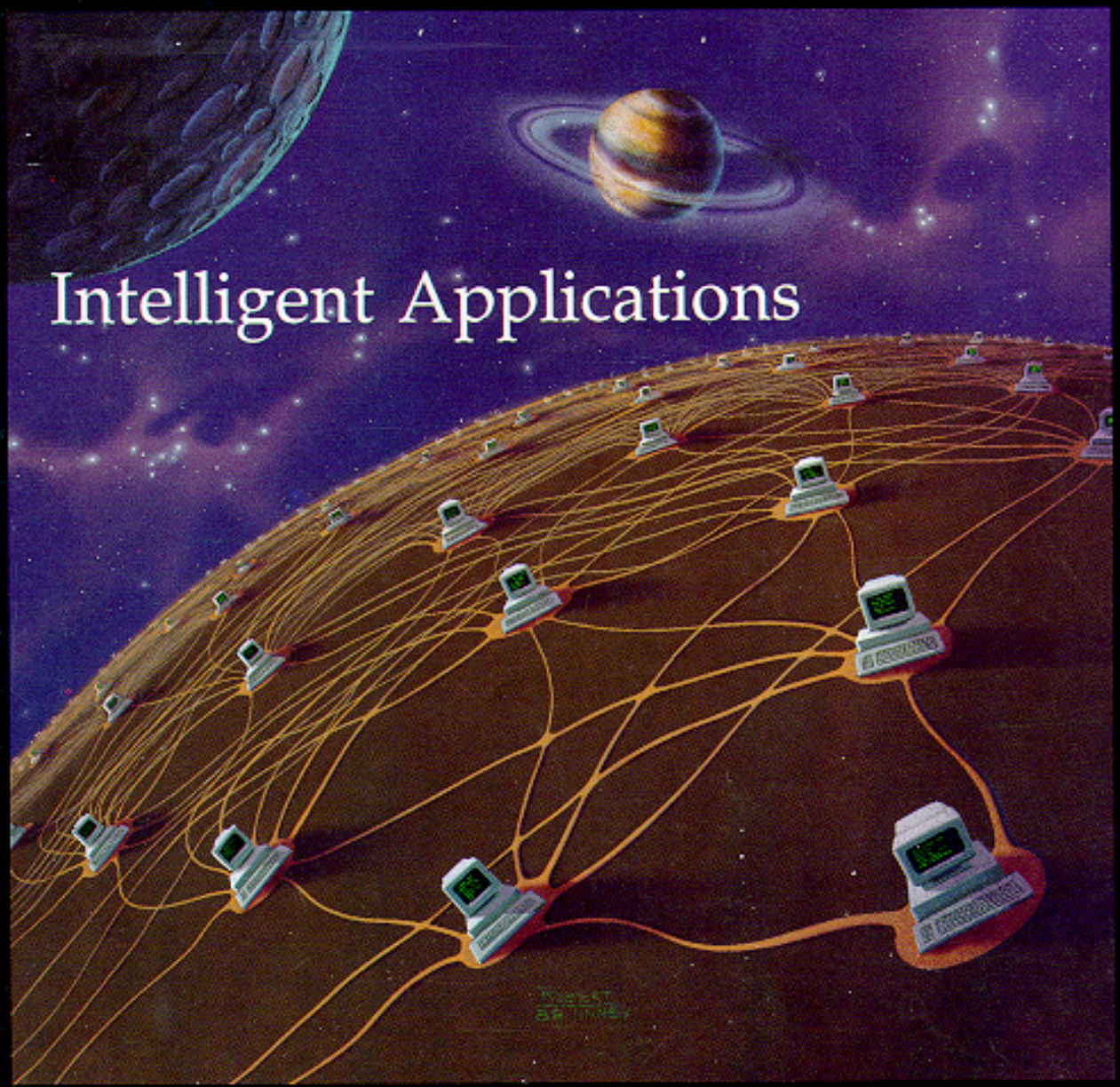
Circuit Cellar INK®

THE COMPUTER APPLICATIONS JOURNAL



Intelligent Applications

June/July 1989 — Issue 9

$3.95

Ctrl

EDITOR's INK

Working Smarter, Not *Faster*

It's time we admitted to a character flaw common among engineers. Now, I'm not talking about the kind of problem that leads to major stories in the newspapers and evening newscasts. No, this problem is insidious, working its way into the applications we design and the products we design. What is this terrible problem, this scourge of the honorable profession of engineering?

Horsepower Lust.

Yes, whether you're an automotive engineer trying to shoehorn a V-12 into an econobox, a mechanical engineer still searching for the fulcrum and lever to move the earth, or a civil engineer reflecting on a dam for the Strait of Gibraltar, there's a general search for more and more power to use in bigger and bigger projects. Computer engineers are as much afflicted as any. I've never met an application designer who wasn't looking for more MIPS, MOPS, FLOPS, or other unit of power. The problem from all of these manifestations of Horsepower Lust, is that the afflicted tend to try to substitute raw horsepower for intelligent design.

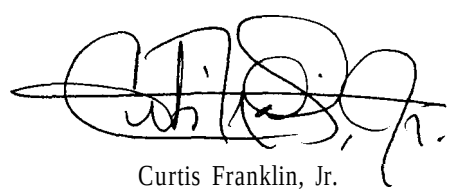
The computer and controller industry has been particularly offensive in this respect, largely through the heroic efforts of microprocessor and microcontroller designers. If you want to do something that simply soaks up too many clock cycles to be reasonable, just wait six months for a faster processor. Hardware speed improvements have stayed ahead of applications needs for the last 15 years or so, letting application designers and programmers "shoot for the moon" and get away with it. Now, for better or for worse, we're running out of silicon's ability to give us more horsepower. (OK, I know about GaAs and optical computers and other new technologies, but they only postpone the problem.) It looks like we're all going to be forced to (gasp!) look for new ways to solve problems, search for new ways to save cycles, and find new applications for old control and computing solutions. That's where CIRCUIT CELLAR INK comes in.

In this issue on Intelligent Applications, we look at intelligence from a couple of different prospectives. The first is the quest to make an electronic analog of the human brain. Neural networks hold much promise for grand design in the future, but they're also useful for problem-solving today. We show you two different approaches to neural networks,and hope that you will experiment with what you see. The second intelligence perspective is that of offloading processing to outboard devices. I think it makes sense: If the applications are getting too complex for a single central processor, start feeding the processor predigested information. It works for baby birds, why not for data acquisition systems? In short, what we're talking about is substituting designintelligence for raw horsepower. It's a trade that can get you some surprising performance wins, and let you bring in cost-effective solutions when others are offering gold-plated bells and whistles. (Of course, if you can combine lots of horsepower with an intelligent design, then you're way ahead of the game, aren't you?)

Odds and Ends

You've probably noticed that there's been an interruption in the Home Satellite Weather Center series. We're not abandoning those who have been following the series: Mark has just taken some extra time to get everything in order for the conclusion of the project. The Weather Center should be back soon, getting wound up for a big finish.

In the next issue, CIRCUIT CELLAR INK will focus on controlling one of the most complex environments around: The Home. The center of the theme will be built around networking. In addition, there will be articles on control of and by telephone, as well as a fascinating project for receiving television broadcasts from the Soviet Union. Think of it as a technical response to the promise of Glasnost.

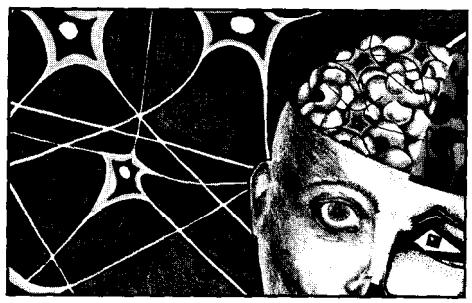


Curtis Franklin, Jr.
Editor-in-Chief

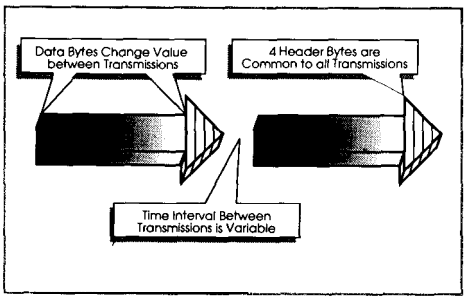
CIRCUIT CELLAR INK®

Cover Illustration
by Robert Tinney

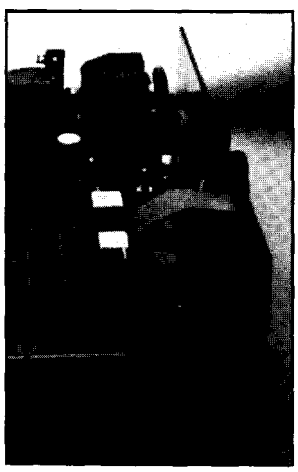
FEATURES



DEPARTMENTS



Most neural network research involves large budgets and even larger mainframe computers. But artificial neurons can be harnessed for useful work on a modest desk-top computer, Handling "fuzzy" data is not a problem limited to the laboratory: a simple neuron on a common microcomputer can bring irregular data into useable focus.



The SCSI Bus offers important performance benefits compared to standard RS-232 links. Properly implementing the protocol requires an intelligent peripheral, but the intelligence and performance can bring impressive application results. The first of two parts looks at the hardware for an intelligent 12-bit A/D converter. An Apple Macintosh is the host computer for this *application of distributed intelligence.*

Circuit Cellar BBS-24 Hrs. 300/1200/2400 bps, 8bits, no parity, 1 stop bit, (203) 871-1988.

The schematics provided in Circuit Cellar INK are drawn using Schema from Omation Inc. All programs and schematics in Circuit Cellar INK have been carefully reviewed to ensure that their performance is in accordance with the specifications described, and programs are posted on the Circuit Cellar BBS for electronic transfer by subscribers.

Circuit Cellar INK makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors. Furthermore, because of the possible variation in the quality and condition of materials and workmanship of reader-assembled projects, Circuit Cellar INK disclaims any responsiblity for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar INK.

CIRCUIT CELLAR INK (ISSN 0896-8985) is published bimonthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (203) 875-2751. Second-class postage paid at Vernon, CT and additional offices. One-year (6 issues) subscription rate U.S.A. and possessions $14.95, Canada $17.95, all other countries $26.95. All subscription orders payable in U.S. funds only, via international postal money order or check drawn on U.S. bank. Direct subscription orders to Circuit Cellar INK. Subscriptions, P.O. Box 2099, Mahopac, NY 10541 or call (203) 875-2 199.

POSTMASTER: Please send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 2099. Mahopac, NY 10541.

cm

READER'S INK

Letters to the Editor

I have just finished reading the January/February issue of CIRCUIT CELLAR INK. It reminded me why, after quitting the computer business several times, I am still so attracted to silicon. CIRCUIT CELLAR INK has rekindled that wide-eyed feeling of awe and power I felt after constructing my first 2-bit adder in college.

After seeing a letter from Dr. Huong of Computer Age Ltd. in one of last year's issues, I went to see him; something I hadn't done in five years! Dr. Huong showed me a voice card he built and I purchased it as a companion to the ImageWise transmitter I own. Now my IBM PC can see, hear, and talk. Dr. Huong was talking about some of the data compression techniques used by Toshiba in the voice chip he uses. I wondered if similar techniques could be used on a .PIC file. After some preliminary investigation it seems that 64- or 256-color files can always be compressed from 63K to about 8K! The conversion process is not yet perfect but under certain circumstances could be useful. I can send you a copy of the compress and uncompress program I am developing if you would like a first-hand look. I would welcome the opportunity to do an article for CIRCUIT CELLAR INK readers. High-resolution pictures could move over ISDN line at one frame per second or faster.

With more sound and images being digitized, and the quality of the renditions improving, I am starting to worry about the opportunity this presents for distortions of the truth. I can't help but wonder when the first news clips could be generated entirely from bits. How do other CIRCUIT CELLAR INK readers feel about the technologies that are being developed?

Finally, **have you** considered doing case-study reports on interesting hardware/software systems? These wouldn't be reviews, they would be **done** by someone who used an interesting system to solve interesting problems. As an example, those new neural network programs--can they be used to keep track of all the facts I forget as the years pass?

Steve Carter
Miss., Ontario

If you're using a new technique to solve a problem, you bet we'd like to hear from you, *Many* of our *articles are written by working engineers or designers who write in to* describe their *latest project. If you would like an Author's Guide, send your request to:*

Circuit Cellar INK
Author's Guide
4 Park St.
Vernon, CT 06066

The ethical considerations of *engineering are many and varied. If anyone else has an answer (or maybe just an opinion) on the subject, let us know.*

When you mention "case *study," you come vey close to a detailed review.* We *haven't had any reviews so* far *(the "April Fool" article notwithstanding), and we're not sure that we need to. Affer all, there are many other magazines fhat* focus *on reviews. It boils down to this: if we feel that we can offer CIRCUIT CELLAR INK readers information fhat they* need *and can't get anywhere else, we'll do if. Until fhen, we'll stick with the technical information and leave the reviews to others. Editor.*

In Circuit Cellar BBS On Disk Issue 5 you indicated that you would like suggestions for topics and content. Here are mine...

Reviews-None of any kind.

My preference is for construction project articles or hardware/software combined application information. I feel that the software source for projects should be made available so that we can modify it for our own uses.

Iamnot interested inany IBM PC-related applications except (perhaps) for development purposes. I would rather see dedicated stand-alone microprocessor applications.

Please try to maintain the continuity of articles by keeping them together and not continuing the last part on back pages. It would also be nice if all advertising could be

put **between articles or on the back pages, not put in the middle of articles.**

Deslar Kyn Patten
Hayward, CA

Thanks for the input: we really do *depend on readers to let us now how we're* doing. We *started the IRS at the end of each article to make talking to us easier. Fill out the post-paid card, drop it* in *the mail, and you've told us what you did and didn't like about theissue. Ofcourse,westill like theletters wegetfrom readers; we just know how busy most of you are.*

We try for a balance of articles in CIRCUIT CELLAR INK. You'll see hardware, software, stand-alone, and host-based articles in our pages. We hope that each one will have something engineers can use, even if they don't do development for the particular system used in the article.

Running the beginningand end of anarticlein two different parts of fhe magazine is known in the publishing industry as "jumping." CIRCUIT CELLAR INK has not used jumps, and there aren't any in our future *plans. As for advertising, we feel that CIRCUIT CELLAR INK Advertisers provide a service to OUT readers. Having a mix of advertising and editorial material in our pages makes for a better magazine for everyone. Editor.*

Ctrl

VISIBLE I N K Letters *to the INK Research Staff*

Answers; Clear and Simple

A Little Misdirection

We are users of an Epson Equity III+ microcomputer. We run under MS-DOS 3.20 and do most of our programming with RM-COBOL 1.5D. We have two Epson printers attached to our system via parallel ports.

The interface between major and minor screens of our menu-driven program consists of CALLs to subprograms. The main program is called by typing the name of a batch file from the DOS prompt. The problem is this: We need to address either of the two printers from within the subprograms depending on what type of report we need to print. Our compiler doesn't provide a facility to do this, and we can't use the MODE command since the choice isn't made at the DOS prompt level. We have resorted to the following unsatisfactory method of printing:

```
fa.bat
   echo off
   mainprog
   MODE lpt1:=lpt2
   LISTING
   MODE lpt2:=lpt1
   LISTING
```

(`mainprog` is the main COBOL program that displays the main screen, CALLs the subprograms, and creates the LISTING.BAT file. LISTING.BAT contains the invocation of the printing program.)

This is unsatisfactory because the system hangs on the first MODE command. Please help us find a simple way to make this work.

Ramon Gonzales **Barroeta**
Caracas, Venezuela

This is a fairly common problem of *swapping printer ports. The simplest programming to do the swap uses BASIC. This program* simply *swaps the contents* of *the memory locations* in *DOS where the two parallel ports are referenced.* If *you run it once, it sends LPT1 output to LPT2. Run it again and everything is back to normal.* If you *compile this with QuickBASIC, you should be able to call if from within another compiled program.*

```
DEF SEG = &H40
TEMP1 = PEEK (8)  : TEMP2 = PEEK (9)
POKE  8, PEEK  (10): POKE 9, PEEK (11)
POKE 10, TEMP1: POKE 11, TEMP2
SYSTEM
```

The only problem with this is that if compiles into a relatively large .EXE file. If you want *compact, though,* you'll have *to dig out your assembler and do your own research!*

Just Give Me Time

I was one of the early buyers of the SB180 single-board computer("Ciarcia'sCircuitCellar,"Sept/Oct 1985BYTE). I've since added a number of upgrades, including the COMM180 SCSI board with XBIOS and ST225N.

The only thing my system lacks is a permanent clock. The easiest way to add one is through a SmartWatch under the boot ROM. The problem is, the SCSI board uses a lot of real estate for a modem chip which I don't need, and the board leaves insufficient space over the boot ROM for me to install the SmartWatch. Just to complicate matters, I built the system into a very small case. I only have room for one expansion board.

My way out was to buy some SCSI chips and build my own board. I used the 53C80 since heat was a problem in the small case. I read a number of relevant articles and used the circuit shown. I needed the lowest chip count to reduce both the board size and the number of connections. It almost worked!

Running with XBIOS the machine hangs in the boot process, but with the SB180 3.1 BIOS I get a more useful "phase error" message. I can't decipher it, however, since I have no hardware debugging tools except for a logic probe and DVM. I have traced to be sure that the constructed hardware is identical to the schematic. Can you spot some problem with my circuit?

I'd be grateful for any help. This is a trivial design, but when you build without proper debugging tools you have

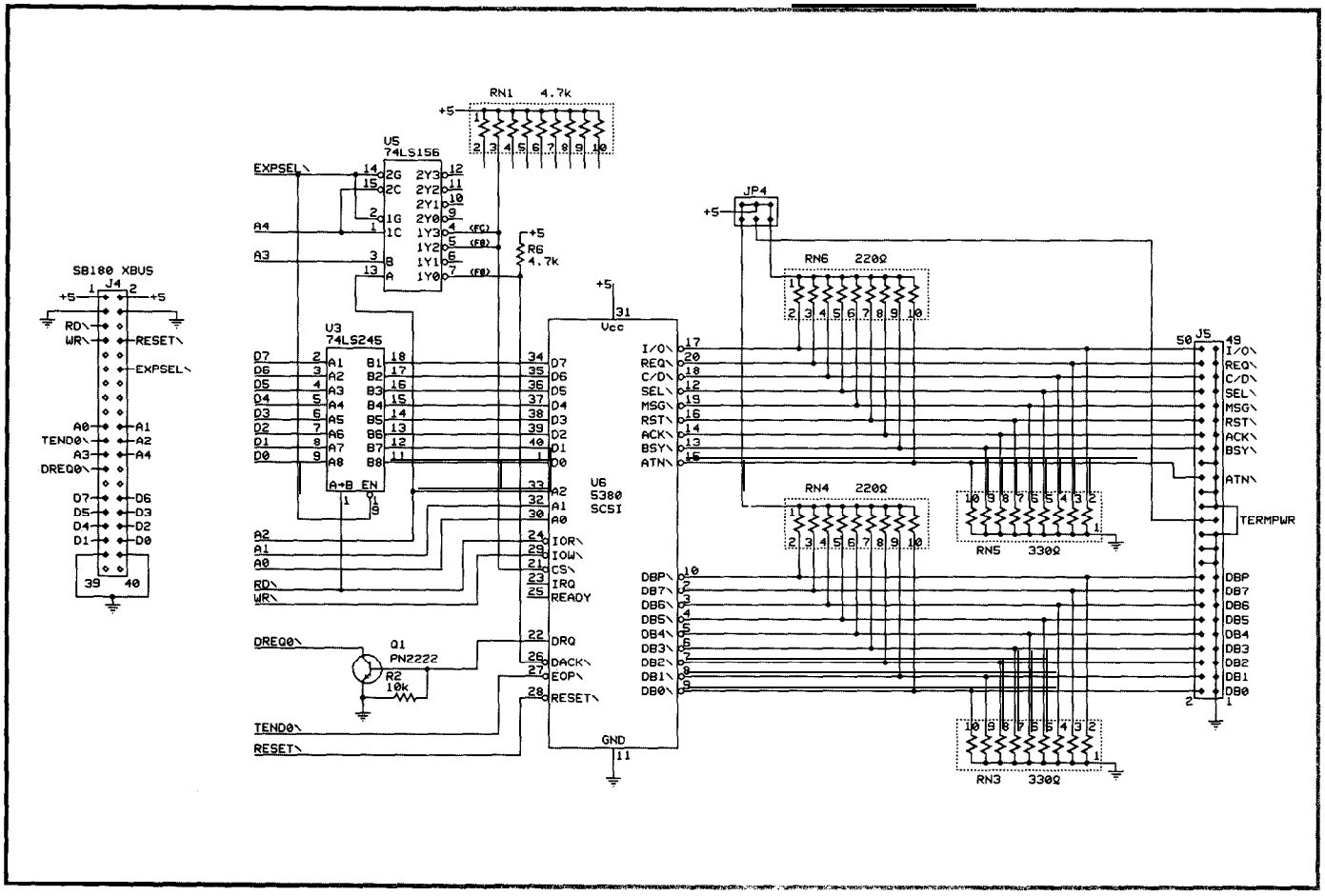


Figure *1 —The SCSI portion of the COMM 180 uses the NCR5380 to interface to the SCSI bus.*

no **way of telling** if the fault is in concept or execution-an original design that is known to work is very comforting.

C.W. Rose
San Diego, CA

P.S. The 220/330-ohm resistors on the SCSI bus consume an awful lot of power. Can they be increased to 440/660 or even 2200/3300?

You *are correct that a Smart Watch is the best way to add a real-time clock to an SB180 system. Thepreferred way to provide* clearance is to extend the SCSI board above the main board *with an extension connector, but that obviously won't work if clearance is too tight.*

Your schematic looks fine. There isn't a lof involved in connecting the 53C80 to the *SB180 XBUS. It might simplify thingsfor* **you** *to know that neither interrupts nor DMA are used* in *current* software, so you *can* get *away with* leaving *off* the *connections to EOP\, DRQ, READY, and IRQ\ on the 53C80. (EOP\ should be pulled up; just connect it to DACK\.)*

The phase error *you're receiving generally points to a problem with the processor talking to fhe SCSI chip, so it's not terribly helpful by itself. A number of very useful tests could be performed with an oscilloscopeand the SB180's monitor, but we won't go into solutions involving equipment you don't have. We'll just suggest checking your wiring against the schematic one more time and makingsure that you've properly interpreted the pin numbering on the XBUS connector itself.*

As for the terminating resistors, fhe ANSI SCSI spec specifically calls for 330-ohm pull-up and 220-ohm pull-down termination on the bus. If you want your interface to agree with thespec, you'll have to leave them *as is. Only* if you *have a short run* of cable and if you **don't** *mind* possible *noise or echoes on the line and* if you don't care whether *your* bus matches the *spec should you play with the values* of *the resistors.*

IRS

201 Very Useful
202 Moderately Useful
203 Not Useful

In Visible INK, the Circuit Cellar Research Staff answers microcomputing questions from the readership. The representative questions are published each month as space permits. Send your inquiries to:

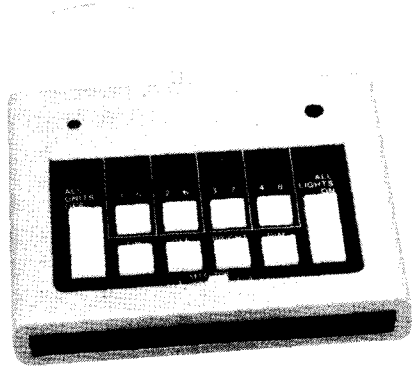
INK Research Staff
c/o Circuit Cellar INK
Box 772
Vernon, CT 06066

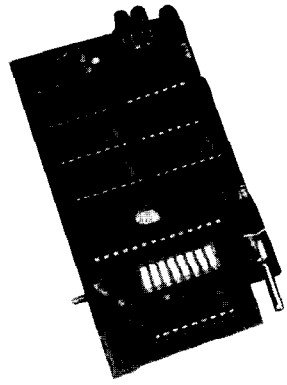
All letters and photos become the property of CCINK and cannot be returned.

The X-10 IR543 Infrared Gateway/Controller

Control your *lights* wifh your *trainable IR* remofe

by Ken Davidson





How does the television commercial go? "My wife left me, so I bought an expensive television set, and it came with a remote. Then my dog died, so I bought a VCR, and it came with a remote. Then I was transferred to Alaska, and it came with two remotes.. ."

Well, you get the picture. It seems every device even closely related to consumer electronics comes with a hand-held infrared remote control these days. **And once** you start putting together your entertainment center, you end up with a pile of incompatible, but necessary, remotes.

To combat the problem, a host of trainable controllers have been introduced to themarket. Indeed,Stevedid a Circuit Cellar article for the March '87 issue of BYTE which described his design of just such a trainable remote.

So now we can control the TV, VCR, cable box, CD player, and so on with a single IR remote. What about thelightsorotherappliances **we might** have plugged into X-10 modules? How can we remotely control those without using the hard-wired consoles?

When BSR first introduced their System X-10, there was a hand-held ultrasonicremoteavailableforit. Back in the late seventies, those TVs that had remotes usually used ultrasonic, so the choice was appropriate.

When System X-10 was taken over by X-10 (USA) Inc., the ultrasonic remote/base pair was discontinued and replaced by an RF system, the RC5000. The RT504 hand-held **remotecan** send out signals to theRR501 base unit to turn modules on or off and can dim or brighten lights. While you have all the advantages of RF (youcanusetheremotefromvirtually anywhere within your house), you still can't point the hand-held RF unit at your trainable IR remote and expect the IR unit to learn the signals. Add one more box to the remote pile.

A Solution Appears

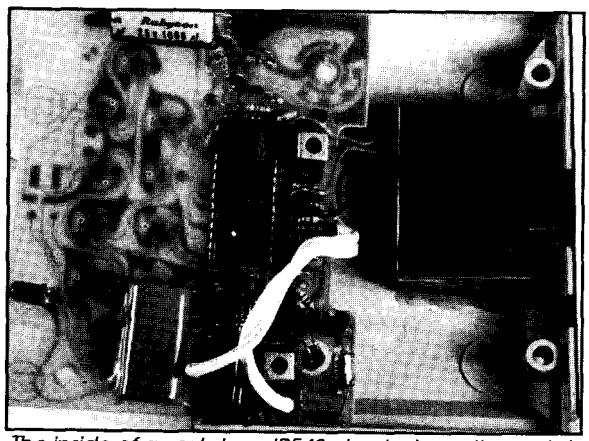
Thelatest addition to X-lo's product line goes a long way toward rectifying the situation. The IR543 is an infrared-to-X-10 gateway/controller which will receive X-10 commands from a hand-held IR remote control, tack on house code information, and broadcast them over the power line.

