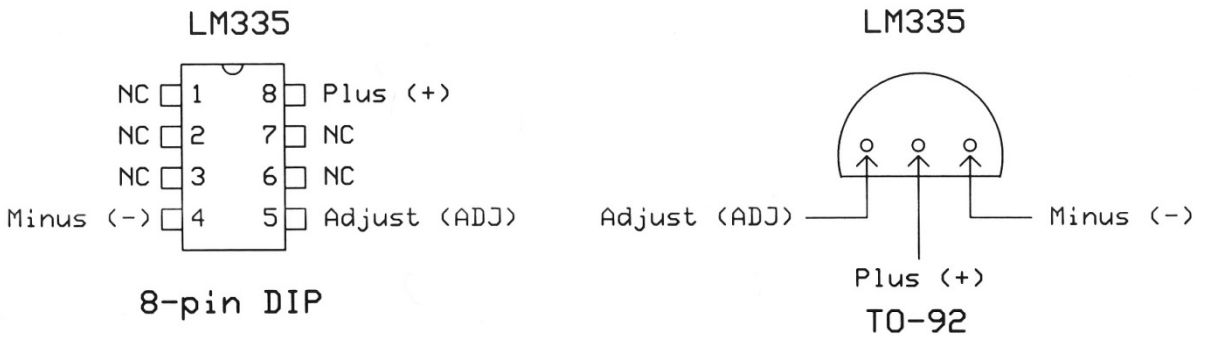


Figure 17.1.
A view from the top shows pin connections for an LM335 temperature sensor in an 8-pin dual inline package (DIP) and in a 3-lead TO-92 plastic transistor package. The latter package provides 0.6-inch (15-mm) wire leads.

I particularly like the LM335 sensor in a TO-92 plastic package because it lets people easily waterproof the sensor for use in an area of high humidity or outdoors. I have dipped sensors in clear nail polish to provide waterproofing. Several dips usually provides a good protective coating. Or, use Plasti-Dip synthetic-rubber coating. Non-conductive coatings also insulate the sensor thermally, so responses to temperature changes occur more slowly. The LM335 family of sensors can operate hundreds or thousands of feet away from a

measurement instrument or ADC, so you could use them for remote-monitoring devices. For more information about the LM135, LM235, and LM335 sensors, refer to the Texas Instruments data sheet listed in the Reference section.

I tested an LM335 with several supply voltages and it worked well from 5 to 15 volts. In the following steps, I recommend you use a 9-volt power supply, or a 9-volt "transistor-radio" alkaline battery; the type with the snaps on the top.

Step 1.

You should have your circuit set up from Experiment 16. If not already off, turn off power to your breadboard and remove the USB cable from your P8X32A board. Remove from your breadboard one of the 10-kohm variable resistors that connects to the MCP3202. Place an LM335 temperature sensor in your breadboard and connect it as shown in **Figure 17.2**. Use the remaining 10-kohm variable resistor in this circuit. I placed my LM335 sensor on a separate breadboard so I would not inadvertently cross connect the 9-volt battery with the MCP3202 ADC or the P8X32A board. DO NOT connect the LM335 circuit to your P8X32A board or to the MCP3202 ADC circuit.

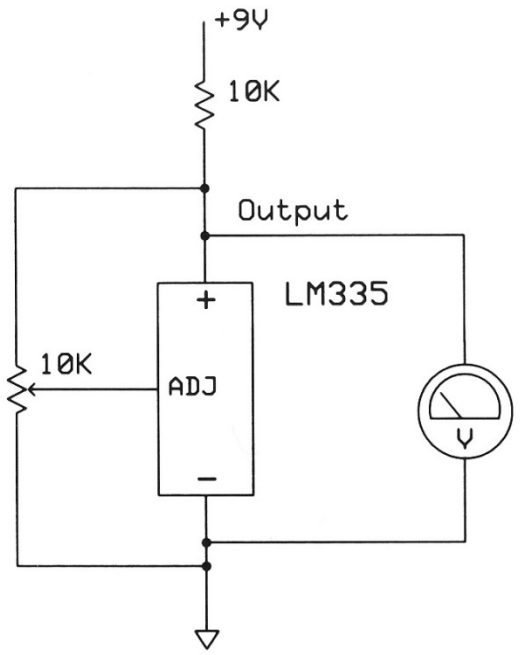


Figure 17.2.
Circuit diagram for an LM335 temperature sensor with a 10-kohm variable resistor used to calibrate the temperature setting. The 10-kohm fixed resistor limits current through the sensor.

Connect your voltmeter as shown in **Figure 17.2** and turn on your 9-volt supply, or connect your 9-volt battery to power the LM335 sensor. The voltmeter should show a voltage between about 2.9 and 3.0 volts. You might observe a voltage above or below these values, but don't worry. Let the powered sensor circuit sit for a few minutes so it can come to room temperature. Measure your room temperature with a thermometer in degrees Celsius – or convert a Fahrenheit temperature to Celsius – and note it in the space below. Add 273 to your Celsius temperature, and note that temperature below in units of Kelvins:

My room temperature: _____°C

My room temperature: _____K (°C + 273)

My lab temperature measured 20 °C, or 293 K. (No degree symbol for the unit Kelvin.) Once you have your room's Kelvin temperature, carefully adjust the 10-kohm variable resistor until the voltage on your meter equals the Kelvin temperature in your work area divided by 100. In my lab: 2.93 volts or 293 K.

As I adjusted my 10-kohm resistor, I got readings from 2.904 to 2.920. You might not get an exact correspondence of the voltage with the ambient temperature in Kelvins. You just want to get close. Adjusting the variable resistor lets you calibrate the voltage output so it will accurately "track" the sensor's temperature. The sensor provides a linear response, which means as the temperature changes by ±1 K, the voltage output always changes by ±10 mV. You don't need any math to convert a voltage into a temperature. A reading of 2.80 volts, for example, equals a temperature of 280 K.

Step 2.

Use a hair dryer or another heat source to slowly warm the LM335 sensor and watch the output voltage, which should increase. Turn off the heat and wait a few minutes. Then use some coolant spray, or a piece of double-bagged ice, to cool the sensor. The voltage should decrease. After you test the sensor, disconnect the 9-volt power source and go on to the following step.

Step 3.

Now you will combine the LM335 sensor and the MCP3202 analog-to-digital converter. Connect the LM335 voltage output to the MCP3202 CH1 input (pin 3) as shown in **Figure 17.3**. Keep your voltmeter connected to the LM335 sensor output. As always, you must have a good ground connection between the MCP3202 ADC ground and the LM335 ground, also shown in the diagram. Your 9-volt power source should connect only to the LM335 sensor through the 10-kohm resistor. If you mistakenly connect 9-volt power to the MCP3202 or the P8X32A board, you will damage them.

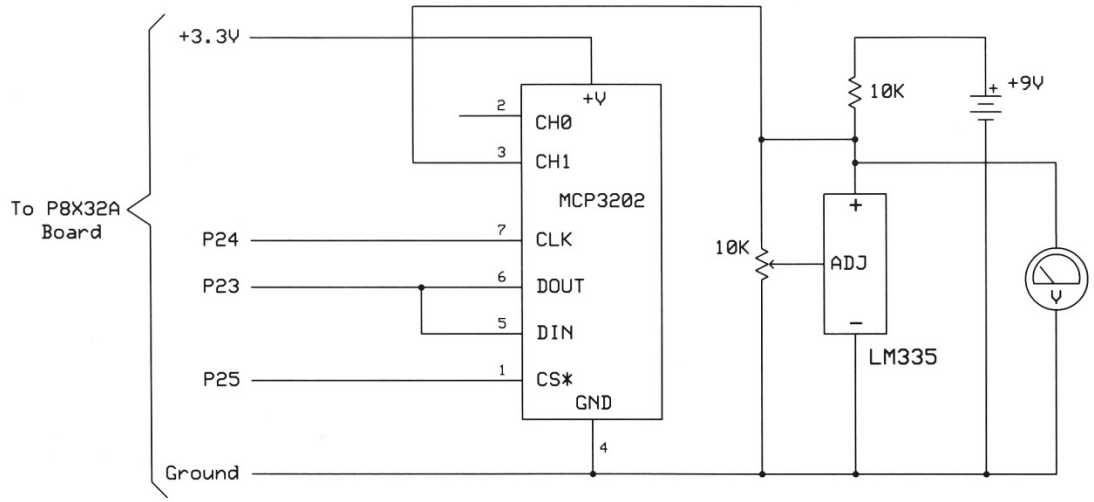


Figure 17.3.
Circuit diagram for an MCP3202 ADC connected to an LM335 temperature sensor and to a P8X32A Propeller MCU board. The meter lets you check the voltage put out by the LM335 sensor.

After you recheck your wiring, connect the USB cable to your Propeller board, connect the 9-volt power source to the LM335 sensor via the 10-kohm resistor, and run **Program 17.1**. Open the PST window and click the Enable button to see information received from the P8X32A board. You should see a value for Channel 0 (disconnected) and another value for Channel 1, the LM335 temperature-sensor voltage. Let the temperature sensor operate for a few minutes. What is your room's temperature? I still measured 20 °C in my lab area.

Program 17.1.

```
''*****************************************************
''*  Program 17.1, Analog-to-Digital Converter Test
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*  with different pin numbers and potentiometers
''*  instead of joystick inputs.
''*  Modified by Jon Titus, 11-22-2014 Rev. 1
''*  Copyright 2014
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays the decimal value for Channel 0
''*  and Channel 1.
''*****************************************************
OBJ
  pst       : "FullDuplexSerial"    'serial comm objects
  MCP3202   : "MCP3202"             'MCP3202 driver objects
  delay     : "timing"              'timing.spin file

CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock
  _XINFREQ      = 5_000_000         '5MHz Crystal

  clock_pin         = 24            'MCP3202 Pin assignments
  data_pin          = 23
  chip_select_pin   = 25

VAR
  long  CH0                         'Channel-0 data
  long  CH1                         'Channel-1 data

  byte PropPin_SerOut               'P8X32A pin P30
  byte PropPin_SerIn                'P8X32A pin P31
  byte PropPin_SerMode              'Serial comm format
  word SerPort_BaudRate             'Comm bit rate

PUB Main
 PropPin_SerOut    := 30            'P30 for USB transmit
 PropPin_SerIn     := 31            'P31 for USB receive
 PropPin_SerMode   := 0             'Invert RX mode
 SerPort_BaudRate  := 9600          'Set baud rate

 'Serial-communication start-up operation
 pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

 'MCP3202 ADC start-up configuration
 'Enable Channels 0 and 1 for single-ended analog inputs
 'See MCP3202 datasheet for configuration options
 MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

repeat
  pst.tx(1)                                 'Move cursor home
  pst.str(string("MCP3202 12-bit A/D"))     'Display this string
  pst.Tx(13)                                'New line
  pst.str(string("=================="))     'Display this string
  pst.Tx(13)                                'New line

  CH0 := MCP3202.in(0)                 'Get digital value for
```

```
                                    'signal on Channel 0
  pst.str(string("Channel 0: "))    'Display this string
  pst.tx(11)                        'Clear old Channel 0 data
  pst.dec(CH0)                      'Display Channel-0 data
  pst.Tx(13)                        'New line

  CH1 := MCP3202.in(1)              'Get digital value for
                                          'signal on Channel 1
  pst.str(string("Channel 1: "))    'Display this string
  pst.tx(11)                        'Clear old Channel 1 data
  pst.dec(CH1)                      'Display Channel-1 data
  pst.Tx(13)                        'New line

  delay.pause1ms(100)               'Cause a short delay

  ' –  – -end –  – -
```

Step 4.
Does the voltage on your meter indicate about the same temperature as your nearby thermometer? For me, the temperature in Kelvins comes to 293 K (20 °C + 273 K) and the measured voltage comes close to 2.93 volts. If necessary, adjust the 10-kohm variable resistor near the LM335 sensor so the output voltage comes close to your ambient temperature in Kelvins.

Now look at the values displayed in the PST window for the MCP3202's CH1 input. Do these decimal values match your work-area temperature expressed in Kelvin? In my case, the PST window showed Channel 1 values between 3648 and 3653, with an average of 3651. But those readings don't look like what I expected for 293 K. Can you determine why we see such a difference between the voltmeter reading and the values shown in the PST window? Think about this difference for a few minutes before you go to the next step.

Step 5.
Remember, the voltmeter measures volts over a selected range, say 0 to 10 volts. The MCP3202 ADC measures the sensor's output voltage, but it reports a corresponding 12-bit *binary value between 0_{10} and 4095_{10}*. You apply a voltage to the ADC CH-1 input and get an ADC "count" rather than a voltage value. To get the actual voltage, we need a bit of math that takes the ratio between the value reported by the ADC and the ADC maximum value (4095) and multiplies it by the maximum ADC-input voltage (3.30 volts). For my average ADC output value 3651 I used the following equation and a calculator to get:

$$(3.30 \text{ volts} / 4095 \text{ ADC max value}) * 3651 \text{ ADC value} = 2.94 \text{ volts}$$

The 3.30 volts corresponds to the ADC input signal that results in an ADC output of 4095. The calculated result above comes very close to the voltmeter reading: 2.93 volts. Before you go on, think about this for a bit: What does the expression: 3.30 volts / 4095 ADC max value give us?

That calculation provides the voltage per step for the ADC output. In this case, we have 3.30 volts times 2^{12} - 1 (4095), which equals 0.806 mV/step. The ADC produces the number of steps between 0 volts and the unknown signal so we can calculate the signal's voltage.

Take five or six ADC values from your PST window – use the PST Pause button to temporarily suspend ADC-value updates – and average them. Then apply the math steps shown above. Did your calculated voltage agree with the ambient temperature in Kelvins? It should. If it does not, recheck the voltmeter readings from your sensor and if necessary, recalibrate the sensor output as described in the first paragraph of Step 4.

Step 6.

Unfortunately, a Propeller MCU cannot handle numbers with decimal fractions such as 3.30 volts or 0.806 volts/step. You could find appropriate math software in the Parallax Object Exchange (OBEX) to handle decimal fractions, but a bit of algebra lets us *eliminate* decimal fractions in the ADC-count- to-voltage-conversion math shown above. Here I rearranged the equation:

$$(3.30\ volts\ /\ 4095\ ADC\ max\ value)\ *\ 3651\ ADC\ value\ =\ 2.94\ volts$$

to

$$3.3\ volts\ *\ 3651\ value\ from\ ADC\ /\ 4095\ value\ of\ ADC\ max\ =\ 2.94\ volts$$

Although we know 2.94 V equals a sensor temperature of 294 K, we need an equation that gives us a *temperature* and not a voltage. To make the conversion, we use the ratio of 10 mV/1 K, or the inverse 1 K/10 mV. By using basic algebra the generalized conversion equation becomes:

$$3.3\ V\ *\ \frac{ADC\ Output}{ADC\ Max}\ *\ \frac{1000\ mV}{1\ V}\ *\ \frac{1\ K}{10\ mV}\ =\ temp\ (K)$$

The units of volts (V) and millivolts (mV) in the numerators and denominators cancel one another:

$$3.3\ \cancel{V}\ *\ \frac{ADC\ Output}{ADC\ Max}\ *\ \frac{1000\ \cancel{mV}}{\cancel{1\ V}}\ *\ \frac{1\ K}{10\ \cancel{mV}}\ =\ temp\ (K)$$

And the equation becomes:

$$3.3\ *\ \frac{ADC\ Output}{ADC\ Max}\ *\ \frac{1000}{10}\ *\ 1\ K\ =\ temp\ (K)$$

Rearrange to:

$$3.3\ *\ \frac{ADC\ Output}{ADC\ Max}\ *\ \frac{1000\ K}{10}\ =\ temp\ (K)$$

Divide the 1000K by 10 and substitute 4095 for `ADC Max`:

$$3.3\ *\ \frac{ADC\ Output}{4095}\ *\ 100\ K\ =\ temp\ (K)$$

which equals:

$$330\ K\ *\ \frac{ADC\ Output}{4095}\ =\ temp\ (K)$$

Now none of the values in the final equation has a decimal fraction and the Propeller can handle the math just fine. Likewise, the result does not include any decimal fraction. It gets "lost" when the Propeller performs the division ADC Output by ADC Max. (This loss of fractional information is not a flaw. It occurs normally due to the architecture of most microcontrollers that handle integer math in hardware. Some MCUs include floating-point hardware that handles decimal fractions. The ARM Cortex-M4 processor provides a good example.)

When I put my average ADC value (3651) in the equation above, it becomes:

$$330 * \frac{3651}{4095} * K = temp\,(K)$$

And my calculator shows 294.219. On the Propeller MCU, the decimal fraction of 0.219 gets eliminated, so the PST window showed a temperature of 294 K.

Step 7.

Run **Program 17.2**, which includes the math steps above, as well as a new legend for the ADC information from Channel 1. You should see:

```
MCP3202 12-bit A/D
******************
Channel 0: XX
Temp in K: 294
```

The Channel-0 value will change, depending on whether or not you have something connected to this input on the MCP3202 ADC. The temperature you see depends on your lab temperature and on whether you calibrated the LM335 sensor with the 10-kohm variable resistor. Due to how the Propeller MCU processes information, you do not see a decimal fraction for this temperature reading.

Program 17.2

```
''*****************************************************
'''*  Program 17.2, Analog-to-Digital Converter Test
'''*  Based on code created by Ken Gracey, Parallax
'''*  Modified by John Abshier (jabshier on Forum)
'''*  with different pin numbers and potentiometers
'''*  instead of joystick inputs.
'''*  Modified by Jon Titus, 11-23-2014 Rev. 1
'''*  Copyright 2014
'''*  Software for a Microchip Technology MCP3202 12-bit
'''*  ADC that displays a temperature from an LM335
'''*  temperature sensor in Kelvin.
'''*  Displays integer values only.
''*****************************************************
OBJ
  pst       : "FullDuplexSerial"    'serial comm object
  MCP3202   : "MCP3202"             'MCP3202 driver objects
  delay     : "timing"              'timing.spin file

CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

  clock_pin         = 24            'MCP3202 connections
  data_pin          = 23
  chip_select_pin   = 25

VAR
  long  CH0                         'Channel-0 data
  long  CH1                         'Channel-1 data

  byte PropPin_SerOut
  byte PropPin_SerIn
  byte PropPin_SerMode
  word SerPort_BaudRate
```

```
  long Kelvin_Integer_Data

PUB Main

 PropPin_SerOut    := 30            'P30 for USB transmit
 PropPin_SerIn     := 31            'P31 for USB receive
 PropPin_SerMode   := 0             'Invert RX mode
 SerPort_BaudRate  := 9600          'Baud rate of 9600 per sec

 'Serial-communication start-up operation
 pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

 'MCP3202 ADC start-up configuration
 'Enable Channels 0 and 1 for single-ended analog inputs
 'See MCP3202 datasheet for configuration options
 MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

repeat
  pst.tx(1)                            'Move cursor to home
  pst.str(string("MCP3202 12-bit A/D"))'Display this string
  pst.Tx(13)                           'New line
  pst.str(string("=================="))'Display this string
  pst.Tx(13)                           'New line

  CH0 := MCP3202.in(0)                 'Get Channel 0 data
  pst.str(string("Channel 0: "))       'Display this string
  pst.tx(11)                           'Clear old Channel-0 data
  pst.dec(CH0)                         'Display Channel-0 data
  pst.Tx(13)                           'New line

  CH1 := MCP3202.in(1)                 'Get Channel 1 data
  Kelvin_Integer_Data := 330 * CH1/4095  'Convert to Kelvin
  pst.str(string("Temp in K: "))       'Display this string
  pst.tx(11)                           'Clear old Channel-1 data
  pst.dec(Kelvin_Integer_Data)         'Display Kelvin temp
  pst.Tx(13)                           'New line

  delay.pause1ms(100)                  'Short delay

  ' –  – -end –  – -
```

Step 8.

Suppose calculator math produces the temperature 278.921 K. In an MCU without floating-point math capabilities, the 0.921 part gets eliminated. But eliminating decimal fractions in MCU calculations we could introduce inaccuracy in the results. Normally we would round up 278.921 K to 279 K.

Can we get around this problem? Yes, but if you don't want to learn how to round the result into an integer, feel free to go ahead to Step 10 in the "Measurements with a Light Sensor" section. Later you can come back and review Step 9. **Program 17.3** in the Experiment 17 folder includes the rounding steps for the ADC temperature values.

Step 9.

To round a value with a decimal fraction into an integer, we look at the digit to the immediate right of the decimal point – the tenths digit. The tenths values .0 through .4 indicate no rounding. We just "cut off" the decimal fraction. On the other hand, the tenths values .5 through .9 indicate we should remove the decimal

fraction and increase the integer number by 1. The numbers 56.390, 56.498, and 56.091 become 56, while the numbers 56.501, 56.712, and 56.699 get rounded up to 57. Instead of inspecting these values and deciding which to round up and which to leave alone, you can use math to do the rounding. For the six values above, add 0.5 and remove the decimal fraction.

$$56.390 + 0.5 = 56.890 -> 56.\cancel{890} -> 56$$
$$56.498 + 0.5 = 56.998 -> 56.\cancel{998} -> 56$$
$$56.091 + 0.5 = 56.591 -> 56.\cancel{591} -> 56$$
$$56.501 + 0.5 = 57.001 -> 57.\cancel{001} -> 57$$
$$56.712 + 0.5 = 57.212 -> 57.\cancel{212} -> 57$$
$$56.699 + 0.5 = 57.199 -> 57.\cancel{199} -> 57$$

Math handles the rounding. Adding the 0.5 value causes tenths values .5 through .9 to increase the original value by 1. Take the equation below and calculate the temperature for an ADC Output of 3462.

$$330\,K * \frac{3462}{4095} = 278.989\,K$$

We would mentally round this answer to 279 K. Software can make those types of rounding decisions by using similar math operations. Here's how rounding works with binary values from our calculation:

When the Propeller MCU (and other MCUs) multiplies 330 by 3462, the answer comes to:

$$330 * 3462 = 1,142,460$$

an integer number with the binary equivalent:

0000_0000_000**1_0001_0110_1110_1011_1100**

Now to simplify the division by 4095, I "fudge" a bit and use 4096 instead. You'll see why in a minute. The increase to 4096 means a change of only 0.02 percent, a difference so small we can ignore it.

The value 4096 equals 2^{12}, which lets us use 12 shift-right operations to do the same thing to a binary number as a mathematical division by 4096. But we lose the binary fraction part of the result. It gets "shifted out" to the right. (A bit "shifted out" to the right doesn't go anywhere, it simply gets replaced by the bit to its left.)

When we divide the binary value for 1,142,460 by 4096 with 12 shift-right operations we go from:

0000_0000_000**1_0001_0110_1110_1011_1100**

to

...000**1_0001_0110**, which equals 278 (256+16+4+2).

What happens if we use only 11 shift-right positions to divide by 2048? The result becomes:

000_0000_0000_0000_0000_00**10_0010_1101**

That result equals 278.5 because the LSB would represent 1/2 (0.5) in the final answer: 256 + 16 + 4 + 2 + 1/2. (In binary numbers we can have an implied "binary point" that separates an integer and a fraction in a binary value. For example, we could have 1101011.1011, where the integer equals 64 + 32 + 8 + 2 + 1 and the fraction represents 1/2 + 1/8 + 1/16. An MCU doesn't keep track of the binary point; that's up to us.)

So to round the result, we add 1/2 just as we did for a decimal value, and the addition looks like this:

```
   ...0000_00 10_0010_1101
 + ...0000_0000_0000_0001
   ...0000_00 10_0010_1110
```

Then we perform another shift-right operation to divide the result by 2. In the end we divided by 2048 * 2, or 4096. Now we have.

...0000_000**1_0001_0111** = 279.

Measurements with a Light Sensor

Step 10.

In this step you will connect a light-sensor circuit to the MCP3202 ADC to measure light intensity. I used an Advanced Photonix PDV-P9003-1 cadmium-sulfide (CdS) photoresistor, which according to the company's data sheet, changes resistance from about 1 Mohms (1,000,000 ohms) in darkness down to between 23 to 33 kohms in bright light. You may substitute a photoresistor with similar characteristics if you cannot find the Advanced Photonix part.

Data sheets for electronic sensors provide useful values, but when you first use a sensor your own tests often provide better information. I measured the resistance across the photoresistor as 30 Mohms (30,000,000 ohms) in total darkness and 4.5 kohms in low ambient room light. A third resistance reading showed about 400 ohms in direct incandescent light. So the measured resistances can fall in the range 400 ohms to 30 Mohms. A voltage-divider circuit with the photoresistor and a fixed-value resistor will provide voltages we can measure and relate to specific illumination levels. (**Figure 17.4**.)

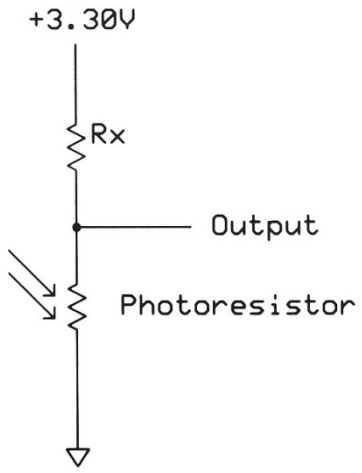


Figure 17.4.
A photoresistor and a fixed-value resistor (Rx) placed in series lets resistance changes in the photoresistor cause a change in the output voltage.

This circuit needs a fixed resistance with a value that will let the photoresistor cause the output voltage to have a wide range; ideally 0 to 3.3 volts. In practice, though, the circuit will get close to, but not reach, these voltages. I measured the voltage between the Output connection and ground for three resistances and **Table 17.2** shows the results. (I made several measurements; the table shows one set.)

Table 17.2. Voltage measurements for a photoresistor circuit under light and dark conditions.

	Measurement Condition	
Resistor (Rx)	**Full Light**	**Full Dark**
(ohms)	(Volts)	(Volts)
1000	0.791	3.25
10k	0.109	3.24
100k	0.00114	3.20

I used a sealed cardboard tube to make the Full Dark measurements and a 60-watt incandescent lamp placed a foot away for the Full Light condition. In each case, I let the photoresistor sit in its environment for 15 seconds before I took a measurement.

The 100-kohm resistor provides the widest voltage "swing" between dark and bright conditions. Add the 100-kohm resistor (brown-black-yellow) and photoresistor shown in **Figure 17.5** to your breadboard circuit. Connect the photoresistor output to the CH0 input at pin 2 on the MCP3202 ADC IC. If you still have an LM335 sensor attached to your MCP3202 ADC, you may leave it in your breadboard.

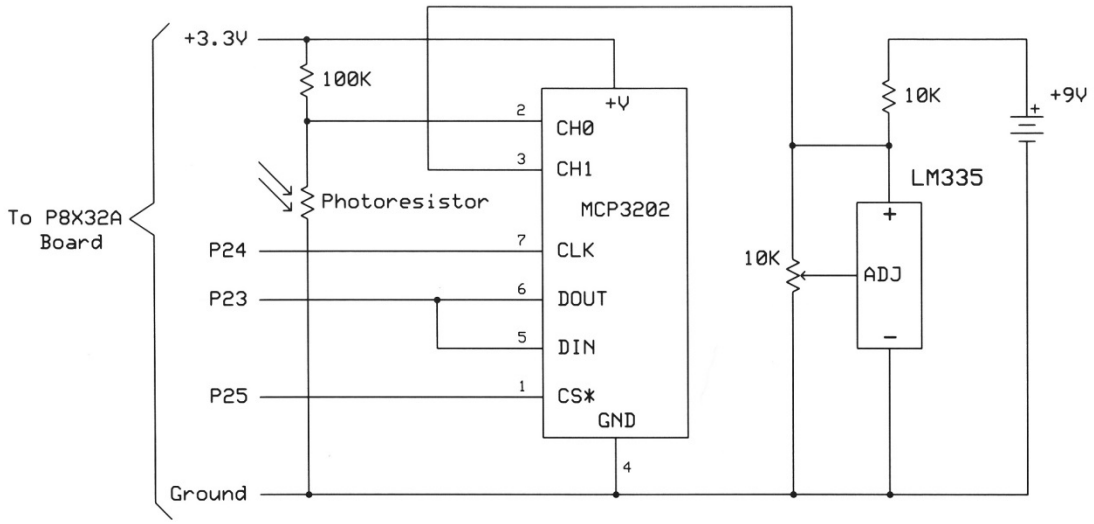


Figure 17.5.
Add the photoresistor and a 100-kohm resistor (brown-black-yellow) to your MCP3202 ADC circuit so the Propeller can measure voltage changes caused by changes in light intensity. If you still have an LM335 sensor connected, you may leave it connected.

Run **Program 17.1** again (see Step 3 above) and subject your photoresistor to bright and dark conditions. Watch the ADC values for Channel 0 in the PST window . Did you see the ADC values change as you moved the photoresistor from dark to bright environments? I recorded a full-darkness ADC value of 3499 and a full-brightness value of 19. (I took several measurements and averaged them.) Could you use those values to calculate light intensity? Yes, but...

A cadmium-sulfide (CdS) photoresistor exhibits a logarithmic, or exponential, response as shown in **Figure 17.6.** That response means instead of seeing a straight-line, or linear, relationship between photoresistor resistance and illumination intensity, changes occur as powers of 10, or from 1 to 10 ohms, 10 to 100 ohms, 100 to 1000 ohms, and so on. Engineers call this an order-of-magnitude change. The logarithmic relationship means you cannot use a simple equation to calculate light intensity for a given ADC value. (For a short review of logarithms see Step 6 in Experiment 5.)

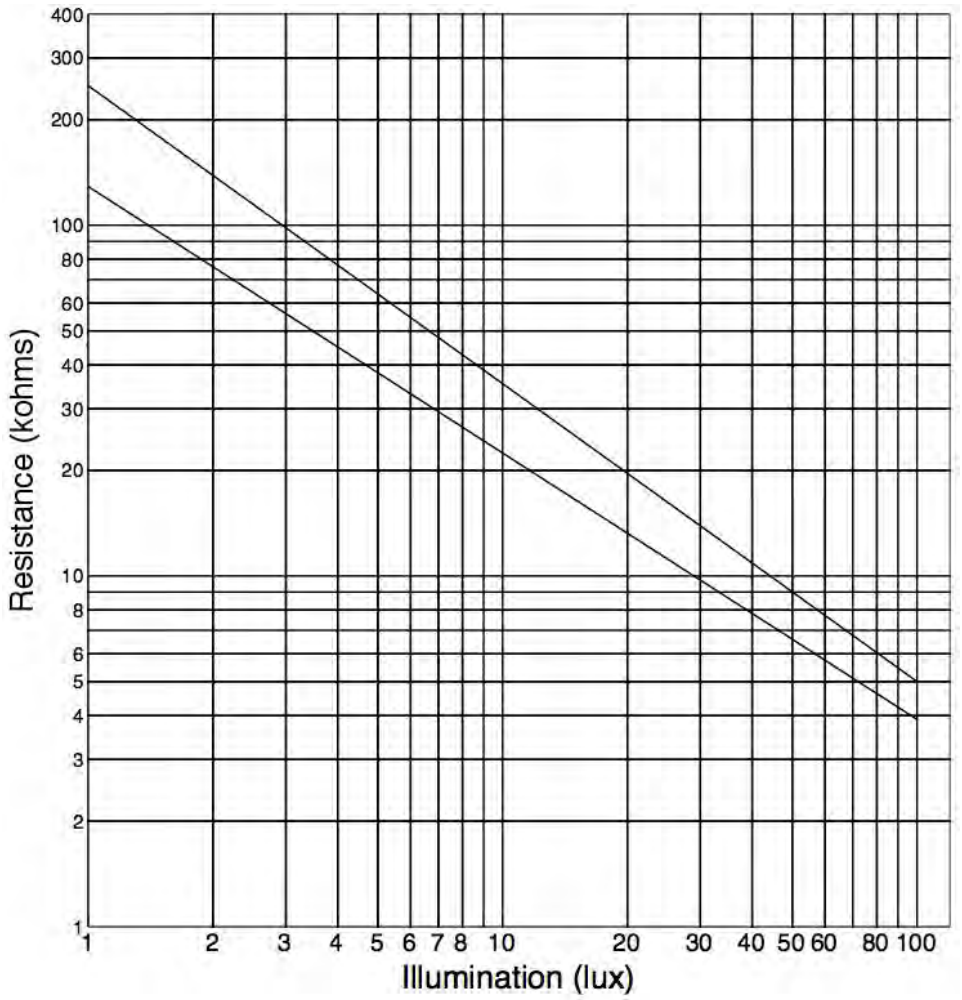
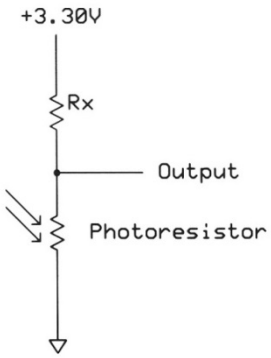


Figure 17.6.
This type of graph, called a log-log plot, uses powers of 10 to plot photoresistor resistances in ohms against light intensity in units of lux, also plotted along a powers-of-ten scale. Note the plot uses units of *kohms* along the Y axis, so the scale runs from 1 *kohm* to 400 *kohms*. The two diagonal lines represent the highest and lowest resistance for a given light intensity. A typical sensor would provide a result on or between these two lines. A full moon on a clear night gives about a 1-lux intensity and a 100-lux intensity approximates light on a dark overcast day.

Step 11.

If you know how to use algebra and logarithms you can work out the equation for a straight line in a log-log plot and calculate a lux light-intensity value for each ADC value. Instead of taking that path, you'll learn how to use the plotted information in an example. Then I'll provide an Excel table that gives you the same information in an easier-to-use format. You'll probably find the Excel technique easier!

The circuit shown below duplicates the one shown in **Figure 17.4** so you can better follow the next steps. Assume a 3.3-volt power source for the photoresistor circuit and a 3.3-volt reference for the MCP3202 ADC IC. This circuit uses a 100-kohm resistor in series with the photoresistor.



a) To use the information in **Figure 17.6**, you must calculate the *resistance* of the photoresistor based on the voltage at the Output between the two resistors.

b) The MCP3202 ADC provides a measurement range from 0 to 3.30 volts. We know the 12-bit ADC value for 0 volts equals 0_{10} and the value for a 3.30-volt signal equals 4095_{10}. Thus the ADC's LSB represents 0.806 mV (3.3 V / 4095 counts), or you can think of this as 0.806 mV per ADC count.

c) Suppose the photoresistor circuit exposed to a lamp causes the ADC to put out the value 1301, which corresponds directly to the *voltage* across the photoresistor. To calculate that voltage, multiply the ADC-output value by 0.806 mV per ADC count:

$$1301\ count\ *\ 0.806\ mV\ per\ ADC\ count\ =\ 1.049\ volts$$

d) Because the two resistors in series have 3.30 volts across them, the potential across the 100-kohm resistor amounts to:

$$3.30\ V\ -\ 1.049\ V\ =\ 2.251\ V$$ (see **Figure 17.7**)

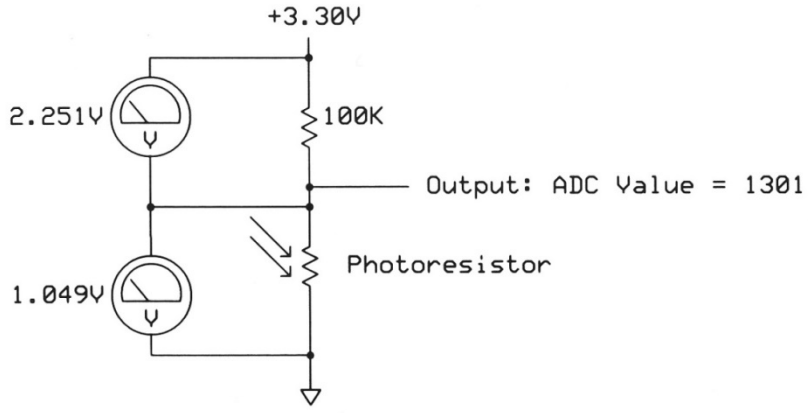


Figure 17.7.
Given an ADC output of 1301, equivalent to 1.049 volts, the potential across the 100-kohm resistor must equal 2.251 V. With this information you can calculate the resistance of the photoresistor.

e) Use Ohm's Law, $I\ =\ E\ /\ R$, for each resistor and the voltage across it to calculate current flow:

$$\frac{Voltage\ across\ 100,000\ ohms}{100,000\ ohms} = Current\ (I_1)$$

and

$$\frac{Voltage\ across\ photoresistor}{Photoresistor\ resistance} = Current\ (I_2)$$

By definition, two resistors in series have the same current flowing through them, so $I_1 = I_2$, and the equations then equal each other. Thus the ratio:

$$\frac{Voltage\ across\ 100,000\ ohms}{100,000\ ohms} = \frac{Voltage\ across\ photoresistor}{Photoresistor\ resistance}$$

And by algebraic rearrangement:

$$Photoresistor\ resistance = \frac{100{,}000\ ohms * photoresistor\ voltage}{Voltage\ across\ 100,000\ ohms}$$

f) In this example, the photoresistor resistance comes to:

$$\frac{100,000\ ohms * 1.049\ V}{(3.30\ V - 1.049\ V)} = 46,600\ ohms$$

g) Go to **Figure 17.8** and find the dashed horizontal at about 47,000 ohms. This line intersects the lower diagonal line at about 4.1 lux on the X axis and it intersects the upper diagonal line at about 7.4 lux. The illumination intensity exists somewhere between 4.1 and 7.4 lux. (For a better answer, you could plot data obtained from your sensor and use plot to determine lux for a measured resistance. That process requires a calibrated light meter and it would take a long time to accumulate all the data.)

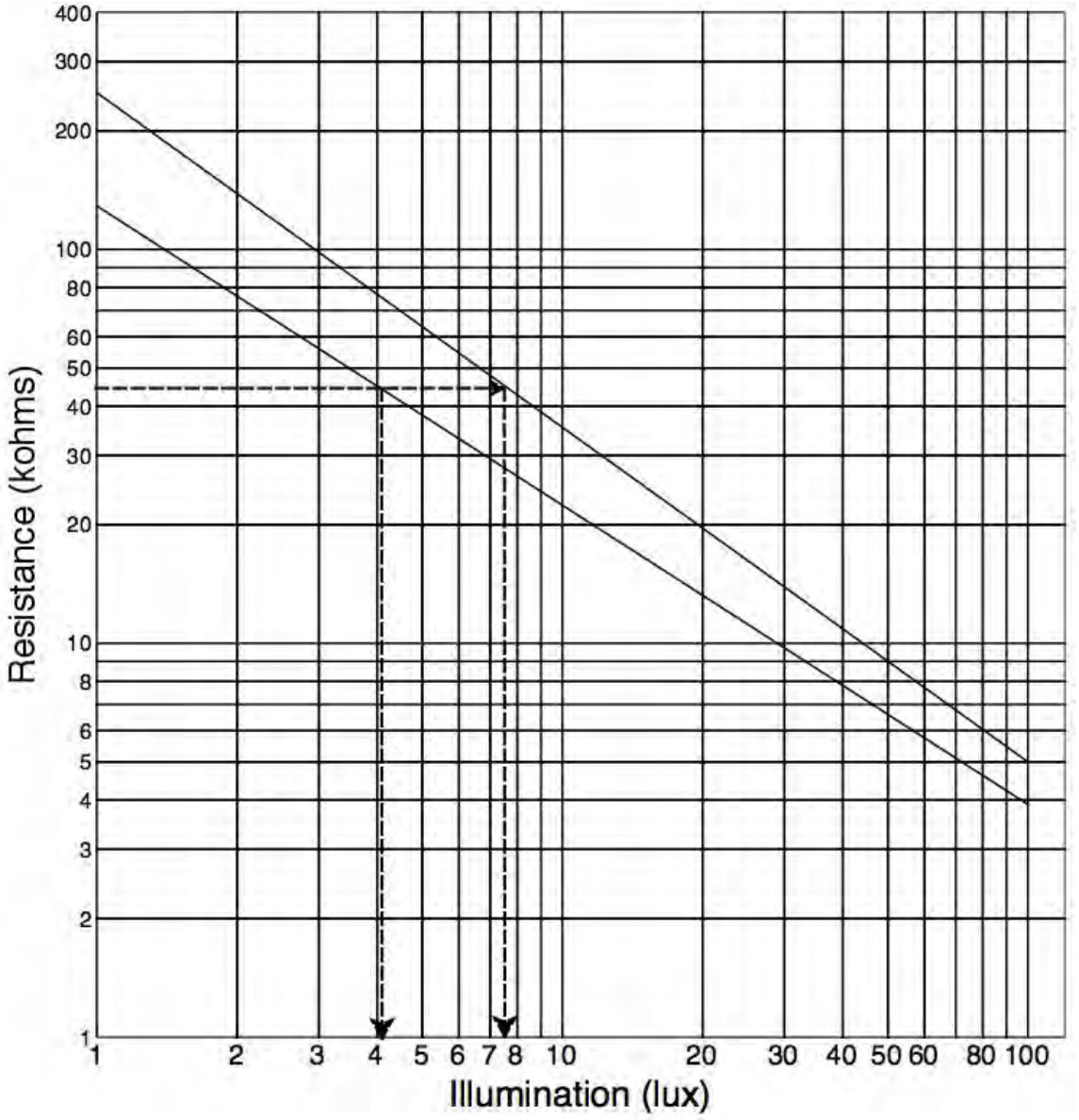


Figure 17.8.
In this diagram, the horizontal dashed line from about 47,000 ohms intersects one diagonal line at about 4.1 lux and the second diagonal line at about 7.4 lux. The measured resistance indicates a light intensity between these two values.

Step 12.

The graph method seems like too much work! Instead of calculating each light-intensity value, engineers could rely on a look-up table to match the ADC value (0 to 4095) with an illumination-intensity value. You would create this type of table – probably as an array – to match an ADC value (0 to 4095) with a specific light intensity (1 to 100 lux). This approach eliminates the math steps just explained in detail. You can use the LightSensorData.xls spreadsheet included in the Experiment 17 software folder to calculate the photoresistor resistance that corresponds to each of the ADC's 4096 outputs. The spreadsheet contains values for a 100,000-ohm series resistor and a 3.30-volt power supply, but you can change these values as you wish. To create this spreadsheet I extracted data from the diagonal lines in **Figure 17.8** and used them to derive an equation for each line. (For more information see the Notes section at the end of this experiment.) With this calculated information in the Excel results you could create an array with index values from 0 to 4095 and put the

corresponding lux value in each array element. The information below shows two array elements for the ADC output 1301. A program could retrieve the values stored at array locations `lightHIGH(1301)` and the `lightLOW(1301)` and display them:

```
lightHIGH(1301)   := 7
lightLOW(1301)    := 4
```

I have used integer values in the example above for light intensity and assume a 12-bit ADC. If you need more resolution, you can scale the values or use floating-point values and a floating-point Propeller object. More about scaling values in Experiment 20. Keep in mind, though, the data sheet for the Advanced Photonix PDV-P9003-1 lacks an error or accuracy specification, so it's unlikely you need more than fixed-point (integer) results.

If you need floating-point math you have another option – use a math coprocessor IC. The Micromega Corp. sells 32- and 64-bit floating-point math ICs for under $US 20 each. I have used the 8-pin uM-FPU V2 Floating Point Coprocessor with an SPI interface and it handled my floating-point information very well. Newer Micromega ICs have additional capabilities. These ICs handle trigonometric, logarithmic, and other complicated functions. To learn more about floating-point math, see the References section at the end of this experiment. In particular you want information about IEEE-754 floating-point math, the industry standard.

In some situations you can find circuits that linearize the output from a sensor, or at least a portion of the sensor's output range. The added components complicate a design, require calibration, and can pick up electrical noise. I prefer to handle nonlinear sensor outputs with math rather than electronics.

Thankfully, many sensors have a linear output that does not require much math to convert into useful engineering units such as kilograms, Celsius temperatures, times, and so on.

Conclusion

The circuits and software used in this experiment work reasonably well and should work properly in many projects and experiments. But in real-world equipment, an ADC will provide accurate and meaningful information only if you carefully follow good design procedures. Those aspects go far beyond the scope of this experiment and involve topics such as sampling rates, filters, grounding, switching analog signals, and so on. Over a decade I wrote many columns and blogs about measurement topics for *Design News* magazine, www.designnews.com. If you run a search on my name you will find more information about how to use ADCs. Application notes and data sheets from Analog Devices, Linear Technology, Texas Instruments, Microchip Technologies, Maxim Integrated, STMicroelectronics, and other companies provide a wealth of information if you want to learn more about ADCs and DACs.

1. If you have a sensor that produces a voltage between 0 and 10 volts, how could you adjust that signal so an MCP3202 ADC could properly measure it with only a 0-to-3.3-volt signal input?

2. Suppose a device must detect when a certain intensity of light reaches a preset limit. How might you set up some electronics and software to handle this task?

Find out in the **Answers** section at the end of this experiment .

References

For more information about light measurements, see "lux" at Wikipedia. http://en.wikipedia.org/wiki/Lux

"Equation of Straight Line on the Log-Log Scale," http://mathforum.org/library/drmath/view/69930.html.

"LM135, LM135A, LM235, LM235A, LM335, LM335A Precision Temperature Sensors," Texas Instruments. http://www.ti.com/lit/ds/symlink/lm135.pdf.

CdS Photoconductive Photocells, PDV-P9003-1," Advanced Photonix, Inc. http://www.advancedphotonix.com/ap_products/pdfs/PDV-P9003-1.pdf.

Overton, Michael L., "Numerical Computing with IEEE Floating Point Arithmetic," Society for Industrial and Applied Mathematics, Philadelphia, PA. 2001. ISBN: 978-0-89871-482-1.
You might find a used copy for sale, but try a technical library first. Unfortunately I have not found a good introduction to floating-point numbers and math on the Internet. Most articles either cover floating-point math from a mathematician's perspective or focus on it for users of a specific computer language.

Notes

The log-log plot of photoresistor resistance (kohms) vs. illumination intensity (lux) provides the two straight lines shown in **Figure 17.6**. For a linear graph with a straight line you would use the equation: $y = mx + b$ and x and y data from two points on the line to find the values for the slope (m) and the y-axis intercept (b).

For a log-log graph, you use a similar equation:

$$log(y) = m * log(x) + b$$

Again, two x and y points along a straight line on a log-log plot let you solve for m and b.

For the upper diagonal line in the log-log plot I used the end points on the line where:

$x = 1$ and $y = 250{,}000$ and where $x = 100$ and $y = 5000$

Remember, the *y* axis represents units of kohms!

With these two points, we have two equations:

$$log(250000) = M * log(1) + b \quad \text{and} \quad log(5000) = M * log(100) + b$$

The log of 1 equals 0, so: $5.398 = M * 0 + b, and\ b = 5.398$

Now, $log(5000) = m * log(100) + 5.398$

$$3.699 = m * 2 + 5.398$$

$$m = -0.845$$

Therefore, $log(y) = -0.845 * log(x) + 5.398$

You can use the resistance (*y*) to find the lux value (*x*) in a spreadsheet:

$$x = EXP((log(y) - 5.398) / -0.845)$$

Then you match this light-intensity value with the array index; that is, the ADC digital value. Look at the spreadsheet formulas to see how to calculate the resistance values.

Answers

Experiment 17, Conclusion:

1. If you want to use an ADC with an input range of 0 to 3.3 volts with a sensor that will produce a signal between 0 and 10 volts, you will need a voltage "divider" circuit similar to the one used with the photoresistor. Start with a circuit as shown in **Figure 17.9** and use a 10-kohm resistor as shown with a resistor of some value we'll calculate.

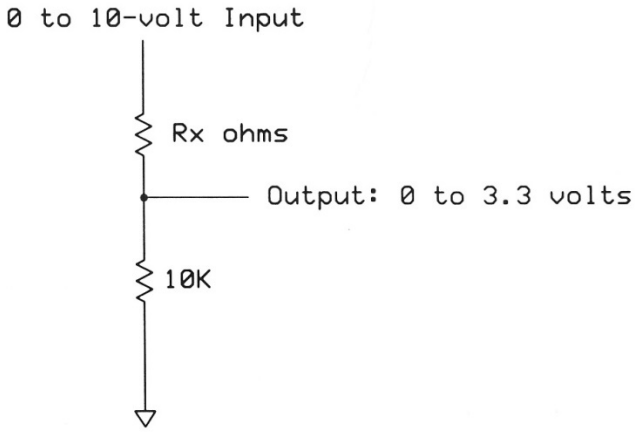


Figure 17.9.
A typical voltage divider circuit created with two resistors.

In this circuit you want a maximum of 3.3 volts at the output that goes to an MCP3202 ADC. Thus the maximum voltage across the 10-kohm resistor must equal 3,3 volts. That gives you the relationship:

$$10{,}000\ ohms\ /\ 3.3\ volts\ =\ 3030.30\ldots ohms/V$$

The unknown resistor must have a 10V - 3.3V voltage drop, or 6.7 volts.

Multiply the 3030 ohms/V by 6.7V and the volt units cancel out. That leaves you with:

$$3030\ ohms\ *\ 6.7\ =\ 20301\ ohms$$

You cannot buy a resistor with this value, but you could combine an 18-kohm fixed resistor with a 5-kohm variable resistor. The variable resistor provides an overall range of 18 to 23 kohms, plenty of range so you could adjust the total resistance to get close to 20300 ohms.

2. If you have an ADC input available, use a short program that gets the ADC value and compares it to the preset limit. To avoid any math, use the 12-bit value equivalent to the ADC value expected at the limit.

If you don't an ADC, use a comparator IC and compare the voltage from a resistor-photoresistor circuit, such as the one created in this experiment, with a voltage preset at the limit voltage expected from the photoresistor.

Experiment No. 18 – How to Use Infrared LEDs for Remote Control

Abstract

The previous experiments all used LEDs that emit light within the visible spectrum – red through violet. Now you will work with LEDs that emit infrared "light" that human eyes cannot detect. You might use infrared (IR) LEDs when you don't want people to see a beam of light, or where you need wireless communication over a short distance. Examples of these applications include security systems that detect an intruder and remote controls used for televisions and digital-video recorders. Also, robots built by experimenter and hobbyists often use an IR remote control.

Unlike most earlier experiments, this one includes more tutorial information than hands-on work, but you still get to experiment with IR LEDs and sensors. The first section of this experiment covers fundamental information. The second section involves using a microcontroller to interpret IR codes from a handheld remote control. In Experiment 19 you will learn how to communicate information back and forth with remote devices.

Keywords

Infrared, IR, LED, sensor, remote control, 38 kHz, encoder, decoder, modulation, demodulation, NEC, pulse-distance encoding, optical interrupter, optical switch, incremental encoder

Requirements

(1) - Solderless breadboard
(1) - Propeller P8X32A microcontroller board
(1) - USB cable for Propeller board
(1) - Digital voltmeter or analog voltmeter
(1) - Infrared LED, 940 nm, 50 mA, Lite-On LTE-302, or LTE-4208
(1) - Infrared sensor, 940 nm, Lite-On LTR-301, or Everlight Electronics PT928-6C-F
(1) - Vishay TSOP38238 or Panasonic PNA4602M 38-kHz IR receiver/sensor
(1) - Sparkfun IR kit, part no. COM-11759
(1) - 120-ohm, 1/4-watt resistor, 5% (brown-red-brown)
(1) - 2200-ohm, 1/4-watt resistor, 5% (red-red-red)
(1) - 4700-ohm, 1/4-watt resistor, 5% (yellow-violet-red)
(1) - 74HC04 inverter IC, 14-pin DIP
(1) - 5-volt power supply

Introduction

Human eyes detect energy, or light, at wavelengths between about 390 and 700 nanometers (nm), which we call the *visible-light* spectrum. Most data sheets for LEDs report light color in nanometer units. Scientists and engineers use wavelength units of *nanometers* (nm, 10^{-9} meters) or *micrometers* (μm, 10^{-6} meters) to describe light signals. One-thousand nanometers equals one micrometer. For the sake of clarity and to provide consistency with manufacturer specifications, this experiment uses nanometer units. Infrared light extends beyond the red end of the visible-light spectrum and people usually define the IR spectrum between 700 nm (short-wavelength IR) to 24,000 nm (very long wavelength IR). Most IR LEDs produce light between about 850 or 940 nm, depending on the materials used to create them. (Although we can't see IR light, I'll use the word "light" to describe an IR signal.)

If we cannot see IR light, how did someone determine it existed? In 1800, Sir Frederick William Herschel (1738-1822), a well-known astronomer, set up an experiment to measure the heat "content" of light from the

sun. He wanted to measure the heat caused by light and he used a glass prism to produce a rainbow-like array of colors. Then Herschel placed thermometers at several places in spectrum – red, orange, green, blue, and so on. He discovered temperatures increased as he made measurements from the violet to the red portions of the spectrum.

Herschel then placed "control" thermometers just outside the red and violet ends of the visible spectrum to obtain temperatures he could compare with those made at the various colors. To his surprise, the thermometer beyond the red end of the spectrum showed a higher temperature than that measured in the red light. Obviously heat energy existed in this portion of the spectrum, even though Herschel couldn't see any light there. He called the invisible heat energy "calorific rays." So although we cannot see IR light from the sun or another source, we can detect it.

But you don't need sunlight, a prism, and thermometers. Silicon sensors used in many digital cameras respond to IR light and in many cases you can see the results on a camera's display. Cameras used by professional photographers include a filter to remove IR light because it can slightly distort an image.

Why Use an IR LED?

In some cases you don't want to see light. When you use a remote control with home-entertainment equipment, it could get annoying to see a bright LED flash on and off each time you press a button on a remote control. So equipment manufacturers use IR signals to transmit control codes. Also, a robot could use an IR sensor and an IR LED to help it avoid a nearby object.

At times you might not want others to see visible light. Suppose you have a security system that uses a light beam to control an alarm when someone enters a doorway. An IR light source makes the beam invisible. And unlike radio communications that anyone can listen to, people cannot intercept IR light communications unless they position a detector in the light beam.

In the following steps you will explore uses of IR LEDs and sensors in simple circuits that detect the absence or presence of an object that blocks a light path. After that you will learn how to use a handheld IR remote control and sensor to control nearby devices.

Create and Detect Infrared Light: Sensors and LEDs

In this part of Experiment 18, you will create a circuit with an IR LED and determine whether the LED emits IR light or remains "dark." I recommend you set up this experiment in an area where you can darken your work area, although you won't need complete darkness. I used an IR LED that produced light between 940 and 950 nm.

An IR LED, like its visible-light LED cousins, needs a current-limiting resistor. The LED used here can handle a 50 mA maximum continuous current with a typical forward voltage (V_F) drop of 1.2 volts. Assume a 5-volt power supply for the LED and calculate the series resistance needed to pass 30 mA through the LED. If you cannot remember how to perform the calculation, find it in the Answers section at the end of this experiment. What resistance did you calculate?

I calculated a value of 130 ohms but will use a more-common 120-ohm resistor. The diagram in **Figure 18.1** provides a mechanical drawing of two IR-LED types in clear-plastic packages. The arrangement of the LED semiconductor in these package distributes most of the IR light LED in a cone of about 30 degrees from the central axis, much like a flashlight beam. Companies sell IR LEDs that have other conical illumination patterns from about 5 to 180 degrees.

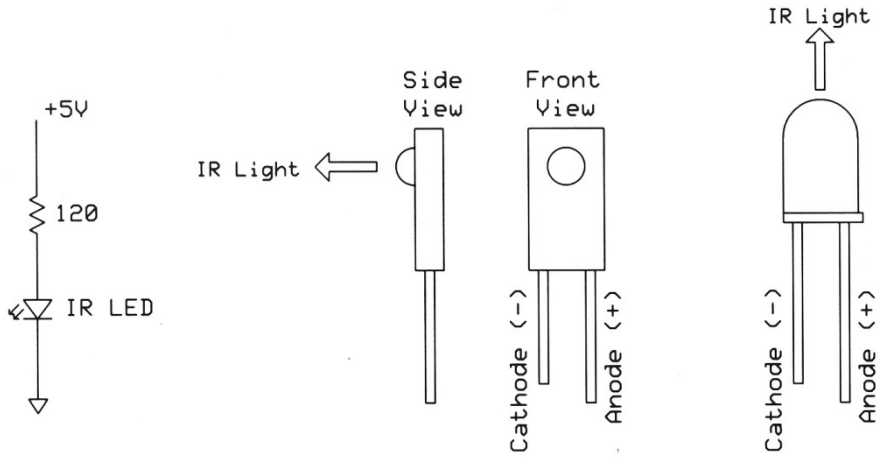


Figure 18.1.
Companies manufacture infrared LEDs in many sizes and shapes. The Lite-On Electronics LTE-302 (middle) comes in a rectangular package and emits light sideways through a clear plastic lens. An IR LED in a standard T1¾ format (right) emits light centered on the package axis.

Keep in mind that light intensity decreases the farther you move from its source. Engineers call this type of change an "inverse-square" relationship, or the inverse-square law, illustrated in **Figure 18.2** (Ref. 1). Say you measure light intensity at a distance of 10 cm (**Figure 18.2**, point r) as seven microwatts per square centimeter (7 μw/cm^2), Then you measure light intensity 20 cm away from the source (point 2r). Here the intensity becomes 7 μw/cm^2 divided by 2 squared (2^2), or 1.75 μw/cm^2. When you measure the light intensity 30 cm (point 3r) from the source, the intensity decreases to 7 μw/cm^2 divided by 3 squared (3^2), or 0.778 μw/cm^2. When you need IR light concentrated at a small sensor, choose an IR LED with a small projection angle to put most of the light on the sensor. If you need IR light over a wide angle, use an IR LED with a wider projection angle. Or use several narrow-angle IR LEDs that point in off-center directions.

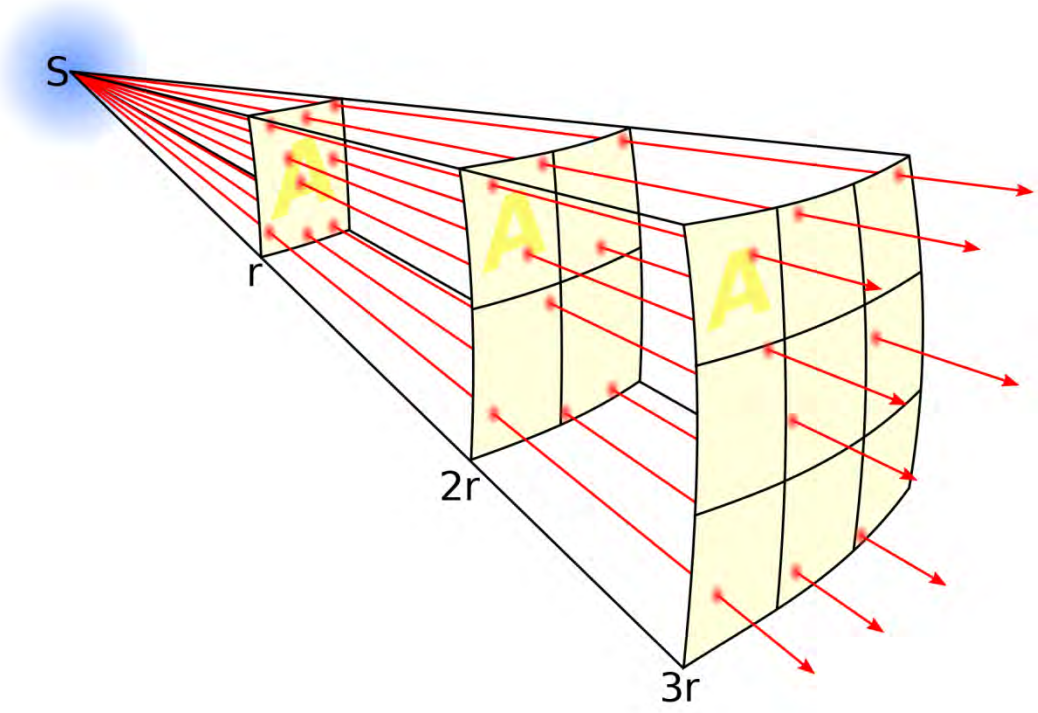


Figure 18.2.
The inverse-square law as it applies to light intensity vs. distance from a point source of light at location S and at multiples of the distance r.
Courtesy of Borb and provided under the terms of the GNU Free Documentation License, Version 1.2.

Basic Infrared Experiments

Step 1.

Take a solderless breadboard and connect the Lite-On LTE-302 IR LED and the 120-ohm (brown-red-brown) current-limiting resistor as shown in **Figure 18.1**. If you choose a different type of IR LED, check its data sheet for the forward-voltage (V_F) and maximum-current values, and calculate the needed current-limiting resistance. An alternate IR LED should emit light at or close to 940 nm.

Place the LED in a breadboard so the IR-light beam will point horizontally. If necessary, bend the LED leads at a 90-degree angle. The longer LED lead provides the anode (+) connection and the shorter lead provides the cathode (-) connection. Turn on your circuit. Do you see any light from the IR LED? Probably not.

If you have a cellphone with a built-in camera, turn it on, select the camera, and point the lens at the powered IR LED. Now you might see a light-blue or pink light from the IR LED appear in your camera's display. My Samsung smartphone detects the IR light. As mentioned earlier, not all cameras exhibit this IR sensitivity. I have an older Sony camera that includes an internal "hot-mirror filter" that removes IR light before it reaches the image sensor. According to a Wikipedia article, a hot-mirror filter can remove IR light between about 750 and 1250 nm., so I see nothing from the powered IR LED on this camera's display (Ref. 2). Turn off power to the breadboard circuit.

Step 2.

Although a smartphone provides a convenient way to quickly test an IR LED, we need something simpler – and less expensive – to create a practical sensor. As described above, a silicon image sensors can detect IR

light, so manufacturers have tailored semiconductor materials to create stand-alone IR-light sensors. Now you will use such a sensor with the IR LED already in your breadboard. Place a Lite-On LTR-301 sensor in the breadboard opposite the IR LED, and so the sensor lens faces the lens on the IR LED across the breadboard's center slot. I left about 1 cm between the LED and the sensor. Wire the circuit shown in **Figure 18.3** and connect a voltmeter between ground and the sensor collector terminal. **Figure 18.4** shows my breadboard arrangement.

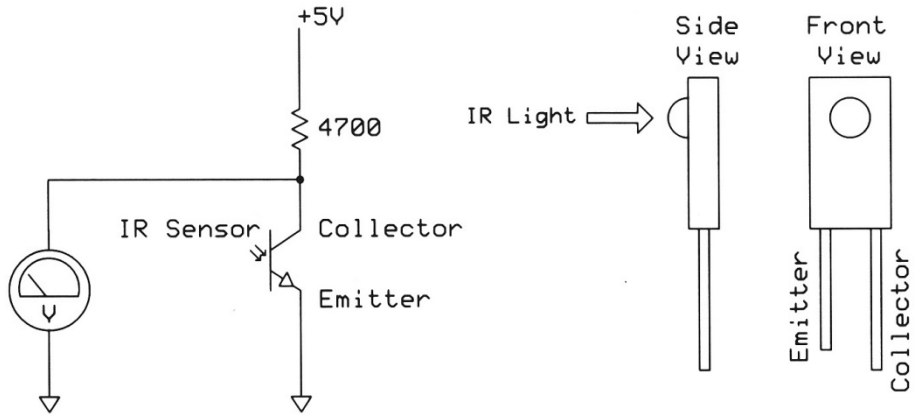


Figure 18.3.
An IR sensor such as the Lite-On LTR-301 uses a semiconductor that changes the amount of current it can pass based on the amount of IR light it receives. This circuit shows the connection of the sensor and a series resistor. The voltmeter will indicate the presence or absence of IR light.

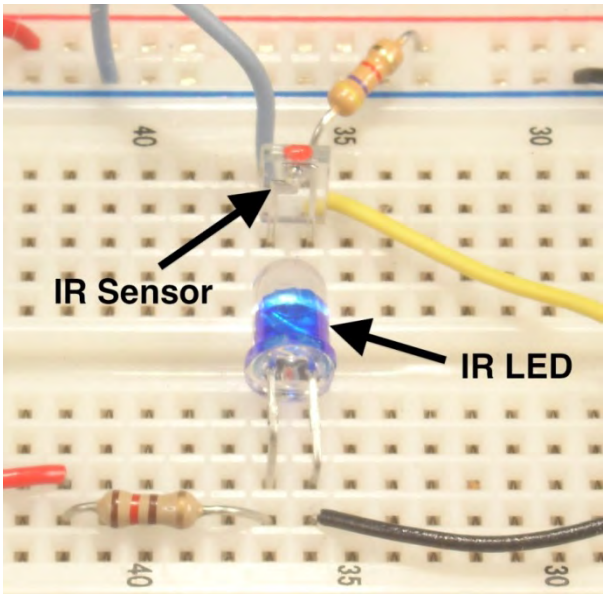


Figure 18.4.
Mounting configuration for an infrared LED and an infrared sensor. I used a blue marker to colored the sides of the IR LED in this figure to make it easier to see. The LED has a water-clear plastic body. The IR sensor has a red dot on top of its plastic package.

Cut a piece of cardboard or thick paper approximately 1-by-2 inches (2.5-by-5 cm) and set it aside. Clear any obstructions – wires, power-supply connections – out of the light path between the IR LED and the sensor. Turn off lamps and lights in your work area so their IR output will not interfere with measurements. Turn on power to the circuit and measure the voltage at the sensor.

Sensor voltage output: _____ volts

Place your piece of cardboard or paper between the sensor and the LED. Note the sensor voltage below:

Blocked-sensor voltage output: _____ volts

I measured 155 mV (0.155 volts) when the sensor had an unobstructed light path to the IR LED. The voltage increased to 3.33 volts when I blocked the light path. Your measurements will vary from mine due to the distance between your IR LED and sensor, ambient light in your work area, and alignment of the IR LED beam and the sensor in your breadboard.

How does the IR light affect the sensor? The more IR light the sensor receives, the more current it can carry. The increase in current flow reduces the voltage at the sensor's collector terminal. Conversely, less IR light causes the sensor to pass less current, which means the measured voltage increases. You could use this type of IR LED and sensor arrangement to detect the presence or absence of something that blocks IR light. But how fast can the IR sensor react as an IR LED turns on or off?

I used a PropScope to capture the fall time and the rise time of the sensor voltage when I used a digital circuit to turn the IR LED on or off. You can see the results in **Figure 18.5**. In the upper set of traces the sensor takes about 100 microseconds to respond and have its output reach 4 volts. In the lower pair of traces, when the IR LED turns on it takes the sensor only about 6 microseconds to respond. According to the Lite-On data sheet for the LTR-301 sensor, these response times depend somewhat on the load resistance – 4700 ohms in this case. If you increase this resistance the rise and fall times increase.

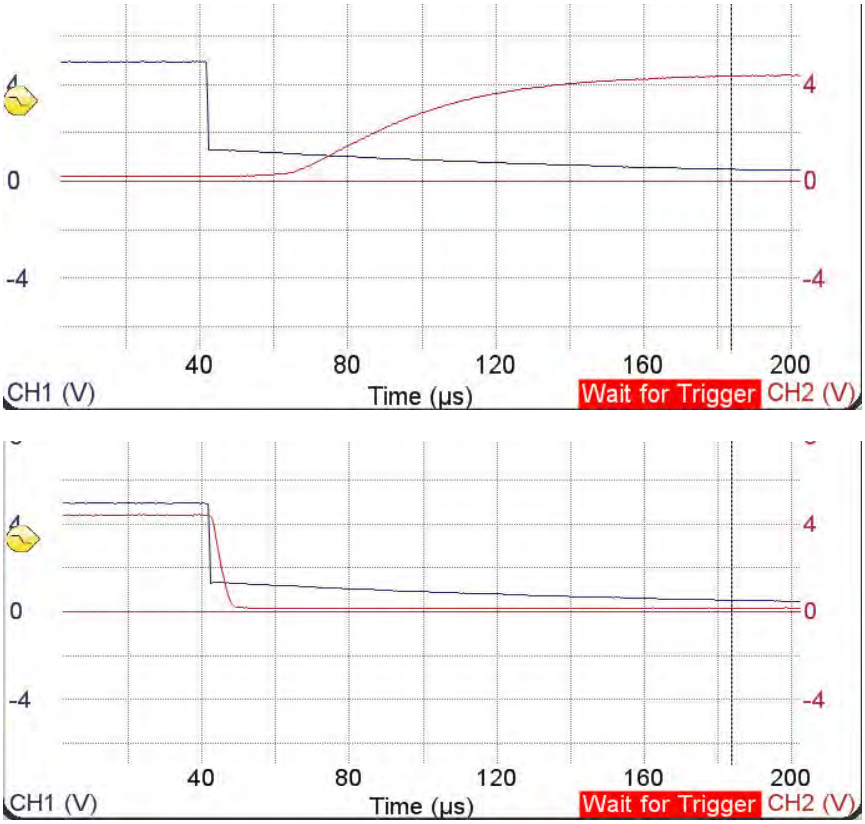


Figure 18.5.
The upper scope plot shows the turn-off characteristics of a Lite-On LTR-301 IR sensor when an IR LED pointed at it turns off. The lower plot shows how quickly the sensor turns on when the IR LED turns on. The fast logic-level transition at about 40 microseconds controls the IR LED and served as a trigger signal for the Propscope.

