Don Lancaster's

MACHINE LANGUAGE PROGRAMMING COOKBOOK

Part Two

Machine Language Programming Cookbook II

by Don Lancaster

An eBook reprint of chapters 8 and 9
of Micro Cookbook Volume II

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com

THIRD EDITION
FIRST PRINTING—2010

International Standard Book Number: 1-882193-15-8

Created in the United States of America.

ABOUT THE AUTHOR

Don Lancaster heads *Synergetics,* a new-age software, prototyping, and consulting firm involved in micro applications and electronic design. Don is the well-known author of the classic *CMOS* and *TTL Cookbooks.* He is one of the microcomputer pioneers, having introduced the first hobbyist integrated circuit projects, the first sanely priced digital electronics modules, the first low cost TVT-1 video display terminal, the first hobbyist keyboards, and lots more. Don's numerous books and articles on personal computing and electronics applications have set new standards for understandable, useful, and exciting technical writing. Don's other interests include ecological studies, firefighting, cave exploration, tinaja questing, and bicycling.

Other Howard W. Sams books by Don Lancaster include *Active Filter Cookbook, CMOS Cookbook, TTL Cookbook, RTL Cookbook* (out of print), *TVT Cookbook, Cheap Video Cookbook, Son of Cheap Video, The Hexadecimal Chronicles, The Incredible Secret Money Machine, Don Lancaster's Micro Cookbook,* Volume 1, and the continuing *Enhancing Your Apple II* series.

Preface

Machine Language Programming is the second of three volumes on the fundamentals of microprocessors and microcomputers. In this volume, we (that's you, me, and that gorilla) look into the details of the micro's own language.

Volume 1 covered the fundamentals of microprocessors needed for us to start understanding machine language programming. Volume 3 is a reference volume containing detailed descriptions of hundreds of popular and micro-related integrated circuits.

Why machine language? Because, as it turns out, virtually *all* winning and top performing microcomputer programs run *only* in machine language. The marketplace has spoken. It has not only spoken but is shouting: BASIC and PASCAL need not apply.

Volume 2 will show you the fundamentals of machine language programming through a series of *discovery modules* that you can apply to the microprocessor family and the microcomputer of your choice. Once you get past these modules and gain a deep understanding of what machine language is all about, then you can step up to the wonders of *assembly language,* which is really nothing but automated machine language programming that is made much faster, lots more convenient, and bunches more fun.

Volume 2 picks up at Chapter 6 in this continuing series. Here we look at address space and addressing concepts, as well as working registers and how they are used. Next is a study of system architecture, seeing what goes where in a typical microcomputer, with heavy emphasis on understanding system buses and how they work. From there we go into memory maps and on to addressing modes, those all-important methods a microcomputer's CPU has of accessing memory and its own working registers. We look at seven fundamental addressing modes that apply to most micros one way or another, either by themselves or in combination.

Address modes are then summarized in a group of quick-reference charts. Next come some stock forms useful for hex dumps, machine language programming, and assembly language programming. This chapter ends up with a toolkit that you can put together for machine language work.

Chapter 7 is the real heavy of this volume. Here we actually do lots of machine language programming. We use the "those #$!#$ cards" method, in which you work one-on-one with each individual op code as the need arises, again on the microprocessor of your choice. There is a series of nine *discovery modules* here. These are elementary programming problems that start with the simplest of

op codes and programming concepts and work their way up into some fairly fancy results, using practically all the available microprocessor op codes on the way. As we go through the modules, we also pick up details on flowcharting and using programming forms; measuring time and frequency; calculating branch values; using a stack; testing individual bits; creating text messages; using files, subroutines, interrupts, breakpoints, arithmetic, and much more.

We do not dwell on micro arithmetic because math uses of micros are not all that important when it comes to real programs doing real things for real people. Math on micros simply does not deserve the overblown treatment some texts give it.

While many examples are given that involve the 6502, you can easily do the discovery modules on any micro of your choice—4-bit, 8-bit, 16-bit, or whatever. All program problems and examples have purposely been done on a mythical and nonexistent trainer, so that you are forced to think things out on your own, solving your own problems in your own way on your own machine.

In Chapter 8, we take a detailed look at I/O, or input/output. We find there are four levels of I/O and then explore the two lowest levels in detail. At the device level, we check into parallel and serial ports, look at the different port types, and examine specific chips. Then we find out how to interface such things as keyboards and displays, using a minimum number of port lines.

Next, at the circuit level, we examine the simple circuitry needed to "amplify," "isolate," or "convert" micro port lines into signals powerful enough to go out into the real world with a vengeance. Here we include such things as transistor drivers, triacs, optocouplers, input conditioners, analog-to-digital converters, digital-to-analog converters, and things like that.

Chapter 9 both wraps up this volume and completes the "how" part of the trilogy. First and foremost, we check into the *micro applications attack,* a real-world problem-solving method that I use. It has been thoroughly tested and, above all, it works. Emphasis is placed on everything that has to be done *away* from the micro, using the "stickiest box" method to zero in on the real problem hidden inside what you are trying to do.

The micro applications attack is followed by some real-world problems that you can solve using this method. Project "F" is particularly challenging. Then we consider where you have to go from here. Finally, for those of you still wondering "What good is all this stuff?," we end the book with a list of sixty-three microcomputer ideas that you can immediately put to challenging, unique, and profitable uses.

DON LANCASTER

Contents

This book is dedicated to microcomputer pioneers everywhere. You can tell them by all the arrows in their backs.

— eight —

Interface and I/O

The word "interface" means many different things to many different people and it can be used in many different ways with microcomputers. Here's a totally general and totally worthless definition for you . . .

> **INTERFACE**—Any way to connect a micro to something else.

Interface is obviously important, since any computer is totally useless if there is no way to put things into it or to get stuff back out of it. Many of the problems to be solved and the dollars to be made in the micro world have to do with interface hassles.

To some, interface involves providing port lines on a microcomputer board. To others, interface centers on the use rules for those ports. To still others, interface is the external hardware needed to sense or power real-world sources and loads, such as motors, lamps, humidity sensors, and so on. Some people see interface as the high-level software design needed to handle a total problem. Then there's the really big interface picture of actually interacting with people and getting a project put to actual use.

As I see it, there are four different levels of interface. Each of these levels takes different skills and uses different ideas and concepts . . .

INTERFACE LEVELS
MICRO LEVEL—Ways to get signals onto or off of a microcomputer. A parallel I/O port is an example.
CIRCUIT LEVEL—Ways to get from real-world inputs to micros and from micros to real-world outputs. An optocoupler and a triac driving a 100-watt light bulb is an example.
SYSTEM LEVEL—Ways to let a microcomputer solve real-world problems. The hardware and software needed to connect a disk drive to a microcomputer is one example.
PEOPLE LEVEL—Using microcomputers to interact with social and cultural institutions. Using a microcomputer to help solve part of a public transportation problem is an example.

The *micro level* is the lowest of the four interface levels. Here we worry about how to get any signal into or out of the microcomputer system we are working on. Any input and output signals should be low-level ones that are fully compatible with the signal levels the micro needs and expects.

Parallel ports and *serial ports* are two examples of micro level interface. Unless you are building your own microcomputer from scratch, you are probably more interested in learning the use rules for existing hardware than in creating your own new circuits. The skills you will need here involve both low-level software and board-level hardware design.

The big hassles with signals going into a micro's ports are that the signals must be small, safe, isolated, digital, and microcomputer compatible. The problems with signals coming out of a micro's ports are that the signals are fairly weak and may need safety isolation or conversion to something else. That something else could be an analog signal or a mechanical motion.

This leads us to the second or *circuit level* of interface. The circuit level involves us with "shirtsleeve electronics," or the nuts-and-

bolts skills of bolting "amplifiers," "isolators," or "converters" onto our port lines so we can do some useful stuff with them. For instance, if we want to distinguish night from day, we somehow have to start with a light sensor, and then change this slowly varying signal to a crisp, noise-free digital signal just right to go into a port. If we want to light a 100-watt light bulb, we have to take the weak signal output from a port line, safety isolate it with an optocoupler, and then "amplify" it with a triac so it is powerful enough to control the lamp.

The circuit level of interface usually takes more hardware than software and involves a lot of non-computer, traditional electronic concepts such as power control, signal conditioning, analog-to-digital and digital-to-analog conversion, use of sensors, and so on. The person doing the circuit level interface will treat the microcomputer as just another device. There's this sensor, that motor, and this computer, all to be interfaced.

The *system level* of interface is third up from the bottom. Here we worry about what is involved in doing a complete job. For instance, if we want to add a disk drive to a microcomputer, first we need to get some ports on the computer at level one. Then we need to add some level two circuits to those ports to interconnect with the drive. Finally, at level three, we have to decide how the drive is going to be used, what software is involved, what the maintenance procedures will be, who will be using the drive for what purposes, and so on.

System level skills are involved in high-level software design, protocols, human engineering, handshaking, ergonomics, training, user manuals, repair methods, and so on. Persons working on level three must be good communicators and must concentrate first on the forest and then on the trees.

The *Micro Applications Attack* of the next chapter will show you how to handle level three system interface problems.

Finally, there's level four. The *people level* of interface. Argh.

Just because a simple, cheap, and elegant technical fix for a problem exists, don't expect for an instant that it will be widely accepted and immediately used. Let's look at three wildly different examples. First, we see there is simply no solution for public transportation problems, because there are lots of people and institutions around whose very existence depends on there continuing to be no cheap, reliable, and widely used public transportation system. As a second ferinstance, the greatest disaster ever to befall the *March of Dimes* people was the discovery of a cure for polio. Finally, and obviously, the QWERTY keyboard typing arrangement is so incredibly stupid that it isn't even funny.

You see, once anything hangs around for a while, it becomes an institution. Combine this with the way that many people and *all* institutions hate any kind of change, and you have the roots of the problem. Millions of dollars are lost per day worldwide by not immediately switching to the *Dvorak* typing keyboard with its 2:1 speed, 5:1 energy, and 3:1 error advantages over QWERTY, yet this isn't about to happen, at least not overnight.

The typing keyboard is one example where the benefits of a switch are obvious, conversion costs are minimal, the results are clearly defined, and not too many people in not too many places of power are threatened. Many social level problems are formulated in such a way that there is *no* solution simply because those doing the formulating do not even want there to be a solution, let alone for you to find it. A strong case can be made that federal solar energy funding was blown on totally ridiculous research to "prove" that things don't get warm when they sit out in the sun.

Anyway, level four takes politics, an understanding of human nature, posture threats, power balances, couched verbiage, PR puffery, ego suppression, group manipulation, and so on. If you are great at interface level one, you will surely make a fool of yourself, or worse, at level four, and vice versa.

Recognize that the four interface levels need fundamentally different types of skills and totally different personalities to handle them successfully.

I'll assume you are more comfortable with level one or level two interface, because otherwise you wouldn't have gotten this far in this book. If are a level four person, you are not reading this. So, let's split this chapter roughly in half, and start with micro level interface and finish up with circuit level interface. Then, in the next chapter, we will look at level three interface where you'll find a good method to solve clearly defined system level problems.

MICRO LEVEL INTERFACE

We now know that micro level interface consists of getting small low-level digital signals onto or off of a microcomputer's circuit board. We obviously need ways to put stuff into micros and get things back out. Micro level interface will always be involved in this.

In micro level interface, we always assume that the signals going into and those coming out from the micro are of just the right size and the precise shape to make the micro happy. Most often, these

will be low-level LSTTL or CMOS digital integrated circuit signal levels.

What are these levels?

LSTTL likes something near zero volts for a zero and something above +2.4 volts for a one. CMOS likes something near zero volts for a zero and something near +5.0 volts for a one.

Here are the signal levels involved . . .

INTEGRATED CIRCUIT SIGNAL LEVELS

- LSTTL -

LSTTL *circuits* normally work on a +5-volt power supply and like zero volts for a zero and 2.5 or more volts for a one.

LSTTL *inputs* need current sinking to be held low, and normally pull themselves high.

LSTTL *outputs* normally can sink 20 milliamperes or so to ground, but are fairly weak at pulling output loads positive. LSTTL CANNOT pull an output above +2.5 volts without outside help.

- CMOS -

CMOS *circuits* may work on a +5-volt power supply and like 0 volts for a zero and 5 volts for a one.

CMOS *inputs* are very easy to hold low or high but always must be connected to something else to prevent noise and power problems.

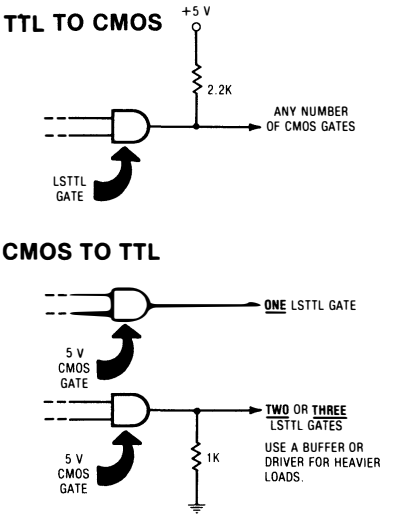
CMOS *outputs* are fairly weak but can usually source or sink 4 or more milliamperes of current.

LSTTL circuits can drive other LSTTL circuits without any problems. CMOS circuits can drive other CMOS circuits with no problems. If you want to mix and match LSTTL and CMOS, though, you have to watch what you are doing.

A CMOS circuit working on a +5-volt DC supply can normally drive one or two LSTTL circuits, but its drive is so limited that you aren't allowed simply to hang bunches of LSTTL on a CMOS output. If you have to drive lots of LSTTL with a CMOS output, you could use a CMOS buffer or else a single LSTTL gate, or whatever, to "amplify" the CMOS output.

LSTTL signals do not usually get high enough to guarantee a CMOS one level. To get around this, you can try adding a single pullup resistor to a LSTTL gate's output to insure a +5-volt one.

Here's how you interface LSTTL to CMOS and vice versa . . .



The pulldown resistor in the CMOS to LSTTL interface is only needed if you are driving two or three LSTTL inputs from one CMOS output. This same circuit can be used to drive one old-fashioned regular TTL input.

The newest "74HC" series of CMOS is more-or-less LSTTL-compatible and is intended as a power-saving LSTTL replacement. But you still have to watch input levels (pullups may be needed), and you still have limited output drive. Read the fine print on the data sheet whenever you connect an LSTTL output to a 74HC input.

NMOS circuits usually behave just like CMOS ones if they work on a single +5-volt supply. Some older NMOS integrated circuits also need a negative supply voltage. Thankfully, these are becoming rare.

The peripheral ports and other chips we may add to a microcomputer to do micro level interface normally are LSTTL, CMOS, or NMOS and use the same single +5-volt supply that the micro does.

Some safety rules . . .

INTEGRATED CIRCUIT SAFETY RULES

The input to a typical LSTTL, CMOS, or NMOS integrated circuit must NEVER be allowed to go either below ground or above the positive supply!

NEVER assume that an unconnected input is in a certain state!

Unused inputs on typical LSTTL, CMOS, or NMOS cicuits should USUALLY be tied to the positive supply or ground!

CMOS circuits should NEVER have their power removed if you are still supplying very strong or low impedance input signals!

Unused integrated circuits must ALWAYS be stored in anti-static protective foam!

Very simply, if you try to put too much into or take too much out of your typical IC, the chip may destroy itself. At the very least, the results won't be logically useful.

Tying unused inputs somewhere like ground or +5 volts is essential. This way, you always know what is going into leads that are not in active use. An unused LSTTL input normally tries to pull itself high or to a logical one. An unused CMOS input is so sensitive it will try to remember the last signal state it was in, and may do so for minutes or even hours. You can even get the local radio station to appear on unused CMOS inputs.

So, you always should tie unused inputs somewhere. If the input affects the logic of what you are trying to do, you would normally tie it to the supply or ground as needed to meet the logic required. For instance, many integrated circuits have chip-enable pins that are enabled by grounding them. It is also possible to tie one input to a logically similar second one. If you have a two-input NAND gate, you can tie both inputs together to convert this logic block into a one input inverter.

Good practice says that you always tie all unused CMOS inputs either to ground or to the positive supply, depending on the logic needed. This is very important for low power CMOS circuits, since a

"floating" input that gets half the supply voltage on it can dramatically increase the power used by the circuit and be logically destructive as well.

Although most people simply tie unused LSTTL inputs directly to the +5-volt line, you really should do this through a pullup resistor to keep funny things from happening when you first apply power. Similarly, if you remove power from a CMOS circuit but are still driving it from very stiff or low impedance signals, you can damage the integrated circuit through internal latching.

It obviously pays to watch these details.

Occasionally, circuits might have their logic ones at ground and their logic zeros at +2.5 or +5 volts. If so, simply go along with the rules. More details on the use rules of LSTTL and CMOS appear in the Howard W. Sams *TTL Cookbook* (21035) and the *CMOS Cookbook* (21398).

At any rate, most micro level interface is concerned with LSTTL, CMOS, and NMOS devices and their intended signal levels. Be sure you know what these levels are and what is legal in the way of input and output signals.

Naturally . . .

DON'T EVER TRY TO INPUT A HIGH SUPPLY VOLTAGE, A NEGATIVE VOLTAGE, OR THE AC LINE DIRECTLY INTO A MICRO'S INTERFACE PORTS!

If you must interface such signals, be sure to put some level two interface external circuitry between the source and the port to chop things down to size and to provide safety isolation.

One exception: carefully controlled negative voltages are permitted into specially designed RS-232-C ports.

"LESS THAN A PORT" OUTPUTS

There are several good ways to get micro level signals into and out of microcomputers. Many of these input and output methods involve *ports*. But there are some simple and sneaky ways of outputting a single bit from a micro that do not need a full-blown port interface. Three of these methods are the *address flasher*, the *address toggler*, and the *soft switch* . . .

ADDRESS FLASHER—A simple circuit that does something anytime it is addressed from its address space location or locations.

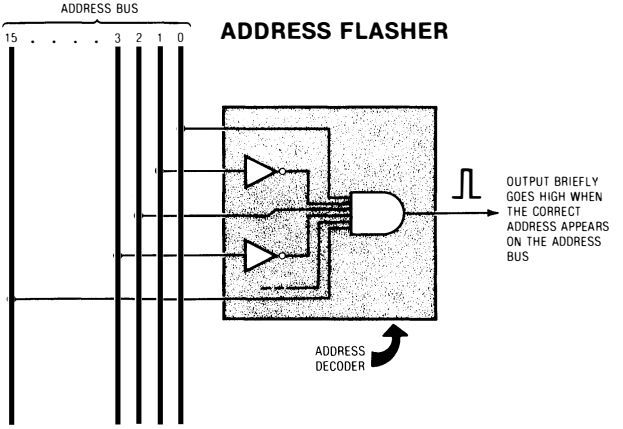
ADDRESS TOGGLER—A simple circuit that *changes* its state anytime it is addressed from its address space location or locations.

SOFT SWITCH—An address flasher with memory. Usually a flip-flop or latch that sets on one address and resets on a second address.

An important advantage of these "less than a port" output schemes is that they do not have to get involved with the data bus or any of the working registers of a micro. Thus, you can get a single-bit output through any of these without changing anything important in your microcomputer. This output can go to the outside world or be used to change the operating mode of your micro.

Disadvantages of the "less than a port" output schemes are that they are limited to a single bit line and normally can output only. They are also not mainstream, are very hardware intensive, and are very system specific.

The address flasher is the simplest of the three "less than a port" I/O schemes. Here's an example . . .



What you do is decode the address bus to a unique state or group of states. Anytime you hit one of the magic addresses, the output of

the decoder briefly goes high and can be used to control something. That something can be a line to the outside world or anything inside the system that needs a brief pulse to attract its attention, such as a handshaking reset or a strobe.

Although we have shown an "active high" output, you could instead use "active low" decoding if you prefer. In fact, active low decoding is much more common inside microcomputers, since many chip-select pins on many integrated circuits are active low.

We have shown the address decoder as a bunch of inverters and a single AND gate. As we saw in Volume 1, you can decode any address in a 16-bit or 65536-word address space with a single 16-input AND gate and sixteen or fewer inverters. Normally, of course, you use some more elegant way of decoding addresses, such as splitting the high address lines from the low ones and separately decoding each half. Often you may find that partially decoded addresses are already available from other parts of the system and can be partly reused here.

You don't absolutely have to decode to a unique 16-bit address. For instance, if you decode only the top four address lines, the address flasher will activate on any of 4096 consecutive addresses. If you decode all but the bottom four address lines, the flasher will activate on any of sixteen consecutive addresses. This may simplify decoding, but does so at the cost of wasted address space locations.

It is always a good idea to include some control lines in your address decoding. There will be times when the address bus has glitches or invalid addresses. You get around this by gating an "everything is legal" signal into your address decoder. The signal to use depends on the system and the micro school in use. It might be a $\overline{\text{VMA}}$ or *Valid Memory Address* signal, a R/$\overline{\text{W}}$, or *Read/NOT Write* signal, a $\overline{\text{RD}}$ or *Read* signal, a clock phase such as $\phi 1$, or whatever.

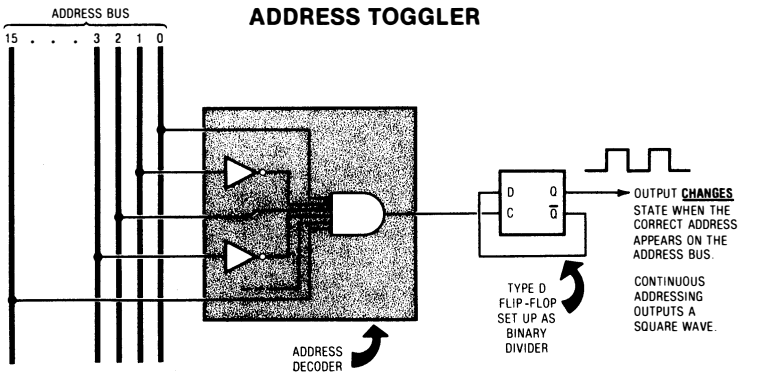
The Apple II computer uses many address flashers. One resets its keyboard strobe so a character gets entered only once. A second resets the game paddles at the start of a paddle measurement. Yet another address flasher is a strobe that activates I/O add-ons via the game connector. As a example, the keyboard strobe tells you when a key has been pressed but not used. After a keycode is accepted, the key strobe is reset for the next keystroke. Only a partial decoding is used on older Apples, so any of the sixteen addresses of $C010 through $C01F may be used. In this case, fifteen of these locations are not needed and are normally not used, but the decoding hardware ends up simpler. The Apple IIe upgrade has reserved these "extra" addresses for other uses.

You can activate an address flasher with any op code that addresses this location. You can write to the magic location, read from it, perform logic on it, or test it. For instance, a BIT $C010 on the Apple II will reset the keyboard strobe. You must, of course,

make sure that the instruction you use will not destroy any valuable registers or flag values.

The obvious route of using the address flasher magic location as a "write only" memory will work on most micro systems without hurting any register or flag values. But, owing to a multiplexing quirk on the Apple II, you can sometimes get *two* address pulses out if you store to an Apple address flasher. This can cause trouble. Load commands and BIT tests do not have this problem.

One interesting variation on the address flasher puts a binary dividing flip-flop on the flasher's output. This gives you a circuit called an *address toggler* that *changes* state every time you address it . . .



When you address the magic location, the output changes state. Address the location again, and the output changes once more, going from zero to one or vice versa. To output a square wave, you continuously address the magic location at *twice* the output frequency you want . . .

DOING IT:

Why twice?

The address toggler is rather limited in what it can do because there is no way to tell what state the output is in at any given time. Nonetheless, there are a few interesting applications.

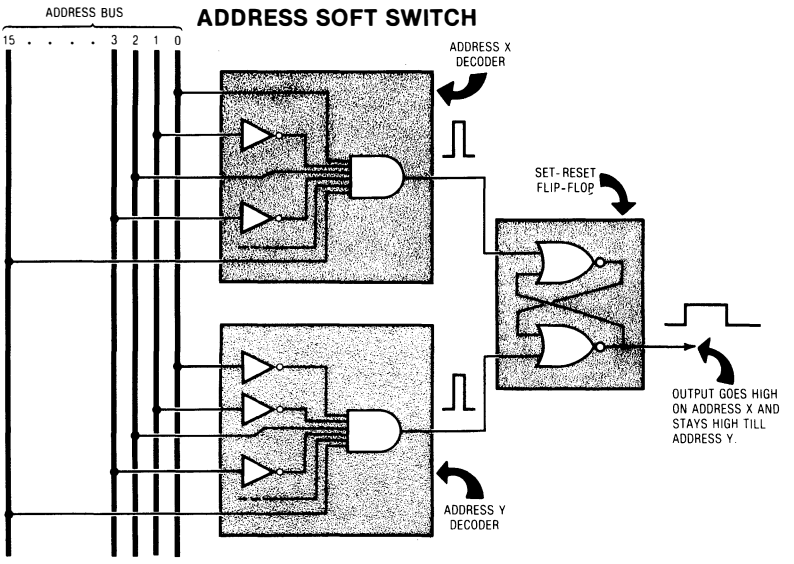
In the Apple II, there are two address togglers. One drives the cassette output from location $C020, and the second drives a speaker from location $C030. Thus, you can either save programs to cassette tape or generate tones or voice for your speaker without tying up any

ports and without worrying too much about what the data bus, the accumulator, the flags, and most registers are up to. You can also "liberate" these locations for your own special output uses.

A BIT test is the quickest and easiest way to snap an address toggler into the other state. On most micro systems a simple store to the magic location will do the trick. But . . .

> **Because of a quirk in the Apple II multiplexing, you cannot STA or do any other store to either address toggler location.**
>
> **What happens is that the address gets flashed TWICE if you try this, putting the output back to where it was a fraction of a microsecond before.**
>
> **Use the BIT command instead.**

Our final "less than a port" output scheme uses two address decoders to drive a soft *switch* . . .



Here we have two decoders set to two different addresses. Address X *sets* a set-reset flip-flop and drives the output high. Address Y *clears* the set-reset flip-flop and drives the output low.

The output stays low till address X is hit and then stays high till address Y is flashed.

There are eight soft switches on the older Apple II. Four soft switches are intended for internal use. These pick text versus graphics, display page one versus display page two, HIRES versus LORES, and full versus mixed graphics. Four soft switches intended for external use activate any of four annunciator outputs. These outputs are reached by way of the game paddle connector.

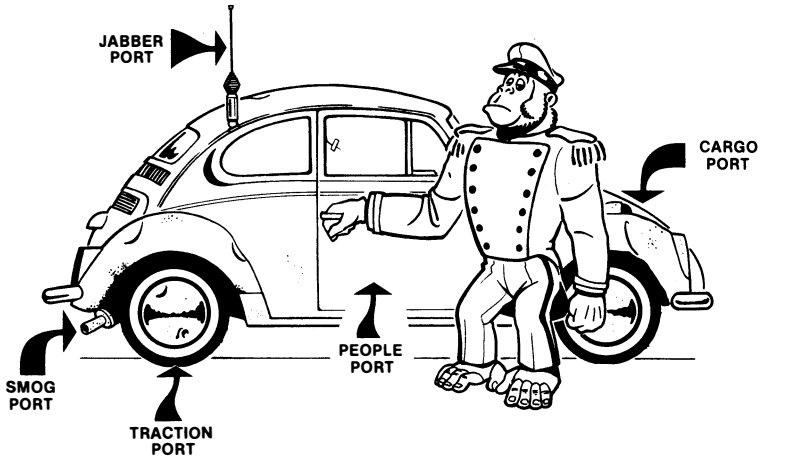
Two fine points. Note the extra inverter on the bit line one of the Address Y decoder in the last figure. Most often the addressed soft switch locations will be one away from each other in the address space. Obviously, a different address is needed to set than to clear the flip-flop. As a use example, the Apple II command BIT $C058 clears annunciator zero, and BIT $C059 sets annunciator zero. The output from this soft switch is routed to the game paddle connector.

Second, while you cannot really tell what state the output of a soft switch is in at any instant, you can always put the output where you want it by whapping the correct address. It's a good idea to route a system reset signal into your flip-flop so that it starts off in a known state and stays that way till you change it.

These "less than a port" schemes are all simple ways of getting an output signal line or two that is useful to the outside world or of changing internal operating modes. But to be able to input as well as output, and to be able to select what data we are going to output, we really have to use . . .

REAL MICROCOMPUTER PORTS

The usual ways we get signals into or out of a microcomputer involve *ports* of one type or another. Ports are by no means limited to micros, though. For instance, how many of what types of ports can you find on a car?



There are many different ports available on a car. All of these serve different uses and act in different ways.