The schematic for the IR543 is shown in Figure 1. The box contains the usual requirements for an X-10 transmitter: power supply, zero-crossing detector, free-running 120-kHz

oscillator, and output drive circuitry. (For details about how the X-10 system works, see "Power-Line-Based Computer Control" in issue #3 of CIRCUIT CELLAR INK.)

The key element in the IR543, though, is the custom 78542C controller chip. The 78542C takes care of all the unit's operating details such as scanning a keyboard, translating a keycode into an X-10 bit sequence, tacking on housecode information, and sending the code onto the power line by watching the zero-crossing input and gating the 120-kHz signal onto the power line using the correct timing. In addition, the chip has a surreptitious "serial" pin for accepting serial input.

This serial input pin has been quietly overlooked for years. It turns out that the 78542C was used in the original BSR ultrasonic unit. The ultrasonic front end's output was sent to the chip's serial input so the signal could be rebroadcast over the power line. The end of production of the ultrasonic control unit didn't mean the end of the 78542C, however. The chip has been used for years as the basis for the SC503 maxi controller and, until recently, the MC260 mini controller. (The MC260 has eight small push buttons and two large buttons. The newer MC460 mini controller has six rocker switches and uses a different controller chip.) The serial input pin has simply been tied to ground in these controllers for all these years.



The inside of a prototype IR543 clearly shows the metal-encased IR receiver section.

It logically follows that the overlooked capability of the 78542C would one day be tapped again. With the addition of an IR front end in place of the old ultrasonic front end, we can extend the system's functionality with a remote control. The IR front end of the IR543, which is encased in a metal can for added noise immunity, is responsible for receiving the IR from the hand-held remote and translating it into a series of bits for the 78542C's serial input.

Hand-held Dilemma

So now we have a box that will receive IR commands and retransmit them onto the power line. How do the IR commands get to the box, though? Unlike the RF remote/base pair, X-10 doesn't make a low-cost, hand-held transmitter that is dedicated to the IR543. The IR543 was originally designed for a company who built the commands into their own "universal"

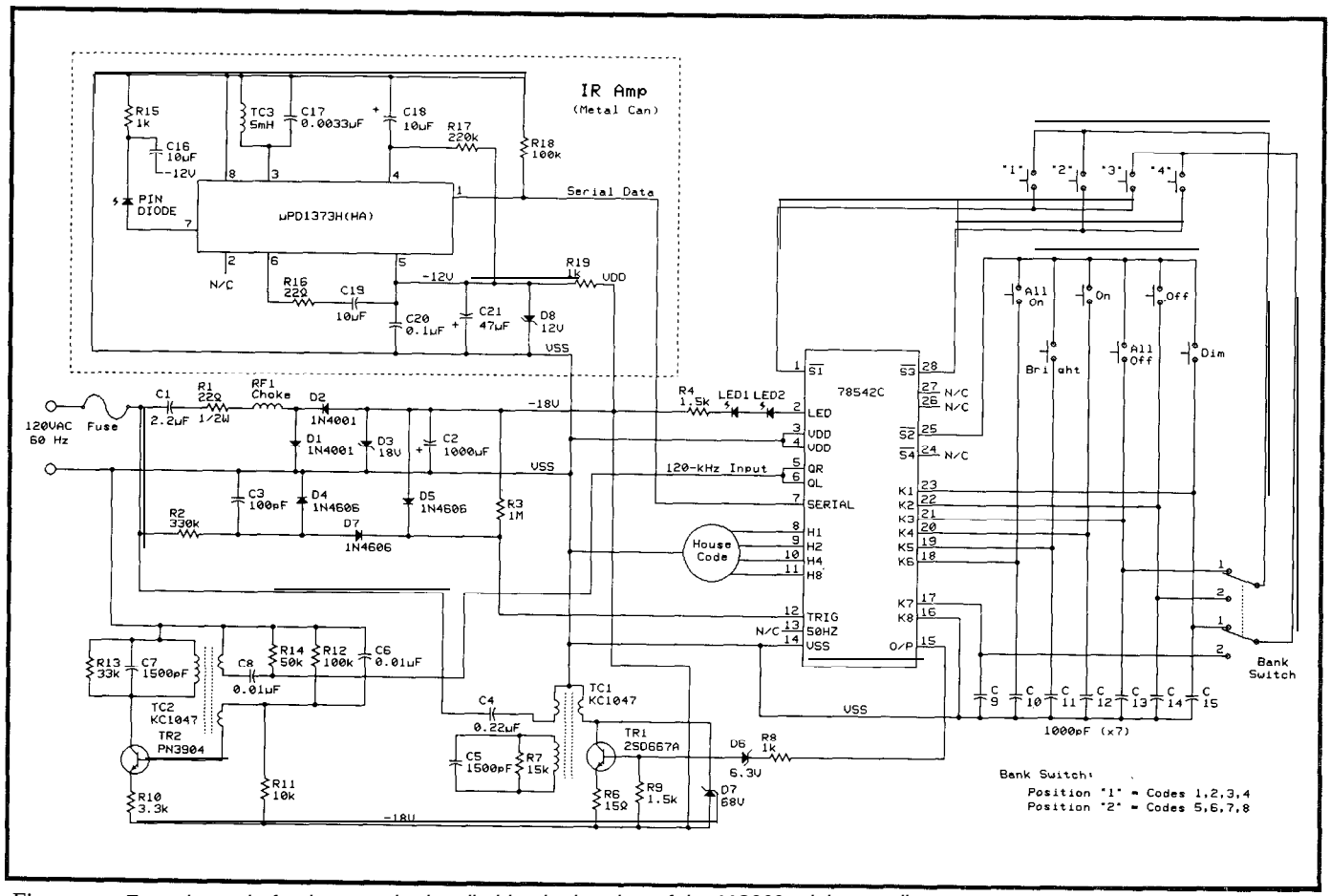


Figure 1—*The schematic for the IR543 is virtually identical to that of the MC260 mini controller.*

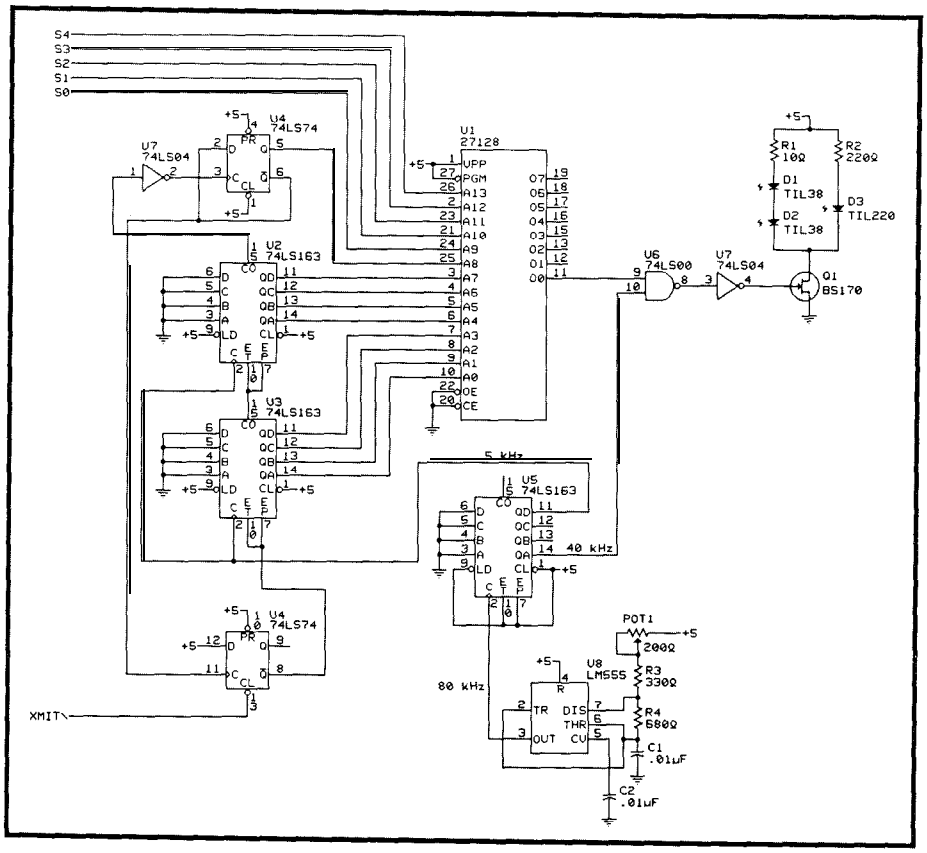


Figure 2—*The basic circuit used for generating X- 10 IR codes consists mostly of an EPROM, a few counters,* and a master clock.

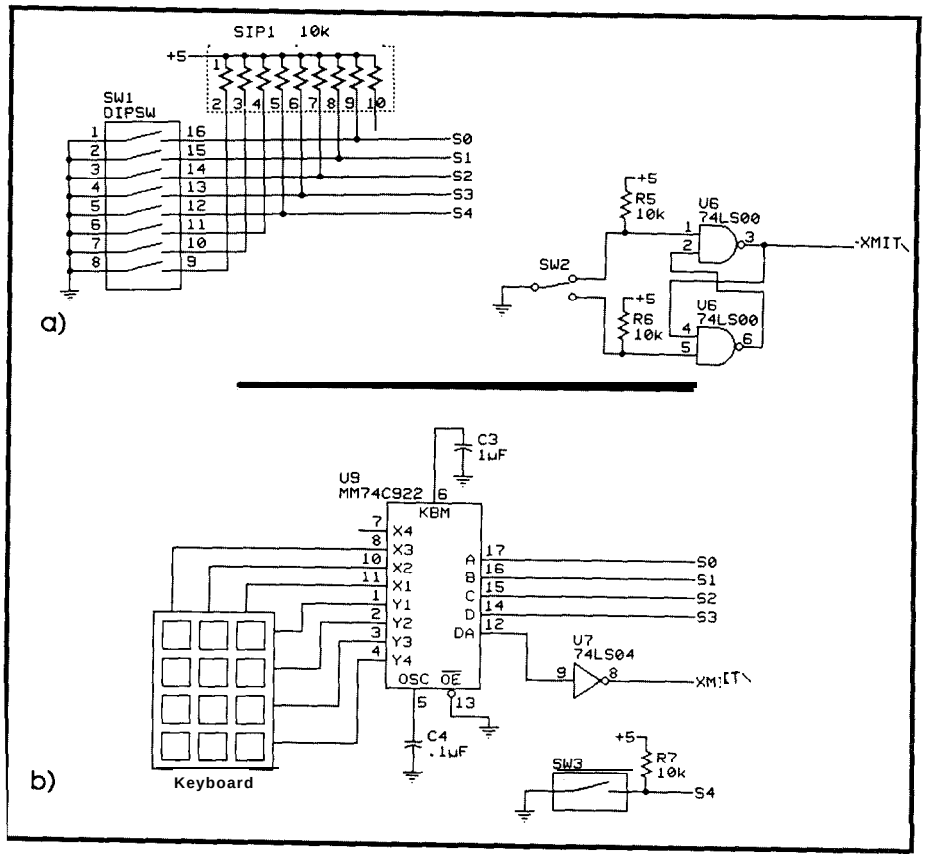


Figure 3—*a) A set of DIP switches plus a 'transmit' switch are all that are necessary for a one-time-use circuit. b) A keyboard may be added for everyday use of the circuit.*

IR remote (the URC-5000 "ONE FOR ALL" from MTC/USA). If you're already in the market for an all-in-one remote, buying the URC-5000 for use with the IR543 may be a good alternative. However, if you already own a trainable remote, or don't want to spend the $100 for one, there must be a better way to generate the proper IR signals for use with the IR543.

Even though X-10 tells me they may make such a low-cost, dedicated remote in the future, there is a definite gap that needs filling. Since X-10 was able to give me complete specs on the IR codes being used, I decided to build my own IR command generator. And since it would only be used to train my trainable remote anyway, it could be quick and dirty.

The Better Alternative

It is possible to interface a simple IR LED driver circuit to an output bit on a personal computer, then drive the LED with some software to train the trainable remote, but it was more fun to build a dedicated, hardware-only board to generate the IR command strings. Using a hardware-only solution also opened the door to the addition of a real keyboard so those who don't have a trainable remote and don't want to spend the money on one can build a usable dedicated remote for use with the IR543. Figure 2 shows the basic circuit for generating the IR commands. Figure 3a has the additional circuits necessary to build a one-time-use-only circuit for use with trainable remotes, and Figure 3b shows what is necessary to add a keyboard to the circuit.

Figure 4 shows a sample command string expected by the IR543. "One" bits are represented by a 4-ms burst of 40-kHz signal, followed by 4 ms of silence. "Zero" bits are represented by a 1.2-ms burst of 40 kHz, followed by 6.8 ms of silence. In both cases, one bit time is 8 ms long.

The command string consists of a "one" bit, followed by five command bits, the complements of the five command bits, then an "end code," which is a 12-ms burst of 40 kHz followed by 4 ms of silence.

As far as timings go, a complete command string consists of an 8-ms start bit, ten 8-ms data bits, and a 16-ms end code, for a total length of 104 ms. Suppose we divide this string into discrete time segments, where each segment is described with a "1" to represent the presence of 40 kHz, or a "0" to represent its absence. We need to generate envelopes for the 40-kHz bursts of 1.2, 4, 6.8, 12, and 16 ms, so the largest subdivision we can use is 0.2 ms. The complete command stringcan then bedescribed asa series of 520 bits, eachbit representing a 200-microsecond slice of time.

Theoretically, we could assemble a table of all 22 commands (16 module numbersand six truecommands) with the bits packed into 8-bit bytes that would end up being 1430 bytes large. If this command table is going into EPROM, though, why throw away most of the EPROM and make the external circuitry more complex by trying to pack things together? If we encode just one bit per byte, we still only take up 11K of a 16K 27128 EPROM.

For the sake of simplicity, let's round the 520 bits down to 512 bits. The missing eight bits represent 1.6 ms, so the delay between bursts will only be 2.4 ms instead of the 4 ms called for in the spec, however the spec also says the minimum delay between bursts can be as short as 2.5 ms, so we're pretty close. It turns out 2.4 ms works just fine. Now all we have to do is clock 512 bits out of the EPROM, one bit every 200 µs.

Two 74LS163 synchronous counters (U2 and U3) plus half of a 74LS74 (U4a) provide us with the 9-bit address necessary to access the 512 bits. The outputs of synchronous counters change simultaneously, so we don't have to worry about glitches coming from the EPROM data line caused by rippling address lines. The output of the EPROM is fed into a 74LSO0 (U6c) which gates the 40-kHz signal coming from the master oscillator. The output of the LSOO is inverted and drives a high-current FET (Q1). This FET drives the IR LEDs (D1 and D2) plus a visible LED (D3) so we know something is being sent.

The master clock for the counters is derived from an 80-kHz oscillator made up of an LM555 (U8) and a few discretes. The 80 kHz is divided down by another synchronous counter (U5) to generate the 40-kHz IR signal (÷2) and a 5-kHz signal (÷16) whose period is 200 µs. Almost any counter can be used (synchronous or asynchronous), but why not keep the parts list simple and use the same kind all around?

To start the transmission of the command string, we use a push button which is debounced with a pair of NAND gates (U6a and b). When the button is pushed, U4b is cleared, the Q\ output goes high and enables the low-order counter, and code transmission begins. When U3 overflows (the count has reached some multiple of 16), it enables U2 for one clock cycle so the high-order counter increments once. When U2 overflows (the count has reached 256), the overflow output clocks U4a, causing the Q output to go high so the second 256 bits of the command can be accessed. Also note that the Q\ output of U4a goes low at the same time.

When counters U2 and U3 reach 256 for the second time, the overflow output of U2 clocks U4a again, causing the Q output to go back to zero and Q\ to go to one. If the push button is still being held down, U4b will continue to be held clear and a second transmissionofthecodebegins. Transmissions continue until the button is released. Once the button is released, the rising edge of U4a Q\ generates a clock signal for U4b, which clocks a one bit into the flip-flop. This action causes the Q\ output to go low, disables the counters, and stops the transmission process. To restart things, the button must be pushed again.

To select which command is sent, a set of DIP switches is connected to the five high-order address lines of the EPROM. Just set a code from 0 to 21 on the switches, press the "send" button, and the selected code is sent out. Obviously, setting DIP switches and pressing a button would be pretty inconvenient for everyday use, but remember, all we want to do is train the trainable remote. After that we can throw the circuit away.

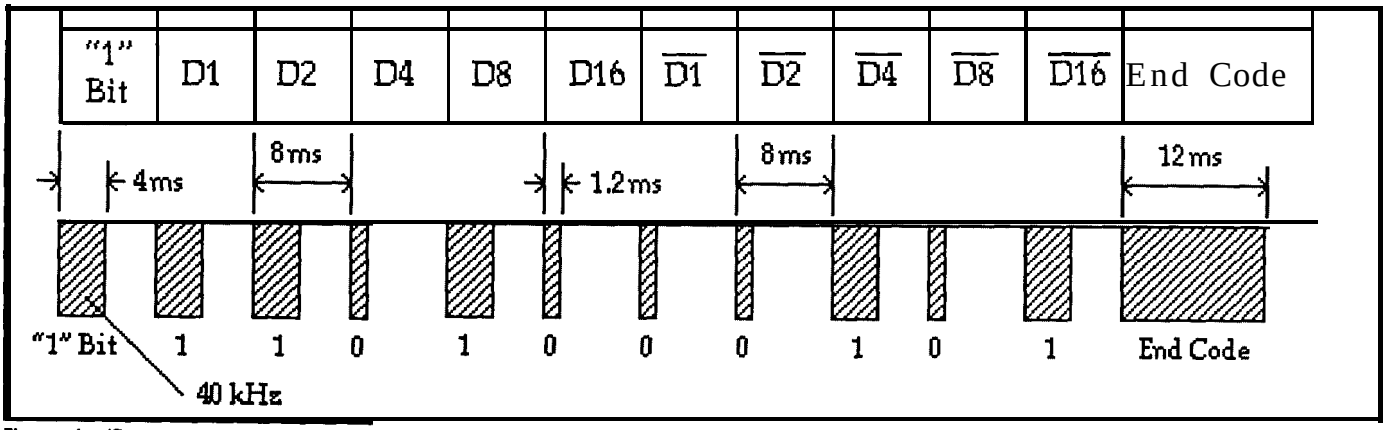


Figure 4—*IR command strings consist Of a leading start bit, the command code, me complement or me commana code, and an en* **code.**

For those who do want to use the circuit every day, a keyboard circuit can be added so that any standard 16- or 20-key matrix keyboard can be used to select which code to send and to start code transmission. The National Semiconductor MM74C922 is a 16-key encoder chip that scans a 4x4 array of SPST push buttons and outputs a 4-bit scan code and a "data available" strobe.

Ideally, since there is a total of 22 commands, a 22-key keyboard would be used with an encoder that presents a 5-bit code. Since the best we can do with the '922 is 16 keys, I decided to use a 12-key keyboard with an extra switch to select between modules 1–8 and modules 9-16.

Figure 3b shows the keyboard interface circuit. The '922 contains all the necessary pull-up resistors and debounce logic, so implementing it in a circuit requires just two capacitors: one for the master scanning oscillator and one for the debounce circuitry.

When a key is pressed, the '922 latches the corresponding keycode into its data output latches and the "data available" (DA) output high. DA is held high for as long as the button is held down.

Looking back at the original transmitter circuit, all that must be done to use the '922 in the circuit is to connect the data outputs to the upper EPROM address lines and an inverted version of DA to the CLEAR input of U4b. When a button is pressed, the correct X-10 code is selected by the data lines and it will continue to be sent as long as the button is held down.

Command	D1	D2	D4	D8	D16
1	0	1	1	0	0
2	1	1	1	0	0
3	0	0	1	0	0
4	1	0	1	0	0
5	0	0	0	1	0
6	1	0	0	1	0
7	0	1	0	1	0
8	1	1	0	1	0
9	0	1	1	1	0
10	1	1	1	1	0
11	0	0	1	1	0
12	1	0	1	1	0
13	0	0	0	0	0
14	1	0	0	0	0
15	0	1	0	0	0
16	1	1	0	0	0
AllUnits Off	0	0	0	0	1
All Lights On	0	0	0	1	1
on	0	0	1	0	1
off	0	0	1	1	1
Dim	0	1	0	0	1
Bright	0	1	0	1	1

Figure **5**—*The command codes sent via IR are identical to those sent over the* power *line.*

The EPROM

Now that we have the circuit, we need to put something in the EPROM. I wrote a quick program in Turbo Pascal to generate an Intel hex file that can be sent directly to an EPROM programmer. The basic 5-bit codes for each command are placed in an array. The program then generates a legal command from the base bit sequence, inserting the proper start code, complemented bits, and end code. Then it breaks the command sequence up into its 512 component bits, and finally writes them out to a file. The extra 5K at the end of the EPROM is filled with zeros so we don't flood the room with IR should the switches be set to an illegal command code. *[Editor's Note: Software for this article is available for downloadingfrom fhe Circuit Cellar BBS and on Circuit Cellar INK Software On Disk #9. For information on downloading and disk orders, see page* 78.1

The beauty of using an EPROM in this application is that the keyboard or DIP switches may be mapped to any corresponding X-10 codes simply by remapping the EPROM.

Parts cost for the quick-and-dirty version of the circuit should be under $20 (add about $10 for the keyboard version) and the circuit can probably be built in an evening or two. Construction techniques are very noncri tical; the highest frequency we're working with here is 80 kHz, so noise isn't much of an issue (we'll save the 16-MHz 68020s for another day). ✣

Special thanks fo Dave Rye for his contribufions to this article.

Diagrams and schematics pertaining to the IR543 are reprinted by permission of X-10 (USA) Inc.

Ken Davidson is the managing editor and a member Of the CIRCUIT CELLAR INK engineering staff. He holds a B.S. in computer engineering and an MS. in computer science from Rensselaer Polytechnic Insfifufe.

IRS

204 Very Useful
205 Moderately Useful
206 Not Useful

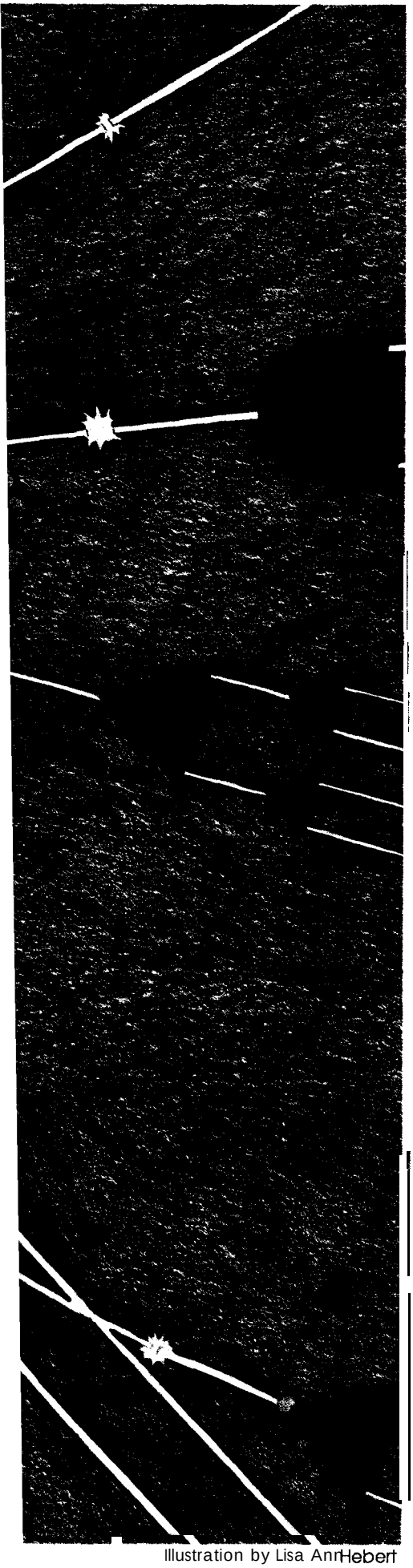


Illustration by Lisa Ann Hebert

FEATURE ARTICLE

A Neural Network Approach to Artificial Intelligence

Using *a Neural Network* for dealing *with Real- World Data*

by Christopher Ciarcia

When you think about modern digital computers, you see fast, dedicated machines that tear through their programmed algorithms at terrific speeds. What you don't see are flexible, adaptable devices that can quickly react to changing circumstances. Unfortunately, the complex demands of many modern applications cry out for flexible solutions rather than mechanical brute force. What we're really looking for, when you get right down to it, is a brain just like ours that can be harnessed to a particular task. Since brains working away in bubbling chemical stews are still the stuff of horror films, however, what can we do to solve advanced application problems?

We can create neural networks of our own! We can emulate our own self-learning, interactive awareness, by creating an artificial neural network (ANN) that reproduces the major components of our own central nervous system. To provide the best possible emulation, our ANN should include sections that correspond to the cerebral hemisphere, which handles sensor processing, abstract thought, and gross motor control; the diencephalon,

which is composed of the thalamus forinformationexchangebetweenthe cerebralcortexand therestof thebrain, and the hypothalamus for regulation of autonomic and endocrine systems; the cerebellum for fine motor control; the brainstem, which acts as an interface for the spinal cord and input for hearing, balance, and taste; and the spinal cord, which is the biological analog of a computer bus connecting to our peripheral nervous system.

We know that our human brain works, so why not create a computer neural network that mimics our own brain's vast web of interconnected neuronic structure? All we need to do is study how our brain's estimated 10 billion neurons and its 10^{14} interconnections are configured and reproduce that structure on our home PC. Simple, isn't it? Just map the brain's complexity onto your computer.

For better or for worse, even if your computer is a Cray X/MI' you won't be able to fully duplicate the complexity of a human brain. What you can do, however, is gain some insight into the "thinking" process while modeling a system that is capable of some real work.

By simulating the basic features of our brain's individual processing units (PU), "neurons" with their associated decisionand learning rules, and structuring "neuron nets" with interconnections that emulate the "learned-stored experience" of the brain's synapses, we can construct an Artificial Neural Network which is "a dynamical system with the topology of a directed graph that can carry out informationprocessingbymeansofitsstate response to continuous or initial input (thedecision and learningrules), with thenodesbeingcalledprocessingunits (neurons) and the directed links or information channels where memory resides being called interconnects (synapses)." [1]

Using a computer architecture similar to our brain, we can develop a system that is highly parallel, highly integrated, noise tolerant, has graceful degradation, contains simple processing units, is memory intensive, associative in nature, and taught, not *programmed.*

Don't Call it a Computer

The ANN is distinguishable from theordinarydigitalcomputerbecause of its radical departure from standard internal organization. Most of today's computer designs call for a separation of a computer's memory and its processor, with a communications link in between. While this arrangement provides for tight control, it has speed limitationsduetobusaddressinteractions. Theneural-netapproachavoids this bottleneck by mimicking the brain's own structure by storing experience and memory in the nodal processor's interconnects. Table 1 shows how the two architectures differ.[2]

The ANN is not built like a "normal" computer. Instead, its memory lies within the path between two elements. It is not stored separate from the "CPU," but is considered an intrinsic part of the information processing. This "informational" connectivity between elements also has arbitrary dimensionality. Any linkage configuration is allowed. And most of all, patterns and response rules are generated internally by correlating inputs and outputs, so the system is not programmed, but taught.