Can you think of practical uses for the type of circuit you created? If you place a piece of metal on a door, and attach a sensor and an IR LED on the door frame, the metal could block the IR signal when the door latches closed, and pass the IR signal when the door opens. This type of arrangement, along with a visible LED controlled by the sensor could indicate to a driver whether or not a garage door has completely opened or shut.

Step 3.

Manufacturers don't want to align an IR LED and a sensor in products they manufacture. So instead of employing separate devices, they use an *optical interrupter* , also called an optical switch. An interrupter incorporates an IR LED and an IR sensor in an opaque plastic package with a slot between the two devices. The small modules shown in **Figure 18.6** provide good examples. Plastic tabs provide mounting holes. Interrupters without mounting holes would connect directly to a circuit board. Many manufacturers produce optical interrupters in a variety of standard and custom-designed packages.

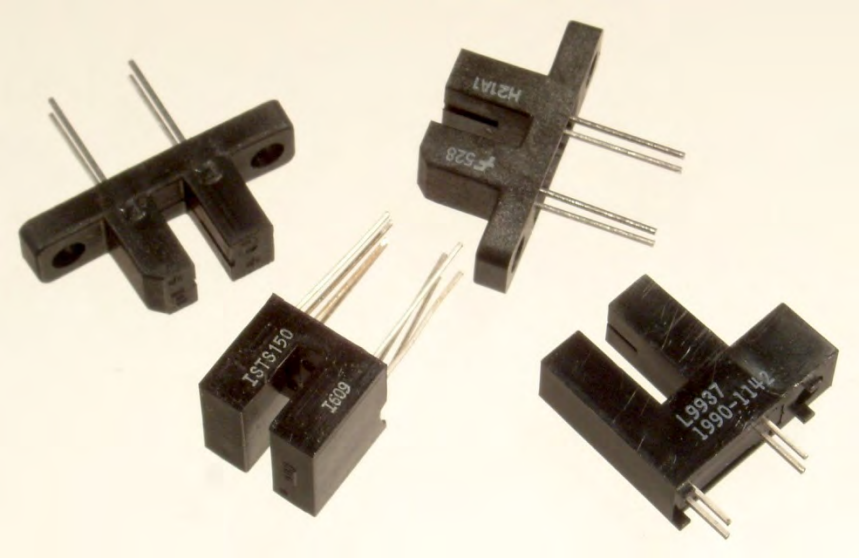


Figure 18.6.
These optical interrupters include an IR LED on one side of an open slot and a IR sensor on the other side. Devices such as these can detect the absence or presence of an object and provide a signal to a circuit or microcontroller. An optical interrupter requires four connections, two for the IR LED on one side of a module, and two on the other side for the IR sensor.

In addition to detecting an opened or closed door, optical interrupters can detect when an object has moved into a proper position. A machine might incorporate several optical interrupters to let electronic controllers know when various gears, belts, and moving stages have reached a specific position. In a small robot, for example, a slot in a gear and an optical interrupter could let an MCU know an actuator had reached its "home" position. Early designs for computer mice and trackballs use optical interrupters and small wheels with slots in them to indicate movements. A Google search will yield many interesting circuits and projects that use optical interrupters.

When mated with a thin metal disc with holes punched near its outer edge and at equal intervals, an optical interrupter and an MCU could count the interruptions of the IR light over a set time. Then the MCU could calculate a revolutions-per-second value for the wheel. By using two optical interrupters and two sets of offset holes in a wheel, an MCU also could determine the direction of rotation as shown in **Figure 18.7** (Ref. 3). Commercial rotary encoders, also called incremental rotary encoders, use this technique to track the position of a rotating shaft.These encoders can produce from eight to several hundred pulses per revolution, depending on design requirements.

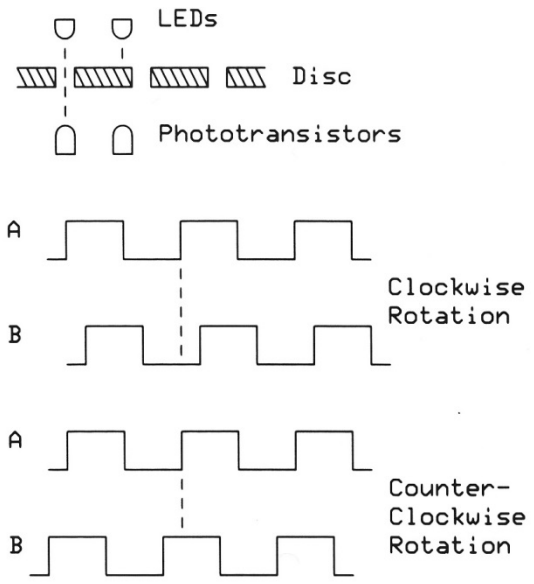


Figure 18.7.
An opaque disc perforated at regular intervals with a hole-to-hole spacing *offset* from a pair of IR LEDs and IR sensors (phototransistors) lets equipment determine the direction of a shaft's rotation.

In **Figure 18.7**, note the phase, or offset, of the positive-going edge of signal A with respect to the logic level of signal B. An MCU could test the state of the B signal whenever the A signal changes from a logic-0 to a logic-1. Based on the state of signal B, the MCU can determine the direction in which the encoder's shaft has turned. And by counting the pulses from one of the sensors over a preset period, an MCU could calculate the rotation speed.

Infrared Remote-Control Signals, A Tutorial

When you made voltage measurements in Step 2, the IR sensor required a darkened environment so ambient IR light from windows, incandescent lights, and compact fluorescent lamps (CFLs) would not greatly affect the sensor's operation. When engineers design an IR remote control, they cannot know in advance what lighting conditions will exist in a home, office, or manufacturing plant. So to overcome varying lighting conditions, remote controls do not simply turn an IR LED on or off to create a a logic-0 or a logic-1 signal. Instead, they use a *modulated* IR signal created by combining a standard 38-kHz *carrier frequency* and the data to transmit, as shown in **Figure 18.8.** In this example, a logic-1 modulating signal lets the carrier signal pass to the IR LED and turn it on and off at 38 kHz. A logic-0 blocks the carrier signal and keeps the IR LED turned off.

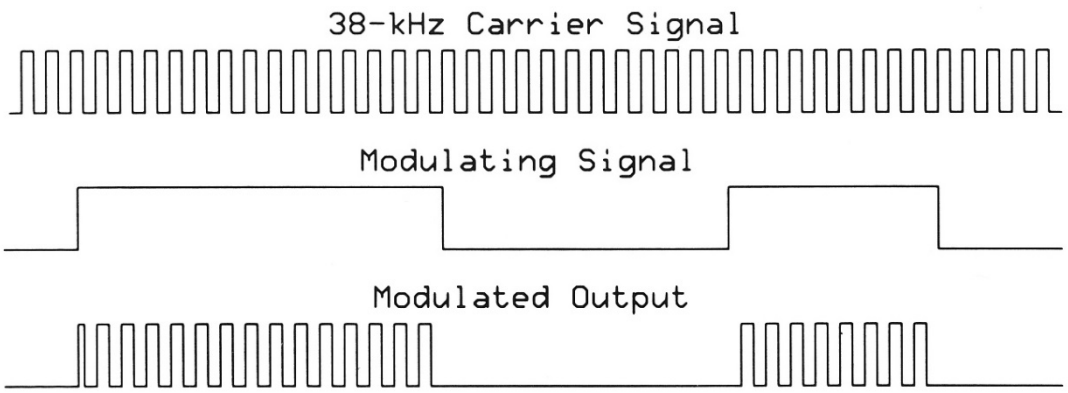


Figure 18.8.
Modulation involves one signal altering another signal. In this case, a 38-kHz signal acts as the "carrier" signal that remains constantly on. The modulating signal – logic-1 or logic-0 – control the carrier to create a modulated output. In this type of modulation, the carrier must have a higher frequency than the modulating signal.

Modulating a signal at a high frequency lets a receiving device discriminate between the wanted 38-kHz IR-LED signal and unwanted lower-frequency signals. Interference can arise from IR-source "flicker" at power-line frequencies of 50- or 60-Hz and at the corresponding 100- or 120-Hz harmonics. Using a high-frequency carrier signal rather than a steady-on or -off signal also lets a receiving circuit filter out constant or slowly varying IR light from various sources. The use of LEDs with infrared energy between 930 and 950 nm has an added benefit: water vapor in the atmosphere attenuates IR light from the sun at these wavelengths.

A receiving device takes the modulated IR signal and converts it back into the modulating signal shown in **Figure 18.8**. The diagram in **Figure 18.9** shows the circuit blocks built into an IR receiver/demodulator integrated circuit (IC). Receiver ICs have only three pins – power, ground, and a logic-level output. Because so many manufacturers of consumer-electronic devices use these IR receiver ICs, they cost only a few dollars each.

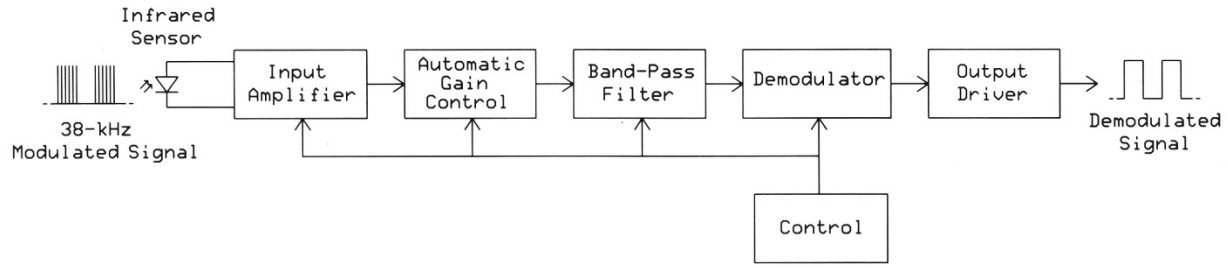


Figure 18.9.
This block diagram shows the path for information that arrives as a modulated 38-kHz IR signal and goes through five stages to produce a demodulated output. IR sensors such as the Vishay TSOP38238 and the Panasonic PNA4602M include these circuit elements in a package with only three electrical leads.

The five functional blocks in a receiver IC perform the following operations:

1. The input amplifier increases the amplitude of the electrical signals from the IR sensor so the circuits that follow have a larger signal to work with.

2. The automatic-gain-control (AGC) block keeps its output at a preset level for the band-pass filter. Amplifiers boost weak signals and attenuate signals that could saturate, or "swamp," the band-pass filter.

3. The bandpass-filter section removes signals outside a preset bandwidth. Most IR remote controls use a modulation frequency between 33 and 40 kHz or between 50 and 60 kHz. The popular and widely used NEC Corporation IR-control standard specifies a 38-kHz carrier frequency.

4. A demodulator removes the 38-kHz carrier signal from the received IR signal and produced voltages similar to those for logic-1 and -0 signals.

5. The output driver creates the proper logic-level signals for MCUs and other types of devices.

In the following explanations and examples I used an IR Control Kit from Sparkfun Electronics, part number COM-11759. The kit includes a ready-to-use handheld IR remote control, two IR LEDs, two Vishay type-TSOP38238 receiver ICs, and 20 330-ohm resistors. (Put the resistors in your parts bin; this experiment does not use them.) Each of the eight pushbuttons on the remote control causes the internal IR LED to transmit a unique code. **Figure 18.10a** shows two remote controls and **Figure 18.b** provides the circuit for an IR receiver and a 74HC04 inverter I used as a buffer on the sensor output.

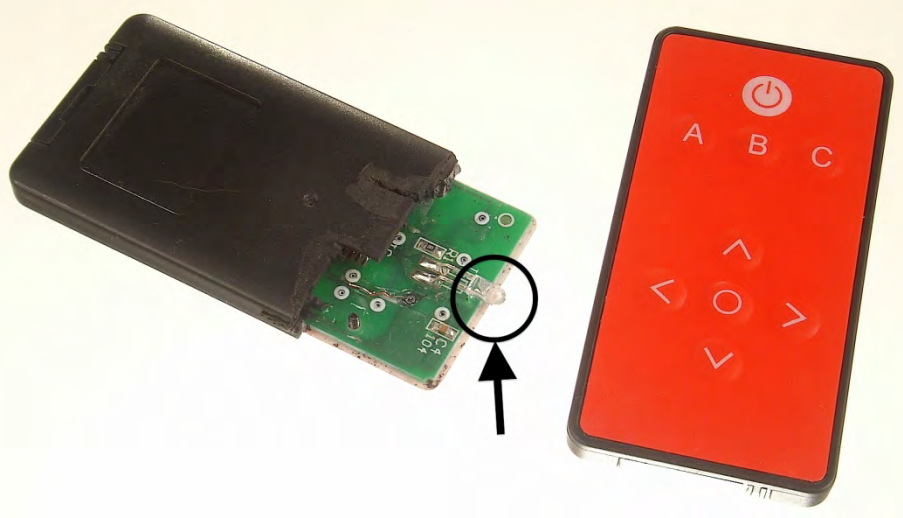


Figure 18.10a.
I used one control as supplied (right). I removed some of the plastic case for a second control (left) so I could connect probes to the IR LED transmitter. The arrow and black circle identify this LED.

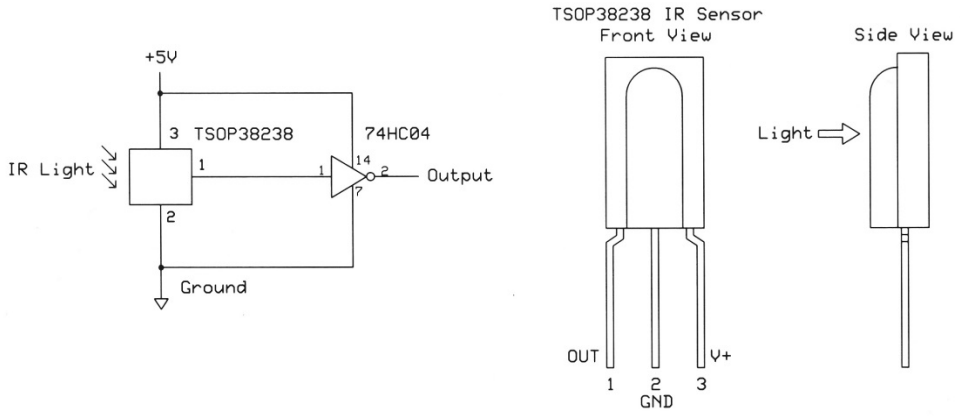


Figure 18.10b.
This diagram shows a simple circuit for a Vishay TSOP38238 IR sensor that demodulates logic signals sent on a 38-kHz carrier signal. The 74HC04 inverter will drive other circuits.

I powered the sensor and used my oscilloscope to examine the 74HC04 output (pin 2) when I pushed buttons. **Figure 18.11** shows the first part of the sensor output when I pressed the right-arrow button (>).

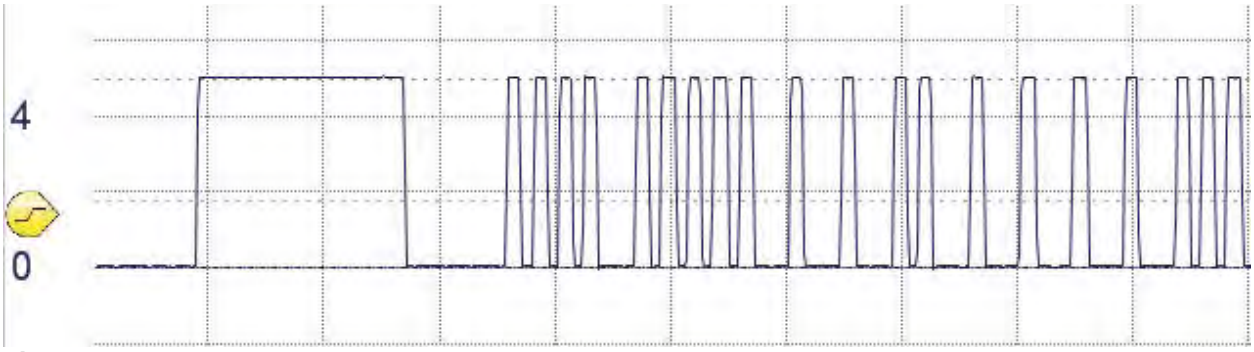


Figure 18.11.
The timing diagram for signals produced by an IR sensor illustrates the type of information transmitted by a remote control. An MCU can decode this series of pulses and provide useful information. The horizontal time axis has units of 5 msec per division, while the vertical axis shows 2 volts per division.

Infrared remote controls transmit information in several standard formats, depending on a manufacturer's preferences or requirements. Those formats include Sony SIRC (Serial Infra Red Control), Matsushita, NEC, and Philips RC5 and RC6. The Sparkfun remote control uses the format devised by NEC Electronics.

At first look, the NEC communication format shown in **Figure 18.11** can seem complicated, but breaking a transmission into sections makes it easier to understand. **Figure 18.12** shows a top-level view of an NEC information packet. You might remember that serial communications via a universal asynchronous receiver transmitter (UART) begin with a start bit and end with a stop bit. An NEC-packet begins with a 9-millisecond logic-1 signal followed by a 4.5-millisecond logic-0 signal. A 560-microsecond logic-1 end-of-packet pulse terminates each packet.

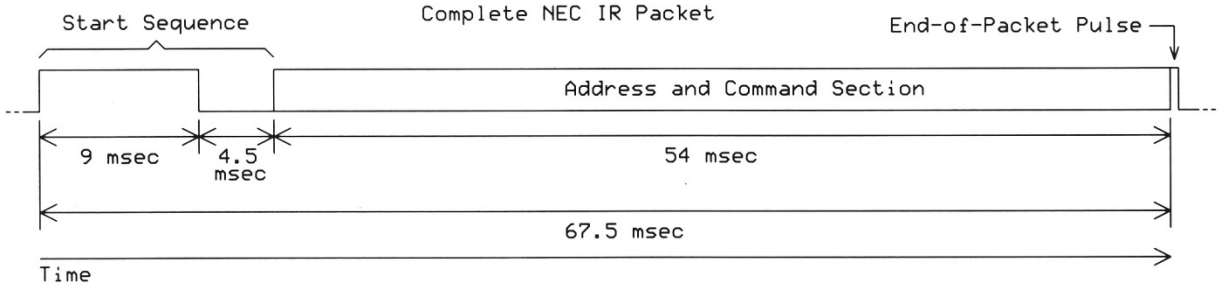


Figure 18.12.
An NEC-compatible transmission includes a 13.5-millisecond start sequence (9 plus 4.5 msec), followed by address and command information. Each transmission includes a 560-microsecond logic-1 end-of-packet pulse.

The Address and Command portion of a packet provides an 8-bit address of the device you want to control followed by an 8-bit command (**Figure 18.13**). Note: The NEC format specifies the command and address bytes must start with the least-significant bit (LSB) first and end with the most-significant bit (MSB). In NEC-formatted *timing diagrams*, the LSB appears to the left and the MSB to the right. Unless noted otherwise, binary numbers always place the MSB to the left.

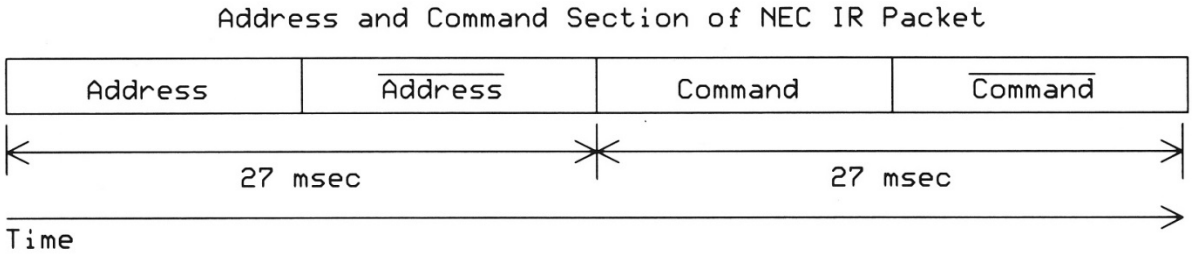


Figure 18.13.
The contents of the address-and-command section of an NEC-compatible transmission includes an 8-bit address and an 8-bit command. The bitwise complements of each byte maintains a constant period and helps software detect errors in received information.

The NEC specification calls for the address-and-command section of a packet to include the address *and its complement* (bitwise inversion), followed by the command and its complement. So, when you control a device with the address 00001000_2, the bit-wise complement, 11110111_2 must follow it. The command portion of a packet uses the same format. Because the packet includes the complement of the address and command, a receiver may check the data for errors. When a receiving device performs a bitwise-OR operation with the address and its complement, the result should equal 11111111_2.

	00001000_2	Address
OR	$\underline{11110111_2}$	Address Complement
	11111111_2	Result

Add 1 to the result and you should get 00000000_2 when the address has no single-bit error. Any answer except 0 indicates a transmission error.

Instead of sending a logic-0 (off) or a logic-1 (on), the NEC format produces *a logic-1 pulse for every bit followed by a logic-0 pulse*. The length of the logic-0 pulse – 560 microseconds for a logic-0 or 1690 microseconds for a logic-1 – determines the state of the bit sent. This arrangement might sound confusing, so diagram in **Figure 18.14** shows the timing, often called *pulse-distance encoding*.

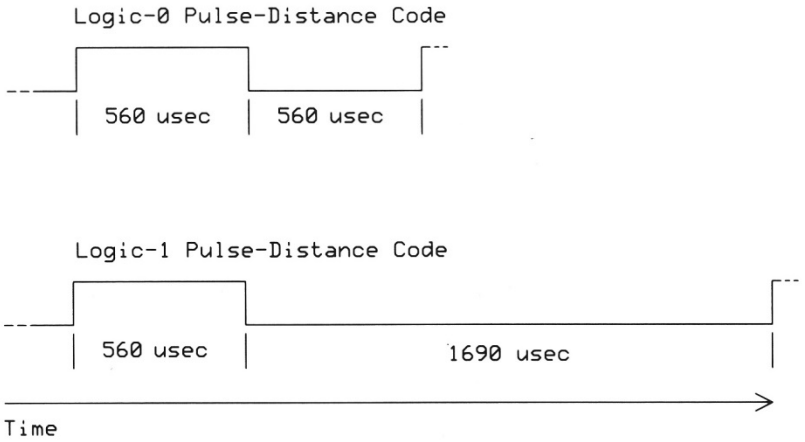


Figure 18.14.
The NEC IR remote-control format uses pulse-distance encoding to represent logic-1 and logic-0 signals. Each bit starts with a logic-1 pulse. The time to the next logic-1 pulse determines the value of the encoded bit. Software can handle the timing and provide received information as byte values.

The information and descriptions above let us decode the address and command sent by the Sparkfun remote control when I pressed the ">" button. Figure **18.15** illustrates the address and data portions of a transmission and shows how to interpret it.

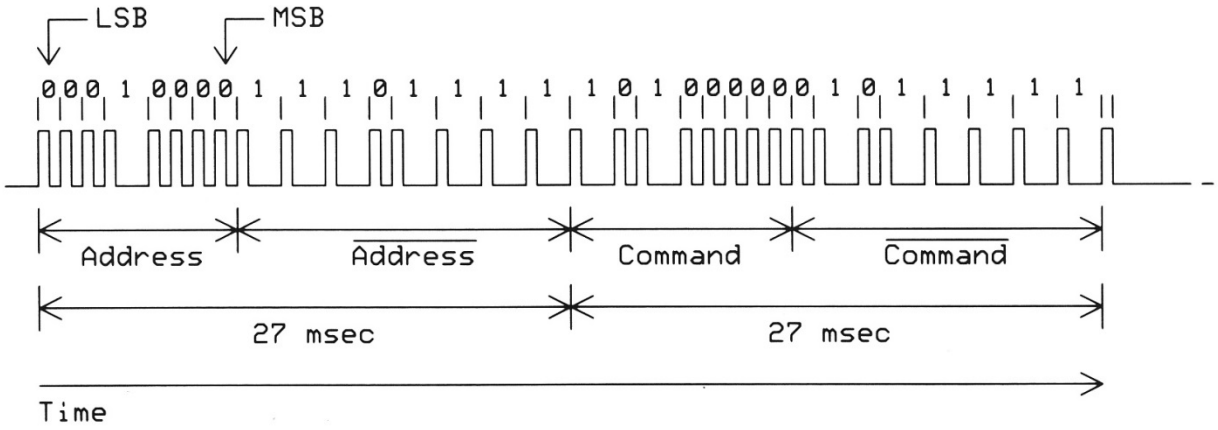


Figure 18.15.
NEC-type encoding of the address and command produced when I pressed the > button on a Sparkfun IR remote control. Remember, this diagram shows an NEC-standard communication.

What address and command bytes do you find in **Figure 18.15**? Look in the Answers section at the end of this experiment.

Because the NEC standard uses pulse-distance encoding, you might think the time needed to send an 8-bit address will vary depending on the number of logic-0 and logic-1 bits it contains. But no matter what 8-bit address gets transmitted, the time needed to transmit it *and* its complement always equals 27 msec. Why? This combined transmission always includes eight logic-1 "distances" and eight logic-0 "distances." You can confirm this situation in the signal shown above.

For the eight logic-0's: $8 * (560\,\mu sec + 560\,\mu sec) = 896\,\mu sec$ or $8.96\,msec$
For the eight logic-1's: $8 * (560\,\mu sec + 1690\,\mu sec) = 18{,}000\,\mu sec$ or $18.0\,msec$.

Total the two results 8.96 milliseconds plus 18.0 milliseconds and you have 26.96 milliseconds, or 27 milliseconds. Write down an 8-bit number and its bitwise complement and see for yourself.

The NEC format has an interesting and helpful way to indicate when someone holds a button for longer than it takes to send a packet. Instead of resending address and command information again and again, the transmitter sends a simple and short "repeat" packet until the remote-control button gets released. So if you hold down a button, the receiving device could:

1. Continuing to take the action specified in the preceding address/command packet. For instance, continue to increase or decrease volume, motor speed, and so on.

2. Perform the action once and wait until the user releases the button before proceeding. You want this action for an on-off control.

Figure 18.16 shows the timing relationship between an address-and-command packet and any "repeat" packet or packets. These "repeat" packets reduce power consumption in a battery-powered remote control.

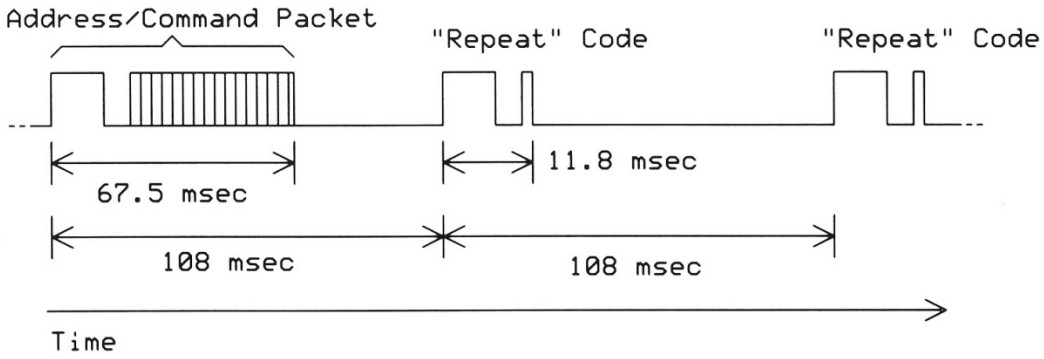


Figure 18.16.
An NEC-compatible remote control transmits "repeat" packets as long as you hold down a button. These packets include the Start Sequence shown earlier in **Figure 18.12**, followed by a single 560 microsecond logic-1 signal. As long as someone holds down a pushbutton, the control will send a repeat packet every 108 milliseconds.

The examples above for an NEC-standard remote control all used the demodulated output of the receiver IC that accepts IR signals and provides logic signals equipment can then process. You might like to see the type of modulated signal the LED transmits, so I cut apart a Sparkfun remote control to expose the IR LED connections for scope probes. The scope trace in **Figure 18.17** illustrates the signal that drives the IR LED in the remote control.

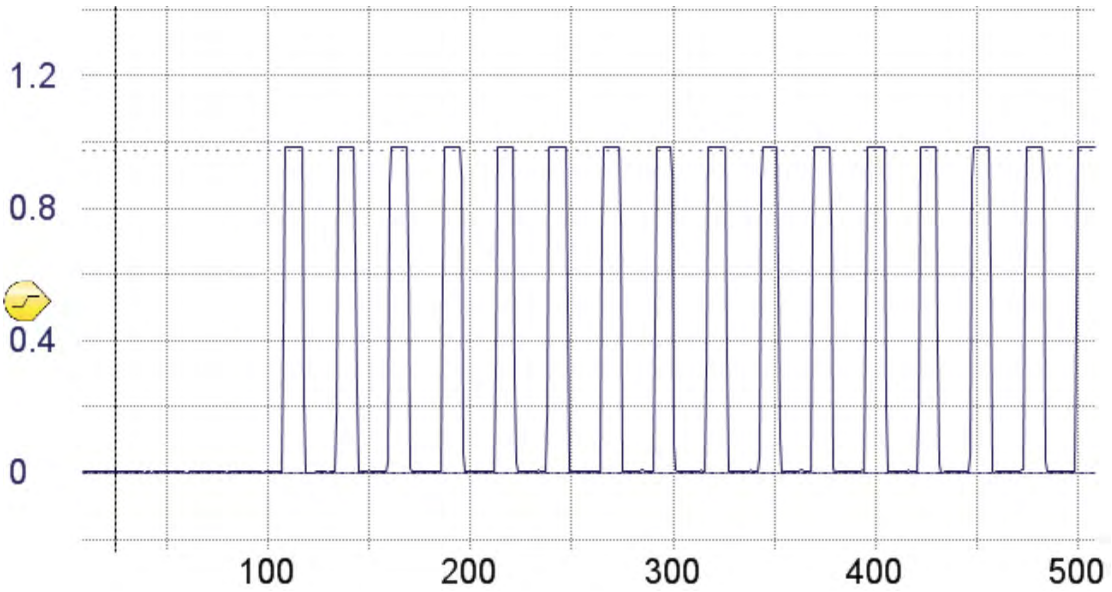


Figure 18.17.
This oscilloscope trace shows a portion of the signal that drives the IR LED in a Sparkfun remote control. Remember, this image shows the modulated signal output, not the transmitted data. I measured a frequency of 38.4 kHz. The horizontal axis has units of microseconds and the vertical axis has units of 0.2 volts per division.

You can program an MCU to generate the properly timed signals to send remote-control IR signals in one of the formats noted earlier. And MCU manufacturers might have applications information and ready-to-use code available. Check the Propeller Object Exchange site: http://obex.parallax.com/. Also, Adafruit has a library of code for the Arduino (https://github.com/adafruit/Adafruit-NEC-remote-control-library) and for the Raspberry Pi computer. I found code and information about an NEC IR application for the mbed community (https://mbed.org/users/vin_jm/notebook/ir-remote-control/.)

IMPORTANT CORRECTION: The Sparkfun remote control sends 32 bits of information that comprise an address (first byte) and the bitwise complement of the address (second byte), followed by the command (third byte) and the bitwise complement of the command (fourth byte). The remote-control documentation notes all "codes" begin with 10EF, but the Sparkfun documentation *incorrectly* assumes the remote-control data sends the most-significant bit (MSB) first in each of the four bytes (32 bits).

When properly interpreted, the Sparkfun "address" of 10EF hex (00010000 11101111_2) becomes address 08 hex (00001000_2), followed by its complement, F7 hex (11110111_2). Likewise, the documentation shows a two-byte code for each button rather than a single-byte command and a single-byte complement. According to Sparkfun, the code 807F (10000000 01111111_2) corresponds to the ">" pushbutton. Actually, this represents the command 01 hex (00000001_2), followed by its complement FE hex (11111110_2).

Software Decodes Infrared Remote-Control Signals

Now, back to hands-on experimentation. In this section of Experiment 18 you'll use software to decode an IR signal sent from a remote control to an IR decoder that connects to a Parallax Propeller MCU. You will learn how to use the address and command information in an NEC-protocol packet to send commands to a specific device and have that device do something. In a small robot, for example, an IR remote control could turn servo motors on or off to change the robot's direction or to lift or lower an arm. To avoid confusion with earlier experiment sections, I start this section of Experiment 18 with its own set of step numbers.

Step 4.

This portion of Experiment 18 assumes you have a NEC-compatible IR remote control. I used a remote control from Sparkfun Electronics. You do not need to know the address or the commands issues by this control.

Turn off power to your breadboard. If you have components in the breadboard, you may remove them now. Place a Vishay TSOP38238 IR sensor in your breadboard and connect it as shown in **Figure 18.18**. The "bump" on the sensor should point outward so it will "see" IR light from your remote control. Although the sensor operates from a 5-volt power supply, the 2200-ohm resistor (red-red-red) limits current into the Propeller MCU and makes the 5-volt sensor signals compatible with a 3.3-volt input on the MCU. Connect the sensor output to the Propeller P8 signal at pin 9 on the expansion connector.

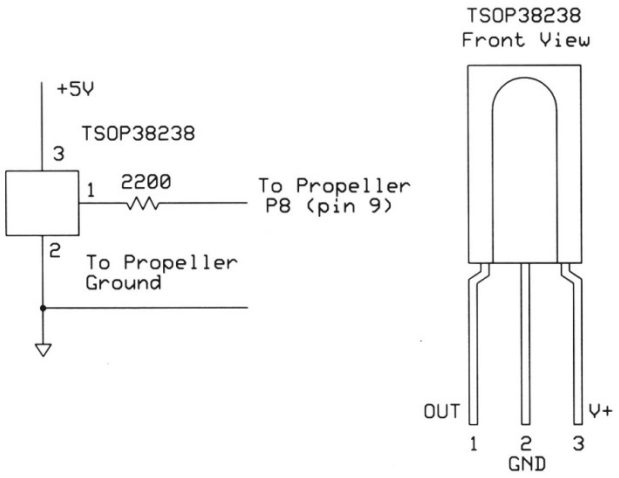


Figure 18.18.
The connection between a Vishay TSOP38238 IR sensor with a 5-volt supply and a Parallax Propeller MCU at 3.3 volts requires a 2200-ohm resistor. The resistor limits current into the MCU and thus provides a signal compatible with the Propeller IC.

Several years ago, Beau Schwabe, an integrated-circuit layout engineer formerly with Parallax, posted the following information in an online forum to answer a question about 5- and 3.3-volt logic levels and a Propeller IC:

> With 5V directly applied to an input on the Propeller, the VDD substrate diode will be stressed. The PN junction of the diode only wants to "see" a maximum of 0.4V. More than this and you will prematurely shorten the life of the Propeller.
>
> To solve this, all you need is a current limiting resistor in series with the I/O. This in turn will form a "voltage divider" with the substrate diode.
>
> In most applications a 1 K [1000-ohm] resistor will work just fine, limiting the current to 1.3 mA from a 5V input. Where you might want to consider another option is if the input speed is very high. The I/O pins also exhibit a small capacitance which in conjunction with an input resistor form a low pass filter.

In other words a very high frequency on the input "might" get filtered out and not seen by the Propeller. (Ref. 4.)

I recommend a 2200-ohm resistor in the circuit shown in **Figure 18.18**. Remember to connect the ground on the breadboard to the ground connection on the Propeller expansion connector, pin 39.

Step 5.

Because human eyes cannot see IR light, we must ensure the TSOP38238 IR sensor receives information from a IR remote control. Bob Belleville contributed several IR-receiver programs to the Parallax OBEX, one of which uses the Parallax Serial Terminal to indicate IR activity at the attached TSOP38238 IR sensor. The program detects IR signals modulated at 38 kHz, but it does not decode them. The Experiment 18 folder includes Belleville's program, ir_oscope.spin as well as the required FullDuplexSerial.spin and timing.spin objects. I modified the ir_oscope program slightly to provide consistent results in the PST window and to start the display only after the Propeller MCU detects the first IR communication. **Program 18.1** includes these revisions.

Program 18.1.

```
''*****************************************************
''*  Program ir_oscope.spin, Rev. 2.
''*  Created by Bob Belleville
''*  This object receives input from an IR remote
''*  and produces a stream of "." and "0" on the Parallax
''*  Serial Terminal to give a quick and simple check of
''*  Propeller connections and the IR remote.
''*  Using a period instead of a 1 makes the 0s stand out.
''*  Modified by Jon Titus 10-17-2013 and re-commented.
''*  Period replaced with a dash.
''*  See the readme.pdf file in the IR Kit OBEX zip file
''*  for more documentation.
''*****************************************************
{{
''*********************************************
''*  Program 18.1
''*  Author: Jon Titus 12-02-2013 Rev. 2
''*  Titus Revisions Copyright 2013
''*  Substituted a dash "-" for the period in original
''*  Fixed problem with synchronization of serial-receive
''*  code to ensure stable results.
```

```
''*  Uses P8X32A Propeller-I board.
''*  SET BAUD RATE AT 9600 IN PST!!!
''*****************************************
}}
CON
  _clkmode  = xtal1 + pll4x
  _xinfreq  = 5_000_000
  irpin     = 8              'IR receiver input on P8 to
                                 'P8X32A board

VAR
  byte PropPin_SerOut         'Variable for P8X32A pin P30
  byte PropPin_SerIn          'Variable for P8X32A pin P31
  byte PropPin_SerMode        'Variable for comm format
  word SerPort_BaudRate       'Variable comm bit rate

OBJ
  term    : "FullDuplexSerial"   'Use Parallax Serial Terminal
  delay   : "timing"

PUB start
  dira[irpin] := %0           'Set P8 for input

  'Information for serial-communication with PST
  PropPin_SerOut    := 30     'P30 USB transmit
  PropPin_SerIn     := 31     'P31 USB receive PC
  PropPin_SerMode   := 0      'Invert RX mode.
  SerPort_BaudRate  := 9600   '9600 bits per second

  'Serial-communication start-up operation

  term.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode,
SerPort_BaudRate)

  repeat while ina[irpin] == 1    'wait here until first logic-0
                                  'pulse detected from IR sensor

 'IR-input test loop; run "forever"
 repeat
    if ina[irpin]             'test input on irpin, P8
      term.tx(45)             'no input, print a dash (ASCII 45)
    else                      'OK, some IR data present
      term.tx(48)             'so print a zero (ASCII 48)

 ' –  – - end Program ir_oscope.spin –  – -
```

Turn on power to your breadboard circuit. Locate the Experiment 18 software folder, open **Program 18.1** in the Propeller Tool window, and run it (press F10). Then, open the PST window (press F12) and *set the Baud Rate to 9600*. Next, click the PST Enable control and then click the Clear control to ensure you start with a "blank" display area. Now aim your IR remote control at the TSOP38238 IR sensor and press one button at a time. The first button you press starts the display of dashes and "0"s in the PST window.

After the first button press the PST should display a series of dashes until the Propeller MCU receives signals from the IR sensor, indicated by 0's within the dashes. **Figure 18.19** shows what I observed when I ran the program. This type of information indicates the sensor detected modulated IR light from your remote control.

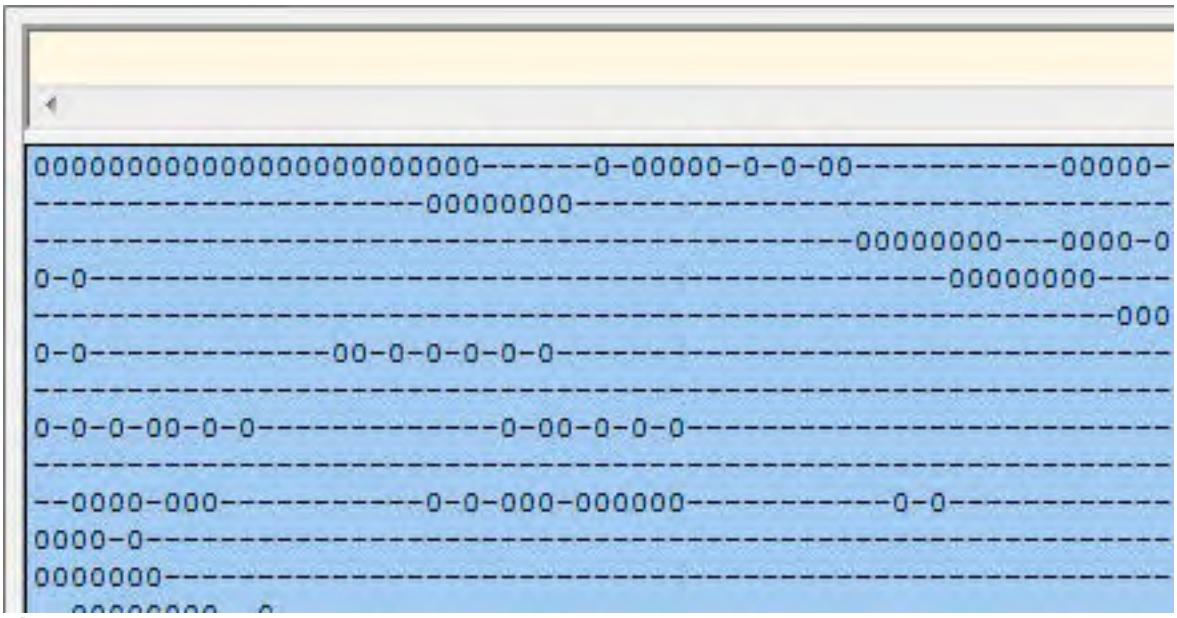


Figure 18.19.
A simple Propeller program displays zeros only when it receives signals from a 38-kHz IR sensor. The program does not decode the received information; it simply lets you know the remote control has sent a signal to the IR receiver and the Propeller MCU detected this activity.

If you cannot see the display of dashes along with 0's when you press a remote-control key, check the wiring of your sensor. Ensure you have +5 volts at your breadboard circuit and a common ground connection between the Propeller QuickStart board and the breadboard. Also check your sensor connections and ensure you have a battery in the remote control. Have you aimed the remote control at the proper IR sensor? You might still have the IR detector used in earlier steps in a breadboard.

Step 6.

After you use **Program 18.1** to confirm your sensor detects a 38-kHz modulated signal, you can use another program to decode serial data into address and command bytes. The Propeller OBEX library includes two programs written by Jon McPhalen to decode IR signals sent as NEC-protocol packets. The receiving program, jm_nec6121_rx.spin will handle the stream of logic-0 and -1 signals from the IR sensor and convert them into a series of 32 bits that include the address and command bytes, and their complements. The demonstration program, jm.nec6121_rx_demo.spin, displays the 32 bits as eight hexadecimal digits. The demo program also relates codes to a remote control's functions and displays a corresponding message.

I modified McPhalen's software, which you'll work with next (**Program 18.2**) so it now displays the address and command bytes, and it uses the complements of each to check for bit errors. Should a bit error occur, the MCU displays an error message. If the MCU cannot match a command to one within the software, it displays a "?" and the unknown command code. (If you want McPhalen's original code, please go to the Propeller Object Exchange at: http://obex.parallax.com/object/316.)

Within the Experiment 18 folder, locate and open programs jm_nec6121_rx.spin and **Program 18.2**. Keep the latter program open in the Propeller Tool window and press the keyboard F10 key to start the program. Press the F12 key to open the PST window. **Change the Baud Rate for the PST to 115,200.** Then clear the PST window, enable the PST, and press keys on the remote control. **Figure 18.20** shows the output in the PST window after I pressed three buttons on the Sparkfun remote control. If you wish, skip past the **Program 18.2** listing and review it later.

Figure 18.20.
Program 18.2 produced these results when I pressed three keys in sequence on the Sparkfun remote control. The information describes the pressed key and shows the address and command associated with it.

Program 18.2.

```
{{
'''*********************************************
'''*  Program 18.2
'''*  Author: Jon Titus 12-02-2013 Rev. 2
'''*  Titus Revisions Copyright 2013
'''*  Program used to obtain a 32-bit packet in NEC
'''*  infrared (IR) remote-control format. This program
'''*  extracts the command and address information, checks
'''*  for bit errors and displays the associated pressed
'''*  key information.
'''*  Uses P8X32A Propeller-I board.
'''*  SET BAUD RATE AT 115,200 IN PST!!!
'''*********************************************
}}
'' ============================================================
''   File....... jm_nec6121_rx.spin
''   Purpose.... NEC6121 IR protocol receiver demo
''   Author..... Jon "JonnyMac" McPhalen (aka Jon Williams)
''   E-mail..... jon@jonmcphalen.com
''   Updated.... 27 FEB 2010
''
''  Modified, expanded, and comments added by Jon Titus
''  December 2, 2013.
'' ============================================================

'Set up clock frequency and calculate number of clock cycles
'in a millisecond.

CON
```

```
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  'Get  pll16x constant of 100_0000_0000 (binary) and shift
  'bits to the right to get 1_0000 (binary), or decimal 16.
  'Multiply by clock frequency to get 80,000,000 Hz

  CLK_FREQ = ((pll16x) >> 6) * _xinfreq

  'Divide by 1000 to get clock "ticks" for a one-millisecond
  'period (80,000)
  MS_001   = CLK_FREQ / 1_000

'Set up data for remote-control unit in use (here Sparkfun 9-
'button IR remote control). Use only the command byte, not
'the entire 32-bit data packet to identify a pressed key.

CON
  KEY_ON_OFF  = $1B           'Key name and key code
  KEY_A       = $1F
  KEY_B       = $1E
  KEY_C       = $1A
  KEY_UP      = $05
  KEY_DOWN    = $00
  KEY_RIGHT   = $01
  KEY_LEFT    = $08
  KEY_CENTER  = $04

  CR          = 13          'ASCII code for a new line in PST

VAR
    byte address            'storage for address
    byte address_comp       'storage for address-complement
    byte address_check      'storage for result of address
                                 'bit-error check
    byte command            'storage for command
    byte command_comp       'storage for command-complement
    byte command_check      'storage for result of command
                                 'bit-error check
OBJ
  ir   : "jm_nec6121_rx"    'Object to decode NEC packet
  term : "fullduplexserial" 'Object to control PST

'Main program starts here
PUB main | code
  ir.init(8)                            'IR-sensor input on P8
  term.start(31, 30, %0000, 115_200)  'Start PST with 115,200
                                        'bits per second
  pause(1)                              '1-msec delay

  repeat                                'Run repeat loop forever
    code := ir.rx                       'Get rcvd addr and command
                                        'via the
                                        'jm_nec6121_rx.spin object
    address := code & $FF               'Extract address
    address_comp := (code >> 8) & $FF 'Extract addr complement
    command := (code >> 16) & $FF     'Extract command
```

```
    command_comp := (code >> 24) & $FF 'Extract cmd complement

    'Print legend for corresponding received command from
    'Sparkfun remote control. These translations use only the
    'command byte, not the entire 32-bit packet.
    case command
      KEY_ON_OFF  : term.str(string("On/Off......... "))
      KEY_A       : term.str(string("A.............. "))
      KEY_B       : term.str(string("B.............. "))
      KEY_C       : term.str(string("C.............. "))
      KEY_UP      : term.str(string("Up............. "))
      KEY_DOWN    : term.str(string("Down........... "))
      KEY_RIGHT   : term.str(string("Right.......... "))
      KEY_LEFT    : term.str(string("Left........... "))
      KEY_CENTER  : term.str(string("Center......... "))
      other       : term.str(string("Key Error...... "))

    'Print 32-bit hex code as received
    term.hex(code, 8)
    term.tx(CR)                         'Go to a new line on PST

    'Print hex values for command and address information
    term.str(string("Address:  "))      'Print hex address value
    term.hex(address, 2)
    term.tx(CR)
    term.str(string("Address Complement  "))
    term.hex(address_comp, 2)
    term.tx(CR)
    term.str(string("Command  "))
    term.hex(command, 2)
    term.tx(CR)
    term.str(string("Command Complement  "))
    term.hex(command_comp, 2)
    term.tx(CR)
    term.tx(CR)

    'Check address and command information for a bit error
    'Add address, address-complement, and 1
    address_check := 1 + address + address_comp

    'Add command, command-complement, and 1
    command_check := 1 + command + command_comp

    if (address_check <> 0)         'If address_check not zero
       term.str(string("Address Error"))
    if (command_check <> 0)         'If command_check not zero
       term.str(string("Command Error"))

    pause(250)  'Wait 250 msec for the release of the
                'pressed button

'==========
'Millisecond timing routine
PUB pause(ms) | t               'Local variable t

  t := cnt                      'Get System Counter value
  repeat ms                     'Repeat this loop for each msec
```

```
    waitcnt(t += MS_001)        'Add MS_001 value to t each time
                                'through this loop
' – -end of Program 18.2 –  – -

{{
Original program Copyright (c) 2010 Jon McPhalen (aka Jon Williams).
Revisions to original program Copyright (c) 2014, Jonathan A. Titus
}}
```

Step 7.

Program 18.2 introduced a new statement: `case`. This statement can greatly simplify a program when you need to take an action based on a value or condition. It eliminates many of the `if-else-elseif` statements you would otherwise need. Refer to the latest Propeller Manual for a complete explanation and examples.

Although **Program 18.2** can detect bit errors in the command and address information, it lacks steps that check the address for a match with an address preset in the Propeller program. If you have a remote control that produces address 08, for example, you might want only a device with that address to respond to commands. Can you think of a way to include simple statements in **Program 18.2** that only proceeds if a received address matches the one preset in the code? Find a suggestion in the Answers section at the end of this experiment.

Step 8.

Program 18.2 displays information in the PST window, and with a few modifications it could control I/O pins based on keyboard commands. The Propeller P8X32A board includes eight LEDs, so I modified **Program 18.2** to turn on the rightmost LED (P16), shift LEDs back and forth, and clear all LEDs. Run **Program 18.3** and use the remote control to change the LED pattern. If you wish, remove the commands that print the address and command information in the PST window.

Program 18.3.

```
{{
''*********************************************
''*  Program version 18.3.spin
''*  Author: Jon Titus 12-02-2013 Rev. 2
''*  Titus Revisions Copyright 2013
''*  Program used to obtain a 32-bit packet in NEC
''*  infrared (IR) remote-control format. This program
''*  lets you turn LEDs on and move them back and forth.
''*  Uses P8X32A Propeller-I board.
''*  SET BAUD RATE AT 115,200 IN PST!!!
''*********************************************
}}
'' =================================================================
''
''   File....... jm_nec6121_rx.spin
''   Purpose.... NEC6121 IR protocol receiver demo
''   Author..... Jon "JonnyMac" McPhalen (aka Jon Williams)
''   E-mail..... jon@jonmcphalen.com
''   Updated.... 27 FEB 2010
''
''  Modified, expanded, and comments added by Jon Titus
''  December 2, 2013.
''
'' =================================================================


'Set up clock frequency and calculate number of clock cycles
```

```
'in a millisecond.

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  'Get  pll16x constant of 100_0000_0000 (binary) and shift
  'bits to the right to get 1_0000 (binary), or decimal 16.
  'Multiply by clock frequency to get 80,000,000 Hz

  CLK_FREQ = ((pll16x) >> 6) * _xinfreq

  'Divide by 1000 to get clock "ticks" for a one-millisecond
  'period (80,000)
  MS_001   = CLK_FREQ / 1_000

'Set up data for remote-control unit in use (here Sparkfun 9-
'button IR remote control). Use only the command byte, not
'the entire 32-bit data packet to identify a pressed key.

CON
  KEY_ON_OFF  = $1B              'Key name and key code
  KEY_A       = $1F
  KEY_B       = $1E
  KEY_C       = $1A
  KEY_UP      = $05
  KEY_DOWN    = $00
  KEY_RIGHT   = $01
  KEY_LEFT    = $08
  KEY_CENTER  = $04

  CR          = 13            'ASCII code for a new line in PST

VAR
    byte address              'storage for address
    byte address_comp         'storage for address-complement
    byte address_check        'storage for result of address
                              'bit-error check
    byte command              'storage for command
    byte command_comp         'storage for command-complement
    byte command_check        'storage for result of command
                              'bit-error check
    byte LED_pattern          'storage for LED pattern

OBJ
  ir   : "jm_nec6121_rx"      'Object to decode NEC packet
  term : "fullduplexserial"   'Object to control PSTP

'Main program starts here
PUB main | code
  ir.init(8)                                'IR-sensor input on P8
  dira[23..16] := $FF                       'Set LED pins for output
  outa[23..16] := $00                       'Turn all 8 LEDs off
  LED_pattern := 0                          'Initial LED pattern
  term.start(31, 30, %0000, 115_200)        'Start PST
  pause(1)                                  '1 millisecond
```

```
repeat                                  'Run repeat loop forever
  code := ir.rx                         'Get rcvd addr and cmd
  address := code & $FF                 'Extract address
  address_comp := (code >> 8) & $FF
  command := (code >> 16) & $FF         'Extract command
  command_comp := (code >> 24) & $FF

  'Print legend for corresponding received command
  'and control LEDs with Sparkfun remote control

  case command
    KEY_ON_OFF  :
       LED_pattern := LED_pattern ^ %00000001 'turn on P16 LED
       outa[23..16] := LED_pattern
    KEY_A       : term.str(string("A.............. "))
    KEY_B       : term.str(string("B.............. "))
    KEY_C       :                       'turn off all LEDs
      LED_pattern := 0
      outa[23..16] := LED_pattern
    KEY_UP      : term.str(string("Up............. "))
    KEY_DOWN    : term.str(string("Down........... "))
    KEY_RIGHT   :                       'shift LEDs right
      LED_pattern := LED_pattern >> 1
      outa[23..16] := LED_pattern
    KEY_LEFT    :
      LED_pattern := LED_pattern << 1  'shift LEDs left
      outa[23..16] := LED_pattern
    KEY_CENTER  : term.str(string("Center......... "))
    other       : term.str(string("Key Error...... "))

  'Print addr and command codes
  term.hex(code, 8)
  term.tx(CR)
  term.str(string("Address:  "))
  term.hex(address, 2)
  term.tx(CR)
  term.str(string("Address Complement  "))
  term.hex(address_comp, 2)
  term.tx(CR)
  term.str(string("Command  "))
  term.hex(command, 2)
  term.tx(CR)
  term.str(string("Command Complement  "))
  term.hex(command_comp, 2)
  term.tx(CR)
  term.tx(CR)

  'Check address and command for a bit error
  address_check := 1 + address + address_comp
  command_check := 1 + command + command_comp
  if (address_check <> 0)
      term.str(string("Address Error"))
  if (command_check <> 0)
      term.str(string("Command Error"))

  pause(250)                                '250 msec delay
```

```
'==========
'Millisecond timing routine
PUB pause(ms) | t
  t := cnt
  repeat ms
    waitcnt(t += MS_001)

' – -end of Program 18.3 –  – -

{{ Original program Copyright (c) 2010 Jon McPhalen (aka Jon Williams).
Revisions to original program Copyright (c) 2013, Jonathan A. Titus.
}}
```

Program 18.3 uses an LED-control command `outa[23..16] := LED_pattern`, four times within the `case` section. Could you: **a)** modify the code to use this statement only once and **b)** modify the program so a remote-control button would selectively turn off an LED at P16? Find a suggestion in the Answers section at the end of this experiment. I'll let you determine how to make the LED pattern "circulate" through the LEDs. That is, as the LED "moves out" the left side of the LEDs, it should shift into the right side, and *vice versa*.

How far can you move away from the IR sensor and still control the LEDs? I could still control the LEDs from about 20 feet away.

Conclusion

This experiment showed you how sensors can detect IR light and decode pulses of IR light from a remote control. The control used here complies with the NEC standard, but other manufacturers have their own codes and standards. You can find timing information, software, and hardware specifications and circuits for these other formats on the Internet.

The next experiment builds on the knowledge gained here and shows how you can use IR light to transfer any information, not just addresses and commands, to other devices.

Reference

1. "Inverse-square law," Wikipedia. http://en.wikipedia.org/wiki/Inverse-square_law.

2. "Hot mirror," Wikipedia. http://en.wikipedia.org/wiki/Hot_mirror.

3. Titus, Jon, "Encoders Know all the Angles," *Design News*, January 1, 2013. http://www.designnews.com/author.asp?section_id=1419&doc_id=256713.

4. "Propeller Chip 5V tolerant?," http://forums.parallax.com/showthread.php/85841-Propeller-Chip-5V-tolerant.

Find data sheet for the Vishay Semiconductors TSOP38238-family IR receiver ICs at: https://www.sparkfun.com/datasheets/Sensors/Infrared/tsop382.pdf.

For more information about the NEC IR remote-control protocol and timing, see the “Pulse Distance Protocol" section of the Freescale Semiconductor application note, AN3053, “Infrared Remote Control Techniques on MC9S08RC/RD/RE/RG Family,” at: http://cache.freescale.com/files/microcontrollers/doc/app_note/AN3053.pdf.

For information about a state-machine approach to decoding the NEC IR data packets, see, "SB-Projects, NEC Decoder," at http://www.sbprojects.com/projects/ircontrol/nec.php.

Answers

Experiment 18, Step 1:

Knowns:

1. LED forward voltage (V_F) = 1.2 volts
2. Power supply voltage = 5.0 volts
3. Current through the LED = 30 mA or 0.030 A

Given this information and Ohm's Law, the resistor will have a 3.8-volt potential across it:

$$(5.0\,V\ -\ 1.2\,V)\ =\ 3.8\,V$$

Next:

$$I\ =\ E\ /\ R \text{ or } R\ =\ E\ /\ I$$

$$R\ =\ 3.8V\ /\ 0.030\,A\ =\ 126\,ohms$$

You can buy a 130-ohm resistor, but it's more likely you'll find a 120-ohm resistor (brown-red-brown). When I used a 120-ohm series resistor the current came to 32 mA.

Experiment 18, Infrared Remote-Control Signals, A Tutorial:

*What address and command bytes do you find in the information shown in **Figure 18.15**?*

Remember, the address and command bytes place the LSB on the left and the MSB on the right. So, the address equals 00001000_2, or 08 in hexadecimal. The command equals 00000101_2, or 05 hex.

Experiment 18, Step 4.

In **Program 18.2**, set the address of the device as $08 in the CON section:

```
device_address = $08
```

After the program gets an IR code and extracts the address in these two statements:

```
code := ir.rx
address := (code >> 8) & $FF
```

Compare it with your preset address. If the MCU detects a match (your address = preset address), it continues with the rest of the code. If the MCU does not detect an address match, the code points back to the `repeat` statement. An `if` statement and proper indentation of the rest of the program would do the job:

```
code := ir.rx
address := (code >> 8) & $FF
if (address == device_address)
    address_comp := (code >> 8) & $FF
    etc...
```

Try it.

Experiment 18, Step 5:

a) Remove the four `outa[23..16] := LED_pattern` statements within the `case` section of code, and insert a single `outa[23..16] := LED_pattern` statement right after the end of the `case` section. Then, when the code exits the `case` statement section, it will always update the LED pattern.

b) Select an unused remote-control button and use the corresponding statements in the `case` portion of the code to selectively turn off the rightmost LED, just as the statements for the case of `KEY_ON_OFF` turns an LED on. Use a bitwise operation with the `LED_pattern` variable to turn off only the P16 LED.

Experiment No. 19 – How to Create and Use 2-Way Infrared Communication

Abstract

This experiment lets you work with infrared (IR) communications between devices. This type of communication goes beyond the single-direction address-and-command communications illustrated in Experiment 18. The flexible circuits and software in this experiment let you create 2-way data transfers of any type. This experiment comprises an Introduction and two sections, A and B. The former introduces the use of a modulated IR signal to transmit serial data. Section A expands on IR communications and introduces ICs and modules meant specifically for 2-way information transfers. Components for the Section B circuits cost about $US 50, so unless you have a serious interest in 2-way communications, you can skip building and testing the circuits. I highly recommend you read Section B, though, because it introduces important software concepts you can use in project designs.

Keywords

Infrared, IR, LED, sensor, remote control, 38 kHz, IrDA, encoder, decoder, transceiver, MCP2120, 555 timer, modulation, demodulation, array pointers, duplex

Requirements

Components needed only for Section B shown with an asterisk (*).

(1) - Solderless breadboard
(1) - Solderless breadboard (optional)
(1) - Propeller P8X32A microcontroller board
(1) - USB cable
(1) - 5-volt power supply
(1) - Digital voltmeter or analog voltmeter
(1) - 555 timer IC, 8-pin DIP
(1) - Green LED, wire leads
(1) - Infrared LED, 940 nm, T1¾ size
(1) - Infrared Detector, Vishay TSOP38238 or equivalent.
(1) - 10-ohm, 1/4-watt resistor, 5% (brown-black-black)
(1) - 220-ohm, 1/4-watt resistor, 5% (red-red-brown)
(1) - 680-ohm, 1/4-watt resistor, 5% (blue-grey-brown)
(1) - 1000-ohm, 1/4-watt resistor, 5% (brown-black-red)
(1) - 1800-ohm, 1/4-watt resistor, 5% (brown-grey-red)
(1) - 2200-ohm, 1/4-watt resistor, 5% (red-red-red)
(3) - 10-kohm, 1/4-watt resistor, 5% (brown-black-orange)
(1) - 2000-ohm trimmer resistor, cermet, single turn, top adjustment
(1) - 1N4148 small-signal diode, axial leads
(1) - 0.1 µF, 50V disc-ceramic capacitor
(1) - 0.01 µF, 50V disc ceramic capacitor
(4) - *18 pF, 50V ceramic capacitor
(2) - *JayCon Systems IrDA transceivers, part no. JS-5355
(2) - *Microchip MCP2120 Infrared Encoder/Decoder IC, 14-pin DIP
(2) - *Crystal, 7.3728 MHz, 20 pF (FOXLF073 or equivalent)
(2) - *Header pins, group of six, 90-degrees, 0.1-inch (2.5-mm) centers

Introduction

The previous experiment with infrared LEDs illustrated how a small receiver IC can produce digital signals from a modulated 38-kHz carrier signal. A small remote control let you send commands to a Propeller MCU to change the state of LEDs on the P8X32A board. That type of control works well for simple commands, but at times, devices might require the transfer of data in one direction, called simplex communications, or in two directions, called duplex communications. In this experiment you will work with a circuit and an IC that let you do pretty much anything you need with inexpensive easy-to-use IR receiver/transmitter modules.

This type of communication works well when equipment requires a non-electrical line-of-sight connection, such as during electrical or electronic tests run in a sealed toxic environment. Let's stay away from that situation, though! Or you might need to transfer a software update to a device such as a robot, washing machine, or heating system that lacks a hardwired communication connection. In the hobby world, you might need to update an MCU in a model rocket or in a radio-controlled airplane. Connectors and wires cost more than a small IR receiver and a bit of MCU code.

At first, you might think, why not use a logic signal from a universal asynchronous receiver-transmitter (UART), to turn an IR LED on for a logic-1 or off for a logic-0. This approach runs into problems from ambient light and other signals that can "confuse" an IR sensor. To eliminate, or at least reduce, interfering signals, handheld remote controls use a modulated IR signal at 38 kHz and a special sensor IC such as the one used in Experiment 18.

So to start the design of a communication link with a general-purpose transmitter we need a 38-kHz carrier signal, which either an MCU or a 555-timer IC can produce. The Propeller MCU can run square-wave software available from the Propeller Object Exchange. A 555-timer circuit also offers a way to create a carrier signal appropriate for use with an IR LED. I'll take tackle the 555-timer circuit first.

The usual circuit for a 555-timer circuit (**Figure 19.1**) uses resistors Ra and Rb to charge capacitor C, attached to pins 6 and 2. When the voltage across the capacitor reaches a threshold, pin 7 switches to ground and discharges the capacitor, but only through resistor Rb.

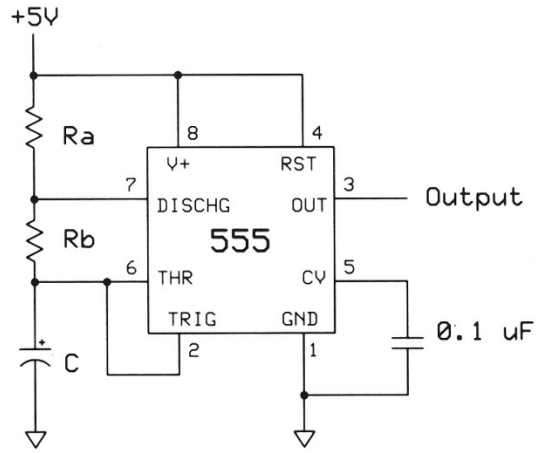


Figure 19.1.
This basic 555-timer circuit uses two resistors and a capacitor to set the on and off periods for the output at pin 3.

You might remember from Experiment 2 that the capacitor-charge time (t1) corresponds to a logic-1 output from the 555 IC. The discharge time (t2) corresponds to a logic-0 output. The following equations show how to calculate these times from the resistance (ohms) and capacitance (Farads) values:

$t1 = 0.693 * (Ra + Rb) * C$ and $t2 = 0.693 * Rb * C$

By choosing a large-value resistance for Ra and a smaller-value resistor for Rb, you can get a logic-1 output at pin 3 that extends from about 51 to 99 percent of the timer-circuit cycle time (**Figure 19.2**). Why wouldn't a 555 timer IC produce a 100-percent duty cycle? That would equal a logic-1 output at all times, which wouldn't give us any timing information.

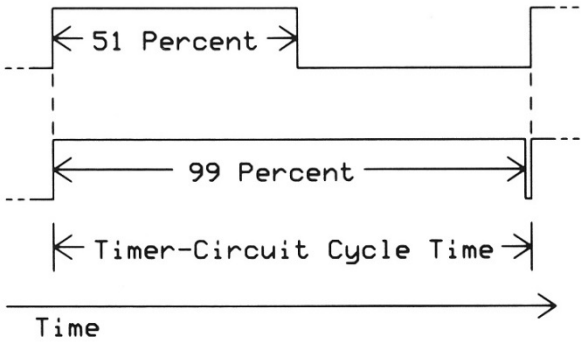


Figure 19.2.
The output from the basic timer circuit shown in **Figure 19.1** can have a logic-1 output that varies between 51 and 99 percent, depending on the values of Ra and Rb.

According to specifications for IR communications, the 38-kHz carrier signal should have a logic-1 period of about 25 percent of the complete cycle time. Such a signal falls below the 51-to-99-percent range. By including a diode in the circuit (**Figure 19.3**) the 555-timer IC can produce a pulse with a shorter duty cycle (Refs. 1 and 2).

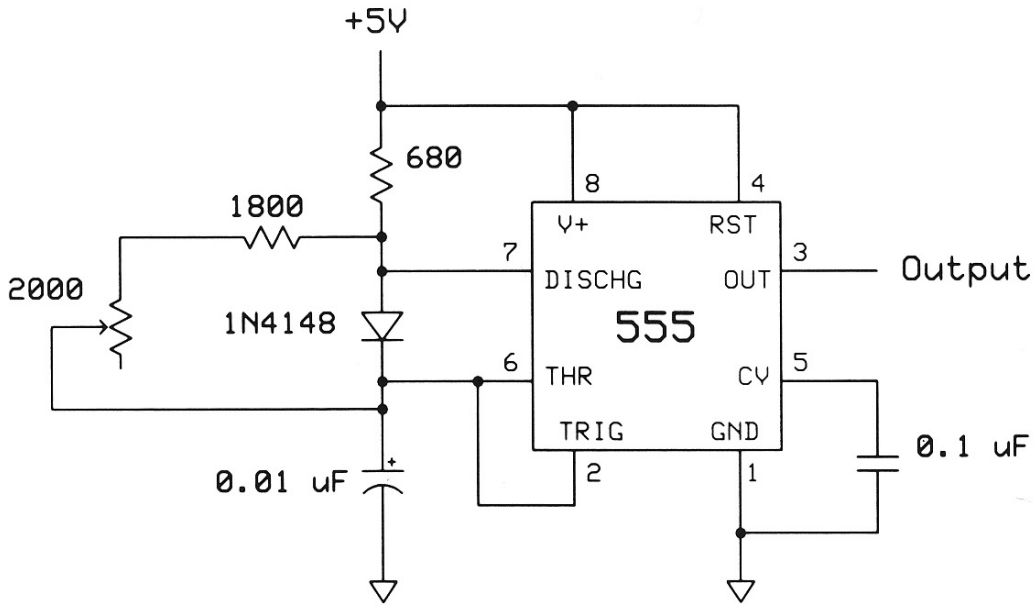


Figure 19.3.
A diode added to a basic 555-timer circuit alters the charge/discharge path for the capacitor between pins 2 and 6, and ground.