Some ports are intended for input only. The gas cap and the oil filler cap are for inputting fluids into the car. Very ungood things will happen if either of these ports tries to output anything. Similiarly, the exhaust pipe is normally an output only port. Nasty things happen if you input water or mud or anything else by way of the exhaust pipe.

Most of the other ports on our car work in either direction. We call these *bidirectional* ports. For example, we can use the doors to either input or output people. We can open the VW's front hood to input or output cargo.

How about the radio antenna? This depends on the radio. If you have a plain old AM/FM radio, the antenna only receives, acting as an input-only port. But, if you have a CB radio, a mobile telephone, or a ham rig in your car, then the antenna will be bidirectional, acting as an input port when you receive and an output port when you transmit.

What do the wheels input or output?

DOING IT:

Show how the tires of a car are bidirectional ports that input or output in at least six totally different ways.

How do the front tires differ from the rear ones?

Microcomputers also have ports. These ports are the main ways a micro has to input and output signals to the outside world. Let's review some port stuff from back in Volume 1.

A port consists of the circuit hardware needed to get signals into or out of a microcomputer. Ports are normally accessed with software, reading from port to micro and writing from micro to port. Additional software may be needed ahead of time to "teach" the port what it is to do.

Parallel ports use parallel words and have everything there all at once on many side-by-side lines. Serial ports use serial words that go over a single line or channel, with the individual bits taking turns in time sequence.

There are many different ports available on a car. All of these serve different uses and act in different ways.

Some ports are intended for input only. The gas cap and the oil filler cap are for inputting fluids into the car. Very ungood things will happen if either of these ports tries to output anything. Similiarly, the exhaust pipe is normally an output only port. Nasty things happen if you input water or mud or anything else by way of the exhaust pipe.

Most of the other ports on our car work in either direction. We call these *bidirectional* ports. For example, we can use the doors to either input or output people. We can open the VW's front hood to input or output cargo.

How about the radio antenna? This depends on the radio. If you have a plain old AM/FM radio, the antenna only receives, acting as an input-only port. But, if you have a CB radio, a mobile telephone, or a ham rig in your car, then the antenna will be bidirectional, acting as an input port when you receive and an output port when you transmit.

What do the wheels input or output?

DOING IT:

Show how the tires of a car are bidirectional ports that input or output in at least six totally different ways.

How do the front tires differ from the rear ones?

Microcomputers also have ports. These ports are the main ways a micro has to input and output signals to the outside world. Let's review some port stuff from back in Volume 1.

A port consists of the circuit hardware needed to get signals into or out of a microcomputer. Ports are normally accessed with software, reading from port to micro and writing from micro to port. Additional software may be needed ahead of time to "teach" the port what it is to do.

Parallel ports use parallel words and have everything there all at once on many side-by-side lines. Serial ports use serial words that go over a single line or channel, with the individual bits taking turns in time sequence.

Input ports are intended only to get stuff from the outside world *to* the micro, and output ports are to get output commands or data *from* the micro to the outside world. Bidirectional ports can pass data in either direction, going from micro to whatever is out there, or vice versa.

To recap . . .

PORT—Circuit hardware that gives you the usual method of inputting or outputting small signals from a microcomputer.

PARALLEL PORT—A port that has many output or input lines, with all the bits in a word present all at once.

SERIAL PORT—A port that has only a few output or input lines, with all the bits in a word taking turns in time sequence, one bit at a time.

INPUT PORT—A port that is intended only to input signals from the outside world to the micro.

OUTPUT PORT—A port that is intended only to output signals from the micro to the outside world.

BIDIRECTIONAL PORT—A port that can be used in either direction, inputting to the micro or outputting from the micro.

Parallel ports are usually very fast, need lots of wire, and have many side-by-side channels. Serial ports are usually much slower, need little wire, and offer one or only a very few channels over which digital signals must proceed on a bit-by-bit basis in time sequence. Parallel ports are more common *inside* a local microcomputer system, and serial ports are more commonly used *between* microcomputer systems and the real world. Thus, you are likely to use a parallel port to interface a floppy disk and a serial port to access a modem connected to the phone line for distant communication.

Surprisingly, ports are *not* normally available as ready-to-go pins on the microprocessor chips of most micros from most micro families . . .

Most microprocessors from most micro families do NOT have port lines available and ready to go.

You usually have to add your own ports using extra hardware and software.

There are several good reasons why the average microprocessor CPU does not have ready-to-use ports. First, port lines waste package pins and gobble up space inside the chip that could be used for more important things. Second, adding your own ports gives you much more flexibility, because you can add what you want, where and how you want it. Third, it is no big deal to add another chip or two for I/O to a system that already has bunches of chips in use for CPU, RAM, ROM, and support.

The fourth and final reason is probably the most important. The address bus and the data bus of a microcomputer both are involved with critical timing and very crucial access rules. You don't want just anybody grabbing a bus and running all over the place with it whenever they feel like it. Instead, you want to be able to control very carefully who gets bus access when, all the while following some very strict use rules.

There is one major exception to micros needing external ports as added hardware. The single-chip micros intended for dedicated and high volume applications usually have many I/O lines available and ready for use. This gives you the convenience of immediate outside access, but at the fairly heavy cost of having limited RAM and ROM available, often a smaller address space, and difficult expansion . . .

Single-chip micros do have port lines available and ready to go.

The tradeoff is that these chips have limited RAM and ROM, may have a smaller address space, and can be hard to expand.

So, it pays to find out how you add ports in a microcomputer system, because this is just what you have to do in most multi-chip applications. Even if you use a single-chip dedicated micro, the use

rules and the behavior of the ports will be pretty much the same as the add-on ports of the larger systems.

There are two ways to go when you add micro ports. You can use a *simple* port or a *fancy* port . . .

SIMPLE PORT—A port that has just enough hardware to do one fixed I/O task.

FANCY PORT—A port that is programmable or flexible enough to do many different I/O tasks many ways. Other non-I/O features may also be included on-chip.

Simple ports have just the bare minimum of hardware they need to shove something out to the cruel world or get stuff back. They are always ready to use without any software hassles. Probably the simplest ports ever used on any microcomputer system appear on the PAIA 8700, where the ports are built from bits and pieces of CMOS, resistors, and transistors, none of which cost over a quarter.

Simple ports are great for learning the fundamentals of how to get outside access of the data bus on a microcomputer system, but they have severe limitations. For one thing, you are stuck with just what you have. There is no obvious way to change port direction, to find the last thing you sent out, or to help the communication process through "handshaking." More on this shortly.

You cannot self-increment or self-decrement a simple port. It is also a waste of dollars and space to use simple ports, since you can build a thousand times as much performance into a four-dollar integrated circuit as you can into a forty-cent one. So, simple ports are limited to trainers and the barest of cost-conscious controllers.

Most personal computers and most trainers use fancy ports instead. These are usually 24- or 40-pin integrated circuits that give you simple ports, plus all the stuff you need to teach these ports how they are to behave and how they are to interact with the micro. These ports are usually software controlled, so you can teach them many different tasks.

The teaching process is called initialization . . .

INITIALIZATION—The process of teaching a fancy port what it is to do.

Virtually all fancy ports have to be taught how to behave ahead of time, or they just plain won't work. Usually there is both software and timing involved in initialization. Most often, the same system reset pulse that gets the microprocessor started on the right foot also goes to a fancy port to get that port started in the right direction. Often this system reset pulse will clear all command modes in the fancy port and force all of the I/O lines into inputs.

At the very beginning of any program that uses fancy ports, you have to add initialization software to teach your ports what they are to do. This usually consists of a group of load immediate and store absolute commands that tell the port how it is to behave . . .

TEACHING COMMANDS—Software commands that MUST be included in the start of any program that makes use of fancy ports.

If you check into the data books and catalogs, you'll find a bewildering array of fancy ports available. These come with a mind-boggling number of control pins and very confusing use rules.

Where and how do you start?

We'll look at some specifics shortly, but the first rule is that the fancy ports are usually built by one manufacturer for use on their particular chip family . . .

Fancy port chips are intended for use in one particular micro school.

Although you can use any fancy port on any micro system, the further afield from the "parent" school you go, the more ridiculous the interconnection hassles get.

This says that you should use 8080 ports on 8080 systems, 6502 ports on 6502 systems, and VCIW ports on VCIW systems. Sometimes a port IC is good and useful enough that you can move it over to another system without too many hassles, but this is usually an exception. The forthcoming *Intel* 8212 is simple and flexible enough to be used with many micro systems, although it is intended for 8080 school use.

So, the best rule on mixing and matching fancy ports is not to do it. Sometimes there is an advantage to using one school's fancy ports on the "wrong" type of system. The hassles get almost unbearable if you try to interface a port with separate address and data lines to a micro school that needs multiplexed data and address lines, or vice versa. Good luck.

It turns out that many of the features and a few of the pins on very fancy ports are rarely if ever used. But, if any of these pins end up at the wrong logic level, the port may shut itself down entirely or go into a very strange use mode. Somehow you have to sort out the essential stuff from the geegaws. The best way is to steal the plans . . .

The best way to understand a fancy port is to find a trainer or a micro system that is using the port and play with it.

Hopefully the trainer or micro system will have the port connected in a more-or-less useful way. This lets you separate the initialization and use software from the connection problems. It also shows you what pins go where.

Naturally, it also pays to study the software use examples that are related to the port. Once you find out how others connect and use their fancy ports, then you can go and do your own thing with your ports anyway you like.

Let's continue our study of ports by starting with simple parallel ports and then going to fancy parallel ports. After that we'll find out how to minimize the number of port lines in use when we interface things like keyboards and displays. From there, we'll go on to serial ports and then take a brief look at the "more than a port" chips that can do lots of specialized stuff for us. That should get us through level one interface.

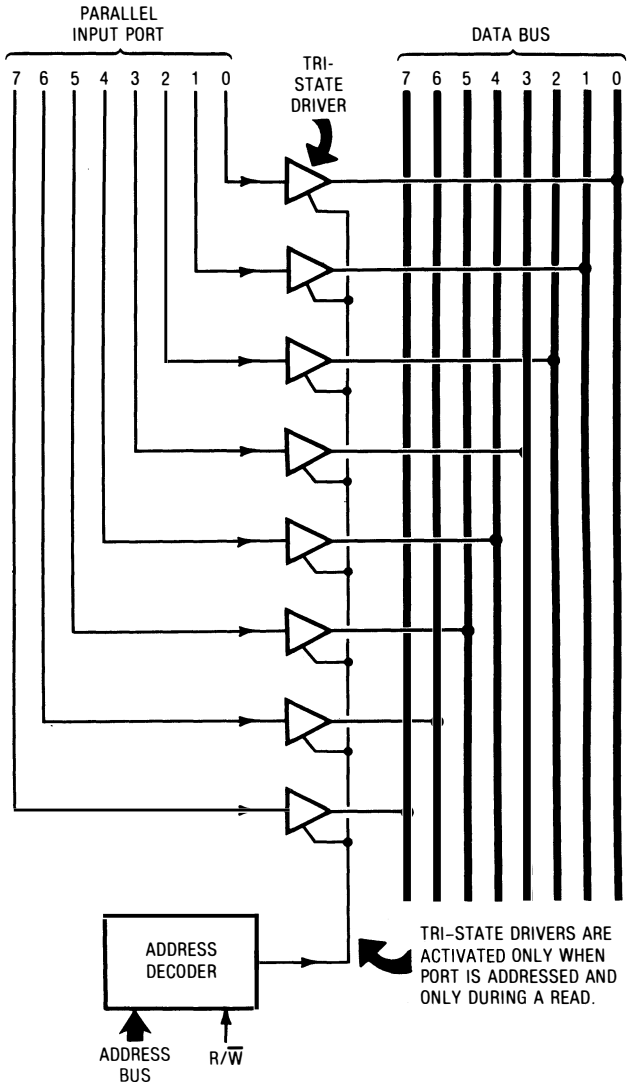
I'll assume you want to add your own ports to a new micro system. If you are simply using ports on an already built and tested system, all you need are the use rules rather than the gory details that follow.

SIMPLE PARALLEL PORTS

Let's look at a simple input port first. All we need is some way to connect an outside world 8-bit digital signal to a data bus at just the right time, and we are home free.

Like so . . .

SIMPLE INPUT PORT



There are two areas to our simple input port circuit. At the top we have eight tri-state drivers that will connect our eight input port

lines to the 8-bit data bus when they are activated. Naturally, we have to use tri-state access since the data bus will be used by other things at other times, and our outside world lines must stay invisible until used.

At the bottom we have an address decoder. The simple port will have an address set aside for it somewhere in the address space. When that address is activated as part of a load or move op code, the address decoder will activate the tri-state drivers, connecting the input port to the data bus.

It is very important to connect the input port to the data bus for *only* the brief instant when the CPU is ready to accept the outside world info onto the data bus . . .

You CANNOT decode just any old address to gain data bus access.

You MUST combine the address decoding with timing and control signals to grab the data bus ONLY and EXACTLY when allowed.

Note the R/$\overline{W}$ line going into the address decoder. Other systems may use other control lines and one or more clock phase timing signals, but the key point is that *you must turn the tri-state drivers on* only *at the instant that the micro's CPU is willing to accept data*. Activation at any other time will end up with bus contention and severe system problems.

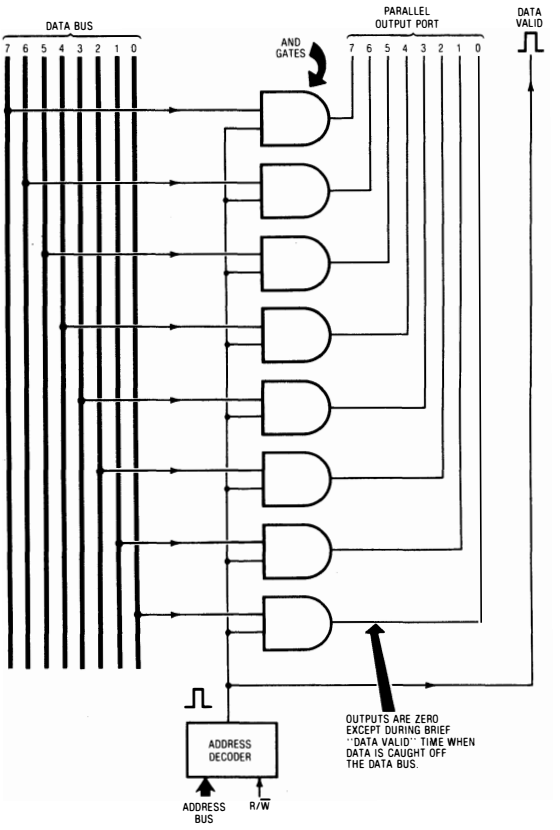
While you could use any old tri-state buffer or gate in a very small microcomputer, it pays to use heavy tri-state drivers in anything bigger. This way, you are sure that you can drive anything else hung on the data bus and that you will get the best possible noise immunity. The 74LS541 is one good choice.

So, a simple input port consists of eight input lines going to eight tri-state drivers. These drivers are turned on only when the input port is addressed and then only at exactly the right time needed to safely connect the input port to the data bus.

As a reminder, this is a *memory mapped* input port. The microcomputer can't tell the difference between a RAM, a ROM, and an I/O location in the address space, and all are reached with similar op codes as part of a working program. Although special port access commands are available on some micros, this special use is not mainstream.

Here is a simple but rarely used output port . . .

SIMPLE PULSED OUTPUT PORT (RARE)



All we have done here is "insided out" the simple input port, putting the data bus on the left and the parallel output lines on the right. If these lines are the only thing going to the outside world, we may not have to worry about tri-state access, so we have shown plain old AND gates.

What the circuit does is grab what is on the data bus and throw it out the port whenever the magic address appears and whenever system timing says the data is valid. Each AND gate gives an output only when the address line is high.

The output will be a *brief* flash of data that is there only when the address decoder says the data is valid. Most of the time you get all zeros at the output port. Only when it is addressed do you get a brief flash of valid data.

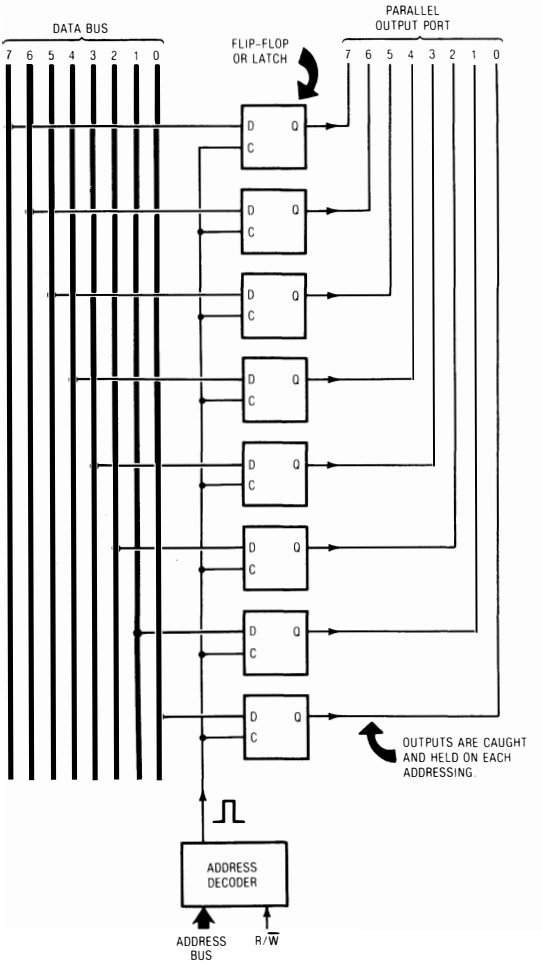
Catch it quick. Whoops, there it went. Gone.

This old circuit, called a *pulsed I/O port* by the dino minicomputer people, is more-or-less worthless. All it is good for is to convince us that we have to be very careful to filter out all but the intended values we want off the data bus before we output it.

For most uses, we want to catch and hold the old output data. We keep this data till we want new data and then keep the new data till it in turn gets updated. Obviously, we need some storage. This storage can be in the form of eight type D flip-flops or else eight hold-follow latches.

Here is your everyday, simple latched output port . . .

SIMPLE LATCHED OUTPUT PORT



The address decoder works the same way as it did before. At the instant the address is valid, the data on the data bus is "caught" by the D-flops or latches and is then held. The output lines continuously output the data bus info saved during the last time the port was updated.

Some specifics. Suppose we want to output a %0010 1110 pattern to our output port that has an address of $E035. That pattern translates to hex $3E, so somehow you find a $3E somewhere in your computer, move it into a working register, and then store the $3E to location $E035. On the 6502, an immediate load followed by an absolute store is the simplest way to do this, using LDA #$3E followed by STA $E035.

On any system, you first decide what data you want to output and then move or store that data to the magic address that will activate the storage latches and update the output.

The storage latches, in turn, will hold the output data from the time the CPU stores the data until the time the data is actually used by the outside world. Depending on the application, this holding time can range from microseconds to months.

There is one critical little detail that you must pay close attention to if you are designing a port like this on your own. You have to understand *exactly* how the latch behaves and *exactly* when it will catch valid data. The time of catching the valid data must, of course, be the same time that the valid data you want appears on the data bus.

Remember that there are two common types of storage, the *clocked* type D flip-flop, and the *hold-follow* latch. With the type D flip-flop, valid data is clocked into the device the instant that the clock changes a specified edge. Most often, this is the ground-to-positive transition, or the *positive edge* of the clock. You want this clocking positive edge to be somewhere just beyond the *middle* of the time the data is valid on the data bus. If it's too close to the beginning or the end of the data valid time, you get into all sorts of temperature problems and unit-to-unit ungoodness.

On the hold-follow latch, one clock level follows and the other clock level holds. You must switch from follow to hold at the instant that you are sure you are *already* holding valid data.

As an even nastier sub-detail, almost all clocked logic devices have what they call a *setup time*. This is the time before or after clocking during which data is guaranteed to be what you think it is. Most modern devices have a zero setup time, so that what you grab is what you get. But always check the data sheet. If the setup time is *positive,* it means you have to *wait* awhile until

after things are good before you grab them. If the setup time is *negative,* you have to be sure you grab valid data sometime *before* it goes away. To sum this up . . .

> **On any latched output port, make sure you grab the data ONLY when what you think you have is what you thought you really wanted.**

The output port data will be continuous. It pays to make your output data all zeros, or some other benign value, early in your program so that funny things don't happen before you start actually using the port. You could add tri-state drivers to your output port if the port is to share its information with other sources, but this is usually not necessary. If you do add these drivers, they will be controlled by the outside world and activated as needed.

Note that you cannot self-increment or self-decrement a simple output port, since there is no way to read what is already stored in the port back into the micro. You can fake port incrementing and decrementing by keeping a copy of the port values in another register or RAM memory slot.

THE 8212

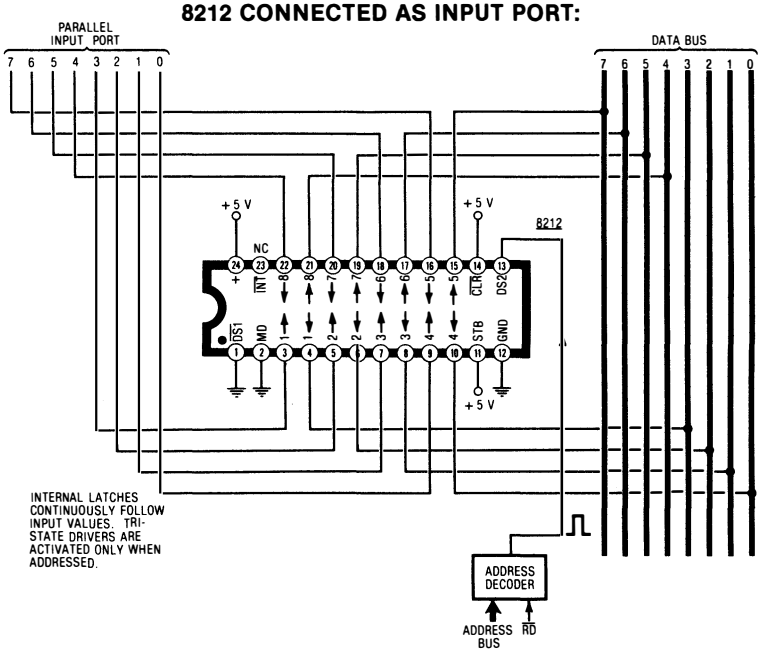
The *Intel* 8212 was one of the first fancy port chips available. This device can be used either as an input port or as an output port, and it provides some optional handshaking features. Unlike many newer fancy port chips, the 8212 is *hardware* rather than software programmable. This means that the 8212 is always ready to go when you plug it into your circuit board but you cannot change what the chip does as part of your program.

The 8212 is general enough that it can be used with almost any micro family or as a general "port manager" for peripherals or add-on cards.

Inside the 8212, you will find eight hold-follow latches, each of which outputs to a tri-state driver. To make a long story short, the chip is controlled by a *mode* pin that decides whether the 8212 is going to be an input or an output port. Mode goes into pin two. Wire this pin positive to build an output port, or ground it to form an input port.

Here's an input port using the 8212 . . .

8212 CONNECTED AS INPUT PORT:



The eight parallel port lines are wired to the inputs of the eight storage latches, and the eight tri-state driver outputs go to the data bus. Note that the mode input on pin two is grounded.

In the input mode, there is a strobe pin, or STB, that decides when new data is to be accepted by the latches. If you make STB high, this makes the latches continuously follow the input. Normally, in an input port without handshaking, you do not want any latching. Instead, you follow the input continuously. So, holding STB high defeats the latches and lets the input data transparently "fall through" as far as the tri-state drivers.

An external address decoder decides when to activate the tri-state drivers inside the 8212. This decoder provides a positive pulse during a valid read time when the magic address is present. We've shown the 8080 school $\overline{RD}$ control signal rather than 6502's R/$\overline{W}$ here, but you can use any signal that makes sure it is legal to grab the data bus for an input.

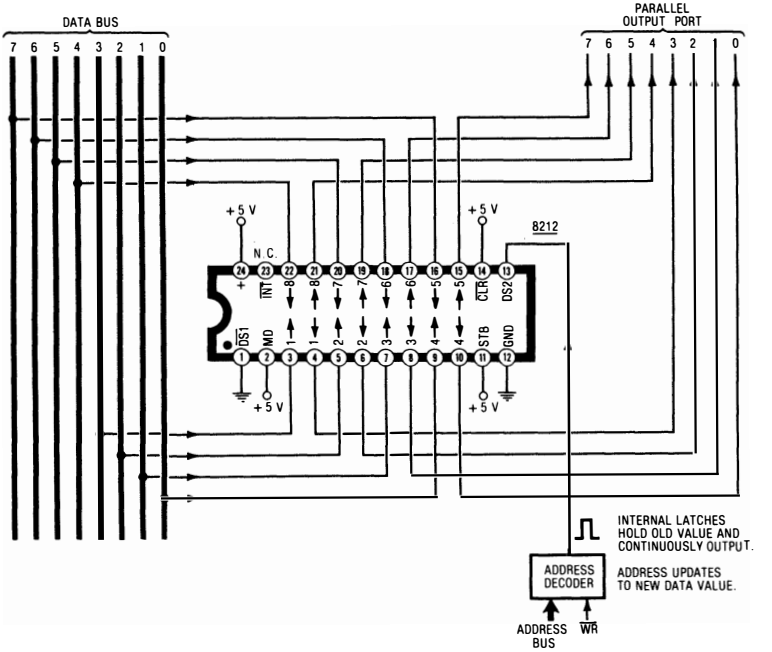
Operation is the same as our earlier simple input port. Address the chip, provide a valid read signal, and the parallel input lines get

temporarily connected to the data bus. The CPU in turn grabs the information on the data bus and does something with it under program control.

There are several more pins on the 8212. The +5-volt supply goes into pin 24, and ground goes to GND on pin 12. The remaining three pins are unused in this circuit, but you must keep DS1 and CLR at ground, and you should not connect INT. More on these later.

Here is how you wire the 8212 as an output port . . .

8212 CONNECTED AS OUTPUT PORT:



This time, the mode pin is positive. The mode pin changes the inside logic on the 8212. In the output mode, the address decoder decides when to update data into the internal hold-follow latches, and the tri-state drivers are continuously enabled by the high STB line.

All this magically changes our 8212 into a device that latches data off the data bus *when addressed* and then continuously outputs the old data to the parallel output port.

So, with one set of connections, the 8212 behaves as a simple input port and, with a different set of connections, the 8212 behaves as a simple output port. As with simple ports, though, you still cannot self-increment or self-decrement what is stored in the 8212.

Before you get all excited about how easy it is to change only the mode pin, note that the eight data bus lines go to the 8212's *input* pins in the *output* mode, and vice versa. Thus, to get from an 8212 as input port to an 8212 as output port, you have to interchange at least sixteen pins, besides flipping the mode command.

What you really have in the 8212 is a device that can be plugged into one of two sockets on a circuit board. Put it in socket "A" to output and into socket "B" to input . . .

The 8212 changes from an input port to an output port only by making extensive wiring changes.

Fancier chips are needed if you want to input or output to the same port under program control or want to self-increment or self-decrement latched data.

The 8212 is very versatile whenever you want to output only to a port or, elsewhere in your circuit, when you want to input only from a port. Since no software initialization is needed, the 8212 is always ready to use. Its also cheaper than many of the fancier ports.

You have several fancier use options for the 8212, provided you use it only to output or only to input.

There are really two address input pins available. One of these is DS2 on pin 13 that must be high to activate. The other is $\overline{DS1}$ on pin 1 that must be low to activate. Inside the chip is a "DS2 AND NOT DS1" logic gate. These two pins both have to be happy to activate the 8212. You can input a positive going address command into DS2 and keep $\overline{DS1}$ grounded as we have shown. Or, you can input an active low address command into $\overline{DS1}$ and keep DS2 positive. Or, you can route part of your address decoding into an active low $\overline{DS1}$ and the remainder into an active high DS2. Still another possibility is to route your "everything is okay to use" signal into one of these pins and a straight address decoding to the other. There is a CLR for clearing available as pin 14. If you con-

nect this to your system reset line, the hold-follow latches will be emptied during power-up. This keeps you from outputting garbage to your port before you actually use the port. It is always good to prevent garbage from appearing anywhere, anytime you can help it.

handshaking

Here is one big problem with any simple input or output port: there is no way the micro can be sure that inputs are accepted only once and that outputs are used only once. For instance, suppose you have an ASCII keyboard going to a simple input port, and you press key "J." How do you know that you used the J once and only once? How do you know that you didn't miss something in between? What if the J key is hit twice in a row? How can you tell?

To be absolutely sure that everything is used once and only once, you have to add *handshaking* to any micro ports . . .