The essential components defining the Artificial Neural Network are its "architecture" which controls the flow of information within the net, its "transfer function/decision rule" which determines how input information and interconnection weights are used to calculate an output value, and its "learning rule" which defines how interconnection weights are adjusted while educating the network.

A System of Modified Inputs

A typical processing unit (PU) operationisshowninFigure2. Within the PU, inputs are modified by the interconnection weights. Given a positive input, a positive interconnect weight will be excitatory, a negative interconnect weightwillbeinhibitory, and a zero-valued interconnect weight interrupts the current link. This filtered input is then used by the transfer function to calculate an output value. Then both the input and output information are used by thelearningrule to "teach" or adapt the weights. These modified weights then alter the processing element's future operation by filtering the input data in a new way.

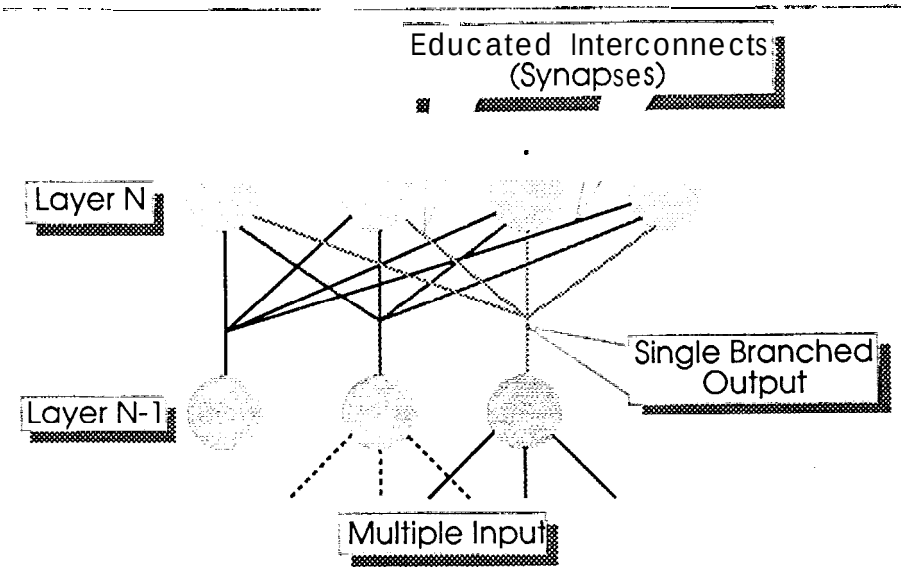


Figure 1 -Each node *at a given level* of *the neural network can* connect, *with varying* levels of 'weight,' to each of the *nodes at the next /eve/ of the network. This* system *emulates the synaptic connections of a biological brain.*

The Transfer Function

The typical transfer function is composed of two parts: an input operator, *f()*, that combines the inputs and interconnection weights to form a single value ready for discriminatory action; and a discrimination function

Digital Computer	Neural Networks
process digital data in binary form	process analog signals that fluctuate continuously
make yes/no decisions based on mathematical or logical functions	make weighted decisions based on fuzzy, incomplete, and contradictory data
rigidly structured sequence of operations with predictable results	independently formulated methods of data processing
definitive answers, given enough time	approximate answers to highly complex problems
sort large data bases for exact matches	sort large data bases for close matches
specific data storage	associative data storage

Table 1 -While *a neural network may successfully be modeled on a digital computer,* there *are important fundamental differences between the methods the two systems use to solve* problems.

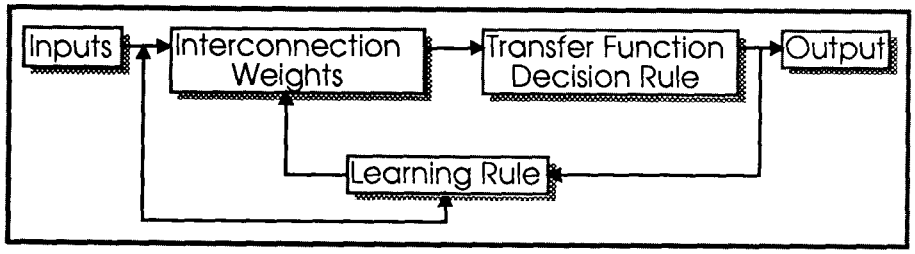


Figure 2-*The Learning Rule modifies the interconnection Weight of a Processing Unit (PU), which in turn modifies the Input to the PU. The Transfer Function modifies the learning rule and determines whether the output of the PU will be excitatory, inhibitory. or disrupted.*

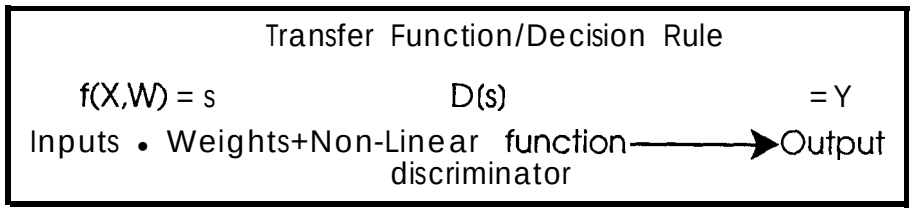


Figure 3—*The Transfer Function/Decision Rule has two basic parts. The input operator prepares data for discriminatory action. The Discrimination Function determines the range of the processing unit's output.*

that governs the output range of the PU. Figure 3 shows the basic form of the Transfer Function / Decision Rule.

But how do we choose the appropriate transfer function for our specific net? What must we consider for an efficient design?

Well, we must first specify what our neural net is designed to do. Will it make decisions or emulate some functional system? And then we must decide on the following details:

1. Are we emulating a biological system? Is it necessary to reproduce the neuron firing rate curve?

2. What are the speed requirements? Can learning and decision making take place within real-time constraints?

3. Should the transfer function be invertible, monotonic, or continuous?

4. What are the nature of our interconnecting weights? Will they be binary, continuous, or discrete?

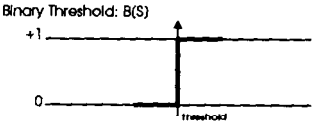
5. And finally, what type of output do we want?

As you can see, there is a lot of flexibility and several degrees of freedom allowed in our choice. There have been many designs created since the early 1950s when the first workable ideas were forwarded by Widrow

Type	Input Operator	Discriminator	Comments
Simple Linear	$S = \sum_i W_i X_i + \theta$	none. Y=S	pure linear function used for associative recall
Weighted Sum	$S = \sum_i W_i X_i + \theta$	Y=D(S)	commonly used; some cases θ=0; Barto and Sutton (1981) add noise to S.
Feedback	$S = \alpha Y_{old} + \beta \sum_i W_i X_i$	Ynew=R(S)	gives persistance to output state.
Sigma Pi	$S = \sum_i W_{ik} \pi X_k$	Y=D(S)	One input can gate another or act as a gain control.
Thermodynamic	$S = \{\sum_i W_i X_i + \theta\} T^{-1}$	p(Y=1)=S(S) p(Y=0)=1-S(S)	p is the probability that a specific state is realized. T is a temperature that is systematically lowered.

(Note: θ is the *Transfer Function Threshold.* α and β are *Feedback Weighting Constants. In all* cases, these *are set* according to the *individual circumstances of the network.)*

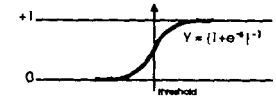
Binary Threshold: B(S)



Advantages:
- Fast & easy use, esp. in hardware
- Makes hard decisions

Disadvantages:
- Not invertible
- No linear zone
- Can't smoothly imitate functions

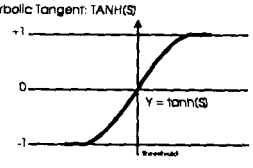
Sigmoid: S(S)



Advantages:
- Invertible and continuously differentiable
- Semilinear zone for function imitation
- Can make "soft" or fuzzy decisions
- Biologically relevant—similar to neuron firing rate curve

Disadvantages:
- More difficult to implement than piecewise linear functions
- Output limited to positive values

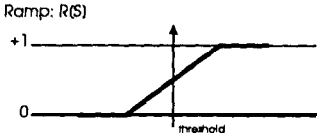
Hyperbolic Tangent: TANH(S)



Advantages:
- Invertible and continuously differentiable
- Semilinear zone for function imitation
- Can make "soft" or fuzzy decisions
- Can produce positive and negative outputs

Disadvantages:
- More difficult to implement than piecewise linear functions

Linear Ramp: R(S)



Advantages:
- Has o linear zone
- Can Imitate functions
- Easy to implement

Disadvantages:
- Not invertible

Figure 4—*There are* several possibilities when *choosing a transfer function and discriminator. Each of the choices has advantages and disadvantages that* make them suitable for *particular classes of problems.*

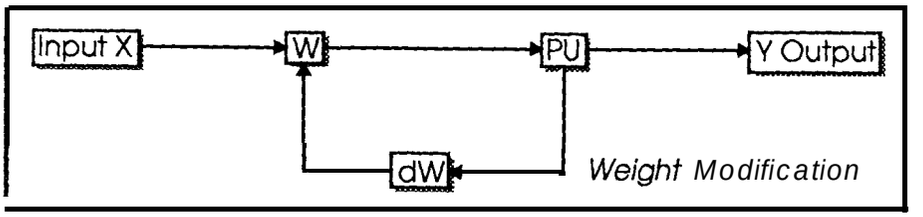


Figure 5—The Learning Rule correlates the input and output values of the processing ur by adjusting each element's interconnection weight. The Learning Rule is the key ; simulating a biological learning process in silicon.

and Rosenblatt et al. We can't discuss all of them here, but we can provide a summary of themoreprominent types of transfer functions and their associated discriminators as compiled by Shepanski[3]. Abreakdown isshown in Figure 4.

The learning Rule

The learning rules's function is to correlate the input and output values of the processing unit by adjusting each element's interconnection weights as shown in Figure 5. It imitates the evolutionary process of loading information into our net, with the ability of our net to "survive" measured by how well it can learn its appointed task or function. This survivability depends heavily on matching our design need to the application. Do we have a purely dynamic system? Will the learning phase rely on local or global information? Is convergence speed important? Each decision will have profound effects on the shape of the final network.

Again, there are many types of learning rules currently in use. We will present here the three **most** widely used.131

Types of Learning Rules

I. Hebb's rule **(1949)**

$$dW_{jk} = nY_jX_k$$

where: $Y = B[\sum_k W_{jk}X_k]$

Here, weight modification occurs only if both input and output are nonzero; so the weight only decreases.

II. Windrow-Hoff rule **(1960)**

$$dW_{jk} = n(d_j - Y_j)X_k$$

where: d is the desired output and

$$Y = R[\sum_k W_{jk}X_k]$$

Here, weight modification occurs only if X is nonzero and the actual output does not match the desired output. The weight adjustment can be positive or negative with Y tending to converge to d.

III. Generalized Delta Rule (Rumelhart et.al., 1986) (backward error propagation)

This is a generalized version of the Widrow-Hoff rule. The difference between the desired output and actual output of a PU is used to correct the interconnection weights. This differential ("delta") for an output layer is calculated by:

$$\delta_k = Y_k (1 - Y_k)(d_k - Y_k)$$

where:

$$Y_k = S[\sum_j W_{kj} X_j + \theta_k]$$

Index j refers to the PU closer to the input layer while index k refers to the PU closer to the output layer.

For interior layers the difference between the actual output of the PUs and their optimal output is propagated backward through the network:

$$\delta_j = X_j (1-X_j) \sum_k \delta_k W_{kj}$$

The interconnection weights for all layers are corrected using:

$$dW_{kj} = n\, \delta_k\, Y_j$$

An Application: Pattern Recognition

Now that we're experts on how to develop our own "PC-resident Expert System," let's apply some of the information to a specific example: a self-learning pattern recognition system.

What we want to create is an ANN that will learn different images or object patterns. From that experience, we want our "educated computer" to have the ability to "remember" and "identify" those images or objects accurately. In addition, it must also recognize the object if we change its orientation (e.g., tilt or rotate it).

How do we do this? Let's start by choosing a net structure, a transfer function, and then a learning rule.

Choosing the Net Structure

To specify the net structure we must determine the nature of the input. Since large images require large inputs, and thereby large array space and processing time, we will confine our example to 16x16 image arrays, based on a 0 to 255 gray-level scheme. Each pixel gray value is the initial input into a net processing unit. There are 256 initial inputs.

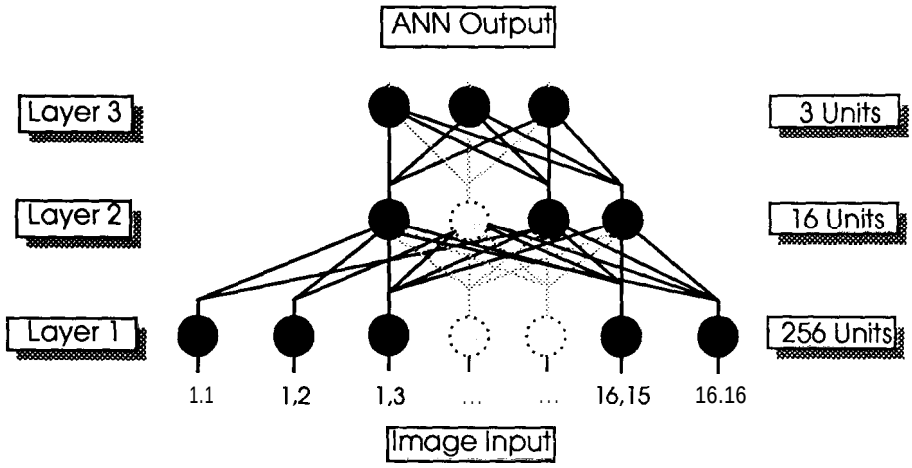


Figure 6— The *neural network of the sample problem operates on a simple 16 x 16 pixel matrix. Despite the simplicity of the input,* the *network has 275 processing units and over 4.100* interconnects, *This illustrates why large neural networks are* still *the province of large* universities and research laboratories which have access to supercomputers.

image size: *16x16*
number of inputs: *256*
input **values:** *O-255*

The actual number of layers of PUs and PUs per layer is determined by the sensitivity and speed of learning that we want within our net. I have found that a single-output PU is

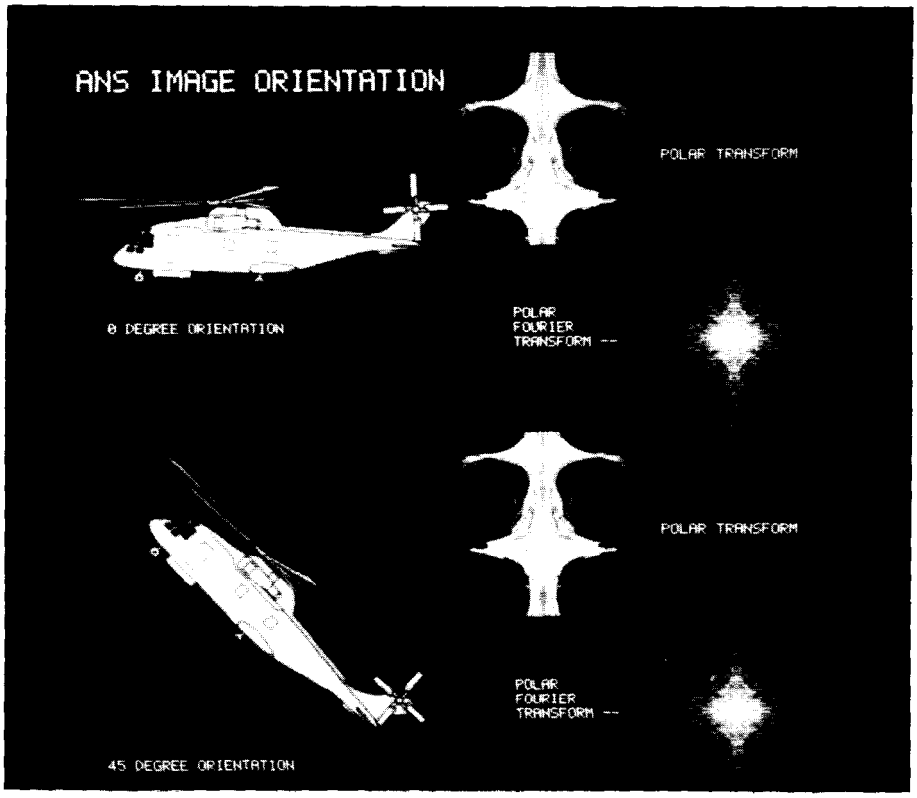


Photo 1—*Our sample used several images to test the network. To examine network sensitivity, we expanded the images to 256 x 256 pixels. By using a Polar Fourier Transform on the input picture, an image is produced which is consistent regardless of the orientation of the original. This removes a major variable in image recognition.*

insufficient for our problem. Instead, we will use three PUs in our final layer and 16 in our middle layer.

number of layers: 3
units in layer 1: 256
units in layer 2: 16
units in layer 3: 3

We will also allow for maximum interconnectivity between each layer. So the output of each PU on a layer is an input to each PU to the layer above.

# of interconnects for PUs on layer 2: 256
# of interconnects for PUs on layer 3: 16
total # of interconnects: 4163 within the net

As you can see, the total number of interconnects becomes very large very quickly. The number of interconnects required by a large network is one of the main factors that prevents us from modeling the complexity of even a simple biological brain.

Choosing the Transfer Functionand the Learning Rule

Our application requires sufficient sensitivity to differentiate images. But we want it truly "smart," so it must have the ability to make "soft" or fuzzy decisions. At the same time, it must have the ability to learn efficiently. That is, it must have the ability to adjust its interconnection weights in a backward error propagation manner. This will allow the desired output **and** the actual output of the "learned-experienced" net to converge to a preset value. For that reason, our example uses:

Transfer Function
input operator: weighted sum

$$[S = \sum X(L,i)W(L,i,j)\ 1$$

discriminator: Sigmoid

$$[S[S]\ Y = (1+e^{-s})^{-1}]$$

Learning; Rule
Generalized Delta Rule

Because of the Sigmoid discriminator function, our net is extremely sensitive to the range of the input values on the first layer. The image input was therefore renormalized (using ANORM) to range from 0 to 0.010. Remember, if the initial input X is too large, Y converges to 1 extremely quickly. If X is too small, Y converges to 0.50 slowly. So, if you try various values for the normalization constant ANORM, you can crudely adjust the net's ability to learn and later discriminate. You will also see marked changes in the time-rate of convergence of the net towards its 'learned" state.

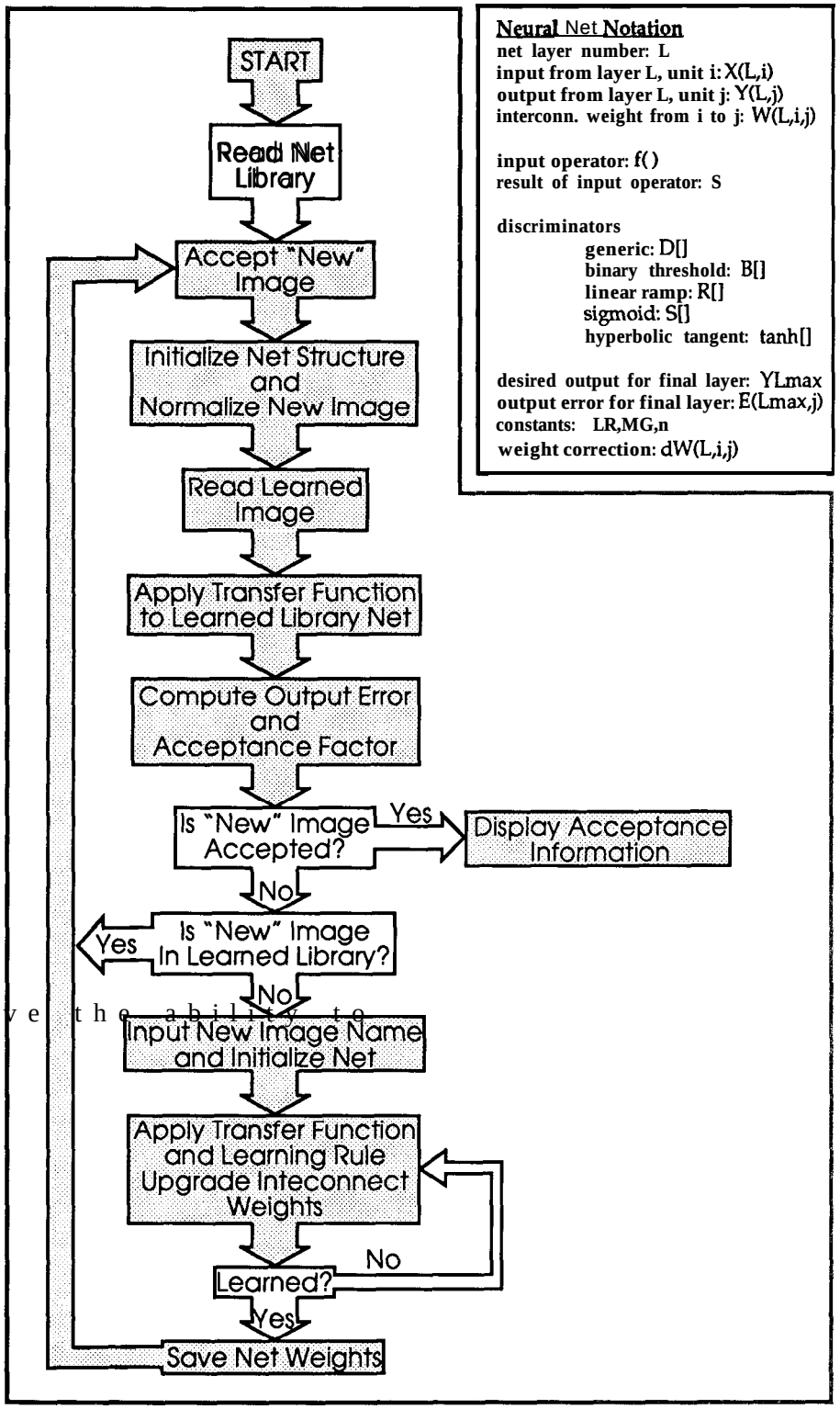


Figure 7–*The program flows in a relatively straightforward manner. Termination can occur either after display of acceptance data or as an alternative to new image input.*

The Pattern Recognition Code

Our example contains three basic components. First is a library utility that keeps track of the different learned net values. Then there is a recognition utility that runs an input image through a specific learned net and decides if it recognizes it or must learn it as a new image. Finally, there is the learning mode. Here the new image is iteratively cycled through the net, applying the transfer function and learning rule for each cycle until the net converges to a specified "learning sensitivity." This sensitivity is measured by the difference between the actual output and desired output for each of the PUs in the top layer. An acceptable learning sensitivity is a difference of 10^{-10} for each output.

In order to study network sensitivity, our example was expanded to handle 256- x 256-pixel images. Several different images were then created to test the net's ability to learn and recognize small variations. Some examples of these images are shown in Photo 1.

I tried several different types of tests. I created several different images with the same weight density of pixels in order to test for true image differentiation. The net had good sensitivity for both the 16x16 and 256x256 images. I then rotated various images to see if the net would still recognize them when using the zero-degree learned orientation. And it could, but it was highly resolution dependent. When using 16x16 images, of little or no specific detailed structure, I could still recognize the aircraft versus the helicopter no matter its orientation (using fairly broad acceptance settings). With 256x256 images there was much more detail learned by the net, requiring initial processing of the input image data.

What was desired was a rotationally invariant image that the net could learn. This was achieved by taking a polar transform of the initial image, mirroring the transform to form an nxn image, and creating a power spectrum image of its two-dimensional FFT. This is called a polar Fourier transform (PFT).

To demonstrate how this works, consider Figure 7. The polar transform is readily created. Starting from the center of the image, extend a radius vector out one pixel and rotate about the image 360 degrees. At each increment, store that value down the first column of the polar transform array. Then, increment the radius vector out another pixel in length and cycle around again. Continue this until the radius vector reaches the edge of the image. This results in a 16x8 polar transform of the 16x16 image, and a 256x128 for the 256x256. Now mirror the polar transform to create a 16x16 array. Notice that both polar transforms shown in Figure 4 are the same, even though they are rotated by 45 degrees. This rotation only shifted the polar transform in a wrap-around style. In the Fourier domain this is only a phase shift and does not affect the power spectrum [4]. So, our polar Fourier transform is the same no matter the orientation. We can feed it to the net for learning. Then, if we process each of our "new" images similarly, it will recognize a rotated version of itself.

Now, all of this works perfectly well on paper, but don't be fooled. Theoretically these power spectra are invariant under rotation, but we must still consider the image digitization process. A straight line at zero-degree orientation becomes a jagged line at 45 degrees. This extra "degradation" of the edge detail can slightly modify the PFT and add to the uncertainty of the recognition. But then, isn't that why we want to use a neural net approach? The ability to work with data that's "the same, only different" is the fundamental benefit of working with neural nets. Where other computerized methods use precise templates of objects to recognize them, neural nets can learn those characteristics that distinguish a class of objects and apply them to the general case.

The Learning Continues

Well, have fun running the neural networks on your computer. There are lots of different applications to which you can apply this code. *[Editor's Note: The code for this article is available for downloading from the Circuit Cellar BBS, or on Circuit Cellar INK Soffware On Disk #9. For downloading and* ordering *information, see page 78.1* The initial inputs **need not be image pixel values. They could be the signature of a scope trace or some characteristic of a system you wish to sort. Or you could build an intelligent security monitor. Teach it to recognize a certain area; then sample an image of that area periodically and run it through the net. If there is a change, the net will notice. Your imagination is your only limit. Because, unlike conventional computers, neural nets can learn from their experiences and make those "soft-fuzzy" decisions which we humans are so famous for.**

Acknowledgements

The author would like to extend special thanks to TTC Seminars, and *especially John J.* Rosafi, *John Shepanski,* Michael *H. Myers, and* Robert *M. Kuczewski, for the excellent qualify of* their *classes on "Artificial Neural Systems" that he recently attended in Anaheim, California. Many of the ideas and concepts presented there educated his infernal "neural networks," making this work possible.*

References

1. **R.** Hecht-Nielsen,Hecht-Nielsen Neurocomputer Corporation.
2. Business Week, June 2, 1986.
3. J.J. Rosati, J. Shepanski, M.H. Myers, and R.M. Kuczewski, "Artificial Neural Systems Seminar," presented by TTC Seminars, May 1988, **in Anaheim CA.**
4. **E.O. Brigham, 'The Fast Fourier Transform," Prentice-Hall, Inc.,** Englewood Cliffs,New **Jersey, 1974.**

Chris Ciarcia has a Ph.D. in experimental nuclear physics and is currently working as a staff physicist at a national lab. He has extensive experience in computer modeling of experimental systems, image processing, and artificial intelligence.