A 1N4148 diode, like an LED, conducts current in only one direction, so it can change the way capacitor C charges. In effect, the diode provides a short circuit across the 1800- and 2000-ohm resistors, so only resistance Ra (680 ohms) determines the capacitor charge time. The small resistance for Ra lets the capacitor charge quickly, so the 555 IC will produce a short logic-1 pulse. Because the low-resistance path through the diode lets capacitor-charging current bypass resistance Rb, the equations for t1 and t2 become simpler:

$t1 = 0.693 * Ra * C$ and $t2 = 0.693 * Rb * C$

When the voltage across capacitor C reaches a threshold preset in the timer, pin 7 connects to ground to discharge the capacitor. Now the capacitor discharges through only the 1800- and 2000-ohm resistors because the diode cannot conduct current in the reverse direction. **Figure 19.4** shows the charging and discharging current paths. The 2000-ohm trimmer resistor lets you adjust the frequency of the pulse created by the timer circuit. The value of resistor Ra alone determines the logic-1 period, while resistor value Rb controls the logic-0 period. You can change these values independently to vary the logic-1 pulse width and the overall frequency of the 555-timer output.

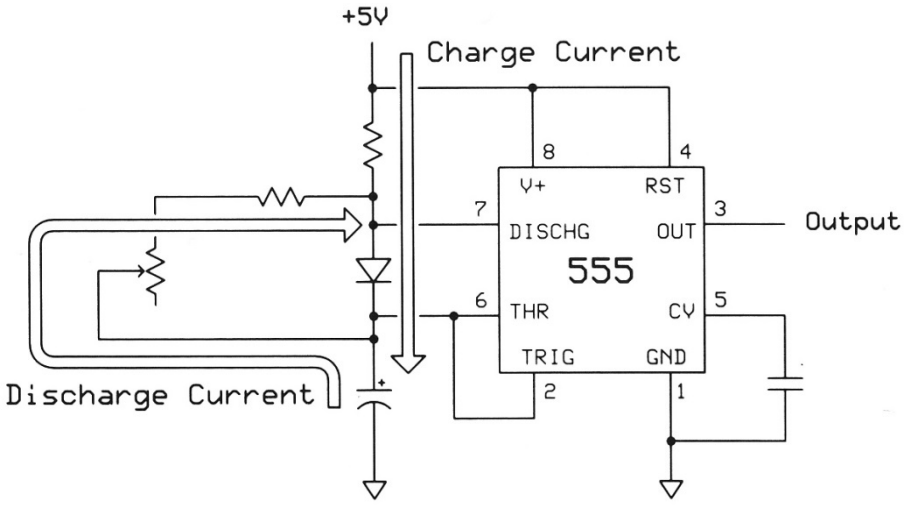


Figure 19.4.
The diode in this circuit isolates the charge and discharge current paths as shown here. (Component values not shown for the sake of clarity.)

You might think, "Can't I set the duty cycle for the logic-1 pulse at 75 percent and then invert the output to get a 25-percent logic-1 duty cycle?" You could, but at the cost of adding an inverter IC to the circuit.

Section A:

Step 1.

If you have components in a breadboard, please remove them now and start with a "clean" work area for a timer circuit. Turn off power to the breadboard and construct the timer circuit shown in **Figure 19.5.** This circuit includes a 2000-ohm trimmer. And it includes a Vishay TSOP38238 IR sensor that will demodulate the 38-kHz signals sent from an IR LED connected to the 555-timer IC. Mount the IR LED and the IR sensor in a breadboard, or breadboards, so they face each other across a short distance. This setup lets you adjust the 555-timer circuit for an output close to the needed 38-kHz modulation frequency. (I used two breadboards, one for the IR LED and and 555-timer circuit, and another for the sensor. This arrangement let me place the IR LED and the sensor about 10 cm apart so I could align them without bending or twisting component leads.)

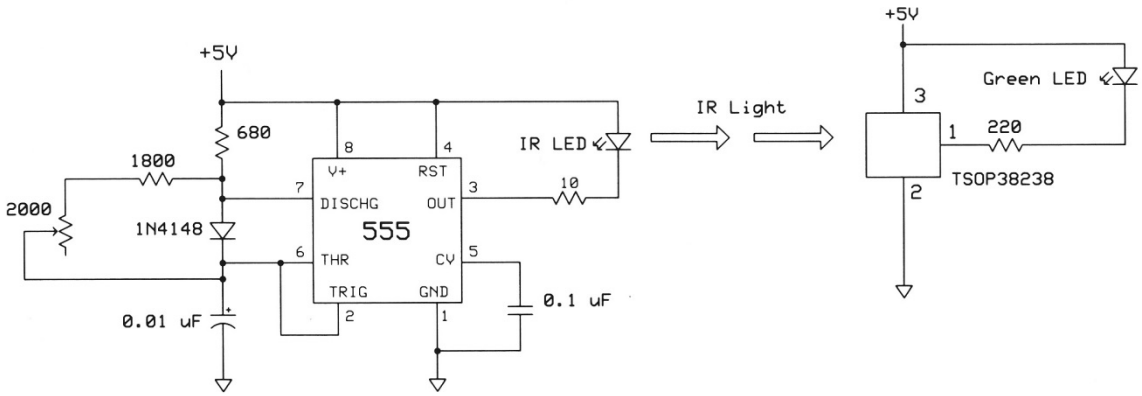


Figure 19.5.
This circuit produces a 38-Khz signal that drives an IR LED. An IR receiver/sensor (TSOP38238) converts this signal into a logic level.

The 2000-ohm trimmer might include an embossed or printed notation, "202," which indicates a resistance equal to "20 with two zeros after it," or 2000 ohms. Or you might see "2K" printed on the trimmer package. Connect one of the end terminals on the 2000-ohm trimmer to the unconnected end of the 1800-ohm (brown-gray-red) resistor in your breadboard (**Figure 19.6**). Connect the middle terminal on the trimmer to pin 2 *or* pin 6 on the 555-timer IC. If trimmer terminals do not fit into the solderless-breadboard receptacles, solder wires to them and insert the free end of the wires in the breadboard. The trimmer shown in **Figure 19.5** needs only two wires; one to the wiper and one end of the resistor. See the trimmer data sheet for terminal information.

Figure 19.6.
Trimmer resistors come in several forms. A screwdriver or fingers can adjust the resistance via a "knob" or small screw. The center terminal connects to the movable wiper to provide a varying resistance. The side terminals connect to each end of the resistive element. A conductive plastic, carbon, wire or ceramic serves as the resistive material. Trimmers have an arrow or tab on the control to indicate the wiper's position. This image includes an 8-pin DIP for size comparison.

Step 2.

Rotate the variable-resistor control – the knob or the screw – to about the midpoint between its two end positions; that is, the points at which the control can rotate no farther. Turn on power to your breadboard circuit. With the trimmer set near its midpoint (for a resistance of about 1000 ohms), I measured a frequency of

38.5 kHz, and the green LED connected to the TSOP38238 turned on. But what if you can't measure the timer-circuit's frequency?

Rotate the trimmer control *clockwise* (CW) until it stops. The green LED should turn off. Then slowly rotate the control *counterclockwise* (CCW) until the green LED turns fully on. Mark this position on the trimmer, or make a drawing to show its position. This position represents one end of the IR sensor's bandwidth.

Next, rotate the control *counterclockwise* (CCW) until it stops. The green LED will turn off. Then slowly rotate the control clockwise (CW) until the green LED turns fully on. Mark this position on the trimmer or make a drawing to show its position. This position represents the other end of the IR sensor's bandwidth.

Finally, rotate the control so it points midway between the two marks that indicate the sensor's bandwidth. The green LED should remain on. At the ends of the bandwidth, I measured frequencies of 41.6 and 35.7 kHz. At the midpoint, the 555-timer circuit produced a 38.4-kHz signal; close enough to 38.0 kHz for experimental work. (You don't always need lab instruments to properly set up circuits.)

Step 3.

To send information to a remote device over an infrared communication "link" we must turn the 38-kHz signal on or off to represent logic-1 and logic-0 states. To test the link, the Propeller MCU can provide an 8-bit UART output along with start and stop bits. A 2-input AND gate with the 38-kHz signal on one input and the UART signal on the other would work nicely to produce the required signals, but that approach requires another IC that can complicate a design – something more to troubleshoot and to take up space.

Fortunately, the 555-timer IC provides a way to directly control the output signal via the Reset input at pin 4. A logic-1 at this pin turns on the timer output and a logic-0 turns it off. (When left unconnected, the timer IC's Reset input floats to a logic-1 state.) The image in **Figure 19.7** show the UART output from a Propeller board and the resulting output from the 555 timer. The modified circuit diagram in **Figure 19.8** shows the UART-output connection to the 555-timer IC at pin 4 and the scope connections.

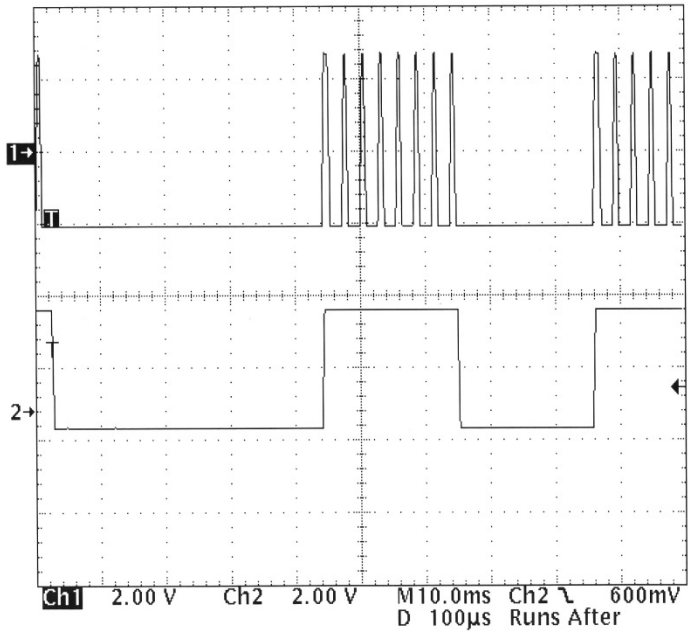


Figure 19.7.
This scope image shows a modulated 38-kHz signal and below it, the modulating UART signal. Both traces have a 2-volt-per-division amplitude and each division along the time axis represents 10 milliseconds. (I used a Tektronix TDS-460A digital-storage oscilloscope for signal displays in this experiment.)

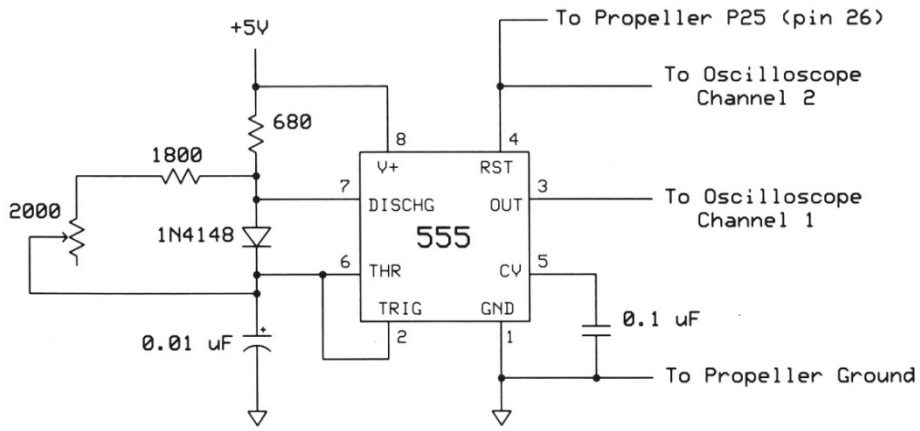


Figure 19.8.
The UART connection to the 555-timer IC at the Reset input (pin 4) lets Propeller software directly modulate the 38-kHz carrier signal with no extra components. The scope connections let me grab the signals shown in **Figure 19.7**.

Step 4.

The 555-timer IC now produces a modulated 38-kHz signal that indicate the logic levels from a UART transmitter. How does the IR sensor react, and what signals does it produce? The image in **Figure 19.9** shows three signals that represent (top to bottom) the Propeller UART output signal, the modulated signal at the IR LED, and the output from the IR sensor. Note that the IR sensor output shows a slight delay when compared with the UART output. Also, the sensor produces an inverted signal, so a logic-1 from the Propeller UART output becomes a logic-0 at the sensor output.

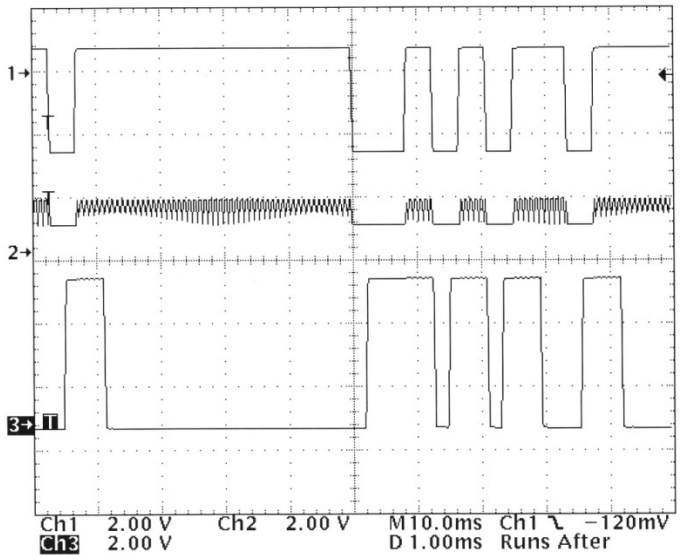


Figure 19.9.
This image shows the UART output (top), the modulated signal at the IR LED (middle), and the IR sensor output (bottom). The modulated signal has a small amplitude due to the low voltage changes across the IR LED. All traces have a 2-volt-per-division amplitude and each division along the time axis represents 10 milliseconds.(For more information about sampling signals and why they show an "aliasing" effect, see the References section at the end of this experiment.)

Program 19.1 sets up a Propeller UART, called UART1, that uses the 555 timer as a 38 kHz modulator and the IR LED. UART1 also receives information from the IR sensor and places the data in the byte variable `char_in`. (Remember, a UART has a receiver *and* a transmitter. They can act independently, but the same bit rate, number of stop bits, and use of a parity bit apply to both.)

A second UART, named `term`, communicates with the Parallax Serial Terminal (PST) at 9600 bits per second. For now, **Program 19.1** transmits a constant byte, 01101010_2, or the ASCII character "j".

Program 19.1.

```
{{
''*********************************************
''*  Program 19.1.spin
''*  Author: Jon Titus 12-12-2013 Rev. 2
''*  Program transmits a byte via UART1. 555 timer modulates
''*  data and drives an infrared (IR) LED. Modulation at
''*  38 kHz. IR sensor Vishay, type TSOP38238. Program
''*  receives byte from IR sensor and
''*  displays byte as an ASCII character.
''*  Note "receiver-inversion bit" set for UART1 in
''*  the UART1.start command.
''*  Uses P8X32A Propeller-I board.
''*  Set PST to 9600 bits/sec.
''*********************************************
}}

'Set up clock frequency
CON
   _clkmode = xtal1 + pll16x      'Set MCU clock operation
   _xinfreq = 5_000_000           'Set for 5 MHz crystal

VAR
    byte char_in                  'Storage for byte received
                                  'from UART1
OBJ
  UART1 : "fullduplexserial"      'Object for IR LED and sensor
  term  : "fullduplexserial"      'Object to send data to PST
  delays : "timing"               'Timing object

'Main program starts here
PUB main
  UART1.start(24, 25, %0000, 2400)     'Start IR LED UART
  term.start(31, 30, %0000, 9600)      'Start PST with these
                                       'settings
  repeat                          'Repeat this loop "forever"
    delays.pause1ms(8)            'Wait 8 milliseconds
    UART1.tx(%01101010)           'Transmit this byte to 555-
                                  'timer IC
    char_in := UART1.rx           'Receive this byte (inverted)
                                  'from IR sensor
    term.tx(char_in)              'Display received byte in PST

 ' – -end of Program 19.1 –  – -
```

Step 5.

The circuit diagram in **Figure 19.10** shows the necessary connections between the Propeller MCU and the breadboard circuit. Turn off power to the breadboard circuits, make the new connections, remove the LED attached to the IR sensor output, turn on power, and run **Program 19.1**.

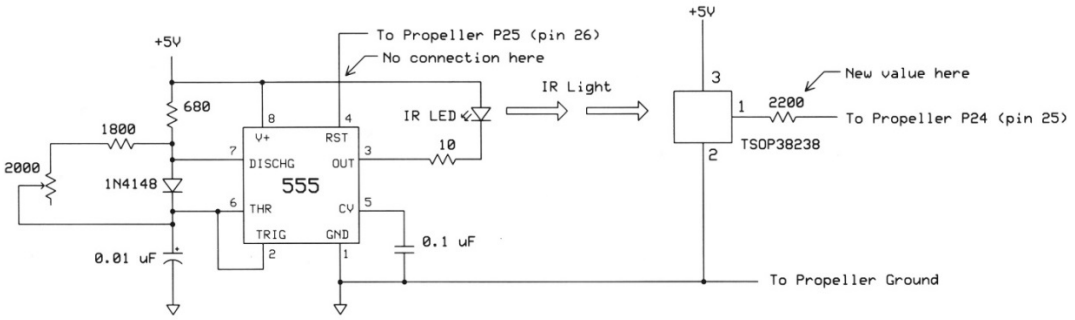


Figure 19.10.
This circuit diagram shows the complete IR-communications circuit and connections to a Propeller P8X32A board. Note the connection between pin 4 on the 555-timer IC and the P24 output on the Propeller board. A 2200-ohm resistor replaced the green LED and 220-ohm resistor. The resistor connects the TSOP32832 sensor with the MCU and ensures compatibility of the sensor's 5-volt logic levels with the 3.3-volt inputs on the Propeller MCU.

After you make the changes to the circuit, recheck your wiring. Ensure you have a ground connection between the breadboard and the Propeller board. When you confirm the proper wiring, turn on power to your breadboard and run **Program 19.1**. Open the PST window (F12) and set the Baud Rate to 9600. Clear the window, and click on the Enable control. Ensure you have the IR LED and the sensor properly aligned and close to each other. What do you see in the window area? I saw nothing. Do you know why?

Remember, the IR sensor *inverts* the transmitted data (see **Figure 19.9**). Before the Propeller's UART receiver can interpret the sensor's output signals a circuit or software must invert them. An inverter IC such as a 74HC04 would do the job, but the FullDuplexSerial.spin object includes a way to invert bits automatically. In **Program 19.1**, the setup statement:

```
UART1.start(24, 25, %0000, 2400)
```

includes four bits, `%0000`, that control how UART1 operates. When set to a logic-1, bit D0 causes the UART to invert bits as it receives them. The D1 bit, when set to a logic-1, causes the transmitter to invert bits as it transmits them. I chose to invert the received bits and changed the setup statement to:

```
UART1.start(24, 25, %0001, 2400)
```

Make this change in **Program 19.1** and restart it. Now what do you see in the PST window? The letter "j" should appear again and again. If you do not see the j's appear, check your wiring again. Ensure you have a ground connection between the Propeller board and the breadboard that holds the IR sensor. Check the Baud Rate setting in the PST window (9600), and check the Enable/Disable control in the bottom-right corner of the PST window. The control should read "Disable," which means you have the PST turned on. If this control reads "Enable," click it. (Yes, these conditions seem backward, but that's how someone configured this PST control.)

If you still do not see the j's in the PST window, don't worry. Move the variable-resistor control slowly between the two positions you marked and see what happens. If the j's still don't appear, recheck your hardware and ensure you have **Program 19.1** running on the Propeller MCU board.

After you see the letter j in the PST window, you can change the value of the transmitted byte to see the results. Then go on to the next step.

Step 6.

Now you will use program **Program 19.2** to compares the byte sent from the Propeller to the byte the Propeller receives from the IR sensor. UART1 *transmits* data to the 555-timer-IC that controls the IR LED.

The IR light gets detected and demodulated in the IR sensor, and finally returned to the Propeller UART1 *receiver*. Software will compare the byte sent with the byte received. When the transmitted and received bytes match, the Propeller displays the corresponding ASCII character. The decimal value 33, for example, prints an exclamation point (!).

Load **Program 19.2** into the Propeller Tool and run it. The PST window should display the ASCII characters one after another. Don't worry if you see the ERROR message instead. Adjust the 2000-ohm trimmer, and watch the results. You should see the ERROR message only when you move the variable-resistor control beyond the points you marked earlier. Move the trimmer control to see what happens. In between these two points, communications should occur without any errors. Move the trimmer control back to the midpoint you marked in earlier steps.

Program 19.2.

```
{{
''*********************************************
''*  Program 19.2.spin
''*  Author: Jon Titus 12-13-2013 Rev. 2
''*  Program transmits a byte via UART1. 555 timer modulates
''*  byte data and drives an infrared (IR) LED. Modulation at
''*  38 kHz. IR sensor Vishay, type TSOP38238. Program
''*  receives transmitted byte from the sensor and
''*  compares received byte with the byte just
''*  transmitted. Bytes displayed as ASCII characters only
''*  when comparison is true. If false, display "ERROR."
''*  Note "receiver-inversion bit" set for UART1 in
''*  the UART1.start command.
''*  Uses P8X32A Propeller-I board.
''*  Set PST to 9600 bits/sec.
''*********************************************
}}

'Set up clock frequency
CON
    _clkmode = xtal1 + pll16x     'Set MCU clock operation
    _xinfreq = 5_000_000          'Set for 5 MHz crystal

VAR
    byte char_in                  'Storage for byte from UART1
    byte index                    'Storage for index value

OBJ
   UART1 : "fullduplexserial"      'Object for IR LED and sensor
   term  : "fullduplexserial"      'Object to control PST
   delays : "timing"               'Timing object

'Main program starts here
PUB main
  UART1.start(24, 25, %0001, 2400)  'Start UART for IR comms
                                    'and invert received bits
term.start(31, 30, %0000, 9600)     'Start PST

 'Main loop to test IR communications
 repeat
   index := 33                    'Start with 33, 1st printable
                                  'ASCII character "!"
   repeat while index < 127       'Repeat this loop until index
```

```
                                'equals 127, ASCII DEL code
     delays.pause1ms(8)         'Wait 8 milliseconds, an
                                'arbitrary delay

     UART1.tx(index)            'Transmit index byte to 555 IC
     char_in := UART1.rx        'Get received byte from UART1
     if (char_in == index)      'Compare sent and rcvd bytes
        term.tx(index)          'Bytes match (TRUE), so
                                'Display received byte as
                                'ASCII character in PST window

     else                       'If no match (FALSE), print
                                'ERROR message
        term.str(String("  ERROR  "))
        UART1.stop                          'Stop UART1
        UART1.start(24, 25, %0001, 2400)    'Re-initialize
                                            'UART1 after error
                                            'detected.
     index++                     'Increment index value by 1

' – -end of Program 19.2 –  – -
```

I found the Propeller UART implemented in the FullDuplexSerial.spin object can become unsynchronized when it receives a transmission at a bit rate that varies from the ones implemented in the code. This condition arose when I moved the trimmer enough to see ERROR messages in the PST window and then moved the trimmer back to its marked position. Although the trimmer resistor alters the modulation signal, not the data, a modulation signal at the edges of the IR sensor's operating bandwidth can cause problems for a receiving UART.

When I ran an early version of **Program 19.2**, after the Propeller detected an error, the program continued to display ERROR, even when I adjusted the trimmer so the 555-timer would produce a good 38-kHz carrier signal. The statements below in the if-else section of the program will stop UART1 and reset it after the code detects an error. Those actions solved the problem:

```
UART1.stop
UART1.start(24, 25, %0001, 2400)   'Re-initialize UART1
```

Step 7.

I ran the software and hardware in my lab for several days without any problems. At a distance of about 120 cm (4 feet) I still had good results. My IR LED had a narrow beam. Moving the LED slightly from side to side in front of the sensor lets you determine the effective "beam width" of the IR LED at various distances from the IR sensor.

The basic UART communications used so far include no error checking such as a parity bit or a checksum. Even if transmitted data included this extra information, devices would need two-way IR communications so the receiving device could "request a resend" from the transmitter. You could use the UART-communications circuit and software from this experiment for 2-way communications, but you would need a receiver and a transmitter at each end.

Section B: Two-Way Infrared Communications

Within this section, I'll describe how you can create an infrared communication channel to transfer information back and forth between two devices. Rather than have you buy components now, I'll explain what you need and provide a demonstration. Then, if you choose, you can build a 2-way IR-communication system when you

need one. The devices used here can communicate over about one meter. Other IR transceiver modules (IR LED, IR sensor, and encoder/decoder in one package) can extend the range to several meters.

In the early 1990's, about 50 companies formed the Infrared Data Association, now known as IrDA. This group established standards for line-of-sight IR communications, and manufacturers of devices such as cellphones, tablet computers, and laptops adopted them. I own an older PDA (personal digital assistant) that includes an IrDA port, although I used this communication capability infrequently. The IrDA standards – available only to IrDA members – include many "layers" that range from low-level hardware specifications up through higher levels that define protocols for a variety of product types.

The popularity of IrDA communications led many companies to manufacture small IR transceiver modules that include an IR LED and an IR sensor. These devices measure about 5-by-2-by-2 mm and have contacts spaced so closely that soldering them by hand proves almost impossible (**Figure 19.11**). I purchased modules with a Vishay TFBS4711 Serial Infrared Transceiver soldered on each one. Two small LEDs indicate activity on the transmit and receive pins. These boards came from JayCon Systems – part number JS-5355 – and they have six plated-through holes on 0.1-inch centers, which makes them ideal for experiments (**Figure 19.12.**) Olimex sells an IrDA module, too, but I have not used one.

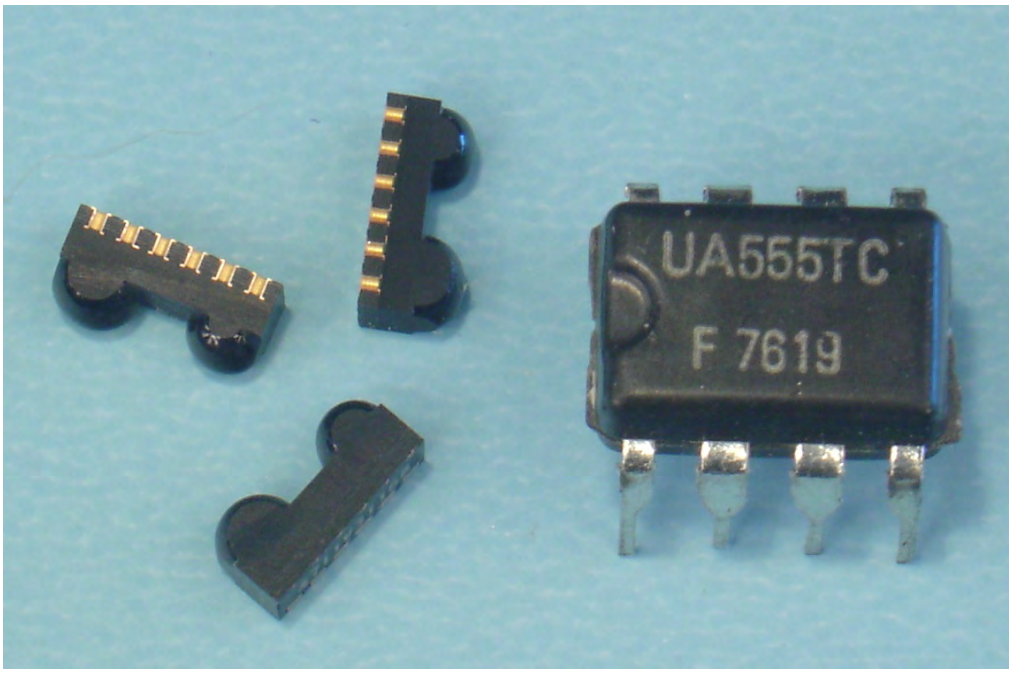


Figure 19.11.
Tiny infrared transceivers such as these have such small contacts that defy soldering by hand. Prebuilt modules that include these types of IR transceivers might cost more, but they reduce frustration and get a project started quickly. This image includes an 8-pin DIP for size comparison.

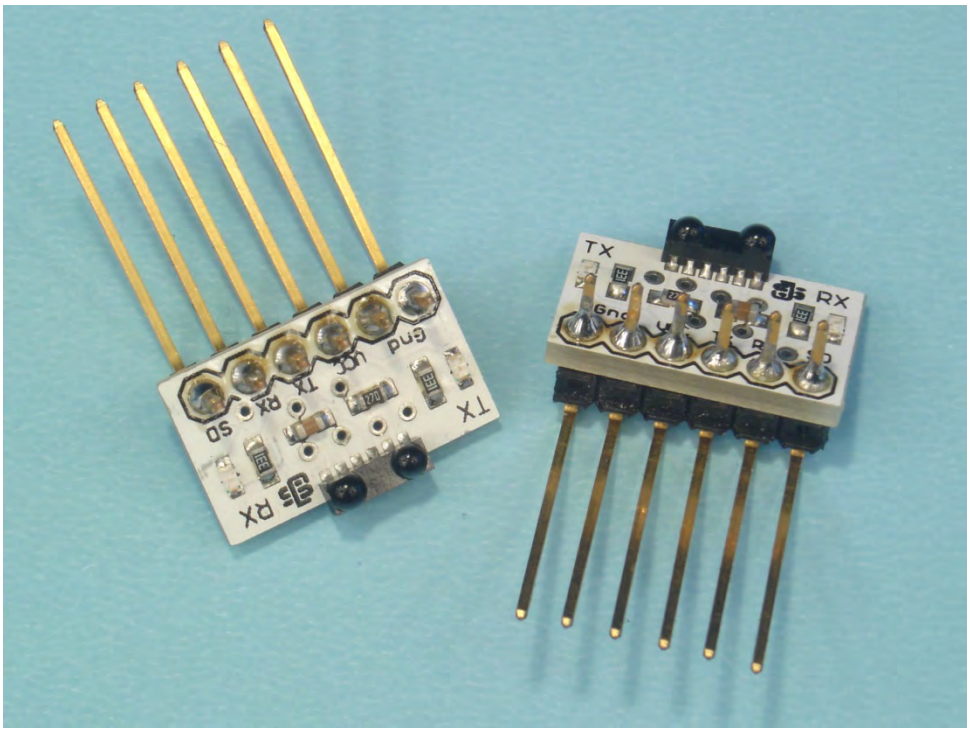


Figure 19.12.
Each of these JayCon boards includes a Vishay TFBS4711 IR transceiver and two LEDs that indicate transmission and reception activity. Right-angle male header pins (not included) soldered to the boards makes them easy to use in a solderless breadboard.

Instead of using software to implement an IrDA protocol, I opted for two Microchip Technology MCP2120 Infrared Encoder/Decoder ICs, one for each "side" of the 2-way communications. These ICs accept a standard UART-type serial stream of bits and convert them to IrDA-compatible serial data that can drive an IR LED. The ICs also accept IrDA-compatible signals from an IR sensor and convert them into a UART-type output.

The MCP2120 IC comes in a 14-pin DIP and requires only a few additional components. **Figure 19.13** illustrates the functional blocks in this IC. See the Reference section at the end of the experiment for a link to the MCP2120 data sheet.

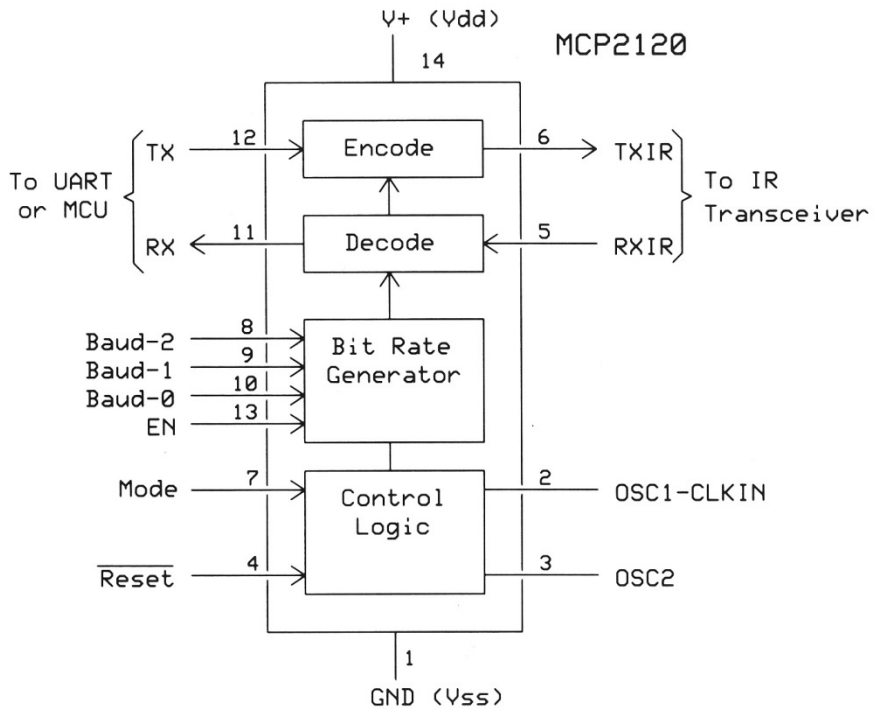


Figure 19.13.
The MCP2120 Infrared Encoder/Decoder takes a UART-type signal at the TX input and encodes it in a standard IrDA format for an IR LED that would connect to the TXIR output. The decoder performs the reverse operation and converts an IrDA-formatted signal from an IR sensor connected to the RXIR input and produces a UART-compatible output at the RX connection.

I had not used an MCP2120 IC before, so I started with a simple circuit and connected the components as shown in **Figure 19.14**. The circuit information came from Microchip's manual for the MCP212X Developer's Daughter Card. Unfortunately, Microchip's documents lack a critical piece of information I discovered during my work with the MCP2120 ICs. I'll explain the problem and a solution later in this experiment.

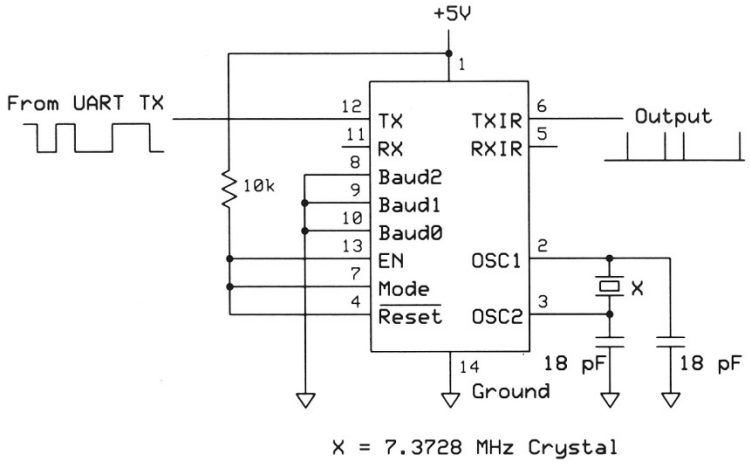


Figure 19.14.
This diagram illustrates the simplest configuration for an MCP2120 that encodes a 9600-bit-per-second UART signal into an IrDA-compatible signal to drive an IR LED. Modifying the BAUD connections at pins 8, 9, and 10 lets you select a transmission rate as high as 115,200 bits per second.

A small 7.3728-MHz crystal governs timing within the IC and lets people choose one of five bit rates for UART and IR communications. Choices range from 9600 to 115,200 bits per second, as set with three pins, BAUD0, BAUD1, and BAUD2. I opted for 9600 bits per second, which required a logic-0 signal, or ground connection at these three pins. Both sides of an IR-communication link must use the same data rate. People can set the bits-per-second rate in software, but I did not attempt to. The MCP2120 data sheet supplies information about how to set the bit rates.

When I powered the circuit shown in **Figure 19.14** and connected a 5-volt 9600 bits-per-second signal to the IC's TX pin (pin 12), a logic-analyzer trace (**Figure 19.15**) showed the UART signal and the IrDA output at the TXIR output (pin 6) on the MCP2120. The IrDA protocol calls for a burst of pulses (shown as a short logic-1 pulse in **Figure 19.15**) in the middle of each logic-0 signal from the external UART.

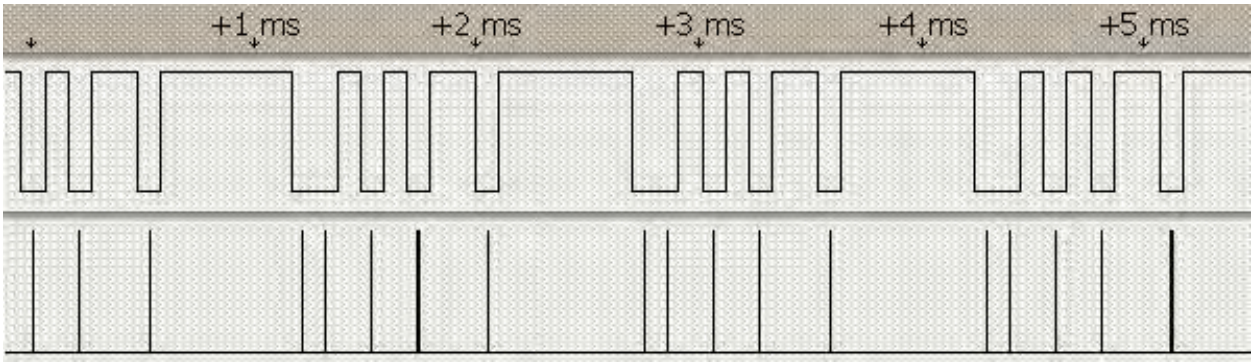


Figure 19.15.
This 2-signal screen image from a Saleae Logic logic analyzer shows a 9600 bits-per-second UART signal (top) connected to the TX input on an MCP2120 IR Encoder/Decoder IC. The resulting TXIR signal output on the IC's pin 6 let me confirm the operation of the circuit shown in **Figure 19.14**.

Next I inserted a JayCon IR transceiver module in the breadboard and connected it to the transmitter TXIR output (see the Notes section at the end of this experiment). Then I placed a second JayCon module in a separate breadboard and oriented it so both modules faced each other. When I connected a logic-analyzer input to the second JayCon module and applied power, the logic-analyzer display showed the inverted version of the transmitted IrDA signal. The JayCon modules have two LEDs, red and green, to indicate IR LED or IR sensor activity. The green LED on the transmitting JayCon module shined very brightly, which I didn't expect. Jiten Chandiramani at JayCon told me the next board layout will ensure this LED turns only as bits get transmitted.

Proof of valid communications required the use of a second MCP2120 connected to the receiving IrDA module, as shown in **Figure 19.16a** and **b**. The image in **Figure 19.17** shows my lab setup of the two MCP2120 ICs and the two JayCon JS-5355 IR transceiver modules. The four logic-analyzer traces in **Figure 19.18** shows the results. As a further test, I could have applied the receiver output to a UART input and compared the byte sent with the byte received.

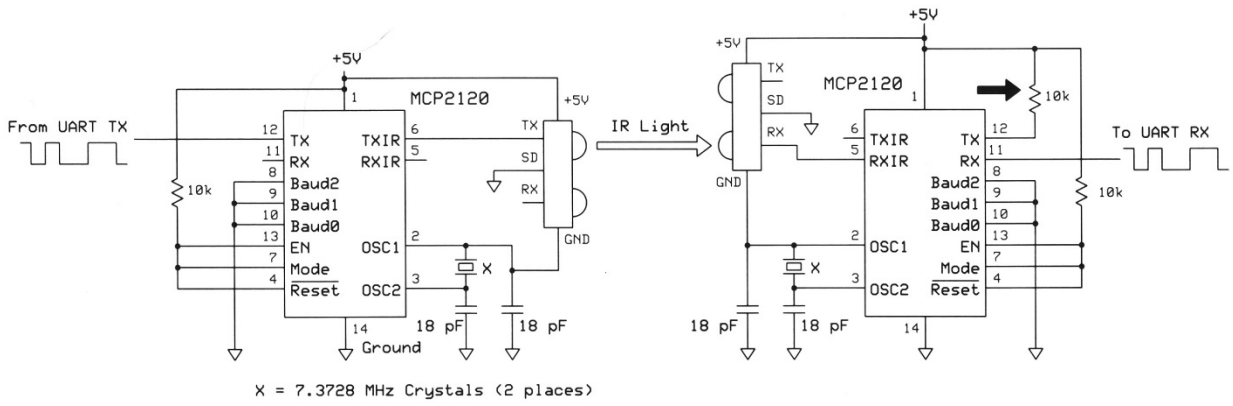


Figure 19.16a.
This diagram shows the complete circuit I used to transfer a UART signal from one breadboard to another via IR light. The two JayCon JS-5355 IR transceiver modules had about 10 cm between them. Refer to the JayCon data sheet for connection details. Note the 10-kohm resistor indicated by the black arrow. You MUST include this to receive a 1-way transmission. Read the text for details.

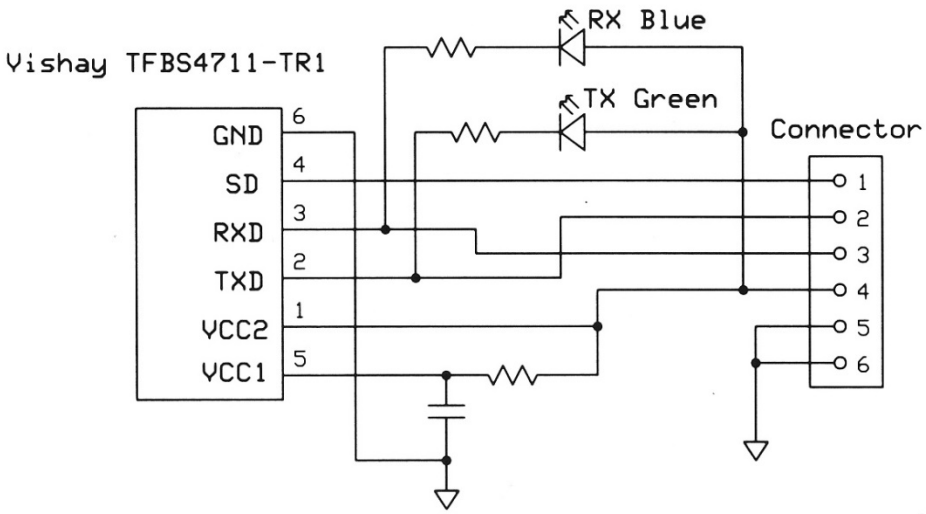


Figure 19.16b.
Schematic diagram for a JayCon JS-5355 IR transceiver module.

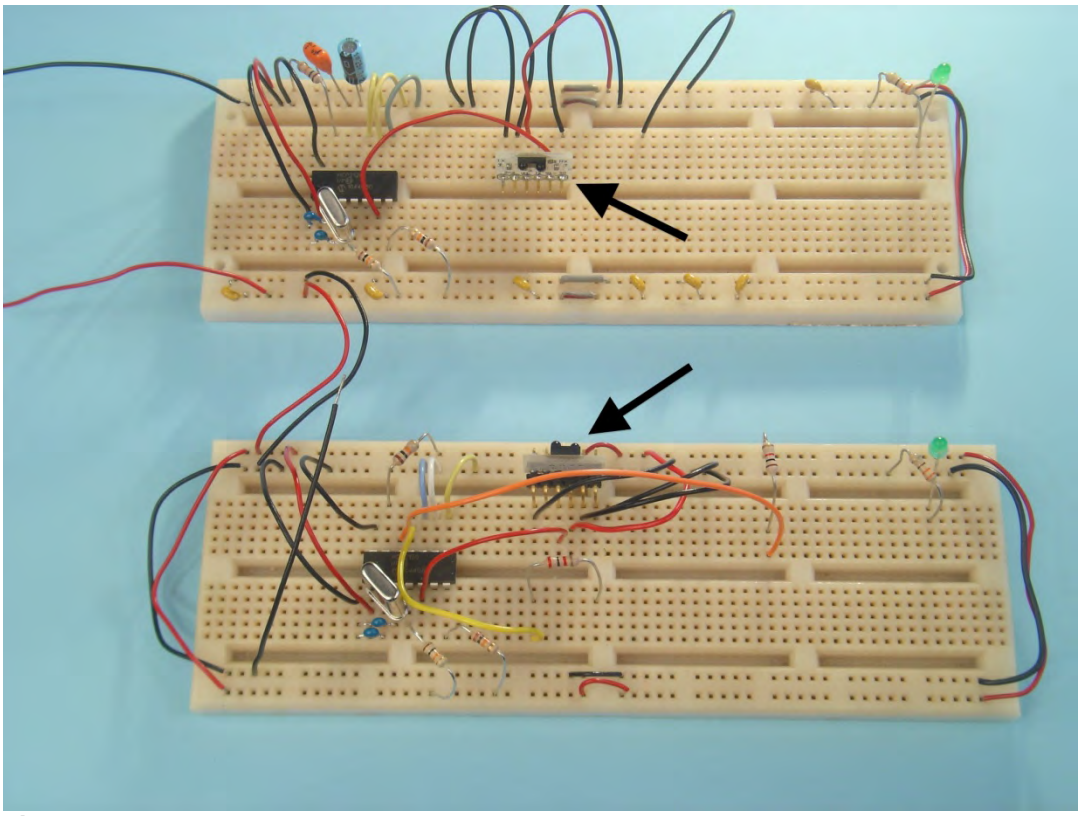


Figure 19.17.
The breadboard setup for IR-communication tests in my lab. The two arrows point to the JayCon JS-5355 IR transceiver modules. Each breadboard has an LED and a resistor to indicate both circuits have power turned on.

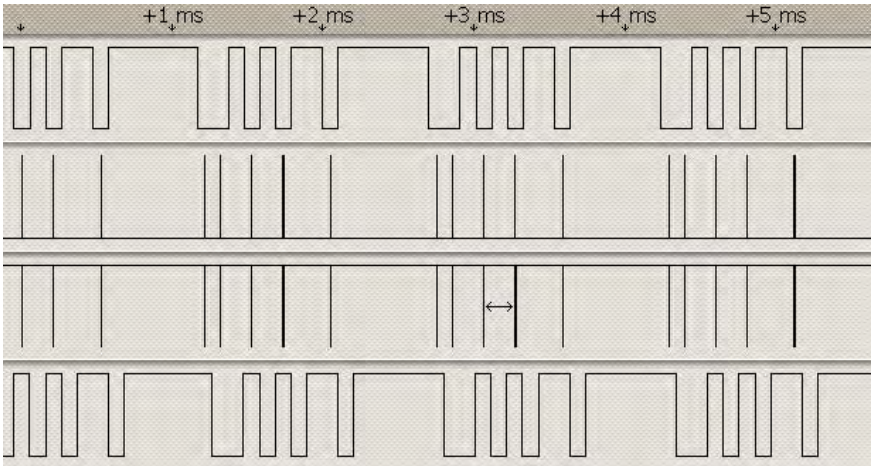


Figure 19.18.
This 4-signal logic-analyzer display shows, from top to bottom, a UART output, the transmitting MCP2120 output to an IR LED at pin 6, the output from a receiving IR sensor RX, and the output at the MCP2120 receiver (pin 11).

As the Propeller continued to transmit a UART signal, an unexpected change occurred. After eight to 20 seconds, the RX signal switched from the UART-type signal to a constant logic-1 output. I investigated the problem and tried to solve it by replacing components, adding capacitors to power lines, and swapping MCP2120 ICs and the IR transceiver modules. Nothing worked. It turns out Microchip expected every circuit would connect the MCP2120 TX input to a UART output, which remains a logic-1 when not sending data. But if you do not "pull up" an unused TX input on an MCP2120 to a logic-1 (+5V) level, the state of the pin

eventually changes internally and shuts down the receiver portion of the IC. Apparently the TX input discharges to ground, which indicates a bit to transmit to the MCP2120. The transmit operation overrides reception, and the receiver section shuts down.

IMPORTANT: In **Figure 19.16a**, you will see the unused TX input on the receiving MCP2120 pulled up to +5V with a 10-kohm resistor. Microchip's data sheet and application information lack this critical information. You *must* ensure the TX input on each side of the link has a logic-1 signal applied to it when not transmitting. A UART transmitter output usually defaults to this state, but check data sheets for MCUs and UARTs to ensure the logic-1 condition exists on any UART-transmitter outputs when not transmitting to an MCP2120. If you do not plan to use one of the MCP2120 ICs to transmit, use the pull-up resistor as shown in **Figure 19.16**.

Of course, you could connect a UART transmitter and receiver at each side of the short communication link as shown in **Figure 19.19**. But you must ensure one side does not try to receive and transmit simultaneously. We consider the MCP2120 transceivers as *half-duplex* devices. That means two-way communications can take place, but only in one direction at a time.

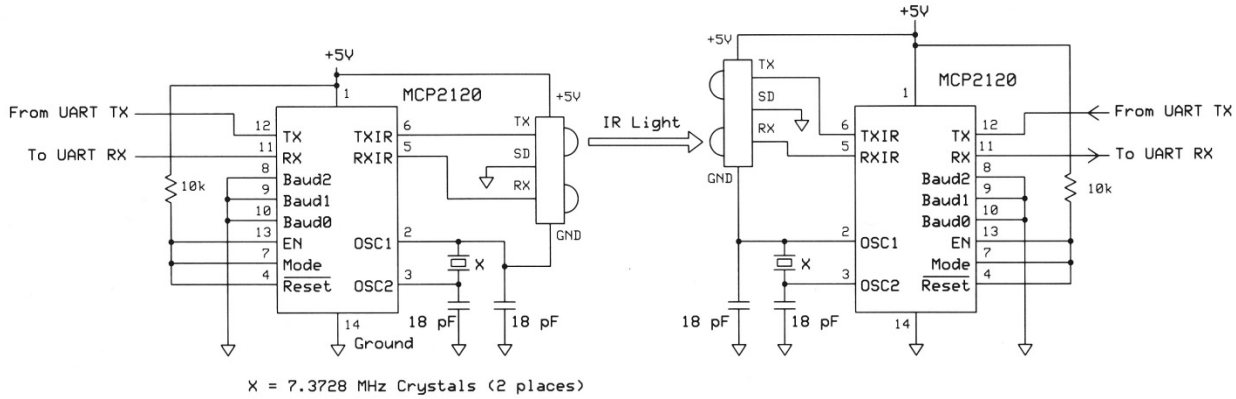


Figure 19.19.
This schematic diagram shows the connections for a complete half-duplex IR-communication link between two devices that each have a UART transmitter and receiver.

An IR-Communication Protocol

Unlike the NEC remote-control encoding used to communicate an address and a command, the IR communications described above let you send one byte or thousands. So far, though, IR communications have included no error checking, no way to ask for a resend, and no way to say "all-OK and over to you" at the end of a transmission. Infrared-communications software should include those features and others. To start, I recommend a simple packet format such as the one shown in **Figure 19.20**. This format includes a fixed *start-byte value* – always 7E hexadecimal – followed by a variable *byte count* that equals the number of bytes in the message portion of a packet. The final byte, a *checksum*, helps a receiving device check incoming messages for errors.

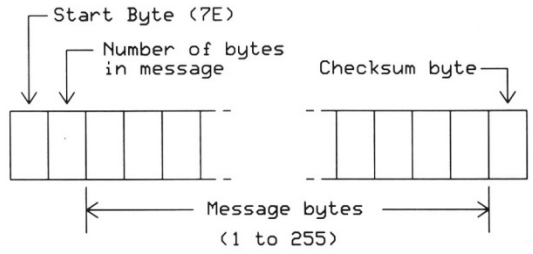


Figure 19.20.
A suggested format for IR communications includes a constant start byte, a message-byte count, message bytes, and a checksum byte for error checking.

You may use the message bytes in any way you choose; perhaps for commands, software updates, addresses, and so on. You might use 1- or 2-byte messages only for commands. A series of communications between a controller and a device such as a robot would need to follow a protocol that describes what happens during communications. Unless you need IR communications with commercial products, you can create a protocol to suit your requirements. **Figure 19.21** illustrates a simple protocol that includes steps that acknowledge receipt of a message or request a resend. You could add other actions as needed.

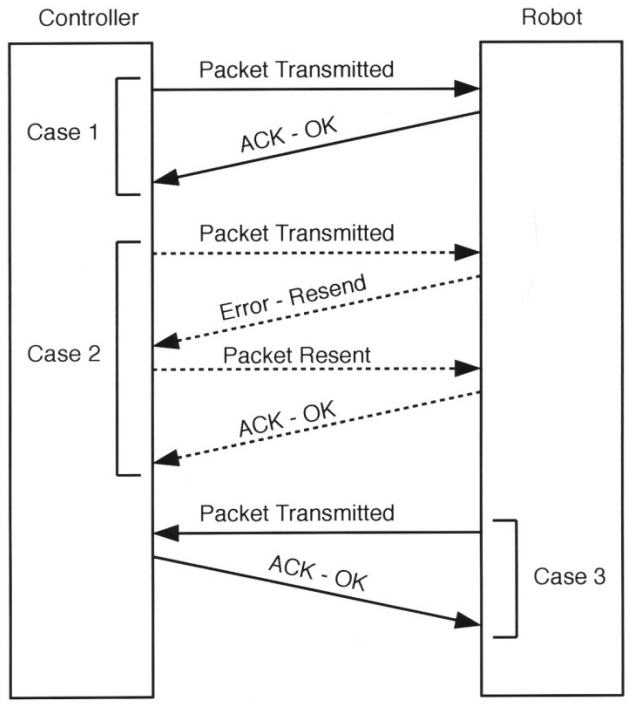


Figure 19.21.
Each IR transmission received by the robot results in one of two types of replies. In Case 1 the robot acknowledged (ACK-OK) proper receipt of a message. In Case 2, the robot received a message but responded with a request for a resend (Error-Resend). The controller complied, the robot received the message properly, and replied with an ACK-OK message. The robot can start a message transmission, too (Case 3). A Error-Resend reply might include information about the type of error detected by the robot's software.

How Does a Checksum Help Us Detect Errors?

The format illustrated in **Figure 19.20** includes a *checksum* byte that lets a receiving device check the message portion of a packet for errors. Without some sort of error checking, a receiving device could perform actions based on faulty information. It might interpret a data byte as a command or *vice versa*. To create a simple checksum, mathematically add all message bytes and discard all but the *least-significant byte* (LSBy) in the result. The following example shows how to calculate a checksum for a 9-byte message, shown as hexadecimal values. (Find a hexadecimal calculator on the Internet here: http://programmerscalc.com/.)

Message: `07 F9 A0 43 00 5B C2 1B 9F`

The sum of these nine bytes comes to: `3BA (11 10111010`$_2$`).` We use only the LSBy, `BA (10111010`$_2$`)`. This value becomes the checksum for the nine-byte message and it gets sent as the last byte in a packet. The complete transmitted packet with the checksum would look like this (message bytes underlined):

`7E 09 07 F9 A0 43 00 5B C2 1B 9F BA`

The 7E (hex) indicates the start of a new message and the 09 (hex) indicates that nine message bytes follow. The checksum ends the packet. Each message will have its own checksum value. When a device receives a packet, it adds the message bytes as they arrive. If the checksum in the packet and checksum calculated at the receiver match, the robot's software would send an OK message back to the controller. If the checksums do not match, however, the robot transmits an Error-Resend message to the controller.

Unfortunately, the use of checksums cannot identify which message byte or bytes contain errors. Other error-detection and error-correcting techniques reduce or eliminate errors, but they take more computer time and add more bytes to a communication. Most discussions of these techniques quickly move into challenging mathematics and statements such as, "...thus the spacecraft were supported by (optimally Viterbi-decoded) convolutional codes that could be concatenated with an outer Golay (24,12,8) code" (Ref. 1). We really need a book such as "Error-Detecting and Correcting for Dummies," but I haven't found one.

IR-Communication Programs

For basic IR communications of UART-type signals via the MCP2120 ICs shown earlier in **Figure 19.14**, I created two Spin programs:

1. `main` that sets up UART communications and handles communication errors, and

2. an object in the file `getIRdata` that receives UART communications, saves message bytes in an array, and reports any communication errors.

This division of tasks lets other programs use an object or objects in the `getIRdata` code. **Figure 19.22** shows the program connections as well as other Spin-language files – available in the Parallax Propeller Object Exchange, OBEX – required for the examples that follow. They're all in the Experiment 19 software folder.

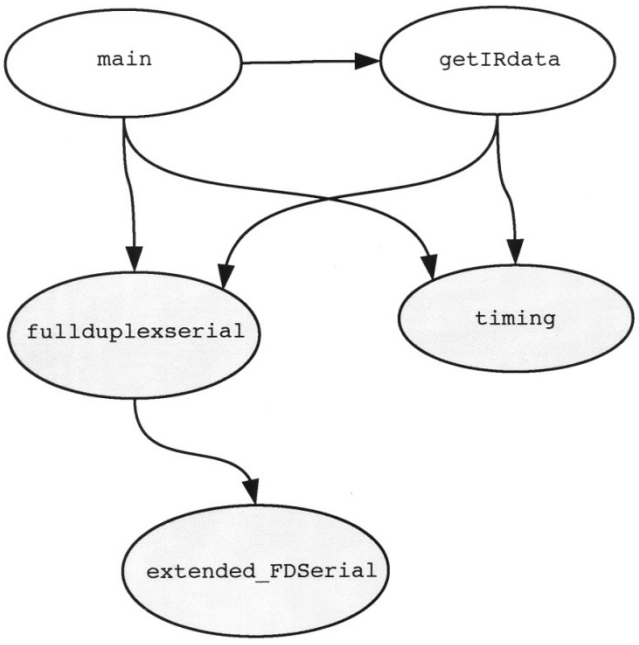


Figure 19.22.
This diagram shows the relationships between the `main` program and the `getIRdata` program that will handle packets received from an IR transmitter. The lightly-shaded programs come from the Parallax OBEX. Programmers refer to this diagram as a *call graph* because it shows which programs, objects, functions, and subroutines depend on each other.

Instead of writing code right away, I created several rough-draft flow charts to test ideas on paper. The flow chart in **Figure 19.23** shows my final algorithm that handles packets received by a Propeller UART. These steps became the foundation for the `getIRdata` program and the `IR_data_input` object within it. Descriptions of the numbered steps follow the flow chart.

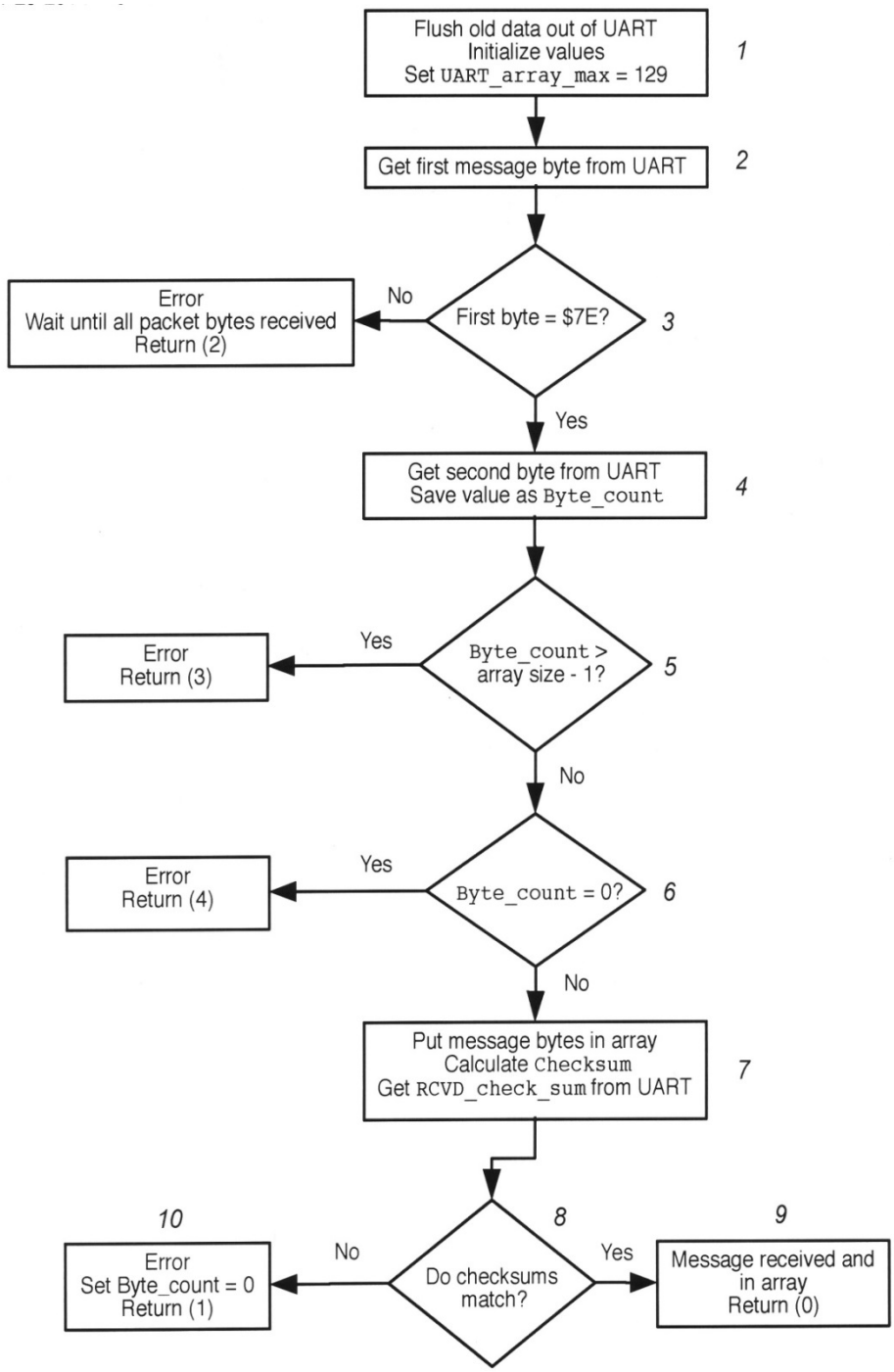


Figure 19.23.
This flow chart shows the steps needed to get messages or detect errors in transmissions received at the Propeller MCU UART. My flow chart shows general operations rather than code statements, which makes it easy to test various conditions on paper rather than trying to immediately write a program.

Flow-chart steps:

1. To ensure a "clean" start, the program "flushes" any old data out of the UART receiver. Then it sets a maximum number of message bytes the object can handle. In this case the value 129 corresponds to 128 message bytes and a byte count, as explained shortly. (You can change this value as needed.)

2. The program next waits until the UART receivers a new byte from the MCP2120 IC.

3. The program compares the received byte with the fixed byte (7E, hex) that starts every packet. If this byte does not equal 7E, the program waits until the UART has received any following bytes and then it returns control back to the main program.

4. When the new byte equals 7E, the program gets the next (second) byte from the UART, which gives us the number of bytes (`Byte_count`) in the message that follows. The `Byte_count` value goes into location `Data_array[0]`.

5. If the `Byte_count` value exceeds the memory space set aside for `Data_array` storage (129 – 1, or 128, bytes), the object returns control to the `main` program.

6. This portion of the object checks for a `Byte_count` value of 0, which should not occur in a valid packet. A valid message must include at least one byte. If the byte-count equals zero, control returns to the `main` program.

7. If the object has received the proper start byte (7E), followed by a valid byte-count value from 1 to 128, the code takes the remaining bytes from the UART and places them in the `Data_array` array. As the *message bytes* arrive at the UART, the code calculates a running `Check_sum` value. After the code has saved all message bytes, it gets the last byte in the packet – the transmitted checksum – `RCVD_check_sum`.

8. Here a comparison occurs between the checksum the program calculated and the `RCVD_check_sum` value from the packet.

9. If the checksums have the same value, the object returns control to the main program. All the message bytes now reside in the `Data_array` array. The code in the `main` program can always get the latest message-byte count from the `Data_array[0]` location so it knows how many message bytes to process.

10. If the program detects a mismatch between the two checksum values, it sets the `Byte_count` value to zero and returns control to the `main` program.

The flow-chart include five `return(x)` statements, each of which includes a number, `x`. A return statement forces an object or section of code to immediately return control to the calling program; in this case, `main`. The fixed values from the `IR_data_input` object indicate how that object "behaved." When the `IR_data_input` object completes its tasks and returns control to the `main` program we have either an array of message bytes or one of several error types.

getIRdata.spin

```
{{
''*********************************************
''*  Program: getIRdata.spin
''*  Author: Jon Titus 01-27-2014 Rev. 3
```

```
''*  Receive bytes and handle as an IR packet in the format:
''*  start byte ($7E), message-byte count, message bytes,
''*  checksum for only message bytes. The IR_data_input object
''*  receives an array pointer from the calling program or
''*  object.
''*
''*  Return to calling program with an error code:
''*  0 = All OK, data in Data_array, mesg-byte count in
''*    Data_array[0] and message bytes start at Data_array[1].
''*  1 = Checksum error, checksum received does not equal
''*    checksum calculated from rcvd message bytes. No data
''*    saved.
''*  2 = First packet byte not $7E, no data saved.
''*  3 = Message byte count exceeds Data_array size limit.
''*    No data saved.
''*  4 = Message byte count = 0. No data saved.
''*
''*  Uses "UART1" designation for Propeller UART
''*********************************************
}}

VAR
  byte Byte_count             'Storage for msg byte count
  byte Array_index            'Index variable for Data_array
  byte RCVD_check_sum         'Checksum in rcvd packet
  byte Check_sum              'Calculated checksum
  byte Trash                  'Dummy variable for unused data
  byte UART_array_max         'Data_array maximum size

OBJ
  UART1    : "fullduplexserial"   'Object for PST comms
  IRdelays : "timing"             'Timing object

'Get IR packet or report an error code
PUB IR_data_input (IRdataptr)
    UART1.start(8, 9, %0000, 9600)  'Start IR-input UART
    UART1.Rxflush                   'Clean out old UART data
    Check_sum := 0                  'Initialize check sum to 0
    Byte_count := 0                 'Initialize byte count to 0
    UART_array_max := 128           'Set maximum message size
    if UART1.rx == $7E              'Look for $7E as start byte
       Byte_count := UART1.rx       'OK, so get byte count
       if Byte_count > UART_array_max  'If byte count >
           return(3)                   'array max? code 3
       elseif Byte_count == 0          'Does Byte_count equal 0?
           return(4)                   'abort with error code 4.

           byte[IRdataptr][0] := Byte_count   'Byte_count OK,
                                              'continue here
              'Get message bytes, put in array
              repeat Array_index from 1 to Byte_count
                 byte[IRdataptr][Array_index] := UART1.rx
                 Check_sum := Check_sum + byte[IRdataptr][Array_index]

              RCVD_check_sum:= UART1.rx       'Get rcvd chk sum
              if RCVD_check_sum <> Check_sum  'Sums equal?
                byte[IRdataptr][0] := 0       'NO: Set byte
```

```
                                        'count to 0
              return(1)                 'Return with error
                                        'code 1.
            else                        'YES, so

              'put byte count in first array location
              byte[IRdataptr][0] := Byte_count
              'return with code (0) no error.
              return (0)

    else                        'First message byte not $7E
      repeat                    'Go through this loop to clean
                                'more incoming bytes from UART
        Trash := UART1.rx
        IRdelays.pause1ms(2)    'wait 2 msec (arbitrary delay)
      until UART1.RxCheck == -1 'repeat loop until no bytes in
                                'UART receiver
      return (2)                'return with error code 2

 ' – -end of Program getIRdata –  – -
```

Program 19.3 gives you a "shell" you can fill with your own code to process a received message or command. This program executes the public object `IR_data_input` in the **getIRdata.spin** file that receives a message and returns a code, 0 through 4. The `main` program evaluates the returned value and takes a specific action. When the code equals 0, the software has valid message bytes in the `Data_array` memory and the message-byte counter in `Data_array[0]`. The message bytes and message-byte counter appear in the PST window as hex values.