HANDSHAKING—Any control scheme that makes sure micro port input and output data is used once and only once.

Normally, handshaking involves one or two signals over and above the eight port lines. Fancier systems speak of the *protocol* of using these lines . . .

PROTOCOL—The "who does what to whom" rules needed to make sure that handshaking works.

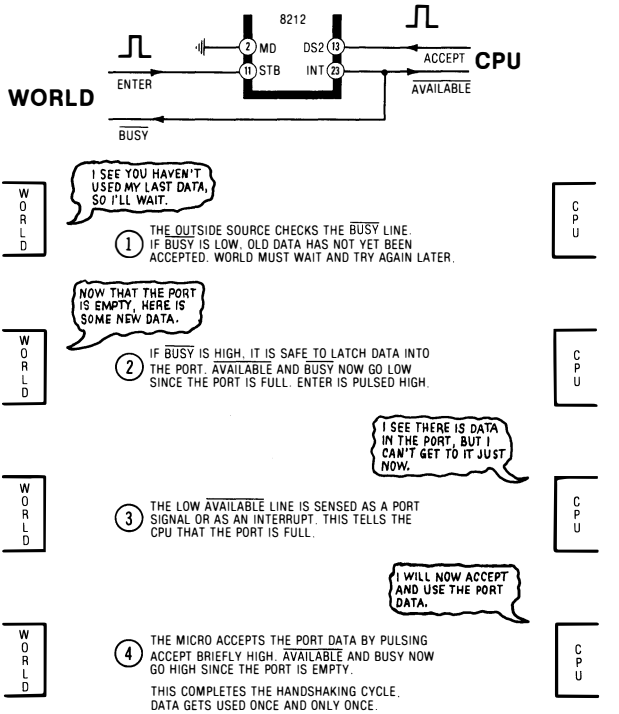
Handshaking can get very fancy. On serial RS-232 lines, it is mostly handshaking that ups what should be three lines into a 25-pin connector. And if any of those pins is in the wrong state or ignored, communications shut down completely till the problem is corrected. The theory is that no data at all is better than bad data.

Handshaking usually involves two storage areas. One is a register inside the port that will hold data till it can be accepted or used. The second storage area is a single flip-flop that can be set to signal when something is in the port and ready to be used.

To the thing filling the port, the flip-flop signal means "busy." To the thing emptying the port, the flip-flop signal means "available."

Let's look at a specific example or two. Here is how you handshake an 8212 hooked up as an input port . . .

8212 INPUT HANDSHAKING



The "port is full" flip-flop outputs on the INT line of pin 23. This pin is active low, meaning that a high signal here equals an *empty* port. For input, the INT pin goes to the CPU as an AVAILABLE signal and goes to the outside world as a BUSY signal.

If the port is not busy, the outside world can fill the port with an ENTER command that strobes the data into the port. This saves the

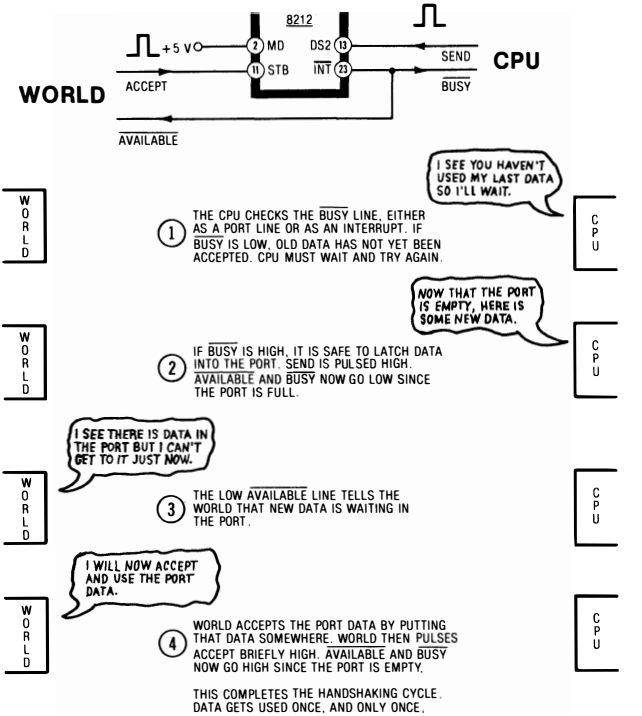
data into the port and at the same time drives $\overline{IRQ}$ low. The low $\overline{IRQ}$ tells the CPU that data is $\overline{AVAILABLE}$ and awaiting it, and tells the outside world that the port is BUSY and to hold up on anything new.

The CPU can use the $\overline{INT}$ line several ways. One way is to read this $\overline{AVAILABLE}$ signal at another port. Another is to let the $\overline{INT}$ line actually interrupt the CPU. Which method you chose depends on how important the data is and how often you have to access this input port.

Once again, handshaking involves a port that stores a data word and a flip-flop that holds a busy signal. Both sides interact with the port as needed to get the value once and only once.

The roles change around on output handshaking . . .

8212 OUTPUT HANDSHAKING



This time, the CPU checks to see if the port is still busy. If not, the CPU sends data by addressing the 8212. This latches data into the 8212 and drives $\overline{\text{BUSY}}$ and $\overline{\text{AVAILABLE}}$ low.

The low $\overline{\text{BUSY}}$ tells the CPU to hold up on anything new. The low $\overline{\text{AVAILABLE}}$ tells the outside world that data is awaiting it. World can then access the port data by reading it. The ACCEPT signal is sent out after the data is received. This clears the port for another cycle.

We've gone through a long, detailed analysis of handshaking in the figures to show you what has to be done. Proper handshaking will make sure that nothing is missed or used twice when inputting to or outputting from a microcomputer port.

Handshaking is not always needed. If you are sure that there is no way to miss or twice-use data, then you most likely don't need handshaking. If you output things much slower than they are used, then you probably don't need handshaking.

Once again, the 8212 $\overline{\text{INT}}$ line does *not* necessarily have to go into the micro's interrupt signal. This $\overline{\text{INT}}$ line is simply a busy signal that can be routed to a micro port or an interrupt line for input or to the outside world for output.

As an example of *not* using interrupts, you can input an ASCII code from a keyboard into seven port lines and use the eighth bit as a keypressed line. You then need one more port line from micro to keyboard to reset the key strobe. This makes sure each key is used once and only once, regardless of the time between keyclosures and repeated hits of the same key.

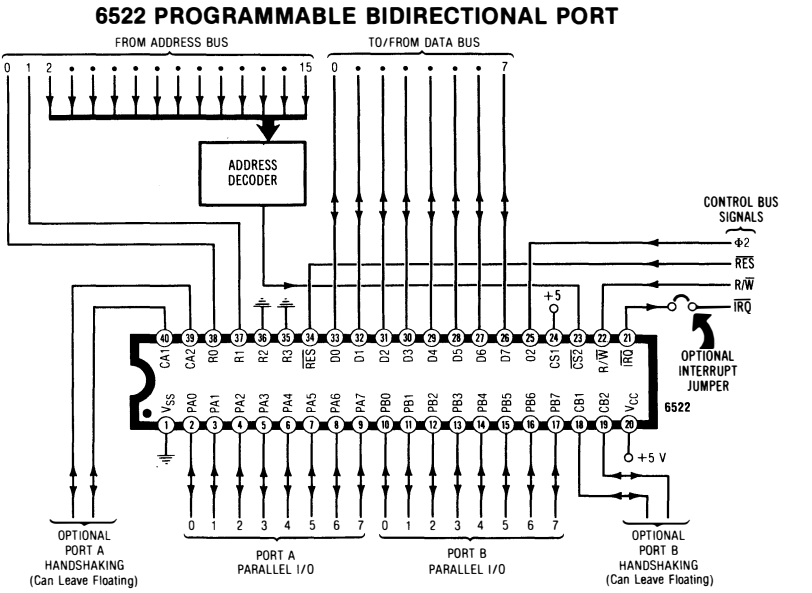
On the Apple II, an input port at location $C000 consists of the keyboard's seven ASCII bit lines and a keypressed line. An address flasher that whaps location $C010 is used to reset the keypressed line.

Enough said on handshaking. The details can get very hairy, and there are lots of different use possibilities. It's best to pick up handshaking details only as the need arises.

Let's go on to a fancier port chip called . . .

THE 6522

The 6522 is a 6502 school peripheral chip that gives you a pair of bidirectional 8-bit ports . . .



The port lines are arranged by eights into an A port and a B port. The supply power and system bus pins are connected the usual way. Four optional input or output lines intended for handshaking are called CA1, CA2, CB1, and CB2. These may be left unconnected for simple I/O uses.

The chip enable might be driven from an address decoder that activates on *four* addresses, ranging from BASE+0 to BASE+3. The lowest two address lines are routed to pins 37 and 38 so these four addresses can be recognized.

Here is what these four addresses do . . .

BASE+0 — Teaches the A port its data directions
BASE+1 — Sends data into or out of the A port
BASE+2 — Teaches the B port its data directions
BASE+3 — Sends data into or out of the B port.

You can think of the data direction addresses as *teaching* locations . . .

> **DATA DIRECTION REGISTER**—An address location inside a fancy port that remembers which port bit lines are to be used as inputs and which are outputs. Data direction registers are usually initialized once at the start of a program.

So, you use a data direction, or "teach" location to set up which port lines are inputs and which are outputs. Normally you do this only once at the start of your program, as part of an *initialization* procedure.

On the 6522, a *one* in a bit location in the teaching port makes that bit in that port location an *output*. A *zero* in a bit location in the teaching port makes that same bit in the port location an *input*. You can mix and match the input and output bits however you like, although it is often best to use the 6502 *high numbered* port lines for inputs and the *low numbered* port lines for outputs.

For instance, to make the lower three bits of port A all outputs and the upper five bits of port A all inputs, we would store an $07 to the teaching address BASE+0. We do this to put the bit pattern %0000 0111 into the data direction register, which teaches the port that the zeros are inputs and the ones are outputs. When it comes time to use the port, we would load or store from the actual A port location of BASE+1.

Note that it is best to do all your port work in straight binary so you can see and understand exactly what each bit line is up to.

The port location is the address where the data is actually shoved into or out of the machine. The teaching location is the address that remembers what each bit line on the shoving port is supposed to be doing.

One more time: The data direction register teaches the port lines which way to go and is taught once during program initialization. The port address is then used to get data into or out of the machine.

There are some fancier uses of B side port lines PB6 and PB7 that involve interrupts and handshaking. Do not use these B side lines for I/O until you understand how they work. There is also a counter and timer circuit built into this chip, along with a shift register that can be used to input or output serial data.

To activate all this fancy and sneaky stuff inside the 6522, you route address lines A2 and A3 into pins 36 and 35, respectively. This activates a total of sixteen internal registers that let you do all sorts of mind boggling tricks. Use details, of course, are on the data sheet

and ap notes. But don't get fancy till after you know how to use the 6522 as a pair of simple bidirectional parallel ports.

You *can* self-increment or self-decrement any 6522 port or internal location since the micro can both read and write to these locations. This can be most useful, and it greatly simplifies any software involved in I/O control.

There are lots of other fancy parallel ports available. Beware of the 6821 from the 6800 school, as this turkey does its port teaching in an astoundingly bizarre manner. The address location can be *either* a teaching command *or* a port read or write. A second address location acts as a switch that flips the first location between the two. Hairy. Also dumb.

Fancy parallel ports from other micro families include the 8255 from the 8080 school, and the Z80 PIO from the Z80 school.

Newer fancy parallel port chips usually combine ports with other features such as timers, RAM, ROM, or elaborate handshaking. Examples from the 6502 school include the 6530 and 6531.

Details on many of these chips appear in Volume 3 of this series.

THE SIMPLIFIED I/O DIAGRAM

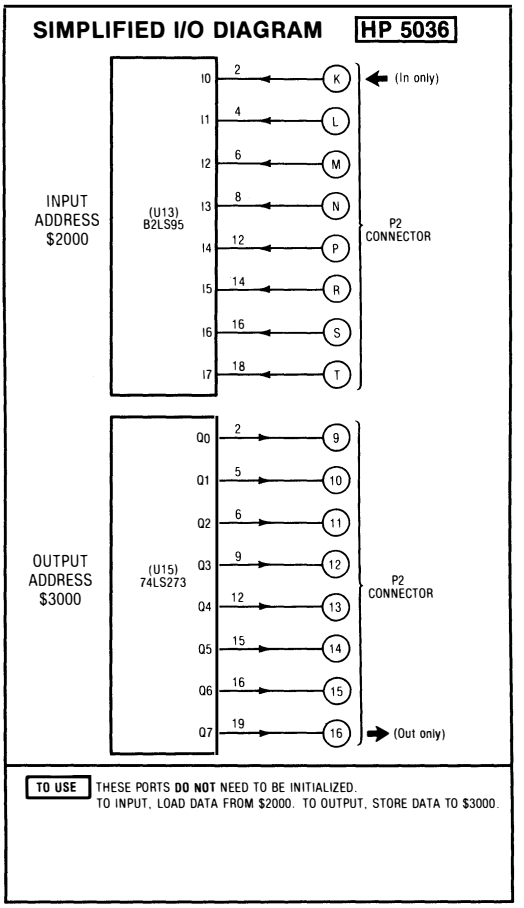
When you first try to use parallel ports on a trainer or a personal computer, you will find a bewildering mishmash of information on where the ports are located and how to use them. Chances are, the information you need will be in six different places, tough to find, and tougher still to understand. The way around this is to make yourself a *simplified I/O diagram* . . .

> **SIMPLIFIED I/O DIAGRAM**—A form that shows only the essential hardware and software information needed to use ports on a trainer or a microcomputer.

This simplified I/O diagram belongs in your *Micro Toolkit* of Chapter 7, along with its companions, the Resource Sheet and the Simplified Memory Map. There's a blank form that you can rip off for your own use at the end of this volume.

The I/O diagram should show you how many of what types of port lines are available, how to access them, where they come from, and how to control them with software.

Let's look at three quick examples. Here's the simplified I/O diagram for the HP 5036, a trainer from the 8080 school that uses simple, separate input and output ports . . .



On this trainer, we have 8 input lines on top and 8 output lines at the bottom, both accessed via connector P2. Looking more closely, we see that input I0 ("EYE-ZERO") enters the trainer via pin K of P2, and so on down the list.

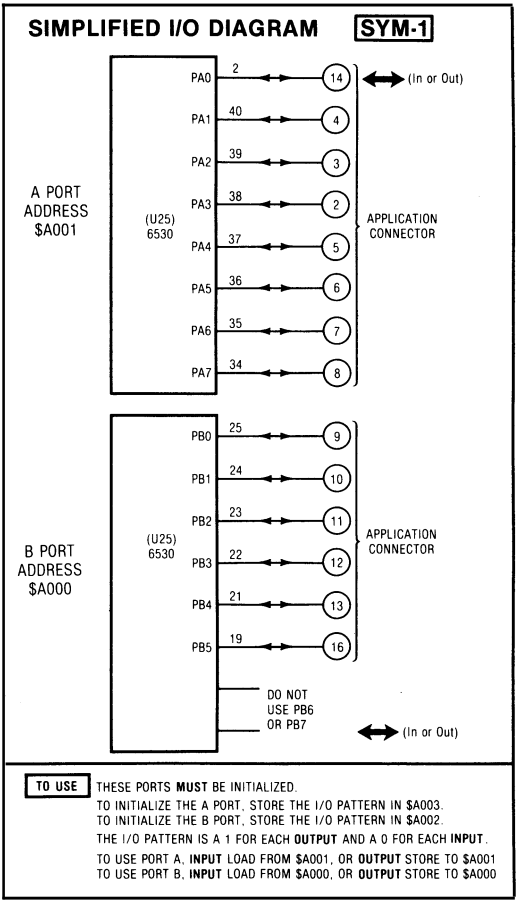
Integrated circuit U13, an offbeat 82LS95 octal bus driver, handles all eight input lines, and connector P2 pin K goes to pin 2 of this integrated circuit and serves as input I0.

At the left, we see port addresses of $2000 for input and $3000 for output. Since the ports do not need to be initialized, all you have to do is load from $2000 or store to $3000 to input or output.

Your simplified I/O diagram for any micro should show no more than you really need for what you are trying to do. Keep things sim-

ple and understandable. Show only how many lines there are, where to reach them, where they come from, how to address them, and how to initialize the port hardware—nothing more.

Our second example is the SYM-1, a 6502 school trainer with bidirectional ports . . .



This time, we have only fourteen I/O lines immediately available, and they come from a single IC, a 6530 called U25. The 6530 is a fancy upgrade of the 6522. All of the lines appear at the "Application" connector on the pins that are numbered as shown.

The "missing" two lines on the B side of the port are reserved for people who know what they are doing, since these can involve interrupts and handshaking.

Turning to the software, we see that these ports need to be taught ahead of time which lines are inputs and which are outputs. The A side is taught by writing to location $A003 with a pattern that has a one for each output and a zero for each input. The B side is taught by writing to location $A002 with the pattern you want on these lines. Just like the 6522.

As we have seen, this teaching process is called initialization, and is normally done only once at the beginning of a program.

After the ports are taught what they are to do, you can read or write to the A port through address $A001 and read or write to the B port through address $A000.

Once again, on either a 6522 or 6530, you teach to the teaching addresses and enter or remove I/O data from the port addresses. Other fancy ports from other families will behave differently, so be sure you understand *exactly* how you initialize and *exactly* how you can input or output data. Then spell these details out at the bottom of your simplified I/O diagram.

Often you'll find lots of other goodies inside a fancy port chip, such as handshaking, interrupt management, timers, shift registers, and similar stuff. Leave these advanced fancy items off your simplified diagram until a specific need arises for their use.

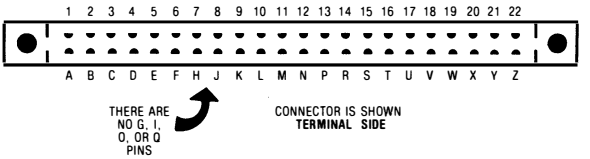
Keep it simple. That's what it's for.

As an aside, many trainers will use double-sided 44-pin connectors for I/O and expansion. This includes the 5036, the SYM-1, and many others.

The numbering and lettering on a two-sided 44-pin connector can cause confusion. The top is numbered from 1 to 22 and the bottom is lettered from A through Z, *omitting the letters G,I,O, and Q*. The A is under the 1 and the Z is under the 22.

Like so . . .

44-PIN CONNECTOR CALLOUTS



In other words, when you are working on the letter side, be sure not to count pins G, I, O, or Q, since these pins do not exist. Obviously, twenty-six letters minus these four leaves you with twenty-two pins.

One more obvious point that can prevent lots of grief. Chances are, the signal on pin 3 of this type of connector is totally different from the signal on pin C which is the "backside" pin immediately below pin 3. If you grab pin 3 with an alligator clip, you may short 3 to C and do anything from bombing the program to blowing up the trainer.

So . . .

DON'T ever short the top to the bottom on a connector that uses both sides of a printed circuit board as contacts!

Alligator clips that grab both sides at once are a no-no.

I almost hate to mention obvious things like shorting connectors. Once you've been around a student lab for a while, however, you learn that you can lead a horse to water, but it's only when you get it to float on its back that you have accomplished anything.

The "obvious" will nail you every time.

Honesty time.

You may have noticed how every figure in these volumes magically appears just when and where it is needed, without any forward or backward figure references. If you think this gets tricky, you are right. If you think this drives typesetters up the wall, you are even righter. And, hooboy, does it ever cost. Which is why you won't see it done this way very often.

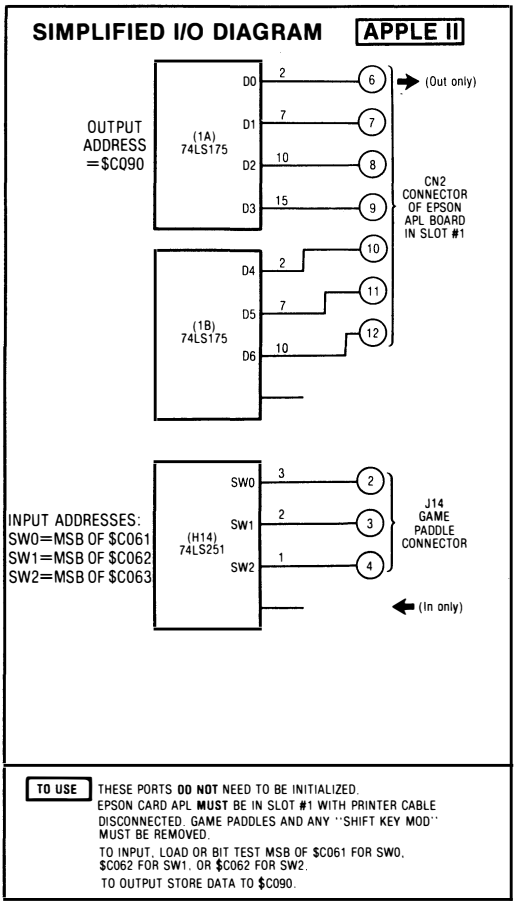
If ever.

But, surprisingly, thanks to typesetter Dave Sechrist, this is the only place in these volumes where a whole passage of padding had to be thrown in to keep half a page from turning up blank.

Just thought you'd like to know.

Where were we? Oh, yeah . . .

As a final example of showing only the essential on a simplified I/O circuit, here is one way you can coax three parallel input lines and seven parallel output lines out of a stock Apple II with an *Epson* printer card in slot 1 . . .



Now, the Apple's I/O system is mind-boggling and incredibly flexible. More so, in fact, than any other personal computer at any price. Look closely enough and you'll find there are at least *sixteen* I/O connectors, not just eight. And there are more than 4096 address locations set aside for exclusive I/O use.

But the Apple does not immediately give you a simple parallel input port and a simple parallel output port ready to go for the discovery modules of the previous chapter. Sure, you can add cards of

your own to pick up as many parallel ports as you want, but this may not be what you have in mind for a quick and simple experiment involving a few I/O lines.

Instead, you can use the *Epson* card and grab seven output lines as shown, provided, of course, that you disconnect the printer and make sure this card is in slot 1. And you can get three input lines read off the game-paddle connector as shown. Four output bit lines are also routed through the game-paddle connector, but these are soft switches controlled by address ports rather than true single-address output locations.

While you can definitely use these annunciator output pins as output bit lines, remember that it takes a *pair* of adresses to activate them, since these are really soft switches. You will need one address for the one and another address for the zero to be output. Most useful, but not quite mainstream.

By the way, if you really want thirty-two bidirectional I/O lines out of your Apple for a serious interface application, check into the *John Bell* 79-295 plug-in card. This card gives you a pair of 6522s and four output connectors at a most reasonable cost. Naturally, the simplified I/O diagram using this card will be totally different from the one we just looked at.

DOING IT:

Show how you can get an eighth output line off an Epson card and a fourth input port line on the Apple main board that fakes a "fourth pushbutton" input.

Make sure that you do not change any signals to the printer when you try this.

And, while we are doing things . . .

DOING IT:

Make up simplified I/O diagrams for several trainers and personal computers of your choice.

Add these to your micro toolkit.

Remember to keep the "simple" in simplified. Show only the essentials, but show *all* the essentials needed to use parallel port lines.

MINIMIZING PORT LINES

When you first attack a micro problem involving parallel ports, you may end up assuming that you need bunches and bunches of port lines. But there are lots of creative games you can play to cut down dramatically the number of port lines you need.

A rule . . .

> **If you have "N" port lines in use, you can input or output any of 2↑N states at any one instant.**
>
> **If you do not use all these states, you are being inefficient and may be wasting port lines.**

For instance, if we wanted to use a 16-button keypad on a micro, we might do the obvious and use one line for each button, for a total of sixteen input lines. Each bit on each line would be is sensed as a pressed key.

But our rule tells us that sixteen port lines should have 2↑16 possible states. Thus, it should be possible to sense 65536 different keys rather than only sixteen. This is so inefficient that it is ludicrous. If you are careful enough about it, you should need only four lines to sense sixteen keys pressed one key at a time.

There are lots of ways to minimize port lines. Some of these are very good ideas, and others are overkill that can cost you in the way of flexibility or outside hardware.

Anyway, here are some of the ways of . . .

MINIMIZING PORT LINES

( ) Use bidirectional ports
( ) Use an "XY" matrix array
( ) Use external encoding
( ) Use time multiplexing
( ) Time share
( ) Use oddball codings
( ) Go serial

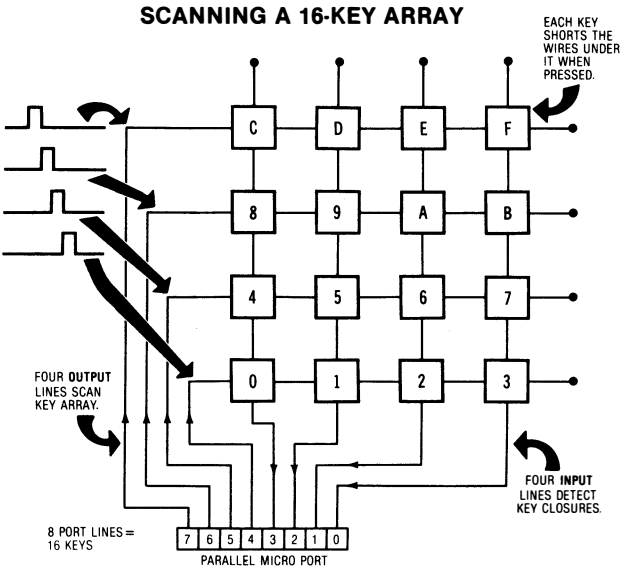
Let's look at a quick example of each method.

Bidirectional ports can save you hardware since you can mix and match input and output lines in any combination rather than "by eights." If you need only two input lines and six output lines, it takes only a single bidirectional port but otherwise demands two simple ports.

Whenever you have a bunch of anything, try to arrange that bunch into so many rows and so many columns. The total number of rows and columns is much less than the full count of the bunch you started with.

To throw some math at you, if you have "R" rows and "C" columns, you will need only R + C lines to access R * C things. Because products are usually much bigger than sums, you save on the lines you need. The bigger the bunch of things, the more you save.

For instance, here is how to arrange a 16-button keyboard into a 4 × 4 array. This combines a bidirectional port with XY techniques into what is called a *scanning keyboard* . . .



We use four output port lines and four input port lines. Only a single output line is allowed to go high at any instant. The output lines take turns going round and round as shown.

Each output line drives a *row* of four keys. Each input line senses a *column* of four keys. If a key is pressed, one row connects to one column. When the row is pulsed high, the column is activated and

sensed if a key is pressed. This scanning method is the standard way of adding a keypad onto a trainer.

To find a pressed key, you use software to pulse the top row positive and check for ones appearing on any column input. Then you pulse the second row down positive and again check for ones on the columns. Next, you hit the third row down and check for ones, and finally you hit the bottom row and check again.

If you get a one input, you have a pressed key. If you don't, keep scanning or go do something else for a while. When you do get a one, you can tell which key was pressed by checking in which column and at which point in the scanning software program the one appeared.

If your scanning software is properly designed, you will also get key debouncing provided at the same time.

Software debouncing is usually preferred over hardware debouncing these days.

DOING IT:

Show how to scan sixty-four keys on an ASCII keyboard.

What is the minimum number of port lines needed?

What is the most sensible number of port lines to use?

How can you handle the shift and control keys?

How can you switch from QWERTY to DVORAK?

Needless to say, it's downright stupid to use sixty-four port lines to sense sixty-four keys. Your first solution probably would be to use sixteen port lines, with eight output lines that pulse rows and eight input column lines that sense key closures.