IRS

207 Very Useful
208 Moderately Useful
209 Not Useful

The ADALINE Learning Neuron

A One-Node Net for Computer learning

by Scott Farley

rtificial intelligence has been quietly diffusing into the world of real applications for 30 years now. The diffusion process is speeding up, so that artificial intelligence techniques are having significant effects in quality control, exploration for oil and minerals, the familiar computer chess games, medical diagnosis, and understanding of natural language by computers. Artificial intelligence is our here and now, and is becoming a large part of our future. The stumbling block, for those working with microcomputers, has been that even simple artificial intelligence techniques require serious computing horsepower and megabyte-sized blocks of available memory.

The horsepower/memory stumbling block has been particularly large in the case of one of the more promising "new" techniques. When they were first conceived in the 1950s, Perceptrons (artificial neurons) were dismissed by the leading experts in artificial intelligence. In the last five years, networks of artificial neurons have been reexamined as tools for working with data incompatible with traditional digital computers. Excitement over the possibilities of neural networks has grown as research has progressed. The problem is that even a small neural network can have thousands of processing nodes (neurons) and tens of thousands of interconnections. The memory and processing requirements have prevented many private engineers from looking seriously at neural networks for solutions to current problems.

Fortunately, artificial neurons do not have to be connected into extensive networks to be useful. Individual neurons can solve many smaller problems, and working with single neurons is a useful step towards understanding larger neural networks. I used a simple artificial neuron to solve a problem dealing with indistinct data. The solution shows that you don't *really* need a Cray to start working on the leading edge of technology! A PC-compatible with BASIC will work.

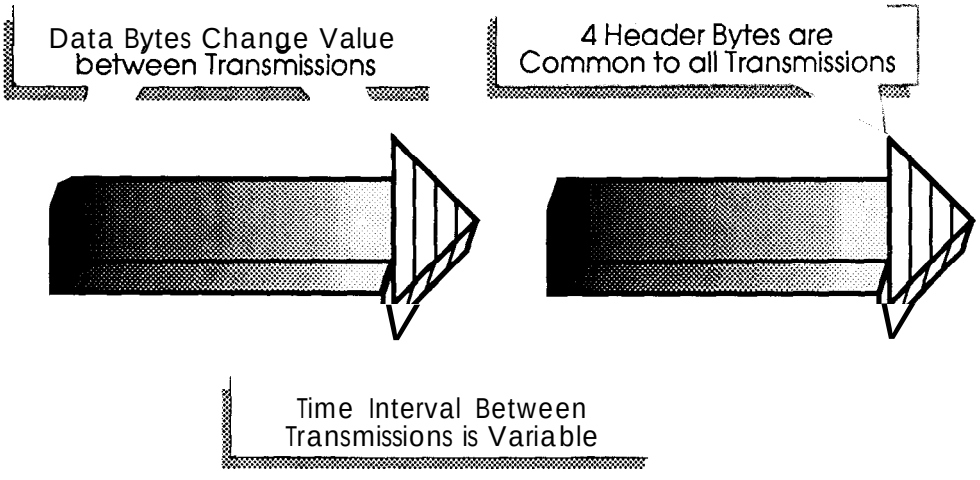


Figure 1—*The data pointer for transmissions needs to be locked on the first header byte. Since there is no specific time interval between transmissions, successful reception of the* header *bytes is crucial* for successful data reception.

A Problem to Solve

Several years ago, I had an oversized hobby which masqueraded as a mobile robot. It was a good scratch-built learning platform for both hardware and software. It was designed to wander through an environment taking sonar range information and passing it to a host computer. The idea was to make the robot as simple as possible and handle guidance issues in the host computer. This scheme required communication links between the two main processors, and radio seemed the obvious choice.

Information transfer was to be asynchronous, so each end of the link had to read a stream of data which would arrive without warning. The timing between data streams varied due to the unpredictable nature of the computers' tasks, but an unchanging 4-byte header was used at the front of each transmission. Software matching of the four bytes would allow us to set a data position pointer which could be used to identify the location of the individual data bytes. Figure 1 shows a simple representation of the data stream.

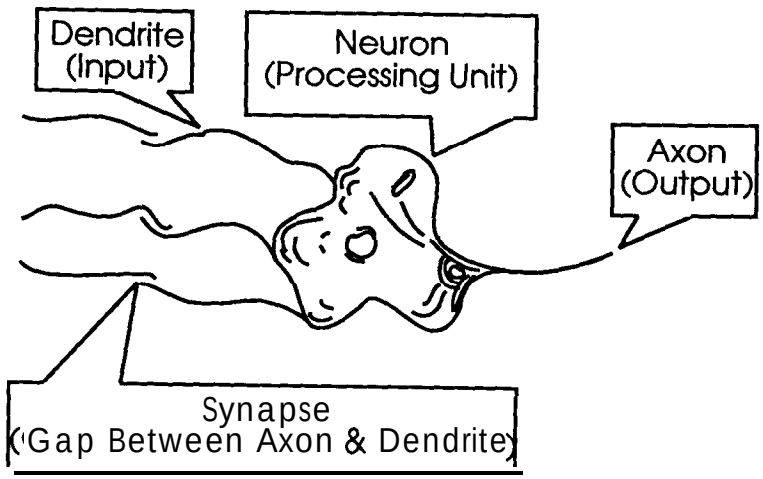


Figure 2—*The pulse rate of the axon* reflects the activation level of the neuron. Any *individual neuron can be* trained to pass. attenuate, or block *inputs as received from other cells. The* activity of *the neuron depends on its past history of training.*

There was one serious hitch in this flawless scheme: The DC drive motors produced enough radio frequency interference to corrupt some of the data coming to the robot. The corruption was not consistent **enough** to allow a simple mechanistic fil tering solution, and it was severe enough to require a working solution. After all, a robot you can't control is entirely too much like a child.

The first solution that came to mind was multiple transmission of data, requiring two or three in succession be the same before we act on them. This should work on the data bytes if we have the data pointer synchronized. But it would not work on the header bytes, since they synchronize the pointer, and not vice versa.

When we tried to dynamically work with individual bit errors in the header bytes, we discovered that the bit position of the error greatly affects the value of the resulting number. Bit position 0 adds or subtracts 1, while bit position 7 adds or subtracts 128. After working with this scheme for a while, we decided that the overhead was just too high. Similarly, calculating CRCs for all of the data would have been fine in theory, but would have soaked up entirely too much of the computing power available on the robot. What we really needed was a flexible, low-overhead **way** of dealing with erratically changing data and decidingwhattheoriginaldatalooked like.

I decided that a neural technique would bean ideal solution, but I didn't need the power or complexity of a full-blown neural network. A single neuron can certainly test for exact matches of binary input bytes. It has a numericaloutput whichrepresents the

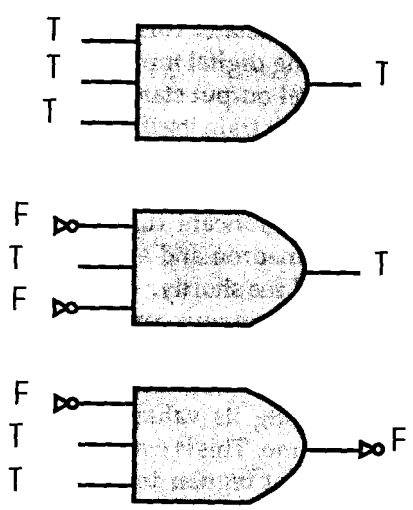


Figure 3—*There are similarities between an artificial neuron and a simple AND gate with inverters. The differenceslie In the neuron's random starting state versus the* constant state *of the* AND gate; the *adaptive* (changing) nature *of the neuron; and the output* of the two *devices. An AND gate produces a simple binary output while the neuron produces an output that reflects the degree of similarity between the trained input and* the presented *input.*

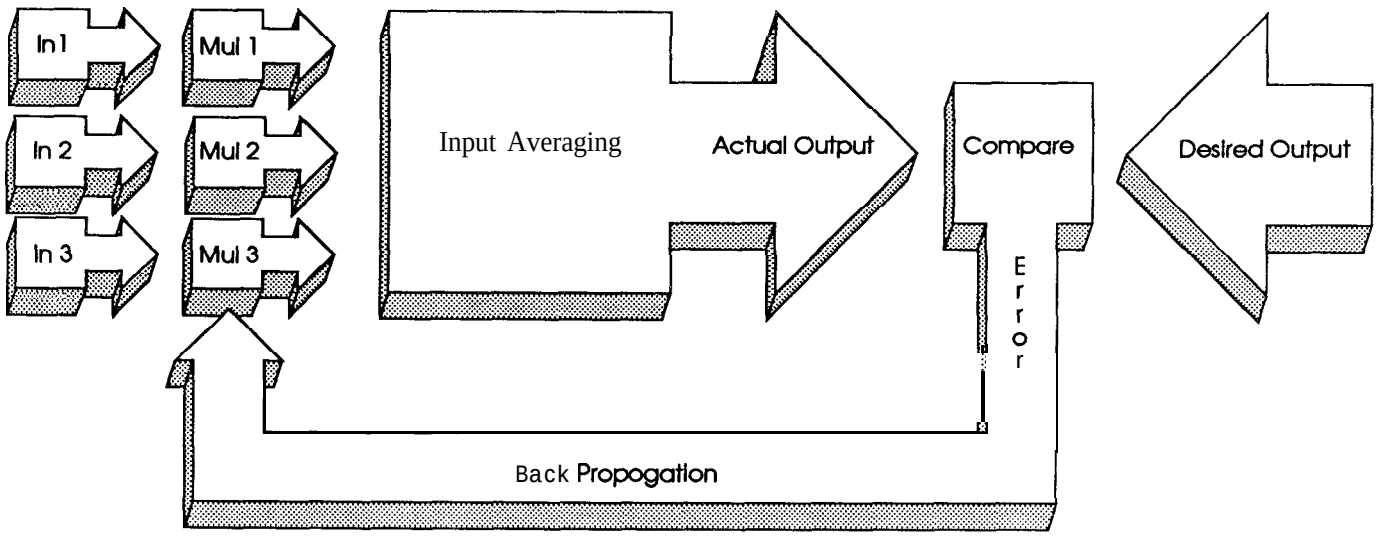


Figure 4-Ihe *multipliers* of *the* neuron begin *at* a *random value between -1 and + 1. They are modified by the back-propagation process to bring the actual output closer to the desired output.*

pattern on which it was originally trained. Simply insist that the output number for the bytebeing tested match the trained output number. For **single-**bit errors, you allow a percentage difference between the two output numbers. For **multibit** errors, you allow a larger percentage difference. The process is the same regardless of the bit position in which the error occurs.

What is a Neuron?

Real neurons exist only as part of a biological nervous system. We choose to copy their actions in hardware or software because they can accomplish things not normally considered possible with digital computer systems. In particular, they lend themselves to pattern matching and parallel processing.

An individual neuron cell has several inputs (Dendrites) and one output (Axon). It is interconnected withmanyotherneuronsand uses the frequency of electrical pulses to have itsoutput influence inputs for several other neurons. The cell has the ability to have inputs from other neurons increase or decrease its output frequency, or have no effect. This is based on its past history and shows that it is "trained" to respond in a particular manner as shown in Figure 2. These neurons are the basis of all of our nervous system and thinking functions.

Understanding Neurons

You can think of a neuron as a biological simulation of a digital AND gate. If we take a multi-input AND gate and "train" it by adding inverters in the input lines and the output line as shown in Figure 3, we can have a particular input pattern **producea par-**ticularoutput state. Thisisequivalent to providing digital inputs and a desired digital output state to a neuron and having it train itself to those conditions. While the neuron/AND gate can help you understand the neuron, there are significant differences between the neuron and an AND gate, as we shall see shortly.

Any continuous process which occurs over a period of time in the real world can be simulated (represented) by measuring its value at different points in time. This is usually done by measuring at regular intervals, such as once per second. The parameter which is measured can be analog or digital. An analog example would be measuring a voltage which changes, but is somewhere between 0 and 5 volts. A digital example might be the opening or closing of a switch.

The program I wrote to solve the data problem simulates an analog system by breaking the analog values into numbers between **+1** and -1. It also simulates the passage of time by taking a set of numbers and modifying them a little bit every time the program loop is run through. In this way, successive loops of this part of the program can cause numbers to change in specific directions at specific rates.

The way these changes are handled is really quite simple. The value of a variable is equal to its old value plus some modifier value. The modifier value is calculated from the group of old values with which we enter the calculation pass. In this program, some of the old values are modified in each pass, and others do not change.

Understanding the ADALINE Neuron

One early artificial intelligence concept (in **thelate** 1950s) was the idea of duplicating the **way** a neuron works using electronics. This single neuron was first **doneby** Bernard **Windrow** of Stanford University. He called it the Adaptive Linear Neuron, which today is shortened to ADA-LI-NE, or **ADALINE.**

You will find three substantial differences between it and the AND gate. TheANDgatewill,fromthefirst operation to the last, give uniform output from uniform input; with

ADALINE, the output for any given input pattern initially depends on random factors. The AND gate is "hard wired" to give a specific output from a known input; given a desired output and a likely input, the ADALINE changes its initial random factors to "train" itself to output the desired state. Finally, where the AND gate is a binary device, outputting either a "1" or a "0" depending on the input, the ADALINE has a number as part of the output which represents the degree of match between the pattern it was trained on and the pattern presented to it for analysis.

Figure 4 shows the various functions the ADALINE uses to perform its processes. The two major divisions are initial learning on a particular pattern, and comparison of other patterns to determine how closely they match the original.

Adaline Learning

You initialize the learning process by presenting the desired input pattern (digital ones and zeros in this case), the desired output state (one or zero), and **randomly** generated multipliers which control how much effect each input bit will have on the output state. The output state is represented by a number which can range continuously between +1 and -1. This process is called Forward Activation and is the first of three steps in the learning process.

The second step consists of determining whether the value of the output state number is close enough to the desired output state to stop the processandsayleamingiscompleted. If the learning is complete, the system can proceed to ADALINE pattern comparison. If the output is still not sufficiently close to the desired result, the process proceeds to step 3.

Back Propagation is the name of the third step. It is started by comparing thedifferencebetween the desired outputstateandtheactualoutputstate from step 1. The error value (difference) is used to modify the randomly generated multipliers from step 1 so that they are changed in a way which will reduce the error value. The amount of change is purposely kept small to prevent oscillation about the correct values when the output is nearly correct. The initial random values will become trained values as this iteration looping proceeds.

Adaline Pattern Comparison

Now that the neuron has left the training iteration loop and the initial random values have been converted to trained values, theoutput state value is now close enough to the desired state for the ADALINE to recognize the pattern it was originally trained on and produce the output state it was trained to have.

When you present the trained ADALINE a bit pattern for it to respond to, it goes through the Forward Propagationprocessdescribedasstep 1 under ADALINE learning. Since the training process is completed, it does not attempt to train itself to the new pattern or alter the values of the multipliers. The output state will be correct (match what it was trained to) **only** for the bit pattern it was trained on. For most other bit patterns, the output value, which can range continuously between +1 and -1, will be far enough from the desired output that they cannot be digitized and have to be represented as indeterminate. For the exact inverse of the bit pattern the ADALINE was trained on, the value of the output value will be the same, with the opposite sign. The digital output state will be the opposite also.

Human Input	Computer Input	Randomly Selected Initial Weight Multiplier	Weighted Input
TRUE	1	0.25	0.25
FALSE	-1	-0.36	0.36
TRUE	1	0.04	0.04

Table 1 — The forward activation process assigns random values to the multipliers which are used to process the input, The output value from this process is the average of the weighted input.

Desired output		Weighted Input		Input Error
1		0.25	=	0.75
1	-	0.36	=	0.64
1	-	0.04	=	0.96

Table 2—*The input error is the difference between the trained output and the output resulting from the provided input.*

This output value can be used to rate the closeness of match between the bit pattern the ADALINE was trained on and another bit pattern presented to it in this mode. This is the area where it transcends comparison to the AND gate.

Simulating the ADALINE

Real neurons change their activation level over a period of time. Your ADALINE will also change over time as its random values become trained values through the iteration process. These internal change processes are analog in nature, so we will represent them by a range of numbers from +1 (TRUE) to -1 (FALSE). This means that input patterns and the output state are represented in the computer program as +1 or -1. Other internal processes require an analog representation, so we will restrict them to any value between +1 and -1. The output state number and random multipliers are two of these.

The iteration includes running the number system through several loops. Each loop can be envisioned as a frame in movie film with small but definite changes with each succeeding frame. This program will usually quit after six to ten loops. At that point the output value has changed considerably, as have the initially random multipliers.

This detailed look at the numbers from the computer program will provide you with a detailed understanding of the ADALINE. The software for this article is written in GWBASIC, and will run on any PC-compatible computer. A companion text file with instructions for running the program is also provided. *[Editor's Note: Software from this article is available for downloading from the Circuit Cellar BBS and on Circuit Cellar INK Software On Disk #9. For information on downloading and disk orders, see page 78.1.*

Inputs are three input bits, random multiplier values, and the desired output.

ADALINE's Numbers

The steps for training our ADALINE include:

1) Forward activation
2) Test. If close enough, Exit
3) Back propagate to make small adjustment. Loop to 1.

Forward Activation

You start the process by inputting the desired input pattern and the output you want it to represent. The

programaddsrandomvaluesbetween 0 and 1 to be used as multipliers. It also assigns 1 to TRUE and -1 to FALSE. The forward activation process is shown in Table 1.

The arithmetic mean of the Weighted inputs is 0.22. This average is the neuron's output and is an analog value between -1 and +1. Note that the desired output is TRUE, so this output is positive. Had you asked for a FALSE output, the signs on the weights would be changed so that the weighted inputs and average would be negative. It's a simple thing so far.

Putting it to An Initial Test

The output needs to be 1 for TRUE, and instead the neuron has 0.22. You are now at the second step where you test. As you go through successive loops in the process, the initial weights which were randomly selected will be adjusted to larger and larger values. This means that the output will also become larger and approach 1. It will never quite get there, so the test needs to determine if it is within 20% of the desired value. The test is whether or not the actual output is greater than 0.80 (80% of 1). It is not, so you continue on to the third step. Note that if the output is greater than 0.80 it is TRUE, if it is less than -0.80 it is FALSE, and if it is in between it is indeterminate. This is how we digitize the "analog" output.

Back Propagation-The learning Process.

Now for the magic: The neuron is finally going to learn something. Back propagation will adjust each of the weights a small amount in the right direction. First you will need to calculate the input errors, which are positive for a TRUE desired output and negative for a FALSE desired output. The initial error values are shown in Table 2.

Our next step is applying the "Delta Rule" which determines the amount of change we will make to the weight value. The adjustment size is:

$$A = \frac{\text{Step Siie} \times \text{Input Error} \times \text{Weighted Input}}{\text{Input}^2}$$

$$A1 = \frac{0.50 \times 0.75 \times 0.25}{1 \times 1} = 0.09$$

$$A2 = \frac{0.50 \times 0.64 \times (-0.36)}{1 \times 1} = -0.12$$

$$A3 = \frac{0.50 \times 0.96 \times 0.04}{1 \times 1} = 0.02$$

We can now adjust the weight multipliers:

$W_1 = 0.25 + 0.09 = 0.34$
$W_2 = -0.36 + (-0.12) = -0.48$
$W_3 = 0.04 + 0.02 = 0.06$

The next move is to loop back to the forward activation part, using these new weight multiplier values, and calculate the next neuron output value.

ADALINE's Loops

Running the actual program with these values results in nine iterations. The output values are:

Iteration #	W1	W2	W3	Output Values
1	0.25	-0.36	0.04	0.22
2	0.34	-0.48	0.06	0.29
3	0.46	-0.60	0.09	0.38
4	0.58	-0.72	0.13	0.48
5	0.70	-0.82	0.18	-0.57
6	0.81	-0.89	0.26	-0.65
7	0.88	-0.94	0.35	-0.73
8	0.94	-0.97	0.47	-0.79
9	0.97	-0.98	0.59	-0.85

With the output greater than 0.80, the network is now trained with the weight values currently in place.

ADALINE Pattern Comparison

You can now use the trained neuron in Forward Activation mode to see the effect of various input patterns. If you go to the program and run it with some input patterns, the result might be as those shown in Table 3.

You can see that inverse input patterns produce inverse outputs; the pattern it was trained on (the first) still produces the same output; and that the last two show something about the weight system that is quite inter-

Input 1	Input 2	Input 3	Actual Output	Digital Output
TRUE	FALSE	TRUE	0.85	TRUE
FALSE	TRUE	FALSE	-0.85	FALSE
TRUE	TRUE	TRUE	0.19	indeterminate
FALSE	FALSE	FALSE	-0.19	indeterminate
TRUE	FALSE	FALSE	0.45	indeterminate
FALSE	FALSE	TRUE	0.20	indeterminate

Table 3—*ADALINE returns a TRUE value if the input produces output within acceptable limits of the trained output. In this case, the program accepts output within 20% of the desired output.*

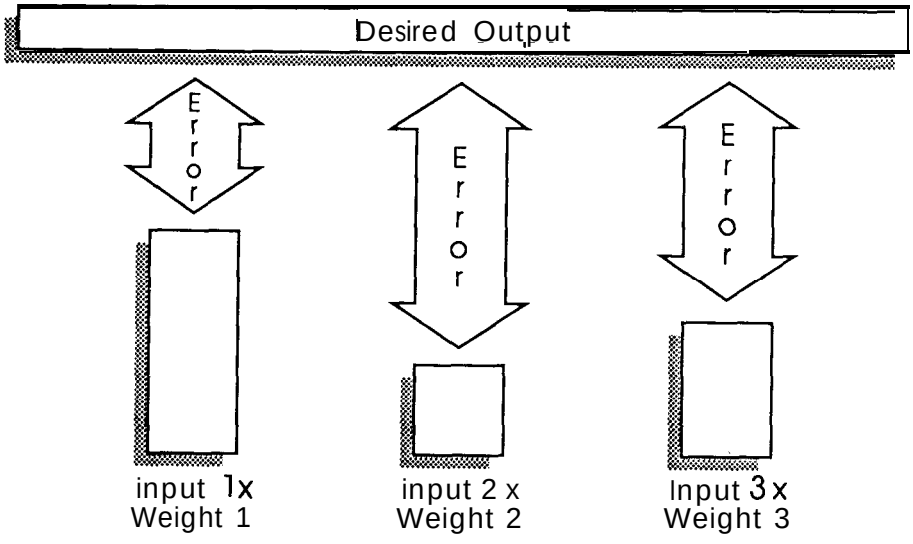


Figure 5—*The error for each input is the difference between the actual output and the trained output. The error for each input is processed separately.*

esting. The difference between 0.97 for W1 and 0.59 for W3 means that the first and third inputs do not have the same effect on the trained neuron. Changing input 1 from TRUE to FALSE changes the actual output from 0.85 to 0.45, while changing input 3 from TRUE to FALSE changes the actual output from 0.85 to 0.20. The other inputs are not changed from the first line for this comparison.

You now have a basic understanding of the ADALINE, but you'll need to look more closely at several items, such as the delta rule, and why it is structured as it is.

ADALINE's Feedback System

The error for each input is derived from subtracting the value of the individual input times its weight from the desired output value. Figure 4 shows it graphically.

Your weight values are adjusted more if the error is large to speed the training process. When they are near the optimum value, the error is small. They are then adjusted in small incre-

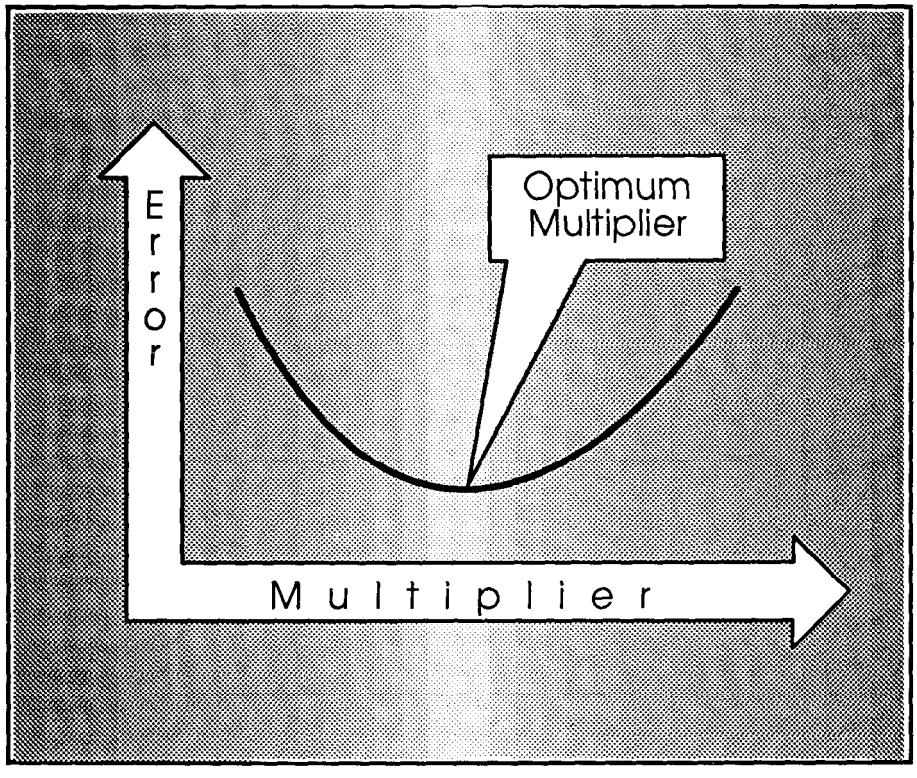


Figure 6—As the iteration process proceeds, the value of the multiplier increases as it moves toward the bottom of the curve. The optimum multiplier is the minimum value which provides an output acceptably close to the trained output. Large-value multipliers tend to oscillate around the trained value while never coming within acceptable limits.

Pattern Trained on	Resulting Output Value	Percent Difference	Decimal Equivalent
11111111	0.84	0	255
Error Test Patterns			
11111110	0.66	22	254
01111111	0.60	29	127
01111110	0.42	50	126

Table 4-A trial run of ADALINE might produce the results shown here. You can set the limits of acceptance so that any or all of the examples shown register as an acceptable input.

ments to prevent the weight value fromovershootingtheoptimumvalue.

You can visualize the learning process as a bowl, with the weight value changing rapidly as it comes down the side of it, then changing more slowly as it moves across the almost flat bottom of the bowl, as shown in Figure 6. Thus, the Delta Rule controls the amount of change in the weights for each iteration. It is:

$$W_{new} = \frac{W_{old} + \text{increment size} + \text{weighted input}}{\text{weighted input } 2}$$

The increment size is a control on the size of each step, which affects the number of iterations needed to train your neuron. A value of 0.50 is used by this program. If it is made much smaller, it would take much more time for the neuron to train. If it is very large, training will be very rapid. However, the value found in the analog neuron simulations will oscillate back and forth around its real value and not settleonit. Thisisparticularly true in multiple-neuron systems.