Codes 1 through 4 indicate error conditions, so I put statements in **Program 19.3** to print the error type in the PST window. But a final program must respond in some other way. Here's an example: A checksum error might cause the MCU to request a resend from the transmitting device, and it would limit the number of resend requests to, say five before it reboots the MCU. If you don't limit the number of such requests, the MCU could stay in a loop requesting resends forever. You must look at software from a top-down level that covers all eventualities. (What should happen after the MCU receives no response from five resend requests?)

You may run **Program 19.3**, but you must set up another UART so the Propeller can *transmit* a packet via the IR link. For testing I used an Arduino to transmit a packet to the Propeller. The Experiment 19 folder includes my Arduino C-language program in the Jon_Test.ino file. A separate MCU let me test the IR communications between two JayCon transceiver modules with separate power supplies – no common ground or power connections. The circuits and software worked fine.

Program 19.3.

```
{{
''*******************************************
''*  Program 19.3.spin
''*  Author: Jon Titus 01-27-2014. Rev. 2
''*  An MCU or other device transmits a packet via a
''*  Microchip MCP2120 IrDA transceiver chip and a Jaycon
''*  IR module (see text for a schematic diagram).
''*  A separate, equivalent circuit receives the IR transmission
''*  that goes to "UART1" on a Propeller MCU module as defined
''*  below. PST window displays the results.
''*  This program is not in a loop. It runs once. Restart
''*  to run again.
''*
```

```
''*  This program determines what action to take based on the
''*  value returned from the getIRdata program. If no error
''*  occurs, the packet message goes into the array
''*  "Data_array".
''*  Data_array[0] indicated the number of bytes in the message.
''*  message goes into array starting at Data_array[1].
"""*
''*  Error codes 1 through 3 cause message to print. You can
''*  add code to these sections to take other actions based on
''*  an error. You could send a "resend" command to the
''*  transmitting device to have it try again, and so on.
''*
''*  Uses P8X32A Propeller-I board.
''*  Set PST to 9600 bits/sec.
''*****************************************
}}

'Set up clock frequency
CON _clkmode = xtal1 + pll16x     'Set MCU clock operation
    _xinfreq = 5_000_000          'Set for 5 MHz crystal

VAR
    byte Bytes_rcvd               'Byte count for message
    byte Array_index              'Index for Data_array storage
    byte Data_array[256]          'Set max size of Data_array
    byte RCVD_check_sum           'Storage for packet checksum
    byte Check_sum                'Checksum calcd in getIRdata
    byte IR_result                'Returned value from getIRdata

OBJ
    term   : "fullduplexserial"   'Object for PST
    delays : "timing"             'Timing object
    IR     : "getIRdata"          'Object for serial IR input

pub main
  Check_sum := 0                  'Initialize checksum to zero
  term.start(31, 30, %0000, 9600) 'Start PST

  'Call IR_data_input object in getIRdata
  IR_result := IR.IR_data_input(@Data_array)

  'Test the error value returned from the IR_data_input object
  case IR_result
       0: 'Code = 0, all OK
          Bytes_rcvd := Data_array[0] 'Get number of bytes in
                                      'msg from Data_array[0]
          'Loop to print message values in hex
          repeat Array_index from 1 to Bytes_rcvd
                term.hex(Data_array[Array_index], 2)
                term.tx(13)
          term.str(String("Bytes received: "))
          term.dec(Bytes_rcvd)
          term.tx(13)                 'New line on PST


    1: 'Error code = 1
          term.str(String("Checksum error!"))
```

```
          term.tx(13)

    2: 'Error code = 2
          term.str(String("Bad start byte, no bytes received!"))
          term.tx(13)

    3: 'Error code = 3
          term.str(String("Mesg exceeds array size, no bytes received!"))
          term.tx(13)

    4: 'Error code = 4
          term.str(String("Byte count equalled zero. Aborted input!"))
          term.tx(13)

  OTHER: 'IR_data_input returned invalid error code
          term.str(String("Error not defined."))

'Your code goes here to process the message information,
'command, etc...

' – -end of Program 19.3 –  – -
```

More About How Programs Handle Arrays

As explained earlier, the `getIRdata` program gets messages from a UART and saves them. A main program can then process the message bytes and determine how to use them. Thus the `getIRdata` program and the `main` program must access information in the same array. Most programming languages don't let us pass or return a complete array between objects. Arrays usually contain too much data to make it practical to create copies. Instead, we use a pointer to identify the address of the first array element in memory. You got a quick introduction to pointers in Experiment 13, and I'll expand that information here. Please refer to **Figure 19.24** as you read the following text.

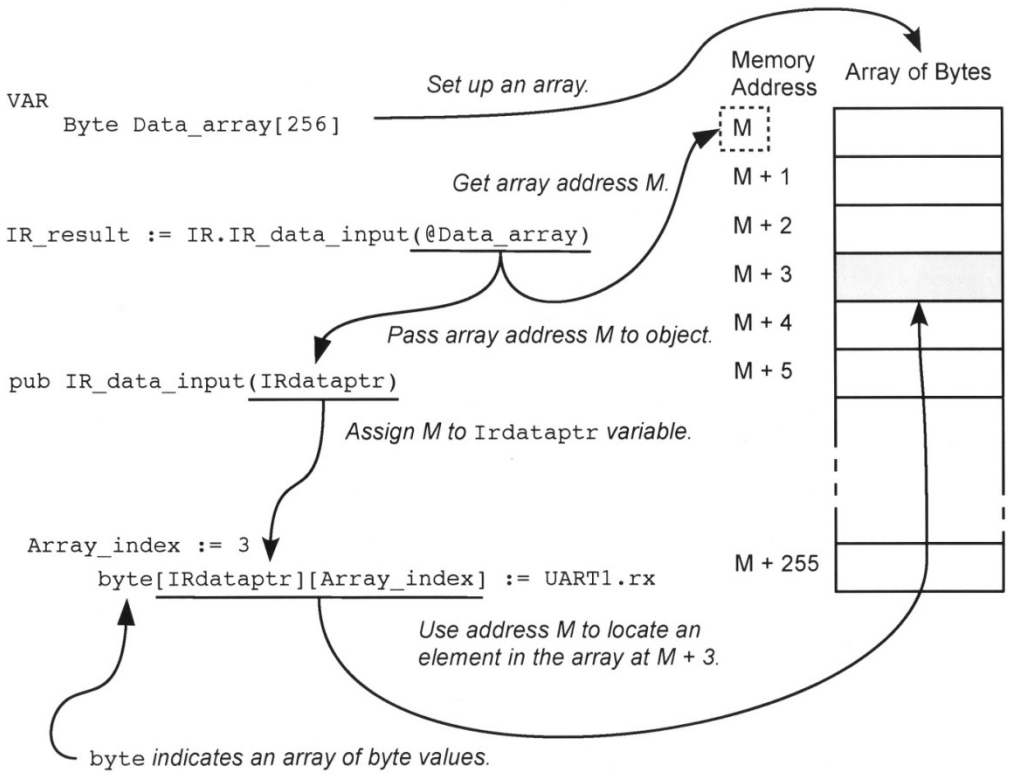


Figure 19.24.
This diagram shows several program steps and how they pass and use an array pointer.

Starting at the top, the **getIRdata.spin** program sets up an array named `Data_array` with 256 bytes in sequential memory locations. Both the `IR_data_input` object and the `main` program will use this array to hold incoming messages. (The Propeller automatically assigns an array a starting memory address.)

When the Propeller executes the statement:

```
IR_result := IR.IR_data_input(@Data_array)
```

it refers to the `IR_data_input` object referenced earlier in the OBJ section of the `main` program. The ampersand character (@) in front of the array name indicates the code will send the `IR_data_input` object the *address* of, or the *pointer* to, the first element in the array. In effect, this statement says, "I cannot give you the complete array, but I can tell you where it starts." I'll use M to represent the starting memory address of the `Data_array` storage area, as shown in **Figure 19.24**.

The following statement in the `getIRdata` file:

```
pub IR_data_input  (IRdataptr)
```

accepts the address value (M) from the calling program – in this case, `main` – and it assigns the value of M to the variable named `IRdataptr` (IR data pointer). This address points to the first element in the array:

`Data_array[0]`.

The statement:

```
byte[IRdataptr][Array_index] := UART1.rx
```

identifies an array of bytes that starts at memory address `IRdataptr`. The sum of the `Array_index` value and the `IRdataptr` address identifies a specific byte in the `Data_array` section of memory. The statement above occurs in a loop that increments the `Array_index` value by 1, so bytes received from UART1 get saved in sequential array locations. In **Figure 19.24**, the `Array_index` equals 3, so the UART1 byte just received goes into `Data_array[3]`. This statement uses the `byte` prefix so the Propeller can properly identify memory locations.

Where do You Go from Here?

You may use and modify **Program 19.3** and the `getIRdata` code as you wish, but as explained earlier the `main` program would need some enhancements before use in a robust project.

1. You would need to add code to take actions based on each error code.

2. You would need to think more about other problems the `IR_data_input` object could encounter. Suppose a packet arrives with the proper start byte (7E) and a byte count of 5F (95_{10}), but the transmission only includes 45_{10} bytes. How should the program handle this situation? As now written, the object will simply sit and wait and wait and wait for additional bytes to arrive, thus "locking up" the Propeller MCU in an endless loop.

To avoid this type of lock-up, you could implement a watchdog timer that would reset your program if a UART event doesn't occur within a specific time. Or you could abort the `getIRdata` object and force a return to the `main` program. For more information about watchdog timers, refer to the References section at the end of this experiment. You can find a good discussion about watchdog timers for the Propeller IC in the Parallax forum at: http://forums.parallax.com/showthread.php/127346-Simple-watchdog-I-hope.

Conclusion

Infrared LEDs and sensors give engineers and experimenters a way to communicate commands and data using off-the-shelf components. Most such communications require software to capture the transmitted information, check it for errors, and take appropriate action.You'll find many ICs and hardware modules as well as open-source software that give you a head start.

References

1. "Error Detection and Correction," Wikipedia, http://en.wikipedia.org/wiki/Error_detection_and_correction.

Berlin, Howard M., "The 555 Timer Applications Sourcebook, with Experiments," Howard W. Sams, Indianapolis, IN. 1976. ISBN: 0-672-21538-1.

Yadav, Amit, *et al*., "PC to PC Infrared Communication," http://www.oocities.org/gauravmendiratta/MinorProjectReport.pdf.

Data sheet for the TSOP382-family IR receiver ICs, Vishay Semiconductors: https://www.sparkfun.com/datasheets/Sensors/Infrared/tsop382.pdf.

Microchip data sheet for the "MCP2120 Infrared Encoder/Decoder," http://ww1.microchip.com/downloads/en/DeviceDoc/21618b.pdf.

For the Microchip "MCP212X Developer's Daughter Board User's Guide," http://ww1.microchip.com/downloads/en/DeviceDoc/51571b.pdf. (See page 23.)

For the Microchip application note AN756, "Using the MCP2120 for Infrared Communications," http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en011931.

For information about the Jaycon Systems IR Transceiver module, JS-5355, http://www.jayconsystems.com/tfbs4711-ir-transceiver.html.

For information about the Vishay TFBS4711 Serial Infrared Transceiver, http://www.vishay.com/docs/82633/tfbs4711.pdf.

"Passing array as an argument," http://forums.parallax.com/showthread.php/133718-Passing-array-as-an-argument?highlight=passing+array.

"Simple Watchdog, I Hope." http://forums.parallax.com/showthread.php/127346-Simple-watchdog-timers-I-hope.

"Watchdog Timer." http://en.wikipedia.org/wiki/Watchdog_timer.

Notes

The JayCon Systems JS-5355 modules have six plated-through holes (0.1-inch centers) for electrical connections along one side of the printed-circuit board. I used some single-row 6-pin male headers with the pins prebent at a 90-degree angle. The short ends got soldered to the module and the long ends got pushed into breadboard receptacles. This type of header lets the PCB stand upright in a breadboard. A Google image search for 'wire wrap male header "right angle"' will show examples and point you to suppliers. "Break away" headers make it easy to snap off in one piece as many contacts as you need.

Experiment No. 20 – How to Use an Infrared Distance Sensor

Abstract

Often devices must measure a distance to position themselves properly, to avoid an object, or to simply report a dimension. In this experiment you will learn how to use a commercial infrared distance sensor, how to interpret the results, and how a different type of plot can make data easier to interpret and use. You will learn how to use a lookup table or math to get the distance to an object. And you will see how a straight-line fit makes results easier to use and how scaling values simplifies math operations.

This experiment includes many details, so do not rush through it, although in some places you may skip ahead and return to the details later. You should have some experience with algebra and the equation for a straight line ($y = mx + b$). Some parts of this experiment rely on supplied Excel spreadsheets.

Keywords

distance sensor, least-squares fit, infrared sensor, reflections, lookup table, scaling, math

Requirements

(1) - Solderless breadboard
(1) - Voltmeter or volt-ohm-milliammeter (VOM)
(1) - DC power supply, +5 volts
(1) - Propeller P8X32A microcontroller board
(1) - USB cable for Propeller board
(1) - GP2Y0A21YK0F distance sensor, Sharp Electronics
(1) - MCP3202 12-bit analog-to-digital converter, Microchip Technology
(1) - Ruler, 1-meter, or longer, with centimeter divisions
(1) - Small wood block (optional)
(2) - #4 wood screws, round head, 3/8-in. long (optional)

Introduction

Now you will learn how to use a prepackaged IR sensor to measure distances. You might think a distance measurement would involve aiming a burst of IR light at a "target" object and measuring the time it takes for the reflected light to return to the sensor. That type of "time-of-flight" measurement will work, but let's look at the physics from a practical perspective.

Figure 20.1 shows a block diagram for a time-of-flight distance-measurement system. This system aims a laser diode at an object and transmits a short pulse of light. Internally, a small amount of the laser beam reflects into a start-pulse sensor that starts a timer. Most of the laser power reaches the object and reflects from it. Some of the reflected light reaches the stop-pulse sensor that turns off the timer. Signal-processing hardware and software convert the start-to-stop time – the laser pulse time of flight – into a distance value.

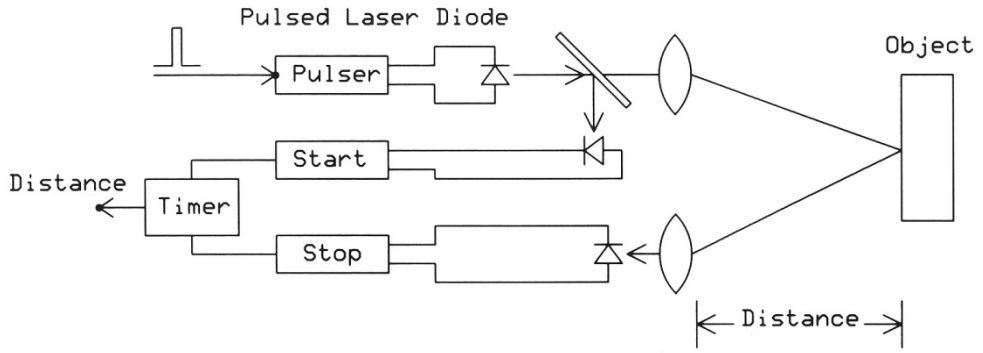


Figure 20.1.
This block diagram shows the configuration of a time-of-flight distance-measurement system. The time it takes for reflected light to reach the stop sensor determines the distance to an object.

Could we create a small time-of-flight circuit on a breadboard to measure distances? Light travels at roughly 300,000,000 meters/sec, or 3.00 x 10^8 m/sec. Given that speed, how long would it take IR light to reach an object two meters (200 cm) away and return to a sensor?

First, invert the speed of light, 3.00 x 10^8 m/sec, to get the time needed for light to travel one meter: 3.33 x 10^{-9} sec/m. Now multiply this value by twice the distance to the object to get the time of flight for the transmitted pulse and the return of reflections:

$$3.33\,x\,10^{-9}\,sec/m\ *\ 2\,m\ *\ 2\,paths\ =\ 13.3x10^{-9}\,sec, or\ 13.3\,nanoseconds\ (nsec)$$

Electronic devices we can use in a breadboard could not react that quickly. A 74HCT00 NAND-gate IC has a propagation delay of about 18 nsec. That means it would take a pulse edge 18 nsec just to go from a NAND-gate input to an output. It seems impractical to try a time-of-flight approach for distance measurements with a small budget.

All-in-one IR-LED and sensor modules can provide good results within specific distance ranges. The Sharp Electronics GP2Y0A21YK0F sensor, for example, measures distances in the range 10 to 80 cm (about 4 to 31 inches). The Sharp GP2Y0A02YK0F measures distances within the range 20 to 150 cm (about 8 to 60 inches). These devices measure the amount of reflected light received by what Sharp calls a "position-sensitive detector." **Figure 20.2** shows a GP2Y0A21YK0F and a GP2Y0A02YK0F module. Both sensors have three electrical connections for power, ground, and an analog-voltage output that represents a distance.

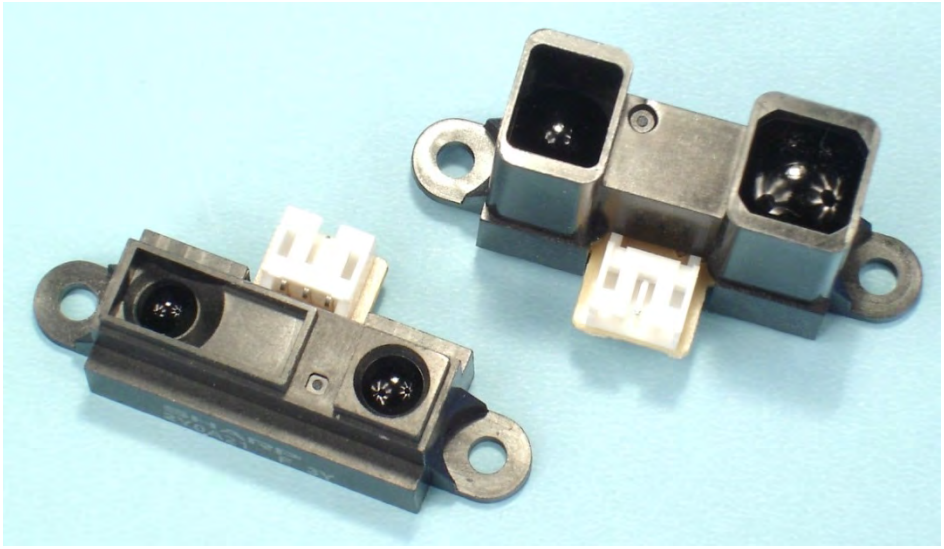


Figure 20.2.
Sharp Electronics GP2Y0A21YK0F (lower left) and GP2Y0A02YK0F (upper right) distance sensors.

I disassembled a GP2Y0A21YK0F sensor (**Figure 20.3**) and could easily see the IR LED and the sensor "chip." The lens for each IC passes IR light, but appears opaque to visible light.

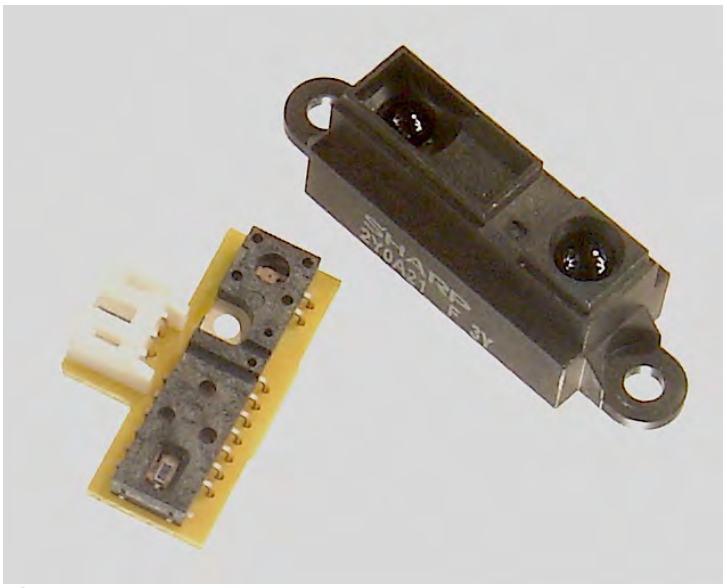


Figure 20.3.
A disassembled GP2Y0A21YK0F distance sensor.

Small inexpensive distance sensors such as these use a triangulation technique to produce a voltage that represents a distance. **Figure 20.4** shows the triangulation of a light beam that reflects from a distant object and from a closer object. The reflected light goes through a lens and strikes the detector at a different point for each distance. The sensor's internal electronics use the position of the light on the detector to create a voltage proportional to the object's position. This type of distance-measurement sensor works well in many environments.

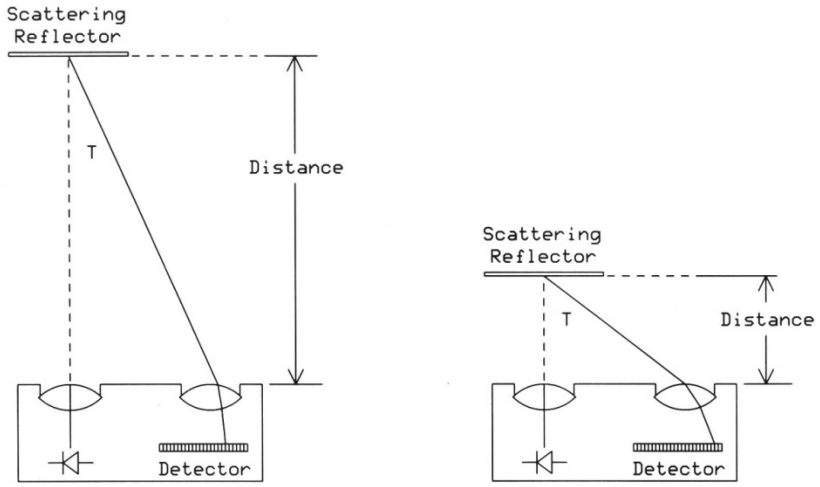


Figure 20.4.
Light from an IR LED reflects from an object at different distances. The angle of reflection (T) determines where the light focuses on a detector. The detector could comprise a row of semiconductor sensors on a chip, or other light-sensitive materials. A reflector, such as a wall, cloth, or cardboard, scatters the IR light in many directions, but most of the IR light still reflects from the center of the light beam (dashed lines).

Highly reflective surfaces, though, can cause problems. **Figure 20.5** shows how a mirror-like reflector parallel to a sensor can affect measurements. The transmitted IR light beam diverges at a small angle as it leaves the module. For simplicity, the figure exaggerates this angle. The light at the right side of the beam reflects off the mirror-like surface at point A and shines into the detector. The sensor then reports a distance of D2 as if it had "seen" a distant object at B. It should have reported distance D1 for the nearer reflective surface.

If the mirror-like object didn't exist and the IR beam (dashed line) had hit a non-reflective, or scattering, surface at distance D2, the most-intense light at the center of the beam would have reached point B. Then the light would have reflected into the detector along the *same path* as the reflection from the mirror-like surface at point A.

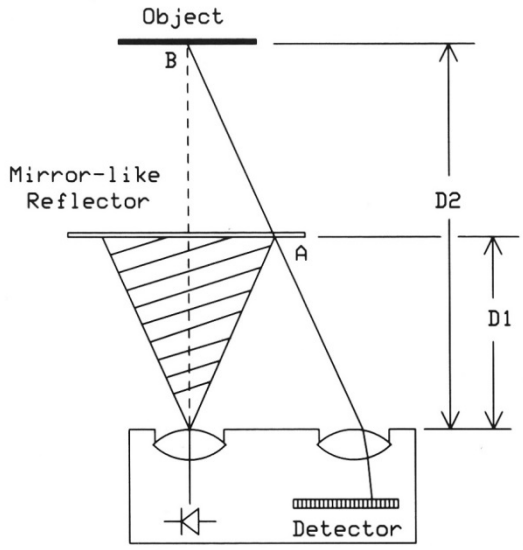


Figure 20.5.
A reflection from a mirror-like surface can cause a sensor to produce an incorrect distance value because a reflection can follow the same path as that from a more-distant object. (IR-light-dispersion angle increased for illustration purposes.)

Figure 20.6 illustrates what can happen with a mirror-like surface placed at an angle to a distance sensor. In this case, IR light from the left side of the beam reaches point C and reflects back to the detector. This reflection follows the same path to the detector as the light would follow had the central beam (dashed line) reflected from an object at point D. Thus, the sensor reports distance D3 rather than distance D4.

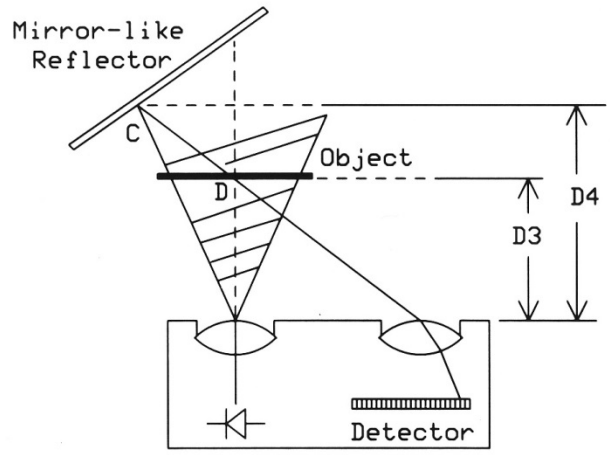


Figure 20.6.
A nearby mirror-like surface can "fool" an IR distance sensor by reflecting the edge of the IR light beam at point C. The sensor interprets this reflection as it it came from point D.

Similar problems can arise when nearby objects cause IR light beam to reflect off their surfaces and the light detector "sees" several distances simultaneously, as diagrammed in **Figure 20.7**.

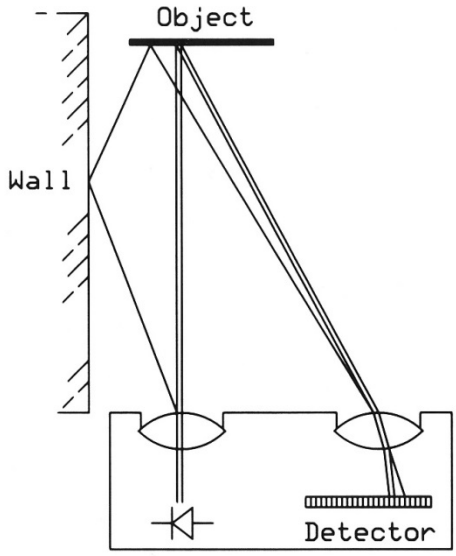


Figure 20.7.
Reflections from nearby objects also can cause a sensor to report an incorrect distance to an object. This figure shows only one reflection from a wall. In real situations, many such light paths might exist.

Although the reflectance of an object's surface might cause a measurement error, and even with the possible problems described above, you can use a distance sensor effectively. For more information about reflections and other types of interference, please read the Sharp Electronics application note listed in the References section at the end of this experiment.

In the steps that follow I used a Sharp Electronics GP2Y0A21YK0F short-distance sensor. Mounting "ears" made it easy to attached this sensor to a small piece of wood to keep it in one place. The sensor has an open back that exposes electrical contacts, so ensure no conductive materials touch them. I recommend you purchase the prepared cable and connector that mates with this sensor. For commercial sources, refer to the bill of materials in the Appendix.

The GP2Y0A21YK0F distance sensors produces a voltage output, so we use an analog-to-digital converter (ADC) to give us digital values to work with. Some MCUs include an ADC, while others require an external ADC. The Propeller MCU falls into the latter category. In Experiment 17, you learned how to connect a 12-bit Microchip Technology MCP3202 ADC to a Propeller MCU, so this experiment will use that ADC, too.

Step 1.

Figure 20.8. Shows front and side views of the GP2Y0A21YK0F sensor and labels the electrical connections. Connect your sensor to a 5-volt power supply and to a voltmeter. Place the sensor about 30 cm (12 inches) above your work surface to minimize reflections. A stack of books or a small rigid box will do.

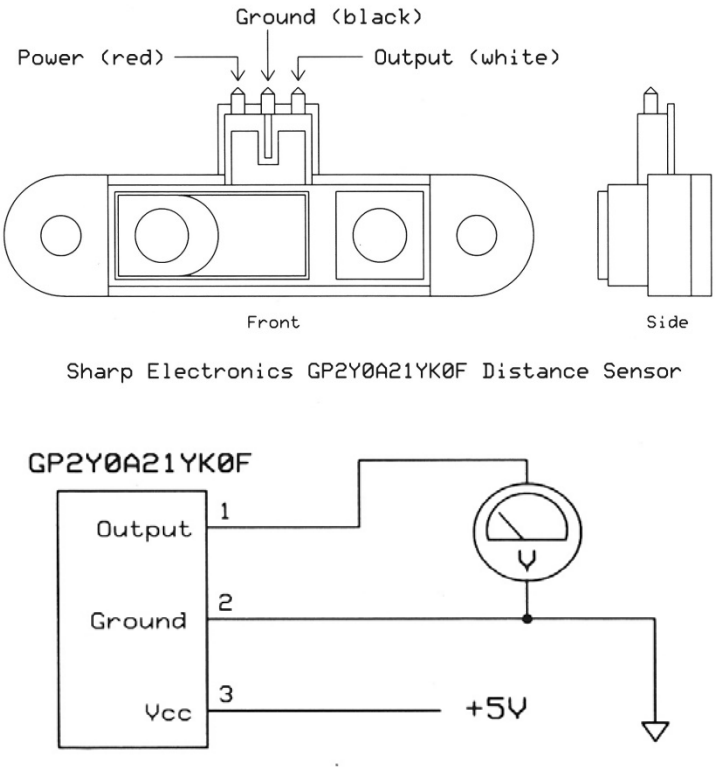


Figure 20.8.
The front and side views (top) of a Sharp Electronics GP2Y0A21YK0F distance sensor labels the electrical connections. The lower diagram shows the electrical connections for voltage measurements. (Pin size enlarged for clarity.)

The **Figure 20.9** diagram shows a side-view of my lab setup. I used a large cardboard box and a piece of rigid cardboard as the "target" for measurements. An extension ruler marked in centimeters and taped to my lab table gave me a stable distance reference. Remember, measure distances from the front of the sensor module.

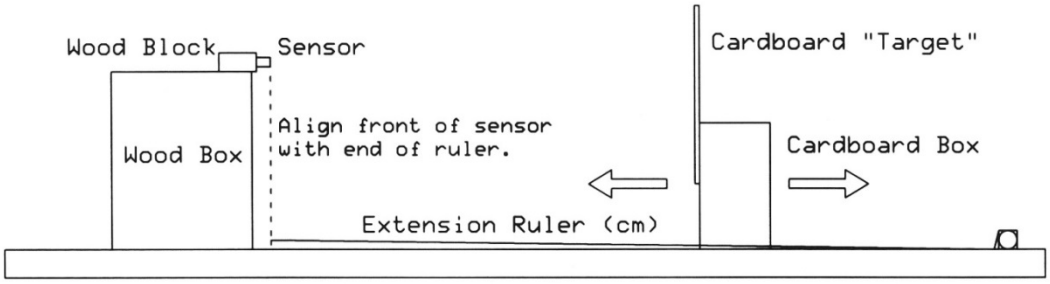


Figure 20.9.
This experimental setup included a ruler attached to my table with tape. A piece of cardboard glued to a box gave me a good target object.

Step 2.

Do not turn on power now. To make measurements, you will move the cardboard, or other target, away from the sensor in 1-centimeter increments at first, and then in 5- and 10-cm increments at longer distances.

Now turn on power to your sensor and place the target 5 cm from the front of the sensor. Take a voltage reading and move the cardboard to the 6-cm position for the next voltage reading. Use **Table 20.1** to record the sensor's output voltage at each position. Continue to make measurements until you have one at 80 cm. Then stop. I suggest you make three runs through the positions and record the voltage at each. That means go through the various distances once in sequence, then go through them again, and so on. Don't simply move to a position and take three voltage readings there before you move to the next position. You want to go through the distances in three runs so you can determine whether the sensor and experimental setup provide reproducible results. In other words, each time you put the target at distance *x*, do you get a similar voltage from the sensor? I found the voltages at each position closely matched each other. If yours don't, you might need a better reflective surface or a weight in the target and sensor boxes to stabilize them.

If you have an analog voltmeter, approximate the voltage reading. Don't worry if you have a measurement with only two significant figures; for example, 3.3 volts. A digital voltmeter might read 3.32 volts, but the extra digit will not mean much in the steps ahead.

Table 20.1. Experimental results for voltage measurements at fixed distances.

	Voltages (V)		
Distance (cm)	Run 1	Run 2	Run 3
5			
6			
7			
8			
9			
10			
15			
20			
25			
30			
40			
50			

60			
70			
80			

Table 20.2 shows my averaged results, based on two runs. The right-most column lists voltages taken from the GP2Y0A21YK0F data sheet. Unfortunately, Sharp Electronics provides a small voltage plot rather than a table of voltages and distances. I used a ruler to "pick off" a voltage from the plot for each distance. But finding values with this technique leads to errors, so I include the values from Sharp only for a rough comparison. I could not determine values on the plot for the 6- through 9-cm distances.

Table 20.2. Results from two runs with a GP2Y0A21YK0F sensor.

Distance (cm)	Avg Voltage (V)	Sharp Voltage (V)
5	3.08	3.13
6	3.07	– – -
7	2.97	– – -
8	2.73	– – -
9	2.48	– – -
10	2.19	2.26
15	1.63	1.68
20	1.28	1.28
25	1.06	1.07
30	0.883	0.934
40	0.679	0.747
50	0.537	0.607
60	0.434	0.514
70	0.336	0.444
80	0.259	0.420

Step 3.

After you complete three experimental runs, average the readings for each distance. Then plot these averages with voltage as the *y* axis and distance as the *x* axis. You can use graph paper or the Excel Sensor_Output.xls file in the Experiment 20 folder. The Excel spreadsheet creates a plot of your averaged data and produces two other plots explained next.

The plot in **Figure 20.10** shows how my voltage-versus-distance values appear. Does your chart look similar? It should come close, depending on your experimental setup. If your chart looks wildly different, I suggest you repeat Step 2 to get new values. Or you can use my values instead.

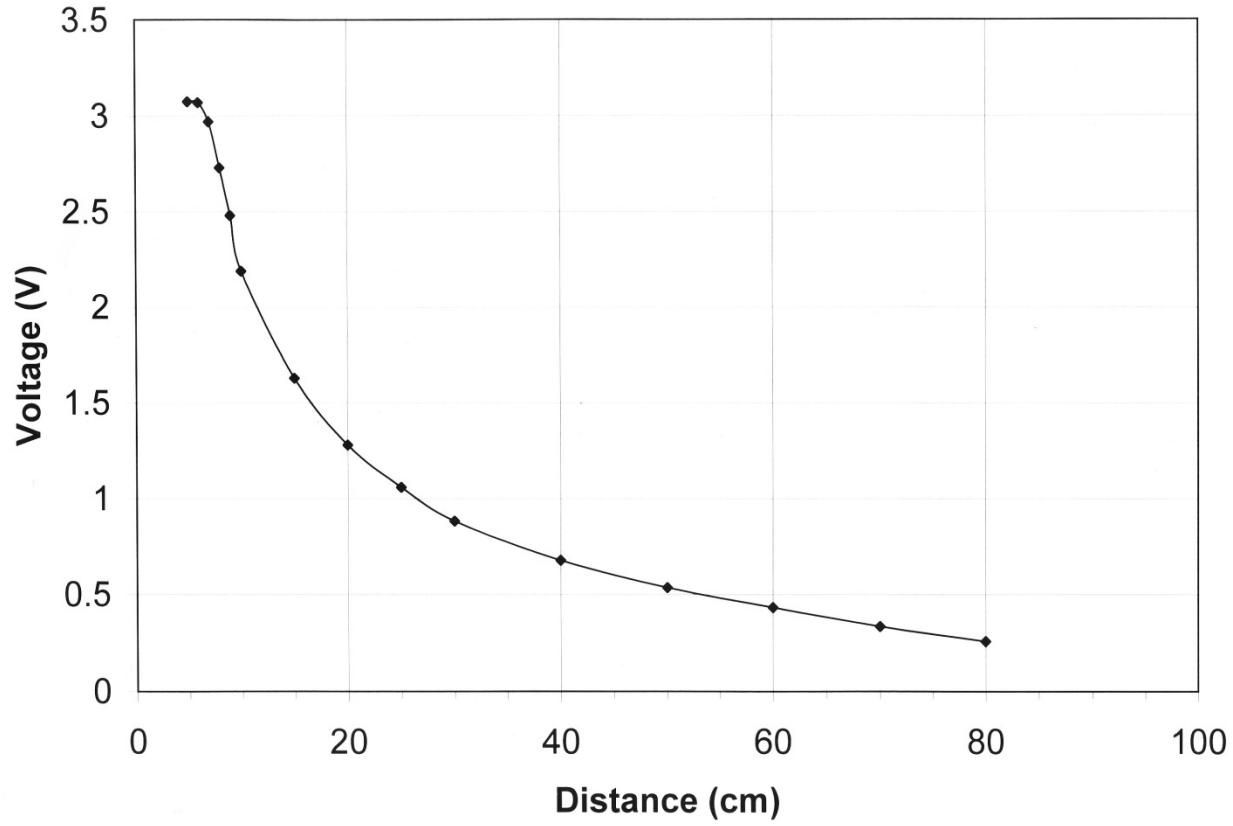


Figure 20.10.
Plot of voltages versus distances for a GP2Y0A21YK0F sensor with a 5-volt power supply. Measurements taken with a Hewlett-Packard 3478A digital multimeter.

The plot in **Figure 20.10** shows a nonlinear (non-straight-line) relationship between distance and voltage, which complicates the conversion of a voltage into a distance. We prefer a linear, or straight-line, plot which would make it easier to find a distance based on a voltage. A distance increment (x) would cause a proportional voltage change (y). Such a relationship means we see a fixed voltage change for each centimeter change in distance. That linear relationship would apply across most or all of the sensor's distance range. (Many sensors produce a nonlinear response, so engineers deal with it in several ways as discussed in earlier experiments.)

Instead of plotting voltage versus distance, plot voltage versus the *reciprocal* of distance, or one divided by distance (1/distance). The plot in **Figure 20.11** – and in the Excel spreadsheet – shows the result of this approach. Now you see a plot, most of which approximates a straight line. To get a distance from the reciprocal-distance value, simply divide it into 1. The reciprocal of 0.0125/cm, for example, equals 80 cm.

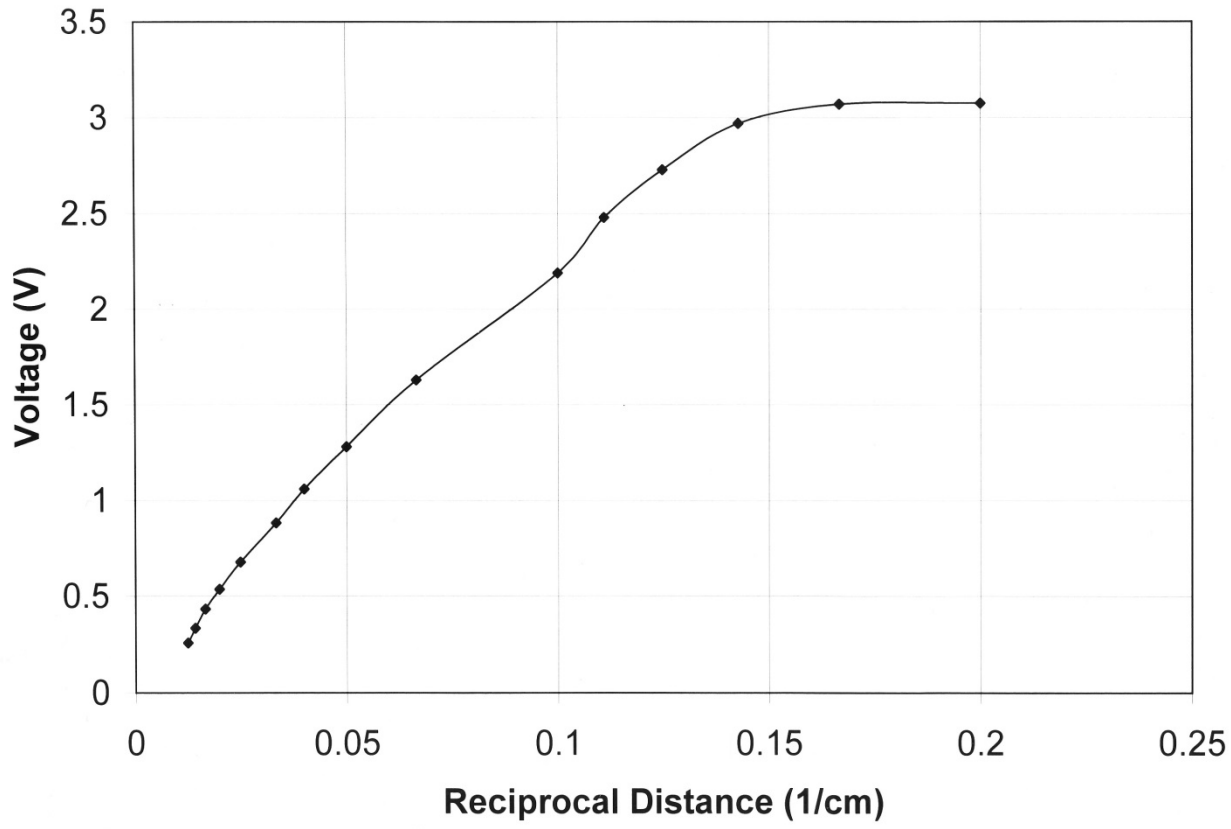


Figure 20.11.
A plot of sensor voltage versus the reciprocal of distance comes close to a straight line for distances in the range 80 to 7 cm. Those two points correspond to *reciprocal-distance values* 0.0125/cm and 0.142/cm, respectively. The left end of the Reciprocal Distance axis represents far measurements, while the right end represents distances closer to the sensor.

Get a Good Straight-Line Fit

When a plot of data shows a somewhat-straight-line relationship, software can perform a *least-squares fit* to create a straight line. That line will best *approximate* the voltage versus distance relationship, based on the plotted measurements. **Figure 20.12** reproduces the plot from **Figure 20.11** and superimposes a least-squares-fit line on it. The Excel spreadsheet plots this straight line for you. I removed the 5- and 6-cm *outlier* values from the analysis because they have almost the same voltage. If you measures 3.07 volts from the sensor at 5 and at 6 cm, how would you know the correct distance? The range from 7 to 80 cm (0.142/cm, 0.0125/cm) will suffice.

Scientists and engineers don't just "throw out" data that doesn't look good or doesn't "fit" the results they expect. First they would rerun tests or experiments several times to see whether they get the same results again. Second, if they get similar results they would look closely at the experiment setup to determine if the equipment caused problems. Third, they would look at their instruments measurement capabilities. Do they have calibrated instruments? Do they have instruments set for the correct measurements? Is there some other effect that causes the problem. In the case of the IR distance sensor, the 5- and 6-cm measurements appear unusual because the sensor just can't provide a useful voltage at close range. So as long as we don't need measurements close to the sensor, we eliminate those measurements.

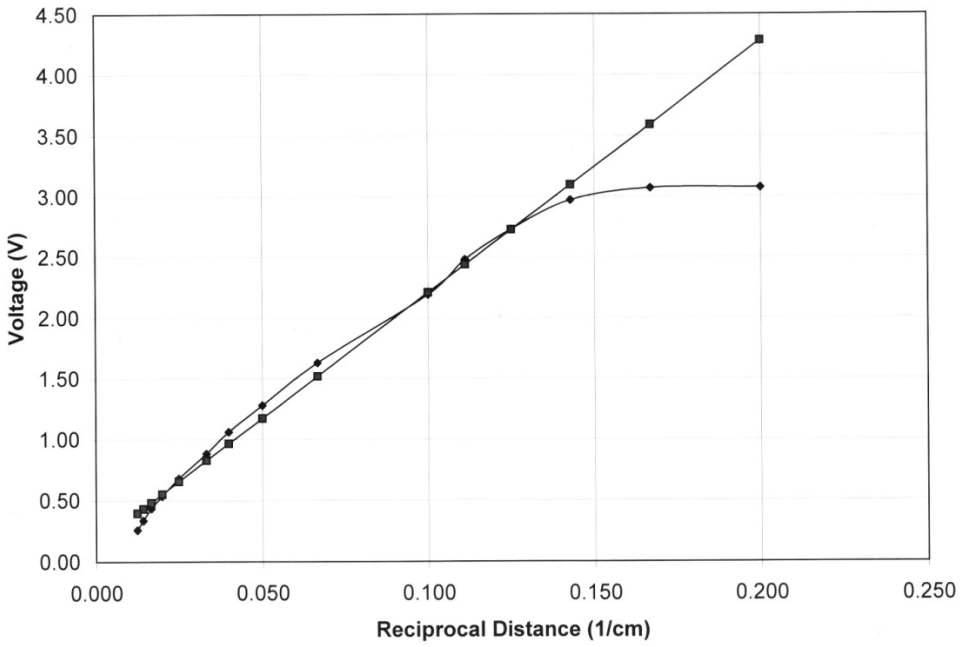


Figure 20.12. This plot shows the results of a best straight-line fit (square points)calculated in an Excel-spreadsheet for the measured voltages (diamond points). The least-squares-fit calculations excluded the measurements at 5 and 6 cm.

You can use the slope and the y-intercept values calculated by Excel in the general equation for a straight line:

$$y = m * x + b$$

In this case, the slope (m) equals 20.7 volt-cm, and the y intercept (b) equals 0.136 volts. The equation for the straight line comes to:

$$y\ volts = (20.7\ volt\text{-}cm * x) + 0.136\ volts$$

Note: The slope is *not* 20 volts-per-centimeter (v/cm) because the x axis in **Figure 20.11** has units of reciprocal centimeters, or 1/cm. Thus 20.7 v-cm

Given a voltage (y) from the sensor, people can use the equation above to calculate the reciprocal of the distance (x). Take the reciprocal of the plotted x value ($1/x$) to convert it back into centimeters. Unfortunately, the data sheet for Sharp's GP2Y0A21YK0F sensor does not include an accuracy or tolerance specification. So I assume an accuracy of about one centimeter. At this point, an engineer or programmer must think about how to use the equation just introduced to convert voltages into distances.

But before we do any calculations, we need an ADC to convert voltages into binary values an MCU can work with. In Experiment 16 you learned how to use a Microchip Technology MCP3202 2-input 12-bit ADC, so this experiment will use it, too. The 3.3-volt output (pin 38) on a Propeller P8X32A MCU board will power the ADC and give the it an input-voltage range from 0 to 3.3 volts. Those voltages correspond to 12-bit outputs from 000000000000_2 to 111111111111_2, respectively.

For a distance accuracy of one centimeter, we need only integer distances such as 7, 20, and 64 centimeters. We won't get a fractional distance such as 22.5 cm. Given the sensor's range from 7 to 80 cm and an accuracy of 1 cm, we can have only 74 possible distances to report. Given that number, do we need a 12-bit ADC with a resolution of 1 part in 4096?

No, we can use a 7-bit ADC that would give us 2^7, or 128, output values, 0000000_2 to 1111111_2. That's a resolution of 1 part in 128, and we need only 1 part in 74. Instead of looking for a 7-bit ADC with a 0-to-3.3-volt input range, the MCP3202 will suffice, although we will not use all 12 bits. In fact we discard the five least-significant bits, D4 through D0, and use only the seven most-significant bits (MSBs), D11 through D5. **Table 20.3** shows the voltage "weight" associated with each of the seven MSBs. An X indicates bits we don't need and won't use. So the 12-bit ADC value (010011000000_2) shown in Table 20.3 amounts to 1216_{10} (1024 + 128 + 64) and represents 0.980 volts, for example. (Assume the don't-care bits have no use.)

Table 20.3. Bit positions and voltages for a 12-bit ADC with a 3.3-volt reference.

Bit Position	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Binary Weight	2048	1024	516	256	128	64	32	16	8	4	2	1
ADC Weight (volts)	1.65	0.825	0.413	0.206	0.103	0.0516	0.0258	X	X	X	X	X
12-Bit Value from ADC	0	1	0	0	1	1	0	X	X	X	X	X

X = Don't care.

To make the ADC values easier for the Propeller to use, software in this experiment performs five bitwise shift-right operations that removes the five LSBs and leaves only the seven MSBs (**Table 20.4**). With the bits in these new positions, the resulting 7-bit values range from 0 to 127, and the shifted value equals 38_{10} (32 + 4 + 2). Note that the *binary weights remain constant* at each bit position, but we think of the ADC voltages as "shifted" with the seven bits that remain. Those bits still represent the voltage above them, just in a different format.

Table 20.4. Result of five bitwise shift-right operations.

Bit Position	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Binary Weight	2048	1024	516	256	128	64	32	16	8	4	2	1
ADC Weight (volts)	1.65	0.825	0.413	0.206	0.103	0.0516	0.0258	X	X	X	X	X
12-Bit Value from ADC	0	1	0	0	1	1	0	X	X	X	X	X
	After Five Bitwise Shift Right Operations											
ADC Weight (volts)						1.65	0.825	0.413	0.206	0.103	0.0516	0.0258
After 5 Right Shifts	0	0	0	0	0	0	1	0	0	1	1	0

X = Don't care.

After the shift operations, the binary value (000000100110_2) shown in **Table 20.4** amounts to 38, *but it still represents 0.980 volts*. So although the *numeric value* of the bits changed, the *voltage* we assigned to each bit remains the same. Keep this handling of the ADC bits in mind in the sections that follow. In effect, the 12-bit ADC has become a 7-bit ADC with the same voltage span, 0 to 3.3 volts. We simply moved the bits to the right to simplify calculations and software. Now that we can get a voltage from the distance sensor as a 7-bit number, how do we convert it to a distance? I'll explain and demonstrate two techniques: one that uses a lookup table and another that uses the straight-line equation shown earlier.

Lookup Tables

A lookup table works as the name suggests. Software can use the voltage value from a sensor as the index into a table that holds corresponding values in units such as centimeters, kilopascals, or degrees. This approach works quickly, but can take a lot of memory – one or two bytes per ADC value. If you have a very accurate sensor and a 12-bit ADC, for example, you might need 2^{12}, or 4096 array elements. Thankfully we don't need that much memory for the Sharp distance-sensor table.

I'll go through the math used to create the lookup table step by step, but you may skip this section and simply put your data into the Excel Sensor_Output.xls spreadsheet in the Experiment 20 folder. Then use the Excel spreadsheet ADC_increments.xls to create the lookup table data for your sensor. If you skip ahead to the **Math by the Numbers** section, please come back and read the steps below.

Step-by-Step Lookup-Table Math

Because the Excel spreadsheet Sensor_Output.xls used *my* experimental data to find the best straight-line fit (**Figure 20.11**), I will use the corresponding straight-line equation for that data in the steps that follow. It will calculate the distance for each of the voltages along the least-squares-fit line.

1. As shown in the bottom row of **Table 20.4**, the 7-bit result from the ADC has a minimum step voltage of 0.0258 volts. This voltage corresponds to the LSB in the shifted 7-bit result from the MCP3202 ADC.

2. Get the *straight-line voltage* values for the 80 and 7-cm measurements. *Do not use your measured voltages.* The least-squares fit process already used those voltages to find the straight line. Now you want information from that straight line. Place the cursor over each of these two points in the best straight-line-fit plot and record the voltage for each:

   Your least-squares straight-line voltage at 80 cm (0.0125/cm on the line): _____

   Your least-squares straight-line voltage at 7 cm (0.143/cm on the line): _____

   My voltages from the Excel chart amount to:

   Least-squares straight-line voltage at 80 cm: 0.395 volts. Straight-line voltage at 7 cm: 3.10 volts.

   Given 0.0258 volts per LSB step in the 7-bit ADC result, you want to know what ADC value corresponds to the straight-line voltages at 80 and 7 centimeters. So, divide each voltage as shown here:

$$\frac{x\ volts}{0.0258\ volts\ /\ ADC\ step} = \_\_\_\_\_\ ADC\ steps$$

For my straight-line voltages, first for 80 cm:

$$\frac{0.395\ volts}{0.0258\ volts\ /\ ADC\ step} = 15.3\ ADC\ steps$$

and then for 7 cm,

$$\frac{3.10\ volts}{0.0258\ volts\ /\ ADC\ step} = 120\ ADC\ steps$$

If necessary, round values to the nearest integer. That yields 15 ADC steps with the target at 80 cm and 120 steps for it at 7 cm. These calculations let us know when the 7-bit ADC puts out the binary value 0001111_2 or 15, that's equivalent to 80 cm. The value 15 serves as the index into the 128-value lookup table. Thus in an array named, say, `Distance_array`, element `Distance_array[15]` would contain the decimal value 80. (Remember: The 12-bit ADC value gets shifted five bits to the right, so ADC values range from 0 to 127_{10}.

3. Instead of slogging through the math to calculate the value for each array element, I created an Excel spreadsheet that will do the job. Locate the spreadsheet ADC_increments.xls in the Experiment 20 folder and open it in Excel. The sheet contains my data and it shows the distances that correspond to each voltage step. You may enter your own data and have Excel calculate values for your Sharp distance sensor. The highlighted section in the ADC_increments.xls sheet shows the distance values for 80 to 7 cm distances. The calculations involve rounding and use only three significant figures, so distances *approximate* the actual on-the-ground measurements. Still, they come close; usually within one centimeter of the actual distance. Later in this experiment you will learn how to use values in the ADC_increments.xls spreadsheet to create values for an array and then see how well your table values match actual distances. Next I'll explain how you can use math, instead of a lookup table, to determine distances based on the sensor's voltage output. You might prefer this approach.

Math by the Numbers

The descriptions that follow start with the same equation for the least-squares-fit straight line shown earlier in Figure 20.12. It calculates a voltage (y) when given a distance in centimeters (x) for the Sharp GP2Y0A21YK0F sensor:

$$y\ volts = \left(20.7\ volt\text{-}cm * \frac{x}{cm}\right) + 0.136\ volts$$

But we need the distance (x) based on the voltage (y). A bit of algebra lets us rearranges the equation in the following ways. To keep the equations easy to understand I removed the units of centimeters and volts:

$$y = (20.7 * x) + 0.136$$

Subtract 0.136 from each side and get:

$$y - 0.136 = 20.7 * x$$

Because x has units of 1/cm, we want the value of $1/x$ to get the distance. Divide each side of the equation by x, and that algebraic rearrangement gets us to:

$$\frac{(y - 0.136)}{x} = 20.7$$

Division of each side by (y - 0.136) gives us:

$$\frac{1}{x} = \frac{20.7}{(y - 0.136)}$$

Now, when we have voltage *y*, we can calculate the distance from a sensor to an object. Say the 7-bit ADC produces the value 0100110_2, or 38. That value doesn't mean 38 volts nor does it mean 38 centimeters! Instead, it means the sensor has produced a voltage equal to 38 times the 0.0258-volt "weight" for the LSB in the 7-bit ADC value. Thus the corresponding voltage input to the ADC equals:

$$n\ steps * 0.0258\ volts\ /\ step = y\ volts$$

Rather than calculate this voltage and then the distance, we substitute the equation above for the voltage, *y*:

$$\frac{1}{x} = \frac{20.7\ volt\text{-}cm}{((n\ steps * 0.0258\ volts/step) - 0.136\ volts)}$$

Because *n* represents the 7-bit ADC output, the equation directly calculates the distance from the 7-bit ADC output. Without the units, the equation becomes:

$$\frac{1}{x} = \frac{20.7}{((n * 0.0258) - 0.136)}$$

For the ADC value 0100110_2, or 38, *n* becomes 38, and the calculation yields:

$$\frac{1}{x} = \frac{20.7}{(38 * 0.0258) - 0.136}$$

which becomes:

$$\frac{1}{x} = 24.5\ cm, \text{rounded to } 25\ cm$$

That value agrees with the Excel spreadsheet values calculated earlier.

Unfortunately, a Propeller MCU – and many other MCUs – do not have circuits that can directly perform math with numbers such as 20.7 and 0.844. But we can *scale* these numbers to create integers that an MCU can handle. And you may apply this technique to help in situations that include decimal fractions.

When you multiply one side of an equation by one, the equality still holds. Take the equality below as an example:

$$5\ cm = 2.5\ cm * 2$$

and

$$5\ cm = 2.5\ cm * 2 * \frac{100}{100}$$

Yields the same result. To scale the values in the equation:

$$\frac{1}{x} = \frac{20.7}{((n * 0.0258) - 0.136)}$$

multiply the right side by 1, expressed at 2048/2048. That multiplication changes the values of the fraction's numerator and denominator, but in the same proportion. (Units removed for clarity.)

$$\frac{1}{x} = \frac{20.7}{((n * 0.0258) - 0.136)} * \frac{2048}{2048}$$

$$\frac{1}{x} = \frac{20.7 * 2048}{\big((n * 0.0258) - 0.136\big) * 2048)}$$

$$\frac{1}{x} = \frac{4240}{((n * 0.0258 * 2048) - 0.136 * 2048)}$$

and finally:

$$\frac{1}{x} = \frac{4240}{((n * 53) - 279)}$$

Now the equation uses only integer values. Substitute 38 for *n* and do the math. What distance did you calculate? I came up with 24.4, or 24 cm. Why not 24.5? The difference relates to rounding in the calculations that led to the final equation with integers. And, anyway we have a sensor with an accuracy of only about 1 cm.

Now an MCU can work with the integers used above and programmers would not worry about how to handle decimal fractions. Keep in mind that multiplying one side of an equation by 2048 / 2048 has the same effect as multiplying it by 1. That's the trick.

IMPORTANT: Scaling values does not mean simply multiplying every decimal fraction by the same number. Ensure you understand the algebra involved before you do scaling. Always use a calculator to test scaled values in an equation.

What size variable do we need for the math operations? The largest value from the ADC equals 127 and the smallest value (other than 0) equals 1, so test the equation with the upper and lower values for *n*. What did you observe? In both cases, the largest value equals the constant, 42400, or $0\_10100101\_10100000_2$. At first glance it looks like a 16-bit word would work, but the Propeller treats the MSB in a word variable as a sign bit for 2's-complement numbers, so a 1 in the MSB indicates a negative number. Not what we want. Use 32-bit long variables for values and results in this case.

If you substitute the value 1 in the equations above for n, the distance comes to -188 cm. As explained earlier, distances less than about 7 cm cause the distance sensor to produce unusable voltages. Any ADC values less that 15 or 16 should cause software to indicate an "invalid measurement."

Sensor Hardware and Software

Now that you have values for an array or a scaled equation, you need a circuit and software for the Microchip MCP3202 ADC. You learned about the ADC in Experiment 16 and will use a similar ADC circuit and software next.

Step 4.
Wire the circuit shown in **Figure 20.13**. Remember the GP2Y0A21YK0F distance sensor needs a 5-volt power source. The ADC can operate from the 3.3-volt supply on the Propeller P8X32A board. After you build the circuit, start the Parallax Tool and open the Experiment 20 folder. Locate **Program 20.1** in that folder and run it. This program takes a12-bit voltage value from the ADC and displays the seven MSBs on seven LEDs (P6 through P0) on the Propeller board. The P6 LED represents the MSB of the 7-bit distance measurement. Remember, the software produces a 7-bit result from the ADC raw data. This 7-bit value represents the raw information from the sensor. How we get a distance value comes shortly.

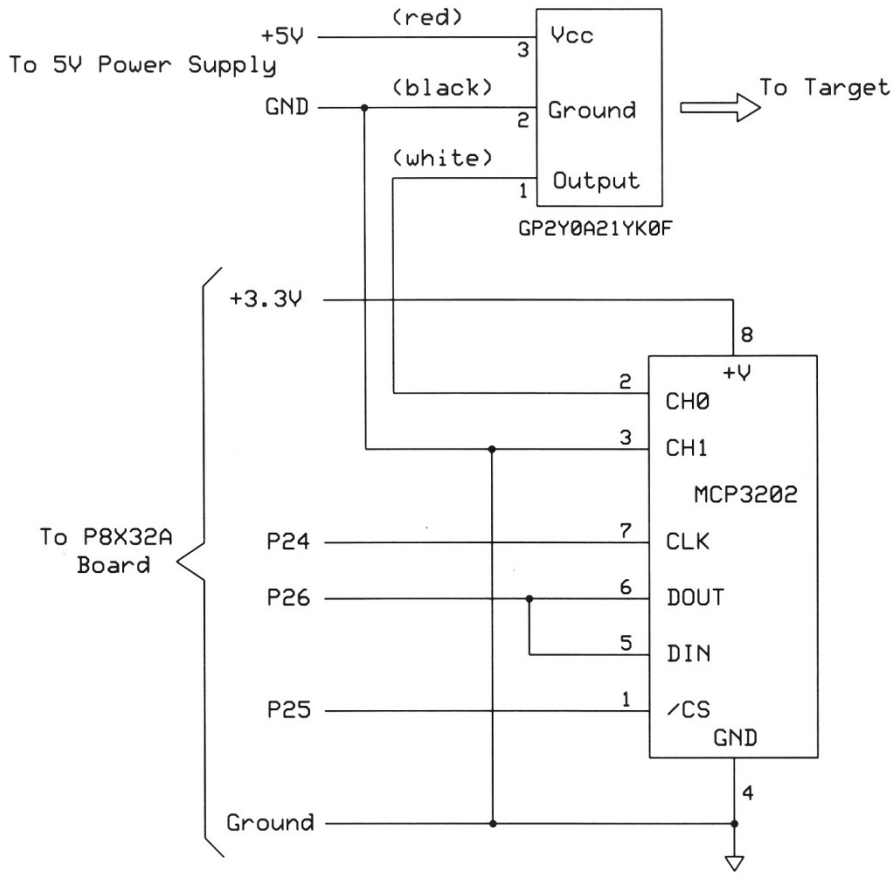


Figure 20.13.
Schematic diagram for a Sharp GP2Y0A21YK0F distance sensor and a Microchip MCP3202 12-bit ADC connected to a Propeller P8X32A board. This circuit includes a ground in common with the 5V and 3.3V circuits. Ensure you connect to ground on the Propeller board and to ground on the 5-volt power supply. Without a common ground, the circuit will not work properly.

Program 20.1

```
''*****************************************************
''*  Program 20.1, Analog-to-Digital Converter and
''*  Distance Measurements
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*
''*  Modified by Jon Titus, 02-03-2014 Rev. 1
''*  Copyright 2014
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays 7 MSBs on LEDs for voltage on
```

```
''*  ADC Channel 0 input, pin 2. Be sure to ground input 1,
''*  at pin 3. This program does not use that input.
''*****************************************************
OBJ
  MCP3202   : "MCP3202"           'MCP3202 driver objects here
  delay     : "timing"            'timing.spin file

CON
  _CLKMODE      = XTAL1 + PLL4X   'Set MCU clock operations
  _XINFREQ      = 5_000_000       '5MHz Crystal

  clock_pin         = 24          'Pins assigned to MCP3202 ADC
  data_pin          = 26
  chip_select_pin   = 25

VAR
  long  CH0                        'Variable for Channel-0 data

'Main program starts here
PUB Main

  'MCP3202 ADC start-up configuration
  'See MCP3202 datasheet for configuration options
  MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

  DIRA[23..16] := %11111111         'Set onboard LED pins as
                                    'outputs

repeat                              'This main loop runs forever
  CH0 := MCP3202.in(0)              'Get digital value for
                                    'signal on Channel 0
  CH0 := (CH0 >> 5) & %01111111     'Shift 5 bits to right to
                                    'move 7 MSBs to D6 – D0
  OUTA[23..16] := CH0               'AND with 01111111 to let
                                    'only 7 LSBs go to LEDs
  delay.pause1ms(100)               '100 msec delay

' –  – -end Program 20.1 –  – -
```

Step 5.

Move an object in front of the sensor and watch how the LEDs react. The binary values should change as you move the object. When you point the sensor at an object 180 to 210 cm (6 to 7 feet) away, what binary value do the LEDs display? What binary value do you observe with an object placed 10 cm away from the sensor?

When I pointed the sensor at a distant book shelf, the LEDs showed binary values of 0000110, 0000111, and 0001000, or decimal 6, 7, and 8, respectively.

For an object placed 10 cm away, I saw binary values of 1010111 and 1010110, or decimal 87 and 86. I also tested the distance to an object 7 cm away, and the binary values varied between 1110000 and 1101111, or decimal 112 and 111.

How well do your ADC values match the distances in your ADC_increments.xls spreadsheet? **Table 20.5** shows my measured results and those calculated in the spreadsheet. Except for a distant object, the results matched or came close to those calculated.

Table 20.5. Measured ADC-output values and corresponding measured and calculated distances.

ADC Output (binary)	ADC Decimal Equivalent	Measured Distance (cm)	Calculated Distance (cm)
0000110	6	>80	1089
0000111	7	>80	460
0001000	8	>80	296
1010110	86	10	10
1010111	87	10	10
1101111	111	7	8
1110000	112	7	8

Step 6.

Because objects can move slightly, or the sensor could move, consider averaging several measurement values to reduce the rapid fluctuations, or "noise," in measured values from the sensor.

Program 20.2 averages eight samples taken 100 msec apart. Then it sends the result to the LEDs. Run this program to see whether or not it reduces the fluctuations in the binary values on the LEDs. Instead of mathematically dividing the sum of the eight measurements by 8, the software shifts the sum three positions to the right. Those shifts have the same effect as dividing a binary value by 8.

You may reduce the time between samples, increase the number of samples to average, or both. I recommend sample sizes of 8, 16, 32, and other powers of 2. In these cases, shift operations quickly perform the needed "division" of the sum. Keep in mind that you will need a 16-bit word variable, or perhaps a 32-bit long variable, for the sum, depending on the number of values you plan to average.

Program 20.2.

```
''*****************************************************
''*  Program 20.2, Analog-to-Digital Converter and
''*  Distance Measurements, averages 8 samples.
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*
''*  Modified by Jon Titus, 02-03-2014 Rev. 1
''*  Copyright 2013
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays 7 MSBs on LEDs for voltage on
''*  ADC Channel 0 input, pin 2, Ground input 1,
''*  at pin 3. Not used.
''*  Averaging eight ADC values helps reduce small distance
''*  changes in sensor readings.
''*****************************************************
OBJ
  MCP3202   : "MCP3202"            'MCP3202 driver objects here
  delay     : "timing"             'timing.spin file
```

```
CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

  clock_pin         = 24            'Pins assigned to ADC
  data_pin          = 26
  chip_select_pin   = 25

VAR
  long  CH0                         'Variable for Channel-0 data
  long  ADC_sum                      'Variable for sum of 8
                                    'values
'Main program starts here
PUB Main
  'MCP3202 ADC start-up configuration
  'See MCP3202 datasheet for configuration options
  MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

  DIRA[23..16] := %11111111       'LEDs as output pins

  repeat                          'This main loop runs "forever"
    ADC_sum := 0                  'Preset sum of ADC values to 0
      repeat 8                    'Repeat this loop 8 times
        CH0 := MCP3202.in(0)      'Get ADC data for Channel 0
        ADC_sum := ADC_sum + CH0  'Add latest ADC value
        delay.pause1ms            '100-msec delay
      ADC_sum := ADC_sum >>3      'Divide ADC_sum by 8
      ADC_sum := (ADC_sum >> 5) & %01111111 'Pass only 7 bits
      OUTA[23..16] := ADC_sum               'Output to LEDs
      delay.pause1ms(100)                   '100 msec delay

  ' –  – -end Program 20.2 –  – -
```

If you haven't already started **Program 20.2**, run it now. Do the values displayed on the LEDs remain more stable than when you ran **Program 20.1**? I found they did.

Step 7.