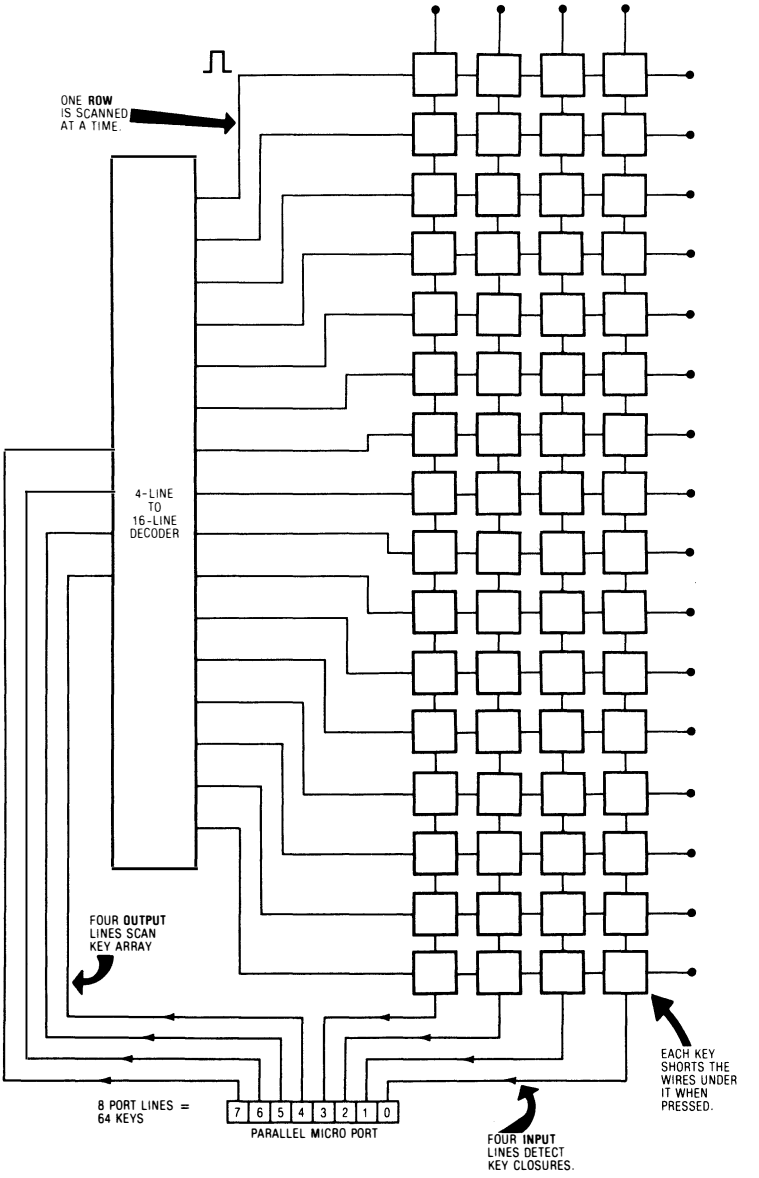
Simple and workable, but nowhere near minimum.

Sometimes we can add a small amount of external hardware to cut down the number of lines needed.

Note that we will always scan only one row of keys at a time. What if we take four output lines, add an external LSTTL or CMOS 4-line-to-16-line decoder, and use a 16 × 4 array instead?

Something like this . . .

SCANNING A 64-KEY ARRAY



We see that we can get by with only eight lines from a single bidirectional I/O port to handle all sixty-four keys.

We see that our eight port lines are split up as four input and four output lines. The four output lines are one-of-sixteen decoded to pulse one of sixteen rows of four keys high. The four columns sense a pressed key as individual inputs.

The software involved pulses the top key row high and checks for inputs. Then it pulses the next key row down to a one and again checks for inputs. The process continues down through all the keys, checking a four-key row each time. If no key is found pressed, the scanning repeats or else the micro goes on to do something else.

Note that we haven't labeled the keys. This could be an ASCII keyboard, an electronic music keyboard, or just a bunch of keys doing different things for you in your system. In fact, the keys do not all have to be in the same physical place or have the same intended uses. As examples, a QWERTY layout and a numeric keypad can be scanned at the same time; or else a trainer's hex entry $0–$F keys can be scanned along with the monitor command keys. The micro doesn't care.

How else could we input lots of key closures to a micro?

We could save another line by using a data selector to route one column at a time to a single input. Or, we could let the entire outside job be handled by a custom *keyboard encoder* chip. This outside chip would also handle hassles like debounce, CTRL and SHIFT keys, and changing a QWERTY pc layout into ASCII output codes.

If we really had to minimize port lines, the absolute minimum route would be to encode each key individually into a 6-bit code and then route only this code into six lines of a port. Use only sixty-three keys and save some state, such as all zeros, as a "no key down" mode.

It all depends on what you want to do. In general, fancy add-on hardware such as keyboard encoders is not the way to go, since these parts are costly and tie you down to doing things exactly their way.

Fancy external hardware does have one advantage, though. These chips can unload the microprocessor so it can spend its CPU time on more creative things than waiting around for someone to press a key. The latest and newest of the larger micro systems do this without extra hardware by using *two* micros, a big one with lots of smarts for the important stuff and a smaller, sometimes dumber one that is committed to busywork like watching keys, making noise, controlling diskettes, and doing other I/O tasks.

Which way to go?

On very small systems, try to have the micro do as much as possible with a minimum of limiting add-ons. On bigger stuff, think seriously about using a second micro that you can dedicate to the dogwork. If some simple, cheap outside hardware helps without limiting you, use it.

To keep things simple and general, I didn't show any input resistor terminations on our scanning key circuits or give you any part numbers. Usually, with CMOS ports and decoders, you will add a 22K resistor to ground on each input to guarantee a zero when no key is pressed. With LSTTL ports and decoders, you will usually hold each input column high with a 2.2K resistor to +5 vdc and pulse each key row *low* instead of high. A low output will now be a pressed key. While this seems backwards, it fits in better with the performance of LSTTL circuits.

Oh yes, one very important gotcha that applies to *all* uses of X-Y matrix circuits . . .

> **Elements in an X-Y matrix MUST prevent *sneak path* currents from occurring.**
>
> **A series diode at each element is one way to stop sneak paths.**

A *sneak path* is some roundabout way that current has of going where you don't expect it in a matrix. For instance, if you press three keys that are arranged as corners of a rectangle in a matrix, you will get a fourth *phantom key* pressing at the other corner. Check it out.

If you ever expect more than two keys to be down, you should add a diode at every key location to eliminate these sneak paths. There are also complicated software routines that can search for possible input sneak paths and handle them as special cases, but this can get even messier than using diodes.

Providing for more than two key hits at the same time is called *N-key rollover* and can be done with either hardware or software. N-key rollover is absolutely essential in polyphonic music keyboards and is very desirable when you have super-fast professional typists. But N-key rollover is not normally needed for most routine micro entry and typing uses. Much simpler logic involving two-keys-down, called *2-key rollover*, may be substituted instead.

Still another way of minimizing port lines involves *time multiplexing*. In time multiplexing, you let outputs *share* the port in time sequence.

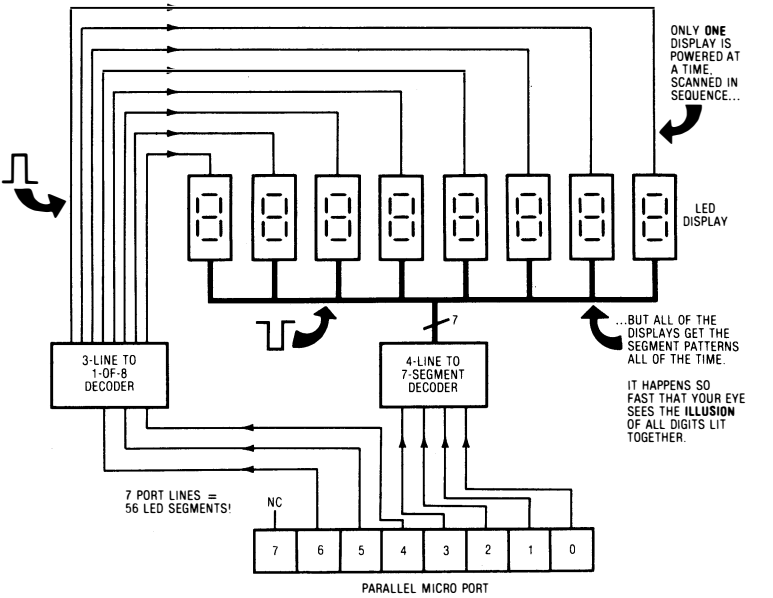
One place this works well is in light-emitting diode (LED) displays. If you want to light eight numeric displays, you light only one display device at a time, but you do so to eight times normal brightness. Then you light each of the digits in a time sequence that is so fast that the eye fills in and gives you the *illusion* of all the digits being lit all the time.

If you simply connected eight LED seven-segment displays to port lines, it would take fifty-six output port lines to do so. Go to time multiplexing and you are down to 8 + 7 = 15 display lines.

But go to time multiplexing and add some outside hardware, and you can drop to only seven port lines.

Like so . . .

TIME MULTIPLEXED LED DISPLAY



Here is what happens. The bottom four lines are routed to an external 4-line-to-7-segment decoder/driver. The four input lines are changed to a pattern that grounds certain segments on *all* of the common-cathode displays shown. Note that all eight displays are driven together in the same pattern.

Thus, the bottom bar of every display is connected to a line at the decoder circuit that controls all the bottom bars simultaneously. There are seven wires out of the decoder integrated circuit. Each wire stops once at each display, connected to the same relative segment.

The next three port lines are routed to a one-of-eight decoder that powers the positive end of only one display at a time.

Only one digit lights at a time, because only a single digit has *both* the correct segment pattern and a positive drive voltage.

The eighth port output line can be used for any decimal points or flashing colons on a clock display. Blanking can be added by using illegal decoder states or by using only seven digits and stopping on digit eight. In fact, it is a good idea to blank characters briefly during switching times. This can eliminate "ghosting" and other unpleasantries.

Here's a problem for you . . .

DOING IT:

Show how you can use the same port lines to scan a keypad and an LED display simultaneously.

This simultaneous scanning is the usual way that such "lightweight" micro applications as trainers, phototimers, microwave oven controls, and so on handle key input and display output with a minimum number of port lines.

Once again, though, don't forget those sneak paths.

On incandescent output displays, sneak paths would light supposedly "off"bulbs to low or medium brightness. Fortunately, on LED displays each light source is also a diode, so we automatically eliminate any sneak paths. As an added bonus, many LEDs are actually more efficient when pulsed to high currents. Thus LEDs are ideal for time-multiplexed display use.

Liquid crystal display (LED) elements become a real hassle if you try to multiplex them in an X-Y matrix. Liquid crystal arrays need AC drive circuits and demand extremely complicated circuit techniques to eliminate sneak paths. Special, ready-to-go liquid crystal driver integrated circuits are available and can be used. The complexity of these circuits is appalling.

Always ask yourself whether there will be sneak paths if you connect anything electronic into rows and columns.

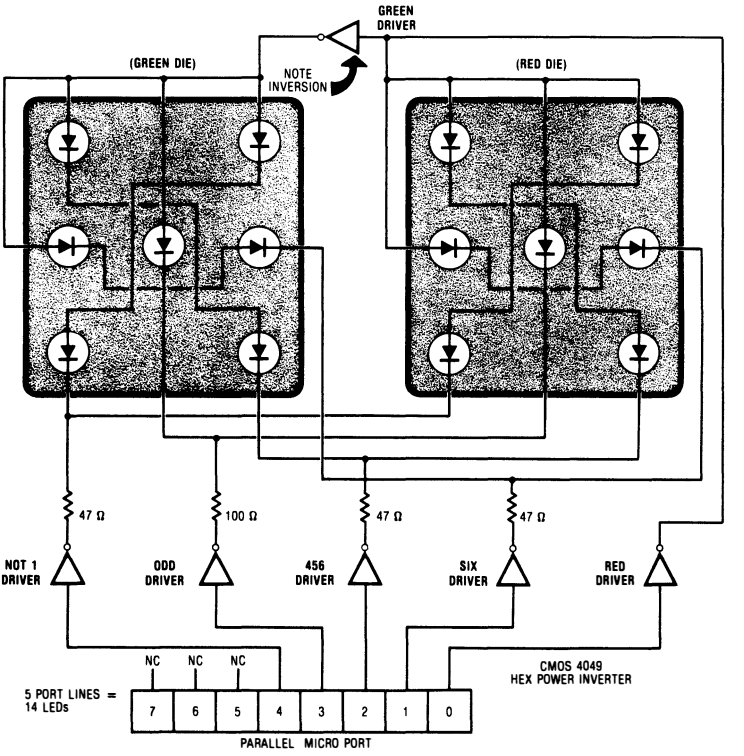
Time multiplexing is really time-sharing on a short-term basis. You whap each element in the display so fast that your eye thinks everything is there at once. But you could also save port lines by time-sharing on a long-term basis.

For instance, you could simply unplug one peripheral and replace it with another, as in switching from a plotter to a printer on the same output line. Or, you could have lots of different output or input devices hung on a few common port lines. Somehow you would have to activate one and only one of these at a time. Take turns. Use "Hey you!" and "Who me?" handshaking.

Another way to save on port lines involves using oddball, or non-obvious codings. Say you are building a simple dice game that is micro-controlled. There are seven spots on each die, so it could take up to fourteen output lines if you weren't careful.

Here is how to do it in five . . .

DICE DISPLAY USING NON-OBVIOUS CODE



What you do is pick a non-obvious decoding. Note that the center spot is lit only on *odd* numbers. So, use one line as an

"ODD" output. Looking closer, you see that a diagonal pair of dots will light on a "NOT 1" condition. Note further that the other diagonal pair will only light on a "4,5, or 6" condition. We'll call this one the "456" output line. Finally, the horizontal pair will light only on a "SIX" condition.

So, you need only four lines encoded ODD, NOT 1, 456, and SIX to provide die logic for one die.

Let's use time multiplexing and light both dice to double brightness for half the time. This takes a fifth line for a "RED" output. When RED goes high, you light the red die. When RED goes low, the "GREEN" inverting driver goes high and lights the green die. You alternate dice much faster than the eye can follow, giving the illusion of both dice lit all the time.

We see that a special coding needs only five lines to handle a two-dice display.

But don't go overboard on oddball codings. If the code gets so strange that it causes trouble later or becomes hard to understand and service, then don't even think about using it.

Our final good way to minimize port lines is to go serial. Output a serial code that is long enough, and you can control anything with a single output or input line to your micro. More on this in the next section.

We have seen that there are lots of good ways to cut down the number of port lines from an insanely large number to something reasonable. But don't go too far. Don't minimize port lines at the expense of getting into bizarre codings or super specialized add-on hardware. Try to find the magic number of lines that will minimize the overall complexity of your entire micro system.

Then, once you have reduced your port lines to the most reasonable minimum . . .

ALWAYS take any port lines that happen to be left over and try to make them do something new and useful.

It should somehow bother you if there is an unused port line or two "left over" on any micro application. Always see whether there isn't some new feature or some new operating mode that you can add to use these lines to make the micro do more than you first expected of it.

SERIAL I/O PORTS

With a serial code, you use a single line to output many bits in time sequence. As we have seen, serial codes are often used between microcomputer systems where we can use a single wire and low bandwidth to send anything we want to in time sequence.

There is really no such thing as a simple serial port, because . . .

> **Any old bit line on a parallel port can be used as a serial input or output just by sending the bits through in time sequence.**

You can also use some soft switches as simple serial ports if their output goes outside the machine. So, simple serial ports are no big deal. Just take any old port or less-than-a-port I/O line and shove some bits into or out of it.

Instead, we almost always use fancy ports for serial I/O. The most important thing a fancy port does for us is *unload* all the long sending time off the CPU. The microprocessor gives a brief send command to the fancy I/O port. The port then takes its good old time outputting the long serial code. The microprocessor itself is then free to do other, more sensible things while the code is being sent.

Fancy serial I/O ports will also pick up things like handshaking and interrupt management and may include interval timers or baud rate generators. Random and fairly slow inputs such as keypressings are sometimes best handled as interrupts, especially if the CPU has lots of other things on its mind.

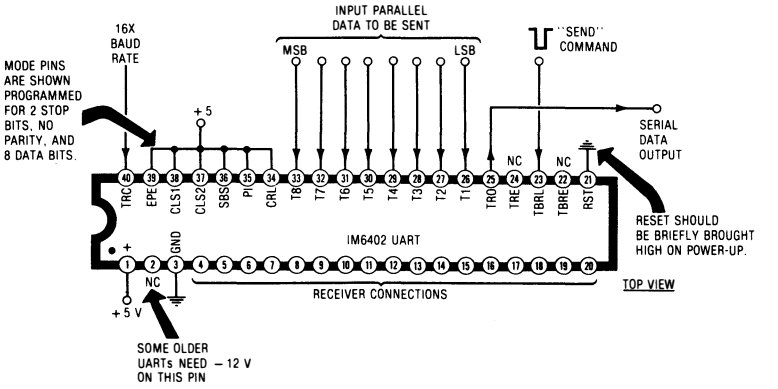
Each microprocessor family has available several fancy serial I/O peripheral chips. As we have seen before, you can get into real hassles if you try to cross school boundaries and mate a serial I/O device from one family with a CPU from another.

So, you will have to dig into the serial I/O peripheral chips of your choice. One good way is to find a trainer or microcomputer that uses the target chip and play with it to see how it works in a real circuit.

Fortunately, all of these fancy serial I/O chips are based one way or another on a plain old hardware UART. Let's look at a classic hardware UART circuit to see how it works.

Here's how you can transmit serial bits with a hardware UART . . .

NON-MICRO UART TRANSMITTER



First you supply this 40-pin chip by putting +5 on pin 1 and grounding pin 3. This *Intersil* 6402 is a modern, single supply chip. Older devices may also want a −12-volt supply on pin 2.

The transmitter is located on the "top half" of the chip, pins 21 through 40. You input a 16X clock to pin 40 to form a baud rate reference.

By the way, here are the popular baud rates and their 16X clock frequencies . . .

BAUD RATE	16X CLOCK
110 baud	1760 hertz
150 "	2400 "
300 "	4800 "
600 "	9600 "
1200 baud	19.2 KHz
2400 "	38.4 "
4800 "	76.8 "
9600 "	153.6 "

As a reminder, 110 baud is the old teletype standard and translates to 10 characters per second (CPS) or 100 words per minute (WPM). 300 baud is popular for modems, and 9600 baud is usually the fastest you can go over the phone line with special equipment.

Speaking of words per minute, watch out for those PR yahoos with their grossly misleading ad copy. The number of newly redis-

covered "words per minute" is *ten times* the industry standard number of "characters per second." Thus, a "new" daisywheel rated at 120 is much *slower* than an "old" daisywheel rated at 40.

Anyhow, we now have powered our UART and sent it a 16X baud rate. A few newer chips have the baud rate generator built in, but most of the serial I/O perhipheral chips do not. You have to get this baud rate from somewhere, and it absolutely *must* agree with the UART on the receiving end.

You then hardware program a traditional UART with jumpers on control pins 35–37. Options here include the number of bits per character (5–8), the presence of a parity bit (yes-no), the type of parity (even or odd), and the number of stop bits (one or two). You also make pin 35 high to accept the input data.

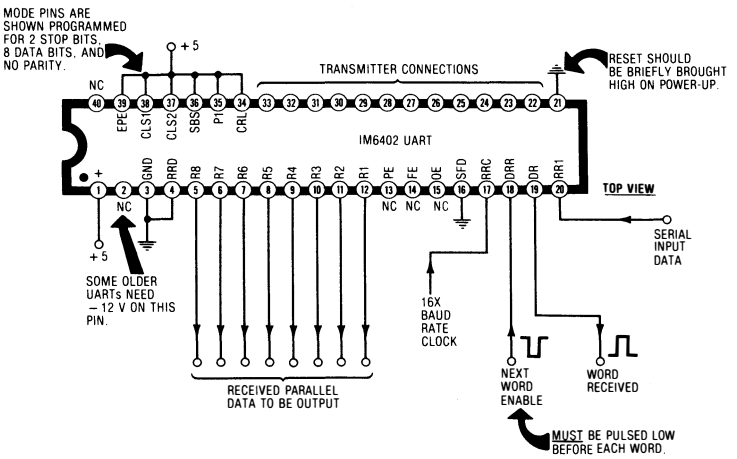
Parallel input data goes to pins 26 through 32, and your serial output appears on pin 25. To transmit with this chip, input your parallel data, then bring the STROBE line low, and back high again. This transmits your serial code, one bit at a time, in the standard teletype format we talked about back in Volume 1.

Note that the serial output is at LSTTL levels. You have to add outside hardware drivers for RS-232-C or other interface standards, if you use or need them.

There is a reset pin that should be brought high on power-up and then held low. There is also an optional pair of handshaking pins that tell you when it is okay to load and when the last serial data string has been sent.

To receive, you use the bottom half of the UART . . .

NON-MICRO UART RECEIVER



The same modes that were hard-wire selected for transmission are also used to receive serial data. A typical baseline setting might be eight bits, no parity, and two stop bits. A 16X receiver baud rate clock is routed into pin 17. For most uses, the receiver baud clock and the transmitter baud clock are supposed to be the same, so we jumper pin 17 to pin 40. Our asynchronous serial data goes in pin 20. The data must come from a CMOS or LSTTL compatible source, such as RS-232-C receiver hardware. When a start bit is detected, the internal UART circuitry starts assembling code one serial bit at a time and putting the result on parallel output pins 6 through 12. A "character received" strobe at pin 19 goes high when the character is assembled and ready for use.

There are several handshaking pins that tell you if you get parity, overrun, or framing errors. These outputs, along with the strobe output, are reset by bringing pin 18 low. One crude way to do this is to invert and delay the DR, or "Data Ready" output and shove it back into the $\overline{\text{DRR}}$ or "Data Ready Reset" input. Full handshaking may need some fancier way to make sure each received byte gets used once and only once.

One good use for UART transmitters is in remote or "lap" keyboards. By converting the keycode into a serial data stream, you end up with far fewer connections between keyboard and system. If you are careful, you can even use ultrasonics or infrared to create a totally wireless remote keyboard, or possibly even one that is solar powered.

There is another UART variant called the 6403 that includes its own built-in baud rate generator. For instance, you can hang a color television crystal (3.58 MHz) on pins 17 and 40 and ground pin 2 to output 110 baud. A 2.46 MHz crystal and a positive pin 2 gets you 9600 baud. See the *Intersil* data sheets for additional use info.

More details on traditional hardware UART uses appears in *The TV Typewriter Cookbook* (Howard W. Sams 21313).

Hardware UARTs are lots of fun to play with, and you really should use one somewhere before you try to understand the fancier serial I/O chips. But hardware UARTs aren't directly microprocessor compatible, since they have to have their operating modes hard-wired, and since the input parallel data and the output parallel data appear on separate sets of lines.

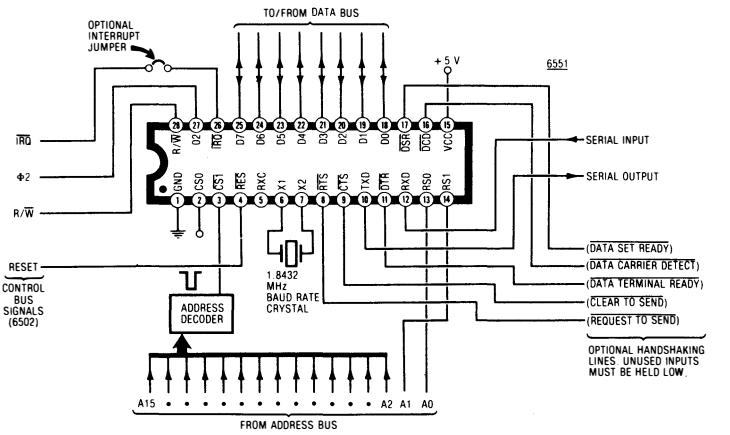
Each micro family has one or more fancy serial I/O chips available. These are usually harder to understand than the hardware UARTs, and you need to do plenty of digging into the data sheets and ap notes, along with lots of hands-on practice.

Important fancy serial I/O chips include the 8251 from the 8080 school and the 6551 from the 6502 school, among with many others.

I like the 6551, first, because it is simpler to use and easier to understand than many of the others and second, because you get an absolutely free baud rate generator built in.

Here is how you might use the 6551 . . .

MICRO UART USING 6551



You power this chip from a single +5-volt source, and hook up the usual 6502 clock, reset, and R/$\overline{W}$ signals as shown. A 1.8432-megahertz crystal gets hung on pins 6 and 7 if you are using the internal baud rate feature. If you are using interrupts with your serial I/O, an IRQ output is connected to the $\overline{IRQ}$ line in the micro system.

Serial data is output on pin 10, called TXD, and serial data is input on pin 12, otherwise known as RXD. An address decoder is routed to the chip-select input, pin 3.

To you as programmer, the 6551 looks like four sequential addresses. The chip-select should activate the chip on these.

Here is what these addresses do . . .

BASE+$00 — Writes data to be transmitted or reads data already received.

BASE+$01 — Writes a reset into the transmitter or reads the receiver condition.

BASE+$02 — Writes or reads a control register that sets the baud rate, word length, and stop bits.

BASE+$03 — Writes or reads a command register that handles handshaking and parity.

You use these four locations as needed. First, reset the transmitter and then initialize your control and command registers to the mode and baud rate you need. You can then write to the UART transmitter or read from the UART receiver as needed. Check the *Synertek* data sheets for more specifics.

There also are five handshaking pins that include two outputs called $\overline{\text{RTS}}$ (Ready to Send) and $\overline{\text{DTR}}$ (Data Terminal Ready), and three inputs called $\overline{\text{DSR}}$ (Data Set Ready), $\overline{\text{DCD}}$ (Data Carrier Detect), and $\overline{\text{CTS}}$ (Clear To Send). You use these as needed to make sure that information is input or output once and only once. Use details will vary with the fanciness of your handshaking.

All unused handshaking inputs must be held *low* for the 6551 to work.

Any fancy serial peripheral chip can be understood by finding out first how you power the chip and then how you connect it to the system control lines. Then you find the data bus connections, the address connections, and the address decoding scheme. Then you find the input and output serial lines and, finally, check into the handshaking.

One good way to test and debug *any* type of serial interface is to check it first at the lowest possible baud rate with the crudest possible "Here it is; take it or leave it" handshaking. This lets you separate the really fundamental problems, such as uncrossed transmit pins or mismatched baud rates, from more subtle handshaking hassles.

There's lots of confusion over RS-232 connector pins. If you are going to a printer or another micro, usually pin 2 and pin 3 *must* be crossed *once* somewhere along the way so that one micro's output is the other micro's or printer's input and vice versa. Modems, on the other hand, do *not* need this crossed signal path and will not work this way.

A good baseline setting on the RS-232-C lashups that do not involve modems would cross pins 2 and 3 *only once* and separately tie pins 6,8, and 20 together. Again, as a baseline trial, start with eight bits, no parity, and two stop bits at both ends. Naturally, the receiving and transmitting baud rates must agree exactly. Start with 110 baud if you can.

After you get your serial interface talking more or less correctly at a low baud rate, then you can speed things up and go on to solve subtleties involving buffer overflows and other complications that show up at faster speeds.

Once again, the output and input lines of your fancy serial chip will usually be low level LSTTL or CMOS compatible signals. External drivers and isolation may have to be added for such things as driving RS-232-C or other serial standards.

"MORE THAN A PORT" I/O

A lot of very fancy peripheral chips available for most micro families do much more than just inputting or outputting serial or parallel data. We can call these "more than a port" I/O.

Here are a few random samples . . .

"MORE THAN A PORT" CHIPS
Timers Real-Time Clocks CRT Controllers Disk Controllers Keyboard Encoders LAN Access ICs Arithmetic Units Universal Controllers

Most of these chips connect to the address, data, and system buses of your micro and then provide some "higher level" interface function for you. Often, these chips are very micro system specific. As usual, you understand them by starting with data sheets and application notes and then going to hands-on practice in simpler use modes.

A *timer* is a "more than a port" integrated circuit that has one or more timing devices inside. These can be used as counters or clocks and are useful to unload the CPU from short-to-medium timing jobs. For instance, a 10-millisecond time delay is an ideal use for one of these. Sometimes timers are built into chips that do other things. The *Synertek* 6530 from the 6502 family combines a pair of parallel ports, a shift register for serial data, a timer, some RAM, and some ROM all on one chip.

A *real-time clock* is a fancier timer that keeps track of "people time," including seconds, hours, days, weeks, months, years, and so on. Often, these circuits are kept alive by a small backup battery when micro power is removed. Real-time clocks are useful to show the time of day and to log execution times into programs. *OKI Semiconductor* is one good source of real-time clocks, their MSM 5832 being more or less typical.

CRT controllers handle some to most of the support circuitry needed to drive a video display. Certain types are intended more for text-only terminal uses, and others are for uses involving fancy color graphics. *Standard Microsystems* has an extensive line of these, and

so do *Intel* and *TI*. At present, *NEC* seems to be leading the field in sophisticated color graphics controllers.

The trend is away from specialized devices that can display only text, since practically all new micro applications *demand* fully mixed graphics and text capabilities. Even text-only displays must have a wide choice of fonts immediately on line in most new applications.

Disk controllers are used to interface disk systems to a micro. Two popular types are *floppy disk* controllers, obviously for floppy disks, and *Winchester* controllers for *hard disks*. *Western Digital* is one leading distributor of specialized disk controller chips.