The weighted input is simply the input value times its respective weight value.

How ADALINE Solves the Problem

You now understand how ADALINE can be used to compare each individual header byte to the value it is expecting. ADALINE will report exact matches in cases where the output values for the two bytes match perfectly. In cases where they don't match, ADALINE will indicate how many bits are corrupted. By running the program, you can present bit patterns to ADALINE and see how much variation there is going to be for one- and two-bit errors. The result of a trial run is shown in Table 4.

Thisinformation from running the neuron can be used to devise a program which will try a strategy of accepting one bad bit in the 4-byte header coming from the radio receiver. We can accept it at any byte position by requiring that three of the four bytes have 0% error, and that the remaining byte not have more than a 35% error whencompared withitsproper value. The bit position will not significantly affect the deviation we accept.

This ability to locate the header in spite of a few dropped bits allows us to locate the start of the data stream and set the data pointer. The pointer can then be used to correctly pick out each data byte so it can be compared to the last two transmissions. If all three transmissions match on this byte, we will believe its value. The end result? A mobile robot which is properly radio-linked to its host computer using low-horsepower computing engines and state-of-the-art software techniques. All in all, it's a low-cost way to get a grip on slippery data. ✣

Scott Farley owns Tempus Consulting, 832 Brown Thrush, Wichita, KS 67212, 316-722-3068. He has 20 years experience in appliance design, working bo th the electronicand mechanical sides. Areas in which he consults include product definition *and development, product liability, codes and standards, computer systems and programming, electronic and electrical hardware, instrumentation, and data acquisition.*

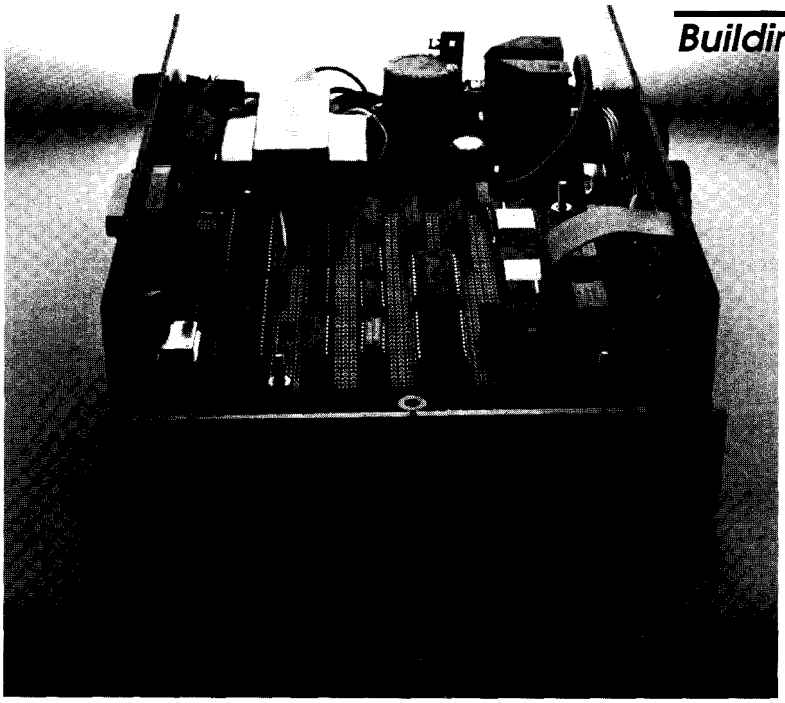
IRS

210 Very Useful
2 11 Moderately Useful
2 12 Not Useful

An Intelligent SCSI Data Acquisition System for the Apple Macintosh

Part 1
Building the Hardware

by John Eng



In the good 01' days of personal computing, there were legendary machines like the Apple II+-designed from off-the-shelf components, it had an architecture so open that you could get inside the computerwithoutusing tools. As a result, numerous plug-in peripherals became widely available for the Apple II bus, and building plug-in boards from scratch was the solution to many specific problems.

For the approximate price of an early Apple II+, today's designer can purchase a Macintosh Plus or, for a little more, a Macintosh SE. The Macintosh Plus has no slots for plug-in peripherals; the SE has one slot. Neither machine can be opened without special tools. Certainly, both machines support some pretty powerful software applications, but what is a hardware designer to do?

My favorite Apple II project was a plug-in board that performed 8-bit analog-to-digital (ADC) and digital-to-analog (DAC) conversions. With this peripheral, my Apple II could perform simple digital audio signal processing, storage,and reproduction. Given this experience with data acquisition on the Apple II, I thought it would be an interesting project to give my Macintosh Plus comparable abilities.

A number of data acquisition products for the Macintosh have recently become available. One of the most popular is the *MacRecorder,* manufactured by Farallon Computing (Berkeley, CA). The *MacRecorder* is an audio digitizer that transmits sampled sound data through one of the Macintosh's serial I/O ports. The maximum speed of the serial ports limits such devices to a digital resolution of 8 bits and a maximum sampling rate of 22 kHz. As demonstrated in the next section, these specifications result in a sound quality somewhat less than that of an inexpensive home stereo system. The *MacRecorder* is also an input-only device, using the Macintosh's built-in 8-bit audio DAC to output digitized sound samples.

While devices like the *MacRecorder* offer an economical way to give the Macintosh decent digitization capability, I wanted to design a device with a greater digital resolution and a faster sampling rate The result is the DAQ3000, an intelligent 12-bit data acquisition peripheral for the Macintosh Plus. The DAQ3000 is a stand-alone, microprocessor-based subsystem that is capable of a 28-kHz continuous sampling rate and communicates digitized data over a SCSI bus. With modifications, the DAQ3000 system can support 16-bit resolution while sampling at 28 kHz.

In this article, I will first explain my basic design goals and how they were implemented in the design. Then, after a more detailed description of the hardware, I will consider some future directions for the DAQ3000. Part 2 of this article will cover software design for both the DAQ3000 and the Macintosh host.

Because my old Apple II data acquisition peripheral was used almost exclusively for audio signal processing, I wanted to optimize my new Macintosh project specifically for audio signals. I also wanted my new project to have genuine audio fidelity, similar to that of an inexpensive stereo tape deck, which may have a frequency response of up to 12-14 kHz and a signal-to-noise ratio (SNR) of 60-70 dB. These two specifications parallel the two main characteristics

of any digital data acquisition system: conversion speed and digital resolution. According to the Nyquist sampling theorem, accurate digitization of a 12–14-kHz signal requires a sampling rate of twice that frequency, or 24-28 kHz. This sampling rate corresponds to an ADC conversion time of no more than 35-40 µs. With regard to the required SNR, we should choose an ADC with at least 12 bits of digital resolution, giving an output voltage resolution of approximately 1 part in 4096 and a corresponding SNR of 72 dB (SNR = $10\log_{10}[V^2_{signal}/V^2_{noise}]$ dB). By comparison, an &bit ADC would only provide a SNR of 48 dB.

Twelve-bit data conversion creates rather strict requirements on the acceptable system error or noise. For example, to achieve the resolution of 1 part in 4096 for a typical 12-bit ADC voltage range of 10 V (±5 V), the total system error and noise should not be greater than 1.2 mV. A sample-and-hold (SAH) device is certainly required here because even with a perfect audio system, a potentially important source of error originates from any significant change in the input signal during the ADC's conversion period. In the worst case of the 12-bit example above, the audio input signal to the ADC should not vary more than 1.2 mV over the entire conversion period of 35-40 µs. This requirement would severely limit the signal frequencies accurately measurable by the ADC without a SAH.

The DAQ3000's analog circuits are based on three Harris devices. The HI-574A is a 12-bit ADC with a maximum conversion time of 25 µs, giving a maximum possible sampling rate of 40 kHz. The HI-5680V is a 12-bit DAC with a maximum settling time of 1.5 µs. The HA-5320 is a SAH chip with a maximum voltage droop of 0.5 µV/µs in the hold mode. Over the 25-µs conversion time of the 574A ADC, the maximum total voltage droop of the 5320 SAH is 12.5 µV, which is well within the 1.2-mV requirement.

With the exception of hard disk drives, most currently available Macintosh peripherals communicate data through one of the Macintosh's two RS-422 serial ports. These serial ports allow a maximum 8-bit transfer rate of about 22 kHz, which is much slower than the 56-kHz transfer rate needed by the DAQ3000 (28-kHz sampling rate x 2 bytes per 12-bit sample). To accomplish the high-speed data transfer, the DAQ3000 communicates instead through the SCSI bus port available on the Macintosh Plus and later models.

SCSI stands for Small Computer System Interface, a standard parallel interface for high-speed intelligent peripherals such as hard disks and

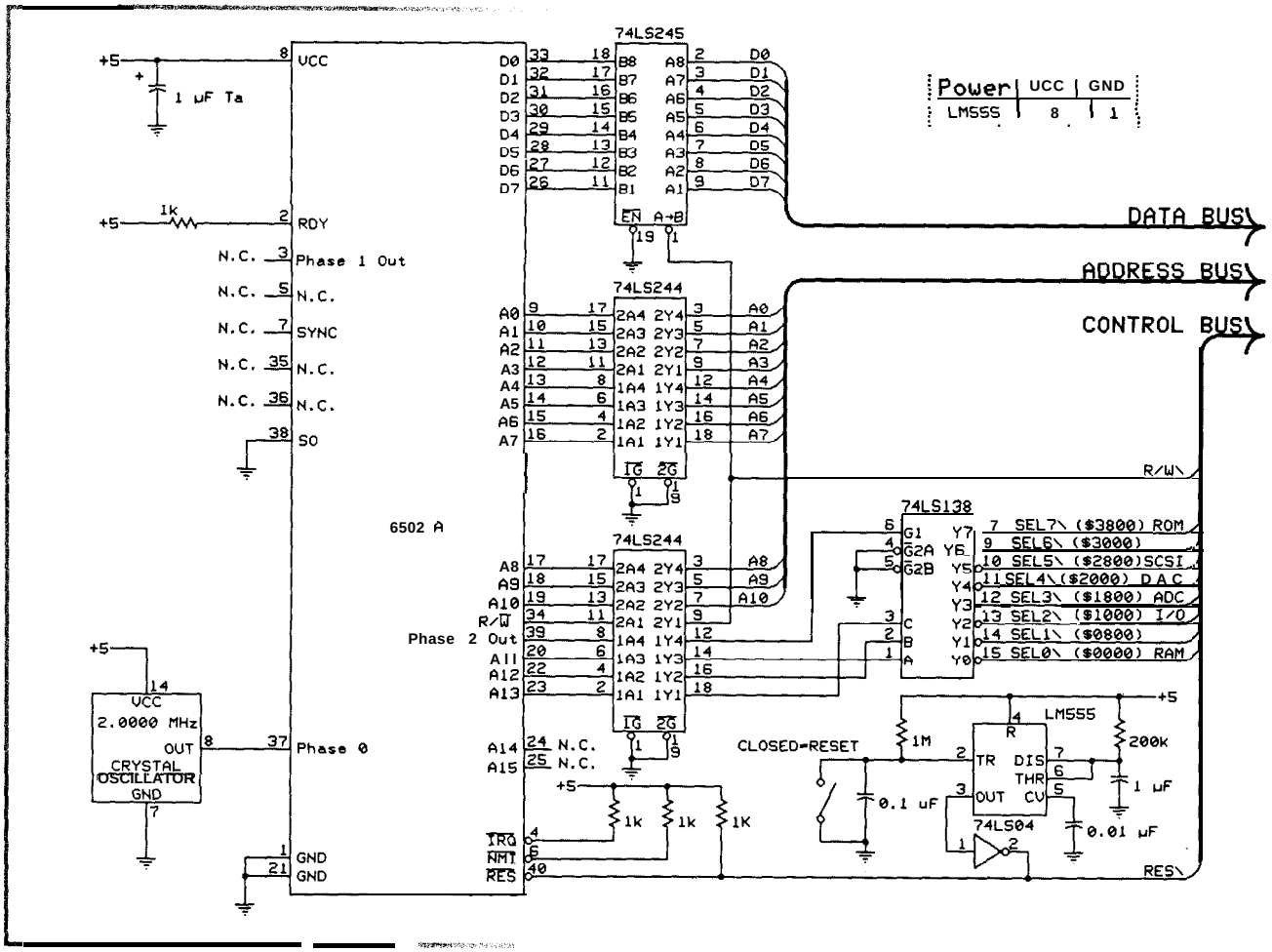


Figure *1 --The main processor section of the DAQ3000 includes a 6502A microprocessor and chip selects that break up the 64K address space into eight sections.*

streaming tape drives. The SCSI standard was developed by the American National Standards Institute (ANSI) and includes standard protocols allowing asynchronous data transfers atgreater than 1 Mbyte/s over a daisy-chained bus. The DAQ3000 uses the NCR 5380 SCSI controller, the same SCSI chip used in the Macintosh.

On With the Design

Use of SCSI protocols requires that the DAQ3000 be "intelligent." As a result, the DAQ3000 was designed as a stand-alone microcomputer subsystem. At the system's center is a 6502A microprocessor, running at 2 MHz, surrounded by bus drivers and address decoding logic (Figure 1). A 74LS245 bidirectionally buffers the data lines; the 6502's address and bus control lines are buffered by 74LS244s.

The 6502's address space is divided into eight 2K-byte areas, six of which are used by the system: RAM, System I/O, ADC, DAC, SCSI, and ROM. Because the most-significant two address bits of the 6502 are not used, the eight 2K-byte areas repeat themselves four times in the 64K-byte total address space of the 6502. This partial address decoding scheme was chosen to minimize the chip count. Address decoding is done by a 74LS138, which generates a selection signal (SELn\) corresponding to each address area.

The RAM and ROM (Figure 2), and System I/O (Figure 3) sections are each designed around one major chip. The 6116P-2 2K-byte static RAM must be configured as address area 0 because the first 512 bytes of the 6502's address space must be present for the zero-page addressing mode and the system stack. The 2816A-25 2K-byte

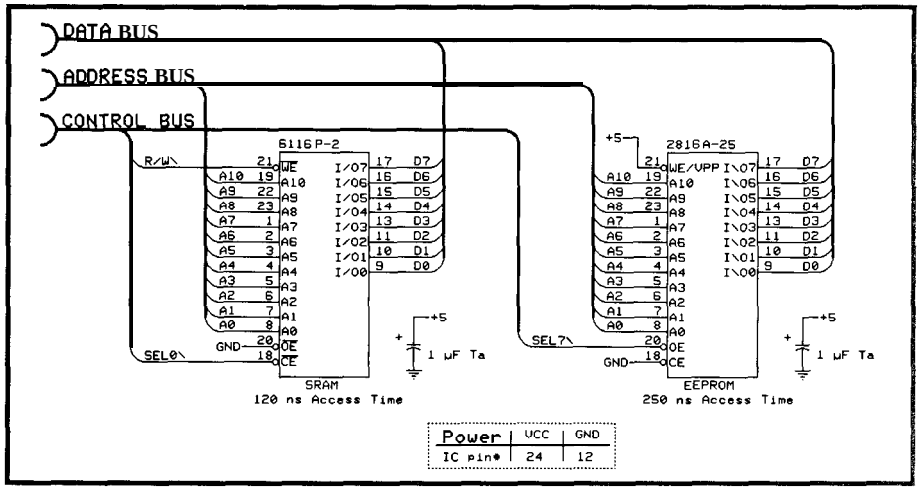


Figure 2—The 2K bytes of static RAM provide plenty of temporary storage and 2K bytes of EPROM make program development easier.

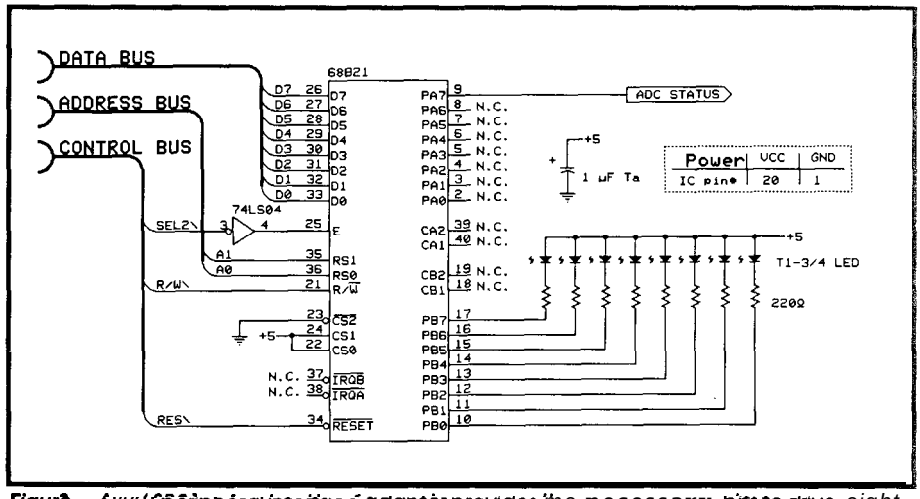


Figure 3—A 68B21 peripheral interface adapter provides the necessary bits to drive eight LEDs and to check the status of the analog-to-digital converter.

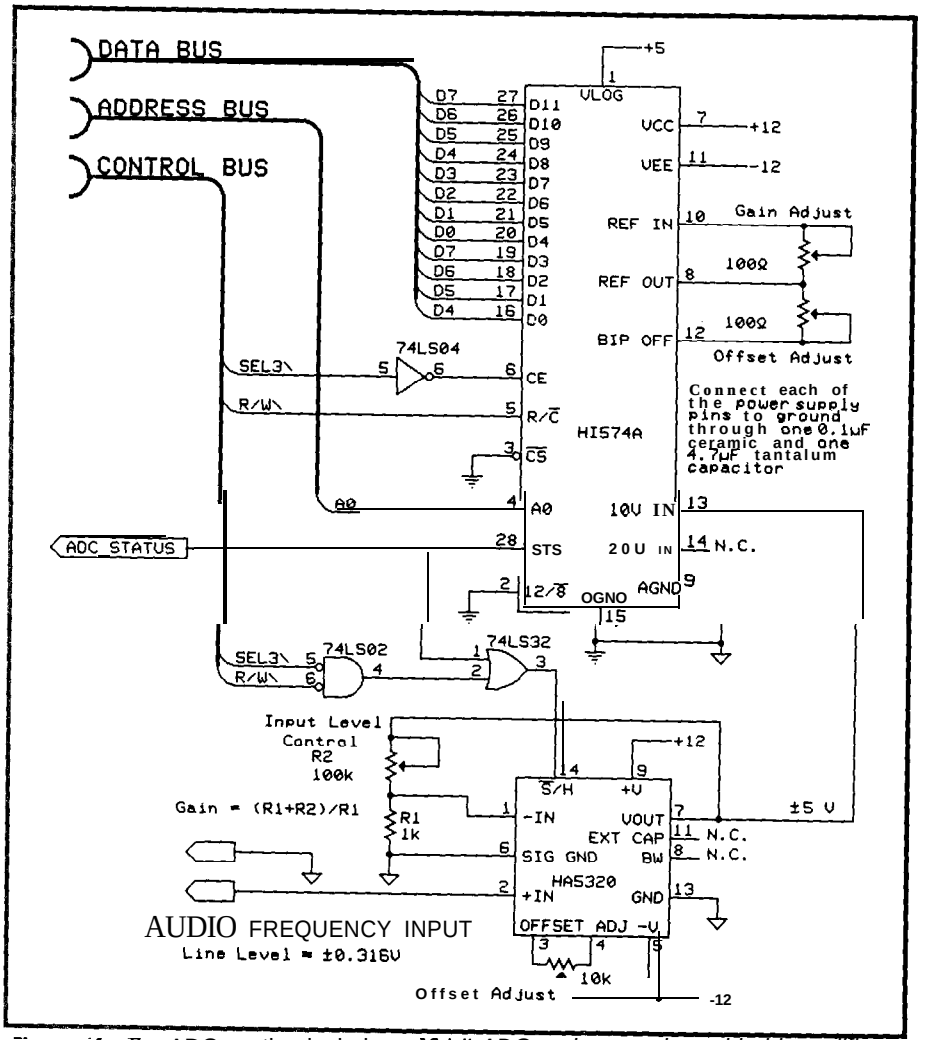
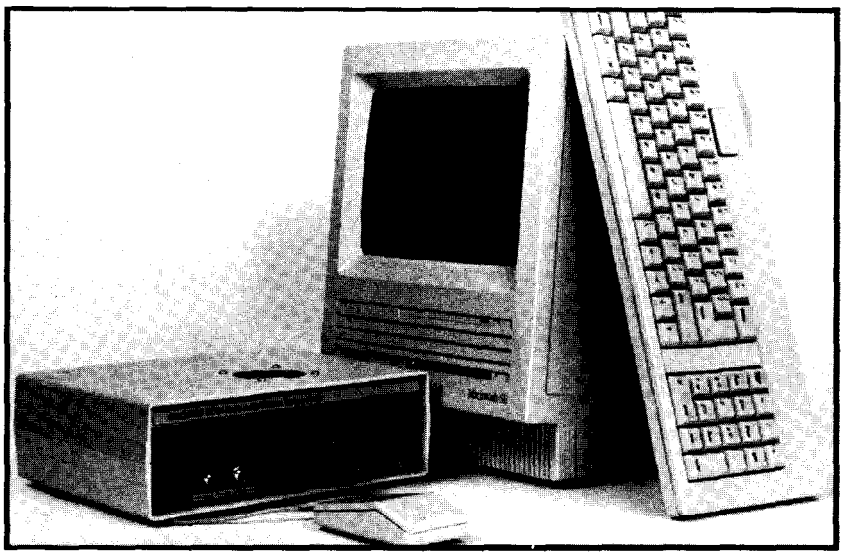


Figure 4—The ADC section includes a 12-bit ADC and a sample-and-hold amplifier.

EEPROM (electrically erasable) must be configured as address area 7 because the power-on reset vector is located in the last bytes of the 6502's address space. The 2816 EEPROM is pin-compatible with the more common 2716 EPROM, but the 350-ns access time of commonly available 2716s is too slow for the DAQ3000 timingrequirements. The System I/O section uses the 68B21 peripheral interface adapter (PIA), which contains two 8-bit parallel I/O ports. One port is used to drive eight indicator LEDs, and the upper bit of the other port allows the microprocessor to monitor the status signal of the ADC section.

The ADC section (Figure 4) consists of two main chips: a SAH amplifier and the ADC itself. When there is a logic 1 at its S/H input, the SAH amplifier holds an instantaneous sample of the incoming signal for the ADC to digitize. Here, the SAH is configured as a noninverting amplifier to convert line-level (±0.316 V) audio signals into the ±5-V range of the ADC.

The conversion process begins when the microprocessor performs a write operation to the ADC's address space, causing both SEL3\ and R/W\ to go low. The HI-574A responds with a logic 1 on its STS output and begins the conversion based on the SAH's output (now held). A combination of NOR (74LS02) and OR (74LS32) gates ensures that the SAH receives a hold signal from the moment SEL3\ and R/W\ go low until STS returns to logic 0 at the end of the conversion. The microprocessor can detect the end of conversion by checking the HI-574A's STS signal through port A, bit 7 of the 68B21 PIA. When the conversion is finished, the microprocessor reads the 12-bit result as two consecutive bytes in the ADC section's address space.



The TTL logic of the DAC section shown in Figure 5 is a little more complicated, but its operation is simpler. To send a 12-bit value, the upper and lower bytes are written into consecutive memory locations. The three 74LS374 latches form a double-buffering scheme that ensures the first microprocessor write operation does not affect the previously latched 12-bit value. All 12 bits to the DAC chip change simultaneously and are latched after the second byte is written by the microprocessor. Latching control signals for the three 74LS374 latches are generated by three OR gates (74LS32) and an inverter (74LS04).

All three analog components (SAH, ADC, and DAC) have separate gain and offset adjustments. The offset adjustments ensure that an input of 0 V to the ADC and an output of 0 V by the DAC correspond to the 12-bit digital representation for OV ($800 for the 574A ADC, $7FF for the 5680V DAC). The gain adjustments are normally used to calibrate the converters so that the maximum digital value ($FFF) corresponds accurately to the stated full-scale voltage. The gain adjustment on the SAH controls the amplification of the incoming analog audio signal. The gain adjustment on the ADC should be set to give a nominal full-scale input of ±5 V. The level of the outgoing analog audio signal is controlled by a potentiometer across the output of the DAC. The separate gain adjustment on the DAC should be set to give a nominal full-scale output of ti.5 V when the Output Level Control is setatmaximum. For precision voltage applications, the Input and Output Level Controls should be eliminated and the SAH should be configured as a simple voltage follower (in Figure 4, remove R1 and R2 and connect pin 1 of the SAH directly to pin 7).