Now that you have shown the measured and calculated distances match well for specific ADC-output values, you need distance values for a lookup table. But no one wants to enter 128 values in an array by hand, so some cut-and-paste and reformatting operations will create the array values for you. If you don't want to work with your own values in a lookup table, you can use mine.

The **Notes on Excel** section at the end of this experiments include steps and illustrations that explain how to use Microsoft Office Excel and a free hexadecimal-editor program to format the spreadsheet values into stand-alone values for a Propeller Spin program lookup table. If you want to use your own values, please go to the notes section and follow the instructions. Then continue to Step 8 when you have the 128 array values displayed in the Windows Notepad editing window.

Step 8.

Program 20.3 takes eight voltage samples, averages them, and then uses the average to locate the corresponding distance in the `Dist_cm` array. After averaging eight values and shifting bits, say the 7-bit result equals 0100011_2, or 35. This value serves as the index into the `Data_cm` array. Thus at array location, `Data_cm[35]`, I have the value 27 (for 27 cm), the distance from the sensor to the object in front of it.

Program 20.3

```
''*****************************************************
''*  Program 20.3, Lookup Table Approach to distance
''*  measurement with a Sharp sensor GP2Y0A21YK0F.
''*  This routine averages 8 samples and finds distance
''*  in a table of distance values.
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*
''*  Modified by Jon Titus, 02-05-2014 Rev. 3
''*  Copyright 2013
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays 7 MSBs on LEDs for voltage on
''*  ADC Channel 0 input, pin 2, Ground input 1,
''*  at pin 3. Not used.
''*****************************************************
OBJ
  pst       : "FullDuplexSerial"    'serial comm objects here
  MCP3202   : "MCP3202"             'MCP3202 driver objects here
  delay     : "timing"              'timing.spin file

CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

  clock_pin         = 24            'Pin assignments for MCP3202
  data_pin          = 26
  chip_select_pin   = 25

DAT
'Lookup table data for Dist_cm[x] array
Dist_cm byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 82, 75, 68, 63, 58,
54, 51, 48, 45, 43, 41, 39, 37, 35, 34, 32, 31, 30, 29, 28, 27, 26, 25, 25,
24, 23, 22, 22, 21, 21, 20, 20, 19, 19, 18, 18, 17, 17, 17, 17, 16, 16, 16,
15, 15, 15, 14, 14, 14, 14, 13, 13, 13, 13, 13, 12, 12, 12, 12, 12, 11, 11,
11, 11, 11, 11, 11, 10, 10, 10, 10, 10, 10, 10, 10, 9, 9, 9, 9, 9, 9, 9, 9,
9, 9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0

VAR
  long  CH0                         'Variable for Channel-0 data
  long  ADC_sum                     'Variable for sum
  byte PropPin_SerOut               'Serial output to PST
  byte PropPin_SerIn
  byte PropPin_SerMode
  word SerPort_BaudRate             'Variable for serial comm

'Main program starts here
PUB Main
  PropPin_SerOut    := 30           'P30 for USB transmit
  PropPin_SerIn     := 31           'P31 for USB receive
  PropPin_SerMode   := 0
  SerPort_BaudRate  := 9600         'Set baud rate at 9600 bps

  'Serial-communication start-up operation
  pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)
```

```
  'MCP3202 ADC start-up configuration
  'Enable Channels 0 and 1 for single-ended analog inputs
  'This experiment uses only Channel 0. Ground Channel-1 input.
  'See MCP3202 datasheet for configuration options

  MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

  DIRA[23..16] := %11111111        'Set LED pins as outputs

repeat                            'This main loop runs "forever"
  ADC_sum := 0                    'Preset the sum to zero
  repeat 8                        'Repeat this loop 8 times
     CH0 := MCP3202.in(0)         'to average 8 ADC values
     ADC_sum := ADC_sum + CH0     'Add the latest ADC value
     delay.pause1ms(100)          '100-msec delay (arbitary)
  ADC_sum := ADC_sum >>3          'Divide by 8 to get average
  ADC_sum := (ADC_sum >> 5) & %01111111
  OUTA[23..16] := ADC_sum         '7 bit averaged data to LEDs
  pst.tx(1)                       'Move cursor to home in PST
  pst.str(string("Distance in cm "))   'Display this string
  pst.tx(11)                      'Clear data
  pst.dec (Dist_cm[ADC_sum])      'Send PST distance as decimal
                                  'number
  pst.tx(13)                      'Move cursor to a new line

' –  – -end Program 20.3 –  – -
```

Step 9.
You may run **Program 20.3** "as is" with my data to see how it works with your sensor. The measured distances appear in the Parallax Serial Terminal (PST) window (9600 bits/sec). Start **Program 20.3** (F10) and then switch to the PST (F12). Click on the "Enable" button in the bottom-right corner of the PST window and then move an object toward and away from the sensor. You should see the distance change as reported in the PST window. Because the program includes time delays, a new measurement won't appear immediately after you move a target object.

Step 10.
If you wish, you may substitute your spreadsheet values for mine in **Program 20.3**. Go back to the Parallax Tool window and find my array values in the green `DAT` area. Delete the entire *long* line that starts: `Data_cm byte 0, 0, 0...` and so on.

Within the open `DAT` area, type and leave a space after `byte`:

`Data_cm byte` < – leave a space here

Next, enter your values – separated by commas – in the Propeller Tool editing window after the space that follows: `Dist_cm byte`. The line of values will extend far to the right and past the right side of the Parallax Tool window, although you can scroll right to see all values. (I found it easier to view the array values in a text editor that wraps long lines so they fit in the word-processor window.

Step 11.
The first 16 calculated distance values in the Calculated Distance column in my ADC_increments.xls spreadsheet appear in **Table 20.5**. The Excel Calculated Distances for ADC Outputs 0 through 14 contain

"nonsense data." The sensor cannot give us minus distances, nor can it work with objects more than 80 cm away.

Table 20.5. Calculated lookup-table values for the Sharp distance sensor.

ADC Output(decimal)	Corresponding Voltage (volts)	Calculated Distance (cm)
0	0	0
1	0.0258	-188
2	0.0516	-245
3	0.0774	-353
4	0.103	-627
5	0.129	-2957
6	0.155	1089
7	0.181	460
8	0.206	296
9	0.232	216
10	0.258	170
11	0.284	140
12	0.31	119
13	0.335	104
14	0.361	92
15	0.387	82

The first few values in your `Data_cm byte` statement might look like this:

`Data_cm byte 0, -185, -218, -376, 1100,` and so on.

Before you enter any data into **Program 20.3**, r*emove the first zero* and change any negative values and any values above 82 to zero. Now, the `Data_cm byte` statement should look similar to my data in **Program 20.3**, the beginning and end of which you see here:

```
Data_cm byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 82...  ... 7, 7, 0, 0,
0, 0, 0, 0, 0, 0
```

After you insert and modify your data, run your version of **Program 20.3**. Do you observe reasonable distances shown in the PST window? If not, go back to the original program and run it gain. Then insert and modify your data again.

Step 12.

Although Sharp provides no voltage data for an object closer than 5 cm from a sensor, my observations show the voltages remain constant or *decrease* for very close objects. How can we solve that problem? Mount the sensor so no object can get closer than 7 cm. Consider the distance from 0 to 7 cm as a "dead zone," or unusable-distance range. Other types of sensors can measure shorter distances.

In Step 11 you replaced "nonsense" data with zeros. How should the software react if the distance value, `Data_cm[ADC_sum]`, it retrieves from the array equals 0? Use software to test the value prior to the statement that displays a distance on the PST:

```
pst.dec(Data_cm[ADC_sum])
```

If the software detects a zero, it could display an "Out of Range" warning. An `if` statement will do the job. Change the code if you want to test your ideas.

Step 13.

The software in **Program 20.4** uses the straight-line equation for my data with the scaled values. Any average of eight ADC values that has a result less than 15 causes an error message to appear in the PST window. Run this program and look at the code. Would you rather calculate the distance or use a lookup table with values entered by hand? I like the calculations because I can easily "tweak" or "tune" the equation for a different sensor without having to create a new lookup table. And the integer-number math takes less memory. But sometimes we don't have a choice.

Program 20.4

```
''*****************************************************
''*  Program 20.4, Math approach to distance
''*  measurement with Sharp sensor GP2Y0A21YK0F.
''*  This routine averages 8 samples and uses scaled-math
''*  operations to calculate a distance.
''*  Based on code created by Ken Gracey, Parallax
''*  Modified by John Abshier (jabshier on Forum)
''*
''*  Modified by Jon Titus, 02-05-2014 Rev. 1
''*  Copyright 2013
''*  Software for a Microchip Technology MCP3202 12-bit
''*  ADC that displays 7 MSBs on LEDs for voltage on
''*  ADC Channel 0 input, pin 2, Ground input 1,
''*  at pin 3. Not used.
''*****************************************************
OBJ
  pst       : "FullDuplexSerial"    'serial comm objects here
  MCP3202   : "MCP3202"             'MCP3202 driver objects here
  delay     : "timing"              'timing.spin file

CON
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

  clock_pin         = 24            'Pin assignments for MCP3202
  data_pin          = 26
```

```
  chip_select_pin   = 25


VAR
  word  CH0                   'Variable for Channel-0 data
  word  ADC_sum               'Variable for averaging sums
  word  ADC_avg               'Average of ADC values
  long  Volt_cm               'Volt-cm constant for calculation
  long  Volts_per_step        'Voltage per step from ADC
  long  Intercept             'Y-axis intercept for straight-
                              'line equation
  long  Distance              'Calculated distance in cm

  byte  PropPin_SerOut        'Variable for P8X32A pin P30
  byte  PropPin_SerIn         'Variable for P8X32A pin P31
  byte  PropPin_SerMode       'Variable for serial comm format
  word  SerPort_BaudRate      'Variable for serial comm bit
                              'rate

'Main program starts here
PUB Main
  PropPin_SerOut    := 30       'P30 for USB transmit
  PropPin_SerIn     := 31       'P31 for USB receive
  PropPin_SerMode   := 0
  SerPort_BaudRate  := 9600     'Baud rate at 9600 bps

  'Serial-communication start-up operation
  pst.Start(PropPin_SerIn, PropPin_SerOut, PropPin_SerMode, SerPort_BaudRate)

  'Scaled constants (multiplied by 2048) for straight-line
  'calculations
  Volt_cm := 42400
  Volts_per_step := 53
  Intercept := 279

  'MCP3202 ADC start-up configuration
  'Enable Channels 0 and 1 for single-ended analog inputs
  'This experiment uses only Channel 0. Ground Channel-1 input.
  'See MCP3202 datasheet for configuration options
  MCP3202.start(data_pin, clock_pin, chip_select_pin, %0011)

  DIRA[23..16] := %11111111           'LED pins as outputs

repeat                                'This loop runs forever
  ADC_sum := 0                        'Preset the sum of ADC
                                      'values to zero

  repeat 8                            'Repeat this loop 8 times
      CH0 := MCP3202.in(0)            'Get value for Channel 0
      ADC_sum := ADC_sum + CH0        'Add the latest ADC value
      delay.pause1ms(20)              '100-msec delay (arbitary)

  ADC_avg := ADC_sum >>3              'Divide SUM by 8
  ADC_avg := (ADC_avg >> 5) & %01111111
  OUTA[23..16] := ADC_avg             'Output to LEDs

  if ADC_avg < 15                     'Out-of-range condition?
```

```
    pst.tx(1)                          'Yes, move cursor to home
    pst.str(string("                     ")) 'Print spaces
    pst.str(string("Out of Range"))    'Print this warning
  else                                 'No, measurement OK
    'Calculate distance
    Distance := Volt_cm / ((Volts_per_step * ADC_avg) - Intercept)

    pst.tx(1)                           'Move cursor to PST home
    pst.str(string("Distance in cm "))   'Print this string
    pst.tx(11)                         'Clear data from end of
                                       'string to end of line
    pst.dec (Distance)                 'Send PST a distance as a
                                       'decimal number
    pst.Tx(13)                         'Move cursor to a new
                                       'line
' –  – -end Program 20.4 –  – -
```

Step 14.

Programs 20.2, **.3**, and **.4** all lack an enhancement you might want to add – rounding. You learned about rounding in an earlier experiment, bit I'll recap the technique here. After you sum the 8, 16, or 32 values, you would usually divide by 8, 16, or 32, respectively. Instead, divide by 4, 8, or 16 respectively. Then add 1 and divide by 2. If the summed value has a 1 in the half-bit position, an added 1 will create a carry into the next most-significant bit position. Dividing by 2 completes the division and you have a more accurate result. Here's a code snippet you can study:

```
repeat 8                               'Repeat this loop 8 times
      CH0 := MCP3202.in(0)             'Get value for Channel 0
      ADC_sum := ADC_sum + CH0         'Add the latest ADC value
      delay.pause1ms(20)               '100-msec delay (arbitary)

  ADC_avg := ADC_sum >>2               'Divide SUM by 4
  ADC_avg := ADC_avg + 1               'round up
  ADC_avg := ADC_avg >>1               'Divide by 2
  OUTA[23..16] := ADC_avg              'Output to LEDs
```

Keep in mind the averaging and rounding operations will give you better accuracy, but the distances still have a resolution of only one centimeter.

Notes on Excel

The following explanation assumes you have used the program ADC_increments.xls to calculate a set of 128 values based on your distance versus voltage information gathered earlier in this experiment. You must have that information before you follow these steps, and you must have a Windows PC and Microsoft Excel; or, if you use a Macintosh, you can use iWorks Numbers as your spreadsheet and a hexadecimal editor such as "Hex Fiend," available for free: http://ridiculousfish.com/hexfiend/.

1. Open the Excel ADC_increments.xls file that contains your data and the 128 "Calculated Distance" values. Open a blank Excel spreadsheet.

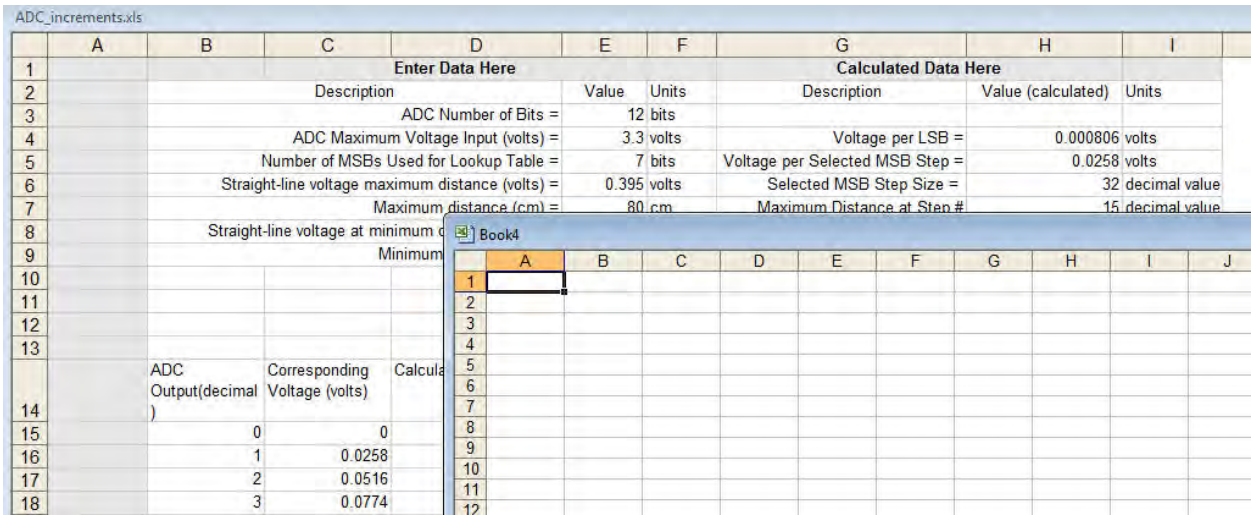


Figure 20.14.
Excel ADC_increments.xls file and a blank Excel spreadsheet

2. Highlight and copy the 128 "Calculated Distance" values. Switch to the blank spreadsheet and highlight cell A1. Right click your mouse and select "Paste Special..." Under the "Paste" section of the window that opens, click on "Values." This step pastes the distance *values* rather than the formulas that calculated them.

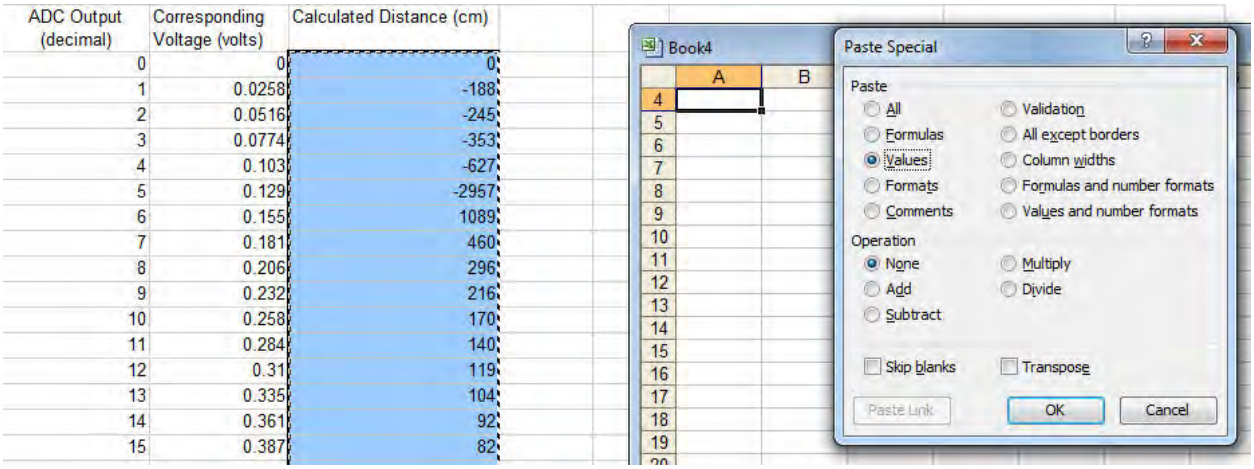


Figure 20.15.
Copy and paste the Calculated Distance values as "Values" in the first column of the blank Excel sheet.

3. Save the spreadsheet that contains the single column of data in your Experiment 20 folder. Use a name such as Distance_data.xls, or joesDistanceData.xls. Then, highlight all 128 values in column A in that spreadsheet. In the top main menu, select "Format" and then "Cells." Choose "Number" and set "Decimal Places" to zero (0). Numbers should change from, say, 12.34567 to 12.

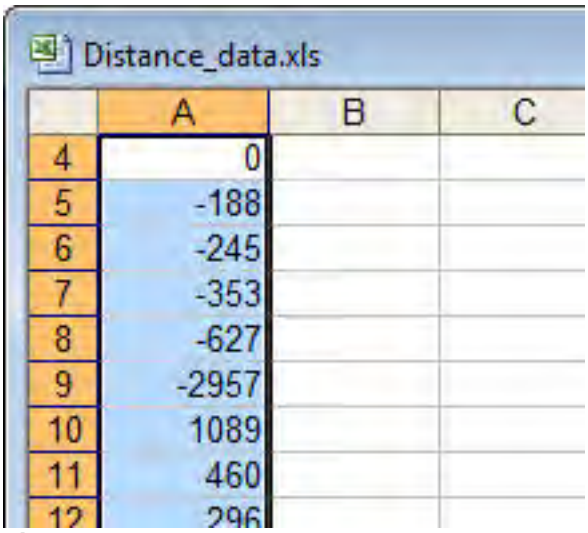


Figure 20.16.
Formatting data as Number with no decimal fractions causes Excel to round values to whole numbers, or integers.

4. In the top main menu, click on "File" and then on "Save As..." Give your file a new name, such as Distance_array or joesDistanceArray and select the Experiment 20 folder for it. Then select "Save as type:" and choose "CSV (Comma delimited)(*.csv)". You might see warning messages about multiple sheets or keeping data in the same format. Click "Yes" or "OK" for these messages.

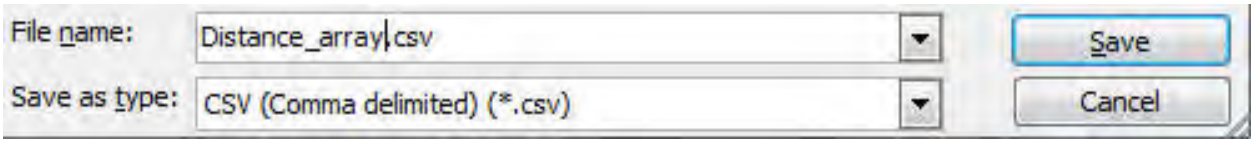


Figure 20.17.
Save the Distance_array information in a CSV file.

5. Download a free hexadecimal-file editor such as HxD (http://mh-nexus.de/en/hxd/) and install it on your computer. Then run HxD and use it to open the your file Distance_array.csv. You should see a display such as that shown in **Figure 20.18**.

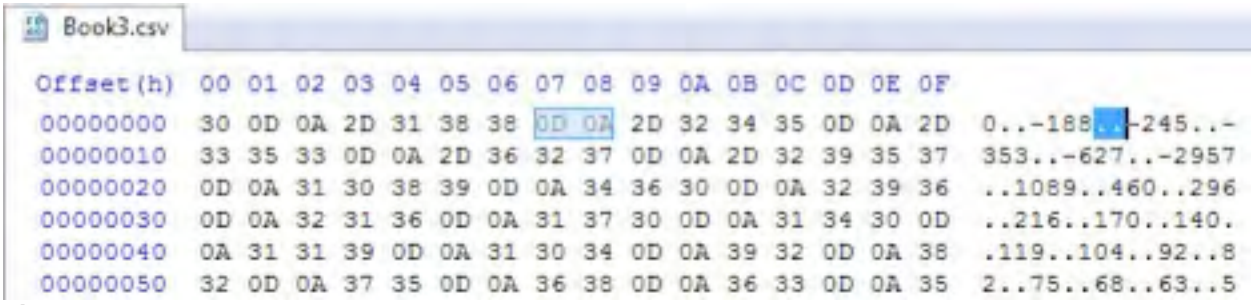


Figure 20.18.
A hexadecimal file editor lets people quickly change values, such as the carriage-return and line-feed on the left (0D 0A) to other hex values.

6. The blue numbers and letters on the top and left side of the window help programmers locate information, but we'll ignore them. Highlight the double periods between values as shown above. The corresponding hex values will become highlighted within the large hex-value area. You can see 0D and 0A highlighted. These ASCII codes represent a carriage return (CR) and a linefeed (LF).

7. In the HxD hexadecimal-editor menu, click "Search" and then "Replace." In the window that opens, enter the values shown in bold letters in **Figure 20.19**. These hex values use the numeral zero and *not* the letter O.

   Search for: **0D0A**
   Replace with: **2C20**
   Datatype : **Hex-values**

8. The hex values 2C and 20 replace the CR/LF bytes with a comma and a space. This operation formats the data for your Propeller program from a columnar list to a linear list with commas and spaces:

```
12    to:   12, 15, 34, 92, ...
15
34
92
...
```

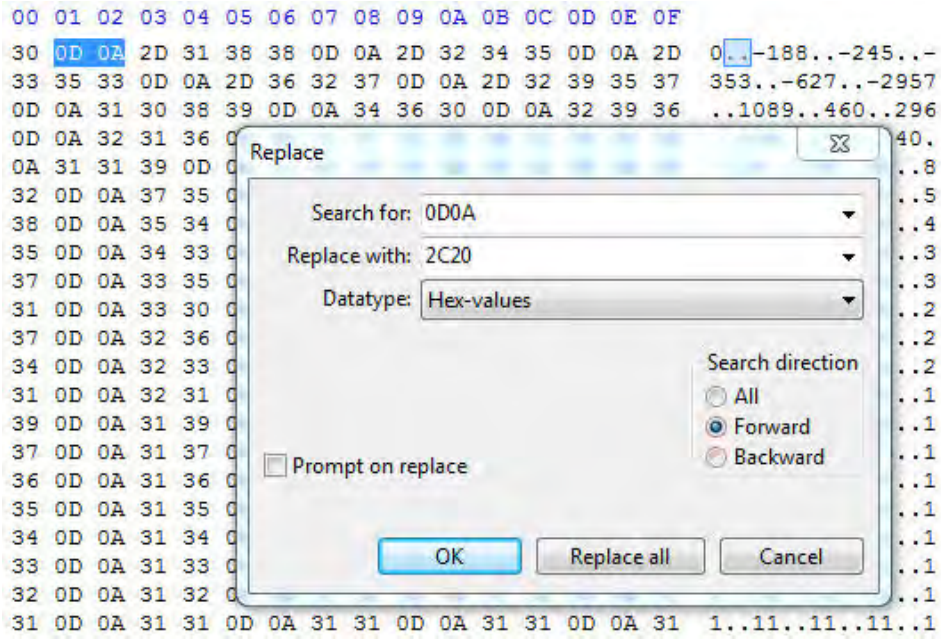


Figure 20.19.
Replacing the 0D0A with 2C20 inserts a comma and a space in the hexadecimal values. This replacement formats the decimal values for use in a SPIN-language array.

9. You should see the notice, "Replaced 127 occurrences..." or "Replaced 128 occurrences..." appear in a new window.

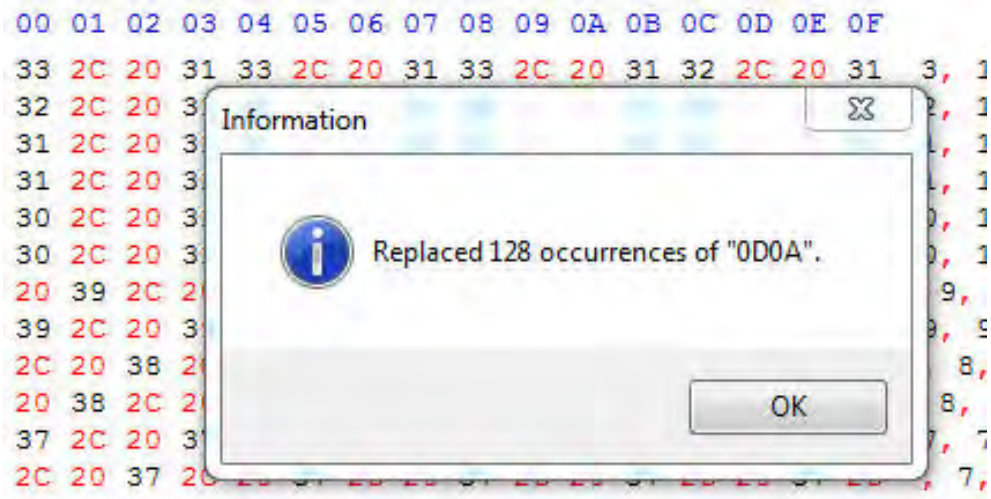


Figure 20.20.
The completion message for the search-and-replace operation.

10. In the HxD main menu, choose "File" and then "Save." The reformatted data goes back into your file.

11. Go to Microsoft Notepad and open your Distance_array.csv file. You should see 128 values separated by commas such as:

    0, 0, 82, 72, and so on...

    Leave Notepad open with your values shown in the editing window. You will use these values to create an array for a Propeller program.

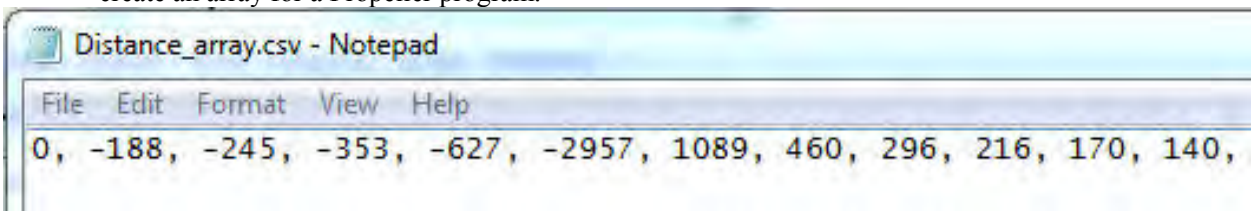


Figure 20.21.
A reformatted set of data with a space and a comma ready for use in a SPIN-language array.

12. When you define an array with a command such as:

    ```
    Dist_cm byte
    ```

    you can copy and past all the values after this statement to put the values in the array declaration in your Spin program.

References

"Range Finding Using Pulse Lasers," Osram Opto Semiconductors, 2004. http://www.osram-os.com/Graphics/XPic2/00054201_0.pdf/Range%20Finding%20using%20Pulsed%20Laser%20Diodes.pdf

"GP2Y0E02A,GP2Y0E02B,GP2Y0E03 Application Note," Sharp Electronics, Sheet No. OP13021EN. http://sharp-world.com/products/device/lineup/data/pdf/datasheet/gp2y0e02_03_appl_e.pdf.

Experiment No. 21 – How to Use PNP and NPN Transistors to Control LEDs

Abstract

Many of the previous experiments used integrated circuits to control LEDs, to receive or transmit signals, and to generate signals. Those ICs all use transistors – in some cases, many thousands. People can easily use these ICs in a circuit without knowing how transistors work. Now, though, you will get an introduction to transistors so you understand how they work and how to use them. Transistors serve several purposes. They can control currents and voltages a typical MCU or digital IC cannot, they help isolate circuits from other components, and they can amplify a signal. You will measure transistor characteristics to better understand how they work and how you can use them in your circuits.

Instead of creating example circuits here, you can find many helpful transistor circuits on the Internet. Use caution, though. Some experimenters and hobbyists cobble together a circuit that works for them, but they haven't used a manufacturer's data sheet to find important design information or they haven't used math to find proper component values.

Keywords

transistor, NPN, PNP, beta, gain, emitter, base, collector, 2N3904, 2N3906, bipolar, junction, LED

Requirements

(1) - Solderless breadboard
(1) - Voltmeter or volt-ohm-milliammeter (VOM)
(1) - DC power supply, 9 volts, or 9-volt battery and connector
(1) - LED, red or green
(1) - 2N3904 transistor, NPN,
(1) - 2N3906 transistor, PNP,
(1) - 220-ohm, 1/4-watt resistors, 5% (red-red-brown)
(1) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(1) - 100-kohm, 1/4-watt resistors, 5% (brown black-yellow)
(1) - Double-pole double-throw (DPDT) toggle switch
(1) - Resistor-substitution box or individual 1/4-watt, 5%. See Table 21.3 for values.
Hookup wires, solder, soldering iron, wire strippers.

Introduction

With the variety of integrated circuits available to experimenters and engineers, why do we need transistors? First, a few transistors might take less room in a circuit than an IC. Second, you can place individual transistors where you need them in a breadboard or on a circuit board instead of running wires to one IC. Third, a transistor can sink or source a higher current than many ICs, and can work at a higher voltage, too. Fourth, transistors cost less than ICs. A 2N3904 transistor, for example, costs *five cents* (US).

Transistors have a few drawbacks. First, they usually require a base resistor, which adds to the parts count for a design. Second, transistors rarely go into a socket, so removing them requires care not required to replace an IC in a socket. Third, depending on the circuit, transistors can take more space than an IC or ICs. Fourth, power transistors might need heatsinks or fans.

Because transistors can handle more power that most ICs, designers use them to control electrical loads such as solenoids, relays, lamps, LED arrays, heaters, and similar devices. Each type of load has its peculiarities that require attention. A relay or solenoid, for example, uses an electromagnetic coil to move a ferromagnetic rod or actuator. You find solenoids in washing machines and dishwashers as part of valves. They also help lock doors, control air flow in engines, dispense candy in machines, and so on.

Relays operate much like a solenoid, but with sets of contacts that let them turn high-current and high-voltage equipment on or off. Small relays route signals in audio equipment, while large relays control heavy-duty motors and pumps. I have a railway-signal relay that measures 15-by-15-by-20 cm and weighs about 4 kg. It makes a good bookend.

Of course when you only need to turn an LED on or off, a simple electrical switch will do the job. Close the switch, current flows, and an LED turns on. Open the switch and the LED turns off (**Figure 21.1**). Often, though, another electrical or electronic circuit must control an LED and in this situation, an inexpensive, small, and reliable transistor can serve as a switch. In this experiment you will learn how to use *bipolar junction transistors* (BJTs) to turn LEDs on or off and also to vary their brightness.

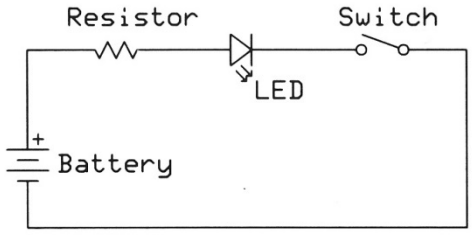


Figure 21.1.
A typical circuit in which a manual switch turns an LED on or off.

Bipolar transistors comprise two types of semiconductor materials noted as N or P. (Some authors use lowercase letters n and p.) Those letters indicate a negative- or positive-type semiconductor material, and semiconductor manufacturers produce N-P-N- and P-N-P-type bipolar transistors. In this experiment, you will learn how to use both types of transistors.

Transistor Basics

You can think of a transistor as a device that regulates the flow of *current*, much like how a fire hydrant regulates the flow of water. A small force on a fire-hydrant valve causes a more powerful force of water to flow into a fire hose. In a transistor, a small current supplied to or taken from it regulates a larger flow of current through the transistor.

Bipolar transistors have three electrical connections. Two serve as the path for current that will control LEDs or other devices. A small current at a third connection regulates the larger current flow. **Figure 21.2** illustrates several types of transistors in various packages. The smaller packages handle small currents of 50 to 200 mA. The larger packages can handle several amperes.

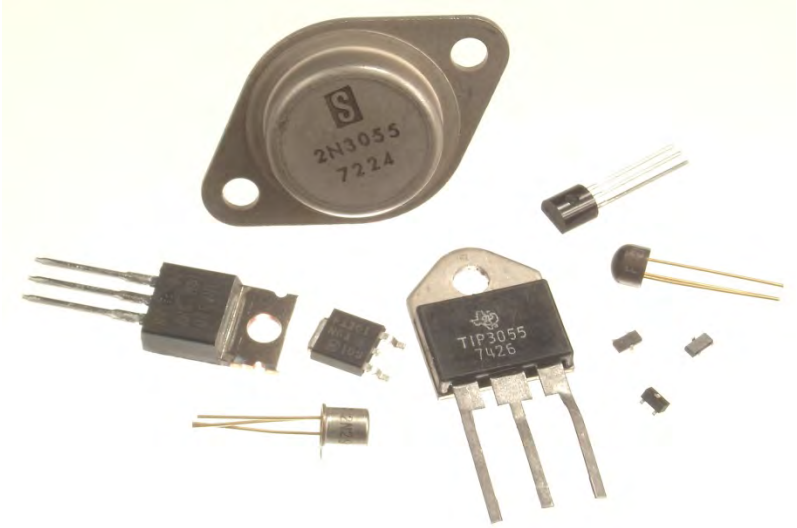


Figure 21.2.
Each bipolar transistor has three connections as shown here for different types of transistor packages. Metal cases for large transistors serve as one of the three electrical connections. The large metal packages usually attach to a heat sink. The small surface-mount technology (SMT) devices get soldered on a circuit board and require only small solder pads, not holes for pins.

This experiment uses transistors with wire leads that easily drop into solderless breadboards. Transistors also come in SMT packages, but you can't use them directly in breadboard, and they can prove difficult to solder. (Schmartboard offers many types of SMT adapters that simplify the use and soldering of small components. See the Reference section at the end of this experiment.)

Instead of drawing images of transistor packages and connections, electronic engineers use standard symbols that show the electrical connections and identify the transistor type: PNP or NPN. The diagram in **Figure 21.3** shows these symbols for both transistor types and the direction of current flow through them. Whether you have an NPN or a PNP transistor, they have the same three connections, labeled *collector*, *base*, and *emitter*. An arrow at an angle represents the emitter (E), a straight line at an angle represents the collector (C), and a horizontal line that forms a sideways "T" represents the base connection (B). The word "base" refers to an electrical connection and not to the bottom or other part of a transistor package.

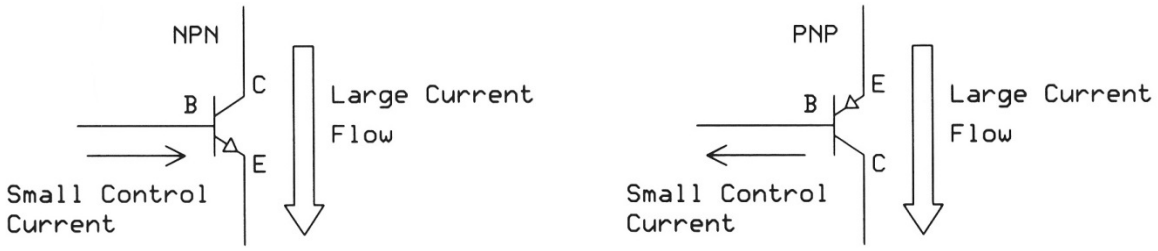


Figure 21.3.
Schematic-diagram symbols for an NPN and a PNP transistor with the designations for the emitter (E), base (B), and collector (C). A small current to or from the base connection controls a larger flow of current through the collector and emitter. The emitter current always equals the base current plus the collector current.

You can distinguish between a PNP and an NPN transistor by looking at the transistor symbol. In a PNP-transistor symbol, the emitter's arrow points to the base, while in an NPN-transistor symbol it points away from the base. Although NPN and PNP transistors have the same connection names, you cannot substitute an NPN for a PNP transistor or *vice versa*, because they behave differently in a circuit.

Note that the largest current always flows through the collector-to-emitter connections and the arrow in a transistor symbol indicates the direction this current flows. Also, the emitter connection always carries the sum of the large and small currents. That means for an NPN or a PNP transistor, $I_E = I_C + I_B$.

This experiment uses the inexpensive and common 2N3904 NPN and 2N3906 PNP transistors in plastic "TO-92" packages. Years ago, JEDEC, an electronics-industry group, adopted a standard nomenclature that uses numbers and a letter to identify semiconductor devices. In the 2N3904 part number, for example, the leading digit (2) indicates two semiconductor junctions, the letter N applies to all semiconductor devices, and the 3904 provides a model or part number. Many companies manufacture 2N3904 transistors as well as thousands of other types. You will see other semiconductor numbering schemes that do not use the xNxxxx format. JEDEC also specified many types of semiconductor packages. The abbreviation TO-92 stands for: Transistor Outline Package, Case Style 92.

Figure 21.4 represents a 2N3904 transistor in a TO-92 package. Labels identify the emitter, base, and collector leads. You will need this information in the steps that follow. The 2N3906 has the same arrangement of E-B-C leads. The transistors have small labels you can read through a magnifier. Note: Before you use other transistors, ensure you know the arrangement of the connections. Manufacturers' data sheets provide that information.

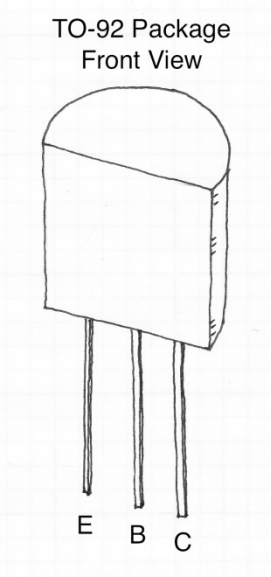


Figure 21.4.
Diagram of the 2N3904 and 2N3906 plastic package and the corresponding terminal designations.

Step 1.

In this step, you will use a 2N3904 transistor to turn an LED on or off. **Figure 21.5** shows the circuit, which you can assemble now in a solderless breadboard. One end of the jumper wire connects to the 100-kohm resistor. The other end should remain unconnected for now. You may use a rectangular 9-volt transistor-radio-type battery or a 9-volt power supply.

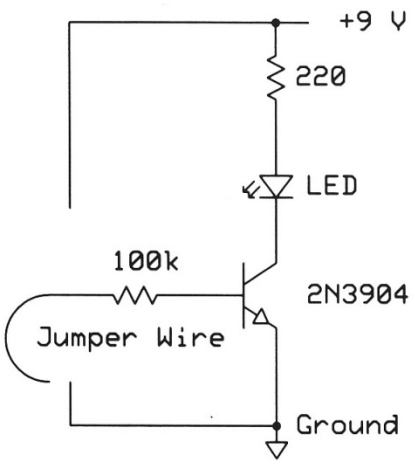


Figure 21.5.
A test circuit for a 2N3904 bipolar transistor used to switch an LED on or off. You will use the jumper wire to connect the 100-kohm input to ground or to +9 volts.

Step 2.

In this circuit, the 2N3904 transistor will operate as an on-off switch. When "turned on," the 2N3904 transistor will let current flow through the LED and resistor, into the transistor's collector, and out the emitter to ground. To turn on the transistor you supply a small current to its base contact. The transistor needs a current-limiting resistor in the base circuit.

Keep the free end of the jumper unconnected and turn on power. The LED should remain off because no current can flow into the 2N3904 base. Now connect the free end of the jumper to +9 volts. The LED should turn on. Move the free end of the jumper from +9 volts to ground. What happens to the LED?

The LED should turn off. Remove the free end of the jumper wire from ground and leave it unconnected. What happens to the LED now?

The LED remains off. A small current *must flow into the base* of an NPN transistor to let current pass through the LED, to the collector, through to the emitter, and then to ground. With the jumper disconnected, no base current can flow. Likewise, when you connect the jumper to ground, no current flows because no voltage difference exists between the base and the emitter.

The transistor's internal base-to-emitter junction has a forward-voltage drop comparable to the the forward-voltage drop of a small diode (but *not* a light-emitting diode that has a higher forward voltage drop due to its semiconductor materials). The 2N3904 data sheet lists a Base-Emitter Saturation Voltage ($V_{BE(sat)}$) as a minimum of 0.65 volts, and a maximum of 0.85 to 0.95 volts. So to control the current flow in a 2N3904, the base *voltage* must exceed this value. **Figure 21.6** shows this relationship in a plot of transistor-base voltage versus LED current measured with the circuit in **Figure 21.7**.

When a bipolar transistor goes into *saturation*, the forward voltage drop between the collector and the emitted drops to between about 0.2 and 0.3 volts. That small voltage drop makes the saturated 2N3904 act much like a true switch that passes current with almost no voltage drop. Find the $V_{CE(sat)}$ value in the On Characteristics section of a 2N3904 data sheet. This experiment discusses transistor saturation in detail later on.

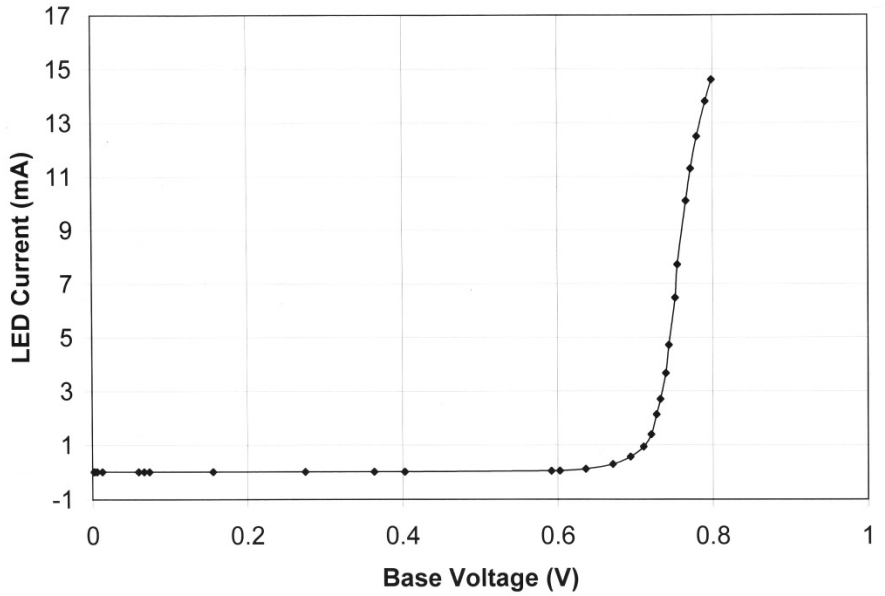


Figure 21.6.
This plot of base voltage against LED current for a 2N3904 transistor shows almost no current flows through the collector to the emitter until the base voltage increases to about 0.71 volts. So the base must have at least this voltage applied to start a substantial current flow.

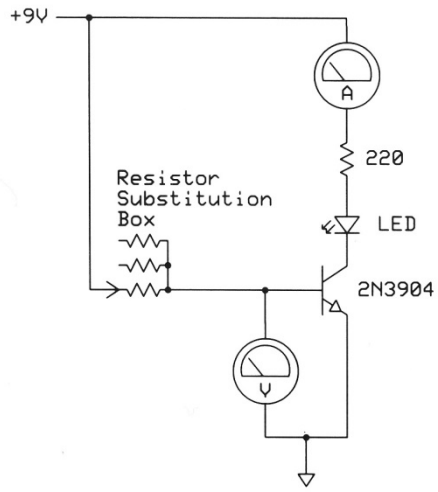


Figure 21.7.
The circuit used for base-voltage and LED-current measurements.

Step 3.

The diagram in **Figure 21.8** shows the current flow through an NPN transistor that turns on an LED. The 100,000-ohm resistor (100 kohms) limits the base current (I_B). I measured a 18.5 mA current into the collector (collector current, or I_C) in my circuit, and a base current (I_B), of 0.110 mA or 110 μA. You can see how a small base current controls a much higher current through the collector to the emitter.

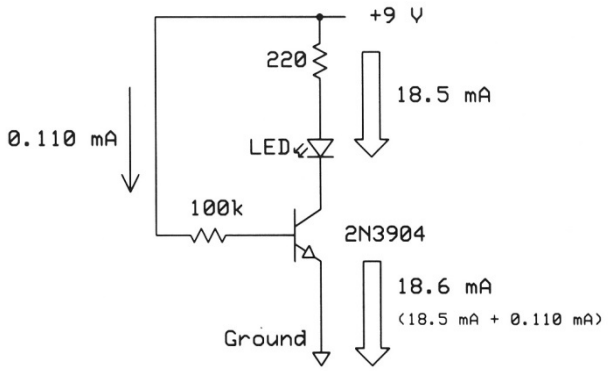


Figure 21.8.
A small base current into an NPN transistor causes a larger current to flow from the collector through the emitter to ground.

An ammeter connected between the transistor emitter and ground would measure about 18.6 mA because the current flowing into a circuit must equal the current flowing out: $I_E = I_C + I_B$. The small current into the transistor's base does not "magically" become a higher current that turns on the LED. The higher current comes from the +9-volt power supply. The transistor simply regulates the current flow, much like the fire-hydrant valve.

The ratio of the collector current to the base current (I_C / I_B) for my measurements equals 168. The ratio has no unit such as volts, amps, or ohms.

Step 4.

Turn off power to your circuit and substitute a 10,000-ohm (10 kohm) resistor (brown-black-orange) for the 100-kohm resistor in the base circuit shown in **Figure 21.8**. Turn on power and then measure the I_C and I_B values in your circuit. You do not need two meters. Measure one current and then measure the other

$I_C =$ $I_B =$

I measured $I_C = 34.2$ mA and $I_B = 0.935$ mA. Use your measurements to calculate the ratio of I_C to I_B. What did you find? Did this value equal the ratio I calculated earlier?

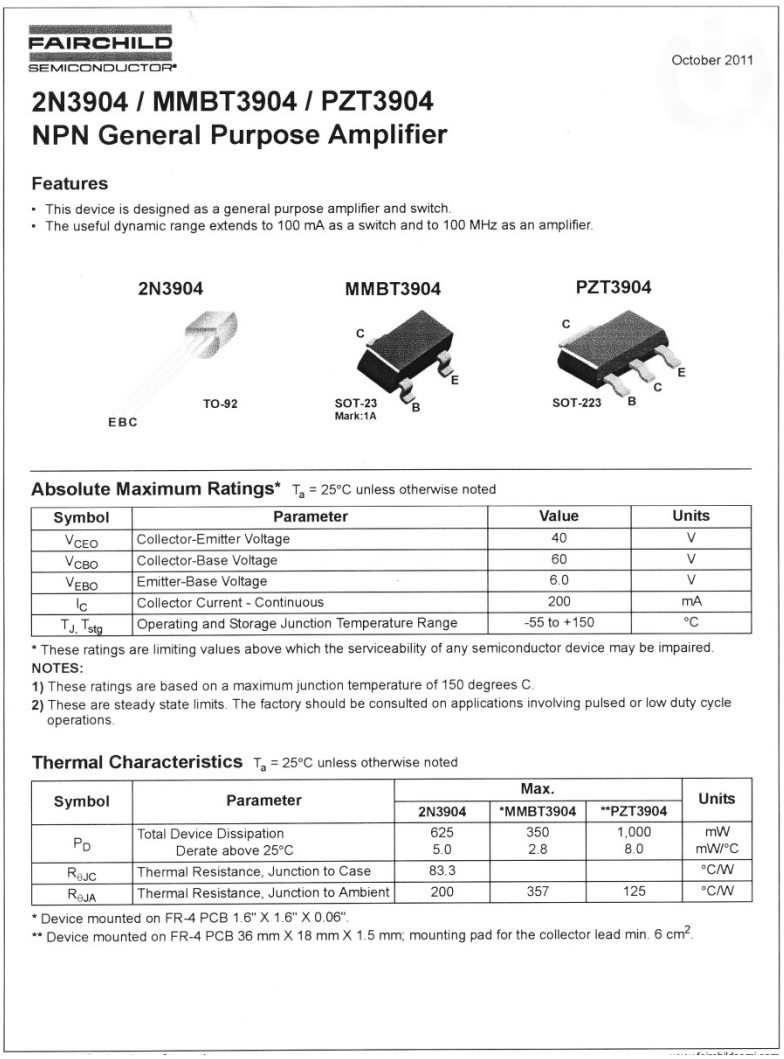
My ratio equalled 36. This value differs from the ratio (168) calculated earlier for my circuit with the 100-kohm base resistor. The I_C / I_B ratio for a transistor depends on the power-supply voltage and the collector current I_C. This experiment will shortly explain that ratio in more detail.

Leave your circuit (**Figure 21.8**) set up in the breadboard for use later in Step 6.

How to Read Transistor Data Sheets

Step 5.

Manufacturers publish data sheets that supply important information about transistor characteristics. In this step, you will learn about the key values that help you determine how to control an LED with a bipolar transistor. The diagram in **Figure 21.9** illustrates the first page of a 7-page data sheet from Fairchild Semiconductor for a 2N3904 transistor. You can find complete data sheets for many transistors on the Internet. Distributor Web sites also include links to data sheets for individual products.



FAIRCHILD SEMICONDUCTOR®

October 2011

2N3904 / MMBT3904 / PZT3904
NPN General Purpose Amplifier

Features

- This device is designed as a general purpose amplifier and switch.
- The useful dynamic range extends to 100 mA as a switch and to 100 MHz as an amplifier.

2N3904 MMBT3904 PZT3904

TO-92 EBC

SOT-23 Mark:1A C E B

SOT-223 C E C B

Absolute Maximum Ratings* T_a = 25°C unless otherwise noted

Symbol	Parameter	Value	Units
V_{CEO}	Collector-Emitter Voltage	40	V
V_{CBO}	Collector-Base Voltage	60	V
V_{EBO}	Emitter-Base Voltage	6.0	V
I_C	Collector Current - Continuous	200	mA
T_J, T_{stg}	Operating and Storage Junction Temperature Range	-55 to +150	°C

* These ratings are limiting values above which the serviceability of any semiconductor device may be impaired.

NOTES:

1) These ratings are based on a maximum junction temperature of 150 degrees C.

2) These are steady state limits. The factory should be consulted on applications involving pulsed or low duty cycle operations.

Thermal Characteristics T_a = 25°C unless otherwise noted

Symbol	Parameter	Max.			Units
		2N3904	*MMBT3904	**PZT3904	
P_D	Total Device Dissipation Derate above 25°C	625 5.0	350 2.8	1,000 8.0	mW mW/°C
$R_{\theta JC}$	Thermal Resistance, Junction to Case	83.3			°C/W
$R_{\theta JA}$	Thermal Resistance, Junction to Ambient	200	357	125	°C/W

* Device mounted on FR-4 PCB 1.6" X 1.6" X 0.06".

** Device mounted on FR-4 PCB 36 mm X 18 mm X 1.5 mm; mounting pad for the collector lead min. 6 cm^2.

Figure 21.9.
The first page of a transistor data sheet includes important information about maximum currents and voltages a transistor can handle.
Courtesy of Fairchild Semiconductor. Copyright 2014.

The 2N3904 data sheet includes a section, Absolute Maximum Ratings, duplicated in **Table 21.1**. The diagram in **Figure 21.10** includes these ratings superimposed on the symbols for a 2N3904 NPN transistor. It might help to create a diagram such as this for transistors you use often. I find graphical information easier to use than data in tables.

Table 21.1. Absolute maximum ratings for a 2N3904 transistor.

Symbol	Parameter	Value	Units
V_{CEO}	Collector-Emitter Voltage	40	Volts
V_{CBO}	Collector-Base Voltage	60	Volts
V_{EBO}	Emitter-Base Voltage	6.0	Volts
I_C	Collector Current - Continuous	200	Milliamps
T_J	Operating Temperature Range	-55 to +150	Degrees C

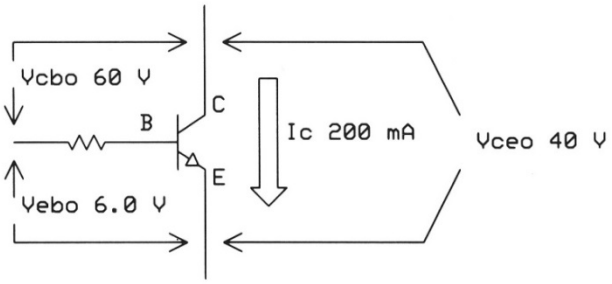


Figure 21.10.
Maximum ratings for a 2N3904 transistor shown graphically. The first two subscripts letters indicate a measurement between the noted pins. The lowercase "o" suffix means measurements made with one pin unconnected, or "open."

Assume the emitter of a 2N3904 transistor connects to ground. The maximum ratings indicate a circuit should not apply more than 40 volts to the collector. The transistor base should not have a voltage of more than 60 volts applied to it. And finally, a circuit should not try to pass more than 200 mA (Ic) through a 2N3904 transistor. (The Vebo specification relates to a reverse-bias condition, which you're unlikely to encounter. In essence, if the base has a connection to ground and the emitter voltage reaches +6 volts – note the positive sign – the base-emitter junction can break down. We call this value a reverse-breakdown voltage.)

Table 21.2 shows the Off and On Characteristics for a 2N3904 transistor. (The table also lists the test conditions used to provide the minimum and maximum quantities.) Within the Off Characteristics section you see some of the same information shown earlier in **Table 21.1**. The I_{BL} and I_{CEX} values indicate the maximum *leakage current* through a transistor when it does not otherwise pass current to ground from a "load" such as an LED. These leakage values do not play a role in determining how to use a transistor to control LEDs. When designers must control the power a circuit uses, they aim to decrease current leakage in semiconductors and they choose low-leakage components.

Table 21.2. On and off characteristics for a 2N3904 NPN transistor.

ON Characteristics (at 25°C)					
Symbol	**Parameter**	**Test Condition**	**Min.**	**Max.**	**Units**
hFE	DC Current Gain	I_C = 0.1 mA, I_{CE} = 1.0 I_C = 1.0 mA, I_{CE} = 1.0 I_C = 10 mA, I_{CE-} = 1.0 I_C = 50 mA, I_{CE} = 1.0 I_C = 100 mA, I_{CE} = 1.0	40 70 100 60 30	300	
$V_{CE(sat)}$	Collector-Emitter Saturation Voltage	I_C = 10 mA, I_B = 1.0 mA I_C = 50 mA, I_B = 5.0 mA		0.2 0.3	V V
$V_{BE(sat)}$	Base-Emitter Saturation Voltage	I_C = 10 mA, I_B = 1.0 mA I_C = 50 mA, I_B = 5.0 mA	0.65	0.85 0.95	V V
OFF Characteristics (at 25°C)					
Symbol	**Parameter**	**Test Condition**	**Min.**	**Max.**	**Units**
$V_{(BR)CEO}$	Collector-Emitter Breakdown Voltage	I_C = 1.0 mA, I_B = 0	40		V
$V_{(BR)CBO}$	Collector-Base Breakdown Voltage	I_C = 10 µA, I_E = 0	60		V
$V_{(BR)EBO}$	Emitter-Base Breakdown Voltage	I_C = 10 µA, I_C = 0	6.0		V
I_{BL}	Base-Cutoff Current	V_{CE} = 30 V, V_{BE} = 3.0 V		50	nA
I_{CEX}	Collector-Cutoff Current	V_{CE} = 30 V, V_{BE} = 3.0 V		50	nA

The On Characteristics list the abbreviation h_{FE}, or DC Current Gain. This value represents the ratio of I_C/I_B calculated earlier in Step 3. For the test conditions I_C = 10 mA and V_{CE} = 1.0 V, the the transistor has an h_{FE} value between 100 and 300. For the values I measured in Step 4, I_C = 18.5 mA and I_B = 0.110 mA, so the ratio equals 168, which seems reasonable, given the h_{FE} range from the data sheet.

Some information about transistors lists both h_{FE} and an h_{fe} values. Note the difference in the subscripts. The h_{FE} value represents characteristics for the transistor when used under a direct-current (DC) condition such LED control. The h_{fe} value represents alternating-current (AC) characteristics found in sound-amplifier circuits, for example. Engineers also use the term *beta*, represented by the lowercase Greek letter β, to represent the I_C/I_B ratio. For my measurements of I_C = 18.5 mA and I_B = 0.110 mA, you could say the 2N3904 exhibited a beta of 168.

The On characteristics also indicate a Collector-Emitter Saturation Voltage and a Base-Emitter Saturation Voltage. These values become important if a circuit causes a transistor to "saturate."

Given the specifications for a 2N3904 transistor, could it control four LEDs connected in series and through a current-limiting resistor as shown in **Figure 21.11**? Find the answer at the end of this experiment.

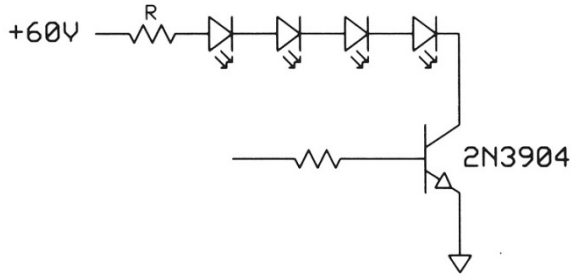


Figure 21.11.
Can this circuit work when you apply the proper current to the 2N3904 base?

Transistor Saturation

For a moment, think about a sponge that absorbs water poured on it. At some point, the sponge "saturates" and cannot hold any more water. The excess water gets wasted. A transistor goes into "saturation" when an increase in base current (I_B) does not cause any increase in collector current (I_C). If you continue to increase base current after a transistor saturates, you waste power and can cause a transistor to operate at a high temperature.

Step 6.

In this step you will measure collector current and base current for a 2N3904 transistor. Then you will create an *x-y* plot to see the effect of saturation. Instead of using two ammeters, a *cross-over switch* lets you first measure base current and then measure collector current with the same meter. The schematic diagram you will see shortly includes a *double-pole double-throw toggle switch* wired with cross-over connections. (If you have a lamp controlled by three wall switches, one of them, called a 4-way switch, includes a cross-over circuit.)

If you know how a double-pole double-throw toggle switch works, feel free to skip to Step 6. Otherwise, please continue here. To understand how a double-pole double-throw (DPDT) toggle switch works, separate the name into two parts; double-pole and double-throw. Double pole means a switch contains two poles – two electrically independent switches – operated by the same control, usually a small handle. Double-throw signifies the switch controls one circuit with the handle in the up position, and a second circuit in the down position. **Figure 21.12** shows a single-pole double-throw switch used to light either a red or a green LED.

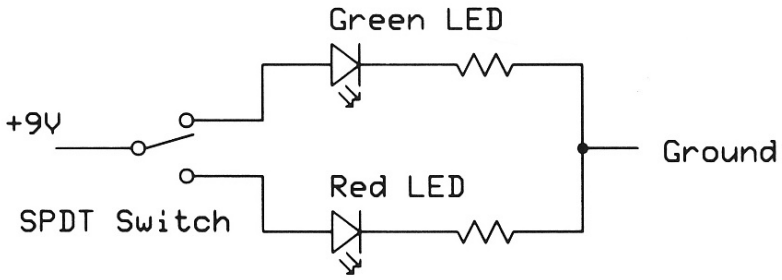


Figure 21.12.
Use of a single-pole double-throw (SPDT) switch to light one of two LEDs from the same 12-volt power supply.

A double-pole double-throw switch controls two separate circuits (**Figure 21.13**); in this case: 1) two LEDs from a 9-volt power supply, and 2) a buzzer and a bell from a 24-volt AC transformer. Although mechanically linked, the two poles keep the circuits separated from each other.

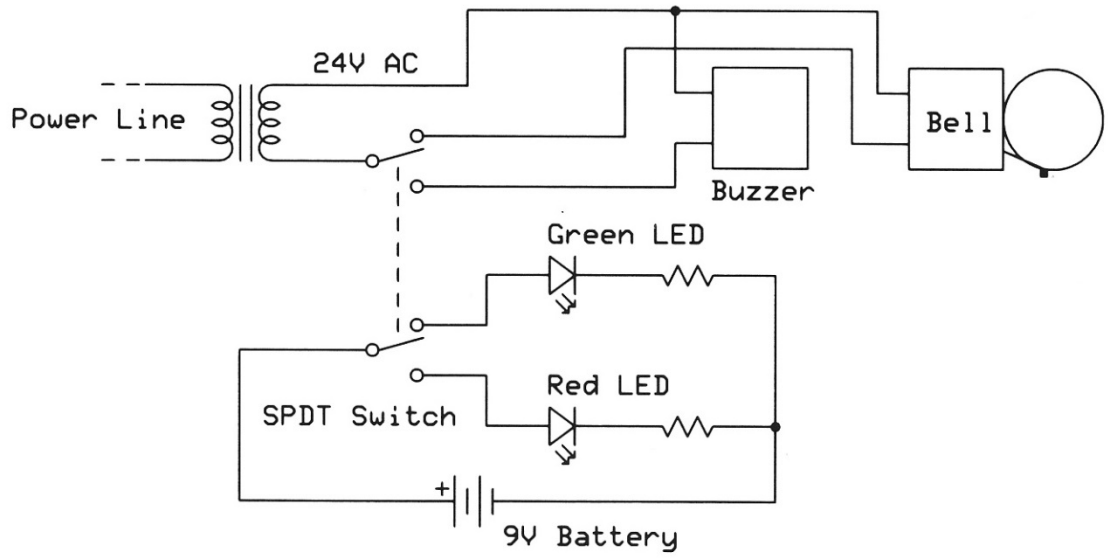


Figure 21.13.
A DPDT switch controls two independent circuits. In one position, the red LED and the buzzer turn on. In the other position, the green LED and the bell turn on. The dashed line represents a nonconductive mechanical linkage.

The image in **Figure 21.14** includes a wired DPDT toggle switch, two SPDT toggle switches, a large SPDT slide switch and two small SPST pushbuttons. Engineers and designers call the silver "bat-like" handle the toggle.

Figure 21.14.
A variety of switches with various electrical-contact configurations. Almost all double-throw toggle switches connect the middle contact in a column of three to the switch "wiper" represented by a line at an angle between two contacts. In some diagrams, the wiper might have an arrow on the end.

Step 7.

Wire your DPDT switch as shown in **Figure 21.15**. I highly recommend you use colored hookup wires to keep track of the connections. If you have only one or two colors, label the wires with the names shown in the figure.

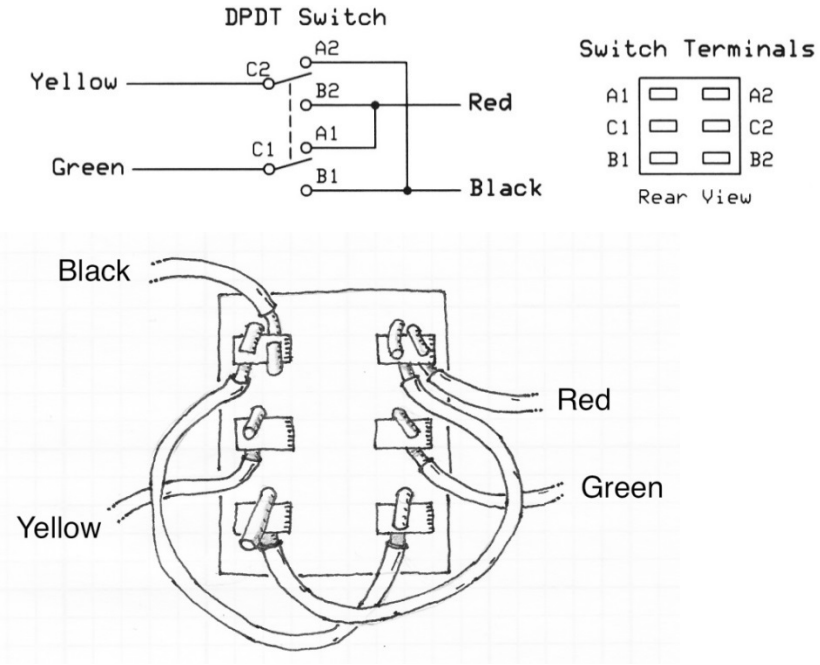


Figure 21.15.
The schematic diagram for a DPDT cross-over switch and the rear-terminal labels (top). A diagram of a wired switch (bottom). Solder these connections. A good solder "joint" appears silver and reflective. If you use too low a temperature or do not provide heat long enough you can get a cold-solder joint. This type of defect can cause intermittent problems that challenge your troubleshooting skills.

Step 8.

Now back to transistor current measurements. Construct the circuit shown in **Figure 21.16**, which includes an unconnected resistor, R. The 18 resistance values – one per set of measurements – will connect to the circuit in this position. For now, connect a 15 kohm resistor (brown-green-orange) between points A and B.

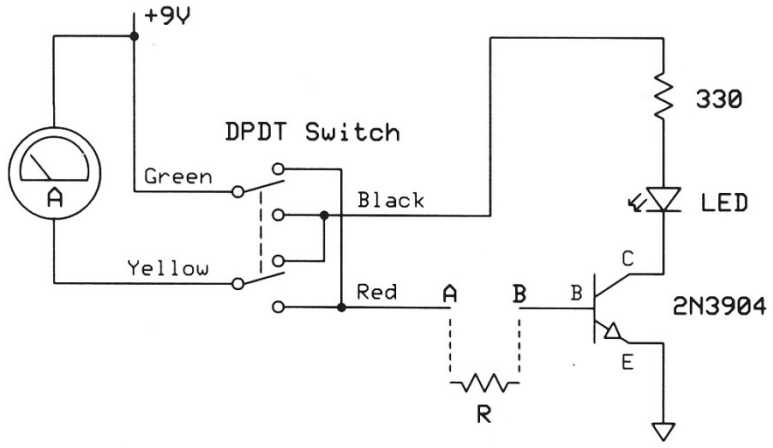


Figure 21.16.
Circuit diagram for measurement of the base and collector currents in a 2N3904 transistor that controls an LED.

Turn on power to your circuit. Depending on the type of LED you used, the ammeter should measure about 20 mA in the Collector position (toggle up in **Figure 21.16**), and about 0.5 mA in the Base position (toggle down). Label the positions to help you remember which current measurement the ammeter shows. If in doubt, the collector position shows the higher current on the ammeter. When you know the switch positions, remove the 15 kohm resistor. You may leave the power on.

Table 21.3 lists 18 resistances and includes spaces for the base and collector currents (I_B and I_C) you will measure. If you have fewer or different resistance values, go ahead and substitute them for the ones in **Table 21.3**. (Write your resistor values in the table!)The resistance range should still go from 1.0 Mohms down to about 1500 ohms. I suggest you buy a resistor-substitution-box kit, assemble it and use it instead of individual resistors. I found an Elenco Kit, Model RS-400 with 24 resistances that would work well in this step. All Spectrum Electronics sells this kit for about $US15. A resistor-substitution box provides a helpful tool for your lab.

If your ammeter does not automatically switch measurement ranges, do this by hand. After you measure a small base current, adjust the meter's current range accordingly. I have a handheld Wavetek DMM with current ranges of 200 μA, 2 mA, and 100 mA, and I switched the range setting by hand.