Keyboard encoders are used to encode standard ASCII keyboards, with *General Instruments* and *Standard Microsystems* being important sources. As we have seen, the trend is away from custom keyboard encoding, since direct micro keyboard access is far cheaper and far more flexible.

There's much interest, these days, in connecting microcomputers into networks, and quite a fight seems to be brewing over networking standards that include *Ethernet* and many others. At this writing, lots of new integrated circuits called *LAN Controllers* are being introduced. LAN stands for *Local Area Network* and means a bunch of communicating computers and peripherals situated reasonably near each other. *Seeq Technology* and *Standard Microsystems* offer LAN controller chips commercially.

If you just want to tie a couple of personal computers, a printer, and a hard disk together, there are easier ways to do it than going to network controllers. You can use current loops, RS-232-C serial at 9600 baud, or even interface directly through game-paddle connectors. For lots of real-world uses, the LAN standards are gross overkill.

Other controllers similar to the LANs are available for GPIB instrument interface uses.

By *arithmetic units,* I mean any peripheral chip that does math faster and easier than you could do directly. This includes such things as fast multipliers, floating point circuits, and calculator chip interfaces. These are most useful when you have to do fancy trig calculations or lots of multiplication in a hurry. *TRW* is a very expensive leader in this field.

Finally, a *universal peripheral controller* is really a "slave" microprocessor that has many on-board ports and is easily programmed to handle fancy I/O tasks such as keyboards, displays, and just about anything else you can dream up. The peripheral controller handles all the dogwork, so that the main micro is free to go on with high level tasks. *Intel* is very heavy into programmable peripheral controllers.

There are lots more of these "more than a port" peripheral chips, and many more are being announced. Some of the most exciting new microcomputer developments center on these. But, unless you can find a reasonably priced "more than a port" I/O chip that does exactly what you want—and that really and truly is available, second sourced, and cheap—you will often do better just throwing in a second stock microprocessor as a slave to the master micro in your system.

Some interesting video uses of slave microprocessors are shown in *The Cheap Video Cookbook* and *Son of Cheap Video* (Howard W. Sams 21524 and 21723).

The trend in single-chip microcomputers today is to put everything into a single package. That includes RAM, ROM, CPU, parallel ports, serial ports, and system timing. Some of the newest beasties even include A/D and D/A conversion and on-board phase lock loop circuits. These are a good choice for simpler micro applications but may be hard to expand.

I'm still looking for a single-chip micro that is genuinely useful for low volume, low power applications. What I would like to see is a one-chip something that speaks "6502" or something equally powerful and easy to use; has twenty bidirectional and parallel port lines ready to go; includes a pair of serial ports with built-in baud rate generation; has at least 256 bytes of RAM and 4K of nonvolatile, built-in EEPROM; and sells for, say, $5 in singles. Naturally, the thing must be micropower and must work over a wide and sloppy power-supply range, with full power-down and "data hold" features. It must also cross-assemble easily on the Apple II as well as download and require at most $30 worth of extra software for a complete development system.

Three immediate uses I have for this are wilderness data acquisition, weather logging, and "intelligent tapping" for cable TV distribution systems. I could easily come up with hundreds more uses.

Any takers?

OPEN COLLECTOR OUTPUTS

Just a quick note here. Three types of output circuit lines are popular for use on peripheral chips. *Direct* outputs are usually CMOS or LSTTL compatible and are there all the time. These vary in drive ability and allowable output loading, so it always pays to check the data sheets.

Tri-state outputs are there only when they are properly enabled; they float otherwise. This lets something else grab the output line without any interference.

Finally, there are *open collector* outputs. Open collector outputs are useful to "wire OR" things like a daisy-chained interrupt line and to minimize internal supply power in real-time clocks and some other low-power CMOS peripherals.

By now, you should know better than to connect two direct CMOS or LSTTL outputs together, because they will fight each other. And, you should know that any number of tri-state outputs can be connected, as long as you enable one, and only one, at a time. Naturally, to view or use a tri-state output, only that output and no others should be enabled.

But sometimes it is easy to forget the pullup resistor on an open collector output. If you don't provide an *external* resistor on an open collector output, you will get output lows or else nothing. On a scope, you will see either nothing at all or just a few millivolts of "fumes" where the output should be.

Typical LSTTL pullup resistors are 2.2K or 4.7K, whereas CMOS pullup resistors normally range from 10K to 100K.

Hence, this obvious rule . . .

On open collector output lines, ALWAYS make sure you provide a pullup resistor somewhere EXTERNAL to the chip.

Otherwise, there is no way to get or see an "output high" state.

Note that the pullup resistor is needed even if there is nothing else sharing the open collector output. This detail is easy to miss, particularly on CMOS clock chips, so watch for it.

That just about gets us through micro-level interface. To interface micros at the chip level you can use "less than a port" stunts such as soft switches, "real ports" such as parallel and serial interfaces, and "more than a port" specialized chips that handle fancy tasks at increased cost and complexity.

Time now to step to the next interface level . . .

CIRCUIT LEVEL INTERFACE

In practically all real-world applications, the signals you try to input to the microcomputer are too big, too small, too noisy, too sloppy, or too dangerous. At the other end, the commands you try to get from the microcomputer are usually too weak, too noise-sensitive, too in need of safety isolation, or not in the right form for

final use. So, you almost always have to add some outside *interface electronics* between the microcomputer and everything else.

This outside level two circuit interface electronics is usually made from noncomputer stuff like plain old transistors, optocouplers, special IC driver chips, relays, triacs, A/D and D/A converters, and similar components.

Add-on interface circuits of some sort will be needed just about everywhere that you try to connect a microcomputer to the rest of reality. About the only time you will *not* need much in the way of circuit level interface is when the signals come from or go to completely micro-compatible, low-level, and *local* sources and sinks.

Thus . . .

External CIRCUIT LEVEL INTERFACE parts are almost always needed between the microcomputer and the real world for most I/O uses.

We have both *input* and *output interface* . . .

INPUT INTERFACE—Electronic components added for compatibility between signal sources and the micro inputs.

OUTPUT INTERFACE—Electronic components added for compatibility between micro outputs and outside loads.

Let's look at output interface first.

OUTPUT CIRCUIT INTERFACE

Output circuit interface goes between the microcomputer and any loads you want the microcomputer to control. These interface electronics will be needed for just about everything *except* driving local and low-level LSTTL or CMOS peripheral output circuits.

Our output interface circuits can usually handle three different tasks . . .

OUTPUT INTERFACE can:

( ) amplify
( ) isolate
( ) convert

By *amplifying,* I mean making the output signals "louder" so they are strong enough to directly power or control a heavy output load, such as a motor, a solenoid, a large lamp, or anything else that needs a stronger signal than you can get directly from a micro's output port lines.

By *isolation* I mean finding a way to let a microcomputer control a load without being physically or electrically connected to that load. Relays and optocouplers are often involved here. Isolation can provide *safety* for loads that are connected to the AC power line. This can eliminate a potentially deadly shock hazard from the micro and its users. Isolation also provides *noise immunity* that can keep any spikes or other high-level garbage at the load end from getting back into the micro's sensitive circuits and raising havoc.

Finally, by *conversion* I mean converting the digital ones and zeros from our output into something else. That something else is often a slowly changing or *analog* signal that can have many possible values, instead of the on-off snap-actions of the digital world.

You can also think of a motor as a voltage-to-rotary motion converter and a solenoid as a voltage-to-linear motion converter. So, your conversions will sometimes be done by special circuit level converter electronics and other times by the devices you hang on the final output. Let's check into these one at a time . . .

circuit level amplifiers

A typical output port will have LSTTL or CMOS compatible lines that can source or sink a few milliamperes of current. Should you need anything more than this as an output, some sort of "amplifier" will be needed to make the signal strong enough to handle the load you are trying to drive.

The most common and most used amplifier is called an *NPN transistor.* This is available as a separate component, or it can be built into husky integrated circuits specially intended for circuit level interface uses.

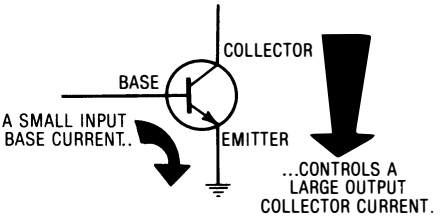
An NPN transistor has three leads, called the *emitter,* the *base,* and the *collector.* Almost always, you connect the emitter to a common path that is usually also zero volts or ground. You normally connect the collector through a load to a positive power supply.

The current needed by the load must be within the range that the NPN transistor can handle. The positive supply voltage must be below the breakdown value of the transistor, but it also has to be of the proper voltage that the load will demand when the load is powered or "on."

Usually you input a small conventional current into the base of the NPN transistor. Conventional current travels in the direction of the arrows on most solid state device symbols. A small current into the base controls a large current into the collector. The transistor is a *current amplifier* that lets a small base current control a large collector current.

Like this . . .

HOW AN NPN TRANSISTOR AMPLIFIES

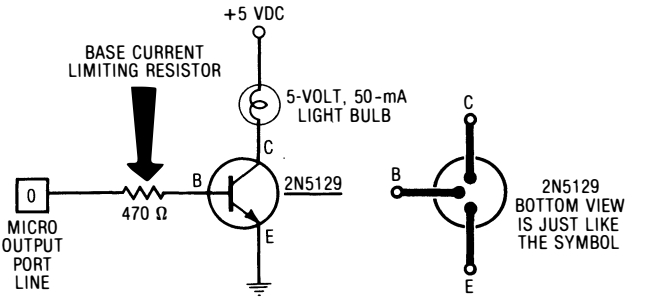


In some non-digital transistor uses, the output current will follow the input current *linearly*. Double the base current and the collector current will also double. But, in the digital world, we usually *saturate* our transistor when we turn it on. To saturate means to turn a transistor as far "on" as you possibly can. This means that you input a weak one for base current and a weak zero for no base current. You get out a "full on" for the weak one input and a "full off" for the weak zero input.

Input a weak zero, and the NPN transistor stays *off*. This outputs a "loud" zero. Input a weak one, and the NPN transistor turns *on*, powering the load and outputting a "loud" one. In the off state, the full supply voltage appears across the transistor and there is zero load current. In the on state, almost the full supply voltage appears across the load, and the load draws its normal current.

Here is how you light a small incandescent light bulb using an NPN transistor . . .

NPN TRANSISTOR DRIVING A SMALL LAMP



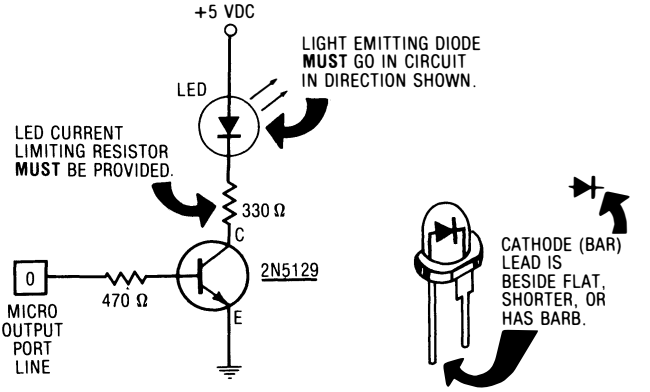
In this case, the lamp is the load for our NPN transistor. If the micro port provides a weak high output, this output in turn gives us transistor base current, turns the transistor on and saturates it, and puts nearly the full supply voltage across the lamp. This, of course, lights the bulb.

A saturated transistor normally has a few tenths of a volt drop across it, so almost all of the supply voltage will appear across the lamp terminals.

If the micro port provides a weak and low output, this output gives us zero base current, which turns the transistor off. There will be no current through the lamp, and nearly all the supply voltage will appear across the "off" transistor. The lamp will be out, since no current is going through it.

You might like to light an LED or *light emitting diode* instead of an incandescent lamp. Here is how to do it . . .

NPN TRANSISTOR DRIVING AN LED



As before, a port high provides base current which saturates the transistor and lights the LED. A port low stops base current which turns off the transistor and puts out the LED.

But note two gotchas. First, most LEDs are *polarity sensitive*. This means they will only light if you put them in the circuit pointing in the right direction. The correct way is with the *cathode* (or "bar") of the LED going to the *least positive* or grounded side. Often the cathode lead of an LED will be shorter, have a small metal flag on it, or will be beside a flat on the plastic. Check the data sheet or try both ways if you have any doubts.

Second, note the resistor that is in series with the LED lamp. A light-emitting diode is a *current* driven device. Connect it to a voltage supply, such as 5 volts, and the LED will self-destruct because it will try to light itself so bright that it melts. Current through an LED *must* be limited to a safe value. This safe value is typically 10 milliamperes or so, and can be provided by a 330-ohm resistor and a 5-volt supply. When lit, a light-emitting diode will have slightly under 2 volts of drop across it, with the current being set by the outside resistor.

Let's repeat that . . .

To light an LED safely:

You MUST put the LED into the circuit in the right direction.

You MUST provide an external current-limiting resistor to set the brightness to a safe value.

By the way, NPN transistors themselves are also current-operated devices, and their input base current also must be limited to keep the transistor from melting. That is what the base resistor does in the last circuits. The base resistor also prevents "current hogging" when you connect more than one transistor amplifier to the same port or try to use the port's logic ones and zeros elsewhere.

Besides NPN transistors, there are *complementary* or "upside down" devices called PNP transistors. These are intended for use on a negative supply. You can also get *field effect* transistors that use an input voltage to control an output current. These are available in complementary pairs, with the *N-channel* devices being used with positive power supplies and *P-channel* devices with negative power supplies.

Power field effect transistors (FETs) are becoming popular for circuit level interface. These are easier to drive and more rugged than NPN transistors, but they still cost more. Power FETs easily handle the high voltages once associated with vacuum tubes. They are

available rated to tens of amperes and hundreds of volts. Because power FETs are new and still expensive, NPN transistor amplifiers still dominate most low-level uses.

The *gain* of an NPN transistor decides how much output collector current you get out for a given input base current. The typical data sheet gain for an NPN transistor will be in the 50 to 500 range.

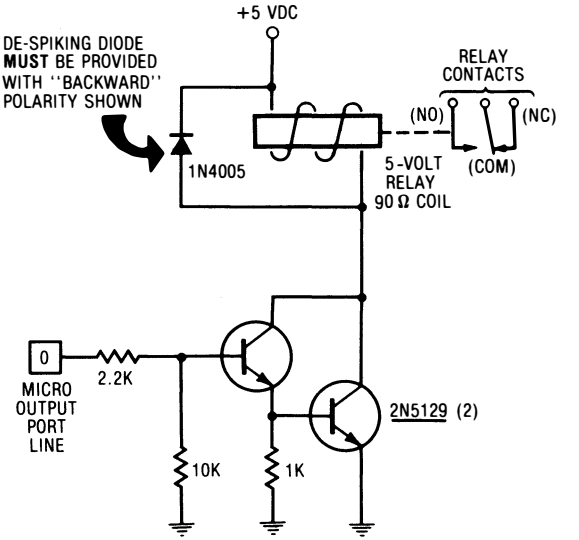
But, when you use an NPN transistor as a digital amplifier or as a saturated switch, you must be sure the transistor stays completely on when you drive it. You make sure by providing *overdrive,* or much more input current than you know you will need. When you allow for overdrive, you can get a gain of 10 to 100 out of your usual single NPN transistor. Thus, with 2 milliamperes out of a port and a saturated gain of 50, you can output 100 milliamperes through a small incandescent bulb load.

What if you want more gain? And how do you control much heavier loads?

One older way to pick up more gain is to add another amplifier to your amplifier. Cascade two transistors this way, and you come up with a circuit called a *Darlington* that can have saturated gains up in the thousands.

Here is a Darlington pair of transistors controlling a power relay . . .

DARLINGTON TRANSISTORS DRIVING A RELAY



A little current into the base of the first transistor puts a fairly hefty current into the base of the second transistor, which gets you a very large combined collector current. As before, a low port line gives you no collector current. A relay with no current does nothing, leaving any contacts in their unpowered condition. A high port line turns on both transistors and applies nearly the full supply voltage across the relay coil. The relay current then magnetically "pulls in" any and all of the relay's mechanical contacts, putting the contacts into their powered condition.

The simplest relay type is called a "make only" or *single pole, normally open* relay. Current flowing into the coil closes the contact. No current leaves the contact open. You can have lots more contacts on your relay doing other things if you want to. I've shown an SPDT or *single, pole, double throw* relay that has a *common* contact (COM), a *normally closed* (NC) contact, and a *normally open* (NO) contact. "Normally" here means *unpowered.* Since the contacts are completely isolated from the coil, you can use them to control power loads however you like, within the rating of the contacts.

One type of very small relay is called the *reed relay.* This one is easily interfaced to a microcomputer port and needs little in the way of coil current. But reed relays are very limited in their ability to handle load current. Most reed relay contacts are restricted to 110-volt AC loads of 10 watts or less and will rapidly burn out or weld on any DC load that is highly inductive or capacitive. Reed relays may look nice but they just aren't all that useful.

Since relays are mechanical, in time they will fail. Relays are also current hogs and are often bulky and expensive. For these reasons, all-electronic switching or amplifying is usually preferred for newer micro applications.

One VERY important gotcha.

See that diode across the relay coil that seems to be in the circuit "backward"? That diode is called a *spike protector* or a *freewheeling diode,* and it MUST be added to any coil that you are trying to control with a transistor.

What happens is this. If you try to turn a coil off suddenly, the energy in the magnetic field collapses very fast, which creates a high negative voltage. That's how the coil in a car generates tens of thousands of ignition volts from a 12-volt battery.

Most transistors do not like tens of thousands of volts applied to them backward. They tend to get very uppity if you try this. In fact they simply die. But when you provide spike protection, the diode conducts immediately when the transistor shuts off, which keeps a current briefly running through the relay. The current runs long

enough to "empty" the magnetic field slowly and safely, thus preventing a circuit-destroying spike.

So . . .

When you are controlling ANY coil with a transistor, ALWAYS add a protecting diode or other spike protection.

Note that the diode points "backward." The diode should conduct only on the reverse relay current and not on the supply current. Put the diode in frontward instead and you will destroy the transistor rather than protect it.

A protecting diode tends to lengthen the relay's "hold" or *drop-out* time. This can be critical in some uses. Other spike protection, such as a zener diode or a transient absorber, can sometimes be used both to provide protection and to minimize dropout times. For most uses, the few milliseconds of extra hold time is no big deal. Ignore it till it catches up with you.

Darlington transistors are less efficient and much slower switches than power field effect transistors. Today the rule seems to be to use Darlingtons for low to medium loads and to use relays, triacs, or power FETs for heavier loads.

If you want to use lots of Darlington drivers at once, it will take bunches of transistors and resistors to do the job. As an alternative, fairly rugged *peripheral driver* integrated circuits are available that give you control of up to eight power loads in one package.

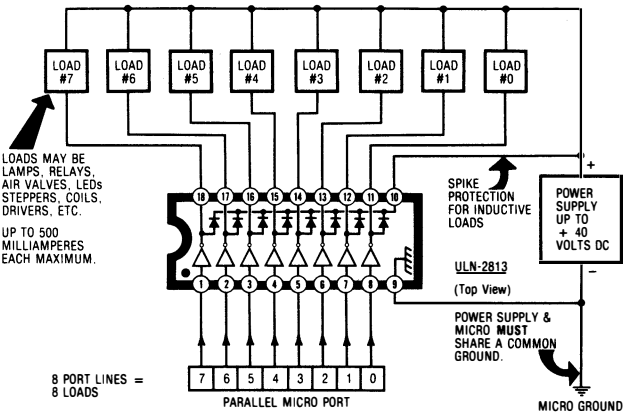
These peripheral drivers are usually used on power supplies of 40 volts or less where the load current is half an ampere or less. Thus, these peripheral chips are often a good solution for "medium power" things like driving print hammers, small stepper motors, LED display arrays, miniature solenoids, pneumatic air valves, larger indicator lamps, and other things in this power range.

The load power supply for these peripheral driver chips *must* share a common ground return with the microcomputer port. This can create noise or current loop problems if you aren't super careful.

Let's look at two examples of medium power peripheral chips . . .

Here's one of my favorites, the *Sprague* 2813 . . .

OCTAL PERIPHERAL DRIVER



This octal peripheral driver comes in an 18-pin package and handles eight separate loads of up to half an ampere each, powered from an outside supply of 40 volts or less. The inside circuitry consists of eight separate Darlington transistor drivers, along with eight separate spike-protecting diodes. Once again, this supply *must* share a common ground with the microcomputer and the peripheral chip.

There is a total package dissipation limit on the 2813, though, so if you are using all of the outputs at once, you have to limit your load current to 200 mils per output or less. See the data sheet for the derating curves.

Heat sinking is recommended for heavier loading. *Wakefield* is one source of "clip-on" fins suitable for plastic DIP packages. Wide printed-circuit runs also help to remove heat from packages.

Each driver is independent. The pins are arranged straight across the package, with all low numbered pins being inputs and all high numbered pins being outputs. Make any input *high,* and the output for that input goes *low,* as the internal Darlington transistor pair conducts heavily. Thus, a *high* input turns the driver *on* and *powers* the load.

The eight spike-protecting diodes are connected internally to a common pin. You connect this pin to the positive supply for the load. This should protect the internal circuitry against inductive surges.

This beast is a good general-purpose driver for most low to medium-level microcomputer output interface. I've used it to drive air valves for pneumatic robotics, among other things.

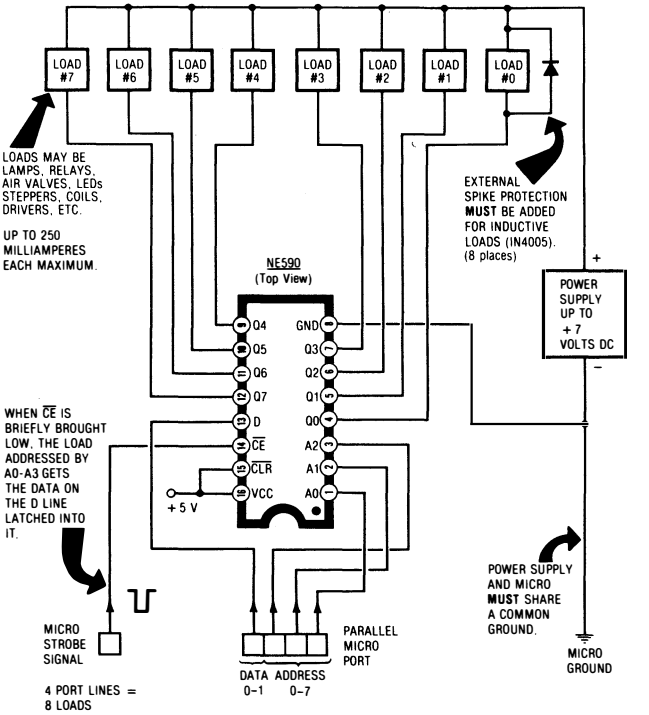
Which, by the way, is a field that's crying to be exploited. Low-pressure air does things so much better than solenoids and mechanical mechanisms that there is no contest. And yet, Detroit surplus three-way air valves go begging at giveaway prices.

Like a quarter each.

It may seem like gross overkill to use a half amp driver on a single LED, but this very tough chip is in fact a good LED driver for student lab projects and is priced under two dollars.

Here's a peripheral driver from way out in left field, the *Signetics* 490 . . .

ADDRESSABLE AND LATCHING OCTAL DRIVER



This one isn't quite as tough as the 2803. It allows only 200 milliampere loads at a maximum supply voltage of 7 volts.

The 490 does have several sneaky advantages. First, there are individual storage latches for each output, so the chip will remember the last desired state of each output so long as power is applied. Second, you can clear all of these internal latches to zeros, to get a safe "all off" condition whenever you power up.

Third, and most sneaky, this chip is *addressable*. You do not input eight commands simultaneously. Instead, you *address* select one of the eight output latches and then write a one or a zero to it. This saves on package pins and means you need only four or five port pins to control eight output lines. It does take extra software, though, to update each of the eight outputs one at a time.

One catch. This behaves backward, compared to the 2813. An input *zero* gives you a low output, which translates to an *on* load. Oh, well.

Ferinstance, to activate load number five, you input a binary %101 pattern to the address lines and input a zero to the the data line. When you pulse the chip-enable low on pin 14, the zero is latched into latch 5, and that load is powered by means of an NPN Darlington output driver.

Here's an obvious application . . .

DOING IT:

Show how to interface the 590 and eight reed relays to the game connector on an Apple II so you can gain eight medium-power outputs with minimum hassle.

Remember that the Apple II does not normally have a plain old 8-bit parallel power port available and that those printer cards, etc., do not have lots of drive ability. With the 590 you can take the four annunciator outputs off the game-paddle connector, use three of them to select an address, and use the remaining one to output data. The strobe output is then used to update the internal latches. Reed relays are easily driven by the 590.

Use peripheral drivers like these for medium-power loads. For heavier loads, you can step up to power FET devices.

If you are controlling AC power loads, though, the only way to go is with a beast called a *triac*. Triacs nearly always introduce a severe shock hazard into the circuit in which they are used. So, before we

find out what a triac is and what it is good for, we first had better learn about . . .

output isolation

Whenever you worry about a shock hazard getting into the microcomputer, or don't want any electrical noise getting back into the works, or simply want to separate the microcomputer system completely from the devices it is controlling, you have to use some sort of *isolation* . . .

ISOLATION—Any scheme to separate a micro physically and electrically from its input sources or output loads

A relay is an obvious output isolator, since there is nothing between the relay coil and the output-switching contacts except a magnetic field acting through a distance. Signals that go through the relay's contacts are electrically and physically separate from the coil circuit.

But the most common, most standard way of isolating things in micro circuits is with a small beastie called an *optocoupler* . . .

OPTOCOUPLER—A lamp and photocell device that provides isolation by communicating over a light beam.

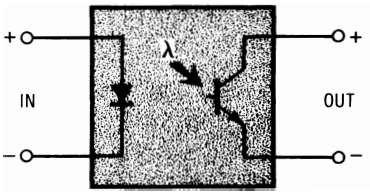
Optocouplers are sometimes called *opto-isolators* or *optically coupled isolators.*

On the input side of an optocoupler, you power or do not power a light source, such as an infrared LED. On the output side, there is some sort of device that responds to the presence or absence of infrared light. All that goes from input to output is a light beam or no beam.

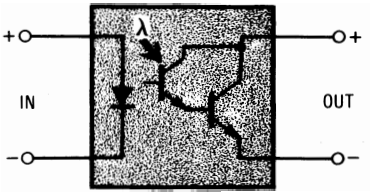
There are many different types of optocouplers. Most of them are packaged in 6- or 8-pin DIP packages.

Here are four popular styles of optocoupler . . .

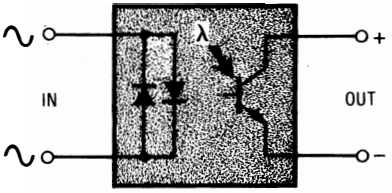
TYPES OF OPTOCOUPLERS



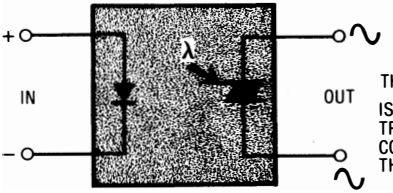
THE **PHOTOTRANSISTOR** OPTOCOUPLER
IS FAIRLY FAST (10 μs) BUT HAS LOW GAIN (.1 TO .5).



THE **PHOTODARLINGTON** OPTOCOUPLER
HAS HIGH GAIN (1 TO 5) BUT IS VERY SLOW (200 μs).



AN **AC INPUT** OPTOCOUPLER
CAN HANDLE INPUT SIGNALS OF EITHER POLARITY. THIS ONE IS SHOWN WITH A PHOTOTRANSISTOR OUTPUT STAGE.



THE **PHOTOTRIAC** OPTOCOUPLER
IS USED TO SWITCH A LARGER, EXTERNAL TRIAC ON AND OFF FOR DIRECT CONTROL OF HIGH POWER AC LOADS. THIS ONE SWITCHES RATHER THAN AMPLIFIES.

Most optocoupler use an LED as a light source. These are easily driven by most micro ports. Infrared light is used since its spectrum is more compatible with many light sensors and since you won't be watching it anyway. Almost all optocouplers are opaque packaged so that room lighting has no effect. As with any LED, you must be careful of the circuit polarity and must limit input current to a safe value, typically 10 milliamperes.

Our first optocoupler uses an infrared LED and a *phototransistor*. This combination is fairly fast but does not have much gain. The

second circuit uses a *photodarlington* instead. This gives you more gain but is much slower than a single phototransistor. Typical gain values are 0.1 for a phototransistor and 1.0 for a Photodarlington. Gain is measured by the ratio of the output current to the input current. Gain tends to decrease as an optocoupler ages.