The last section to be discussed is the SCSI section shown in Figure 6. Four OR gates (74LS32) and two inverters (74LS04) provide various versions of the SEL5\ pulse to the 5380's four chip selection lines (CS\, DACK\, IOR\, and IOW\). The IOR\ and

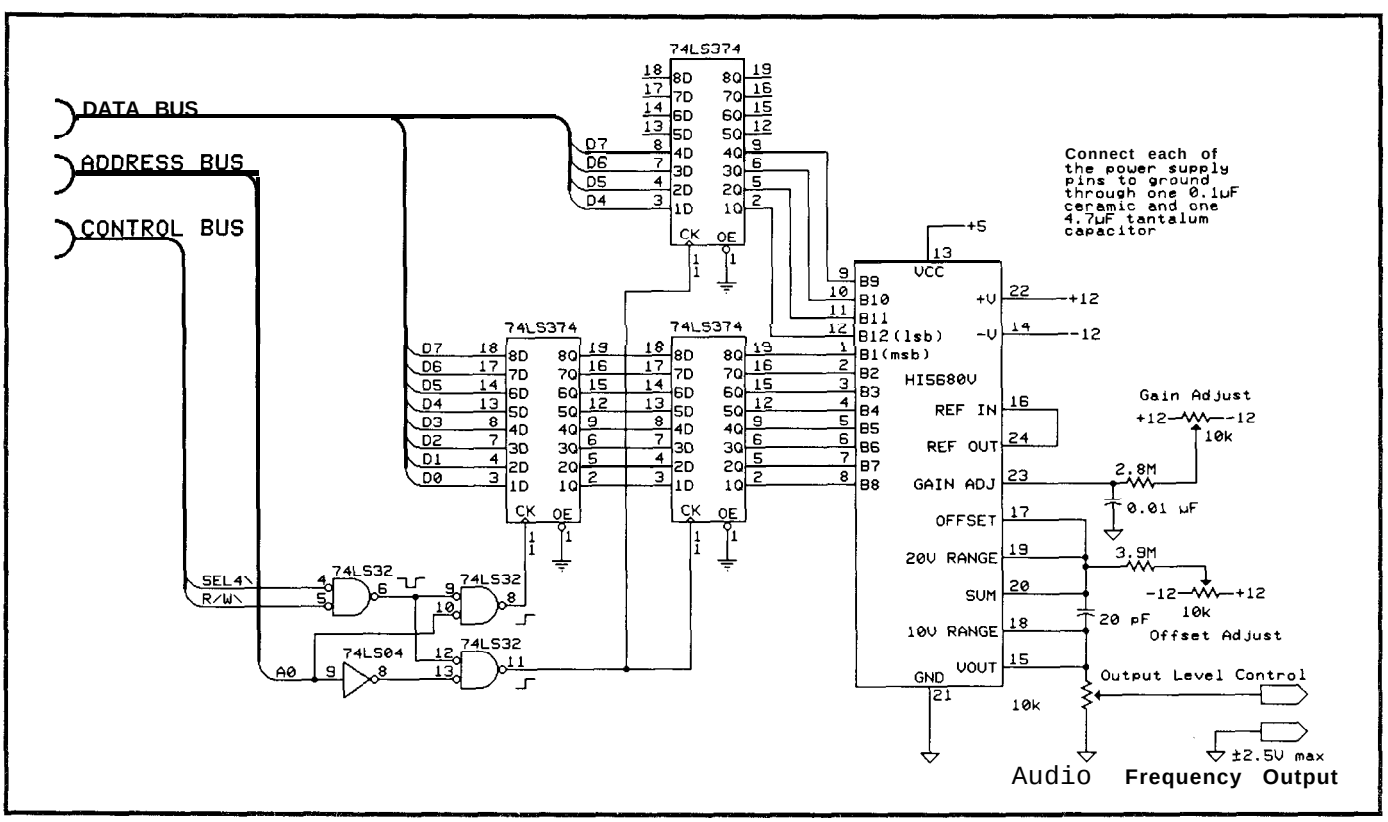


Figure 5—*The DAC section includes a 12-bit digital-to-analog converter* plus *double buffering to allow all 12 bits to be set simultaneously.*

IOW\ signals control the direction of microprocessor data flow. The CS\ signal selects one of the 5380's eight internal registers, as specified by AO-A2 (address offsets $00-$071. The DACK\ line is normally generated by DMA hardware, but here the microprocessor simulates a DMA controller by using address decoding logic to generate the DACK\ signal. This configuration is known as the 5380's *pseudo-DMA operation mode.* In this mode, the 5380 is "selected" using the DACK\ line instead of the CS\ line, which **occurs** on microprocessor reads and writes to address offsets $08–$0F (A3 high).

The 5380's eighteen SCSI signal pins are open-collector outputs and can be directly connected to the SCSI bus of a Macintosh Plus, SE, or II. The network of resistors shown in Figure 6 provides one standard bus termination for each line. There has been some confusion on how many SCSI terminators are needed for the Macintosh. According to Apple, only one terminator should be used when there is only one SCSI device connected to the Macintosh. If there are two or more SCSI devices, then two terminators are required, one at each end of the daisy-chained bus. For a given hardware configuration, it is not always easy to determine how many SCSI terminators are present because they are often built inside of the SCSI equipment. The Macintosh Plus has no built-in terminator; the SE and II models each have one built-in terminator if an internal hard disk is present. All of this means that the DAQ3000 should work, as shown, with a Macintosh hardware configuration that has 0 or 1 SCSI device(s) (including any internal hard disks). This probably applies to most Macintosh Plus/SE/II systems. For Macintosh systems with a properly termina ted SCSI bus already containing two or more devices, the DAQ3000 should work if it is inserted in the middle of the daisy chain. In this case, the DAQ3000's internal bus terminator should be disabled by removing the two jumpers shown in Figure 6.

The 6502 microprocessor accesses the ADC, DAC, System I/O, and SCSI sections as memory-mapped devices. A programmer's model for the DAQ3000 is shown in Figure 7. A moredetailed descriptionof the68B21 and 5380 hardware registers can be found in the manufacturers' data sheets. Note that the 574A ADC and 5680V DAC conversion codes are reversed with respect to their polarity so that a digital value of $000 represents a full-scale negative voltage on the ADC, while the same $000 value represents a full-scale *positive* voltage on the DAC. This polarity reversal should not affect audio applications but can be compensated by adding invertersbetween the 74LS374 latches and the 568OV DAC in Figure 5.

Construction Notes

In order to achieve the theoretical resolutionof its 12-bit ADCand DAC, the DAQ3000 must be constructed to minimize audio frequency noise. Ideally, printedcircuit construction with a ground plane should be used. However, I chose wire-wrap for its ease in prototype modifications, even though there may have been an increase in noise. (The ground plane is still a good idea with wire-wrap.)

Whatever the construction method, proper power supply and grounding practices are important. The goal is to prevent the analog circuitry from detecting spurious signalsgenerated by thedigitalcircuitry's logic switching. The best solution is keeping the analog and digital components as physically and electrically independent as possible. Separate voltage supply lines should be used, and the digital and analog grounds should be connected together at only one point. Decoupling capacitors should be placed at the power supply pins of each chip. A 1–5-µF tantalum capacitor in parallel with a 0.1-µF ceramic capacitor is a typical combi-

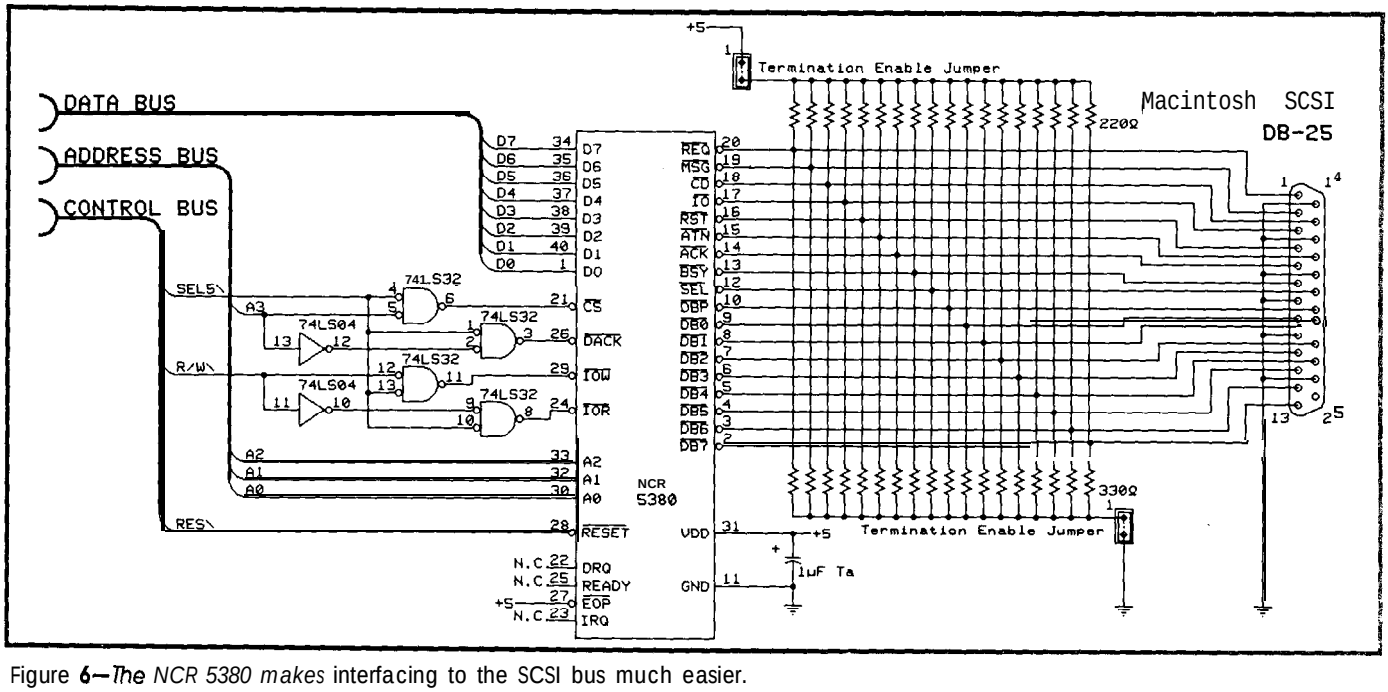


Figure 6—*The NCR 5380 makes* interfacing to the SCSI bus much easier.

System I/O (Module 2)

$1000	Peripheral/Data Direction Register A
$1001	Control Register A
$1002	Peripheral/Data Direction Register B
$1003	Control Register B

*ADC (Module 3)**

	Read	Write
$1800	Read 8 MSBs	Initiate 1 P-bit conversion
$1801	Read 4 LSBs & 4 trailing zeros	Initiate 8-bit conversion

*DAC (Module 4)***

	Read	Write
$2000	No operation	Write 8 MSBs into buffer
$2001	No operation	Write 4 LSBs (4 trailing 0s) & latch

SCSI (Module 5)

	Read	Write
$2800	Current SCSI Data Register	Output Data Register
$2801	Initiator Command Register	
$2802	Mode Register	
$2803	Target Command Register	
$2804	Current SCSI Bus Status Register	Select Enable Register
$2805	Bus and Status Register	Start DMA Send
$2806	Input Data Register	Start DMA Target Receive
$2807	Reset Parity/Interrupt	Start DMA Initiator Receive
$2808	Read Pseudo-DMA Data	Write Pseudo-DMA Data

* ADC conversion codes: $000 = –5.0000 V, $800 = 0.0000 V, $FFF = +4.9976 V
** DAC conversion codes: $000 = +2.4988 V, $7FF = 0.0000 V, $FFF = –2.5000 V

Figure 7—*The DAQ3000's 6502 accesses the ADC, DAC, System I/O. and SCSI sections as memory-mapped devices, as shown in this programmer's model.*

nation used on IC power supply lines. The power supply itself should have a ripple of less than 1 mV while supplying the maximum current requirements (+5V @1.0–1.5A, +12V @0.1A, and -12V @0.1A). Finally, digital signal lines can easily couple capacitively toanalogcircuitry. Thiscouplingmay be prevented by electrostatic shields surrounding wires, connectors, and components carrying audio signals. Such shielding should be connected to the analog ground.

Future Directions

As I developed the project, I realized that the DAQ3000's design has great potential as the basis for an intelligent, general-purpose SCSI controller. The DAQ3000's functional capabilities can be changed/augmented by simply substituting/adding hardware (e.g., additional sections supporting TTL digital I/O, additional ADCs, more memory, etc.).

For some applications, it may be necessary to hook up an external modem or terminal. The current system has no serial ports, but this could be changed, of course, by adding another section. I would suggest using the 6551A asynchronous communication interface adapter (ACIA), a 65xx-series device with an internal baud rate generator.

Although they *are* good chips, there is nothing "special" about the Harris analog devices used in this project; many other manufacturers offer devices with similar specifications. There is also nothing immutable about the DAQ3000's 12-bit resolution. Eight-bit converters could be used if a faster sampling rate is desired at the expense of a lower digital resolution. If 16-bit converters are used, a 4-bit increase in dynamic range can be obtained with no change in the sampling rate. Little, if any, software changes would be required because the DAQ3000 software already handles all data as 16-bit values (four bits are currently "wasted").

An essential analog component not discussed so far is the audio filter. For high-quality audio sampling, a low-pass filter must be placed in front of the ADC input to filter out frequencies too high to be digitized, as dictated by the Nyquist theorem. For the DAQ3000's 28-kHz sampling rate, the filter should have a sharp cutoff at approximately 14 kHz. Without low-pass audio filtering, frequency components in the incoming analog signal higherthan kHz will reach the ADC and be digitized as *aliasing* noise with frequency components below 14 kHz. Aliasingnoise causes an audible whistling or "grunge" when the sample is played back. The vast subject of analog filter design and construction is beyond the scope of this article. I will mention, however, that the newer *switched-capacitor* designs seem well-suited for the sharp cutoffs required by digital sampling. *[Editor's Note: For moye information on filter design, see the sidebar on page 43.1*

Next Time: Software Design

The final DAQ3000 system requires three pieces of software. The first piece, `DAQ3000.ASM`, is written in 6502 assembly language and resides in the DAQ3000's EEPROM. This code is responsible for initializing the DAQ3000 and handling all SCSI data transfer requests from the Macintosh. The second piece of code, `SCSIFast.asm`, contains fast, Pas-Cal-callable SCSI transfer rou tines written in 68000 assembly language for the Macintosh. The third piece of soft-

In any real-time digitizing system where a dynamic analog signal is being recorded, the highest frequency that can be recorded is limited by the sampling rate of the system. This limit is described by the Nyquist theorem, which states that the sampling rate of the system must be at least twice the highest frequency to be sampled. For example, a system which digitizes the input signal at a rate of 30,000 samples per second is capable of recording frequencies up to 15 kHz.

In order to ensure that the incoming signal is within the frequency limits of the digitizing system, the input signal is usually sent through a low-pass filter. While passive filters (those containing just passive components such as resistors and capacitors) are the simplest to build, they usually don't have a sharp enough cut-off to be useful in digital sampling applications. Instead, active filters are used to attain higher performance with a smaller parts count. A simple active filter using a resistor, capacitor, and op-amp is shown in Diagram I.

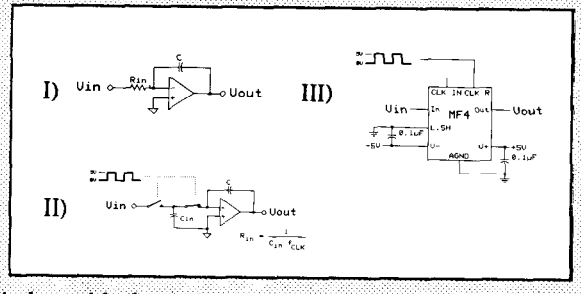
Such a circuit has a fixed cut-off frequency, determined by the combination of R and C. With a little bit of calculation to determine component values, this circuit works just fine in an application where the cut-off frequency is known and fixed.

In applications where control over the cut-off frequency is to be given to a processor, a switched-capacitor filter may be used in its place. A switched-capacitor filter takes advantage of the fact that a capacitor which is alternately charged and discharged looks the same as a resistor, with the resistance depending on the capacitance and the switching frequency. Diagram II shows the same active filter as above with the resistor replaced by an equivalent switched-capacitor circuit.

Because of their flexibility and overall usefulness, switched-capacitor filters are available on chips that require the addition of just a few capacitors to make everything work. One such chip is the National Semiconductor MF4, known as a fourth-order switched-capacitor Butterworth low-pass filter. Diagram III shows just how easy it is to use the MF4 in a circuit.

While the MF4 requires a clock input for setting the cut-off frequency, such clocks are usually easily obtained in computer-based circuits where switched-capacitor filters are often found. The chip also includes circuitry to make constructing an RC oscillator easier, so an outboard clock source isn't always needed.

A National Semiconductor Linear 2 Databook should be consulted for details on how to calculate the proper clock input frequency for the desired cut-off frequency, but most of the filter design work has been done for you. Filter design itself is a subject that easily fills books; however, armed with the MF4 and a few components, you can quickly build a usable low-pass filter with minimum fuss.



ware, **SCSIMover.pas,** is a Macintosh application written in Lightspeed Pascal that performs data acquisition, storage, and reproduction. *[Editor's Note:* Softwarefor *this article is available for downloading from the Circuit Cellar BBS or on* Circuit *Cellar INK Software On Disk #9. For downloading and ordering information, see page 78.1* In the second part of this article, I'll explain the design of these three software components in detail.

John Eng recently completed a research fellowship with the Howard Hughes Medical Institute and is currently a senior medical student at the University of Wisconsin. His interest *in microcomputing began with the purchase* of an *Apple II in 1981.*

IRS

2 13 Very Useful
2 14 Moderately Useful
215 Not Useful

References

American National Standards Institute. *Small Computer System Interface (SCSI).* Document ANSI X3.131-1986, American National Standards Institute, New York, 1986. [A formal statement of the SCSI standard, this document also includes practical information.]

Apple Computer, Inc. *Inside Macintosh Volume IV.* Addison-Wesley, Reading, MA, 1986. [This book includes chapters on SCSI hardware and software on the Macintosh Plus computer.]

Apple Computer, Inc. "SCSI Bugs." *Macintosh Technical Notes, No. 96.* Apple Programmer's and Developer's Association, Renton, WA. [Outlines some quirks in the Macintosh implementation of SCSI.]

Chamberlin, Hal. Musical *Applications of Microprocessors.* Hayden Books, Hasbrouck Heights, NJ, 1980. [Covers basic digital sampling topics such as the Nyquist theorem, aliasing, filtering, and much more.]

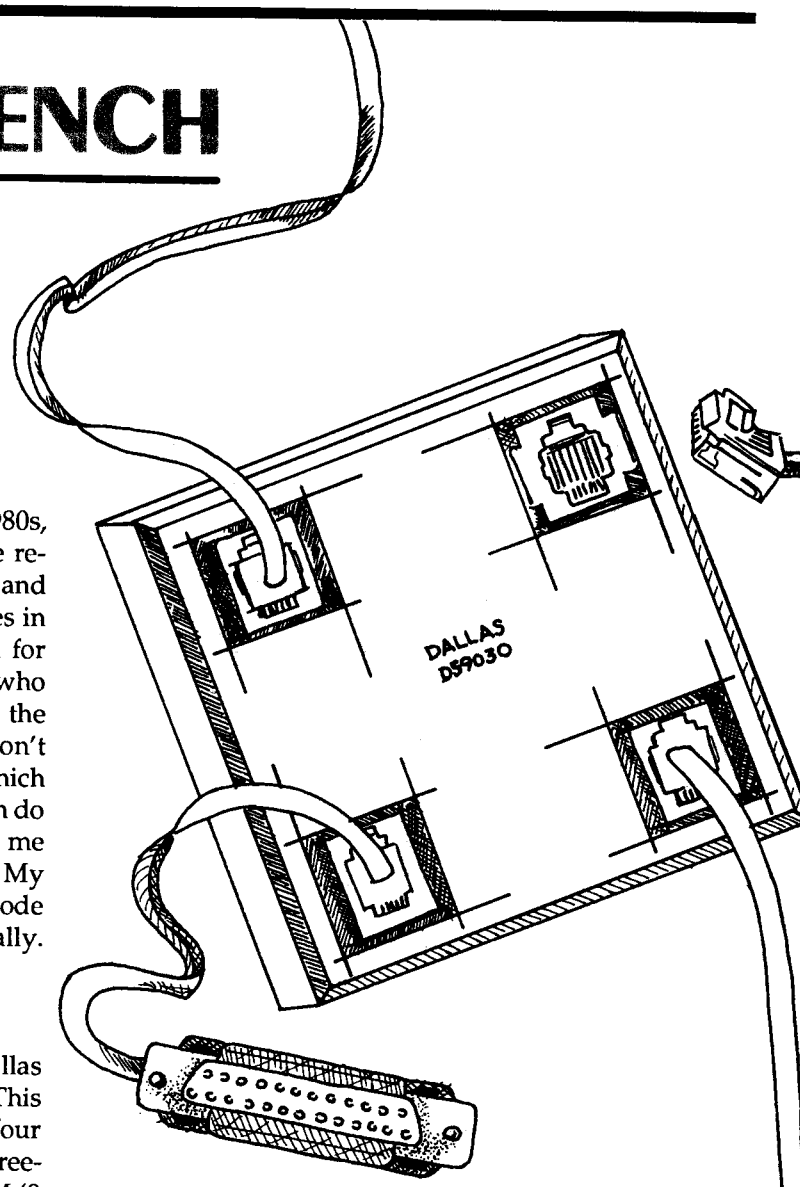
Ciarcia, Steve. "Adding SCSI to the SB180 Computer." Part 1, *Byte,* Vol. 11, No. 5 (May 1986) and Part 2, *Byte,* Vol. 11, No. 6 (June 1986). [These articles include a general introduction to the SCSI protocol.]

Fischer, C. R. "Experimenting with Brickwall Filters." *Electronic Musician,* January 1989. [Presents a design for a low-pass, sharp cutoff filter using switched-capacitor technology.]

NCR Corporation. Standard *Products Data Book.* NCR Microelectronics Division, Colorado Springs, 1988. [This data book contains detailed information on programming the 5380 SCSI chip, actually including some sample 6502 code. Be careful-the 6502 code contains at least one error.]

The Invisible Net

by Jeff Bachiochi



"Networking," the catch phrase of the late 1980s, has penetrated the CIRCUIT CELLAR INK fortress. We replaced our "SneakerNet" with networking hardware and software to link the IBM ATs and Apple Macintoshes in production and editorial. This eliminated the need for most removable media, file conversion, and guessing who had the latest revision of a file. My problem is that the networking hardware sits in a slot on the ATs, and I don't have a spare expansion slot! Trying to determine which board to remove is like deciding which finger you can do without! Wrestling with the "boardectomy" left me wondering about networks that don't rob you of a slot. My solution won't do all the fancy tricks of a $3,000-per-node LAN, but it will let you swap files easily and economically.

The Heart of the Matter

The heart of the Invisible Net is a new chip from Dallas Semiconductor, the DS2015 Quad-Port Serial RAM. This 18-pin CMOS package, shown in Figure 1, contains four bidirectional clocked serial transmission ports plus a three-byte protocol register and a nine-byte quad-port RAM (8 message bytes + 1 check byte) for each port. Each serial port has read/write access to its own message area. In addition, it can read the other three ports' message areas. All four ports can read any message area simultaneously, reducing arbitration concerns to write only. Figure 2 is a block diagram showing all functions of the DS2015.

Each clocked serial port consists of a RST\ line (enables communication), a D/Q line (actual data), a CLK line (indicates data good), and an M\ line (message waiting).

The RST\ pin controls communication. Raising it to a logic 1 signals the start of a transmission sequence. Lowering RST\ terminates the communication. Since this network doesn't include hardware data correction, terminating messages prior to the end of a transmission may cause unexpected results.

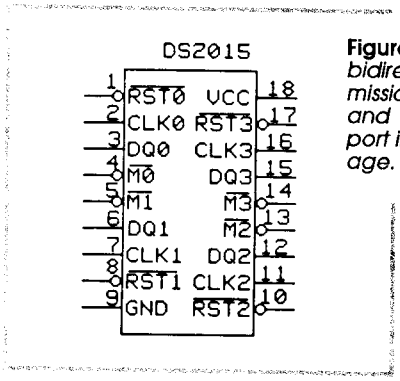


Figure 1—*The DS2015 puts four bidirectional clocked serial transmission ports, a protocol register, and quad-port RAM for each port into an 18-pin CMOS package.*

The CLK pin determines the rate of transmission. (The DS2015 has a maximum clock rate of 4 MHz.) The CLK cycle consists of a falling and a rising edge. Data written to the DS2015 must be valid prior to the rising edge of a CLK cycle. Data read from the DS2015 is valid at the falling edge of a CLK cycle. The CLK line is tristated whenever RST\ is logic 0 (no communication).

The D/Q pin is a bidirectional line for data transmission. Drivers at each end are open-collector which allow data to be transmitted bidirectionally. It is tristated whenever RST\ is logic 0.

The M\ pin is a hardware interrupt line which signals the host a message is ready to be received. The M\ line will go to a logic 0 indicating an awaiting message after the port has gone inactive.

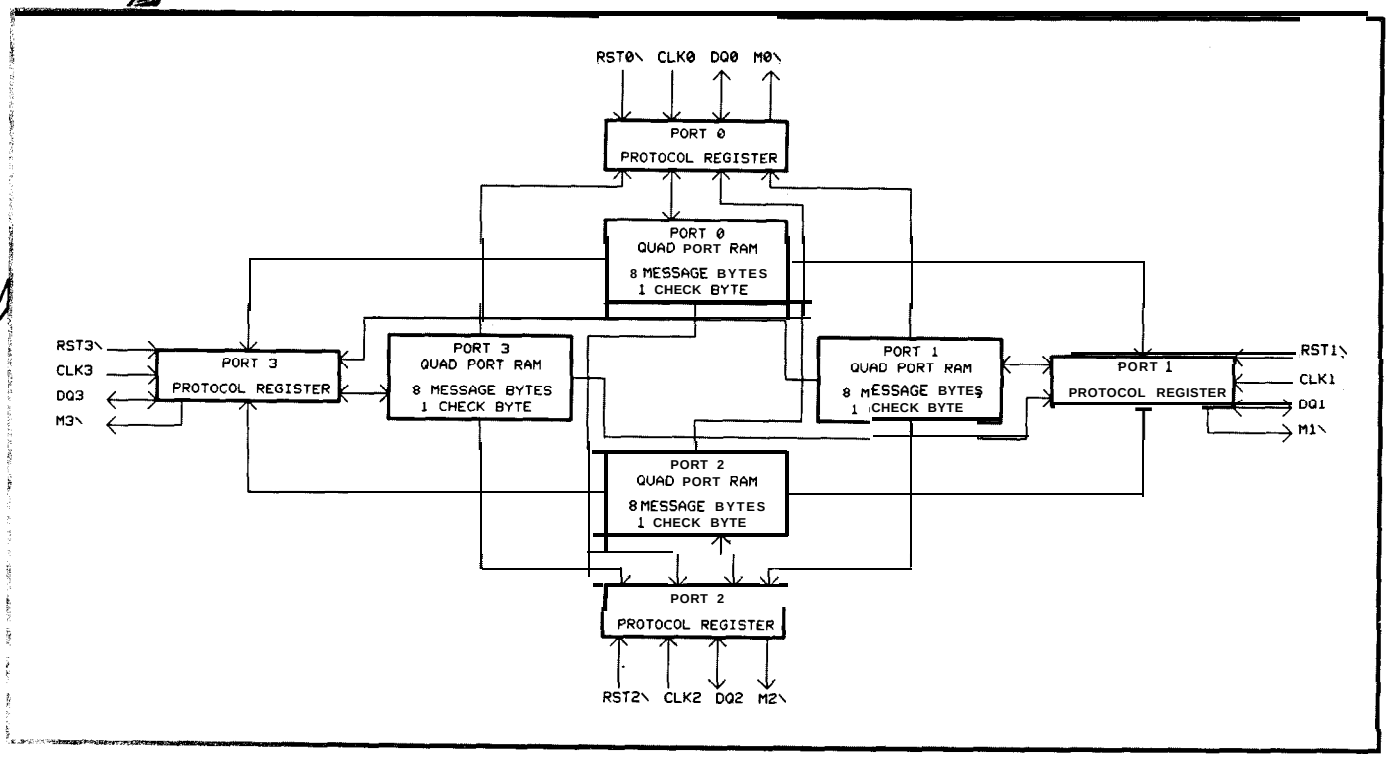


Figure 2—*The DS2015* has a maximum theoretical throughput of *1.52* Mbps at its top clock *speed* of 4 MHz.