Table 21.3. Base- and collector-current measurements for 18 resistances.

Resistance (ohms)			Collector Current (I_C) milliamperes (mA)	Base Current (I_B) milliamperes (mA)
Suggested	Color Code	Your Resistances		
1M	BN-BK-GN			
680k	BU-GY-YL			
470k	YL-VI-YL			
330k	OR-OR-YL			
220k	RD-RD-YL			
150k	BN-GN-YL			
100k	BN-BK-YL			
68k	BU-GY-OR			

47k	YL-VI-OR			
33k	OR-OR-OR			
22k	RD-RD-OR			
15k	RD-GN-OR			
10k	BN-BK-OR			
6800	BU-GY-RD			
4700	YL-VI-RD			
3300	OR-OR-RD			
2200	RD-RD-RD			
1500	RD-GN-RD			
Black BK=0, Brown BN=1, Red RD=2, Orange OR=3, Yellow YL=4, Green GN=5, Blue BU=6, Violet VI=7, Gray GY=8, White WH=9				

Place a 1.0 Mohm resistor in the circuit between points A and B. Measure and record the collector and base currents for this resistance and enter your values in **Table 21.3**. Remove that resistor and go to the next lowest value, and so on until you have current measurements for all resistors.

Step 9.

After you have the current measurements recorded, plot then on graph paper or in an Excel spreadsheet. Place the I_C values in one column and the I_B values in another. Then use the the I_B values for the *x* axis and the I_C values for the *y* axis. **Figure 21.17** shows a plot of the currents I measured. As the base current starts to increase, the collector current also increases. At a base current of about 0.5 mA, the collector current starts to reach a plateau. This flat horizontal portion of the I_C vs. I_B line indicates the transistor has reached saturation. It now carries the maximum amount of current as determined by the LED-and-resistor circuit and the 9-volt power source. Increasing the base current beyond this point has no effect on the collector current, but it might cause the transistor to overheat. In its saturated state, we consider a transistor "full on," which means external components limit the collector-to-emitter current and small increases or decreases in the base current do not affect current flow.

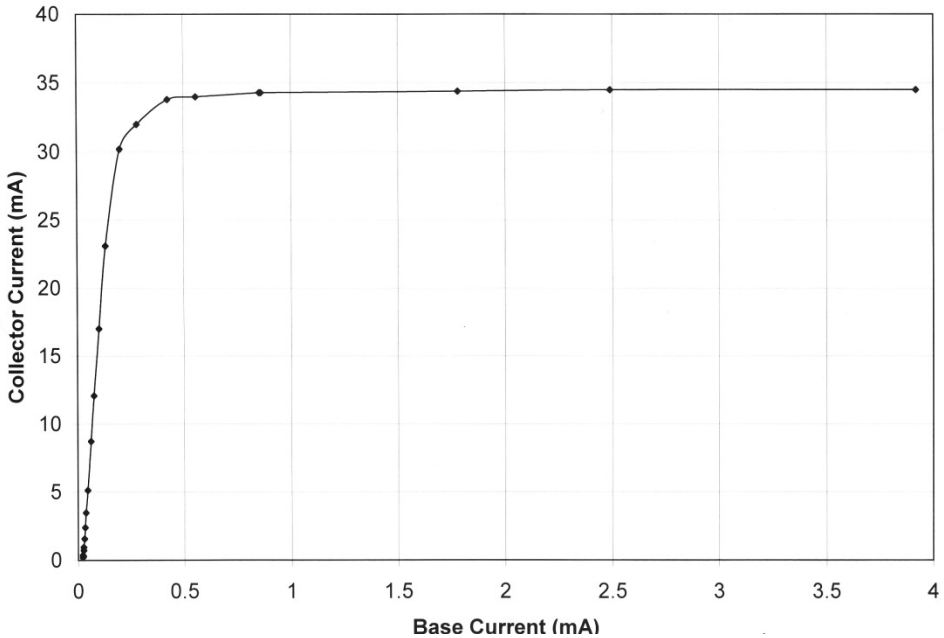


Figure 21.17.
A plot of base current vs. collector current for a 2N3904 transistor shows when the transistor saturates, indicated by the flat horizontal line that starts with a base current of about 0.5 mA.

Does your plot of the current measurements look similar to mine? It should. If it does not, check your circuit and the DPDT switch connections. Many circuits that control LEDs, relays, buzzers, serve motors, and other devices force a transistor into saturation to ensure the attached device receives the full current required to power it.

Transistor-Saturation Calculations

How can you determine when a transistor will saturate without making a base-current versus collector-current graph for every circuit you build? To turn an LED fully on (saturated) or fully off for a given circuit, follow the steps below:

1. Determine the current through the load – LEDs, a small relay, and so on. **Figure 21.18** shows a simple circuit that operates four LEDs in parallel at 30 mA each, or 120 mA total. The 2N3904 data sheet shows a 200-mA maximum for the collector current, I_C, and a maximum 40-volt collector-to-emitter voltage, so the transistor can control the four LEDs.

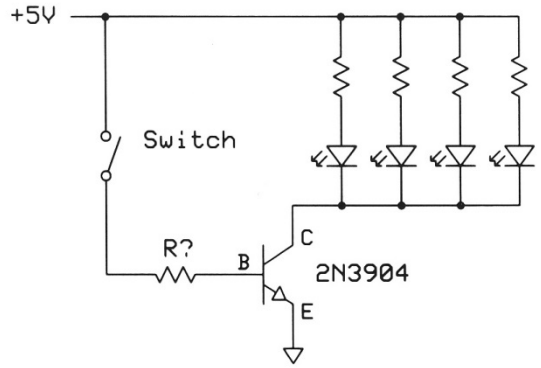


Figure 21.18.
A circuit with four LEDs controlled by a 2N3409 NPN transistor provides an example for calculation of a base current to put the transistor into saturation.

2. Calculate the needed base current (I_B) that lets the 120-mA load current (I_C) flow through the transistor. Within the On Characteristics for a 2N3904 transistor as given earlier in **Table 21.2**, find the *minimum* DC Current Gain (h_{FE}) of 30 for a 100-mA collector current. You know that:

   $h_{FE} = I_C / I_B = 30$

   Rearrange this formula to:

   $I_C / 30 = I_B$

   Then, for I_C = 120 mA the relationship becomes: $120\ mA / 30 = I_B = 4\ mA$. So the circuit needs a 4-mA current into the base. Assume this current comes from the +5-volt source that also powers the LEDs.

3. Determine the needed base resistance. The 2N3904 has a forward voltage drop of 0.7 volts from the base to the emitter. This voltage drop occurs in the base-to-emitter current path. This V_{BE} voltage drop figures into the base resistance calculation the same way an LED's forward-voltage drop figures into the calculation of a series current-limiting resistance. So first find the voltage across the base resistor (V_R):

   $V_R + V_{BE} = 5\ volts$

and by rearranging:

$V_R = 5\ volts - V_{BE}$

$V_R = 5\ volts - 0.7\ volts = 4.3\ volts$

Use Ohm's Law to calculate the base resistance for I_B = 4 mA and V_R = 4.3 volts:

$I = E/R$ or $R = V_R / I_B$, so $R = 4.3\ volts / 0.004\ A = 1075\ ohms$

Although a 1000-ohm resistor will work in the LED-control circuit, I recommend using a somewhat smaller value. A *slightly* higher current will ensure the transistor turns on completely. Use the three steps above to determine the base resistance for a given on-off circuit you create when you want to drive a transistor into saturation.

4. As a final step, I checked the base current (I_B) and the collector current (I_C) for a 2N3904 NPN transistor and four green LEDs, each of which has a 220-ohm resistor to limit current to 30 mA. **Table 21.4** contains my measurements for seven base resistances and thus, seven base currents.

Table 21.4. Base-current measurements for a 2N3904 transistor with a 120-mA load.

Resistance (ohms)	I_B (mA)	I_C (mA)
4700	1.58	92.1
3300	2.20	102
2200	3.47	113
1500	5.26	115
1000	6.92	116
680	10.5	116
470	14.5	117

Even without plotting the results you can see the current starts to level off with a base resistance of 1500 ohms. The calculated value came close but didn't equal 1075 ohms. Do you know why? First, the data sheet lists a *minimum* h_{FE} value, so you can expect a higher h_{FE} in a circuit. Second, the data-sheet test conditions of I_C = 100 mA and V_{CE} = 1.0 volts don't match our circuit conditions. When you design a circuit, I recommend you run a few tests to determine operation characteristics.

Engineers call the NPN-type transistor shown so far as a common-emitter configuration because the emitter provides the common reference for both the emitter and the collector. You also might hear this configuration called a "low-side switch" because it connects a device such as an LED already at a high potential to a lower voltage, often ground.

PNP Transistors

The bipolar-junction-transistor family includes PNP devices that have a symbol shown earlier in **Figure 21.3**. A PNP transistor – like its NPN sibling – has a collector, emitter, and base. But a PNP transistor works in a different way: all current flows into a PNP transistor through the *emitter*, and current flows out through the *collector* and the *base*. The collector provides the current to drive a load. The base current that flows *out of* the PNP transistor governs the amount of emitter-to-collector current. So instead of supplying, or sourcing, current for the base, we must sink current from it in a PNP transistor.

To control an NPN transistor, we raise the base voltage and let current flow into the base and out the emitter. On the other hand, to control a PNP transistor, we lower the base to a voltage below the emitter voltage and current flows out of the base and out the collector. The 2N3906 PNP transistor provides a good example to work with. And the maximum ratings for the 2N3906 NPN transistor looked similar to those for a 2N3904, except the 2N3906 voltage and current maximums have minus signs (**Table 21.5**). Note, though, the 2N3906 has a lower Collector-Base Voltage, and a slightly lower Emitter-Base Voltage.

Table 21.5. Absolute maximum ratings for a 2N3906 transistor.

Symbol	Parameter	Value	Units
V_{CEO}	Collector-Emitter Voltage	-40	Volts
V_{CBO}	Collector-Base Voltage	-40	Volts
V_{EBO}	Emitter-Base Voltage	-5.0	Volts
I_C	Collector Current - Continuous	-200	Milliamps
T_J	Operating Temperature Range	-55 to +150	Degrees C

At first glance the negative values look odd, but only because manufacturers continue to use the PNP transistor's *emitter* lead as the reference for measurements. Take the V_{CEO} rating of -40 volts, for example. This simply means a circuit can have a collector voltage 40 volts *below* the voltage at the PNP transistor's emitter. The negative specification *does not* imply you must use negative voltages with PNP transistors. As you'll see, they work with positive voltages referenced to ground.

In many circuits, a PNP transistor serves as a switch that turns a device on or off. But instead of connecting the device to ground, a PNP transistor connects it to power as shown in **Figure 21.19**. We often refer to this circuit as a high-side switch because it connects a device already at a low potential (often ground) to a higher voltage that supplies power.

Motor vehicles use many high-side switches because the metal frame serves as a common ground. The minus terminal on the battery connects to the frame. A brake-light, for example, uses a high-side switch and needs only one wire to supply power. A low-side switch would require two wires, one for power and another wire to go back to the brake switch that would complete the path to ground.

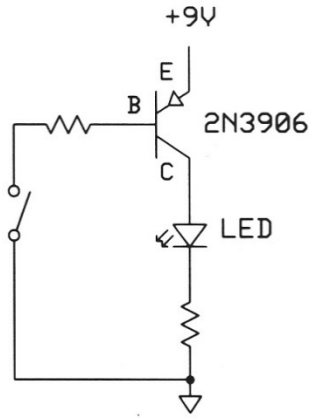


Figure 21.19.
A circuit that uses a 2N3906 PNP transistor to control an LED. When the switch closes, a small current flows through the base resistor to ground and the transistor passes current from the emitter to the collector and through the LED to ground.

You can calculate the current-limiting resistance for the base of a PNP transistor with the same formulas used in the 2N3904 examples. Assume you have a circuit with four LEDs that together draw 120 mA from a

2N3906 transistor that connects to a 9-volt power supply (**Figure 21.20**). We need a base current that will force the transistor into saturation so the LEDs turn completely on.

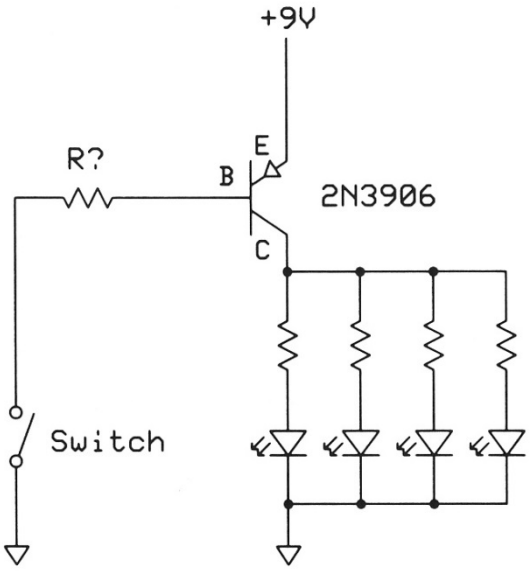


Figure 21.20.
Control of four LEDs by a 2N3906 transistor used as a high-side switch. Power comes from a 9-volt supply or battery.

You learned about base current and base series resistance in the earlier section, Transistor Saturation Calculations, so I'll repeat the calculations next, but without the longer explanations:

Look in a 2N3906 data sheet for the DC current gain (h_{FE}) for a -100-mA collector current, I_C. Although the LED circuit draw -120 mA, the -100-mA current specification gets close enough for our calculation:

$$h_{FE} = I_C / I_B = 30 \text{ and } I_C / 30 = I_B, \quad \text{so} \quad -120\,mA / 30 = -4\,mA$$

For the LEDs' -120-mA load, I_B equals -4 mA. The 2N3906 data sheet shows a forward voltage drop of -0.95 for a -50-mA collector current, so use that value to calculate the voltage (V_R) needed across the external base resistor for a -4 mA base current. Remember, all measurements use the +9-volt power supply as the reference for measurements, so voltages and currents have negative (minus) values.

$$V_R + V_{BE} = -9\,volts \text{ and thus } V_R = -9\,volts - V_{BE}$$

$$V_R = -9\,volts - (-0.95\,volts) = -9\,volts + 0.95\,volts = -8.05\,volts$$

Next:

$$R = V_R / I_B, \quad \text{so} \quad R = -8.05\,volts / -0.004\,A = 2013\,ohms$$

You could use a 2000-ohm resistor, but an 1800- or even a 1600-ohm resistor will work well.You could test several resistances above and below 2000 ohms to find one that puts the 2N3906 transistor into saturation but doesn't cause the transistor to waste power. I measured I_B and I_C values for the 2N3906 circuit and found the LED current from the collector leveled off at 4.0 to 4.2 mA. Those currents correspond to a base resistance of 2000 and 1900, respectively.

Some people recommend increasing the load current (I_C) in the calculations by a factor of 1.2 to 1.3 to decrease the calculated base resistance and ensure resistor saturation (Ref. 1). I prefer to use the true load current and make measurements described above to determine a resistance that doesn't waste power.

Transistor Pullup Resistors

Many circuits that use a PNP transistor include a high-resistance pull-up resistor, as shown in **Figure 21.21.** With the switch in the open position, the base pin has no connection to ground and the pull-up resistor ensures the base "sees" the +9 volts, which keeps the transistor turned off. When the switch closes, a small amount of current flows from +9 volts through the 10-kohm resistor. Current also flows from the transistor base through the switch to ground. The 2000-ohm resistor limits the base current.

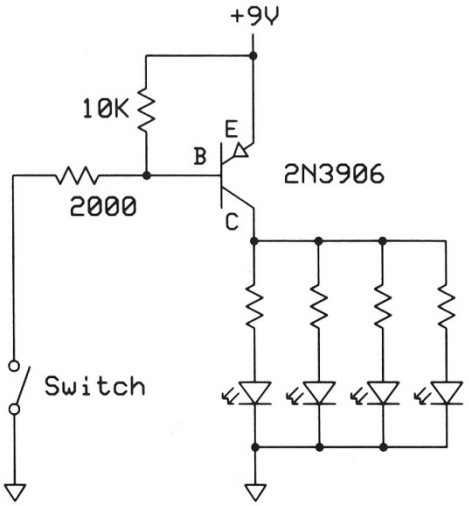


Figure 21.21.
Use of a high-resistance ensures the base gets "pulled up" to a voltage sufficient to ensure no current flows to LEDs or other device when the base disconnects from ground.

When you use a logic output from a gate or MCU to control a PNP transistor, a logic-0 output lets current flow out of the transistor base to ground. A logic-1 output should turn the PNP transistor off. But unless the output has a logic-1 voltage very close to the transistor's emitter voltage, some current might still flow out of the base. Although a device driven by the transistor might not turn on completely, it could still conduct a small current and waste power. The pull-up resistor ensures a high-enough voltage on the base for a logic-1 signal to keep current from flowing out of the base. On occasion I have seen pull-down resistors used with NPN transistors, but recommend using one only if the controlling device cannot produce a good enough ground connection to turn the NPN transistor fully off.

Step-by-Step Transistor Evaluations

Before you use a bipolar junction transistor you must answer several questions:

1. Can the transistor handle the voltage I need to apply across it? This amounts to the voltage applied by a circuit between the collector and the emitter when you have the transistor turned off.

2. Can the transistor handle the amount of current you need to pass through it? The emitter current always equals the base current plus the collector current, so use that specification (I_{CE}) for your evaluation.

3. Given the amount of current, does the transistor need a heat sink? Transistor data sheets and heat-sink manufacturers' application notes can help you. A transistor's capability to dissipate power decreases as the device temperature increases. Transistors in metal packages such as the TO-3 or TO-220 should have heat sinks. Datasheets often include a power-derating graph you should review. Always aim to keep transistors as cool as possible.

4. Can the circuit that you want to control a transistor deliver the needed base current? A 74HCT00 quad NAND-gate IC provides outputs that can sink or source 4 mA. At this current, a logic-1 provides a minimum output between 3.7 and 3.84 volts, according to a Diodes Incorporated data sheet. The logic-0 output has a maximum-voltage between 0.33 and 0.40 volts. Can you control the base of your transistor with those outputs?

Most common bipolar transistors cost only a few cents, so if you want to ensure one will work in a circuit, set up the circuit in a breadboard and see what happens. Always have a data sheet handy as a ready reference.

Conclusion

You still have much to learn about transistors. This experiment serves as an introduction. The Internet provides quick access to articles, designs, datasheets, application notes, tutorials, and so on. Use this information with care. Not all circuits work the way a designer expects. Perhaps they got lucky with a capacitor or resistor and their circuit worked. If yours doesn't work, don't immediately assume you're at fault. Find other references to double check information.

In the next two experiments, you will learn about field-effect transistors (FETs) and how to use them as switches. FETs can carry high currents, so designers use them to control large electromechanical devices.

References

1. Morris, Dan, "Using Transistors as Switches," http://techhouse.brown.edu/~dmorris/projects/tutorials/transistor.switches.pdf.

Schmartboard surface-mount-technology (SMT) adapters for integrated circuits and other components: http://www.schmartboard.com/.

"The ARRL Handbook for Radio Communications," 2009 ed., American Radio Relay League, Newington, CT. ISBN: 978-0-87259-139-4. Excellent discussions of transistors and circuits. The ARRL sells the latest annual edition on its Web site: http://www.arrl.org/shop/What-s-New/

Answers

Experiment 21, Step 5:

For the circuit shown in **Figure 21.9**, with the transistor in the off state, the voltage at the collector would equal 60 volts. The emitter connects to ground, so the 60-volt value exceeds the maximum collector-emitter voltage; 40 volts. You should not use this circuit to control the LEDs. It could ruin the transistor.

Experiment No. 22 – An Introduction to MOSFETs and LED Control

Abstract

Bipolar junction transistors such as the 2N3904 and 2N3906 introduced in the previous experiment, can control LEDs and other devices. But they carry relatively small currents, usually a maximum of a few tens or hundreds of milliamperes. In addition, bipolar transistors require a continuous base current. The family of metal-oxide-semiconductor field-effect transistors (MOSFETs) requires almost no "base" current and can carry tens of amperes. After you learn about MOSFETs and how to use them, your circuits can control high-current and high-voltage devices such as relays, motors, solenoids, and bright LEDs. This experiment provides less hands-on lab work than others, but it covers important concepts you will need to understand for Experiment 23.

Keywords

metal-oxide-semiconductor field-effect transistor, MOSFET, impedance, gate voltage, pullup and pull-down resistors, gate, source, drain, N-channel, P-channel, threshold

Requirements

(1) - Solderless breadboard
(1) - 12-volt power supply
(1) - Voltmeter or volt-ohm-milliammeter (VOM)
(1) - IRF630 N-channel MOSFET
(1) - IRF9520 P-channel MOSFET
(10) - LEDs, any color
(10) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(1) - 1000-ohm, 1/4-watt resistors, 5% (brown-black-red). Optional
(1) - 1000-ohm trimmer resistor. Optional
(1) - 10-kohm, 1/4-watt resistors, 5% (brown-black-orange)
(1) - 470-kohm, 1/4-watt resistors, 5% (yellow-violet-yellow)
(1) - 500-kohm trimmer resistor

Introduction

You just learned about NPN and PNP bipolar junction transistors and how they control current flow in a circuit. A small current to or from the base terminal on a bipolar transistor will control a larger current that passes through the emitter and collector terminals. Metal-oxide-semiconductor field-effect transistors, or MOSFETs, rely on a *control voltage* rather than a current.

Bipolar transistors give us three connections: base, collected, and emitter. MOSFETs, pronounced "moss-fets," also have three connections called *gate*, *source*, and *drain*. A *voltage* applied to a MOSFET gate controls how much electricity flows between the source and the drain terminals. In its simplest form, a MOSFET, includes a thin layer of insulating material – the metal oxide – that separates and electrically isolates the gate from the other semiconductor materials. No current flows from the gate to the drain or to the source. The simplified drawing in **Figure 22.1** shows a side view of a slice through a MOSFET to illustrate the placement of the insulating metal-oxide layer. In practice a few tens of nanoamperes (nA) flow to or from the metal gate, but we consider that an insignificant amount. Engineers would say the gate has a high *impedance* because it draws so little current. (The topic of impedance goes beyond the scope of this book. Find good examples and more information in the latest version of "The ARRL Handbook for Radio Communications," from the American Radio Relay League at: www.arrl.org.)

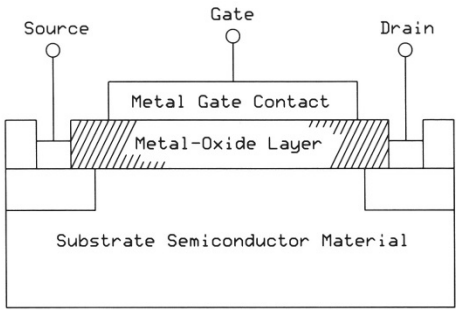


Figure 22.1.
Simplified cut-away side view of a MOSFET. In many MOSFETs, the substrate semiconductor material connects internally to the source pin or contact. (Drawing not to scale.)

Like their bipolar NPN and PNP cousins, MOSFETs fall into two categories, N-channel and P-channel. These designations describe how each type of MOSFET operates. **Figure 22.2** shows the symbols used for an N-channel and a P-channel MOSFET. In both cases, the small arrow indicates an internal connection of the source terminal to the semiconductor substrate. The arrow in a MOSFET symbol can help you identify a P-channel or an N-channel MOSFET in a circuit diagram.

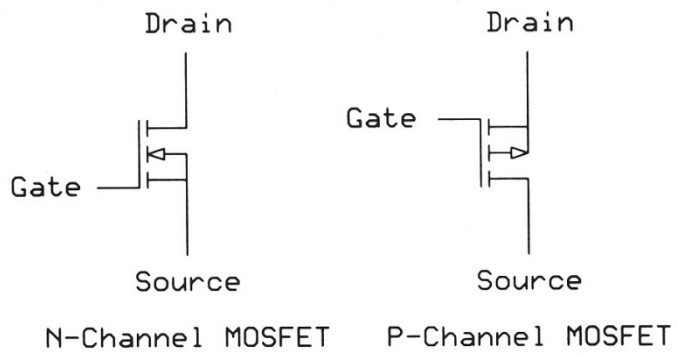


Figure 22.2.
Schematic-diagram symbols for an N-channel and a P-channel MOSFET.

N-Channel and P-Channel MOSFETs and How They Work

Figure 22.3 provides a comparison of an NPN bipolar transistor with an N-channel MOSFET and a PNP transistor with a P-channel MOSFET. An NPN transistor such as a 2N3904 requires a *control current* to regulate current flow from the collector to the emitter. The current flows into the base connection and out through the emitter to ground. An IRF630 N-channel MOSFET needs a *control voltage* to regulate the flow of electricity from the drain to the source. You can make a similar comparison of the 2N3906 PNP transistor and the P-channel IRF9520 MOSFET. Note the absence of a current-limiting resistor on the gate terminal of the MOSFETs. You will learn more about this difference in Experiment 23.

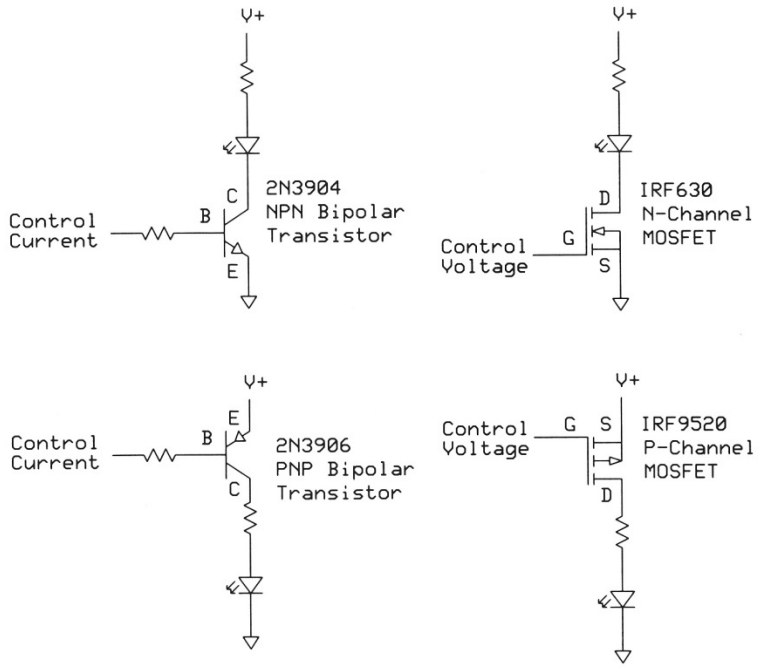


Figure 22.3.
Comparison of the control requirements for an NPN bipolar transistor and an N-channel MOSFET (top), and for a PNP transistor and a P-channel MOSFET (bottom).

When a circuit applies a voltage to a MOSFET gate terminal, the resistance between the drain and source terminals changes. This resistance can vary from almost infinite (open circuit) down to fractions of an ohm (closed circuit). The extremely low resistances available from MOSFETs makes them excellent switches for high-current devices. Circuits such as audio amplifiers use MOSFETs as voltage amplifiers, however the discussions and steps that follow use them solely as on-off switches.

An N-channel MOSFET can serve as low-side switch as shown in **Figure 22.3** and **Figure 22.4**. The MOSFET gives the LED a low-resistance path to ground. In this example, a gate voltage (V_{GS}) greater than the source voltage by a specified amount causes the LED to turn on. Data sheets list that amount as the MOSFET gate-to-source *threshold voltage range,* $V_{GS(th)}$ needed at the gate to start current flow through the MOSFET. You'll see the effect of gate voltage shortly.

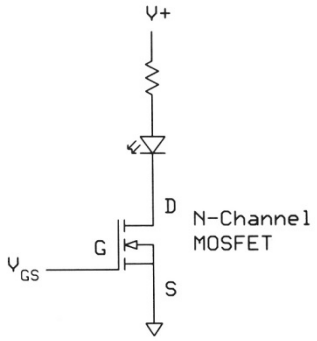


Figure 22.4.
An LED-control circuit uses an N-channel MOSFET as a low-side switch. To lower the resistance between the drain and source, the gate voltage (V_{GS}) must exceed the source voltage by an amount specified in the MOSFET data sheet. The drain must have a voltage (V_D) above that applied to the source (V_S). In summary, $V_D >> V_S$ and $V_{GS} > V_S$.

Table 22.1 shows important characteristics of an IRF630 N-channel MOSFET. You will find more information in a complete IRF630 data sheet available on the Internet (Ref. 1). The upper portion of the table provides the absolute maximum ratings for the MOSFET. This information specifies that the voltage across the drain and source cannot exceed 200 volts without damaging the transistor. Likewise, the voltage between the gate and the source cannot exceed ±20 volts. All measurements for N-channel MOSFETs use the *source voltage* as the reference. Depending on the temperature of the transistor case (T_C) you can use the IRF630 to control between 5.7 and 9.0 amperes. The MOSFET can handle short current pulses as high as 36 amperes.

Table 22.1. Select parameters from an IRF630 MOSFET data sheet.

Parameter	Symbol	Test Conditions	Limit	Unit
Drain-to-Source Voltage	V_{DS}		200	V
Gate-to-Source Voltage	V_{GS}		±20	V
Continuous Drain Current	I_D	TC = 25°C, V_{GS} at 10V	9.0	A
		T_C = 100°C, V_{GS} at 10V	5.7	A
Pulsed Drain Current	I_{DM}		36	A
Maximum Power Dissipation	P_D		74	W
Operating Temperature Range	T_J, T_{stg}		-55 to +150	°C

Parameter	Symbol	Test Conditions	Min	Typ	Max	Units
Drain-to Source Breakdown Voltage	V_{DS}	V_{GS} = 0V, I_D = 250 µA	200	–	–	V
Gate-to-Source Threshold Voltage	$V_{GS(th)}$	V_{DS} = V_{GS}, I_D = 250 µA	2.0		4.0	V
Gate-to-Source Leakage	I_{GSS}	V_{GS} = ±20			±100	nA
Zero Gate-Voltage Drain Current	I_{DSS}	V_{DS} = 200V, V_{GS} = 0V			25	µA
		V_{DS} = 160V, V_{GS} = 0V, T_J = 125°C			250	
Drain-to-Source On-State Resistance	$R_{DS(on)}$	V_{GS} = 10V, I_D = 5.4A			0.40	Ohm
Turn-On Delay Time	$T_{d(on)}$	V_{DD} = 100V, I_D = 5.9A, R_G =12Ω, R_D = 16Ω		9.4		nsec
Turn-Off Delay Time	$T_{d(off)}$			39		nsec

The lower portion of **Table 22.1** includes information from tests that operate an IRF630 MOSFET under standard conditions. This information shows the gate input has a threshold voltage ($V_{GS(th)}$) between 2.0 and 4.0 volts. So drain-to-source current will flow only when the gate voltage exceeds the source voltage by 2.0 to 4.0 volts. We call this the MOSFET's *cutoff state, off,* or *sub-threshold mode*. Let's stay with "off." In essence, when off the MOSFET looks like an open, nonconducting switch. (A small Zero Gate-Voltage Drain Current, I_{DSS}, does flow due to the inherent behavior of semiconductor materials. Find the I_{DSS} specification for an IRF630 MOSFET in **Table 22.1**.)

Bipolar and MOSFET Control Comparison

You might recall that to turn on a bipolar NPN transistor the base voltage must exceed the transistor's base-to-emitter junction voltage; usually about 0.65 to 0.7 volts. Likewise, MOSFETs have a similar requirement, a gate-source threshold voltage, $V_{GS(th)}$ mentioned briefly above. **Table 22.1** and the data sheet for an IRF630 N-channel MOSFET specify this value, which will help you create useful MOSFET circuits.

In the circuit shown in **Figure 22.5**, the gate voltage – when referenced to the source voltage – must exceed the 2.0-volt threshold potential and might need to reach 4.0 volts to turn the MOSFET on (closed switch). The graph in **Figure 22.6** illustrates the relationship I measured between the gate-source voltage, V_{GS}, and the

current into the MOSFET drain from the 10 LEDs. A very small current might flow through the drain and source for gate voltages between 2.0 and about 3.2 volts, but I didn't have a way to measure it.

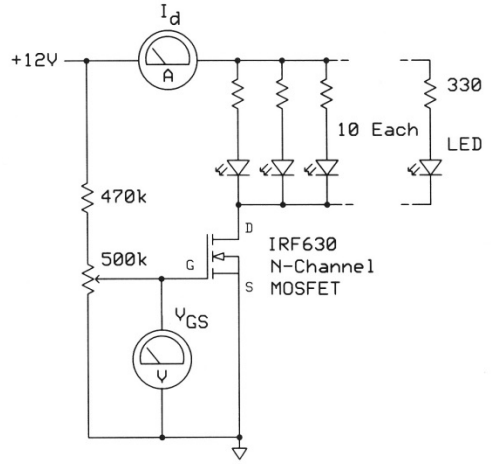


Figure 22.5.
The lab setup used to make the measurements graphed in **Figure 22.5**. The 10 LEDs draw about 300 mA from the 12-volt power supply.

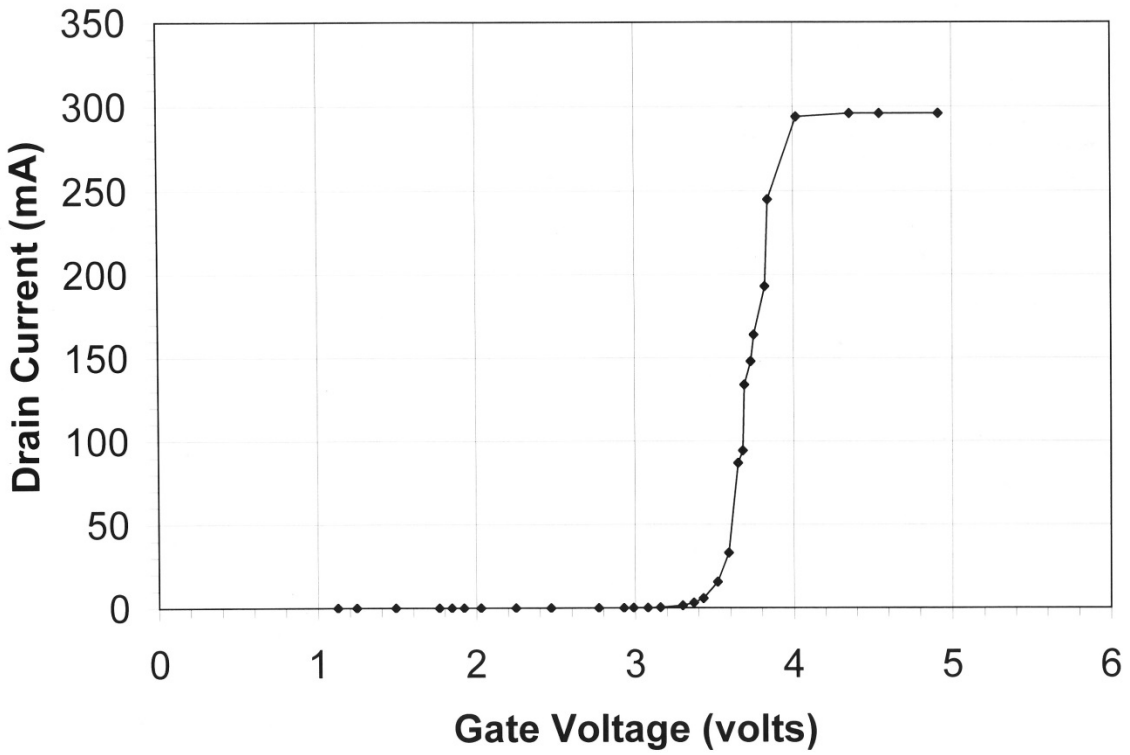


Figure 22.6.
A plot of gate-source voltage (V_{GS}) on the *x* axis and drain current (I_D) on the *y* axis shows a rapid increase of MOSFET drain current when the gate voltage increases from about 3.2 to 4.0 volts.

In this test I used an IRF630 N-channel MOSFET. The parallel circuit of 10 LEDs and 330-ohm series resistors provided a 300-mA load (**Figure 22.5**). When I applied a 3.2-volt signal to the IRF630 gate input, the MOSFET started to conduct enough current to measure accurately. As the gate voltage increased, so did the current through the LEDs. At about 4.0 volts on the gate, the IRF630 passed the maximum current. Increasing

the gate voltage above 4.0 volts had no effect because the *resistors* limited the maximum current through the LED circuit.

My 30 measurements provide an approximation of the gate-voltage and drain-current relationship, but they look somewhat "rough" when graphed. To better show current flow at more voltages I connected the gate input to a 0-to-6-volt sawtooth signal and used an oscilloscope to record that waveform and the current through the LEDs. A typical scope measures only voltage, so I placed a 5.1-ohm resistor between the MOSFET's source pin and ground. Then I could capture the voltages across this resistance and directly relate it to current. (A resistor costs a lot less than a DC-current probe for a scope.)

The schematic diagram in **Figure 22.7** shows my test circuit and includes the two measurement points. **Figures 22.8** and **22.9** shows the captured signals. The upper trace of the sawtooth wave uses a scale of 2.0 volts/division. The lower trace of the voltage across the 5.1-ohm resistor uses a scale of 0.500 volts/division. The sawtooth signal had a frequency of 15.53 kHz. (The 5.1-ohm resistance offsets the source voltage slightly from ground, but for simplicity I ignored it.)

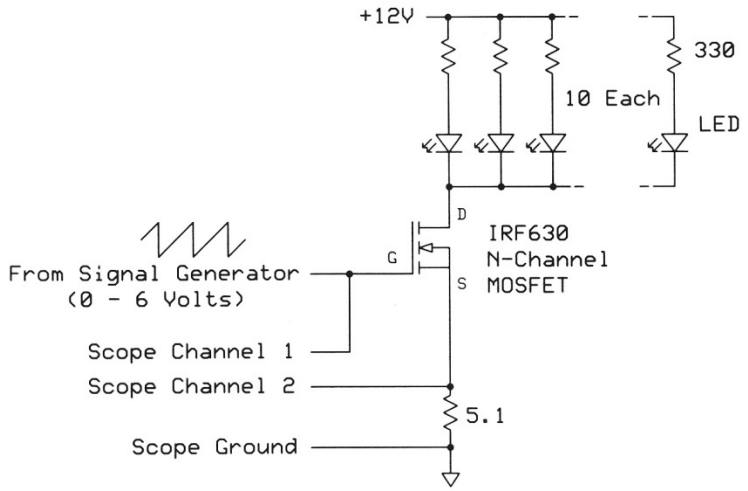


Figure 22.7.
This circuit shows the addition of a 5.1-ohm resistor between the MOSFET source and ground. The voltages measured across this resistance accurately represents current: I = E / R.

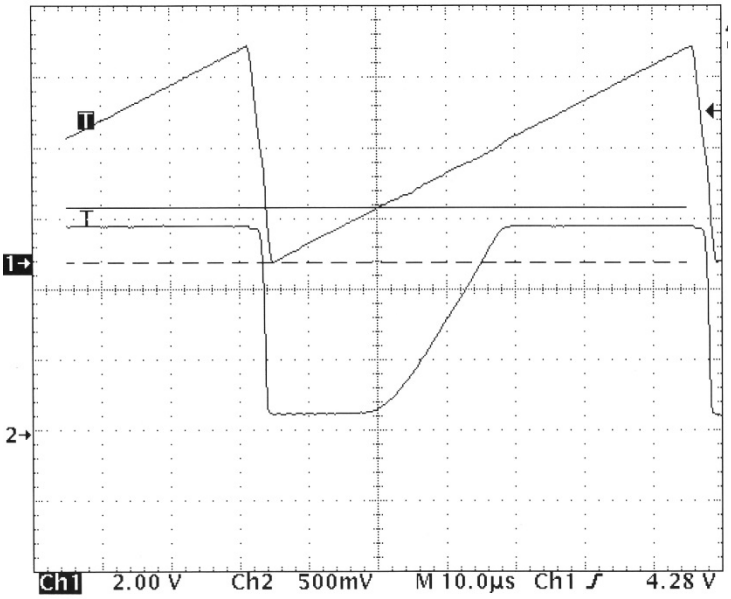


Figure 22.8.
Alignment of the fixed middle vertical scale line with the start of current flow (bottom signal) lets me find the corresponding voltage on the sawtooth signal (upper signal). At this point, the sawtooth signal measured 1.56 volts. The dashed horizontal cursor line represents ground for the sawtooth signal. The dark horizontal cursor line marks the voltage-measurement point.

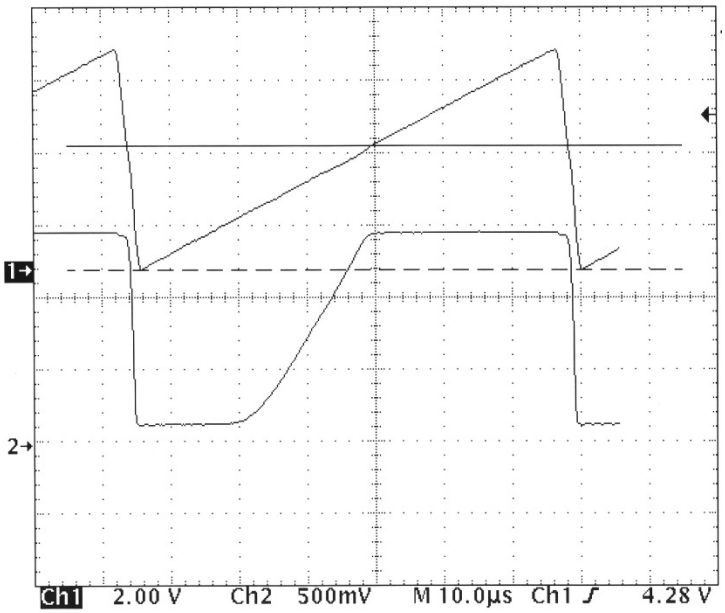


Figure 22.9.
This image displays the same signals shown in **Figure 22.8**, but with the waveforms shifted horizontally. Now the middle vertical scale line intersects the maximum current-flow point (bottom signal) for the LEDs and resistors. The vertical scale intersects the sawtooth wave (upper signal) at 3.44 volts. An increase in the gate voltage (V_{GS}) had no effect on the current flow because the resistors limited it to 300 mA.

The measurement results for the LED test circuit show that current flow starts with a gate voltage of 1.56 volts. That voltage corresponds roughly to the gate-to-source threshold voltage – $V_{GS(th)}$ – for the IRF630. As the gate-source voltage continues to increase, the MOSFET resistance decreases and thus more current flows. You see a straight-line increase, or ramp, for the gate voltage and a corresponding straight-line response of current flow through the MOSFET. Although these signals have different slopes, they still have a proportional

relationship. A side-by-side look at the Celsius and Fahrenheit temperature scales shows the same type of change at a set ratio.

This upward diagonal-line portion of the current-flow trace represents the time during which the MOSFET operates in a linear fashion, so we call it the *linear region*. (Some articles and books call it the ohmic or triode region.) In effect, the MOSFET acts like a variable resistor. Instead of having a moveable mechanical control, a variable voltage at the gate input changes the MOSFET resistance from the drain to the source.

A MOSFET has a third operation mode; *saturation*. (**CAUTION:** When used to describe MOSFET operation, the word "saturation" means something completely different from the same term used with bipolar junction transistors. Do not confuse the two definitions.)

Like most MOSFET datasheets, the Vishay IRF630 datasheet includes a plot of drain current (I_D) vs. drain-to-source voltage (V_{DS}). Rather than reproduce that small diagram I include one from a MOSFET article on Wikipedia; **Figure 22.10**. This diagram identifies linear and saturation regions separated by a curved dashed line. That curved line connects the points at which the voltage-vs.-current plot becomes a straight line for each of the seven gate voltages. In other words, the point at which any increase in the drain-to-source voltage (V_{DS}) causes no increase in the drain current, I_D.

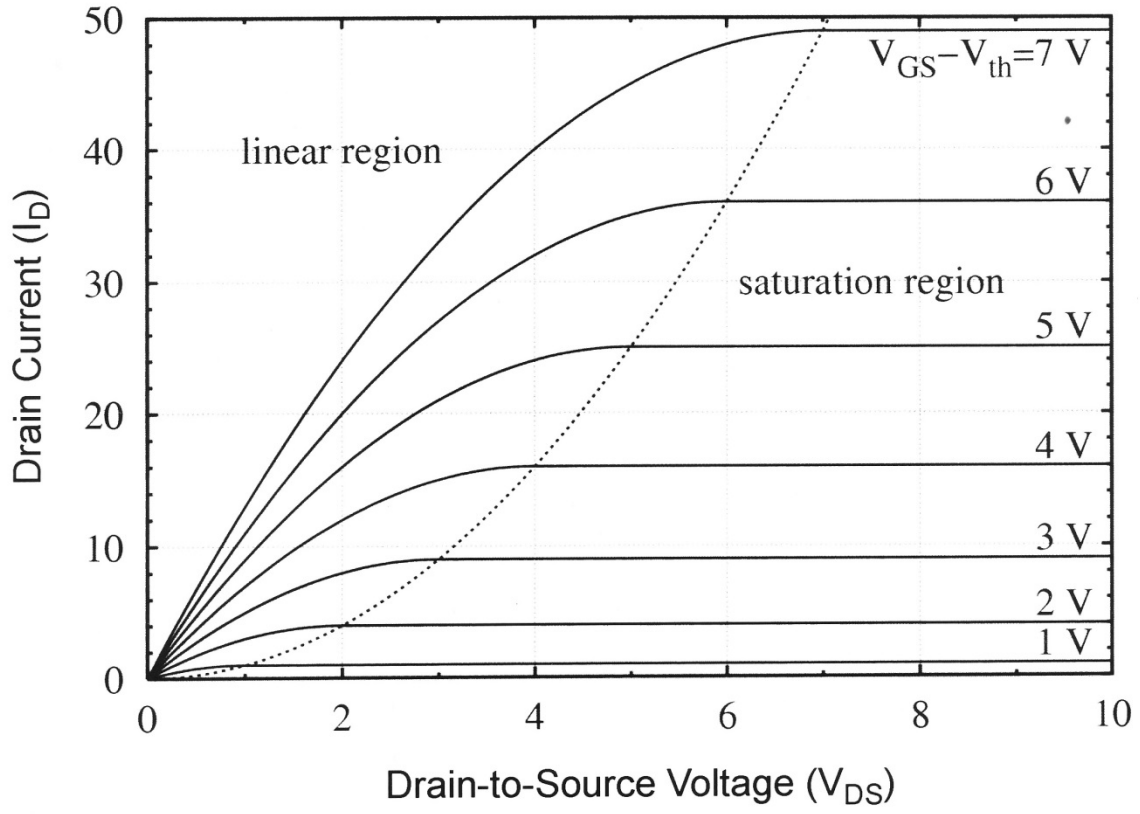


Figure 22.10.
The drain-current vs. drain-source-voltage plot for a MOSFET shows a linear and a saturation region. In the linear region, a MOSFET behaves like a variable resistor. Switch circuits take advantage of this operation mode. Amplifier circuits often operate with MOSFETs in the saturation mode.
Courtesy of Cyril Buttay via creativecommons license. The author used GNUPLOT to reformat the information.

Plots in data sheets might show more lines, perhaps created for half-volt increments of the gate voltage. Always determine whether the plotted gate voltages correspond to the voltage above the source voltage, or the voltage above an N-channel MOSFET's gate-threshold voltage, $V_{GS(th)}$. The plotted gate voltages for a P-

channel MOSFET could correspond to the voltage below the source voltage, or the voltage below the gate-threshold voltage, $V_{GS(th)}$.

Within the linear region of **Figure 22.11** you'll see a dark vertical line, identified with an arrow, drawn upward from the 2-volt V_{DS} scale mark. Wherever this line intersects one of the plotted lines, a dashed horizontal line goes to the *y* axis, Drain Current (I_D). For every 1-volt increase in the V_{GS} *above the gate-threshold voltage* (V_{GS} - $V_{GS(th)}$), you see a proportional and equal increase in the drain current (I_D). In this illustration the drain current scale uses arbitrary units rather than amperes. Actual current depends on the load resistance.

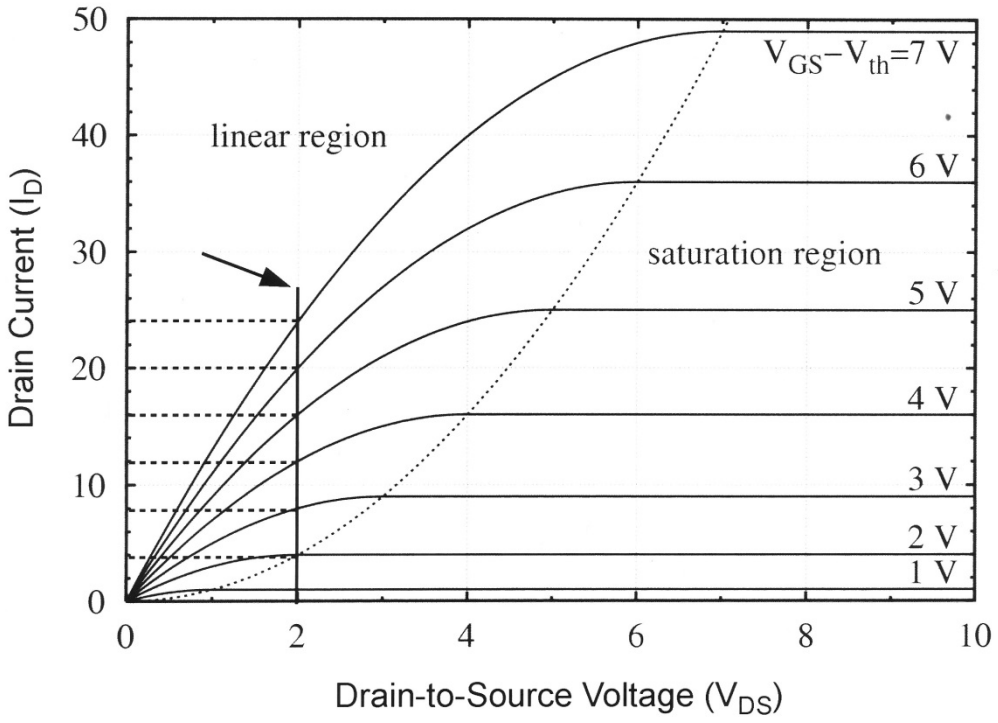


Figure 22.11.
The horizontal dashed lines show the linear relationship between the gate voltage and the resulting increase in drain current in a MOSFET circuit. To get the actual applied voltage, add these reported values to the gate-threshold voltage ($V_{GS(th)}$). The diagram shows this relationship as V_{GS} - $V_{(th)}$ = V. Unfortunately, people do not always use a standard notation for this type of circuit information.

For a gate input of 3 volts (right side of **Figure 22.11**), you would measure I_D equal to 8 current units. Increase the gate input to 4 volts, and current increases to 12 current units, increase the gate voltage to 5 volts, and the drain current rises to 16 current units. Thus, for every 1-volt increase in the gate voltage, the drain current increases by four current units. When you need to use a MOSFET as a switch, operate it in the linear region.

Within the saturation region, any increase in the drain-to-source voltage causes no increase in the drain current. The MOSFET works like a constant-current sink or source. If a circuit increases the gate-to-source voltage of a "saturated" MOSFET, however, the drain current increases. This operation mode corresponds to the linear mode for a bipolar junction transistor. You can see why confusion about the word saturation can cause problems.

When you plan to use MOSFETs for the on-off control of LEDs, relays, motors, or other devices that draw large quantities of current, the MOSFET should turn on or off as quickly as possible. The fast switching speed changes the MOSFET resistance quickly from open circuit to closed circuit and *vice versa* so it doesn't waste energy as heat. MOSFETs can get hot during normal operation and might need heat sinks. A Motorola application note provides helpful information about heat sinks and mounting options (Ref. 2).

Engineers use MOSFETs to control the brightness of LEDs and to produce a range of colors by using green, red, and blue LEDs. If a circuit cannot change the gate voltage to control LED intensity, what other technique could circuit-designers use? Find out in the Answers section at the end of this experiment.

Note: The N- and P-channel MOSFET families have two branches, named *enhancement mode* and *depletion mode*, which describe how the transistors operate and how a gate voltage affects their operation. This experiment and examples all use enhancement-mode MOSFET. The Wikipedia article, "Depletion and enhancement modes," provides a short (edited) explanation of these two modes:

> *Enhancement-mode MOSFETs* remain off when the gate voltage equals source voltage. As the gate voltage changes in the "direction" of the drain voltage; that is, toward the V_{DD} supply bus, the MOSFET starts to conduct current.
>
> In *depletion-mode MOSFETs*, the devices turn on when the gate voltage equals source voltage. As the gate voltage changes in the "direction" of the drain voltage; that is, toward the V_{DD} supply bus, the MOSFET resistance increases and decreases the flow of current.

CAUTION: Because MOSFETs use a thin insulator between the gate material and the underlying semiconductor substrate, even a small amount of static electricity can discharge through the oxide layer and destroy it. Some MOSFETs include an internal conductive "path" that helps dissipate some static electricity, but always use care when you handle MOSFETs. I highly recommend engineers and experimenters use a well-grounded static-dissipating mat, or work surface, and a conductive wrist strap. (Ref. 3.) Electronic-equipment suppliers sell small mats for under $20.

The MOSFETs used and described in this experiment come in standard TO-220 "tab" packages that easily attach to heat sinks as needed. MOSFETs come in smaller packages, but those devices carry less current than their larger cousins. The 0.1-inch (2.5-mm) space between the leads on a TO-220 package lets the larger MOSFETs easily fit into solderless breadboards. The hole through the metal tab takes a 6-32 or M3 machine screw. **Figure 2.12** shows the orientation of the IRF630 MOSFET and the connection labels for the gate, source, and drain.

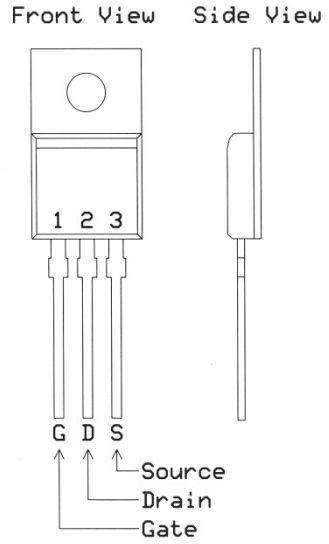


Figure 2.12.
This diagram shows two views of an IRF630 N-channel MOSFET in a TO220 package. Always check a component's datasheet for the pin designations and pin numbers. The two types of MOSFETs used in this experiment all come in this type of package and have the same pin configuration.

In the following steps you will learn how N-channel and P-channel MOSFETs differ, explore the behavior of MOSFETs, measure gate-to-source voltages for MOSFETs in circuits, and learn why MOSFETs often need pullup or pull-down resistors.

Step 1.

Construct the circuit shown in **Figure 22.13**. Connect the all 10 330-ohm resistors to the +12-volt supply. Set the trimmer resistor to about halfway between its end points.

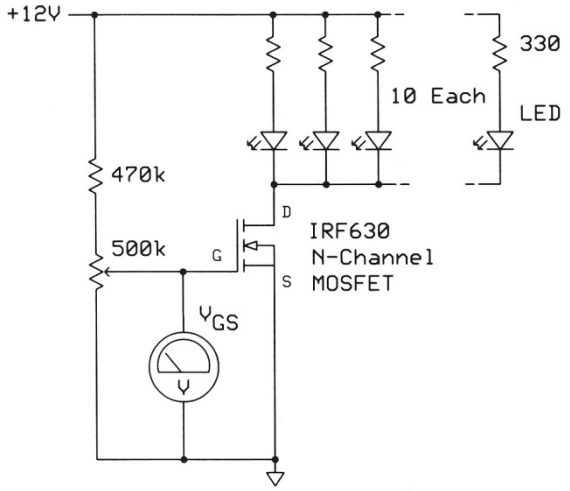


Figure 22.13.
Circuit diagram for the measurement of a MOSFET gate-to-source voltage, V_{GS}.

Turn on power to the breadboard. The LEDs might turn on dimly or might remain dark, depending on the initial setting of the trimmer resistor. At this time, the state of the LEDs does not matter.

a. Adjust the trimmer control until the LEDs turn off. Then adjust the trimmer to the point at which you first start to see light. Note the position of the trimmer control at this point.

b. Continue to adjust the trimmer until the LEDs turn completely on. If you can't determine when the LEDs turn on fully, rotate the trimmer control until the LEDs turn fully on. Then rotate it in the opposite direction until you see the LEDs dim slightly. Note the position of the trimmer control at this point.

c. Adjust the trimmer position about half way between the setting noted in a) and b) above. The LEDs all turn on, but not at full brightness. What gate voltage do you measure with the trimmer in this position? I measured 4.2 volts between the MOSFET gate and source in my circuit – the gate-to-source voltage, V_{GS}. Leave the circuit on for five to 10 minutes. Take a break. Get a snack.

After you return to the breadboard, touch a grounded object or touch your static-dissipating mat to get rid of any built-up static charge. Then carefully, lightly, and briefly touch the metal tab at the back of the MOSFET. With the LEDs only partly on, what temperature did you feel? Hot, room temperature, cool, or cold? Was the temperature about what you expected?

My IRF630 MOSFET tab felt hot. Do you know why?

When you set the trimmer so the LEDs do not turn fully on, the MOSFET still has considerable resistance and a large current flows through it. You might remember that power dissipated by a resistor equals current times the voltage across the resistor, or current squared times resistance (I^2R).

Adjust the trimmer so the LEDs turn fully on. I adjusted my trimmer and measured a V_{GS} as 6.02 volts. Continue to power the circuit and wait about 10 minutes. After that period, again feel the metal tab on the IRF630 MOSFET. Did the tab feel hot, warm, cool, or about room temperature? My transistor felt cool. Why would the transistor feel cooler during this test?

The data sheet for an IRF630 MOSFET specifies a maximum on-resistance ($T_{DS(on)}$) of 0.4 ohms, or 400 milliohms. Given power dissipation equals I^2R, calculate the power dissipation for 300 mA and 400 milliohms (0.4 ohms):

Power $= I^2 * R$ or $(0.3\ amperes)^2 * 0.4\ ohms = 0.036\ watts$

Thus the IRF360 MOSFET operates with little power dissipated as heat – about ambient temperature. Please turn off power to your breadboard.

Step 2.

You can think of the insulated gate portion of a MOSFET as one plate of a capacitor. The other parts of the semiconductor material act like the opposite plate. When you connect the gate to a power source electrons move to or from the gate, depending on the gate-to-source voltage polarity. The accumulation of charge on the gate causes the MOSFET to conduct electricity.

Most capacitors have a small amount of internal "leakage" that lets charge to flow from one plate to the other. The IRF630 MOSFET has a maximum gate-source leakage current (I_{GSS}) of ±100 nA. That value – found in the IRF630 data sheet – represents the current that leaks from the gate to the source under the manufacturer's test conditions.

What happens, though, if you leave a MOSFET gate disconnected or "floating" in a powered circuit? Disconnect the MOSFET gate wire from the trimmer resistor (**Figure 22.14**), but leave a short wire (2 to 3 inches, or 5 to 8 cm) connected to the MOSFET gate terminal. The other end of this wire should remain unconnected.

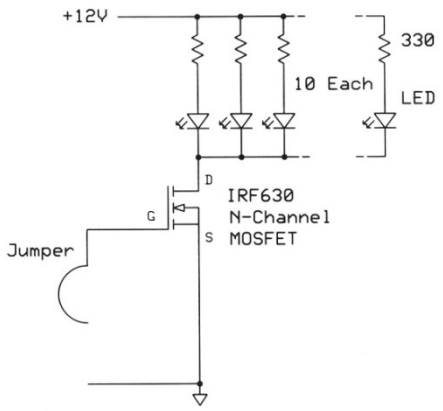


Figure 22.14.
Schematic diagram for a MOSFET circuit with an unconnected gate terminal.

Turn on power to your circuit. If the LEDs turn on, briefly touch the unconnected end of the gate wire to ground. Do not touch the metal part of the wire's free end. The LEDs should turn off. Wait a few seconds and then with your fingers touch the metal at the free end of the jumper wire. What happens to the LEDs?

The LEDs will turn on because your body can transfer enough charge to the MOSFET gate to let current flow from the drain to the source. If your LEDs do not turn on, move a bit in your chair and then touch the bare end

of the gate wire. You should have built up enough charge to inject into the MOSFET gate. Briefly ground the bare end of the jumper. The LEDs should go off. The ground connection removed the charge from the MOSFET's gate.

When you design a MOSFET circuit, you should not leave a gate terminal unconnected or at an unknown voltage. To remove any accumulated charge and overcome the problem of an unexpected on or off condition from a MOSFET, connect a high resistance between the gate and ground for a N-channel MOSFET. The resistor provides a path to ground for any accumulated charge.

Turn off your circuit and modify it to include a 10-kohm resistor between the gate and ground as shown in **Figure 22.15**. The added resistance should ensure any unwanted charge accumulated on the gate gets shunted to ground.

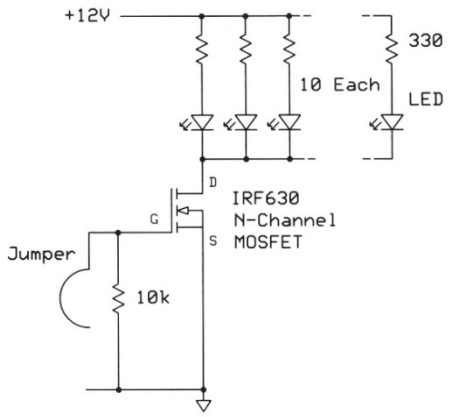


Figure 22.15.
This modified circuit includes a *pull-down* resistor that will shunt to ground any unwanted charge on an N-channel MOSFET gate.

Turn on power to your circuit. Did the LEDs turn on immediately? They should not. Touch your finger to the bare end of the gate jumper wire. Do the LEDs turn on? Again, they should not. A 10-kohm resistor worked well and you can use one as a pull-down resistor in practical N-channel MOSFET circuits. We pay a small price by adding this resistance to the circuit. Do you know why? The added resistance in the gate circuit will draw a small current from the circuit that controls the MOSFET gate. You may turn off the circuit.

Suppose you use a 6-volt signal to control the gate input on a MOSFET that has a 10-kohm pull-down resistor. How much current flows through the resistor? Use Ohm's law to do the calculation. I came up with 0.6 mA – not a lot of current. But if you have many MOSFETs in a circuit, the currents through the pull-down resistors would add up. In battery-operated equipment, saving a few milliamperes here or there can extend operating times.

Could you use a higher-value resistance? Try another experiment. Substitute a 100-kohm resistor for the 10-kohm resistor now in your circuit. Do the LEDs turn on when you apply power? Do they turn on when you touch the bare end of the gate jumper wire? Mine did not. The larger resistance helps save power. Depending on the N-channel MOSFET you work with, a pull-down resistance of 10-kohms to 100-kohms should work well.

Step 3.

Most of the MOSFET circuits I encounter use N-channel devices as low-side switches. Conversely, P-channel MOSFETs serve as high-side switches, so they deserve attention, too. A P-channel MOSFET has many characteristics in common with N-channel devices and its data sheets list the same units, such as V_{GS}, and $V_{GS(th)}$. The P-channel MOSFET symbol in **Figure 22.16** shows the source connected to a V+ power bus and

the drain connected to the load. To turn on the attached LED, the gate requires a voltage *below* that on the source terminal.

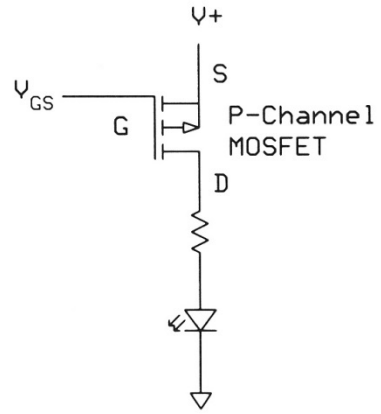


Figure 22.16.
A P-channel MOSFET used as a high-side switch controls an LED. For the LED to turn on, the gate requires a voltage below the V+ voltage on the source terminal. In summary, $V_D << V_S$ and $V_{GS} < V_S$.

In this step you will work with a P-channel MOSFET, the IRF9520. For a complete data sheet, see Reference 4 at the end of this experiment. **Table 22.2** lists some of the most important characteristics for the IRF9520. An IRF9520 can withstand a 100-volt difference between the source and the drain and a 6.8-amp current. Do you notice anything unusual about some of the values in **Table 22.2** compared to those in **Table 22.1**?

Table 22.2. Select specifications for an IRF9520 P-channel MOSFET.

Parameter	Symbol	Test Conditions	Limit	Unit
Drain-to-Source Voltage	V_{DS}		-100	V
Gate-to-Source Voltage	V_{GS}		±20	V
Continuous Drain Current	I_D	T_C = 25°C, V_{GS} at 10V	-6.8	A
		T_C = 100°C, V_{GS} at 10V	-4.8	A
Pulsed Drain Current	I_{DM}		-27	A
Maximum Power Dissipation	P_D		60	W
Operating Temperature Range	T_J, T_{stg}		-55 to +150	°C

Parameter	Symbol	Test Conditions	Min	Typ	Max	Units
Drain-to Source Breakdown Voltage	V_{DS}	V_{GS} = 0V, I_D = 250 µA	-100	–	–	V
Gate-to-Source Threshold Voltage	$V_{GS(th)}$	V_{DS} = V_{GS}, I_D = 250 µA	-2.0		-4.0	V
Gate-to-Source Leakage	I_{GSS}	V_{GS} = ±20			±100	nA
Zero Gate-Voltage Drain Current	I_{DSS}	V_{DS} = 200V, V_{GS} = 0V			-100	µA
		V_{DS} = 160V, V_{GS} = 0V, T_J = 125°C			-500	
Drain-to-Source On-State Resistance	$R_{DS(on)}$	V_{GS} = 10V, I_D = 5.4A			0.60	Ohm
Turn-On Delay Time	$T_{d(on)}$	V_{DD} = 100V, I_D = 5.9A, R_G =12Ω, R_D = 16Ω		9.6		nsec
Turn-Off Delay Time	$T_{d(off)}$			21		nsec

Some values have a minus sign because measurements for a P-channel MOSFET use the *source terminal* as a reference (**Figure 22.17**). In these two circuits, the voltmeter's ground lead always connects to the source.

Why? In each circuits, the source terminal connects directly to a stable and constant voltage; ground for the N-channel transistor and a +12-volt source for the P-channel device. When you use a meter to make measurements equivalent to those reported in a data sheet, ensure you choose the proper polarity setting for your meter. Or pay careful attention to the sign of the resulting measurement in a digital meter.

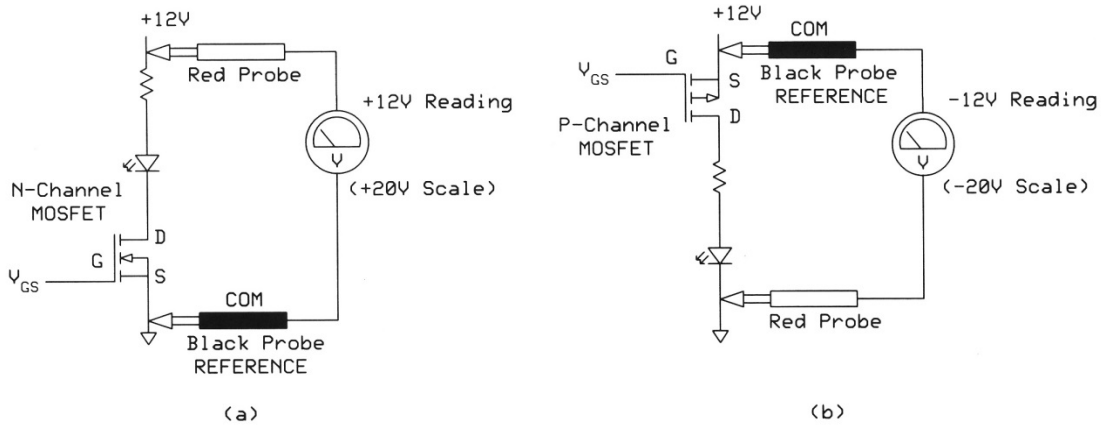


Figure 22.17.
These two circuits show **a)** the connection of a meter that measures the voltage between the source and drain for an N-channel MOSFET and **b)** the same measurement for a P-channel MOSFET. In both cases the voltmeter ground or common (COM) always connects to the source terminal.