Optocouplers are rather slow devices compared to most other electronic components. A phototransistor optocoupler might take 20 microseconds to respond, and a photodarlington optocoupler might take the better part of a millisecond. Faster devices are available at premium prices, and you can get more speed with a *cascode* stunt where you do not let the voltage across the output change during switching. But even with magic tricks or lots of cash, don't expect miracles.

So . . .

Optocouplers are rather slow and rather low in gain compared to most other electronic devices.

Always check the data sheet if you need speed or gain in handling a circuit isolation problem.

In our third circuit, there are a pair of input LEDs connected back-to-back inside the optocoupler. This lets an AC signal drive the optocoupler, and you will get an output for either polarity input signal. One important use is to enter a power line reference frequency into your microcomputer. More on this when we get to input interface. AC input optocouplers can have several different styles of output devices, some of which include extra snap-action circuitry. I have shown a phototransistor output stage for this one.

Sometimes, the light of the optocoupler source is allowed out of the package. One example is the *reflective object sensor* that is useful in detecting "ribbon out" conditions on printers and also useful in heart-pulse rate monitors. A second example is the *photointerruptor* which has a slot in it. We will learn how to use this latter device to sense speed or position shortly.

Our final optocoupler circuit combines an input LED with a phototriac. A *triac* is an AC switch that is useful to control high power AC loads directly . . .

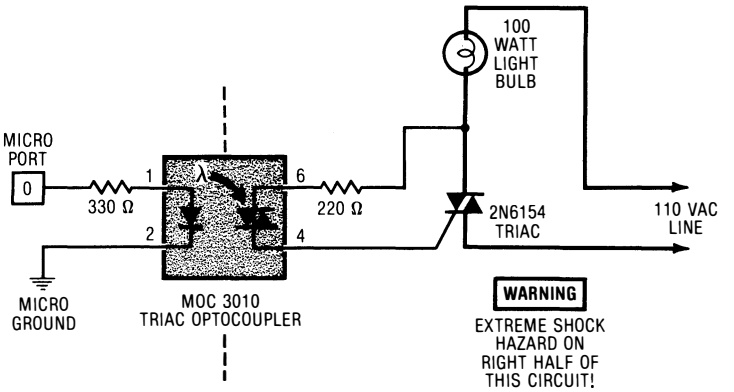
TRIAC—An electronic switch used to directly control high power AC loads.

Triacs are pretty much *the* standard way of controlling any AC power load such as a 100-watt light bulb, an induction motor, or a large heater.

The triac in an optocoupler, though, isn't nearly powerful enough to directly control a big AC load. Instead, you use an optocoupler triac to control a "real" triac.

Like this . . .

MICRO CONTROL OF HIGH POWER LOAD



This circuit shows you how to let a microcomputer directly control a 100-watt light bulb connected to the AC line. In fact, the same circuit can be used to control up to 1000 watts of AC power if you use a fairly large heatsink on the big triac. The microcomputer port output lights the LED, which trips the little triac, which in turn trips the big triac. The big triac turns on and stays on till the main current through it drops to zero, as happens each AC half cycle.

Sometimes your output port may be able to sink current better than source it. If this is the case, connect the input to the optocoupler *between* the port output and the +5-volt DC supply instead of to ground as shown here. Be sure, of course, that the current goes through the LED in the right direction. Sometimes an extra bias resistor can also help out.

Unlike a variable resistance in series with a power load, triacs are off-on switches and are thus far more efficient.

There's lots of different ways to use triac optocouplers. The circuit I have shown you is used for on-off control. If, instead, you carefully time your lighting and unlighting of the LED with respect to the AC power line cycle phase, you can control the brightness of

a lamp hung on your triac. For instance, if you turn the LED on late in each AC half cycle, the lamp will light only dimly. But if you turn on early in each AC half cycle, the lamp will light brightly. Obvious uses include dimmers, theater lighting, programmed disco effects, and burglar deterrents.

Other variations will let you switch only whole cycles of the AC line for minimum radio noise or to provide feedback for constant-torque control of DC motors. But note that you cannot control the speed of an AC induction motor simply by duty-cycling as you would a dimmer on a lamp. You can set up variable speed AC motor controls, but they take more sophisticated circuits that change *both* voltage and supply frequency.

At any rate, if you want to control any large AC load from a microcomputer, the optocoupler triac route is usually the best way to go. The triac efficiently switches the power control for you, while the optocoupler provides safety and noise isolation.

Note that you *must* use an LED-triac optocoupler for AC power control applications. The phototransistor or photodarlington optocouplers will instantly self-destruct if you connect their outputs to the AC power line. Note also that you must use a current-limiting resistor on the phototriac coupler output, as shown, in addition to the usual input LED current-limiting resistor.

OUTPUT CONVERSION

It is rare that we want a strictly electrical output from a real-world microcomputer use. Instead, we will want light, heat, motion, sound, or some form of energy other than pure electricity. So practically anything you hang on the output of a microcomputer will convert from digital ones and zeros to some other energy form.

One very important converter circuit is called the *digital to analog* converter . . .

D/A CONVERTER—Any circuit or device that will convert one-zero, or digital, signals into continuous, or analog, signals.

Important uses of D/A converters include telephone communications, speech generation, and electronic music.

Two key parameters of a D/A converter are the *resolution* and the *settling time* . . .

RESOLUTION—The number of possible output levels of a D/A converter.

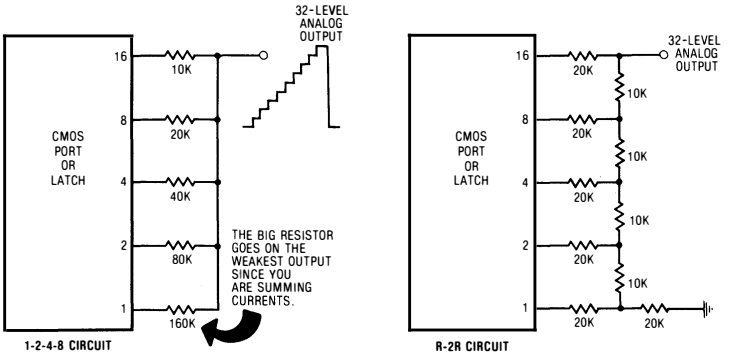
SETTLING TIME—How long it takes for a D/A converter to change to a new output value that is accurate to or better than the resolution.

For instance, an 8-bit D/A converter would have 256 possible analog output levels and thus would be accurate to something like one-half percent. A 12-bit D/A converter would have 4096 possible analog output levels and thus would be accurate to one part in 4096 or around .025 percent.

In general, D/A converters are no big deal. They are far simpler and much cheaper than the A/D converters we will look at shortly. The price does go up with increasing resolution and decreasing settling time, though.

Here are a pair of 5-bit "do it yourself" D/A converters for you . . .

TWO "HOMEMADE" 5-BIT D/A CONVERTERS



In the first circuit, we start with a CMOS port or latch and then add *summing resistors* that are weighted in a 1-2-4-8-16 ratio. These currents are summed to give an output stairstep voltage that has thirty-two levels. Note that the *smallest* resistor value gives the *largest* current, and vice versa.

In the second circuit, we rearrange things to use resistors that are all one of two values. This is done by providing a 2:1 stepdown each time we move one more pair towards the output. This is called the

"R-2R" method and is far and away the preferred circuit, although not quite as easy to understand as the "1-2-4-8" circuit.

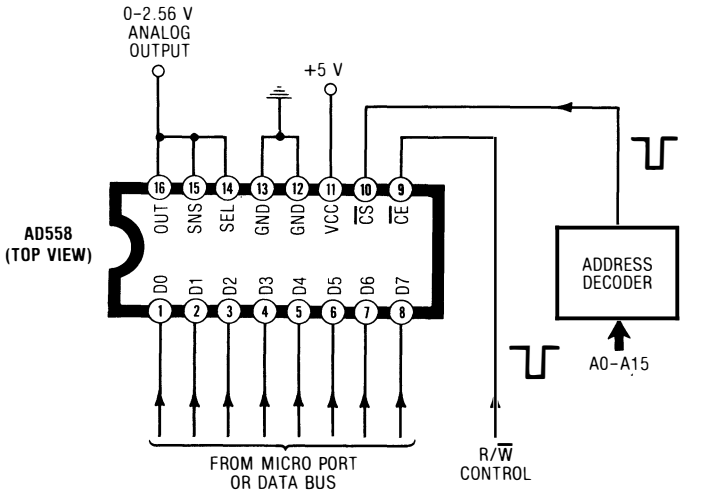
This simple summing of currents of various weights is all that is involved in nearly any A/D converter. The faster you go and the more accurate you want to be, the more precise and more complex the circuit—and the more it will cost you.

Homemade D/A converters are hard to build to accuracies better than five bits because of resistor tolerance buildup problems.

Commercial D/A converters offer much better resolution. To do this, they put all the D/A circuitry into a single well-matched package and add an operational amplifier to the output. This operational amplifier makes the summing process independent of your outside load. It can also speed things up because the currents all sum into the virtual ground of an op-amp's input. A precision voltage reference is also usually included for maximum accuracy independent of power supply variations.

Here is a typical circuit . . .

8-BIT D/A CONVERTER



The way I have shown this circuit, you simply hang the *Analog Devices* AD558 on an 8-bit output port, and you get an output voltage of 0 to 2.56 volts that follows the straight binary coded inputs. All this from the same 5-volt supply the micro uses.

You can get much fancier with this chip. By using a split supply voltage, you can get higher outputs, and you can get faster response

by adding an output pulldown resistor that sinks to a negative voltage. You can also work directly from a data bus, rather than a port, by correctly controlling the chip-select and chip-enable inputs. If both these inputs are grounded, the D/A continuously updates. If both are positive, the D/A continuously holds the old value. Usually, you will route an address into the $\overline{CS}$ input and an "OK to write" command into the $\overline{CE}$ input. Details depend on the micro you are using.

Some fancy variations of A/D converters are available. *Multiplying* A/D converters give you an analog input that gets multiplied by the digital value. These converters range from *one quadrant* multipliers that use positive-only currents and straight binary, through *four quadrant* circuits that handle AC signals and signed binary or special codes. Important uses are audio attenuators, active filters, and electronic music.

Some D/A converters are purposely made nonlinear. The telephone people use *companding* D/A converters to give an approximation to a log response that increases dynamic range. Speech synthesis uses special conversion techniques intended to optimize intelligibility. You can also use oddball resistor weightings to build digital sinewave converters for electronic music or other uses. Details on this appear in Chapter 6 of *The CMOS Cookbook* (Howard W. Sams 21398).

Summing up, output circuit level interface is needed whenever you want the microcomputer to control real-world things such as lamps, motors, solenoids, air controls, steppers, or whatever. Most of the interface electronics will be involved with the non-micro bits and pieces of traditional components.

A level two output circuit interface may do three important things: amplify, isolate, or convert.

Amplification makes things bigger and louder. It usually involves NPN transistors, Darlington transistors, IC peripheral driver arrays, power FETs, or triacs. Triacs are best suited for AC power control of heavy loads.

Isolation separates the load from the microcomputer, both as a safety measure to eliminate shock hazards and as a way to separate load-related noise sources or ground current loops from the microcomputer circuitry. Although relays are one obvious isolation means, optocouplers are more commonly used. Optocouplers use the presence or absence of a light beam to control an output and can use phototransistor, photodarlington, or optotriac output devices.

The D/A or digital-to-analog converter is most often used to convert from digital commands to continuously varying output signals

such as speech, electronic music, or digital audio. Important parameters of a D/A converter are its resolution in bits and its settling time.

Enough said on output circuit interface. Let's now turn our attention to . . .

INPUT CIRCUIT LEVEL INTERFACE

Input circuit interface goes between the sources of outside-world signals and the microcomputer. Once again, this interface is needed for just about every real-world use except those that come directly from nearby, "clean" LSTTL and CMOS compatible sources.

The input interface usually does four things . . .

INPUT INTERFACE can:

( ) protect
( ) condition
( ) isolate
( ) convert

By *protection,* I mean making absolutely sure that all signals actually entering the microcomputer are just the right size to be recognized as legal ones and zeros. The signals should not be so loud that they destroy the micro, nor should they be so small that you cannot reliably tell ones from zeros.

Conditioning is needed whenever you interface a mechanical contact to the micro, because mechanical contacts bounce repeatedly and thus give false counts. Conditioning is also needed to make sure you have a snap-action one or zero as compared to something slowly varying that could cause foul-ups.

Isolation is pretty much the same as it was for output loads. You isolate to eliminate shock hazards and to disconnect the micro physically and electrically from any noise that might be related to the signal source or its power supplies.

Finally, *conversion* takes real-world signals that are not ones and zeros and converts them into suitable digital commands. A temperature sensor or a pressure transducer are two obvious input devices that need conversions. Analog signals are converted with special *A/D converter circuits.*

Let's give the input conditioning the same treatment we gave output conditioning . . .

the protection racket

The only things you are allowed to put into a microcomputer's ports are ones and zeros that are LSTTL or CMOS compatible. Anything else will either fry the chip or, at the very least, cause confusion.

Thus, an important rule . . .

> **The only things you are allowed to input to a microcomputer port are clean LSTTL or CMOS compatible digital logic ones and zeros.**
>
> **Anything else spells trouble.**

For instance, inputting the AC power line into a port will immediately destroy at least the port and possibly also the entire microcomputer. Any signal that is more positive than the positive power supply or more negative than ground, even by a few tenths of a volt, will cause serious problems.

Any signal that changes from a one to a zero very slowly or very noisily is also bound to cause hassles. Although this type of signal probably won't do any physical damage, it is likley to be read wrong and misused.

Most protection circuitry is built into the other input tasks of isolation, conditioning, and conversion. For protection circuits to work, the signal feeding the port must be a legal one or a legal zero, with a rapid snap-action between . . .

> **An input protection circuit MUST provide ONLY legal ones or legal zeros, with a snap-action between.**
>
> **Most protection circuits will also isolate, condition, or convert.**

The point here is to use plain old common sense. If something looks noisy or too big or too small, put some circuitry between it

and the micro. This circuitry should safely handle the signal at the input and safely pass ones and zeros to the micro at the output.

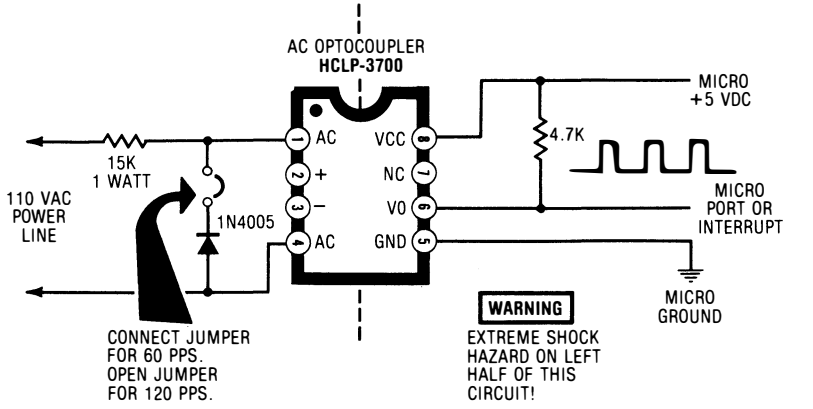
input isolation

Input isolation works nearly the same way as output isolation. You use isolation to avoid shock hazards and to keep the noise associated with input sources from getting to your micro.

Optocouplers are almost always used for input isolation. The phototransistor style is used when you need maximum speed, and the photodarlington is used for maximum gain.

Here is an example of how you use an AC optocoupler to provide a safe power line reference for a microcomputer . . .

POWER LINE FREQUENCY REFERENCE



This special *Hewlett Packard* AC optocoupler directly and safely accepts a full strengh 110-volt AC line signal at its inputs by way of a large current-limiting resistor. This lights one or the other of the input LEDs, except during line zero crossings. The output of the optocoupler will go positive on each zero crossing of the AC line, giving you 120 pulses per second into your port. Extra goodies inside the chip improve the snap-action and stability.

You can add an extra circuit on the output to narrow and invert the pulses if you need this sort of thing for an interrupt line. By adding the diode shown on the input, you can get 60 rather than 120 pulses. Sixty is better for clocks, while 120 is better suited for dimmers and other phase controls.

Note that this AC optocoupler is a very special device. Send the AC line into just about any other integrated circuit or peripheral chip, and you will instantly destroy the works.

If some lower AC voltage is available, such as a transformer winding in a power supply, use it instead of the raw power line. It's far safer and more usable.

This conditioned and isolated power line signal can be used as a source for clocks and counting or as a phase reference for triac dimmers and zero voltage switches.

What other uses can you think of?

DOING IT:

List a dozen uses for a power line input reference sent to a microcomputer.

Whenever you get *any* signal from anywhere remote from your microcomputer, be sure to provide input isolation. In this case "remote" means "not plugged into the same AC wall outlet at all times."

A bad thing can happen when you make physical connections to a remote "ground": you can get heavy ground currents through your micro. At the very least, this changes ones to zeros and vice versa. At worst, it fries the works.

A very important rule . . .

ALL remote inputs and outputs to or from ALL micros MUST be isolated to prevent ground loops!

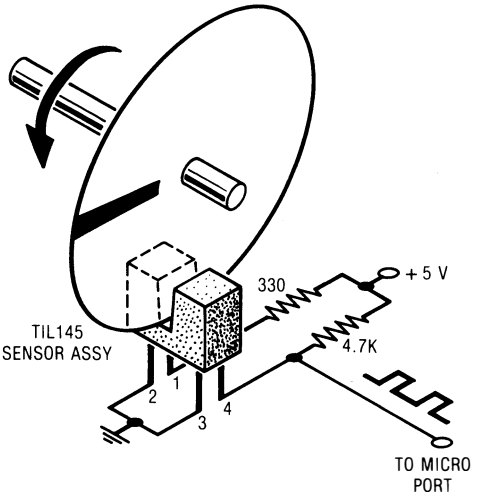
"Remote," means not plugged into the same grounded AC outlet at all times.

And remember, optocouplers can be infuriatingly slow at times. Always check the data sheets to be sure you have enough speed and gain for your application.

Also, many older optocouplers will "run down" after a few months with a time-deteriorating response. Be sure to have enough excess gain or extra drive to allow for this long-term effect.

Here's another sort of use for a different optocoupler . . .

TACHOMETER OR POSITION PICKOFF



What I have done here is to take an interrupting optocoupler pair and interpose a disk with clear and opaque sectors on it. Depending on your use, this can be a tachometer or a position sensor, and it can be used for anything from an anemometer to a trackball to a robotic position sensor to a miles-per-gallon meter.

Since nothing but a light beam touches the disk, there is no mechanical loading and plenty of free play. For more accuracy, you can add extra stripes to the disk. Should the stripes get very close together, you can add focusing optics to increase the resolution.

If you have stripes all the way around the disk, you can also add a second interrupting optocoupler that is spaced so that it sees black when the first one sees white. When combined with some digital logic, this gives you the sine and cosine channels needed to handle the dual problems of detecting position and sensing changes in the direction of rotation.

Still a third interrupting optocoupler can be added to pick off a single *index* stripe that can be used for absolute positioning.

A gotcha: Interrupting optocouplers use infrared light, which can zip right through some very opaque-looking plastics and other

materials. Make sure anything that is supposed to break the infrared beam does in fact do so. Sheet aluminum definitely works.

conditioning inputs

Digital circuits expect clean ones and clean zeros. No "maybe" or "later on" allowed. So, *conditioning* circuitry often has to be added to verify inputs and make sure they are truly ones and zeros . . .

> **INPUT CONDITIONING**—Any add-on circuitry used to eliminate noise and to guarantee a snap-action between input ones and zeros.

Let's look at two examples of input conditoning.

The first involves mechanical contacts. A mechanical contact, such as a pushbutton or a relay closure, does not close all at once. Instead, you get *contact bounce* that may last for a few milliseconds as the contact settles down. During that time, you could get many hundreds of output counts. Input this to a micro that is counting inputs, and you add many hundreds of counts instead of just the one you wanted.

Back in Volume 1, we learned how to add a set-reset flip-flop to a pushbutton to make it bounceless. On the first contact made, the flip-flop changes and "fills in" during the noisy contact-making time.

These days, it is often much better to use software to debounce mechanical contacts. Usually, you wait a few milliseconds after a contact is read and verify that it is still down and not just a noise glitch Then you accept the command as valid. Then you delay as long as needed to be sure you don't get a second hit off a single event.

Whether you use hardware or software, though . . .

> **Some sort of debouncing is needed for any mechanical contact used to input single events to a micro.**
>
> **This debouncing can be done either with hardware or software.**

Note that you don't usually have to debounce a reset button, for resetting your system several hundred times in a millisecond or two will do the job just as well as resetting it once.

But for any contact where you are counting the closures, some sort of debouncing is an absolute must. The debounce time should be long enough to eliminate all bounce effects but short enough to catch repeated legal hits of the contact.

As a second example of conditioning, let's talk about a night/day photocell. For this we might use a *cadmium sulfide photoresistor*, the kind used on street lamps and security controls. The resistance of a photoresistor changes with the amount of light falling on it. Typically, you get 200 ohms in bright sunlight and 200,000 ohms or more in the dark.

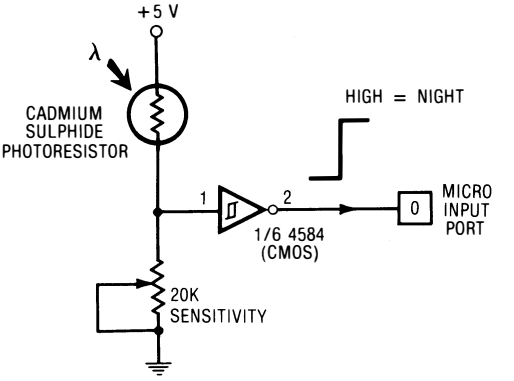
Suppose we are dumb enough to try to input the slowly varying voltage from night to day directly into a micro port. Sure enough, if it is the middle of the day, we get a solid zero. In the middle of the night, we get a solid one.

But in between we get a "maybe."

At the time we want to tell night from day, the voltage is halfway between a one and a zero. The port gets very confused.

Instead, let's add a circuit called a *Schmidt trigger* to our photocell so that it swiftly and sharply changes from night to day . . .

NIGHT AND DAY DETECTOR



What happens is this: The photocell and calibration resistor on the input form a voltage divider. This voltage will be very positive during the day and very near ground at night, since the photoconductive cell's resistance varies inversely with incident light. A regular CMOS inverter has been taught at the factory to provide a "one–zero" decision when its input is precisely halfway between the +5-volt supply and ground. Thus, you always get a one or a zero. A one for day and a zero for night, since this circuit inverts.

But the typical Schmidt trigger circuit does us one better. The circuit also adds internal *noise immunity* or *snap-action* for us. Another name for this snap-action is *hysteresis*.

The snap-action sets a *lower trip level* when the input is *decreasing* and an *upper trip level* when the input is *increasing*. Thus, if there are moving clouds at dawn or dusk, they will get ignored, for, once snapped, it takes a *bigger* change in the "wrong" direction to flip it back the other way.

The Schmidt trigger it converts a slowly varying or noisy analog input signal into a clean digital output circuit that snaps from a one to a zero.

Two popular CMOS Schmidt triggers include the hex inverting 4584, otherwise known as the 74C14, and the 4093, a quad 2-input NAND version.

Conditioning should be applied to any slowly varying or noisy input . . .

Snap-action conditioning is needed for EVERY micro input that is noisy or slowly varying.

A Schmidt trigger is one good way to provide snap-action or hysteresis for noise immunity.

Many line driver and receiver circuits also have built-in snap-action, as do a few of the faster optocouplers. Be sure to use something like this whenever you have noisy or slowly varying input sources.

input conversion

Just about any device you hang on the input of a microcomputer will do some sort of conversion. A temperature sensor converts

temperature to an electrical signal, and a pressure transducer or load cell gives pressure the same treatment. Even a keyboard can be thought of as a mechanical-to-electrical converter or, for that matter, a *thought*-to-electrical converter.

Some of the most challenging and most interesting conversions of microcomputer inputs involve circuits called *A/D converters.* These can change continuously varying electrical signals into digital ones and zeros . . .

A/D CONVERTER—Any circuit or device that will convert continuous, or analog, signals into one-zero, or digital, values.

A/D converters are the opposite of D/A converters. A/D converters take one or more varying, continuous-valued, analog inputs and convert them to a bunch of microprocessor-compatible digital output ones and zeros. These output ones and zeros can be in parallel or serial form, although parallel is more common.

There are lots of important uses for A/D converters. In *data acquisition*, A/D converters are involved wherever analog signals need to be measured and routed to a micro—for example, temperature, humidity, pressure, flow, power, liquid levels, voltage, gas spectra, stress, energy, position, hinklefarbs, force, resistance, current, and just about anything else.

Medical uses for A/D converters include pulse monitors, EKG units, brainwave sensors, biofeedback, thermometers, and many specialized diagnostic instruments.

Voice and *music* uses include speech recognizers, electronic music synthesis, and digital audio, which is delivering us zero surface noise, wow, and flutter, along with incredible dynamic range.

There are many *video* and *television* uses of A/D converters, including video disks, time base correctors, studio processors, special effects generators, and change-detecting security monitors. Once video is digitized, there are all sorts of mind-blowing ways to enhance or otherwise modify the stored images.

Unfortunately, A/D converters are more complicated and more critical in their use than D/A converters, so they often cost more. The price of an A/D converter goes up with increasing resolution and response speed. Often, a small increase in the number of out-

put bits or in the maximum conversion rate will skyrocket the cost of the devices.

Costs of some older A/D converters that are both fast and accurate have gone into the hundreds, even thousands, of dollars. Fortunately, newly available integrated circuits are dramatically lowering these costs.

The Schmidt trigger photoresistor sensor we just looked at is really an A/D converter. And it is quite cheap (around 20 cents) and very fast (around 200 nanoseconds) compared to most A/D converters. As with any converter, it has a slowly varying or continuously changing input signal, which in this case is decided by the changing resistance of the photocell. The A to D circuitry changes this into a sharp, snap-action one or zero digital output.

This sounds great, except for one tiny detail. This is a one-bit converter that can give only one of two possible binary output values, a one or a zero. Thus, you could use this one-bit A/D converter as a temperature controller or a freezer alarm, but you could not use it to measure and display temperature because it gives only two possible output values.

For most A/D uses, we need many more output bits and thus much more conversion accuracy. Eight-bit converters are fairly popular, reasonably priced, and easy to use. These can slice an input signal one of 256 ways, giving a potential accuracy of a fraction of a percent. This is good enough for many industrial uses, where things were traditionally measured by analog means to only a few percent accuracy.

To capture video, we can often get by with only six to eight bits of resolution but the converter has to run very fast. Conversion rates of twenty megahertz are not unusual. Digital audio goes to the opposite extreme. An A/D converter for digital audio only has to run medium-fast, but it may take an ultra-precise sixteen bits of resolution to pick up the needed dynamic range.

Digital voltmeters and other instruments also need very accurate A/D converters, but these can take their good old time about making a measurement. So here you need lots of accuracy but very little in the way of speed. In fact, too many measurements per second can become annoying and hard to read.

Because of the many possible uses for A/D converters, there is no one best type that will do everything for everybody. For some uses, you just plug in a cheap and ready-to-go chip and you are home free. Others will take card after card of complicated and critical circuitry.

Here, in order of *decreasing* cost, are the more popular ways of doing A/D conversion . . .

TYPES OF A/D CONVERTERS
Brute force Feed forward Successive approximation D/A and compare Multiple slope Voltage to frequency

The *brute-force* or *flash* converter is the most obvious, the fastest, and takes the most parts. It is the best way to digitize video or to process sophisticated radar signals. With a brute force converter, you make one comparison for every possible analog signal level. Thus, for an 8-bit converter, you use 256 comparison circuits tied into a reference voltage divider with 256 different levels. The 256 outputs from the comparison circuits are then encoded into an 8-bit, 1-of-256 output code. Since the conversion process takes place continuously in a single step, it is the fastest possible converter you can build.

At this writing, single-chip integrated circuit brute force A/D converters are becoming available from *TRW, Motorola,* and *RCA,* with costs in the $30 to $90 range. This pricing should dramatically drop in the near future.

The *feed-forward* converter is a fairly new idea that looks like a real winner. What you do here is brute force convert the upper bits by brute force, D/A convert this result, subtract the result from the input, amplify the input, convert the remaining lower bits by brute force, and then combine the upper and lower output bits for the total conversion.