Let the Data Flow

You gain access to the quad-port RAM message area by sending a port-select identification byte (8 bits). Each of the four ports has a specific ID value and will disregard any transmissions that do not start with the correct ID. The 8-bit ID codes are as follows:

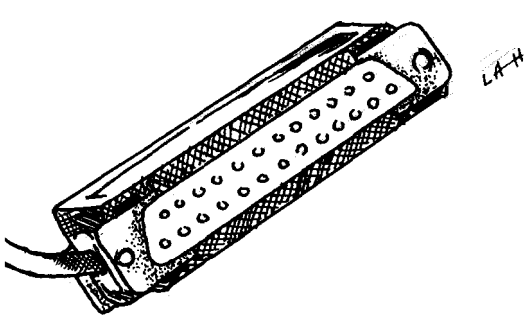
```
PORT0  =      11001011 (CBH)
PORT1  =      11011011 (DBH)
PORT2  =      11101011 (EBH)
PORT3  =      11111011 (FBH)
```

All bits are sent from least-significant bit to most-significant bit. If the correct ID is received, the port will move into the next phase of the protocol, sending the message center info to the host computer. The Message Center Information Byte is read by the host and contains two pieces of information. The lower nibble of the message center byte indicates to which ports you have sent a message that has not yet been received. The upper nibble indicates any messages ready for you to receive from other ports. All messages sent by the other ports to you should be read prior to sending any message to prevent a "message jam" from occurring. The message center byte format is as follows:

```
Message Available From:          Message Sent To:
Port0  Port1  Port2  Port3    Port0  Port1  Port2  Port3
(D7)   (D6)   (D5)   (D4)     (D3)   (D2)   (D1)   (D0)
------Upper  Nibble-------    -----Lower  Nibble-
```

Communication can stop at this point if there are no messages to receive and none to send. If you do continue, the port will move into the execution phase. The Execution Code is sent by the host and again contains two pieces of information. The lower nibble indicates which action is to follow: a read, a write, or a write with "more to follow." The upper nibble indicates which port the message will be read from or sent to. Messages can be read from only one port at a time but may be sent to all ports at once. The execution code byte format is as follows:

```
   Message Destination:              Action Code:
Port0  Port1  Port2  Port3          ------- Code -------
 (D7)   (D6)   (D5)   (D4)        (D3)  (D2)  (D1)  (D0)
-Upper  Nibble-----               ---Lower  Nibble-
                  0000 (OH) = read
     Action codes: 1000 (8H) = write
                   1100 (CH) = write with more to come
```



If you have chosen to read a message, the next eight bytes will be a message sent by the DS2015. These are read from the source port's message area and sent to the host computer. A ninth byte containing status information can

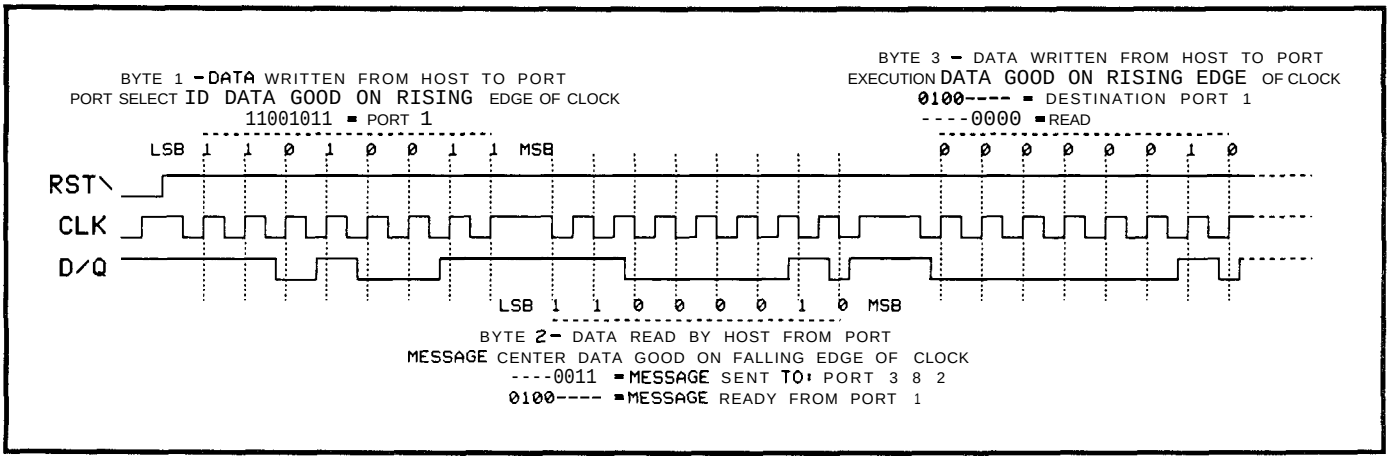


Figure 3—*The* Invisible *Net system requires 80* bits *to transmit 64 bits of message data, and 88 bits (including the check byte) to receive the same 64 bits.*

be optionally read. The status is a message indicating corrupted data, good data, or good data with more coming. A corrupted data status tells the host that the sending port has overwritten the message before it was read and should be discarded. Good data with more coming lets the host know that there is more to the message. Legal check byte values include:

```
Corrupted Data                = 10101010 (AAH)
Good Data                     = 01010101 (55H)
Good Data With More Coming = 01011010 (5AH)
```


If you have chosen to write a message, the next eight bytes will be sent from the host computer; no ninth byte is sent. When the last bit of the eighth byte is sent, the "message from" bit is set to a "1" in the target port's message center, indicating a message is ready. When the target port reads the last bit of your message, the "message to" bit in your message center is reset to a "0," indicating the message has been received. If you write another eight-byte message before the last message was read by the target port, a corrupted data code will be sent to the target's check byte. The data transmission sequence is shown in Figure 3.

The system requires 80 bits to transmit 64 bits of message data and 88 bits (which includes the check byte) to receive it. The time required is based on the CLK speed of the host for the send portion and the CLK of the target for the receive portion. If we assume the absolute maximum speed of 4 MHz for both sending and receiving, each bit takes 25 nanoseconds. Therefore,

$$\begin{aligned}\text{Throughput} &= (0.000000025 \times 80 + 64) \\ &\quad + (0.000000025 \times 88 + 64) \\ &= 31.25\ \text{ns/bit} + 34.38\ \text{ns/bit} \\ &= 65.63\ \text{ns/bit or } 1.52\ \text{Mbps}\end{aligned}$$

Realize, though, that this is a theoretical limit that you won't even approach in practical applications. Demo software written for MS-DOS machines, which allows viewing target directories and copying and erasing of target files, runs at about 5,000–10,000 bps depending on the speed of the machines being used. Obviously, the speed of Invisible Net is not its main selling point!

Invisible Hardware

Thisnetworkconnects inan "invisible" manner when it receives a "data good" strobe. Data can be placed on the port and as long as the strobe line is never cycled this data will not affect a printer in any way. Unfortunately, 8-bit data cannot be read in through the printer port quite as easily, since the printer port is a latched output-only port.

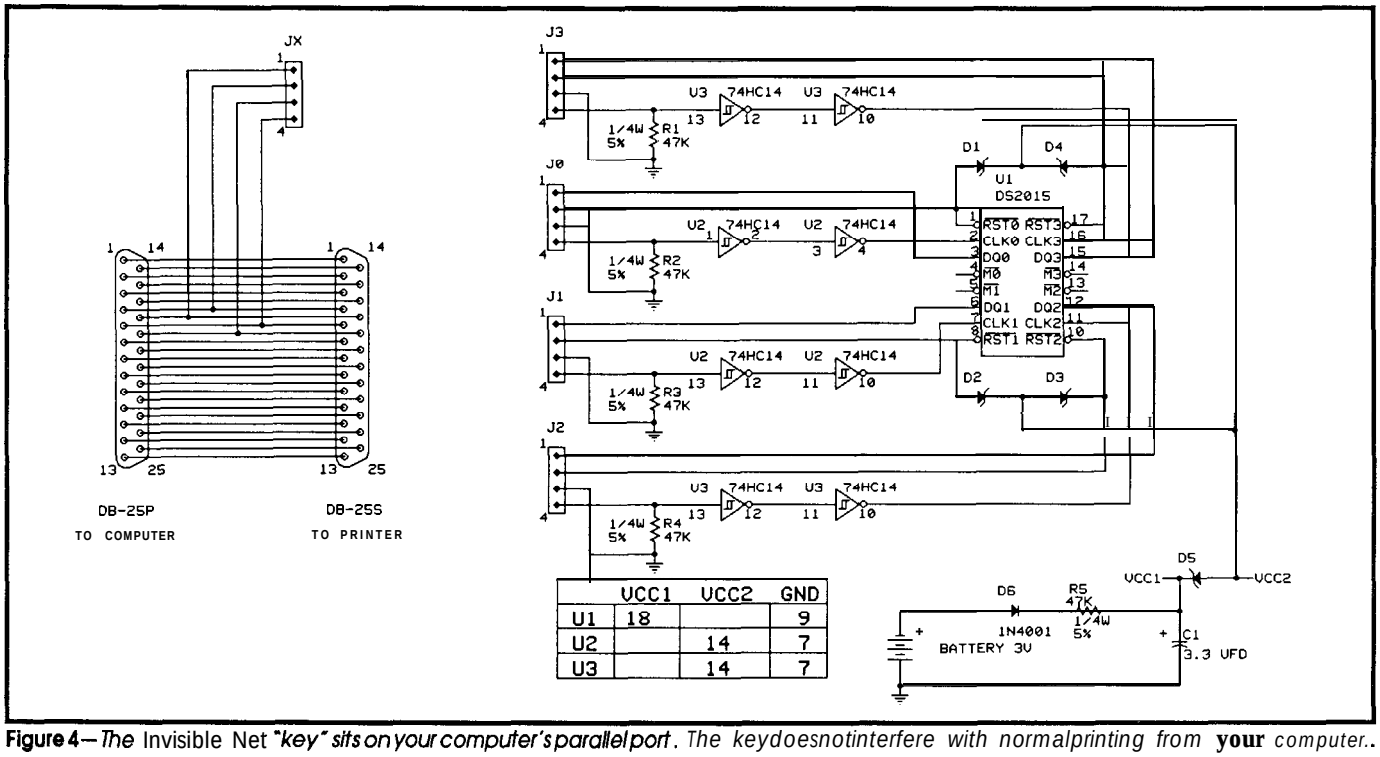


Figure 4—*The* Invisible Net *"key" sits on your computer's parallel port. The key does not interfere with normal printing from* **your** *computer.*

However, there is a status port where four open-collector output bits are also input bits. If the outputs are left high, they can be pulled low by an external source and read as inputs by the status register.

Previously, I described the I/O necessary for communications with each port of the DS2015. Now let's assign a few bits to do this communication. The RST\ line will be driven from D2 and CLK will be driven by D3 of the printer port data bus. The D/Q line carries data in both directions, so it will be connected to the SLCT\ line. This line is actually D3 on the status port. It's important to note that this line is open collector; if a printer is attached, it must be on and selected, or the SLCT\ line will be pulled low and no data can be sent in either direction. Also be aware that the status port inverts all data written to or read from it.

If we ignore the M\ line on the DS2015 for the present, the three signals RST\, CLK, and D/Q along with ground create a four-wire communication line just right for use with inexpensive modular phone cable. The schematic for the Invisible Net hardware is shown in Figure 4.

The program in Listing 1 is written in BASIC and will communicate with the DS2015 when connected to the printer port. The program attempts to establish communication with each of the DS2015's ports at each of the possible printer port addresses. When it finds an active port, the port is identified along with the printer port address, and contains all the building blocks for you to create your own network.

A Network You Can Call Your Own

The DS2015 is available through distribution for about $10.00. Call Dallas Semiconductor at (214) 450-0400 for the nearest distributor. But wait, there's more! Dallas makes a port adapter, the DS1256, which goes in-line with the printer port. This adapter has a modular jack installed to make an easy connection to pins 4, 5, 17, and 18 on the

printer port with **a modular phone cable.** In fact, Dallas has packaged the DS2015 in a complete product: the DS9050. This package includes two DS1256 port adapters and one four-way junction box (two additional adapters are necessary to link together four systems). Priced at about $80.00, the DS9050 is tough to beat.

Dallas Semiconductor is not in the software business, but have agreed to allow distribution of evaluation software. The software lets any PC/AT user view directories and copy and erase files from any other PC/AT system online, and creates a simple network that is totally invisible to the users. The executable code is available free for downloading from the Circuit Cellar BBS. *[Editor's Note: See page 78 for more information about Circuit Cellar INK Software On Disk #9 and how to download from the Circuit Cellar BBS.]* This code is not supported in any way by Dallas Semiconductor or Circuit Cellar.

I use this **hardware and** software combination to transfer files between my XT and AT. Receiving a 20-file directory takes under 10 seconds. Copying a 50K file between machines to a floppy disk takes about 75 seconds. This exceeds 5 kbps by quite a bit if you take into consideration the overhead of reading and writing the file from and to floppies. All things considered, this makes an inexpensive network for up to four PCs without taking up valuable expansion slots or ports. ✣

Jeff Bachiochi (pronounced "BAH-key-AH-key") is a member of the CIRCUIT CELLAR INK engineering staff. His background includes work in both the electronic engineering *and manufacturing fields. In his spare time Jeff enjoys his family, windsurfing, and pizza.*

IRS

216 Very Useful
217 Moderately Useful
218 Not Useful

```
10 PP(0)=&H278:PP(1)=&H378:PP(2)=&H3B0:
   AS-"TestData"
20 DP(0)=&HCB:DP(1)=&HDB:DP(2)=&HEB:DP(3)=&HFB
30 FOR X=0 TO 2
40 P=PP(X)
50 FOR Y=0 TO 3
60 D=DP(Y)
70 GOSUB 2000
80 D=DP(Y)
90 WC=8
100 GOSUB 1000
110 D=DP(Y)
120 GOSUB 2000
130 IF D=85 THEN PRINT "Connection to Printer
    Port #";HEX$(PP(X));" Interlink Port #";Y;
    " 8 bytes =";B$
140 NEXT Y
150 NEXT x
160 END
1000 REM ****************************************
1010 REM * Write to Interlink Port Routine
1020 REM ****************************************
1030 GOSUB 3ooo:GosUB 4000:GOSUB 5000
1040 D=2^(7-Y)+WC:GOSUB 4000
1050 FOR M=O TO 7:D=ASC(MID$(A$,M+1,1)):GOSUB
     4000:NEXT M
1060 GOSUB 6000
1070 RETURN
2000 REM ****************************************
2010 REM * Read from Interlink Port Routine
2020 REM ****************************************
2030 B$="":GOSUB 3000:GOSUB 4000:GOSUB 5000
2040 D=2^(7-Y):GOSUB 4000
2050 FOR M=0 TO 7:GOSUB 5000:B$=B$+CHR$(D):NEXT M
2060 GOSUB 5000
2070 GOSUB 6000
2080 RETURN
3000 REM ****************************************
3010 REM * Set CLK (D3=1) then set *RST (D2=1) to
3015 REM * start cycle
3020 REM ****************************************
3030 OUT P,8:OUT P,&HC
3040 RETURN
4000 REM ****************************************
4010 REM * Send 8 bits of data
4020 REM ****************************************
4030 REM Do all 8 bits
4040 FOR Z=0 TO 7
4050 REM Clear CLK (D3=0) on the printer port
4060 OUT P,4
4070 REM Is bit Z a '1' in byte 'D'?
4080 REM If yes, Clear D/Q (D3=0 of printer
4085 REM status port) to output a '1'
4090 REM If no, Set D/Q (D3=1 of printer
4095 REM status port) to output a '0'
4100 IF (D AND 2^Z)THEN OUT P+2,INP(P+2) AND
     &HF7 ELSE OUT P+2,INP(P+2) OR 8
4110 REM Set CLK (D3=1) on the printer port
4120 OUT P,&HC
4130 REM If not done do next bit
4140 NEXT Z
4150 REM Set D/Q (D3=1) so input can be read
4160 OUT P+2,INP(P+2) AND &HF7
4170 RETURN
5000  REM  *******X******************************
5010 REM * Receive 8 bits of data
5020 REM ****************************************
5030 REM Clear data 'D' to null
5040 D-0
5050 REM Do all 8 bits of data 'D'
5060 FOR Z=0 TO 7
5070 REM Clear CLK (D3=0) on the printer port
5080 OUT P,4
5090 REM Read 'D3' on the printer status port
5100 REM If '0', Set bit 'Z' of data 'D'
5110 REM If '1', then NOP
5120 IF (INP (P+2) AND 8) = 0 THEN D=D+2^Z
5130 REM Set CLK (D3=1) on the printer port
5140 OUT P,&HC
5150 REM If not done do next bit
5160 NEXT Z
5170 RETURN
6000 REM ****************************************
6010 REM * Clear RST\ (D2=0) leaving CLK at its
6015 REM * present state to end cycle
6020 REM ****************************************
6030 OUT P,INP(P) AND &H8
6040 RETURN
```

listing 1 *–This BASIC program provides a framework for building your own networksoftware. If you wouldrather start with fully operational networking, see the text of the article for information on downloading or ordering more complete software.*

SILICON UPDATE

The Waferscale Integration PAC 1000

Microcontroller, RISC, or PLD?

by Tom Cantrell

I know what you're thinking as you're reading this—what aspiring techno-guru could skip an article with a three-buzz-word headline?

The PAClOOO from Waferscale Integration in Fremont, CA combines aspects of all three au courant technologies. Like a microcontroller, the PAClOOO integrates CPU, memory, and I/O on a single chip. Like a RISC processor, it has lots of registers; load-store architecture; a simple, hardware-oriented instruc tion set; and single-cycle execution. Like a PLD, the PAC1000 can programmably integrate various high-speed logic functions in a single chip.

Perhaps the PAC1000 is best described as a single-chip user-microcoded logic unit. But let me tell the story and you can decide for yourself.

Microcode Basics

Usually, microcode refers to a level of software hidden beneath user-written software. A good analogy is the way an interpretive high-level-language program, such as BASIC, serves as "data" to a program (the BASIC interpreter) that determines which machine codes are ultimately executed. Similarly, for a microcoded CPU, the machine code itself acts as data for a microcoded program that determines which operations the machine performs. Indeed, some machinescarry the principle further with "nanocode" and even "picocode." The point is that each level consists of a program which treats (or interprets) the next higher level as program data.

Just as a single statement in BASIC will result in the execution of many primitive machine codes, a single primitave machine code will result in the execution of many microcode steps.

Consider a typical machine code instruction, INC (REG), which increments the contents of a memory location whose address is contained in a register. As far as the machine-level (i.e., assembly language) programmer is concerned, the hypothetical INC(REG) instruction is a single monolithic operation.

But, looking deep inside the CPU we find that one INC (REG) instruction actually causes many low-level machine operations to occur:

1) Output the PC (program counter) address
2) Read the instruction (INC(REG)) at the PC address
3) Increment the PC (prepare for the next instruction)
4) Decode the instruction (INC) and operand ((REG))
5) Output the operand address (REG contents)
6) Read the operand
7) Increment the operand
8) Write the operand

This is a simple example; In the real world things get a lot more complicated. For instance, my "read the operand" primitive causes all sorts of lower-level activity-drive the address bus, assert control signals, sample the READY line, input the data, deassert the bus, and so on. Then there are those nasty asynchronous events, like DMA and interrupts, to deal with. The real power of microcode is that a single instruction performs many hardwareoperationsatonce.

Historically, microcoding emerged in the '60s as the preferred approach for CPU design. Compared to complex hard-wired designs, mi-

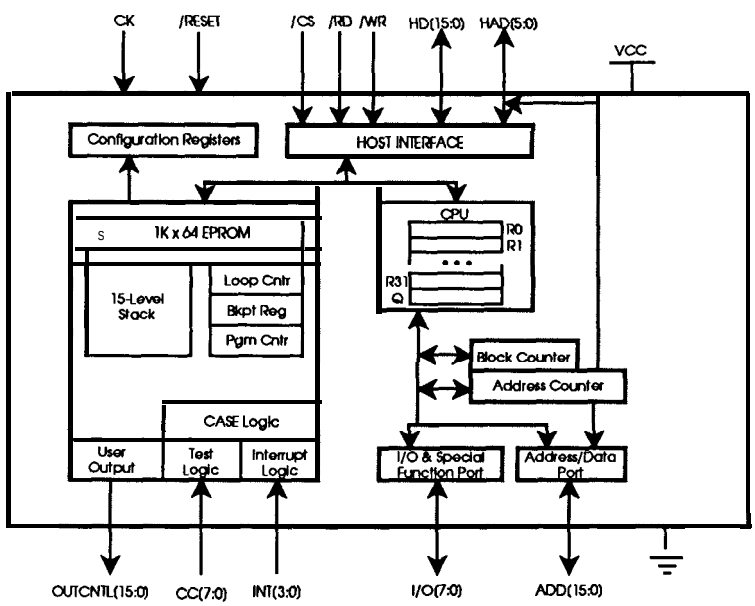
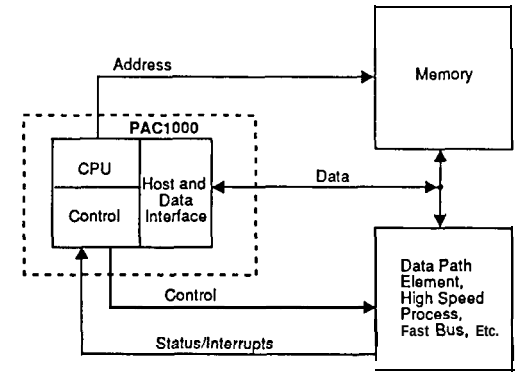
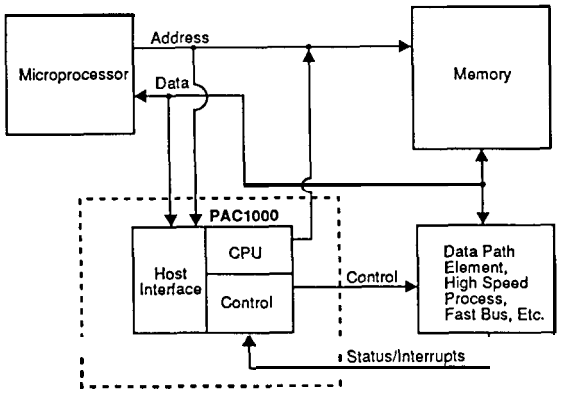


Figure 1 -The PAC *1000* includes microcode memory, registers, *ALU, sequencer, and more on a single CMOS chip.*



STANDALONE MODE



PERIPHERAL MODE

Figure **2**—*Part of the PAC 1000's flexibility is its capability to run in either stand-alone mode or peripheral* mode.

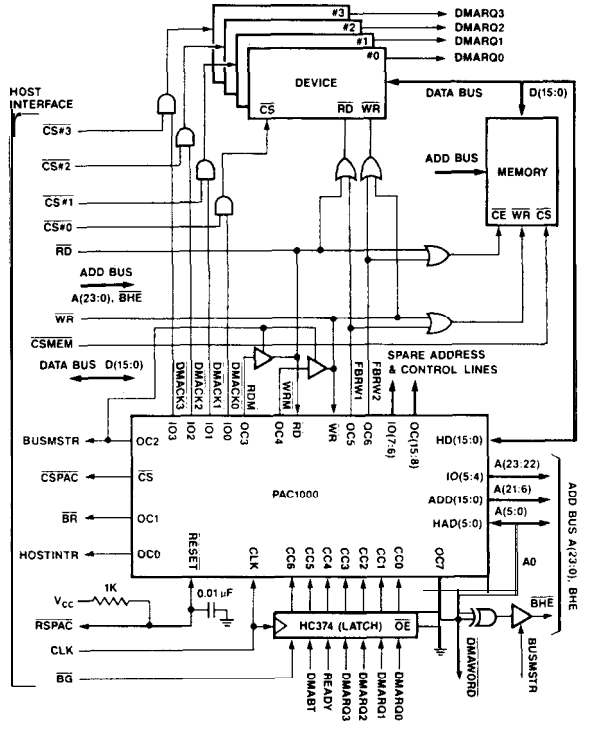


Figure *3-A typical application In which the PAC1000 may be found is a* high-speed *16-bit DMA controller.*

```
/* inner transfer loop - source to destination*/
/* read data and wait for READY                */
WAIT:JMPNC  READY WAIT,      /* stmt.1 PRCGCNTL */
     OUT READ_CYCLE;         /* stmt.1 OUTCNTL */
/* decrement count, update address, write data*/
     LOOPNZ WAIT,            /* stmt. 2 PROGCNTL*/
     ACL := ACL + Q,         /* stmt. 2 CPU     */
     OUT WRITE_CYCLE;        /* stmt. 2 OUTCNTL */
```

listing 1—*Microcode* is actually quite different *from* standard *assembly language due to its parallel nature.*

crocode simplifies design and development, and is easy to upgrade and fix. Many of today's popular CPU chips are microcoded.

In contrast to the factory microcode buried in chips, user-programmable microcode has been relegated to a high-performance/very high cost-power-size niche. Typically, a user-microprogrammable CPU requires a board full of logic built by a team of mad scientists (and maybe a faith healer for debugging).

The PAClOOO combines all the arcane pieces-microcode memory, registers, ALU, sequencer, and so on-on a single CMOS chip (Figure 1). A key point is that the microcode memory is user-programmable EPROM in contrast to the preprogrammed ROM found in a fixed CPU design.

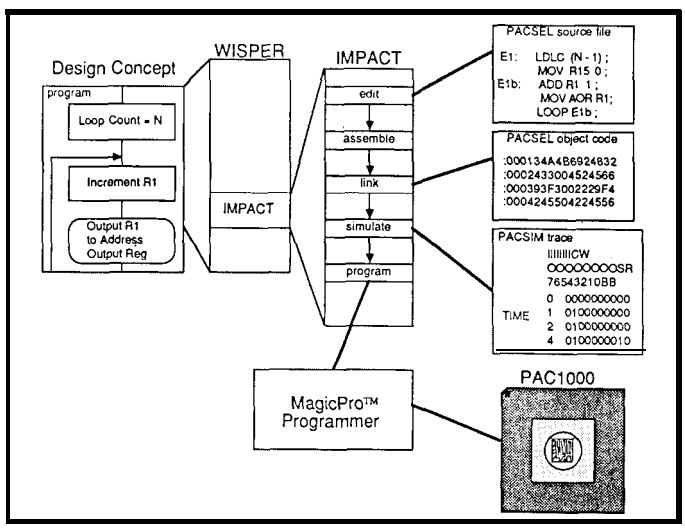


Figure 1—*The Waferscale PAC-SDT development system goes a long way* towards *simplifying microcoding, but* R's *still* not *a* trivial process.

With the PAClOOO, any budding Cray wanna-be can roll their own special-purpose chip which can function stand-alone or as a peripheral for another CPU (Figure 2). To specifically support the latterapplication, the PAClOOO includes interface logic which eases host CPU connection. Thus, a preprogrammed PAClOOO can hook to another CPU, just like any other peripheral.