Of course, you can measure the voltages in a circuit any way you choose. The use of the source as a reference applies mainly to the datasheet values, but you must understand what the published information means so you can interpret specifications properly and use them to create a circuit that works.

When you use a P-channel MOSFET in a circuit, the source connects to a voltage higher than that at the drain. Just remember that the source in a **P**-channel MOSFET connects to the more **P**ositive voltage. The source in an **N**-channel MOSFET connects to the more **N**egative voltage. And the MOSFET **D**rain always connects to the **D**evice you want to control.

Step 4.

Now you will construct a P-channel MOSFET switch circuit to control the 10 LEDs in your breadboard. If you still have power applied to your circuit, please turn it off now. The circuit in **Figure 22.18** shows the needed connections and components. The IRF9520 MOSFET has the same package type and pin configuration shown in **Figure 22.10**. In this circuit, each LED cathode connects to ground and each LED anode connects to the MOSFET drain through a 330-ohm resistor. This circuit does not include a pull-down or pullup resistor on the IRF9520 gate terminal.

Ensure you connect the voltmeter ground (COM, usually black) lead to the +12-volt bus and connect the voltage-input lead (usually red) to the gate terminal. If you have an analog meter, set the meter polarity to minus (-). Digital meters will automatically change polarity and show a minus sign on the display. For an illustration of meter connections, see voltmeter 2 in **Figure 22.19**.

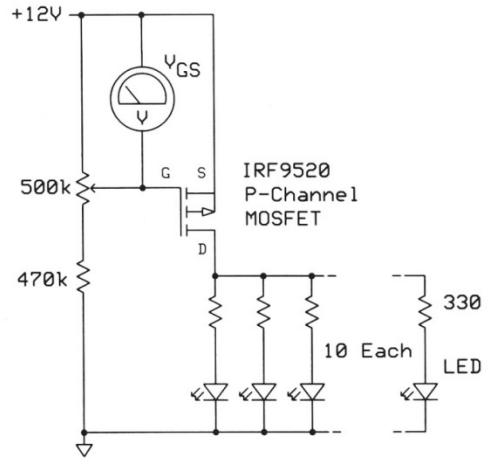


Figure 22.18.
Control of 10 LEDs by an IRF9520 P-channel MOSFET.

Set the 500-kohm trimmer resistor to a position about halfway between its end points and turn on power to your circuit. The state of the LEDs doesn't matter. Adjust the trimmer control until the LEDs turn off. Then adjust the trimmer in the opposite direction until you first start to see light from the LEDs. Now measure the voltage and write it in the space below.

LEDs first turn on at _____ volts.

This measurement provides a good approximation of the *gate-to-source* threshold voltage, $V_{GS(th)}$. How well does it match the $V_{GS(th)}$ in the IRF9520 data sheet (**Table 22.2**)?

I found the LEDs started to turn on at -2.89 volts. They came to full brightness at -4.02 volts. When I used an ammeter instead of my eyes to detect current changes, current started to flow through the LEDs when V_{GS} = -2.94 volts. Current reached a maximum when V_{GS} = -4.31 volts. At that point the LED series resistor limited current flow. **Figure 22.19** shows three ways to measure the voltage at a P-channel MOSFET gate.

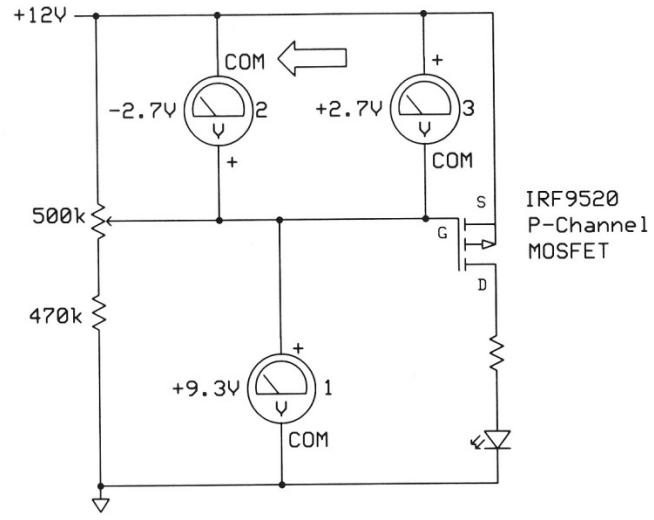


Figure 22.19.
Voltage measurements in a P-channel MOSFET circuit. Note that Voltmeter 2 has its COM lead connected to the +12-volt bus at the MOSFET source terminal (large arrow). This meter measures the voltage *below* 12 volts and gives it a minus sign. You use the meter-2 connections in Step 4.

The result from this step shows that to control a P-channel MOSFET, a circuit must *reduce* the voltage at the gate below the voltage at the source. The IRF9520 MOSFET data sheet specifies this voltage difference can range from -2 to -4 volts. In practice, I would recommend a gate voltage at least 4.0 volts *below* the source voltage. To ensure an unconnected or unpowered gate does not turn on a load, use a *pull-up* resistor between the gate and the source. The same values discussed for an N-channel – 10 kohms to 100 kohms – will work well.

Step 5.

Given that a P-channel MOSFET should have a pull-up resistor on its gate terminal, how would adding one affect the circuit in **Figure 22.15**? Let's find out. Turn off power to your circuit and add a 10-kohm pullup resistor to the circuit as shown in **Figure 22.20**. Turn on power and adjust the 500-kohm trimmer. At what voltage do the LEDs turn on or off? Do you see any change? Can you explain the results?

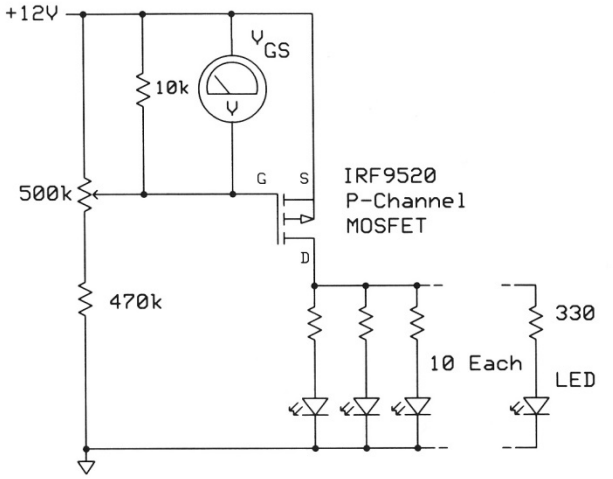


Figure 22.20.
The circuit used to measure the gate-to-source threshold voltage with an added 10-kohm pull-up resistor.

The added pullup resistor does more that remove unwanted charge from the MOSFET gate. It affects the overall circuit behavior. The diagram in **Figure 22.21** shows two configurations and the equivalent circuit for each set of resistors. The (a) portion shows the trimmer control at one extreme where the "wiper" connects directly to the +12-volt bus. This circuit keeps the V_{GS} at the same voltage as the source, so the MOSFET will not conduct current and the LEDs remain off.

The (b) portion of **Figure 22.21** shows the trimmer control at the opposite extreme – the connection between the 500-kohm trimmer and the 470-kohm resistor. Now the circuit has two resistors in parallel and a 470-kohm resistor is series with them. What voltage exists at the gate in this case, and will the LEDs turn on?

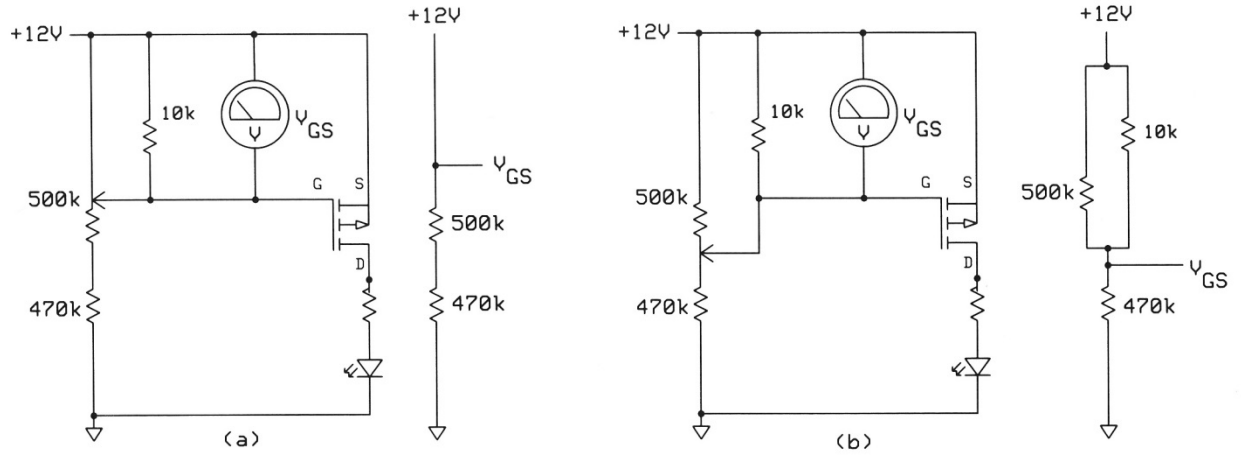


Figure 22.21.
Equivalent circuits for the trimmer resistor at one extreme (**a**) and the opposite extreme (**b**). The resistances and the +12-volt power value let you calculate the gate voltage for the (**b**) circuit.

Again, Ohm's law comes to the rescue. For any number of resistors in parallel, the total resistance is *always* lower than the smallest resistance in the parallel circuit. So for the 10-kohm and 500-kohm resistors, expect a resistance below 10 kohms. For *n* number of resistances in parallel, the following formula lets you calculate the equivalent resistance:

$$Equivalent\ resistance = 1 \,/\, [(1/R_1) + (1/R_2) + (1/R_3) + \ldots + (1/R_n)]$$

When a circuit has only two resistances in parallel, the formula below simplifies calculations:

$$Equivalent\ resistance = \frac{(R_1 * R_2)}{(R_1 + R_2)}$$

For the two resistances in parallel, the equivalent resistance amounts to:

$$\frac{(10{,}000\ ohms * 500{,}000\ ohms)}{(10{,}000\ ohms + 500{,}000\ ohms)} = 9800\ ohms$$

Add this equivalent resistance to the 470 kohms for a total resistance of about 480,000 ohms between the +12-volt bus and ground. Next, divide 12 volts by the total resistance. Now you know the voltage you would measure across each ohm in the total resistance. Of course you can't actually do that.

$$12\ volts \,/\, 480{,}000\ ohms = 0.000025\ volts/ohm \quad \text{or} \quad 0.025\ mV/ohm$$

But you can calculate the voltage across the 9800-ohm resistance (500-kohm and 10-kohm resistors in parallel) and at the MOSFET gate terminal:

$$9800\ ohms * 0.025 mV/ohm = 250\ mV \quad \text{or} \quad 0.25\ volts$$

Remember, this represents the voltage *below* the 12 volts on the power bus. Thus the gate voltage decreases by only -0.25 volts *below* the the 12-volt bus voltage. This small difference does not get close to the gate-to-source threshold voltage ($V_{GS(th)}$) of -2.0 to -4.0 listed in **Table 22.2** for an IRF9520 MOSFET. In other words, the gate "sees" 11.75 volts when you move the trimmer resistor to the position shown in **Figure 22.21(b)**.

Why did the 10-kohm resistor cause this type of behavior? Should you remove the 10-kohm resistor from the circuit?

No, leave the 10-kohm pullup resistor in place. This step illustrated what happens when you try to turn on a MOSFET with a gate-voltage signal from a high-resistance, or high-impedance output from a microcontroller or logic circuit; the 500- and 470-kohm resistors in this example. If the circuit had used a 1000-ohm trimmer resistor and a 1000-ohm fixed resistor with the 10-kohm pullup resistor, the pullup resistor would not have had such a large affect on the gate voltage. Instead, the 1000-ohm trimmer resistor could have adjusted the gate voltage far enough below 12 volts to properly control the MOSFET and thus the LEDs. You can substitute a 1000-ohm fixed resistor for the 470-kohm resistor and substitute a 1000-ohm trimmer for the 500-kohm trimmer if you wish.

When you drive, or control, the gate on a MOSFET, you need a low-resistance, or low-impedance connection to a voltage for the MOSFET gate. You can create good gate drivers with bipolar transistors, special MOSFET-driver ICs, logic-ICs, and MCU outputs. In the next experiment you will learn more about MOSFET gate-drive circuits, about practical applications for MOSFETs, and how to protect circuits from the high voltages MOSFETs might switch on or off.

MOSFET Power Ratings

Before you start the next experiment, you should know whether or not a given MOSFET will handle the load you plan to control. The IRF630, for example, has a maximum power dissipation (P_D) of 74 watts. An IRF9520 can dissipate only 60 watts. Always ensure the P_D value for a circuit does not exceed that for the MOSFET you plan to use. Find the P_D values for MOSFET in their datasheets. All too often, instructions for hobbyist and experimenter circuits use so-called "standard" MOSFETs that many other project or circuit also uses. Few such projects include an analysis of operating voltages and power dissipation.

As an example, suppose we have a control circuit that will use a MOSFET to adjust the current through a 6-volt DC heater rod. To investigate a MOSFET's power-handling requirements start with the circuit diagram (**Figure 22.22**) and information about the components.

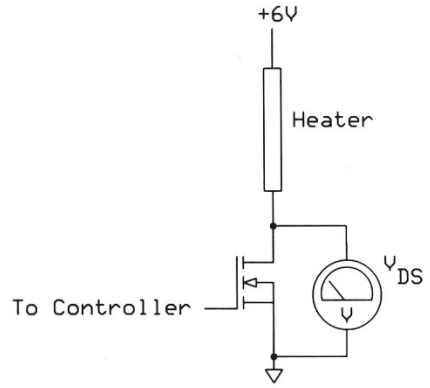


Figure 22.22.
A simple circuit that helps demonstrate the importance of a MOSFET power-dissipation rating.

When the MOSFET turns off (open circuit), the V_{DS} equals the supply voltage; 6 volts. No current flows, so the load cannot cause a drop in potential. When the MOSFET turns on completely (closed circuit), the V_{DS} drops to 0 volts because the measurement point connects directly to ground. Now the load draws 50 amperes. That's a lot of current, but it serves well in this hypothetical example. Assume we know the family of MOSFETs chosen for this circuit can handle 50 amperes.

Figure 22.10, shown earlier, provides the drain current (I_D) vs. drain-to-source voltage (V_{DS}) plot for the MOSFET in the circuit. I have added a straight *load line* that connects the two points just described: the 0 V_{DS}, 50 I_D point on the *y* axis and the 6 V_{DS}, 0 I_D point on the *x* axis (**Figure 22.23**). You can make measurements of current at various supply voltages across the load by itself to provide more points to plot. Along the load line you see that when V_{DS} equals 4 volts, the MOSFET carries about 14 amperes, and when V_{DS} equals 2 volts, the current increases to about 34 amperes.

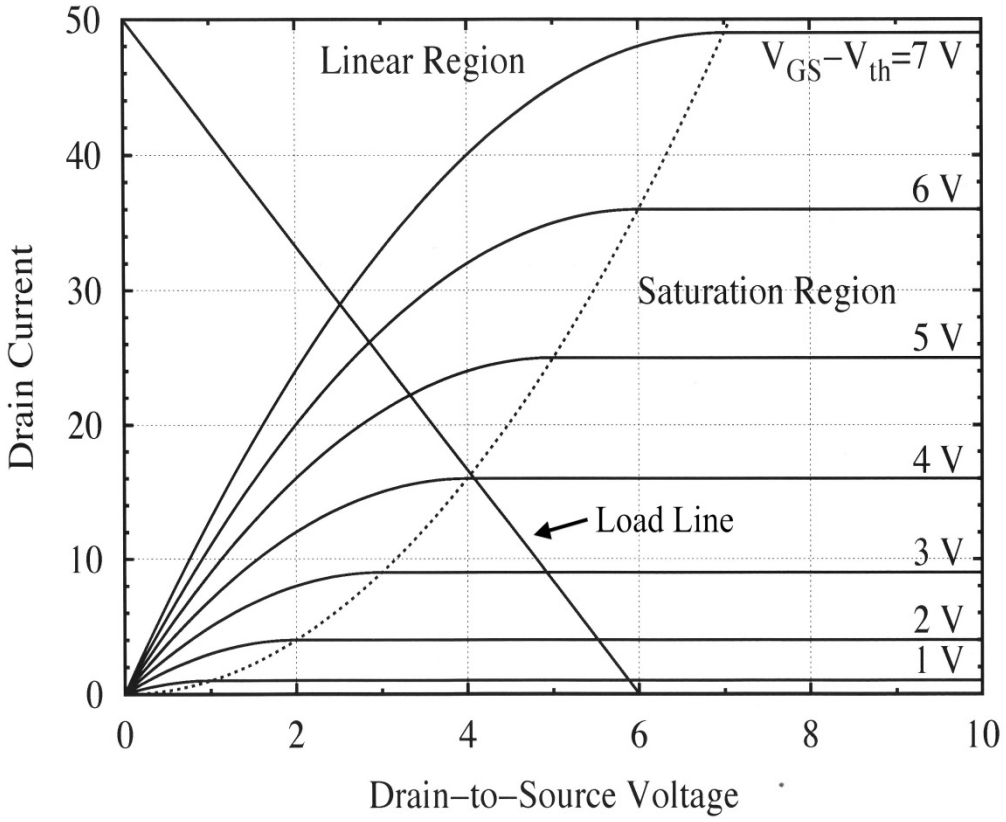


Figure 22.23.
A load line superimposed on a plot of the drain current (I_D) vs. drain-to-source voltage (V_{DS}) at seven gate voltages for a MOSFET. The gate voltages take into account the gate-threshold voltage, $V_{GS(th)}$. That means a gate voltages shown in the plot *added to* the $V_{GS(th)}$ value equals the voltage applied to the MOSFET gate.

Based on the portion of the load line in the MOSFET's linear region, only gate voltages from about 4.5 volts to 7 volts or greater let the MOSFET act as a switch. The load line comes too close to the 4-volt gate signal for us to include it in the gate-voltage range. Small circuit variations, temperature changes, and shifts in the heater's 6-volt power source could cause problems if we use a gate voltage close to 4 volts. **Figure 22.24** eliminates the information for the 1- through 4-volt gate-voltage lines.

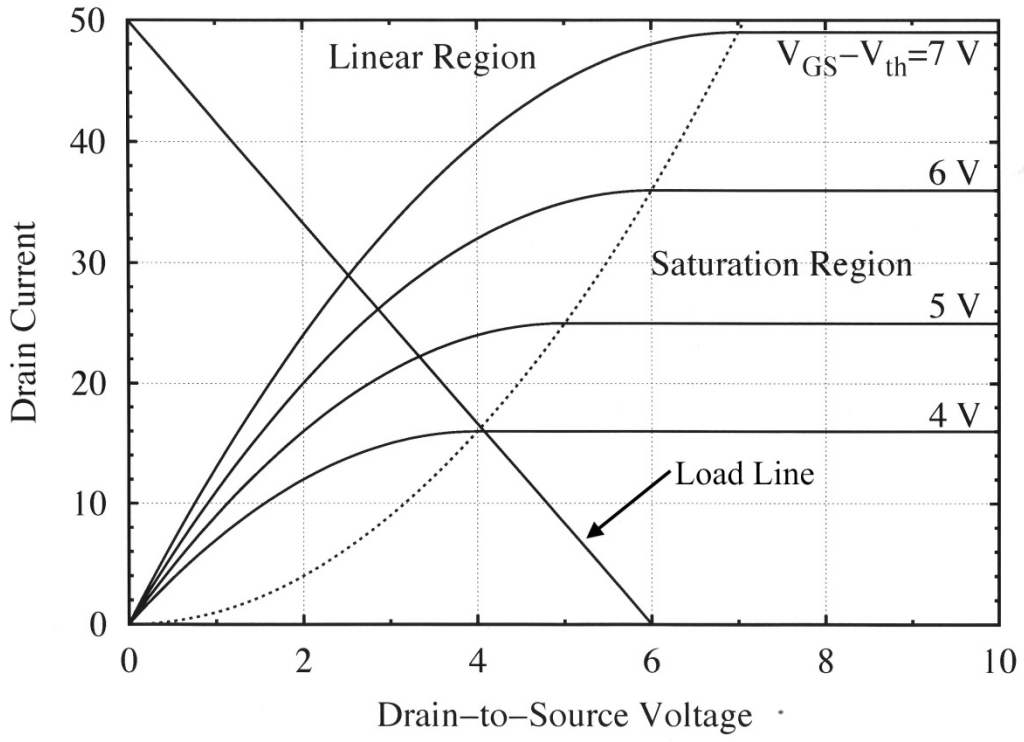


Figure 22.24.
This plot of the load line includes the load line but only the the 5-, 6-, and 7-volt gate-input information for our hypothetical MOSFET and heater load.

Now you must determine whether the power dissipation for a MOSFET meets the heater-control requirements. A MOSFET manufacturer offers one N-channel device with a 60-watt P_D rating and another – at slightly higher cost – with an 85-watt rating. If you use the 60W MOSFET, you save money, but the design might need the 85W device to handle the power and provide a "margin" for slight increases in current or voltage. For the 60-watt MOSFET use the equations below to calculate the current to plot for a given voltage on the *x* axis:

$$Power\ (watts) = I\ (drain\ current) * E\ (drain\text{-}to\text{-}source\ current)$$

$$\frac{60\ W}{E} = I$$

Figure 22.25 shows two calculated power curves. The 60-watt curve dips below the load line, so it will not meet the heater circuit's power-dissipation requirements. The 85-watt MOSFET remains above the load line, so it should work well. (This example assumed the same gate-voltages for the 60- and 85-watt MOSFETs.)

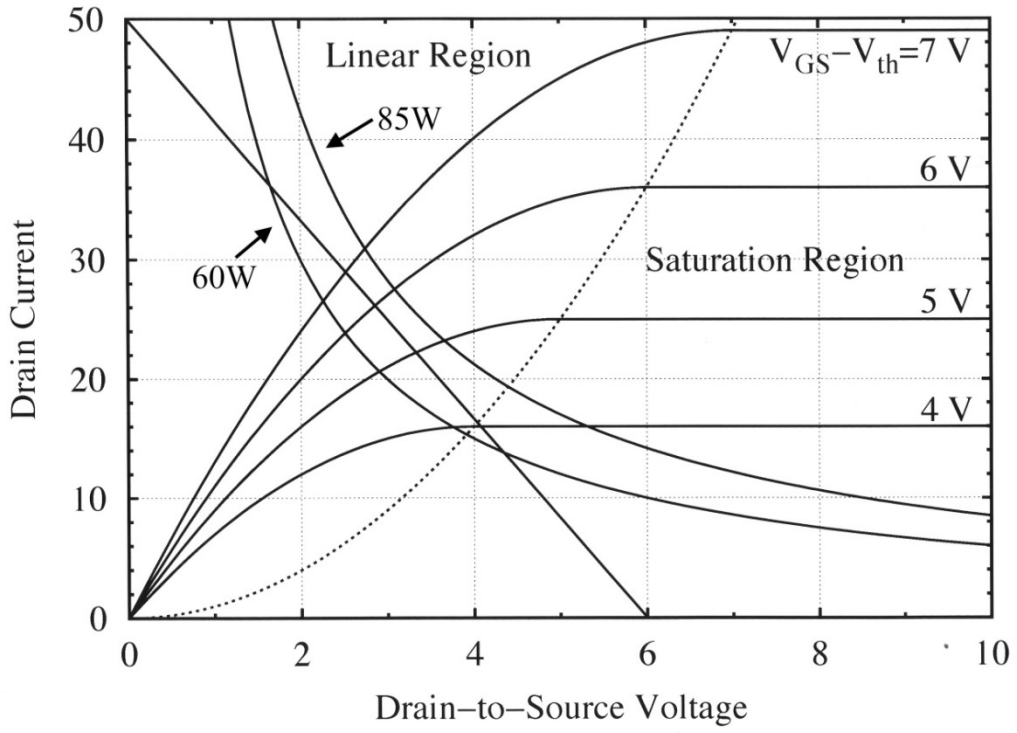


Figure 22.25.
Plot of power-dissipation values for a 60- and an 85-watt MOSFET superimposed on the circuit's load line and the gate voltages for two MOSFETs.

Although the 85-watt MOSFET can handle the heater load, you face another problem: How will a 3.3- or 5-volt microcontroller create a high enough gate voltage to control the MOSFET? The next experiment provides several techniques that help overcome that problem. And you will learn more about how to use MOSFETs to control real-world devices such as relays, solenoids, motors, LED arrays, and so on.

References

1. "IRF630 Data Sheet," Vishay, 2011. http://www.vishay.com/docs/91031/sihf630p.pdf.

2. Roehr, Bill, "Mounting Considerations for Power Semiconductors," Application Note AN1040, Motorola Semiconductor. Undated. http://www.freescale.com/files/rf_if/doc/app_note/AN1040.pdf. (The Motorola semiconductor business now belongs to Freescale Semiconductor.

3. Doumergue, Pierre-Laurent, "Handling Instructions & Protection against Electrostatic Discharges," Application Note APT0502, October 2005, Microsemi Corp. http://www.microsemi.com/index.php?option=com_docman&task=doc_download&gid=14730.

4. "IRF9520 Data Sheet," Vishay, 2011. http://www.vishay.com/docs/91074/91074.pdf.

For More Information

– -, "The ARRL Handbook for Radio Communications," 2009 ed., American Radio Relay League, Newington, CT. ISBN: 978-0-87259-139-4. Excellent discussions of transistors and circuits. http://www.arrl.org

.

– -, "MOSFET Basics," Application Note AN-9010, Fairchild Semiconductor. 2013. http://www.fairchildsemi.com/an/AN/AN-9010.pdf.

– -, "Understanding Power MOSFET Data Sheet Parameters," Application Note AN11158, NXP Semiconductors, February 2014. http://www.nxp.com/documents/application_note/AN11158.pdf.

GNUPLOT home page: http://www.gnuplot.info/.

Answers

Experiment 22, Step 1:

Question: Engineers do use MOSFETs to control the brightness of LEDs. Given that when partially turned on, or operated in its linear "zone," a MOSFET will get hot, how can such LED-intensity control occur?

Answer: Instead of using a MOSFET in its linear zone and varying its resistance, LED dimming circuits can use pulse-width modulation. The duration of the pulse turns the LED on for longer or shorter periods at frequencies the human eyes cannot detect. The pulses require *"sharp" edges so the gate voltage quickly transitions for off to on, and vice versa*.

Experiment No. 23 – Use MOSFETs to Control LEDs and Motors

Abstract

In the previous experiment you examined the relationship between the gate-to-source voltage for a MOSFET and the current flow between the source and drain terminals. Now you will learn how to use MOSFETs to control real-world devices such as motors, relays, lamps, and so on. The steps in this experiment continue to use the N-channel IRF630 and the P-channel IRF9520 MOSFETs. You can skip building the circuits if you don't plan to use MOSFETs, but because these transistors have so many uses, I encourage you to experiment with them in the circuits that follow. This experiment concludes with helpful design information you can use in many situations.

Keywords

metal-oxide-semiconductor field-effect transistor, MOSFET, optical isolation, inverter, open collector buffer, transistor, motor, relay, Zener diode, transient voltage suppression

Requirements

Consider the circuits optional in this experiment. If you want to build, test, and use the circuits, feel free to do so. The following parts list includes they key components for the optional steps.

(2) - IRF630 N-channel MOSFET
(2) - IRF9520 P-channel MOSFET
(1) - IRL620 logic-level N-channel MOSFET
(1) - 2N3904 NPN transistor
(10) - LEDs, red
(1) - LED, green
(1) - 7407 open-collector buffer IC, 14-pin DIP
(1) - 74LS04 inverter, 14-pin DIP
(14) - 330-ohm, 1/4-watt resistors, 5% (orange-orange-brown)
(1) - 470-ohm, 1/4-watt resistors, 5% (yellow-violet-brown)
(4) - 10-kohm, 1/4-watt resistors, 5% (brown-black-orange)
(1) - 100-kohm, 1/4-watt resistors, 5% (brown-black-yellow)
(1) - 5-volt power source
(1) - 12-volt power source
(1) - Solderless breadboard
(1) - Voltmeter or volt-ohm-milliammeter (VOM)

Introduction

Circuits that use a MOSFET as a switch can drive the transistor's gate input in several ways, but they must provide a gate voltage that ensures the transistor turns fully on. Some of those circuits will control high voltages that require electrical separation between a control circuit and a MOSFET. So in this experiment you will learn how to properly drive a MOSFET gate and how to isolate a MOSFET from the rest of a circuit.You will learn about several ways use isolation devices in these types of applications.

To this point, the MOSFET experiments and examples have used circuits powered by a 12-volt supply, and the IRF9520 has used a gate voltage obtained from the same supply. Most microcontrollers and logic ICs, however, cannot operate with a voltage greater than about 5 volts. Newer ICs operate below 3 volts, which can

further complicate a MOSFET circuit design. How can a 3-volt MCU drive the gate in a MOSFET that switches a 60-volt motor on or off? The $V_{GS(th)}$ probably exceeds the MCU's supply voltage.

MOSFETs and Bipolar Transistors

A bipolar transistor can connect to a low-voltage base signal and control a larger voltage between the collector and emitter as you learned in Experiment 21. So bipolar NPN and PNP transistors can control MOSFETs. You learned how to use PNP transistors as high-side switches and NPN transistors as low-side switches to control LEDs. The transistors also can control MOSFETs in a similar way. They provide a low-resistance or low-impedance path to or from a MOSFET gate. **Figure 23.1** illustrates a simple circuit that uses a 2N3904 NPN transistor as a low-side switch to control a P-channel IRF9520 MOSFET. To give you a practical circuit, I chose a 74LS04 5-volt inverter IC to drive the 2N3904 transistor base.

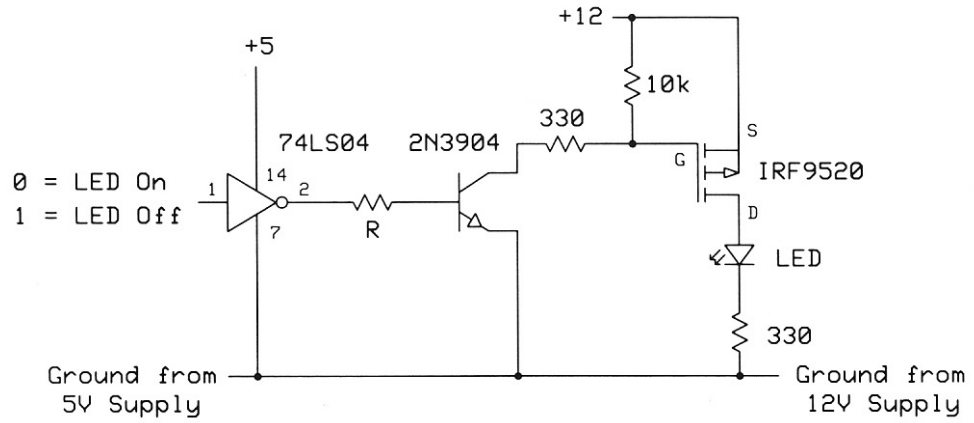


Figure 23.1.
A basic circuit that uses a logic signal to control a MOSFET used as a high-side switch. The 74LS04 inverter provides a base current to the 2N3904 transistor to turn it on. The transistor then creates a low-impedance path to ground, which causes the MOSFET to turn on.

A close look at the circuit from the 74LS04 input on the left to the LED on the right will help you understand how to use this configuration of components. You can find similar circuits in application notes, in Internet discussions, and so on. Before you use one of these circuit you should know how people design them based on device specifications and a few calculations. All too many hobbyist and experimenter circuits simply cobble together components until a circuit works... somewhat. A methodical approach to circuit design provides the best results.

Start at the inverter, which accepts a logic-1 input and produces a logic-0 output, and *vice versa*. **Table 23.1** lists the specifications for the 74LS04 inverter output. Note the 74LS04 inverter needs a +5-volt power source. And that source and the +12-volt power supply require a common ground, as shown in **Figure 23.1**.

Table 23.1. Output specifications for a 74LS04 inverter IC.

Parameter	Abbreviation	Minimum	Typical	Maximum	Units
Voltage out when output = logic 1 (output high)	V_{OH}	2.4	3.4		V
Voltage out when output = logic 0 (output low)	V_{OL}		0.2	0.4	V
Current sources when output = logic 1 (output high)	I_{OH}			-0.4	mA
Current sunk when output = logic 0 (output low)	I_{OL}			16	mA

Does a logic-1 output from the 74LS04 inverter provide a voltage high enough to control the 2N3904 transistor? You need information from the datasheet for both these devices to get an answer. A logic-1 output from a 74LS04 inverter typically produces 3.4 volts (V_{OH}, voltage output high). That voltage exceeds the

maximum base-to-emitter junction voltage (0.85 volts, V_{BEsat}) for the 2N3904 transistor. As a result, current can flow from the logic-1 output at the inverter to the 2N3904 base. (Find complete datasheets on the Internet.)

Likewise, a logic-0 output provides a low-enough voltage (0.4 volts) to ensure no current will flow from the inverter output to the transistor's base. Also, the 2N3904 datasheet shows it can withstand a 30-volt difference between the collector and emitter, so it will handle the 12-volt power applied to the MOSFET. So far, the circuit looks like it will work. But before we can construct this circuit we must know if the 74LS04 inverter can supply enough *current* to control the 2N3904 base.

The current output-low (I_{OL}) value for the 74LS04 shows a logic-0 output can *sink* as many as 16 mA, but we don't need to sinking current from the transistor base, so this specification doesn't matter. On the other hand, the 74LS04 must *source*, or supply, a small current to the 2N3904 base to turn the transistor on. The current output-high (I_{OH}) value shows the 74LS04 can source only 0.4 mA from an output in the logic-1 state. Will the 0.4 mA from an inverter logic-1 output suffice to turn the transistor on fully? Let's find out.

First, look at the load the transistor has attached to its collector and the amount of current the transistor will pass. The P-channel MOSFET draws an insignificant amount of current, so consider the 10-kohm and the 330-ohm resistors as the only load. How much current will pass through them?

$I = E \,/\, R$ or $I = 12\ volts \,/\, (10{,}000\ ohms + 300\ ohms)$

$I = 1.16\ mA$

The data sheet for a 2N3904 NPN transistor shows a gain of 70 for a collector current of 1 mA and a 1-volt collector-to-emitter voltage. For a bipolar transistor, h_{FE}, or beta, comes from the following equation. Remember, h_{FE} has no units because it expresses a ratio:

$h_{FE} = Collector\ current \,/\, Base\ current\ = I_C / I_B$

$h_{FE}\ =\ 70\ =\ 1.16\ mA \,/\, I_B$ or $I_B\ =\ 1.16\ mA \,/\, 70$

$I_B\ =\ 0.0229\ mA$ or $22.9\ \mu A$

The 2N3904 transistor requires only a small base current – less than the 0.4 mA, or 400 µA, provided by a logic-1 output from the 74LS04. Because a logic-1 typical supplies a 3.4-volt output, what size resistor do we need between the 74LS04 output and the 2N3904 base to limit the base current to 22.9 µA?

$R\ =\ E \,/\, I$, so $R\ =\ 3.4\ volts \,/\, 22.9\ \mu A$ or $3.4\ volts \,/\, 0.0000229\ A$

$R\ =\ 148{,}000\ ohms$

I didn't account for the base-to-emitter voltage drop in these calculations, but they give us good approximations to start with. You could get close with a 130 kohm resistor. Transistor and resistor values don't always conform to "typical" specifications, so this circuit could safely use a 120-kohm or even a 100-kohm resistor between the 74LS04 output and the 2N3904 base. When I used a 100-kohm resistor in place of the "R" resistor shown in **Figure 23.1**, I measured a 1.17 mA collector current through the 2N3904, and a 44 µA base current from the 74LS04 to the 2N3904.

OPTIONAL: You may wire this circuit and test it. If you do, remember for a logic-1 input to the circuit connect the 74LS04 inverter input (pin 1) to +5 volts, or connect it to ground for a logic-0 input.

In Experiment 22 you learned about the need for a pullup resistor on the gate of a P-channel MOSFET. The 10-kohm resistor shown in **Figure 23.1** works well in the LED-control circuit. But why does the circuit

include a 330-ohm resistor between the 2N3904 collector and the IRF9520 gate? A MOSFET gate draws very little current, so the circuit doesn't need a current-limiting resistor here. The 330-ohm resistance still serves a useful purpose: It limits the inrush or outrush current that flows between the transistor's collector and the MOSFET gate.

A rapid current change can cause problems in circuits. In some cases, the current can exceed the capability of the driving device to absorb it. The fast-rising edge of a current pulse also can cause "ringing" and overshoot as shown in **Figure 23.2**. By using a small resistance to impede the current flow slightly, engineers reduce signal ringing that might disrupt other circuits. Many experimenter circuits available in magazines and on the Internet do not include a resistor between a driver and a MOSFET gate. Unless you use a special MOSFET-control IC, I recommend you always include a small resistance as shown in **Figure 23.1**. My circuit happened to use 330 ohms, but a 100-ohm resistor would work well instead.

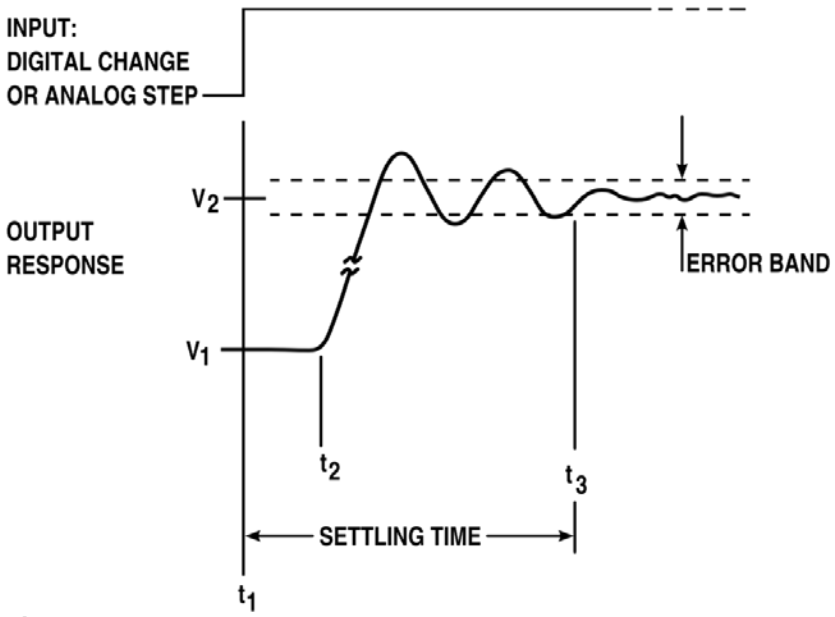


Figure 23.2.
The signal shown here has a fast rise time that can cause the signal to overshoot its normal output voltage (V_2). This type of signal also can create ringing, or a damped oscillation, on a conductor. Ringing signals can disturb nearby circuits or generate signals that affect other electronic devices.
(Courtesy of Wikipedia, see Ref. 1.)

In the next step you will learn about a similar type of circuit for a low-side N-channel MOSFET switch.

The 5-volt 7400 family of logic devices includes several that have an *open-collector* output. That means the collector of the output transistor within the IC does not connect to anything except a pin on the IC package. Unlike other 5-volt logic ICs that produce a logic-1 (3.4 volts) or a logic-0 (0.2-volt) output, an open collector provides the same connection to ground for a logic-0, but instead of a logic-1 voltage output the collector "floats." That means no internal circuit causes the collector to provide a logic-1 voltage. **Figure 23.3** shows the internal 7407 open-collector circuit and six such circuits that operate independent of each other in a 14-pin package. The arrow in the diagram identifies the open-collector on the output transistor. You may ignore the other components. Circuit designers might refer to an IC with open-collector outputs as a *buffer*. Typically a buffer circuit amplifies current or voltage.

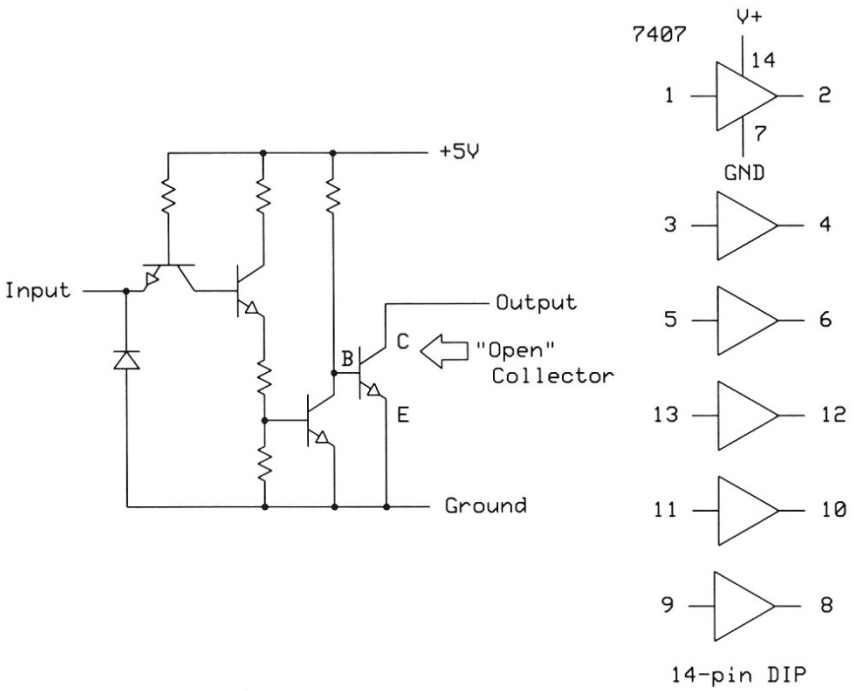


Figure 23.3.
This diagram for one of the six circuits in a 7407 IC uses an arrow to identify the open-collector output. When you apply a logic-0 signal to the input, the output connects the collector to ground through the right-most transistor. A logic-1 signal at the input turns off the output transistor and the collector "floats." A pull-up resistor usually connects the collector to a power bus. The symbols to the right of the circuit represent all six buffers in a 7407 IC, and they include the power (pin 14) and ground (pin 7) connections.

The 7407 buffer has two characteristics that make it a good choice to control a MOSFET directly. A 7407 can sink to ground as many as 40 mA and the collector can withstand a 30-volt signal. Thus it will easily replace the 2N3904 shown earlier in **Figure 23.1**. You simply treat a 7407 buffer as a combination of a 7404 inverter IC and a 2N3904 transistor. **Figure 23.4** provides a working circuit.

OPTIONAL: You may wire and test this circuit. Remember to connect the 7407 input to +5 volts for a logic-1 signal and to ground for a logic-0 signal.

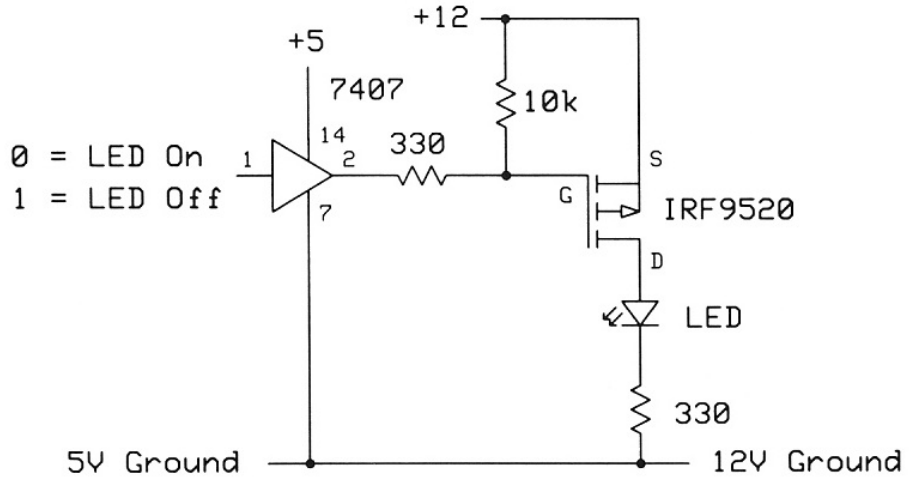


Figure 23.4.
Circuit diagram for a high-side switch that uses a 7407 buffer IC to directly drive an IRF9520 P-channel MOSFET. Although the output transistor in the buffer IC can withstand a 30-volt signal, the IC itself requires a 5-volt (maximum) power source. Note the common ground between the 5- and 12-volt power supplies.

The 7407 output connects to a 330-ohm resistor in the gate-control circuit for the same purpose explained earlier; it limits inrush current and helps prevent ringing and overshoot on the connection.

Can a 7407 buffer also control an N-channel MOSFET configured as a low-side switch? Yes. The circuit in **Figure 23.5** easily controls the N-channel MOSFET gate voltage. This circuit operates in the same way as the circuit in **Figure 23.22**. The 10-kohm resistor pulls the gate up to +12 volts, which turns on the MOSFET and thus the LED. When the 7407 open-collector output grounds the gate, the MOSFET turns off and so does the LED. As an option, you may build this circuit and test it. Again, remember for a logic-1 input connect the buffer input (pin 1) to +5 volts for a logic-1 signal, or connect it to ground for a logic-0 signal.

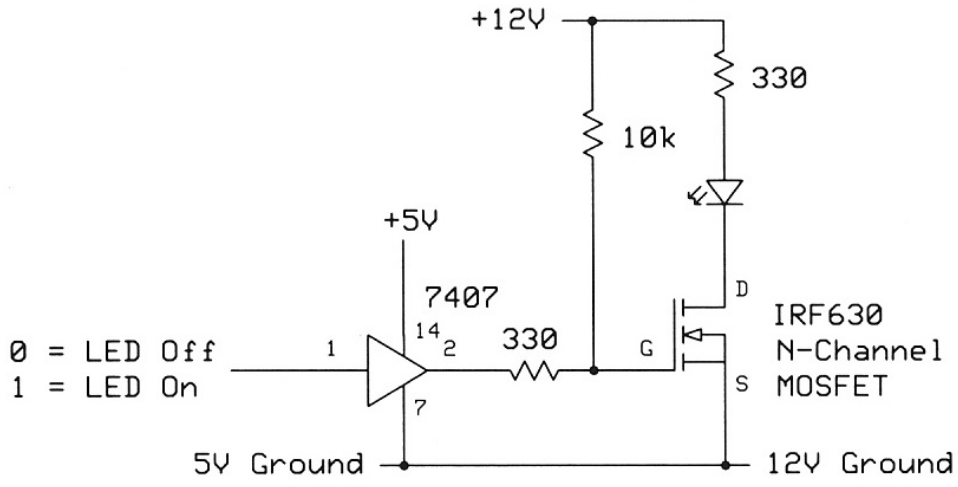


Figure 23.5.
Control of an N-channel MOSFET by a 7407 buffer IC creates a low-side switch for an LED.

Look again at the information in **Figures 23.4** and **23.5**. Do you see a difference in how the circuits work? Look in the Answers section at the end of this experiment.

Motor Control with MOSFETs

Some circuits need a high-side and a low-side switch to control a device such as a 2-terminal 2-color LED. Current through the LED in one direction makes the LED turn on red, and current in the reverse direction causes the LED to glow green. In this situation, one MOSFET would source current to the LED and another MOSFET would sink current from it. MOSFETs control DC motors in a similar fashion. A 12-volt power supply connected to such a motor turns the rotor in one direction. Reverse the polarity of the supply – swap the positive and negative connections to the motor – and current flows in the opposite direction. The rotor turns in the opposite direction. Thus to run a motor in both directions a circuit must reverse the power connections.

Four MOSFETs, arranged as an "H bridge" circuit can control a reversible DC motor. **Figure 23.6** shows the basic configuration that includes two high-side and two low-side switches and the two current paths through the motor. When another circuit (not shown) turns on only MOSFETs Q1 and Q4 the motor runs clockwise. Turn on only MOSFETs Q2 and Q4 and the motor turns counterclockwise.

CAUTION: The circuit that controls the four MOSFETs MUST NOT turn on Q1 and Q3 or Q2 and Q4 simultaneously. Those connections would create a short circuit between the 12-volt source and ground, and might destroy the transistors.

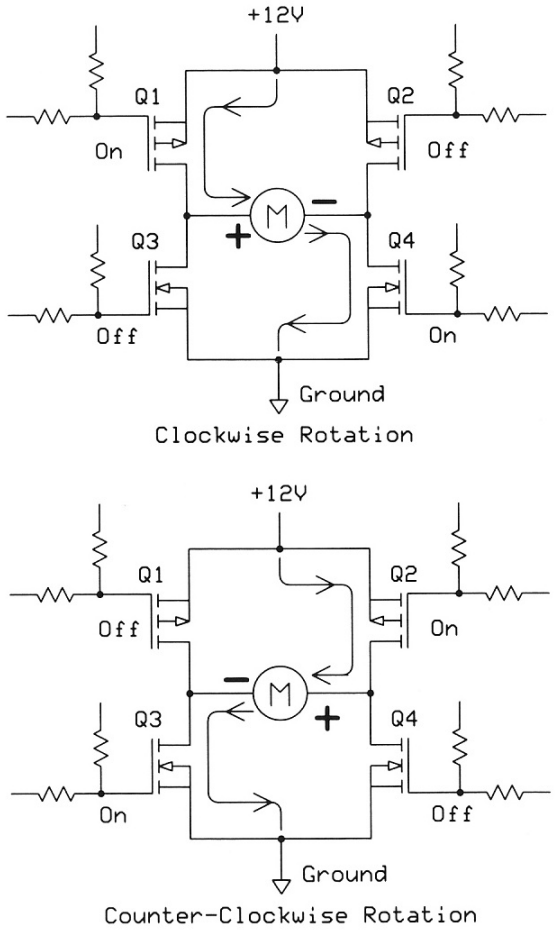


Figure 23.6.
Current paths through the MOSFET H-bridge circuit control the direction of rotation for a DC motor. Other paths exist, but they can cause a catastrophic short circuit or turn off the motor.

Given the H-bridge circuit, how can a logic circuit or an MCU control the MOSFETs to operate the motor properly? Instead of trying to drive the MOSFET gate inputs directly, use the 7407 open-collector buffer circuits shown previously in **Figures 23.4** and **23.5**. **Figure 23.7** shows a complete circuit for a 12-volt DC motor control.

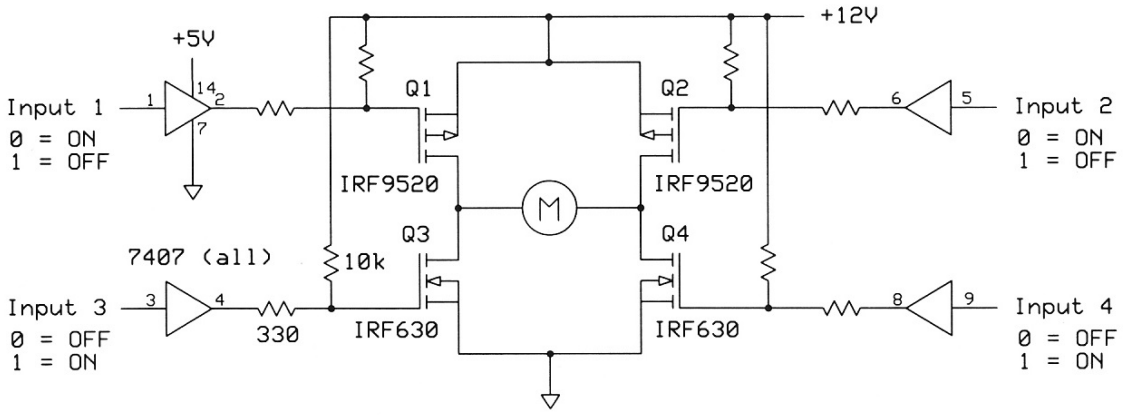


Figure 23.7.
A complete H-bridge circuit that includes four 7407 open-collector drivers. The inputs on the 7407 accept 5-volt logic levels from an Arduino or other 5-volt MCU.

If you don't have a small DC motor on hand, you can find many inexpensive types available from suppliers such as All Electronics, Marlin P. Jones and Associates, and Jameco Electronics. Choose a motor that will not overload your power supply. If your power supply can deliver 12 volts at 2 amps, I suggest a motor that draws less than one ampere.

Given the circuit in **Figure 23.7**, how would you start to construct it on a breadboard? I recommend a step-by-step approach that lets you test portions of the circuit before you assemble them all for the complete motor controller. You can perform the lettered steps that follow if you wish, but it's not necessary to do so to learn more about MOSFETs.

a. Build each high-side and low-side MOSFET switch separately and one at a time. Test each switch with an LED and current-limiting resistor as the load and ensure the switches work as expected. Colored wires can help follow circuits, so I use them extensively on breadboards. See **Figures 23.4** and **23.5** for example circuits.

b. Turn on power and connect each 7407 input to a logic-1 or a logic-0 signal to test each switch. In your lab notebook, record the logic-input values and how they affect the attached LED for the high-side and low-side switches.

c. Turn off power. Remove the LEDs and the 330-ohm resistors from the four test circuits. Now the drain lead on each MOSFET switch should have nothing connected to it. Select one high-side switch circuit and a low-side switch circuit. Connect the drain lead on the high-side switch to the drain lead on the low-side switch (**Figure 23.8**) to create one side of the H-bridge circuit.

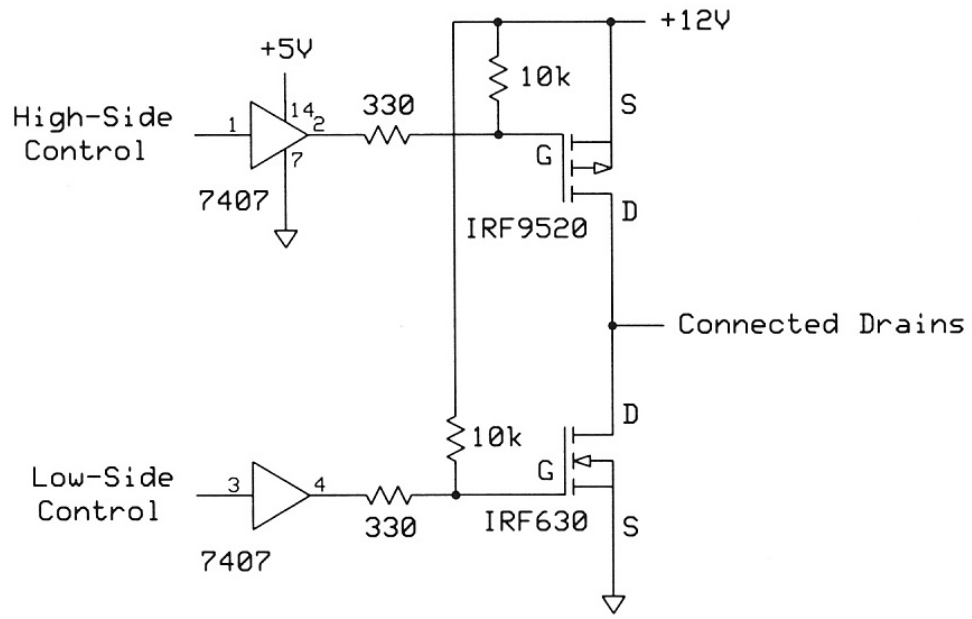


Figure 23.8.
Connect the drain leads on a low-side and a high-side MOSFET switch so you can test one half of an H-bridge circuit for motor control.

d. Use the information written in your lab notebook during step b to set the inputs of the two 7407 buffers so only the high-side switch turns on. Connect a voltmeter between the connected drains and ground (**Figure 23.9a**). Apply power. You should measure about 12 volts at the connected drains. If either MOSFETs get hot, you probably have a short circuit. Turn off power!

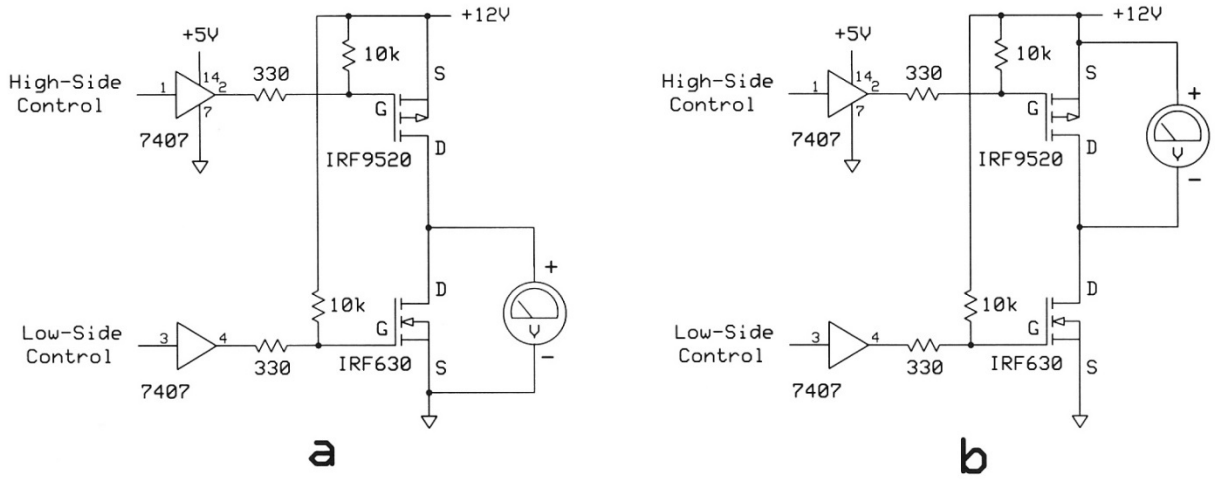


Figure 23.9.
Test circuits for **a)** the high-side switch and **b)** for the low-side switch. Note the polarity of the meter leads.

e. Turn off power and change the logic signals at the 7407 buffer inputs so only the low-side MOSFET switch turns on. Connect the positive meter lead to the +12-volt power bus. Connect the negative meter lead to the drain connection (**Figure 23.9b)**. Turn on power. You should measure about 12 volts because the low-side switch connects the negative meter lead to ground to complete the meter circuit from the 12-volt bus. If either MOSFETs get hot, you probably have a short circuit. Turn off power!

f. Turn off power and repeat steps c through e for the remaining low-side and high-side MOSFET switches. After you have wired and tested each side of an H-bridge circuit, insert two LEDs, one green and one red between the drain connections on each side of the H-bridge circuit as shown in **Figure 23.10**. The LEDs provide a simple way to test the entire bridge circuit. This circuit uses only one current-limiting resistor for both LEDs. Do you know why?

Current can flow only in one direction at a time, so either the red or the green LED – or neither – will turn on. One 470-ohm resistor suffices for the LED that lights.

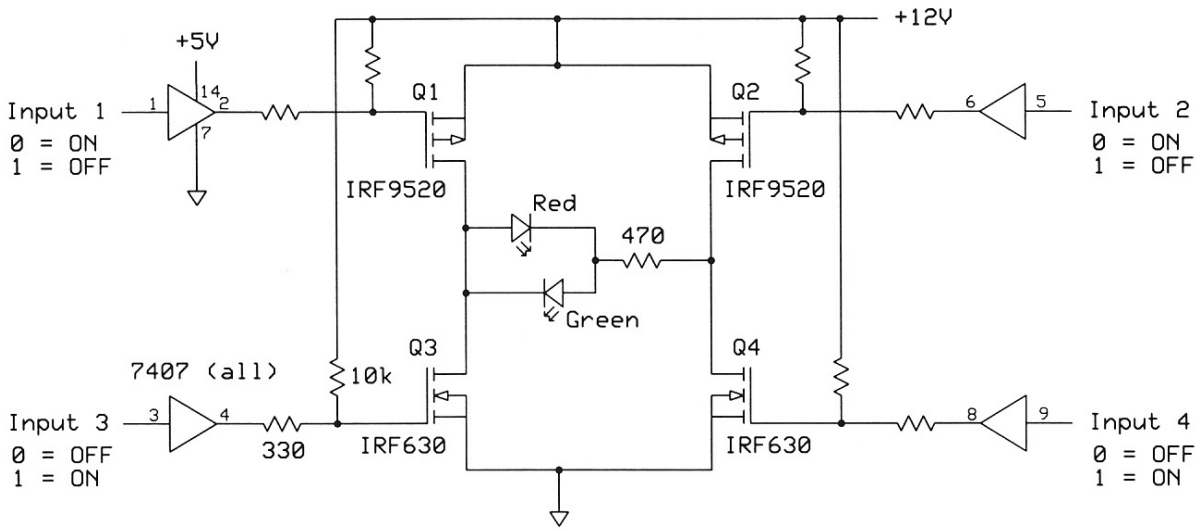


Figure 23.10.
A test circuit for a MOSFET H-bridge circuit. Instead of using a motor, this circuit uses two LEDs so you can determine the direction of current flow. Use the same resistor values as shown for those unlabeled in the diagram. You need 4 100K- and 4 330-ohm resistors.

Do not apply power to the completed test circuit. First you must determine the logic levels needed at the four 7407 buffer inputs, labeled 1 through 4, to light each LEDs or to turn both off. Several input settings will cause a short circuit. If you turn on both Q1 and Q3, for example, you might destroy those two MOSFETs. The four 7407 inputs give us 2^4 or 16 combinations of logic signals for the buffer inputs. Based on your observations in step **b**, you'll next determine the combinations needed to turn on each LED, turn off both LEDs, do nothing, or cause a short circuit.

Table 23.2 provides all 16 logic states for the 7407-buffer inputs. Use information in **Figure 23.10** to determine how each state affects the circuit and write the outcome in the Action column. Hint: Remember to consider ALL four MOSFET switches, not just those on one side of the H-bridge circuit. The table includes two reported actions to give you a quick start.

Table 23.2. The 16 logic states that will affect the DC-motor H-bridge circuit.

7407 Input States				Action
Left Side		Right Side		
Input 1 High Side	Input 3 Low Side	Input 2 High Side	Input 4 High Side	
0	0	0	0	Q1 and Q2 on, no action
0	0	0	1	SHORT CIRCUIT, Q2 to Q4
0	0	1	0	

0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Table 22.3 lists the 16 input conditions for the four inputs shown in my circuit (**Figure 23.10**), and the Action column shows what happens for each. Seven settings cause short circuits, which you must avoid. Two settings control the LEDs and one setting turns off all four MOSFETs.

Table 23.3. Input settings for MOSFET H-bridge control of a small DC motor.

7407 Input States				Action
Left Side		Right Side		
Input 1 High Side	Input 3 Low Side	Input 2 High Side	Input 4 Low Side	
0	0	0	0	No action
0	0	0	1	SHORT CIRCUIT
0	0	1	0	No action
0	0	1	1	Red LED on
0	1	0	0	SHORT CIRCUIT
0	1	0	1	SHORT CIRCUIT
0	1	1	0	SHORT CIRCUIT
0	1	1	1	SHORT CIRCUIT
1	0	0	0	No action
1	0	0	1	SHORT CIRCUIT
1	0	1	0	No action, power off
1	0	1	1	No action
1	1	0	0	Green LED on
1	1	0	1	SHORT CIRCUIT
1	1	1	0	No action
1	1	1	1	No action

If you plan to use software and four output pins on an MCU to control a motor or LEDs, your code *must* ensure the short-circuit pattern of control bits NEVER gets sent to the MOSFET switches. Of course your patterns of bits might differ from mine (**Table 23.3**). Always go through your circuit diagram and look for MOSFET-control situations or settings that could create short circuits. Also ensure only your control code can alter the outputs for the MOSFET-driver ICs.

Now that you know the settings that control the red and green LEDs, use jumper wires and configure the 7407 inputs to turn on one LED. Turn on power. Did the correct LED turn on? Turn off power and try the setting for the other LED. Did it turn on, too?

After you complete testing, the H-bridge circuit – without the LEDs – will drive a small 12-volt DC motor clockwise or counterclockwise, or turn it off. I used the settings in **Table 23.3** to turn on and run a small motor for several hours. The motor warmed a bit, but the MOSFETs remained cool.

OPTIONAL: Given the test circuit in **Figure 23.10**, could you modify it to indicate short circuits? If so, what changes would you make and what components would you add or remove? Find out in the Answers section at the end of this experiment.

MOSFETs and Low-Voltage MCUs

Many years ago the electronics industry standardized on families of logic ICs that require a 5-volt power supply. The outputs on IC X could connect to several inputs on IC Y, and the outputs on IC Y could connect to the inputs on ICs Z and G, and so on. This standardization let an IC manufacturer offer many parts that worked with those of other suppliers. A 2-input NAND gate (SN7400) from Texas Instruments, for example, would work with the 7400-series ICs from National Semiconductor, Signetics, Motorola, Fairchild, and other companies.

The demand for portable electronic equipment and consumer products forced semiconductor companies to look for ways to make ICs smaller and to reduce their power use. Now we have devices that use voltages between 2.7 and 3.3 volts, and some devices that operate at 1.8 volts. The choice of lower operating voltages also relates to the semiconductor physics and the types of practical manufacturing processes semiconductor companies use.

Earlier circuits in this experiment used 5-volt ICs such as the 74LS04 inverter and the 7407 open-collector buffer. So a 5-volt MCU board such as an Arduino could control those MOSFET circuits. But how can lower-voltage logic families and MCUs such as the Parallax Propeller and Texas Instruments ARM Cortex-M3 control MOSFETs? Semiconductor manufacturers created new types of N- and P-channel MOSFETs that accept signals from low-voltage ICs. Now, a Propeller board, an ARM mbed module, a Raspberry PI board, or a BeagleBone board can directly control MOSFETs.

When you need MOSFETs that work with low-voltage logic signals, look for the designation "logic level" or "logic-level gate" in a MOSFET datasheet or catalog description. A search for "MOSFET" on the DigiKey Web site yielded more than 15,000 parts, thankfully not all displayed at once. Distributors' Web sites let people narrow their search by choosing requirements from lists of characteristics, or parameters. The choices include FET type, package type, drain-to-source voltage, and so on. You also can choose MOSFETs with special features that include "Current Sensing," "Enhancement Mode," and "Logic Level Gate." I found over 400 MOSFETs with logic-level gate inputs and through-hole pins that simplify breadboarding. (Not all distributors' Web sites offer the option to sort by gate-input type, though.)

Three representative MOSFETs have the following characteristics:

- Infineon SPP15P10PL, P-channel, -100 V_{DS}, and $V_{GS(th)}$ between -1.0 and -2.0 volts
- Fairchild NDP6020P, P-channel, -20 V_{DS}, and $V_{GS(th)}$ between -0.4 and -1.0 volts
- Vishay IRL620, N-channel, 200 V_{DS}, and $V_{GS(th)}$ between 1.0 and 2.0 volts

Remember that P-channel MOSFET measurements use the source connection as the voltage reference, so the V_{DS} and $V_{GS(th)}$ values have a minus sign.

The Propeller MCU operates from a 3.3-volt power source and it has a minimum 2.85-volt output for a logic-1, and a maximum 0.4-volt output for a logic-0. Specifications note a 40-mA maximum current on an I/O pin. Can the Propeller outputs control the logic-level gates on the three MOSFETs? Yes.

You can build the circuits that follow and test them with a Parallax Propeller P8X32A board, or other MCU board that operates at 3.3 volts. Consider the circuit construction and testing optional, but I recommend you read the sections that follow.

Now you can use an IRL620 "Logic-Level Gate Drive" MOSFET to control 10 LEDs. The diagram in **Figure 23.11** shows the needed connections. I used an LED on the Propeller P27 pin simply to test the proper timing of the test program. Remember to include a ground connection between your 12-volt power supply and the ground pin on the Propeller board.