This feed-forward process takes two or three steps, so it is somewhat slower than brute-force conversion, but it takes far fewer comparison circuits. Many of the newer A/D conversion chips are switching to this technique, since it gives you both speed and accuracy with a reasonable number of parts.

Digital audio is an important use area for feed-forward converters.

The classic A/D conversion method that the feed-forward method is replacing is called *successive approximation.* In this, you see whether the input is over halfway up. If it is, you subtract half of the signal and see if the remaining input is over a quarter full size. If it is, you subtract another half of the signal, and again see what is left. You keep this up, subtracting out halves, quarters, eighths, sixteenths, and so on until you get the desired accuracy.

The method of successive approximation requires only one comparison per bit but needs many trips through the circuit to pick up all the comparisons in order. The feed-forward method, though, is turning out to be considerably faster and simpler.

The *D/A and compare* method is very old and very poor. What you do is ramp up a fairly cheap D/A converter and compare the analog result against your input. The conversions are slow and noisy. I include this method here only because it is so obvious.

The *multiple slope* method is slow but very accurate. Its main use is in digital voltmeters and other measuring instruments where only a few conversions are needed per second. The simplest of these is called *dual slope*. In dual slope, you charge up a capacitor for a fixed *time* with your unknown input current, and then discharge the capacitor back to ground with a larger and precision reference *current*. You then measure the ratio between the charging time and the discharging time to find the unknown input current as a ratio of the reference current. Because the capacitor charges up opposite to the way that it discharges, many circuit nonlinearities cancel.

Also, if you charge the capacitor only for an exact multiple of the power line frequency, you can get any hum on your input to cancel itself out almost completely. Refinements in the dual-slope process lead to triple- and quad-slope schemes that are more stable, more accurate, and capable of calibrating and zeroing themselves automatically.

Multiple-slope A/D converters are most often used in digital instruments, where you can get a three or four decade decimal accuracy at the speed of a few conversions per second. *Intersil* is one leading supplier of this type of A/D converter. Some of these are micro-compatible, and others are used to drive digital displays directly. Their cost is in the $4 to $10 range.

Finally, there are the *voltage-to-frequency* A/D converters. These might instead involve current-to-frequency, voltage-to-time, or current-to-time, depending on what you want to do. Any of these is cheap. They are also inaccurate and only medium fast. What you do is build some sort of oscillator or monostable and then voltage control the beast. Then you count output cycles or measure the time per cycle and use this as a digital value. You can build these out of 555 timers or from bits and pieces of the voltage-controlled oscillators intended for phaselock circuits. *Raytheon* has some more precise V/F devices that do a better job than you can manage on the "do it yourself" route.

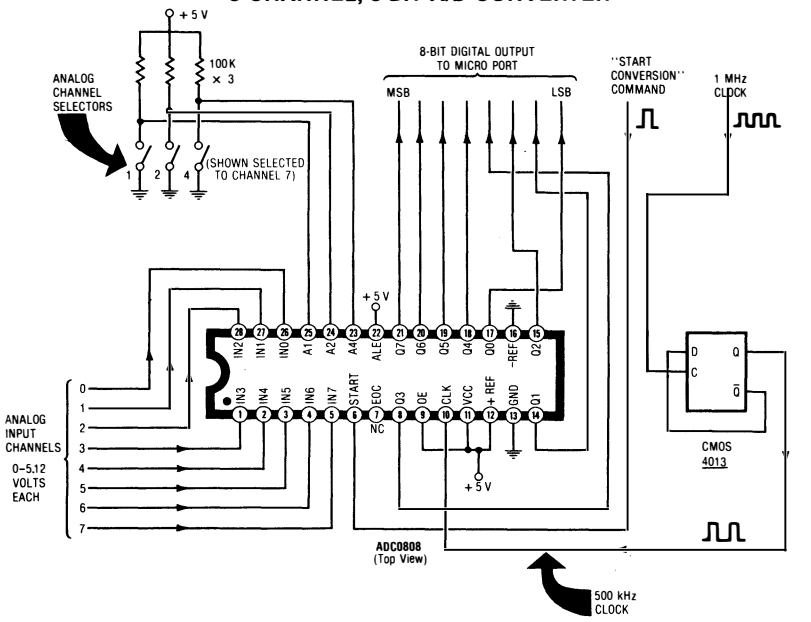
Important uses of these low-cost A/D converters are in game-paddle inputs and other places where a fairly limited accuracy is acceptable and cost is super important.

As an application example, the game paddles on an Apple II are routed to four separate current-to-time converters. The current changes as you change the paddle setting. This changes the time that the output stays active after reset. The output is measured again and again by software, racking up one count each time until the output ends. Your final result is a number in a register that is proportional to the "on" time of the converter, which in turn is proportional to the input current determined by the paddle setting.

Summing up, there are several ways to do A/D conversion that vary in resolution, settling time, and cost. The faster or the more accurate, the higher the cost. Use V/F converters for quick-and-dirty uses like game-paddle controllers. Use multiple-slope converters for digital instruments and other very accurate but very slow needs. Use successive approximation or its newer and better feed-forward replacement for places where you need both reasonable speed and reasonable accuracy. Finally, use brute force if speed is everything and you don't care how much it costs, such as for video or radar uses.

Here's a very cheap, very interesting 8-channel A/D converter that is fun to use . . .

8 CHANNEL, 8-BIT A/D CONVERTER



This has to be one of the cheapest, simplest, and easiest to use A/D converters anywhere. *National* builds it using CMOS technology, with several second sources. You simply hang the chip onto some port lines, apply a single +5-volt power supply, add a clock, and away it goes.

There are eight different analog inputs, selected with some internal CMOS switches. You can pick a channel by giving binary codes to the address lines, either from a switch or from three output port lines. The clock frequency should be 500 kHz or less. Usually you can find a 1-megahertz clock inside your micro and binary divide it by two as shown.

To do a conversion, pick your channel and then briefly pulse the START line high. Wait at least a hundred microseconds and read the answer on the data lines.

That fast and that easy.

One-channel and 16-channel variants are also available from *National, Analog Devices,* and others, and they have many options for use. As shown, the circuit responds to inputs from 0 to 5.12 volts, giving one count for each 20 millivolts of input. You can change this to other values, latch the addresses only when you want, or use an "end of conversion" signal on pin 7 to speed things up. Check the data sheet for more details of use.

The only thing wrong with this chip is the 8-bit resolution. This is great for the fraction-of-a-percent accuracy needed for industrial uses but is far short of the accuracy you would want for audio or speech uses. Since the digital audio market is now turning the corner, we can expect cheap, fast, and accurate A/D and D/A chips to arrive shortly, say sixteen bits in 20 microseconds for $15.

There are several restrictions on the use of the faster and more accurate A/D converters . . .

High accuracy A/D converters:

( ) Need a front-end sample-and-hold circuit.
( ) Need an input filter to eliminate aliasing.
( ) Are very sensitive to ground currents and circuit layout.

The first use restriction is that you usually have to add a circuit called a *sample-and-hold* between your analog signals and the con-

verter. A sample-and-hold catches the signal and keeps it at a constant value during the measurement interval. This makes sure that the value you are measuring does not change during the measuring time. If you do not use a sample-and-hold on an accurate converter and still want to get only correct answers, you are limited to ludicrously low frequencies. Thus, a 16-bit, 20-microsecond A/D converter should be good to 25 kHz with a sample-and-hold but may be limited to 4 Hz maximum conversion speed without. That's hertz, not kilohertz!

Why such a slowdown? Because an input sinewave can change by more than the resolution you want in a fairly short time, which completely goofs up your accuracy. It's somewhat akin to trying to detail paint an automobile while it is speeding down the road. Any motion at all will make for sloppy results.

So, all serious uses of high accuracy A/D converters must input by way of a sample-and-hold circuit. This might be provided internally, or you may need external add-on parts.

Another hassle with A/D converters involves their input frequencies. There's a rule that says you are never supposed to sample anything less than *twice* per cycle if you intend to reconstruct that signal. If you send in faster signals than this, you can get wildly wrong low frequency artifacts that go by the name of *aliasing*.

To prevent aliasing, you must low pass filter your input very sharply to be sure there is no energy at all beyond *one half* the sample frequency of your sample-and-hold. As an example, if you are sampling once each 20 microseconds, this equals a sampling frequency of 50 kHz. Which says that the highest allowable input frequency is 25 kHz. Anything above 25 kHz can cause aliasing and must be eliminated. In this case, a very sharp 20 kHz low pass active filter should do the trick.

More details on active filter design appear in *The Active Filter Cookbook,* (Howard W. Sams 21168). Remember that a very sharp low pass filter is essential in any sampled data system if you want useful results.

Finally, you have to watch your circuit layout and ground currents *very carefully* on a precision A/D converter. Say you are using a 16-bit A/D converter on a 5-volt maximum input signal. The resolution here is one part in 65536, or around 76 microvolts. Should any ground current from anywhere get in series with your input signals, you will lose all your accuracy and then some. It is very easy to pick up *tenths* of volts of microcomputer ground current noise. These ground currents are thousands of times stronger than what you are trying to resolve.

The process of properly protecting A/D inputs is sometimes called *guarding*.

Usually the data sheets spell out how to ground and connect a precision A/D converter properly. If you aren't careful, all precision can be lost simply because one trace on a printed-circuit board is arranged wrong or is just too thin.

That just about wraps up A/D converters and, for that matter, this chapter on I/O. Next, let's find out how to attack real-world micro problems and look into some of the things that may need attacking.

things they never tell you in computer school

IATROGENESIS

That fancy word means "physician-caused disease."

To you as a micro user, it means simply that you are your own worst enemy.

Almost invariably, if something does not work, it is NOT the fault of the hardware. It is, instead, your own stupidity coming home to haunt you.

You may have connected the hardware wrong or left it unpowered. Mode switches may be wrong. Something may not be initialized. You may have the wrong software on the system. Perhaps you bent the pin on an integrated circuit when you inserted it in its socket. You may not have what you think you do in the machine. Or your software is so lousy that it couldn't possibly work anyway.

Or, as a more subtle example, you find a mistake in a program, correct it, and the program *still* doesn't work. Why? Because you forgot to repair the damage done when the program bombed the last time.

Generally, you are never anywhere near where you think you are at any point in attacking a microcomputer problem. Instead, there is almost always something much simpler and much more fundamental between where you are and where you want to be.

The key here is *always* to blame yourself first, and the hardware last.

— nine —

The Micro Applications Attack

By now, you should be able to write, test, and debug machine language modules that are short, simple, and well defined. You should also have a good handle on the micro and device levels of I/O interface.

But real-world problems are *never* short, simple, and well defined. Instead, they are almost always long, and complex, and have lots of loose ends. In fact, punching code into a micro is the *least* of your worries when you attack any serious micro application.

How do we move from here to the real world?

One thing that will help bunches is to step up to *assembly language* programming. Assembly language programming eliminates much of the tedium and dogwork of hand coding, and it makes things very easy to change and save. Besides, assembly language programming is lots more fun. More on this shortly.

The other big thing you will need before you can attack real-world problems is some orderly way to get from problem to solution, using a method that works.

I call my method the *Micro Applications Attack* . . .

> **MICRO APPLICATIONS ATTACK**—An orderly 14-step method for solving real-world microcomputer problems.

The micro applications attack is best used when you want to combine a microcomputer with some outside hardware to do some "shirtsleeves" task.

Remember, the biggest nickels are to be made doing just this—putting micros to use solving everyday problems for non-micro people. And remember that a creative mix of hardware and software will *always* give you a better solution than either software or hardware by themselves.

The micro applications attack shines for these uses. Naturally, the method is totally different from the way the dino people used to do things before they got laughed out of credibility.

Anyway, here are the fourteen steps of the micro applications attack . . .

1

BRIEF WRITTEN DESCRIPTION OF THE PROBLEM

This obvious first step is often left out, which only causes serious trouble later. Tell us—in twenty-five words or less—just what it is you intend to do, for whom, and why.

If you can't reduce your objective to a few simple words, then you probably don't know what you want to do, or you may be attacking a problem that a micro can't solve.

More important, this brief written description gives you something to fall back on should someone paying for your problem solving say "But that's not what I wanted!"

This first step forms both the charter and the focus for all that follows.

2

DETAILED WRITTEN DESCRIPTION OF THE PROBLEM

Next, tell us in people-type words, but in more detail, just what you want to do.

Pretend you are writing an essay or an English theme that will explain to someone from Portugal or Alderon VIII exactly what you plan to do.

Keep this step simple and nontechnical, but do not omit key details. If numbers or constants are involved, spell them out. Fill in the whole problem on the framework you laid down in step 1.

There are two reasons for this step. Once again, if you can't tell us, you probably don't know. This step is another something to fall back on and say "But here is what you asked for, and here is what you got."

The word "grok" comes from *Stranger in a Strange Land* and means to go deeper than and beyond understanding. To grok a problem, you go past total and absolute comprehension.

Don't go past here till you grok the problem.

3

PARTITION HARDWARE AND SOFTWARE

Decide how much of the problem is to be handled by dedicated hardware and how much is to be done with software inside a stock microcomputer.

Some of the problem probably has to be tackled with hardware. Obvious examples are high power interface, signal conditioning, and analog conversion. Other parts of the problem demand software solutions. Examples here are complex calculations and anything irrational or involving communication with people.

The best mix of hardware and software generally uses as little dedicated hardware as possible. Once you have decided on a partitioning, always ask yourself if there is anything else that can be handled by the software.

Chances are that later on you will change your hardware/software mix. But setting down a tentative list of who-does-what now is essential before you can continue.

ASSIGN PORT CODES

This may sound way too early in the game, but now is the time to identify and name each and every wire between microcomputer and hardware interface, and between the hardware interface and the real world.

Show which micro ports are to be used and which are to be inputs and outputs. Name each lead with a suitable mnemonic. Often five letters are a good choice—for example, use RDEST to name a line that controls a red traffic light in the east-west direction.

The reason for naming the leads now is so that you can talk about them. You may want to change the names and their meanings later on. The idea for now is to set up a talking model that you can work with.

If you have too many I/O or interconnect lines, now is the time to rethink what you want to do. Can an X-Y matrix help you? How about multiplexing? How about hardware decoding or encoding? Or, go the other way. To what uses can you put the "free" input and output lines that are left over? How can you make your system more general so that it can solve a whole class of problems instead of one specific one?

The output of this step should be a block diagram of three boxes—the microcomputer, the input hardware interface, and the output hardware interface. All major interconnections should be shown and labeled. The diagram should also show whether the hardware will have its own power supply or will tap the main microcomputer supply.

5

DRAW TIMING DIAGRAMS AND DECISION TREES

Some micro problems are time-intensive and demand outputs that are correctly spaced and sequenced in time. Other problems are result oriented and require certain things to happen if and only if other things have already taken place.

Draw whatever you need to show what will happen why and when. For some problems, this will take the form of a detailed timing diagram plotting all outputs against time for all given input conditions or all chosen courses of action.

Other problems will call for a decision tree that shows the result of every input or command.

Your output for this step should be very graphic. Use charts and plots to show in detail what the system is trying to do. Use pictures, not words.

DO A BLOCK DIAGRAM AND FLOWCHART

The trick in this step is to describe the system completely and accurately without getting bogged down in excessive detail.

At this stage of the game, your blocks on the hardware side should describe each thing the hardware is to do but should not pin down specific devices or part numbers. For instance, a solenoid driver should be labeled "OUTPUT DRIVER" but not be a detailed schematic of one-eighth of a ULN2803.

Your software flowchart should be concerned only with the big lumps, not with actual code. For instance, a 0.1-second delay loop should be shown as a block labeled "STALL 0.1 SECONDS" rather than in individual coded steps.

All interconnection lines, all inputs and outputs, and all port lines and codes should be clearly labeled with the mnemonics and names you have already chosen.

After completing this step, look things over and think them through. Is this what you want? If not, go back to the earlier steps and redo everything till you reach the point where you are happy with your planned problem solution.

7

ATTACK THE STICKIEST BOX

You decide what the stickiest box is. It's the thing in the block diagram that you are the most worried about and feel the least comfortable with. If you are a software person, the sticky box will probably be some hardware interface circuit. If hardware is your bag, it might be a time-critical software loop or fast access to a file of data.

Once you identify the stickiest box, make a simple model of it. Then simplify the simple model. Then make up a really stupid and dumb way to test the simplified version of the simple model, something any idiot could manage. Then do it.

You'll be surprised at the results every time. The dumb test on the simplified version of the simple model will almost always show you a brand new and much better way to attack your original problem. Other times, it may show you that the *real* job to be done is much tougher than you thought.

The important thing here is to zero in on the thing most likely to foul up the works, and then nail it down.

If there are several sticky spots in the problem, attack them in order, starting with the one you feel worst about. Do not waste any time yet on things you know how to do or feel confident about.

Remember that inside every large problem there's a small problem desperately trying to get out.

8

BUILD SOFTWARE AND HARDWARE MODULES

At this stage of the game—and remember it is *all* a game—you should have a fairly good picture of what should be done and how to go about doing it. The "What if?" and "Will it?" questions should have been answered in previous steps.

Now comes the fun part. Build up simple hardware modules and test each module individually. In fact, test each step of the construction of each module individually. Put together only enough of the circuit to make it do something. Then test that something in the simplest and most general way possible.

Do each block on the hardware side, building and testing everything individually. That's by itself. Alone. Singly. One at a time.

On the software side, it is nearly the same game. Work up your utility subroutines, the simple ones that will be called on over and over again both by your main program and by fancier subroutines. Think modular. Test each subroutine separately, doing the simplest possible test in the most simpleminded way you can think of. Keep track of reserved locations, mnemonics, restrictions on working registers, pass-thru conditions, and so on.

The results of this step should be two piles of blocks. Hardware on the right, software on the left. Each block should effectively attack some small part of the whole problem.

DO AN IMPROVED FLOWCHART AND SCHEMATIC

Take all the hardware and software modules that you have tested and use these to put together your complete system schematic and a detailed flowchart.

Your hardware circuitry should require very little other than the modules you have already tested. Your main program should consist almost entirely of subroutine calls to modules that are already debugged and tested.

If there is anything at all fuzzy remaining, go back through the steps, as far back as you have to. Soldering the hardware and coding the software is all that should remain after you complete this step.

10

WRITE, TEST, AND DEBUG YOUR CODE

If your hardware is going to do fancy things with your port lines, you may want to stop here and dream up some simple tests that will let the software and hardware interact.

When you are finally confident that everything is hunky dory, then and only then should you write the main code for the main program. As with any code, you write it on a program form, working from a detailed flowchart. You then list the code. Then single step the program, or trace it, or use breakpoints.

Always test the smallest portion of the code that you can at any time, and NEVER just assume that any code is correct or working. Almost always, the problems that crop up will be far simpler and stupider than you'd expect.

As in mountain climbing, lots of small and short steps are what get you over the top.

Once you have working code, test out your hardware and software. Be sure to make your code "bulletproof." Your tests should allow for hitting the wrong keys at the wrong time, malicious or disallowed inputs, restarting from the middle, and any other fiendish thing you can think up.

If you don't think them up, someone else will.

Remember, Murphy was an optimist. If there is any conceivable way a thing can get fouled up, it will. Even if there isn't, it still will.

11

HAVE A KNOWING OUTSIDER TEST IT

Here's the frustrating part. You get everything working perfectly, and now the real user wants it to speak Flemish. Or couldn't care less about the one feature you went to the most trouble to provide. Part of your solution is gross overkill, and part of it simply doesn't do the job.

Or maybe this is the joy of the whole game. A single "What if?" or "Can you?" question from the user opens up a whole new world for you, a way to use your micro to do things so much faster or better than the old way that it was unthinkable before. What was a handy feature now becomes an overwhelming benefit.

Listen to the user. Listen to several of them if you can. Try not to be defensive. Put yourself in their position.

If something is very wrong with your design, go back and fix it, working back through the steps. But if it's only a nice add-on, or some minor improvement, hold up for now. If the job is more-or-less complete and more-or-less working, that's all we need at this stage.

12

ANNOTATE AND DOCUMENT EVERYTHING

All along, you should have been keeping an engineering notebook, a diary, or whatever on what you were up to. Now is the time to make clean, final copies of everything done so far.

Documentation is well over half of what micro use is all about. Spend lots of time on this. Make clean, neat, and detailed user manuals that show "how it works." Clarify and add to the listings. Show key waveforms and test points. Create software test programs to isolate problems. Design a troubleshooting flowchart.

Make things as complete as you can, so that you or anybody else can return a year later and easily pick up without losing any time wondering what memory location $FC23 is intended for, why there is an illegal op code in location $023F, what the function of C17 is, or how IC6 gets its supply power.

13

SIT ON IT

If you can, set the whole thing aside for a month. This is especially needed if your solution will be written up in a book or something else that's to be widely read or used, or if it will be a high volume item with high "front end" expenses.

During that month, think about how you really would have done things, knowing what you know now. Have others test and evaluate some more. Deliberately look for ways of destroying both the program and the interface.

Explore ways to make what you do more general and more convenient. Find ways of using extra inputs and outputs to provide new and handy features that weren't cost effective before but now are "free."

Seek out ways to improve your coding and make things more compact. Find ways to use mainstream or cheaper hardware, or ways to use very expensive but simplifying parts that are destined to become cheap and mainstream over the life of the product.

14

EVALUATE AND IMPROVE

After all the dust settles, try to separate what you'd like to do from what really needs doing. Working code is a joy to behold and should NEVER be messed with. Don't *ever* write over existing work. Always save old versions of things you know are good.

Now is the time to rework your entire problem solution, doing only those things that clearly improve it and make it cheaper or more flexible. Don't try to save a few words of code unless there is a *very* good reason to do so. All programmers tend to whip dead horses, spending far too much time and effort at the end of a project for only little added benefit.

Improve only what genuinely needs improving. Leave the rest alone.

Iffen it ain't broke, don't fix it.

As you can see, the key to the micro applications attack is getting at the "stickiest box" as quickly and as easily as possible. For it is the sticky box that will show you the *real* problem that you want to solve.

The dino people believe in a quaint and oddball way of doing things called *top-down* programming. Now, top-down programming is very useful if you are a project engineer in a large corporation and want to snow your boss and improve your upward mobility by papering the wall with impressive garbage.

But otherwise, top-down programming is worthless.

Why?

Because of the stickiest box.

Top-down programming breaks things down into big lumps, little lumps, and crumbs. It inherently assumes that the big lumps deserve more attention and effort than the little lumps and that the crumbs are trivial details.

In reality, it is one or two of the crumbs that are the crux of the problem. In these crumbs lies the secret to doing something even better and greater than you had first planned.

With top-down programming, you don't get to the crumbs until it is too late. At that point, you either ignore the key crumb or scrap all the effort made so far. Either way is a bad scene.

There are other programmers who believe in *bottom-up* programming. The problem here is that you tend to do all the trivial and easy stuff first and, once again, save the sticky box for last. As with top-down programming, it ends up with either wasted effort or a solution that really isn't optimum.

So . . .

The key to the micro applications attack is finding the "stickiest box" as fast as you can.

It is always this stickiest box that holds the secret to what you really should be doing and the best way to go about doing it.

Note two very important points about the micro applications attack. First, the process is highly re-entrant. Any time you finish a box, you don't just say "Well, that's done and we got this far." You have to go back up to the top and find out how completing the most recent step changed the earlier boxes.

Always go back after each step and revise all the earlier steps as needed. The micro applications attack is a discovery process that will automatically lead you to an optimum problem solution—if you give it a chance.

The second important point is that very few of the steps in the micro applications attack have anything at all to do with a computer or involve any actual computing. How many of the steps really need a microcomputer? Parts of steps 7, 8, and 14 do, but even here the micro is being used for something besides actual work on the main program.

Only step 10 needs the microcomputer. Hence this very important point . . .

> **If you are new to the micro world and want to solve a real-world problem:**
>
> **STAY AWAY FROM AND KEEP YOUR GRUBBY PAWS OFF THE MICRO TILL THE LAST POSSIBLE INSTANT!**

An all-out attack on a real problem takes lots of careful thinking and lots of pictures, drawings, block diagrams, and charts. Very heavy and thorough annotation is also a must. All this is best done in a quiet place, free of all distractions. And away from the micro.

I've seen it happen time and time again. Beginners *always* want to go right ahead and start punching in code without giving any thought to anything else.

Well, kiddies, it just won't work.

You may fancy yourself a whiz kid, but someplace, somewhere, there is a program that you can't keep in your head all at once. There is also some memory span beyond which you can't remember all the details you think you know for any given program. Let either of these happen and you are back to square one with—absolutely nothing!

Remember the *sooner* you start punching in code, the *longer* the job will take.

If a problem is worth solving, it is worth doing in a way that you or anyone else can return to later, understand, and be able to continue without having to reinvent the wheel. That's why the micro applications attack exists in the first place, and that's why the bulk of your time should be spent *away* from the micro.

As with any problem solving method . . .

> **The first 90 percent of a problem uses up the first 90 percent of the available time.**
>
> **The last 10 percent of a problem uses up the last 90 percent of the available time.**

So budget your time accordingly. The unknowns and surpises along the way will gobble up practically all of the available project time, and then some.

USING THE APPLICATIONS ATTACK

I'm not going to show you a detailed example of how to use the micro applications attack. In the first place, we are far into a very thick book. But second, and more important, remember the winning rule of hands-on being everything.

You must use the applications attack on *your* own terms, not on mine. So, here are a few suggestions for suitable projects to try on your own . . .

DOING IT:

Project A

Use the micro applications attack to simulate a one-intersection traffic light.

Provide "night" and "day" modes that use a photocell to switch to blinking red and yellow at dusk.

Add a switch-tripped fire override that gives a fire truck the green light when activated.

But that's doing things the old way, not the new micro way. Instead, try . . .

DOING IT:

Project B

Show a "generalist" way to output up to 128 patterns and up to 128 time-delay values for up to eight LED lamps. Use data files.

By changing only file values, show the previous traffic light, a new traffic light with an "all red" intersection-clearing safety feature, a model of a 1-second analog pendulum, a disco chaser, and a theater stage lighting system.

Here's one that will give you some device level I/O practice . . .

DOING IT:

Project C

Use a microcomputer and a power line interrupt to control a 100-watt light bulb.

Show that you can turn the bulb on for 10 seconds of each minute. Then run the bulb from bright to dim over a 2-minute interval.

Finally, add a photocell sensor that keeps a constant room lighting level. As the outdoor light increases, the bulb brightness should decrease, and vice versa.

And here's some A/D converter practice . . .

DOING IT:

Project D

Interface an 8-bit, eight-channel A/D converter to a microcomputer.

Display 0–5 volts over one channel set up as a digital voltmeter. Show the indoor temperature over a second channel and the outdoor temperature over a third channel.

Measure resistance with the fourth channel. Measure pressure or weight over the fifth channel, using a piece of IC protective foam as a sensor.

Sense the liquid level in a tank with the sixth channel. Find some mind-blowing uses for the last two channels.

Combine all this into a graphic display that is attractive and easy to use.

And here's some practice interfacing two wildly different electronic systems . . .

DOING IT:

Project E

Interface a microcomputer to a BSR power controller by using an ultrasonic microphone to let the micro simulate the BSR hand-held remote keyboard unit. Or else use infrared pulsed light.

Show some use of your interface that involves a drip-irrigation plant watering system.

And, if those projects are too easy for you . . .

DOING IT:

Project F

Write a machine language program of one hundred bytes or less that will simulate the known universe over all time.

Show all simplifying assumptions you have made. Indicate the initial and final program states.

Be able to run the program both faster and slower than real time, both forward and backward, while using a game paddle as a speed controller.

Remember, *everything* run on a microcomputer is a game. Without exception.

NOW WHAT?

It's sure been a long two volumes. But if you got this far and did everything along the way, you now should be able to write, test, and debug small machine language programs on your own and integrate those programs into solutions of real-world problems.

By now you should be nearing the limits of the trainer you picked for your discovery modules and should want to automate much of what you did, picking up disk storage of programs, printed records, and heavier use of bigger microcomputers.

So where do you go from here? What do you do next?

First and foremost, practice using the micro applications attack on fairly small projects until you have the method down cold.

Second, pick us up in Volume 3 for more details on using the hundreds of mainstream integrated circuits that are popular for microcomputers today.

Third, tear apart the winning machine language programs of others to see what makes them tick. This is one sure way to pick up the mainstream programming ideas and see how the known winners are doing things. In fact, you should *never* buy or use any microcomputer program without first tearing it down to see what is inside it and how it works.

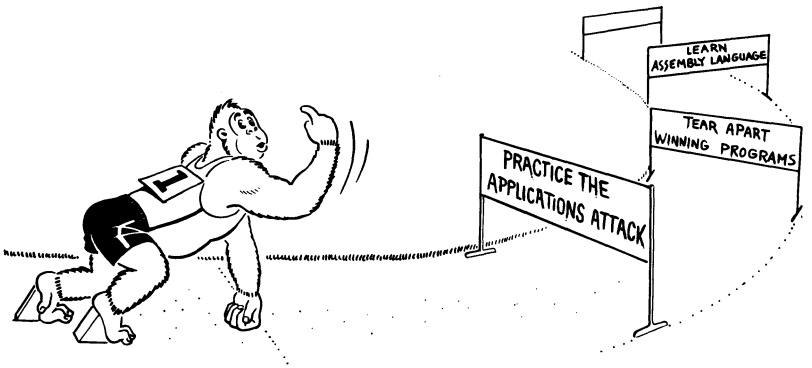
Full details on an astonishingly fast and easy way to tear apart machine language programs appears in Enhancement 3 of *Enhancing Your Apple II, Volume 1* (Howard W. Sams 21846).

Fourth and finally, step up to assembly language. Assembly language eliminates most of the nastiness and tedium of hand-coded machine language programming and gives you ways to easily save, store, and change your programs. Among the other benefits of assembly language programming is that all relative branches are figured out automatically, making it a simple matter to lengthen or shorten a program without hassles. Moving a program to run elsewhere in memory is equally easy.

There is no good way to talk about assembly language programming in general. You are better off picking a microprocessor family and learning a specific assembler system for that family.

But do NOT try to do any assembly work till after you have gone through the discovery modules and after you have solved several real problems using the micro applications attack. Assembly language programming is simply too powerful a tool to use until after you have hand coded and hand debugged not less than several hundred lines of machine language code.

If you did get through the discovery modules and a project or two, then you are ready to step up to the wonders of assembly language. It's a whole new world. One that's lots of fun, very creative, and most profitable. So, where do you go from here? . . .



Where you go from here is all up to you. Remember that hands-on is everything. Follow your own vibes, do what you want, and go in the direction you really want to go.

SIXTY-THREE IDEAS

Let's end this volume with a collection of sixty-three microcomputer ideas that you can profitably use or adapt. Few of these ideas have yet been fully explored at this writing, but all of them seem to point to viable and creative future uses of microcomputers.

Ready? Here we go . . .

—[1]—

One of the big things in computer graphics today is called **anti-aliasing.** This lets you get the "jaggies" out of slanted lines without an increase in display resolution. So far, the idea has been used only on very expensive systems. Can you use anti-aliasing on a micro to upgrade graphics displays cheaply and quickly?

—[2]—

You can now get small keyboards and lap keyboards, but there still is no **truly portable keyboard** that sits on your lap with no apparent connection to the host micro. With CMOS encoders and UARTs, you could build a two-wire interface, possibly using a telephone connector. But what we really want is something that talks to the micro with ultrasonics or infrared and is truly and totally portable. To guarantee long battery life, the system should draw negligible power until a key is pressed. Could this be solar powered?

—[3]—

What can be done by adding **foot pedals** to a microcomputer? On some word processor programs, a left pedal for CTRL and a right one for ESC might really speed things up. Electronic music foot pedals like those made by *PAIA* should be ideal. Or even a bunch of them, pipe organ style. Any game uses you can think of? Perhaps a bulldozer simulator with real controls?

—[4]—

How about a **digital compass** the size of a *Brunton,* accurate to a tenth of a degree and not costing an arm and a leg?

—[5]—

There are all sorts of exciting uses for microcomputers in a **volunteer fire department.** Almost any small department can lower their insurance rating by a full ISO grade simply by cleaning up their paperwork and recordkeeping acts. Obvious and immediate uses include training and attendance records, hydrant checks, planning,

annual reports, standard operating procedures, hose records, engine maintenance logs, and much more.

—[6]—

We still seem to have no cheap and effective **humidity measurement** scheme. How about going back to square one with an old fashioned sling psychrometer, only micro controlled and with a fan instead of the sling? Use the micro to do the involved math conversion and give a direct readout of dewpoint, relative humidity, and temperature.

—[7]—

Here's a real heavy. What happens if you hook up a plotter *backward*, replacing the pen with a photocell? You end up with a **flying spot scanner,** that's what. Scan the plotter and read the photocell. Do either X-Y or edge following scans to capture fancy lettering fonts and to input data much faster and more accurately than a person can by hand.

—[8]—

How can you make a microcomputer **meow** like a cat?

—[9]—

There are lots of old fashioned DNC **numeric machine tools** lying unused in many machine shops. Replace the teletype and the paper tape with a disk-based micro edited by a modified word processor and watch these old beasties shine.

—[10]—

Why not use a **real joystick?** Outfits like *Jerryco* and others still carry World War II surplus B-17 flight controlling joysticks for around $30, actually less than your typical game paddles. These sure are impressive and rugged looking. But, believe it or not, they might not hold up to continuous game use. One way to find out.

—[11]—

We've come a long way in microcomputer **text compression,** but there are still lots of unexplored possibilities. It is theoretically possible to stash characters in 25 percent of their usual one-character-per-byte space for normal text. But adventure text is much more repetitive and redundant than typical text. Can you design a code that *match filters* the actual text and beat the theoretical limit? The benefits include getting much more text into the machine and possibly doing away entirely with repeated disk access.

—[12]—

Word processing is now old hat. But how about using a microcomputer for **picture processing?** I mean much more than the simple "business graph drawing" software. What you need here is high level software that lets you create *any* picture of your choice with a few dozen to a few hundred keystrokes. For instance, one module could take a single pattern and make it any size or boldness. A second module could convert this single pattern into an array of identical patterns so many high by so many wide. Another module could automatically label this array. Still another could fatten lines into, say, printed-circuit traces. Yet another module would build up final code, and so on. Heavy.

—[13]—

There are lots of possible micro uses involving **hot tubs and spas.** The mechanical temperature controllers are very crude. They overshoot, and if you turn the tubs on too early, your heating costs skyrocket. Why not use a micro to hold the temperature within a quarter of a degree by proportional heat control. When needed, it would turn things on just soon enough to get up to the proper temperature for use? And how about a solar booster? The savings on several month's heating bills should pay for the electronics and then some.

—[14]—

The chunky, illegible, and abrupt scrolling on most of today's video screens has got to go. How about retrofit **soft scrolling** or gentle scrollers that let text roll up a screen quickly and smoothly. Or, for that matter, roll down, right, or left, equally quickly and smoothly.

—[15]—

There is a sneaky way to **expand the address space on an 8-bit microprocssor** way beyond 64K *without* the usual bank switching limits. What you do is key on certain op codes to select which part of memory you are using. For instance, you can have 64K worth of files that are read with special or obscure commands, and another 64K worth of everything else that is read in the usual way. You can extend this "keying on op code" idea to lots of exciting new uses.

—[16]—

What the world really needs is a **microprocessor-controlled party doll.** But watch the product liability problems on this one very carefully. Software bugs could be painful.

—[17]—

One very useful instrument for energy conservation is a **heat flux** meter. Putting one of these against a surface will tell you the heat flow through the surface, letting you find R values directly and pinpointing heat loss or gain areas. Can this be done cheaply but with enough accuracy and resolution to be useful? How?

—[18]—

An obvious limitation of most of today's video games is that they are just that—video. You have to watch the screen. How about a **total involvement microcomputer game** that puts you inside a moveable housing with many video screens, moving objects, and whatever else is needed to create a total sensation of being there. Sort of a flight simulator, only much more. Possibly extend this to involve two or more people at once.

—[19]—

Can a **microcomputer-based speech recognizer** be built that can tell the species of a bird by its call? Can this be made portable enough and cheap enough to be useful to ornithologists and serious amateurs?

—[20]—

Computer **touch screens** are still outrageously priced. It is absolutely inexcusable that they should cost more than five dollars, computer store retail. Come up with a transparent switching network that is nothing but two conducting plastic sheets that stick onto the video screen and intercept the game port of a keyboard connector. But watch the effects of static electricity. Big color sets are super nasty this way.

—[21]—

What are the **alternatives to QWERTY?** Today's typing keyboard is a hundred years out of date and was specifically designed to *slow typists down.* Other arrangements such as the Dvorak keyboard have been proved to be much faster, much more accurate, demanding of much less effort, much more easily learned, and much more fun to use. But even Dvorak may not be optimum. How about a "sculpted" keyboard that optimizes keystrokes to the individual? Maybe it could even be available in different sizes like gloves.

—[22]—

Liquid crystal displays are finally getting up to decent sizes. Can you take several 8-line-by-40-character modules and come up with a

retrofit flat panel display for existing micros? Call it the VIDLID and throw away your monitor.

—[23]—

How far are we from a direct **micro-to-brain interface?** Will this be read only? Write only? Both? Can it work from a distance? How far?

—[24]—

We have generic medicine and generic groceries, so how about **generic software?** What we want here is a meat-and-potatoes word processor, a no-frills spreadsheet, a communications module, a data-base manager, and picture drawers, all retailing for $9.99, without any fancy advertising, packaging, or blatent overpromotion. Unlocked, naturally. Authors would make bunches this way, since you can sell more than ten times as many programs if they are one tenth the usual selling price. Bootleg copies would be virtually nonexistent since they wouldn't be worth the time and effort. Supply and demand and all that.

—[25]—

Disk access on many microcomputer systems remains very slow. What can be done in the way of **super fast disk access** on your microcomputer? On some systems, you can change the sectoring order for faster access. On others, you can replace general disk access software or firmware with specific code that can be much faster. Text file speedups are especially needed. What are the limits? Which systems allow this?

—[26]—

I would like to see a **writer's personal computer** with some obvious features not yet available. It must weigh less than five pounds, be only as big as needed to support a full size keyboard, and have a built-in but removable full double-page display of at least 70 rows of 140 characters. It must run off four penlight cells for a month in total darkness and forever in daylight via built-in recharging photocells. It must be waterproof and virtually indestructable. Words and both stroke and X-Y graphics must be available under a common program. The non-volatile built-in RAM must hold at least two books, so a megabyte or better is essential. A fast modem interface to download to the home system is essential. And, of course, it should cost under $2,000. Internal hard copy and plotting would also be nice.

—[27]—

What is the **cheapest time** you can add to a micro? Can you take one of those $4 stick-on LCD clocks, run it into any old micro port, and come up with time displays for under ten dollars?

—[28]—

While we are on the subject of time, why not go the other way and do a **real, real-time clock.** By "real" real-time I mean National Bureau Of Standards time. In short, not just *a* time, but *the* time. Use a WWVB time receiver and a clock card, along with some controlling software. Never needs resetting and always is accurate.

—[29]—

What kind of new applications can you think up that would **combine microcomputers with load cells or strain gauges?** Weighing scales are one obvious use. What else can you think of?

—[30]—

It's high time to move **phototypesetting** out of the dark ages. Those totally ridiculous machines with their obscene pricing and their total incompatibility have got to go. How about a cheap box that bolts onto any old personal computer that generates camera-ready text for under $200, along with a completely free public domain font library? And how about a service that goes with it, typesetting, for, say, 50 cents a foot?

—[31]—

There are a number of **ultrasonic ranging** systems out there. *Polaroid* is one overpriced source. What can you do by linking a rangefinder to a microcomputer that can accurately measure fairly short distances? Handicapped aids and robotics are two obvious uses, but what else can you come up with?

—[32]—

Biologists and naturalists would like to have a sanely priced **wilderness data acquisition system.** This would be a very small box that you bury someplace to record a month's worth of weather, stream height, pollen count, or heaven knows what on its internal nonvolatile RAM. Must be small enough to backpack and priced very low, since these people are usually on a non-existent budget. Also the gathered data must upload into any popular personal computer.

—[33]—

What new micro uses can you dream up that involve **home security?** One novel idea is to make a speech synthesizer sound like a

barking German Shepherd that starts up a few seconds after the doorbell is hit. Very convincing. Particularly if you use this to wake up the real dog.

—[34]—

Can you now simplify **hobbyist printed circuit work** by plotting directly onto artwork without all those expensive tape and dot stick-ons? For that matter, can you directly plot 1:1 resist onto the PC board itself? How cheaply can a complete computerized PC layout system be of an built up? Among the other benefits, all symbols would be accurately sized and accurately positioned without anything slipping.

—[35]—

There's lots more work to be done in **interfacing personal computers to videodisks and videotape.** We are still some distance from a widely available, cheap, and easy to use fully interactive video system under total control by a local personal computer.

—[36]—

How can microcomputers be applied to simplify and improve **cave mapping?**

—[37]—

Just around the corner is the **multi-player real-time video game.** One way to do this is to connect two personal computers, each with a display that shows the game condition only from that player's point of view. Aerial dogfight simulations are one obvious first possibility. *Kriegspeil,* or "blind man's chess" is a second. But, once you get into using a separate second computer, all sorts of new uses open up.

—[38]—

One of the new things to hit video graphics is called **run length encoding.** In run length encoding, you tell the display a color and how many successive pixels (picture elements) of that color are to be put down. The big advantage of this is that far fewer bytes are needed to put down a picture, particularly one that has large areas of one color. You can also combine lots of colors with very high resolution without using bunches of bytes. So far, the idea is used mostly on larger minicomputers and specialized studio tv systems. Now's the time to put this one to use on microcomputers.

—[39]—

We are only starting to see interesting **alternate micro input devices.** Today, trackballs and "mice" are grossly overpriced and not very widely available. Can you dramatically lower the cost of these? What other ways can you come up with that let you enter non-text commands into your micro quickly, naturally, and easily?

—[40]—

What can you do with a **microcomputer on a bicycle?** Obvious uses are measuring speed, cadence, average rate, trip time, wind speed and direction, percentage grade, completion time, and so on. But could you get really fancy and tie a micro into a continuously variable transmission that would optimize your effort for the speed you want to travel?

—[41]—

One interesting piece of hardware would be a **retrofit non-volatile RAM.** You would simply take all the old RAM out of your micro and replace each chip with a module that faked permanent memory. One good way is to use CMOS memory and small rechargable cells. The key here is to make no hardware changes except for swapping chips for chip-like modules.

—[42]—

What happens if you bolt a digitizing sight onto the carriage of a dot matrix or daisy-wheel printer? This gives you a **cheap facsimile scanner** that gives you a fast and easy way to send text or pictures over the phone line. The sight picks off the white and black areas on the page and converts them into a series of bytes to be sent to a second, remote printer.

—[43]—

Can a microprocessor be used to build a really decent **FM car radio?** Something that really pulls in the stations when you are in weak signal areas? Maybe include a steerable antenna, automatic antenna matching, variable sensitivity and bandwidth, a scanner that filters for certain types of programs, and full overload protection.

—[44]—

Can a **voice-recognizing modem circuit** be built that would prevent a computer from beeping into someone's ear at 3AM because it got the wrong number? Can the same circuit tell busy signals and dial tones from ring signals? What you want is something that keeps trying until it gets a legal connection without disturbing any people who inadvertently answer.

—[45]—

There is a unique branch of mathematics that involves **fractials.** Fractials are a way of defining randomness that pretty much matches how things turn out in nature. For instance, an entire mountain range can be defined using only a very few fractial terms. Most fractial applications so far have been done on dino machines. What micro uses for fractials can you find?

—[46]—

Can you fake **full-color hard copy** by using different color ribbons and repeat passes through a dot matrix printer or a daisy wheel? How good can you get? Where do you find a yellow ribbon?

—[47]—

Micros can help in building **speed controls for AC induction motors.** These are available, but they aren't nearly as cheap or as flexible as they should be. Unlike simple dimmers used on AC/DC motors, they require varying both the frequency and the current to control an AC only motor, sensing the load as you go.

—[48]—

What interesting things happen when you add your own "peripheral" to the **ignition and emissions computer on a car?** For openers, you should be able to get instant readouts of miles per gallon and long and short term fuel economy. More importantly, you should be able to optimize for economy, power, or minimum pollution.

—[49]—

A really cheap and effective **remote power interface** for micros doesn't yet exist, although lots of people are working on this one. What you want here is a long RS-232 connector and cable that goes to a box with lots of big power relays and/or triacs and a bunch of optocouplers for inputs. This lets you do very heavy control work with your micro without having to go inside or near the micro with noisy or dangerous loads.

—[50]—

I'm continually amazed that **pneumatic robotics** haven't yet taken off. Low pressure air is a far better, far cheaper, far stronger, and far more linear way of producing straight line motion than using solenoids or stepper motors. Your actuators can be balloons, small bellows, or rolling diaphragms, and you can easily deliver force over distances, around corners, or through a robotic elbow. Suitable automotive EGR valves cost 50 cents each, surplus, or a buck at the junkyard; and an aquarium pump is all you need for an air supply.

—[51]—

While a few rather poor programs are available involving **microcomputers and the hand weaver,** there are lots of opportunities to do the job right. Most of the existing programs concentrate on *synthesis,* or going from drawing-down to the final pattern. Equally important is going the other way and doing *analysis,* starting with an input pattern and finding the threading, tie-up, and treadling needed for that pattern. No complete weaver design package exists at this writing. And nobody yet has added small LED lamps to show the next pattern needed, directly on the loom.

—[52]—

What is the absolute minimum hardware needed to **convert a personal computer into an oscilloscope?** There's lots of expensive stuff out there, but what do you really need? Important advantages of a personal computer over a traditional scope include permanent or long-term storage, the ability to compare and leisurely measure waveforms at high accuracy, averaging, least-squaring, filtering, variable persistence, multiple inputs, and very cheap and easy hard copy.

—[53]—

While we are at it, how about a **cheap paper copier**? Combine a micro with a laser diode and a small scanner and shoot those expensive copy machines right out of the water. Aim for a $99 final parts cost. But stick with black and white for now. Full color with high resolution under $99 might upset too many people too fast. Culture shock and all that.

—[54]—

Now that the prices of **steppers and linear stepping actuators** are dropping down to almost reasonable levels, what new and exciting uses can you think of for micros and incremental motion?

—[55]—

Practically all microprocessors have illegal or unused op codes. How can you **replace "illegal" micro op codes with your own new and useful functions?** The concept is called *microprogramming,* and it's done all the time in fancy bit slice micros. This lets you make the micro do things that it couldn't do before, picking up new and useful instructions along the way.

—[56]—

There are all sorts of exciting new uses for **low cost shaft encoders,** ranging from robotics through hand-held "mice" and other new

input devices to improved hard copy and plotters. Can you build something simple, cheap, brand new, and very useful?

—[57]—

What happens when you use a beam splitter to **combine a computer screen with a slide projector?** This gives you a way to mix computer output with very fancy pictures and graphics, and it is a super interactive computer aided instruction setup as well. Cost is far lower than going to fancy videodisks or whatever, and anybody can lash this sort of thing up simply and quickly. Any takers?

—[58]—

Can you take a few stepper motors and a micro or two and build a **simple programmable animation stand** for cartoon and other video and/or motion picture uses? Real stands cost zillions of dollars. Think scungy.

—[59]—

Take a personal computer and remove the microprocessor chip. Replace that CPU chip with a connector that goes to a faster **emulation microcomputer,** so that you end up using a faster computer to control a slower computer. What you gain is total and absolute control. You can stop any program at any point. Find out what comes off the disk when, and find out where it goes in memory. Move and save memory images anywhere you want. Change things on the fly. And much more.

—[60]—

There's been bunches of work done on **3-D computer graphics** but no clear winners have appeared so far. Four popular approaches are to use a vibrating mirror on a woofer, to split the screen and route half to each eye, to use special switching glasses that match one field to the left eye and the other field to the right eye, and to use the usual red-and-green comic book glasses. Try some of these and see if you can come up with something newer and better.

—[61]—

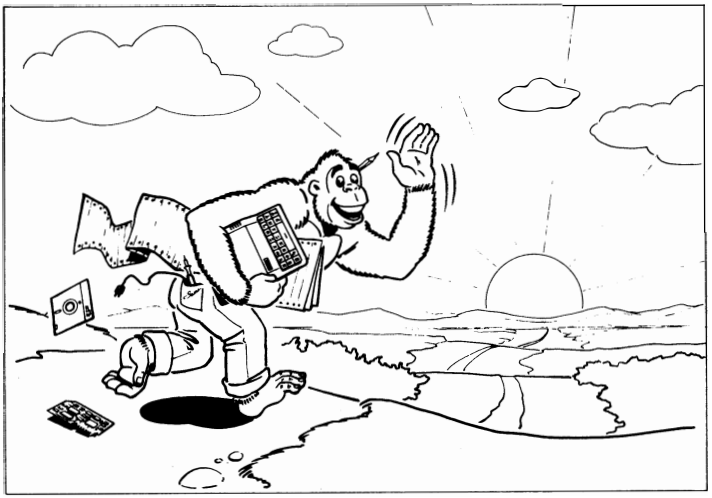
Many computer graphics applications need **fast multiplications and trig calculations.** One way to do this is with special add-on chips, but for many uses even this is too slow. Can you use plain old table lookup out of files to do fast multiplications and trig calculations that are just barely accurate enough for the results you need? This approach should be much faster and much cheaper than any other route, yet it has not seen very much micro use so far.

—[62]—

Lots of opportunities exist in **cable television** for new uses of microcomputers. Two-way communication and subscriber controls are only two of many possible uses. Can UARTs help out here? What is possible? Keep down the costs of anything that goes on the subscriber end while you do this.

—[63]—

Each aware home and business should have current-sensing transformers on all incoming power lines connected to a micro-controlled **power-usage monitor.** This would give you an instantaneous readout of how much energy you are using along with your projected total monthly bills. The power saved in conservation should quickly pay for the system.



PLAYING FOR THOSE FIVE

At a recent rock concert, the opening act was a single flute player, performing in front of the closed stage curtains. His job was to warm up the audience for the high priced help to follow.

He was good. Very good.

But as he went along, the music started getting strange and finally downright weird. He was playing *chords* on his flute, along with notes with unbelievably strong tonal structures. Eventually, the music turned into bunches of impossible sounding, god-awful squawks.

Almost all of the audience got bored and restless as the music seemed to deteriorate. Just then, I happened to notice a friend beside me who had played in and had taught concert band. He was on the edge of his chair with his mouth open.

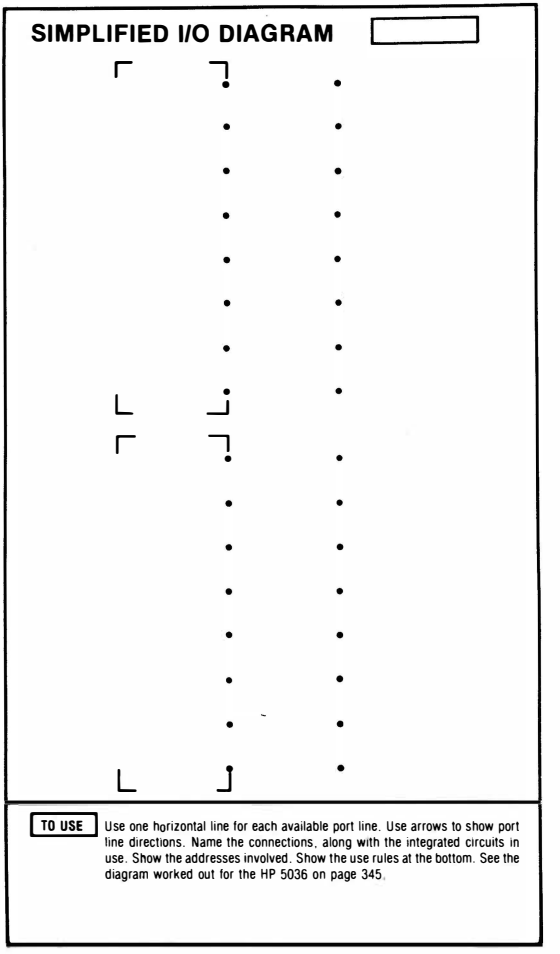
He turned to me and said very slowly, "You can't do that with a flute."

Of the thousands of people in the audience, at most only five realized they were witnessing a once-in-a-lifetime performance involving the absolute mastery of a very difficult musical instrument. To nearly everyone else, it just sounded like a bunch of god-awful squawks.

Always play for those five.

APPENDIX

Here's that blank form you can rip off when you do your own simplified I/O diagram (see page 345) . . .



Index

C

D

E

S

T

U

V

W

X

Previous chapters 6 and 7

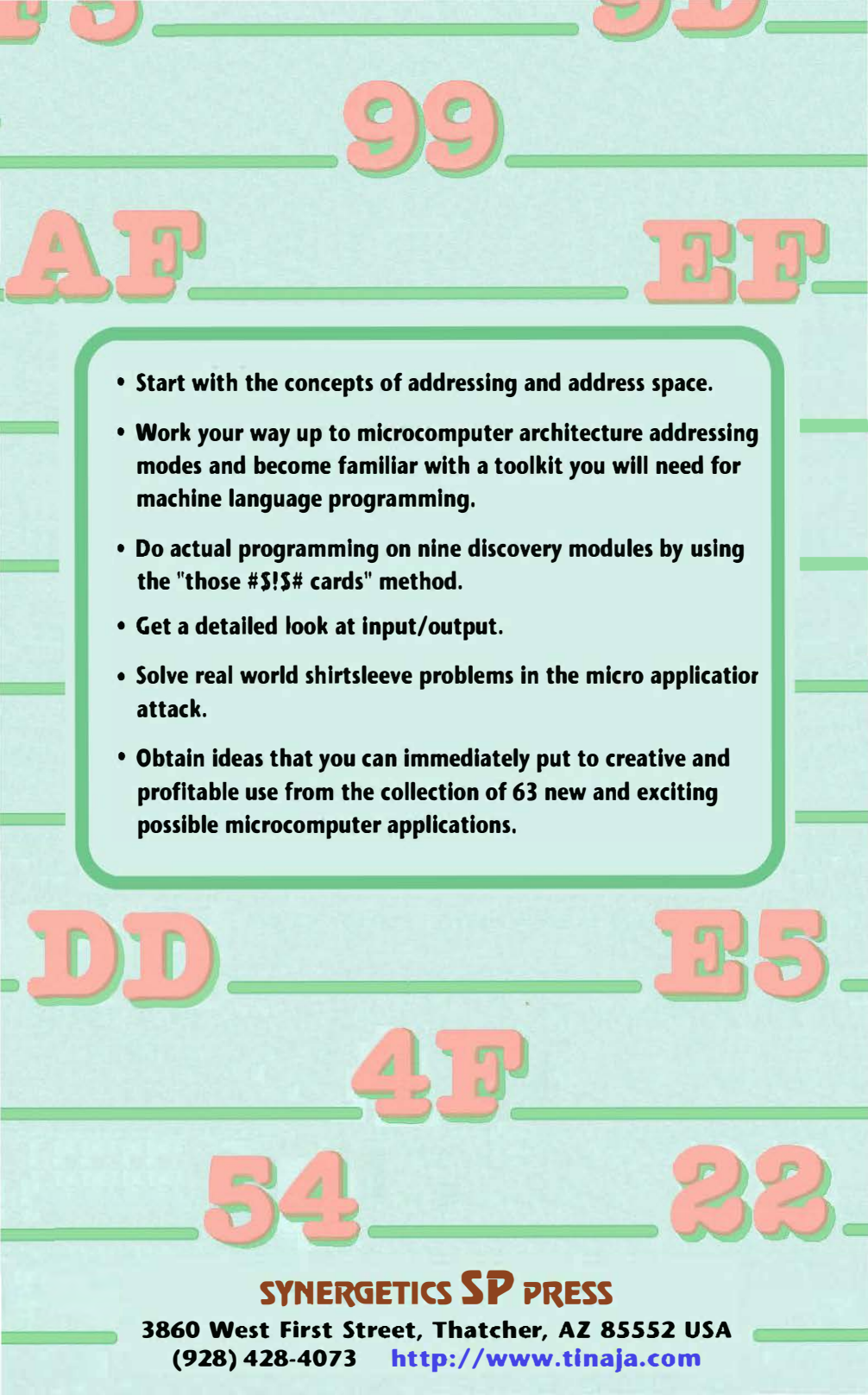
of this eBook may be found at

http://www.tinaja.com/eBooks/MLP1cb.PDF

SYNERGETICS SP PRESS

3860 West First Street, Thatcher, AZ 85552 USA

(928) 428-4073 http://www.tinaja.com



- Start with the concepts of addressing and address space.
- Work your way up to microcomputer architecture addressing modes and become familiar with a toolkit you will need for machine language programming.
- Do actual programming on nine discovery modules by using the "those #$!$# cards" method.
- Get a detailed look at input/output.
- Solve real world shirtsleeve problems in the micro applicatior attack.
- Obtain ideas that you can immediately put to creative and profitable use from the collection of 63 new and exciting possible microcomputer applications.

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com