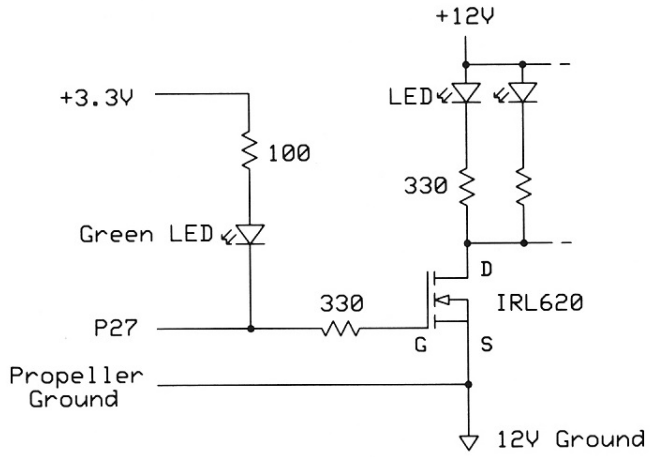


Figure 23.11.
Schematic diagram for the LED MOSFET switch controlled by the P27 output from a Propeller P8X32A board. For the sake of clarity, this diagram does not include all 10 of the LEDs and resistors used as the load for the MOSFET. Your circuit should include them on a breadboard.

Build the circuit in a breadboard and connect it to the P27 output on a Propeller P8X32A. Ensure the circuit connects to both the +12-volt power-supply ground and the Propeller ground pin. To operate properly the circuit requires this common ground.

Connect a Propeller P8X32A MCU board to your computer with a USB cable and start the Propeller Tool program. Then load **Program 23.1** into the Propeller tool and transfer it to the MCU (press the F10 key). Ensure you have the Timing.spin software in your working directory. You will find it in the Experiment 23 software folder, along with **Programs 23.1** and **23.2**. Turn on power to your breadboard. The green LED that connects to the MOSFET should turn on or off every 10 seconds. What logic state at the P27 output turns this LED on? The 10 LEDs also should turn on or off every 10 seconds. What logic state at P27 turns these LEDs on?

Program 23.1.
```
''*****************************************************
''*  Program 22.1, Direct drive of an N-channel MOSFET
''*  by an output pin on the Propeller P8X32A board.
''*  Jon Titus, 07-22-2014 Rev. 1
''*  Copyright 2014
''*  Software uses the I/O connection at P27 on the
''*  Propeller board.
''*****************************************************
OBJ
  delay     : "timing"            'timing.spin file

CON
```

```
  _CLKMODE      = XTAL1 + PLL4X     'set MCU clock operations
  _XINFREQ      = 5_000_000         '5MHz Crystal

PUB Main
 'A simple output program that changes the state of the P27
 'I/O pin (set for output). A 10-second time delay lets you
 'see the action at the P27 pin and on the 10 LEDs controlled
 'by the MOSFET.

DIRA[27] := %1          'Set P8X32A board LED pin as output

repeat                          'This main loop runs "forever"
  OUTA[27] := %1                'Make I/O pin P27 logic-1
  delay.pause1ms(10000)         '10-sec delay
  OUTA[27] := %0                'Make I/O pin P27 logic-0
  delay.pause1ms(10000)         '10-sec delay

' –  – -end Program 23.1 –  – -
```

A logic-0 from the Propeller P27 output turns on the green LED because the MCU sinks current to ground through the LED. The 10 LEDs turn on when the P27 output becomes a logic-1. The IRL620 N-channel MOSFET turns on and connects the 10 LEDs to ground when the P27 output raises the gate above the $V_{GS(th)}$ of 1.0 to 2.0 volts. The Propeller's logic-1 output provided 3.25 volts in my circuit, which exceeded the minimum $V_{GS(th)}$. The MOSFET remained cool.

Now remove, or comment-out, the two delay statements in **Program 23.1**. What will happen when you run the modified program? The LEDs appear dimmer. Do you know why?

The modified program turns the LEDs on and off so quickly that human eyes cannot see them change state. Because the on and off periods have equal length, current goes through the LEDs only half the time, which causes them to appear dimmer than when they turned on fully. **Program 23.2** runs a pulse-width-modulation (PWM) routine similar to the one used in Experiment 6. After you load this program into the Propeller MCU, open the Parallax Serial Terminal window and click the Enable control. One at a time type in several values between zero and 100 to change the PWM on period from 0- to 100-percent. Values outside the 0-to-100 range display an error message in the PST window. (If the PST window cannot communicate with your Propeller board, ensure the "Com Port" setting matches the one used by the P8X32A board. Also, the PST "Baud Rate" setting should equal 9600.)

Program 23.2.

```
{{
''*********************************************
''*  MOSFET Program 23.2
''*  Author: Jon Titus 07-22-2014 Rev. 1
''*  Copyright 2014
''*  Released under Apache 2 license
''*  Ten LEDs dim or brighten depending on the
''*  value entered from the PC keyboard via the
''*  Parallax Serial Terminal. Rejects values
''*  outside the range 0 to 100 and prints
''*  an error message. The controlling circuit
''*  uses an IRL620 MOSFET with an added 330-ohm
''*  gate resistor. PWM output on pin P27
''*  Added statements set PWM period.
''*  pwm1.SetPeriod(1_600_000) = 2 msec.
''*********************************************
```

```
}}

CON _clkmode = xtal1 + pll16x     'Set MCU clock operation
    _xinfreq = 5_000_000          'Set for 5 MHz crystal
    max_duty = 100                'Maximum 100% duty cycle
    pwm1pin  = 27                 'Control LED at P22


VAR byte ser_data                 'Byte for PWM duty cycle

OBJ
  pwm1  :  "pwmasm"               'Ensure you have pwmasm.spin
                                  'in your working directory
  Serial : "Extended_FDSerial"    'Serial communication program
                                  'also in working directory

PUB Start                         'Main program starts here
  pwm1.start(pwm1pin)             'Startup for LED at P27
  pwm1.SetDuty(0)                 'Start duty cycle at 0%
  pwm1.SetPeriod(1_600_000)       'Set period for 20 msec.

  Serial.start(31,30,0,9600)      'Set serial communications
                                  'pin P31 for receive
                                  'Pin 30 TX, 0 = reset options
                                  '9600 = baud rate. Must be the
                                  'same as set for the Parallax
                                  'Serial Terminal (PST)
  Serial.RxFlush                  'Clean out receiver buffer

  repeat                          'repeat this loop "forever"
    ser_data := Serial.RxDec      'Get received data
                                  'convert to decimal
                                  'Check for value 0 to 100
    if ((ser_data <0) OR (Ser_data >100))
        Serial.str(String("Only values between 0 and 100, please."))
        Serial.tx(13)
    else
        pwm1.SetDuty(ser_data)    'Set PWM duty cycles
                                  'to decimal value
    waitcnt(1_000_000 + cnt)      'short delay between changes

' –  – -end Program 23.2 –  – -
```

Suppose the circuit shown in **Figure 23.11** had included a 10-kohm *pullup resistor* between the gate and the +12-volt bus. (DO NOT add this resistor to the circuit.) 1. What voltage would appear on the Propeller P27 pin? 2. If you disconnect the Propeller MCU from the MOSFET gate pin, what would happen?

- **Answer 1**. The pullup resistor would put about +12 volts on the Propeller pin, which would probably damage it. The specifications for the MCU indicate no pin should have a voltage higher than the supply voltage (+3.3 volts) plus 0.3 volts, or 3.6 volts.

- **Answer 2**. If you could use a pullup resistor, the MOSFET would turn on when you disconnect the MCU pin from the circuit. The 10-kohm resistor would cause the gate voltage to rise above the IRL620 MOSFET $V_{GS(th)}$ of 1.0 to 2.0 volts. As a result, without any other voltage applied to the gate, the IRL620 MOSFET would turn on. That's not a condition we want.

Instead, you might include a 10-kohm *pull-down resistor* between the gate and ground. This resistance ensures the otherwise-unconnected gate does not reach the $V_{GS(th)}$ voltage. **Figure 23.12** shows the circuit with an added 10-kohm pull-down resistor. You could use pull-down resistors in place of the pullup resistors on the gates of the two N-channel MOSFETs in the H bridge circuit shown previously **in Figure 23.7**. The pull-down resistors would ensure the associated MOSFETs remain off when you have nothing connected to their gate inputs.

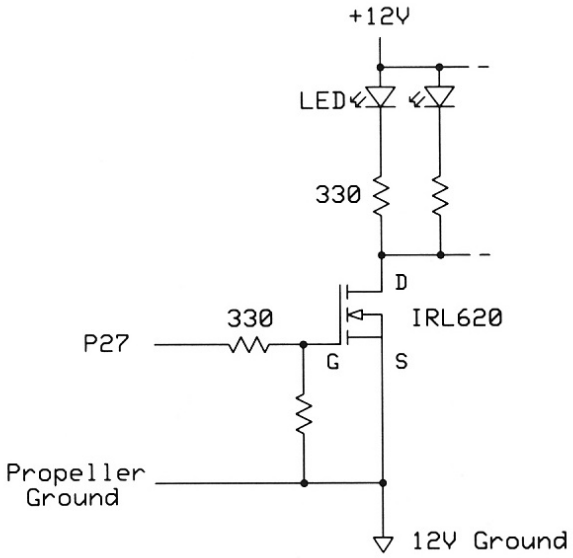


Figure 23.12.
The added 10-kohm pull-down resistor between the gate and ground ensures the N-channel MOSFET remains off when the gate input on the left end of the 330-ohm resistor stays unconnected.

How to Properly Mount MOSFETs and Avoid Short Circuits

Because MOSFETs can handle large currents, they often require a *heat sink* to dissipate energy. Without a heat sink, a MOSFET could get too hot and operate outside its specified temperature range. Sometimes a hot MOSFET will burn out due to thermal runaway. You might think, "I can use the hole in the TO-220 tabs to bolt MOSFETs to a piece of metal or to a heat sink." Unfortunately, that approach can cause short circuits because the metal tab often has an internal electrical connection to one of the MOSFET leads.

In an IRF520 MOSFET, for example, the metal tab, or back of the case, connects to the drain terminal. Many semiconductors encased in TO-220 packages electrically connect the the metal tab and the middle pin of the three signal connections. You can confirm this connection in two ways. First, set up a simple test circuit and compare the voltage on each pin to the voltage on the metal tab. Second, read the MOSFET manufacturer's data sheet. Sadly, the information often gets buried and proves difficult to find. The IRF520 datasheet only notes the connection in a list of specifications: "Maximum Junction-to-Case (Drain)." Nothing specifically calls attention to the tab-to-drain electrical connection.

Several companies provide components you can use to electrically insulate TO-220 packages from heat sinks. A thin insulating wafer between the TO-220 package and the heat sink does the trick. Engineers often place a small amount of thermally conductive grease on the heat sink and on the back of the TO-220 case to fill imperfections in the surfaces. A machine screw placed through the hole in a TO-220 tab fastens it to the heat sink. **Figure 23.13** shows the components used to secure a TO-220 transistor package to a heat sink.

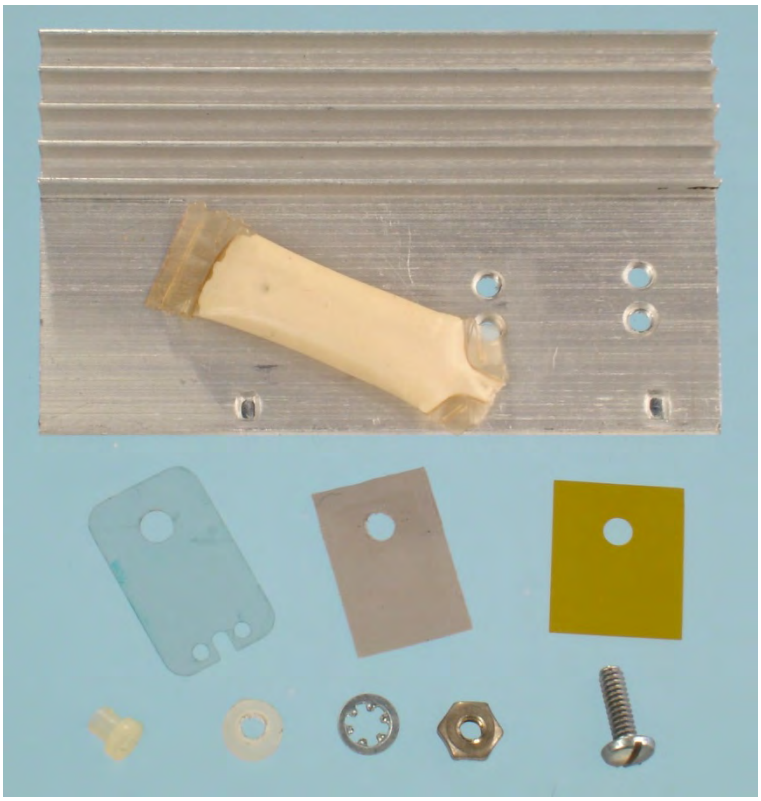


Figure 23.13.
From the top of this photograph, mounting components include an aluminum heat sink with fins and heat-sink grease, three types of insulating wafers (mica, silicone, and Kapton), and mounting hardware (two types of plastic insulating collars, a 4-40 locknut, hex nut, and screw. A metric M3 screw, lockwasher, and nut would also work.
(E. I. du Pont de Nemours and Company owns the Kapton trademark.)

But the screw needs insulation, too. You can purchase small plastic collars that fit in the hole in the tab and insulate the screw from the tab. Then the screw goes into the collar and attaches to the heat sink. A heat sink could have a threaded hole or a bored hole for the screw. When the screw goes through the heat sink, the assembly needs a lockwasher and a nut on the opposite side. Jameco sells a mounting kit (part no.34121) that comprises 10 sets of the mounting components just described. You also can buy "thermal-management" grease and heat sinks separately. If you want a simpler heat sink arrangement, try clip-on heat sinks that don't require a screw. These heat sinks still might need an insulating wafer between the back of a TO-220 package and the metallic heat sink.

CAUTION: I have seen TO-220 mounting diagrams that show a screw that goes into a plastic collar that then fits into the back of a heat sink. The threaded part of the screw goes through an insulating pad, through the hole in a TO-220 tab. Finally, a lockwasher and a nut hold the transistor in place. Mechanically, this arrangement works, but the head of the screw makes contact with the TO-220 tab through the lockwasher and the nut. The unprotected screw head could inadvertently touch another conductor such as a screwdriver or a meter probe, which could lead to disastrous consequences. Always place the screw and the collar through the TO-220 tab first.

Figure 23.14a shows three examples of MOSFETs in TO-220 packages attached to an aluminum heat sink. All three devices use a heat-conducting pad to electrically isolate the TO-220 tab from the heat sink. The left-most MOSFET uses a plastic collar and a screw that goes into a hole threaded into the aluminum. The collar prevents the screw from contacting the metal tab. The middle MOSFET also uses a plastic collar. The machine screw goes through the heat sink. A nut on the other side tightens the TO-220 package in place. The right-most package has a nut and a lockwasher attached directly to the TO-220 metal tab.

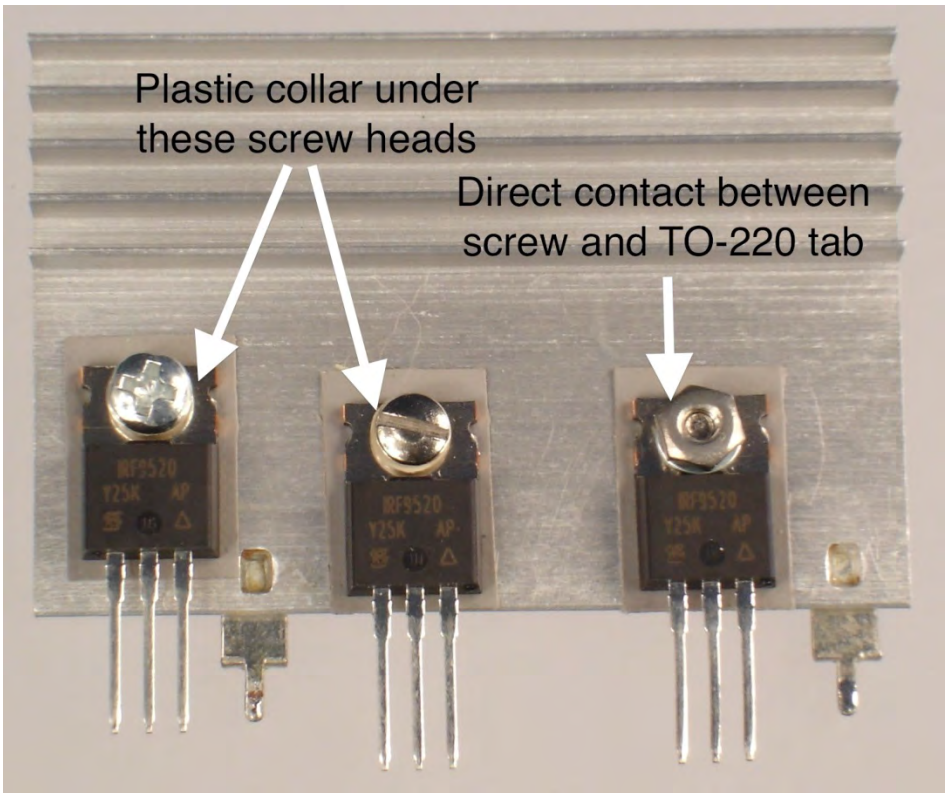


Figure 23.14a.
Three types of mounting arrangements for MOSFETs in TO-220 packages with metal tabs.

The photo in **Figure 21.14b** shows the back side of the heat sink when flipped vertically. The left-most screw holds fast in the threaded hole provided in the heat sink. The middle screw takes a 4-40 locknut and nut. Although the screws for these two MOSFETs make direct contact with the aluminum, the plastic collar underneath the screw heads keeps them electrically isolated from the MOSFET metal tabs.

The right-most mounting arrangement has a problem. Even though a collar under the screw head insulates the screw from the heat sink, the nut and lockwasher on the opposite side contact the MOSFET's metal tab. Thus, the screw head connects to the tab and we consider it "hot," in the sense that it can have a high voltage on it. Inadvertently touching the screw head with a tool, wire, or instrument could cause a dangerous and damaging short circuit. DO NOT use this mounting technique.

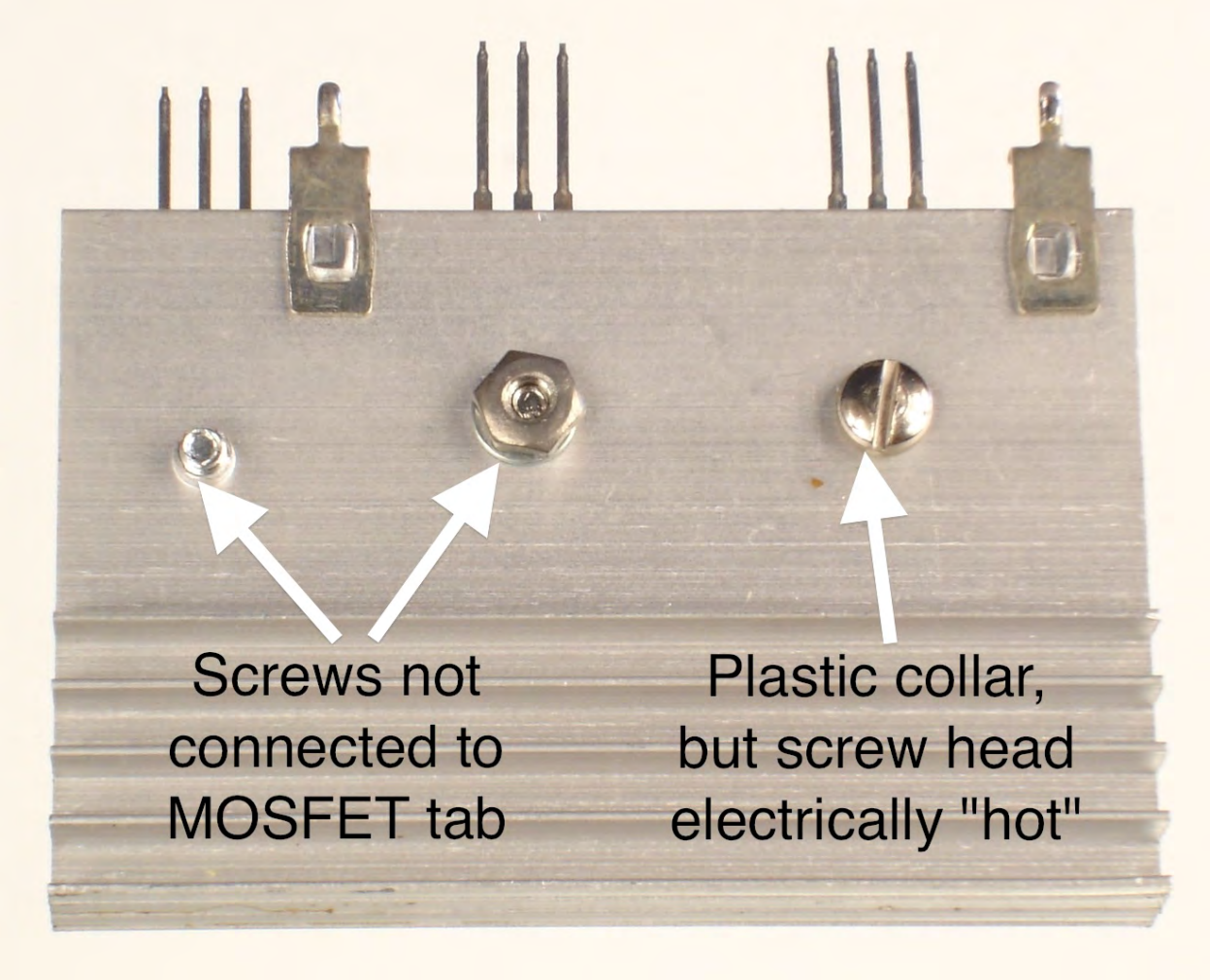


Figure 23.14b.
The heat-sink assembly shown in **Figure 23.14a**, but flipped vertically so you can see the back side.

How to Protect MOSFETs in Circuits

Earlier in this experiment you saw a circuit that would control a small DC motor. MOSFETs also can control large motors. But motors have a problem. When first turned on they can act almost like a short circuit. Compressors, vacuum cleaners, power tools, and air conditioners draw a large starting current that can cause incandescent lamps to dim briefly. When you control a motor or another load that momentarily draws a large current, that current could exceed the limit for the current a MOSFET can pass. (In an older house I saw lights flicker when someone used a vacuum cleaner or when the washing-machine motor turned on.)

The initial *inrush* current also might exceed the amount of current available from a power supply, which could trigger an overload condition and shut down the supply. (While writing this experiment I saw the overload effect when I tried to power a 12-volt motor from a lab power source. Almost as soon as I turned on the motor it forced the supply into a current-limit mode.) So circuits that control loads – usually motors, solenoids, and relays – might need a way to reduce or limit that initial burst of current. Several companies manufacture electrical devices that do just that. Negative temperature-coefficient (NTC) thermistors, for example, normally have resistances from half an ohm to several-hundred ohms and can handle currents from fractions of an ampere to 10's of amperes. Circuit diagrams show a thermistor as a small rectangle with a terminal at the short ends. The "hockey-stick" line through the rectangle indicates "thermistor." **Figure 23.15** shows a circuit that includes a thermistor.

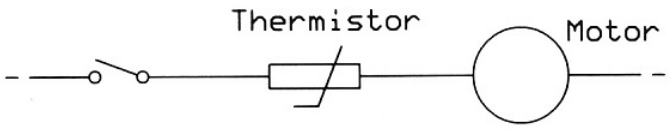


Figure 23.15.
This diagram includes a thermistor identified by the "hockey-stick" line through the rectangle. The thermistor limits an initial surge of current when you first turn on the motor. As current passes through the thermistor, it warms and lets more current flow to the motor.

Initially, a thermistor in series with, say, a motor, has a high resistance that limits the initial current to the motor. That current causes the thermistor's temperature to rise, which in turn decreases the thermistor's resistance, so more current flows. As additional current flows, the thermistor gets warmer still. So the thermistor resistance decreases until it reaches a point at which it becomes negligible in the load circuit. Small DC motors might not need this type of current limiter. But you must ensure the highest peak current a load draws does not surpass the specified peak current for a MOSFET used to control a device. (Ref. 2.) Inrush current-limiting thermistors available through distributors cost from about $0.60 to $15, depending on their capabilities. The Ametherm, Inc. Web site provides useful information and application notes about thermistors used for circuit protection. http://www.ametherm.com/. **Figure 23.16** shows several types of thermistors used in electrical equipment and electronic circuits.

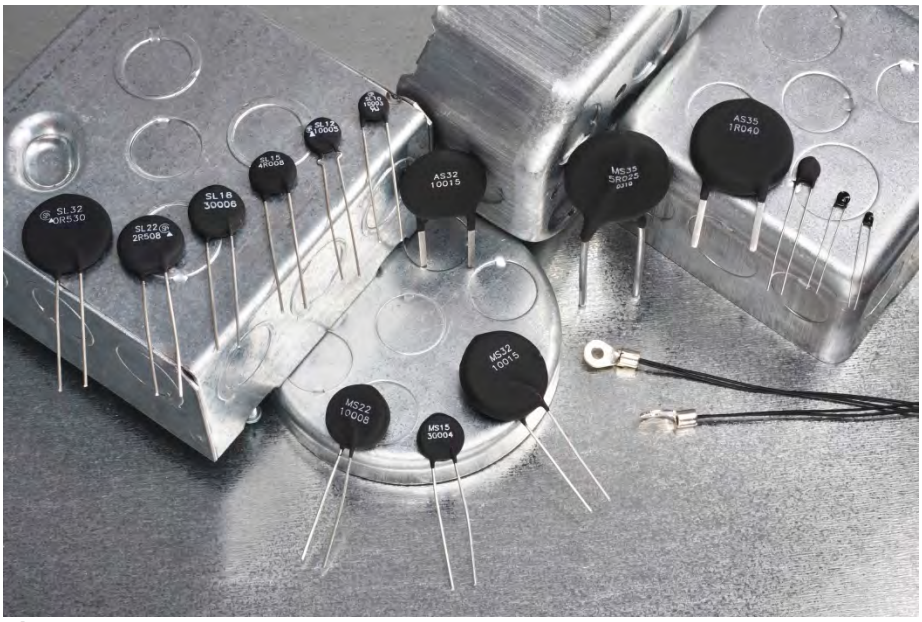


Figure 23.16.
Examples of thermistors used in industrial equipment to limit inrush current. The component dimensions depend on the current and voltage characteristics a load and a control circuit require.
Courtesy of Ametherm, Inc.

When equipment turns off a motor, the control circuit might need another type of protection. A small DC motor such as one found in a handheld vacuum cleaner, comprises permanent magnets that surround a rotor. **Figure 23.17a** and **b** illustrates a small motor and the components from a disassembled motor of similar size. Electricity flows into the rotor through conductive brushes – made of carbon and graphite – that make contact with a commutator. The commutator moves with the rotor and switches current into coils on the rotor that get attracted to or repelled from permanent magnets. The rotor turns and converts electrical to mechanical energy.

Figure 23.17a.
A brushed DC motor and motor components. Clockwise: a small appliance motor, a motor case, motor end plate, rotor with three coil windings (brass commutator on right side), two permanent magnets, brush assembly.

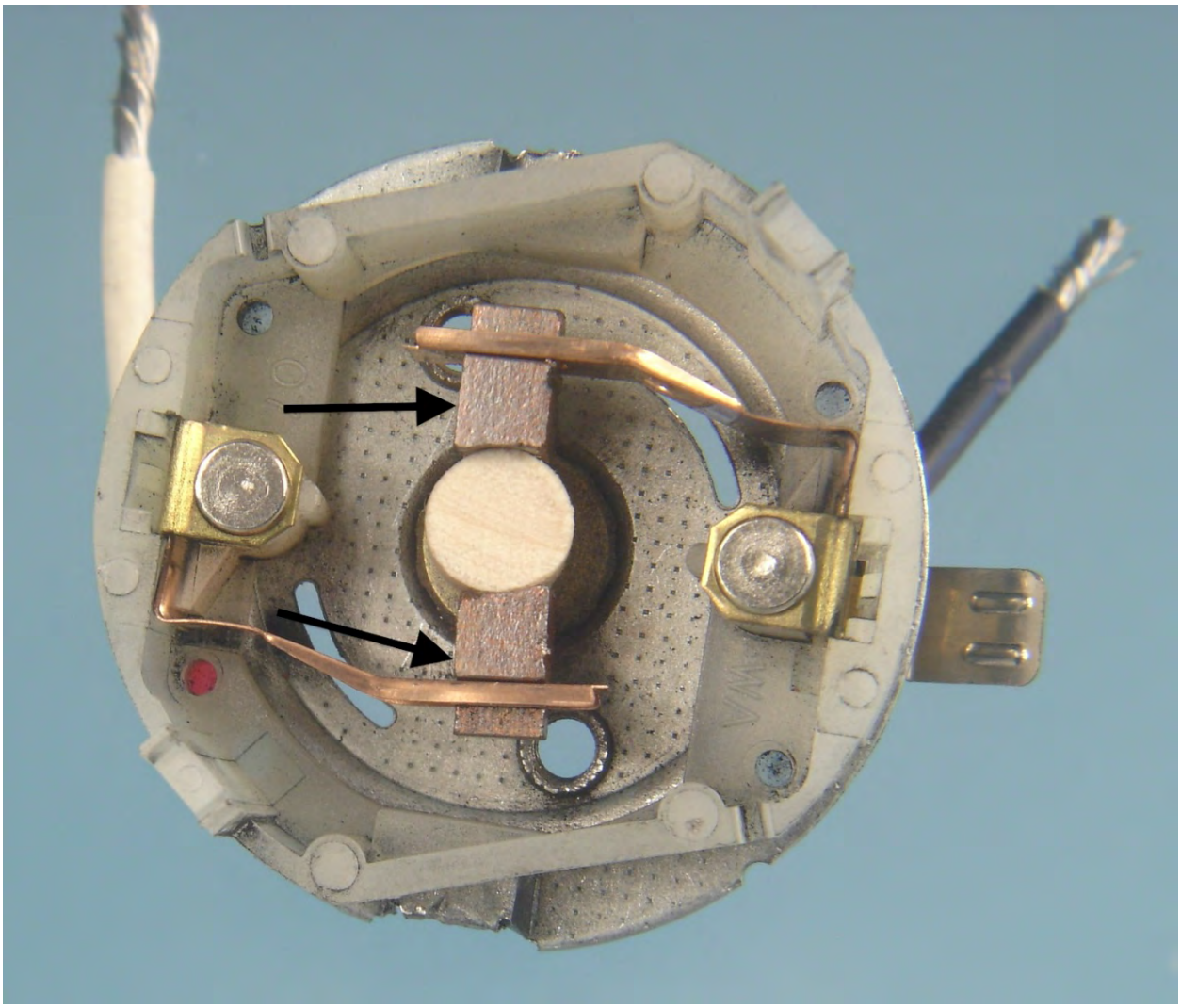


Figure 23.17b.
This photograph identifies the two brushes (arrows) that transfer current to the rotating coils through a commutator. To clearly show the two brushes, I inserted a piece of dowel rod between them to position the brushes much as they would appear with the rotor in place.

A generator uses the same arrangement of magnets and coils on a rotor to produce electricity. Mechanical energy spins the rotor's coils through the magnetic field. Because the coils experience varying magnetic fields, they produces electricity. When you turn off power to an electric motor such as the one shown in **Figure 23.16** it becomes a generator for as long as the rotor continues to spin.

As the coil's magnetic field quickly "collapses," the coils produce current that creates a "back EMF," or a back electromotive force. Measurements of this EMF show the polarity of the voltage and the current opposite to those when power runs the motor. For a graphic demonstration of the power created by the collapsing magnetic field in two large solenoids, watch the video created at the Massachusetts Institute of Technology as a physics-class demonstration: http://tsgphysics.mit.edu/front/?page=demo.php&letnum=H%2017. Small DC motors will not create such a large arc flash, but the back EMF can damage electronic components.

As a youngster, I took a small electromagnetic buzzer and a 6-volt lantern battery to school. I asked classmates to press the buzzer wires against the battery terminals. The buzzer coils had enough back EMF to give the kids a quick shock. They probably figured a 6-volt battery wouldn't do any harm.

If you plan to control DC motors with MOSFETs, the circuit might need a "transient voltage suppressor," or TVS. This type of semiconductor provides a "short circuit" path between the motor terminals so any back EMF dissipates quickly and doesn't damage MOSFETs. Many TVS devices use Zener diodes, named for Clarence Zener, who discovered an interesting semiconductor effect in some diode materials. Normally, diodes pass current from anode to cathode and block current from flowing from cathode to anode. Specifications such as a maximum forward current and peak reverse voltage characterize diode behavior.

Zener diodes have an additional specification, the Zener voltage, abbreviated V_Z. When the reverse voltage exceeds this specification, a Zener diode conducts current. A typical diode such as a 1N4001 or 1N4148 would burn out if the reverse voltage exceeds a specified maximum. Zener diodes and closely related avalanche diodes do not suffer ill effects from a reverse-bias condition (backwards polarity) as long as a circuit limits the current through them. **Figure 23.18** shows a Zener-diode circuit used to provide a 9.1-volt signal to other components. The 1N4739A diode has a Zener voltage of 9.1 volts. Diode manufacturers produce Zener diodes with specific V_Z values, from about 1.8 volts to as high as several-hundred volts. The symbol for a Zener diode looks similar to that for an LED, except the line for the cathode symbol has bent ends. Specifications for TVS diode might specify a reverse standoff voltage V_R instead of a Zener voltage. (The 1N prefix in a standard diode part number indicates one semiconductor junction. Manufacturers also use their own nonstandard part numbers.)

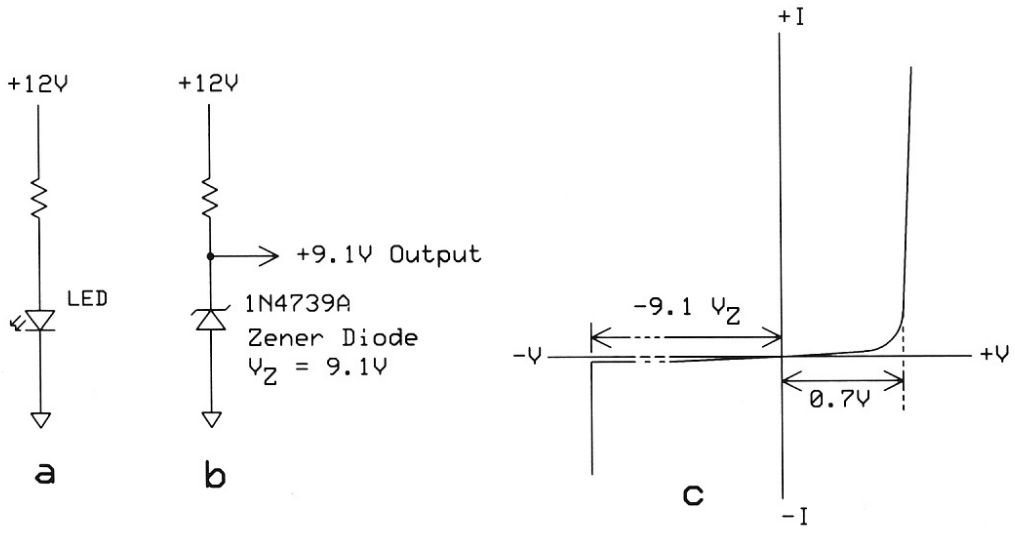


Figure 23.18.
A *forward-biased* LED circuit (a) passes current from the anode to the cathode. A *reverse-bias* Zener-diode circuit (b) provides a 9.1-volt signal to other circuit components. When a circuit applied a reverse-polarity voltage to a Zener diode – positive to cathode, negative to anode – the cathode operates at a fixed voltage, V_Z. A series resistor limits current flow through the 1N4739A diode and to the other components. The plot of current-vs.-voltage characteristics (c) shows the sharp conduction at the 9.1-volt reverse-bias voltage on the left side of the plot.

Because Zener diodes have well-characterized voltages, we use them to suppress transient back-EMF currents. In the case of a bidirectional 12-volt DC motor controlled by a MOSFET H-bridge circuit, the motor could use two 1N4747A 20-volt Zener diodes, placed in series, cathode to cathode. The two anodes would connect to the two motor terminals. This arrangement offers a short circuit between the motor terminals when the back EMF exceeds 20 volts, regardless of the polarity. **Figure 23.19** shows how the cathode-to-cathode Zener-diode circuit works.

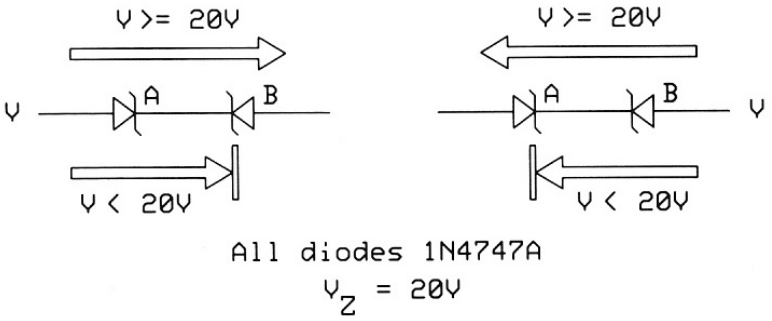


Figure 23.19.
In the circuit on the left, Zener diode A conducts normally (bottom arrow), but Zener diode B has a 20-volt V_Z, so it conducts only when its cathode goes above 20 volts (upper arrow). The circuit on the right behaves in the same way for a signal with the opposite polarity. Now diode B conducts normally but diode A will not conduct until its cathode reaches 20 volts. No current flows through these diodes unless the voltage across the pair exceeds 20 volts. A back-EMF signal above 20 volts or below -20 volts "sees" the Zener diodes as a short circuit, so the attached controllers do not get hit with high- or low-voltage spikes when the motor turns off.

Figure 23.20 shows the arrangement of Zener diodes for a bidirectional and a unidirectional DC motor. The unidirectional motor has voltage applied with only one polarity, so it requires only one Zener diode.

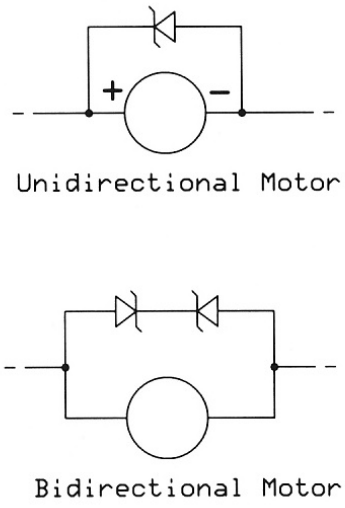


Figure 23.20.
Zener-diode arrangements for a bidirectional and a unidirectional DC motor. In each circuit the Zener diode must have a Zener voltage (V_Z) greater than the voltage used to power the motor.

Instead of connecting two Zener diodes cathode-to-cathode, manufacturers combine two Zener, or avalanche, diodes in one package sold as a bidirectional TVS diode. The 1.5KE22CA TVS diode from Littelfuse, for example, has an 18.8 volt reverse standoff voltage. Below that voltage, no current will flow through the device. You can purchase unidirectional TVS devices, too. Although Zener diodes simplify a discussion of circuit protection, I favor TVS devices specifically manufactured for that purpose.

Protect Circuits from MOSFET Voltages

When engineers design a product that includes low voltages for MCUs and analog circuits, as well as high voltage controlled by MOSFETs, they often isolate the two parts of the circuit. In effect, the design has *no* electrical connection between the low-voltage and high-voltage sections. This type of *galvanic isolation* helps

protect sensitive circuits from transient effects and voltage spikes in the high-voltage circuits. Imagine yourself connected to a medical instrument. You might like to know the measurement circuits have no direct connection to the power lines or other high-voltage circuits. But how can one circuit control another with no direct electrical connection?

You might use relays; devices that incorporate mechanical switches controlled by a separate electromagnet. The magnetic coil has no electrical connection to the switch contacts it moves. A 12-volt control signal could energize the relay coil that actuates the switch contacts to turn a line-voltage motor on or off. You might find relays in control units for furnaces and air conditioners. But relays can wear out, contacts can get dirty, and arcing between contacts can burn them and cause contacts to wear our or to fuse together. Even so, elevators, railway signals, and power-switching equipment rely on relays. These relays range in size from small to very large (**Figure 23.21**). I use a large, heavy railway-signal relay as a bookend in my office.

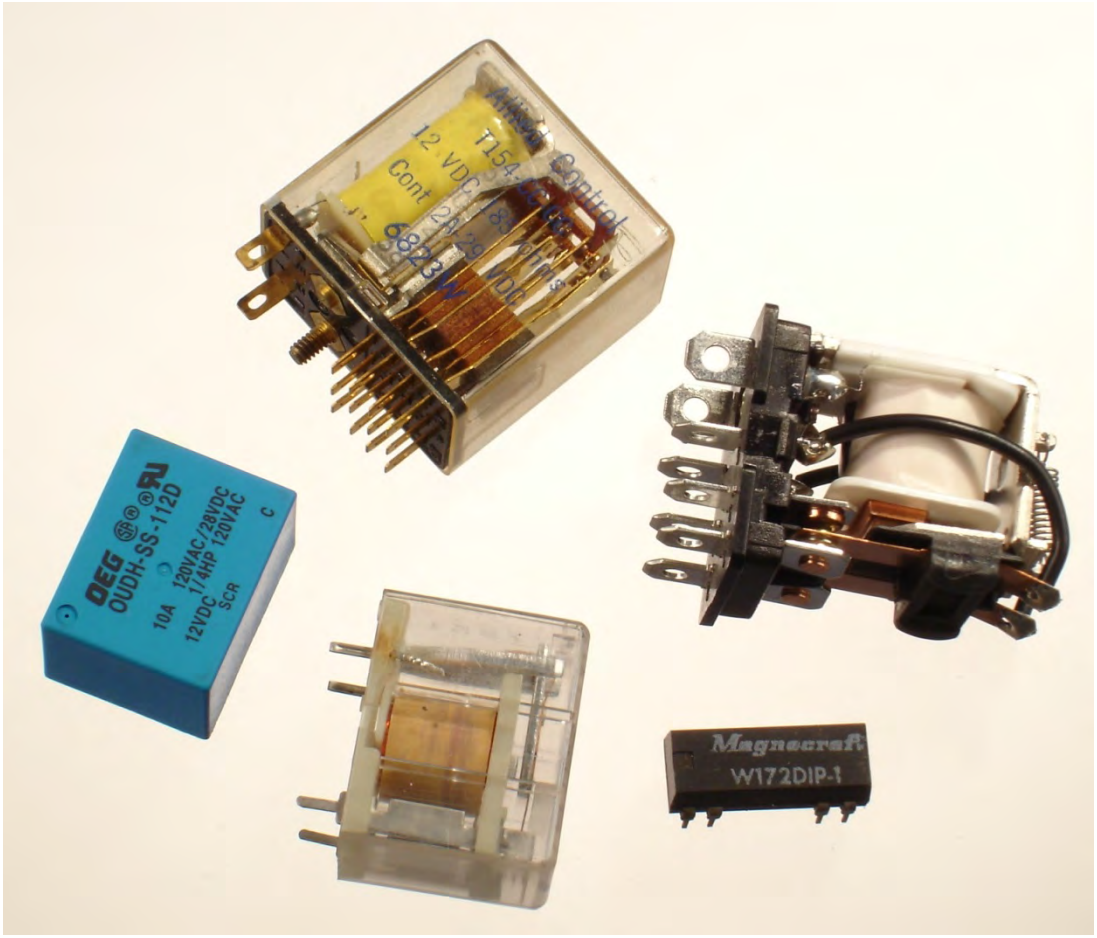


Figure 23.21.
Small relays come in a variety of forms that include a small dual-inline package (lower right) an open-frame relay above it, and a 4-pole double-throw relay in a clear plastic case (top). Relay specifications indicate the coil voltage, the current and voltage rating for contacts, the types of contacts, and whether the relay can run constantly energized or only intermittently.

Electronic-isolation devices include those that use an LED that transmits light to a phototransistor a short distance away. **Figure 23.22** offers a simplified cutaway drawing of an optical isolator, also called an optical switch, optoisolator, or optocoupler. Light from the LED serves much like the base current in a discrete NPN transistor. In most circuits when the LED turns on, the transistor emitter connects to ground.

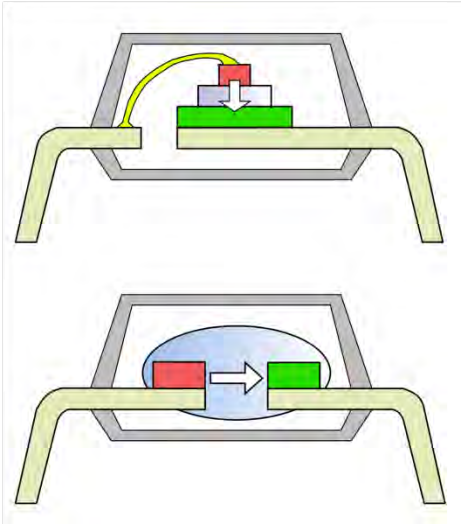


Figure 23.22.
Cutaway side view of two optoisolator ICs in dual-inline packages. A small distance separates the LED (red) and the phototransistor (green) to provide galvanic isolation as high as several-thousand volts. Transparent material transmits the LED light to the transistor chip.
Courtesy of Wikipedia (Ref. 3.)

An 6-pin 4N25 optocoupler (**Figure 23.23**) provides two connections for the LED and three connections for the transistor, although the base (pin 6) usually remains unconnected. Note the lack of any path for current to flow between the LED and transistor. A 4N25 optoisolator can withstand an isolation surge voltage as high as 7500 volts AC. (Semiconductor prefixes 4N through 6N indicate optoelectronic devices, but some manufacturers that use other numbering systems.)

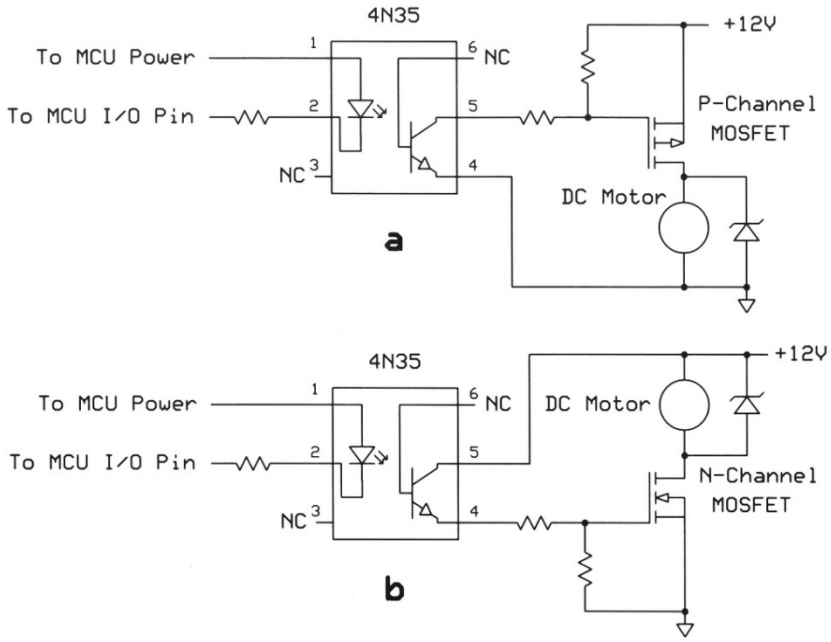


Figure 23.23.
Typical optoisolator circuits in which an MCU drives an optoisolator LED that controls a transistor, which in turn changes the voltage at a MOSFET gate. The upper circuit (a) represents a high-side MOSFET switch, while the lower diagram (b) shows a low-side switch. Neither circuit has an electrical connection between the MCU side and the 12-volt MOSFET side of the optoisolator.

In both circuits just shown, the MCU must turn the LED on to cause the motor to run. If the LED receives no power, the motor cannot run. This type of "fail safe" operation should become a regular engineering goal in circuits, products and home-brew designs.

The circuits provided in **Figure 23.23** include a unipolar TVS diode across the motor terminals to short circuit back EMF created when the motor turns off. Keep in mind that most optoisolators serve as switches that only turn on or off. You could take the motor-control circuit from **Figure 23.7** and substitute optoisolators for the 7407 open-collector buffers. Doing so would isolate the H-bridge circuit from the four MCU I/O pins. Low-cost optoisolators – about $0.25 for a 4N25 – give designers an easy way to isolate portions of circuits from one another.

IMPORTANT: When you evaluate an optocoupler's specifications, treat its internal transistor as a normal bipolar transistor and look at the collector-to-emitter voltage. When you use it to control a MOSFET, you want a low-enough value to ensure that when the transistor turns on the applied gate voltage falls below the gate-threshold voltage, $V_{GS(th)}$ for a P-channel MOSFET or above the gate-threshold voltage, $V_{GS(th)}$ for an N-channel MOSFET.

Some circuits might need more than one optoisolator. The 16-pin Avago Technologies ACPL-244, for example, includes four separate pairs of LEDs and phototransistors. The LED for each of the four channels comprise a head-to-tail pair of LED chips so the LED can function with an AC signal. Optoisolators can include other types of semiconductors in place of a simple NPN transistor. Manufacturers include Avago Technologies, Fairchild Semiconductor, Lite-On, Toshiba, and Vishay.

Optoisolators have signal-rate limits designers must consider (**Figure 23.24**). The transistor in a 4N25 has a typical turn-on time (t_{on}) of 2.8 µsec and it takes 4.5 µsec for it to turn off (t_{off}). A 1.0 MHz square wave has a period of 1.0 µsec, so if you try to drive a 4N25 optoisolator at this frequency, it never has enough time to turn on, let alone turn off.

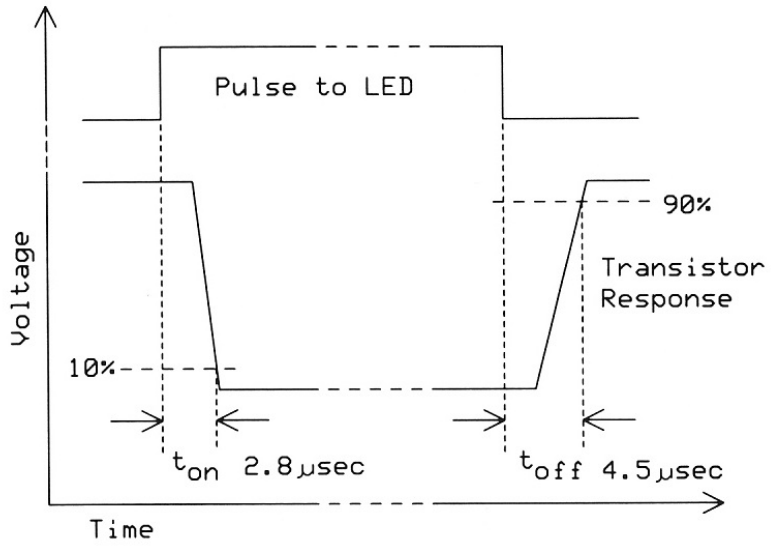


Figure 23.24.
Timing diagram for a 4N25 optoisolator. The turn-on time (t_{on}) and turn-off time (t_{off}) restrict this device to medium-frequency signals. I recommend frequencies no higher than 50 kHz for a 4N25. Other optoisolators can operate at higher frequencies.(The timing in this diagram is not to scale.)

Companies sell optoisolators that respond faster and can handle data rates of several million bits per second. Of course as rates increase, so do prices. A Toshiba single-channel TLP117(F), for example can operate at frequencies as high as 50 Mbits/sec. This device costs about $3.00 in single-piece quantities.

Analog Isolation

Given the use of optoisolators to electrically separate portions of digital circuits that need on-off control, or that communicate on-off signals, you might wonder if galvanic isolation can apply to analog circuits. It does. An engineer might need to measure a high voltage with an ADC, but without exposing sensitive analog circuits to high-voltage transients or short circuits. An analog isolation IC handles this situation nicely in several ways: optical, magnetic, or capacitive isolation.

A circuit designer could isolate analog signals in other ways, too: wireless communications, through a transformer, via a fiber-optic cable, and so on. But those techniques involve more than an IC and a few additional components.

Avago Technologies manufactures analog optoisolators of which the HCNR200 and HCNR201 serve as examples. These devices have a linear response from input to output. That means a signal input with amplitude *A* will appear at the isolated output with an amplitude that has a straight-line relationship with *A*. The HCNR201 has a transfer gain of ±5 percent and the HCNR200 has a ±15-percent transfer gain. For our purposes, the HCNR201 will give us the best results.

These two isolation devices comprise one LED and *two* photodiodes, PD1 and PD2. A typical circuit uses one of the photodiodes to receive analog signals from the LED. The other photodiode serves in a simple feedback loop to help stabilize the LED output. **Figure 23.25** shows a typical circuit for an HCNR201 taken from the Avago Technologies HCNR200 and HCNR201 data sheet. This 19-page document includes another circuit that offers higher-precision results. It requires other components and a well-designed circuit board. Find the HCNR200 and HCNR201 data sheets at: www.avagotech.com/docs/AV02-0886EN. (Note the use of the 2N3904 and 2N3906 transistors introduced in an earlier experiment.)

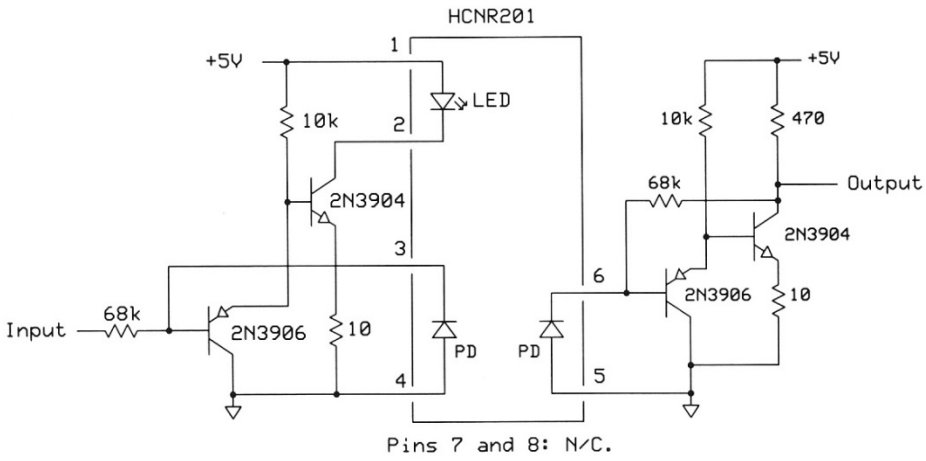


Figure 23.25.
Use of an Avago HCNR201 analog optoisolator IC in a practical circuit. One 8-pin IC provides the LED and the two photodiodes. The isolation barrier in this IC can withstand 5000 volts.

Avago Technologies sent me samples of the HCNR201 in 8-pin DIPs that simplify breadboard experiments. The isolation IC has leads on standard 0.1-inch (2.54 mm) centers, although the two rows have a 0.4-inch (1 cm) spacing. That dimension makes them wider than most of the ICs used in previous experiments, but the HCNR201 still fits nicely in a solderless breadboard. **Figure 23.26** shows my measurements of voltage in (diamond points) and voltage out (square points).

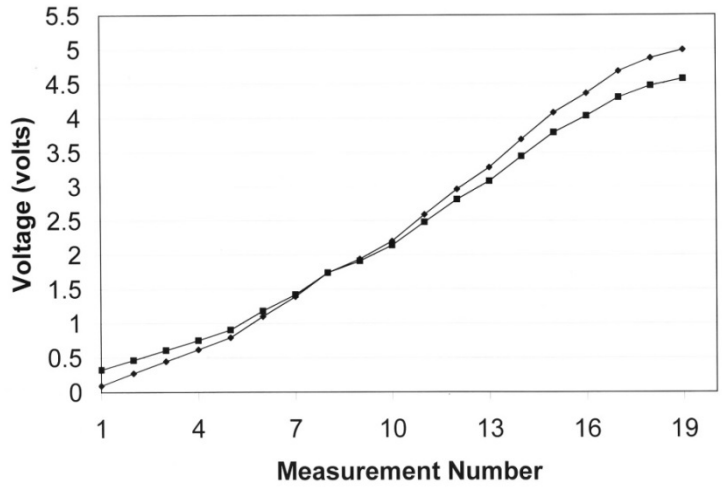


Figure 23.26.
Input and output voltages from the HCNR201 analog-isolation test circuit plotted against the measurement number. Does this information make sense?

Let's analyze what we have: At first the results in **Figure 23.26** don't look good. The two lines seem "wavy" and "lumpy." Not what we expect from a linear device. Do you know why?

The plot has a problem because it uses a sample number for the x axis. *The sample number has no importance to us.* The information looks odd because I should have plotted voltage input vs. voltage output. That graph gives better results, as shown in **Figure 23.27**. Don't be afraid to look at your data in several ways. In this case, the sample number had no relationship to the input or output voltage. We needed the voltage-to-voltage relationship.

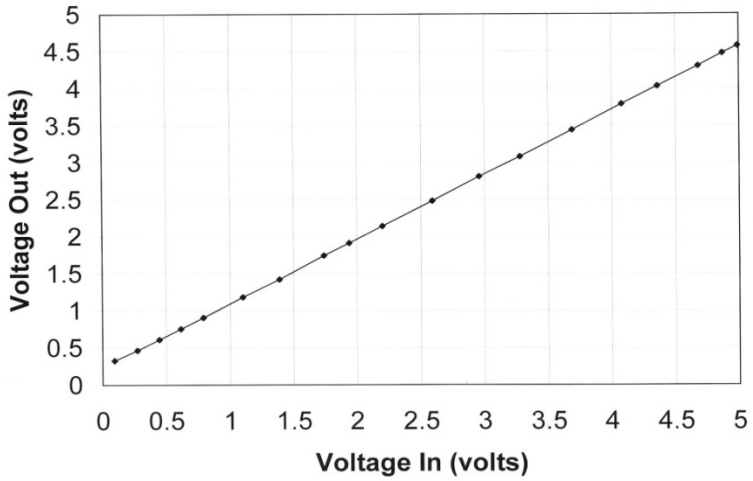


Figure 23.27.
Plot of voltages at the input to the HCNR201 circuit vs. the corresponding voltage at the circuit output. This information shows a linear relationship between the input and output voltages, which is what we expect, based on the information in Avago Technologies' data sheet.

See my lab-test setup in **Figure 23.28**. For the tests above I used the same 5-volt power source for both sides of the "isolated" circuit so I could make consistent measurements with the same power-supply voltage. The circuit worked well with separate power supplies, too.

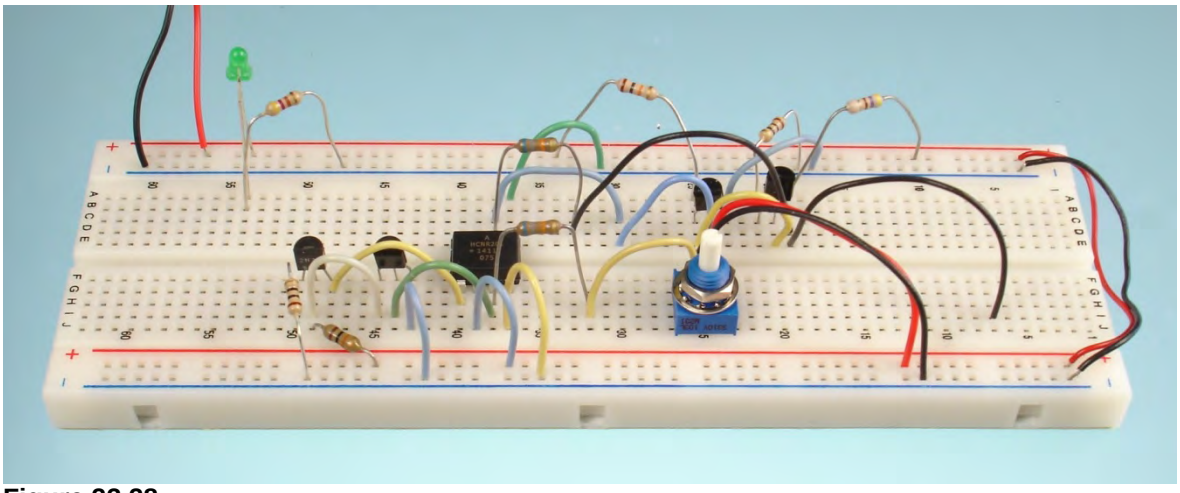


Figure 23.28.
Breadboard used to measure the linearity of an Avago Technologies HCNR201 analog optoisolator IC in an 8-pin dual inline package.

Other Isolation Techniques

Companies offer other types of isolation techniques. Analog Devices, for examples, has a patented transformer technology named iCoupler. Devices based on this technology use small internal transformers to electrically separate one circuit from another. An introductory article from Analog Devices explains how the iCoupler devices work. (Refs. 4 and 5.) An application note explains that iCoupler devices, "may cause radiated emissions and conducted [electrical] noise if not considered during printed circuit board (PCB) layout and construction." See the For More Information section at the end of this experiment.

Texas Instruments uses capacitive coupling in the ISO72XX series of ICs that work with digital signals. The ISO7220, for example, handles digital signals at data rates as high as 1M bits/second. Similar ISO72XX ICs can handle signals at up to 150M bits/sec. Texas Instruments has produced a 17-page design guide for people interested in this family of ICs. (Ref. 6.)

The transformer and capacitor isolation techniques could present design challenges that require shielding, careful PCB layout, power-source decoupling, and filtering, and so on. I would recommend circuit designers and experimenters look first at the optical devices.

Conclusion

Your journey with me through electronics gave you insight into different circuits, how they work, and what you can do with them. Unlike some information that concentrates on projects, I have given you more general information and circuits you can apply as you choose. When you have fun with electronics, you learn a lot. It might surprise you to see parallels between electronics and other fields of engineering and science. You might remember investigating how capacitors charge and discharge in the experiment for a 555-timer IC. The charge-discharge characteristics follow the same mathematical equations as the rates of some types of chemical reactions. Nature offers many surprises.

References

1. "Ringing (signal)," Wikipedia. http://en.wikipedia.org/wiki/Ringing_(signal).

2. Sammi, Mehdi, "Special Thermistors Limit Inrush Current," Power Electronics Technology, October 1998. www.ametherm.com/inrush-current/inrush-current-limiters-pcim.html.

3. "Opto-isolator," http://en.wikipedia.org/wiki/Opto-isolator.

4. Krakauer, David, "Digital Isolation Offers Compact, Low-Cost Solutions to Challenging Design Problems," Analog Dialogue, Analog Devices, December 2006. http://www.analog.com/library/analogdialogue/archives/40-12/iso_power.pdf.

5. Wayne, Scott, "iCoupler® Digital Isolators Protect RS-232, RS-485, and CAN Buses in Industrial, Instrumentation, and Computer Applications," Analog Dialogue, Analog Devices, October 2005. www.analog.com/library/analogDialogue/archives/39-10/iCoupler.html.

6. "Digital Isolator Design Guide," SLLA284, Texas Instruments, January 2009. www.ti.com/lit/an/slla284/slla284.pdf.

For More Information

– -, "Optocouplers and Solid-State Relays," Application Note 02, Vishay Semiconductors, 2011. www.vishay.com/docs/83741/83741.pdf.

– -, "The ARRL Handbook for Radio Communications," 2009 ed., American Radio Relay League, Newington, CT. ISBN: 978-0-87259-139-4. Excellent discussions of transistors and circuits.

– -, "Understanding Power MOSFET Data Sheet Parameters," Application Note AN11158, NXP Semiconductors, February 2014. www.nxp.com/documents/application_note/AN11158.pdf.

Andreycak, Bill, "New Driver ICs Optimize High Speed Power MOSFET Switching Characteristics," Unitrode Integrated Circuits Corporation, Merrimack, N.H. 1999. www.ti.com/lit/an/slua054/slua054.pdf.

Christian, Francis, "Isolation Techniques Using Optical Couplers," Application Note AN571A. Motorola Semiconductor, 1995. encon.fke.utm.my/nikd/Internet/opto-couplers.pdf.

Kennedy, Brian and Mark Cantrell, "Recommendations for Control of Radiated Emissions with iCoupler Devices," Application Note AN-1109, Analog Devices. www.analog.com/static/imported-files/application_notes/AN-1109.pdf.

Avago Technologies has many useful application notes for optocouplers on the Web site: www.avagotech.com/pages/optocouplers_plastic/apnotes.

Titus, Jon, "Power Relays Give Designers Much to Ponder," *ECN*, November 2007. www.ecnmag.com/articles/2007/11/power-relays-give-designers-much-ponder.

Answers

Experiment 23, MOSFETs and Bipolar Transistors (end):

Question: Look again at the information in **Figures 23.4** and **23.5**. Do you see a difference in the way each circuit works?

Answer: A logic-1 at the input to the 7404 inverter in high-side switch turns the LEDs off and a logic-0 turns them on. In the low-side circuit the opposite actions occur: A logic-0 input to the 7407 buffer turns the LEDs off. A logic-1 turns them on.

Optional: Given the test circuit in **Figure 23.10**, could you modify it to indicate short circuits? If so, what changes would you make and what components would you add or remove?

Answer: You could add four more LEDs and resistors to the circuit as shown in **Figure 23.A**. I added orange and yellow LEDs and 150-ohm resistors, and I replaced the 470-ohm resistor with a 220-ohm resistor. Now you can test the H-bridge circuit by applying all 16 input patterns to the 7407 inputs and watching the results. If two orange or two yellow LEDs turn on, you will have a short circuit on that side of the bridge circuit when you remove the LEDs and resistors and try to control a motor (**Figure 23.10**).

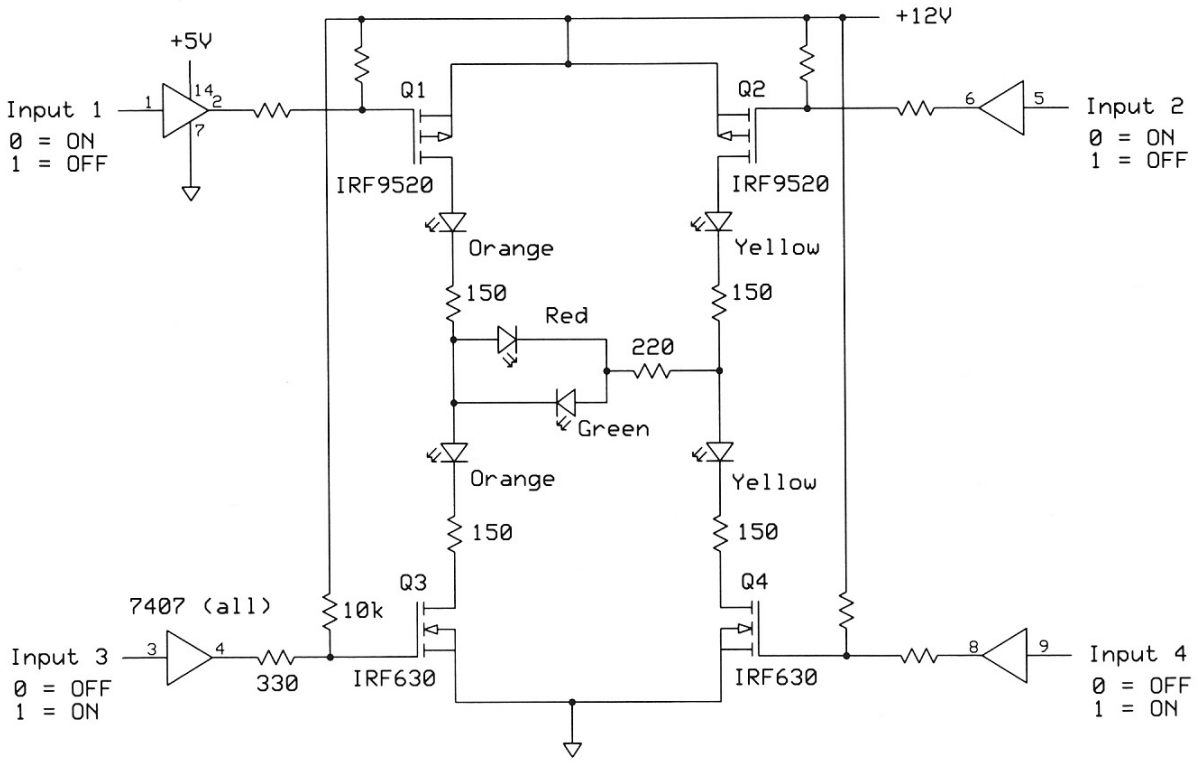


Figure 23.A.
A test circuit that indicates when a short-circuit condition exists.