Thinking Parallel

Besides raw speed-at 20 MHz the PAClOOO executes a (micro)instruction every 50 ns-the key for high performance is the parallelism inherent in microcode. The 64 bits at each of the 1K on-chip EPROM microprogram addresses are divided into fields which simultaneously control different hardware operations. Of course, thinking in parallel isn't easy which is why microcoders are a different breed.

The PAClOOO goes a long way towards making it easy. The CPU includes a number of luxurious features like a stack, case branching, loop counter, interrupt controller, and so on. The PAClOOO micro-assembler even includes extensions (micro-macros?) to simplify common operations and give a high-level-language flavor. Actually, the PAClOOO instruction set seems quite like that of a typical CPU chip-except you get to do three things at once.

Each "step" of your program simultaneously performs a CPU operation, performs an output operation, and determines, by checking internal/external inputs, which instruction to execute next. In PAClOOO terminology, these are, respectively, CPU, OUTCNTL, and PROGCNTL operators.

An example will help clarify things. Figure 3 shows a PAClOOO programmed to implement a high-speed (up to 10M words/s with 20-MHz clock) 16-bit DMA controller. Notice how the PAClOOO I/O lines fulfill the roles of those lines found on a typical DMAC chip (address, data, DMA req/ack, etc.). Listing 1 shows a small portion of the microcode which controls each DMA word transfer.

Notice that a single "statement," terminated by a semicolon, consists of one to three fields (CPU, OUTCNTL, PROGCNTL in any order) separated by commas. If you don't specify all three operations, the assembler will automatically fill in the missing field with a "do-nothing" code: NOP for the CPU, MAINTAIN for the OUTCNTL, and CONTINUE for the PROGCNTL. Of course, since you don't want to "do nothing" often, it's up to you to make sure your algorithms are fully parallelized.

The first statement simultaneously loops on the READY line while outputting the bus control signals (i.e., DMACK) to cause a DMA source data read. Note that a CPU operator is not specified so the assembler will automatically insert a NOP. The second statement simultaneously decrements and loops on the transfer counter, updates the transfer address, and outputs the bus signals to write DMA destination data.

This is a good example how PAClOOO code can look funny to those stuck thinking sequentially. To the unknowing, it looks like the statements after the LOOP statement will only be executed once at the end of the loop. Rather, they are executed each time the loop counter is decremented. Taking advantage of the free-form field ordering, you could place the PROGCNTL field (the LOOP statement) last to clarify program flow. It's probably a good idea to standardize on a fixed order unless you consider unreadable code a badge of honor (or job security).

Puttin' On The Bits

The Waferscale PAC-SDT IBM PC-based development package includes five programs. WISPER and IM-

PACT comprise a menudriven shell, while PACPRO burns the PAClOOO chip using the company's MagicPro programmer. You enter your program with the text editor of your choice, assemble and link with PACSEL, and then simulate your program's operation with PACSIM (Figure 4).

Though Waferscale has gone a long way towards making microcoding easy, developing and debugging PAClOOO applications is not for the faint of heart. Besides writing the microcode, the user faces the dreaded simulation "bottleneck"—after slogging through a trace (20 million operation per second of real time!) it feels like your neck got hit with a bottle!

Based on price, performance, and complexity, the PAClOOO is clearly targeted at only the most demanding applications, for which it offers some unique advantages compared to older multichip offerings. The ideal application for the PAClOOO is one that isn't served by a standard LSI chip, is too complex for PLDs, and isn't high enough volume (or the design isn't stable enough) to justify a custom gate array.

The concept of user-microprogrammable devices filling a gap between general-purpose microprocessors and hardware PLDs is an interesting one. Technology will inexorably reduce chip price and improve performance. Fulfilling the concept's potential depends on the emergence of more powerful and easy-to-use development tools.

Price and Availability

The PAClOOO is available in ceramic and plastic pin grid array (PGA) and leaded chip carrier (PLCC) packages; and commercial (O-70°C) and military (-55–125°C) temperature ranges. In the lowest-cost version (PLCC package, commercial temp), the 100-piece price for the 12-MHz version is $75. The 16-MHz version is $107 and the 20-MHz is $160. The PAC-SDT-GOLD package includes the PAClOOO development software and EPROM programmer for $4995.

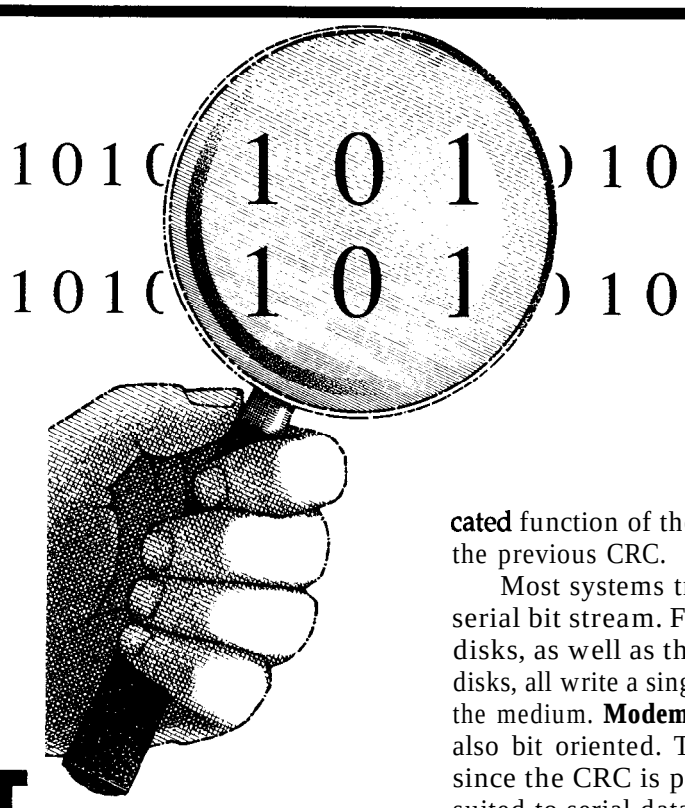
For literature or other information, contact
Waferscale Integration, Inc.
47280 Kato Road
Fremont, CA 94538
(415) 656-5400

Tom Cantrell holds a B.A. in economics and an M.B.A. from UCLA. He owns and operates Microfuture Inc., and has been in Silicon Valley for 10 years involved in chip, board, and system design and marketing.

IRS

219 Very Useful
220 Moderately Useful
221 Not Useful

Circle No. 143 on Reader Service Card



Computing CRCs in Parallel

by Jack Ganssle

The CRC (Cyclic Redundancy Check) is a sophisticated checksum that has long been the most common means of testing data for correctness. Every sector on a disk has a CRC of the data stored to alert the operating system of data dropouts. A CRC doesn't identify which byte is in error, but it pretty much guarantees that you'll be alerted to at least single-bit errors.

All CRCs are binary polynomials that are divided into the data stream. Although the definition and selection of the CRC is quite complex, its use is not.

The most common CRC polynomial is the CCITT form used by IBM's SDLC (Synchronous Data Link Control) protocol. It is of the form:

$$X^{16} + X^{12} + X^5 + 1$$

The physical representation of this CRC is a 16-stage shift register. Unlike a conventional shift register, four of the input terms are not from the previous stage;rather, they're the exclusive OR of the previous stage and the input bit. The inputs to bit positions 16, 12, 5, and 1 (the coefficients from the polynomial) are the exclusive OR of the previous stage and the new data bit. You can think of the CRC as a state machine whose output is a complicated function of the input data and the previous CRC.

Most systems transfer data as a serial bit stream. Floppy and hard disks, as well as the newer optical disks, all write a single bit at a time to the medium. **Modem applications** are also bit oriented. This is fortunate, since the CRC is particularly well-suited to serial data transfers. Serial shift registers, even with the feedback terms, are easy to implement in silicon. Fairchild's 74F401 chip is a 14-pin package that will compute CRCs on serial data using any of eight different polynomials. Many floppy disk controllers use this chip or similar circuitry to automatically append a CRC to the data stream. Using preprogrammed CRC chips, serial CRCs are almost trivial to add to a system.

If the data is transmitted in parallel, the problem becomes more complex. Think about it: the CRC is computed by eight shifts per byte, each shift resulting in the exclusive ORing of several terms within the byte. After eight of these operations, the output is far from obvious. How do you compute a useful CRC on 8 bits at once?

While assuring the integrity of parallel data is difficult, the quest for speed is bringing more parallel devices into the mainstream. An obvious example is a large RAM disk. Most implementations make the hardware look just like a hard disk, so the operating system can handle the device without special drivers. Other peripherals may be connected using SCSI, which almost always is designed to emulate a disk. Whenever the operating system expects to see a disk, it will also expect a CRC.

How do you implement a CRC in a purely parallel interface? An obvious approach is to convert the parallel data to serial, compute the CRC, and convert it back to parallel. Although conceptually easy, fast data transfers will require a mind-boggling clock rate (at least eight times the data rate), and a lot of hardware.

For a more realistic method of computing parallel CRCs, remember how a CRC is derived: The new CRC (after a byte is shifted in) is a function of the input data and the old CRC. Why not derive formulas for each of the 16 bits of the CRC after the eight new bits are shifted in? Then all 16 CRC bits can be computed in a single clock cycle.

Deriving the formulas for the parallel CRC is easy in principle but rather tedious. If the input data is *b0, b1,...,b7,* and the current CRC is *a0,a1, ..., a15,* compute the new CRC after one bit arrives. Then iterate seven more times, using new values of *a0, al, ... a15* each time. You'll get 16 new equationseach timeanewbitisshifted in. The last set of 16 is the result; these define the values of each bit after an entire byte is CRCed. Apply these 16 equations to a byte of data to compute a new CRC in a single cycle.

It turns out that a number of terms are common to many of the 16 equations. To simplify the algorithm, the common terms are identified as *I, J, K, L, M, N, 0,* and P. Listing 1 shows a program written in Turbo C to compute the CRC using the derived formulas.

This program looks horrible! It is far more cumbersome and complicated than the loop to do normal serial

```
/* Compute a parallel CRC */

#include <stdio.h>
int I,J,K,L,M,N,O,P;
int b0,b1,b2,b3,b4,b5,b6,b7;
int a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;
int iter;

main()

int k0,k1,k2,k3,k4,k5,k6,k7;

/* Preset CRC to all ones - CCITT rules */

        a0=a1=a2=a3=a4=a5=a6=a7=a8=a9=a10=a11=a12=a13=a14=a15=1;

/* Compute a number of CRCs with input data of all zeroes */

        b0=b1=b2=b3=b4=b5=b6=b7=0;
        for(iter0;  iter<10; ++iter)crc();

/* Now feed ones complement of the CRC back into calculation;
   result should be FOB8 */

        b0=!a0; bl-!al; b2=!a2; b3=!a3; b4=!a4; b5=!a5; b6=!a6;
        b7=!a7;
        k0=!a8; k1=!a9; k2=!a10; k3=!a11; k4=!a12; k5=!a13;
        k6=!a14; k7=!a15; crc();
        b0=k0; b1=k1; b2=k2; b3=k3; b4=k4; b5=k5; b6=k6; b7=k7;
        crc();
}
/* Remember the following C language operators:
        & = bitwise "and"; != negation; |= bitwise "or"
*/

crc()
{
        I=(b3 &!a3)| (!b3 &  a3);
        J=(b2 & !a2)| (!b2 &  a2);
        K=(b1 & !a1)| (!b1 &  al);
        L=(b0 & !a0)| (!bO & a0);
        M=(b7& !a7 & !I) | (!b7 & !a7 & I)
          I(!b7 & a7 & !I) |( b7 & a7 & I);
        N=(b6& !a6 & !J)| (!b6 & !a6 & J)
          |(!b6& a6 & !J)|( b6 & a6 &  J);
        O= ( b5 & !a5 & !K)|(!b5 & !a5 & K)
          |(!b5& a5 &!K)|(  b5  &  a5  &  K);
        P= ( b4 & !a4 & !L) |(!b4 & !a4 & L)
          |(!b4&  a4  & !L)|( b4  &  a4  &  L);
        a7=(a15 6 !P) | (!a15 & P);
        a6=(a14 & !I) | (!a14 & I);
        a5=(a13 & !J)|(!a13  & J);
        a4=(a12 & !K)| (!a12 & K);
        a3=( all & !L & !M)|(!all & !L &  M)
          |(!all & L & !M)|( all  &  L &  M);
        a2=(a10 & !N) | (!a10 & N);
        al-(a9  & !O) | (!a9 & 0);
        a0=(a8 & !P)| (!a8  &  P);
        a15-M;
        a14=N;
        a13-0;
        a12=P;
        all=I;
        alO=(J  & !M)| (!J & M);
        a9=  (K  & !N)| (!K & N);
        a8=(L&!O)|(!L   &   0);
        printf("\nCRC=   %X%X%X%X %x%x%x%x %X%X%X%X %x%x%x%x",
                a15,a14,a13,a12,a11,a10,a9,a8,a7,a6,a5,a4,a3,a2,
                a1,a0);
```

listing 1 *-Program to compute CRC in parallel.*

CRC calculation. Why go through all of this grief for so little reward? If you don't want to develop a new circuit, the answer is simple: Cumbersome though it is, the parallel CRC program works. If you don't mind adding a little bit of new silicon to your design, though, you can make the solution much simpler and cleaner.

The form of the C program closely resembles that of the equationsneeded to define a Programmable Logic Device (PLD). In other words, there is a very close equivalence between the code and a hardware implementation of thealgorithm. Fortunately, thePLD version can operate in a single clock cycle, and is much more aesthetically appealing than its awkward software cousin.

The PLD

For the uninitiated, a PLD **is** a sort of "super PAL." Based on EPROM technology, the PLD is a device that can be programmed with a user's equations. Like an EPROM, a fairly simple device programmer is used to load the formulas.

PLDs are defined in terms of "macrocells." Every macrocell is a multiple-input OR gate optionally connected to a flip-flop. Each of the OR gate inputs is an extremely wide AND gate; any or all of the PLD pins (and their inversions) can be connected to the AND gates.

You can specify the type of flip-flop; typically D, T, JK, and SR versions are all available. Or, you can disable the register altogether, so the macrocell becomes a big combinatorial gate structure.

A PLD is therefore a very large collection of AND and OR gates with some internal registers also thrownin. The connection of these components is up to the user; you write a set of boolean equations that makes the PLD implement the circuit you need.

Intel's 5C090 (equivalent to Altera's EP900) is a 40-pin PLD with 24 macrocells. It contains enough logic to completely implement the CRC algorithm in parallel. The rest of this article will be based on programming the 5C090.

```
EPLD: 5C090
Comments: This file computes a CCITT-SDLC CRC in parallel

OPTIONS: TURBO=ON
PART: 5C090
INPUTS: b0@2,b1@3,b2@4,b3@17,b4@18,b5@19,b6@22,b7@23,preset@38,
        hicrc@37,locrc@24,clk1@1,clk2@21
OUTPUTS:a0@5,a1@7,a2@9,a3@11,a4@13,a5@15,a6@25,a7@27,
        a8@6,a9@8,a10@10,a11@12,a12@14,a13@16,a14@26,a15@28,
        I@32,J@31,K@30,L@29,M@36,N@33,O@34,P@35
NETWORK:
b0=INP(b0)
b1=INP(b1)
b2=INP(b2)
b3=INP(b3)
b4=INP(b4)
b5=INP(b5)
b6=INP(b6)
b7=INP(b7)
preset=INP(preset)
losel=INP(locrc)
hisel=INP(hicrc)
clkl-INP(clk1)
clk2=INP(clk2)
a0,a0=RORF(a0d,clkl,gnd,gnd,lo)
a1,a1=RORF(a1d,clk1,gnd,gnd,lo)
a2,a2=RORF(a2d,clk1,gnd,gnd,lo)
a3,a3=RORF(a3d,clk1,gnd,gnd,lo)
a4,a4=RORF(a4d,clk1,gnd,gnd,lo)
a5,a5=RORF(a5d,clk1,gnd,gnd,lo)
a6,a6=RORF(a6d,clk2,gnd,gnd,lo)
a7,a7=RORF(a7d,clk2,gnd,gnd,lo)
a8,a8=RORF(a8d,clk1,gnd,gnd,hi)
a9,a9=RORF(a9d,clk1,gnd,gnd,hi)
a10,a10=RORF(a10d,clk1,gnd,gnd,hi)
a11,a11=RORF(a11d,clk1,gnd,gnd,hi)
a12,a12=RORF(a12d,clk1,gnd,gnd,hi)
a13,a13=RORF(a13d,clk1,gnd,gnd,hi)
a14,a14=RORF(a14d,clk2,gnd,gnd,hi)
a15,a15=RORF(a15d,clk2,gnd,gnd,hi)
L,L=COIF(Ld,)
K,K=COIF(Kd,)
J,J=COIF(Jd,)
I,I=COIF(Id,)
P,P=COIF(Pd,)
O,O=COIF(Od,)
N,N=COIF(Nd,)
M,M=COIF(Md,)

EQUATIONS:
lo=/losel;
hi=/hisel;
Id=(b3 * /a3) + (/b3 * a3);
Jd=(b2 * /a2) t (/b2 * a2);
Kd=(b1 * /al) t (/b1 * al);
Ld=(b0 * /a0) t (/b0 * a0);
Md=( b7 * /a7 * /I) + (/b7 * /a7 * I)
 t (/b7 * a7 * /I) t ( b7 * a7 * I);
Nd=( b6 * /a6 * /J) + (/b6 * /a6 * J)
 t (/b6 * a6 * /J) t ( b6 * a6 * J);
Od=( b5 * /a5 * /K) + (/b5 * /a5 * K)
 t (/b5 * a5 * /K) t ( b5 * a5 * K);
Pd=( b4 * /a4 * /L) + (/b4 * /a4 * L)
 t (/b4 * a4 * /L) + ( b4 * a4 * L);
a7d=(a15 * /P) t (/a15 * P) + /preset;
a6d=(a14 * /I) t (/a14 * I) t /preset;
```

Listing 2— PLD *source file for a parallel CRC.*

Putting the CRC in a PLD

Examining the C program, it quickly becomes apparent that the intermediate I through *P* terms must be computed before any of the *a0–a15* outputs, but once *I–P* are known, then all 16 *a0–a15* termscould becomputed simultaneously. We should therefore assign the *I-P* terms to combinatorial outputs in the PLD. The *a0* to *a15* terms (the CRC itself) are assigned to registered outputs. In this case the current CRC is needed as part of the new one; therefore the flip-flops' outputs will be fed back into the equation matrix.

It's relatively easy to translate the Turbo-C program into PLD equations. This was my intention when writing the code; the real purpose of the C program is to provide a simulation of the CRC function as implemented in a PLD. Listing 2 shows the file that defines the PLD.

For those unfamiliar with PLD design, the NETWORK section of the PLD file defines the nature of each of the device's pins. *b0* to *b7* are input bits. I to *P* are combinatorial outputs. The *a0* to *a15* (CRC) terms are defined as RORF (Registered Output, Registered Feedback). The termsarelatched in D flip-flops, but the current value (before the device is clocked) goes back into the equation matrix.

Figure 1 shows the connection of the PLD to a computer's data bus. Most data communication processors manipulate 8-bit data, so an 8-bit data bus is shown. The input data is a byte, but the output CRC is 16 bits. This PLD is designed to let usmultiplex the CRC onto the bus as two individual bytes. The input bits come from the same data bus. *b0* is thus tied to *a0* and *a8*, *b1* to *a1* and *a9*, etc.

LOCRC and HICRC are used to dump the upper or lower CRC byte onto the bus. Once the calculation is complete, the processor asserts these inputs low (one at a time) and reads the two bytes. These inputs are typically connected as input port selects.

The PRESET input initializes the CRC before any data is transferred. The CCITT specification requires that the CRC be initialized to all ones.

```
a5d=(a13 * /J) + (/a13 * J) + /preset;
a4d=(a12 * /K) + (/a12 * K) + /preset;
a3d=(a11*/L * /M) + (/all * /L * M)
  + (/a11 * L * /M) t ( all * L * M) t /preset;
a2d=(a10*/N)+ (/a10 * N) + /preset;
a1d=(a9 * /O) + (/a9 * o) + /preset;
a0d=(a8 * /P) t (/a8 * P) t /preset;
a15d=M t /preset;
a14d=N + /preset;
a13d=O t /preset;
a12d=P + /preset;
a11d=I t /preset;
a10d=( J * /M) + (/J * M) t /preset;
a9d= (K * /N) + (/K * N) t /preset;
a8d= (L * /O) + (/L * 0) t /preset;

END$
```

listing 2—*(continued)*

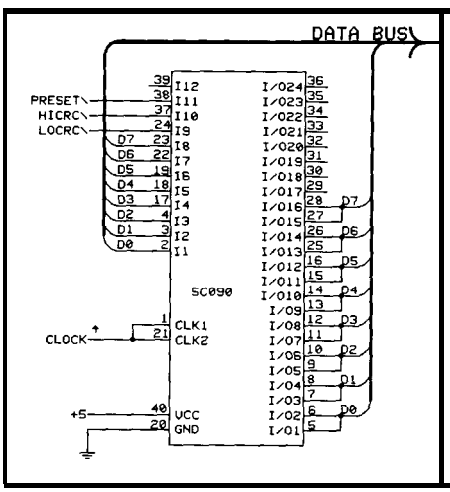


Figure I-A *5C090 PLD makes implementation of parallel CRCs in hardware* easy.

Input data	Output CRC
preset	FFFF
0	0F87
0	F0B8
0	3933
0	0321
0	3088
77	0F48
CF	F0B8

Figure 2—*Given the sample data in the left* column, the expected *CRCs* are listed in *the right* column.

When asserted, PRESET loads a 1 into all of the *a0–a15* terms after the next clock cycle.

Be warned! Your interface circuit must assert CLOCK when PRESET is low. Remeber that CLOCK is high-edge triggered.

To use the programmed PLD, first initialize the CRC by asserting PRESET low and driving CLOCK high. Then transfer data one byte at a time. For each byte, drive CLOCK high once the input data is stable, and after the data has been present long enough to let the *I* to P terms propagate (typically 100 ns). The PLD computes a new value for the CRC and loads it in the *a0–a15* registers when CLOCK goes high. After a block is transferred, the CRC is ready to be read by the computer.

The CRC PLD uses a pretty complex series of equations. How **can** you be sure the circuit works correctly? One way is to compare the CRC to known good values after specific data is transferred. Figure 2 shows CRC values for an input data stream of successive zeros. You can check the CRC after each clockagainst this table.

The CCITT polynomial has a self-checking feature. Once a stream of data has gone through the CRC calculator, you can supply the one's complement of the resulting CRC to the PLD and always get FOB8 as the new value. *Always.* **This serves as a sanity** check when developing hardware,and asaquickwayofverifyingdataagainst a previously computed CRC during read of a device. Be sure to transfer the low part of the CRC first, and then the high byte when making this test.

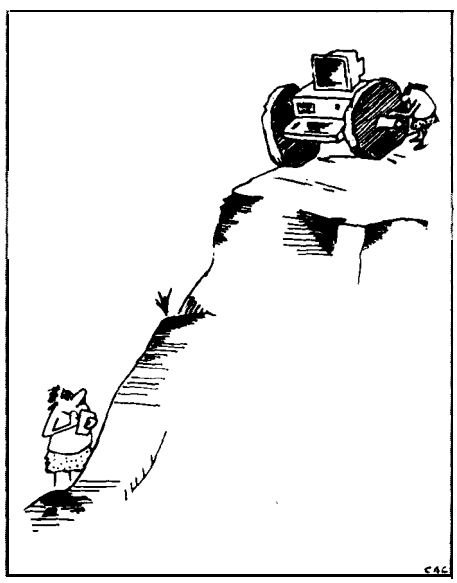
Software or Hardware?

The PLD computes a CRC in parallel using a single IC package. It can support a data rate of at least 10 MHz, even when used with relatively slow devices. Although the equations are messy and tedious, you can use the ones presented here as a "cookbook" solution. Of course, the C version will work in those situations **where you simply cannot afford to add hardware to the design. Either way, the parallel CRC lets you be sure that your parallel data is transferring safely.** ✣

Jack *Ganssle* is *the* president of *Softaid,* a *vendor* of *microprocessor development took When not busy pushing electrons around, he sails up and down the East Coast on his 35-foot sloop.*

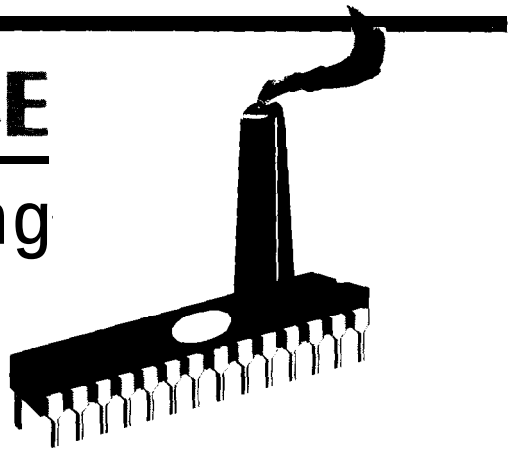
IRS

222 Very Useful
223 Moderately Useful
224 Not Useful



Data communications progress was slow before the invention of wire.

FIRMWARE FURNACE



From Fixed Point to Floating Point and Back Again

Writhing Reals

by Ed Nisley

Real numbers seem to be a hot spot among you folks. Several readers asked for more details on converting between floating-point and fixed-point numbers, because they're planning to use real numbers of one shape or another in their projects.

All this started with the Mandelbrot Engine described in "Ciarcia's Circuit Cellar" in the October, November, and December 1988 issues of BYTE. The Engine is an array of Intel 8751 single-chip microcontrollers programmed to execute the Mandelbrot Setalgorithminparallel. Wedesigned it to demonstrate the advantages of using many cheap processors to solve a problemusuallyassociated with big, fast, expensive machines.

A control program called DRIVER runs on an IBM PC/AT clone to handle keyboard inputs, distribute initial conditions to the array, and display the results on an EGA monitor. I used a special fixed-point numeric format because the Mandelbrot Set calculationsrequire exquisite precision, while 8751 processors aren't well-suited for floating-point arithmetic.

We didn't need fixed-point numbers in DRIVER's calculations, however, because the increased precision is required only in the Mandelbrot algorithm's inner loop. DRIVER uses ordinary C "double" variables and converts the values to fixed point before sending them to the array processors.

At the risk of digressing into software, this column will explore how that conversion works and provide a pair of routines to convert decimal fractions to binary fixed-point numbers. Because DRIVER works with ordinary double-precision floating-point numbers, I'll start by reviewing that format.

Double Dealing

Figure 1 shows the double-precision binary floating-point numeric format defined by the IEEE and used in 80x87 numeric coprocessors. Although there are other types of floating-point numbers around, nearly all contemporary PC languages support 80x87 operations. The examples here are in Microsoft C 5.1, but the bits, bytes, and algorithms can be adapted directly to your favorite language.

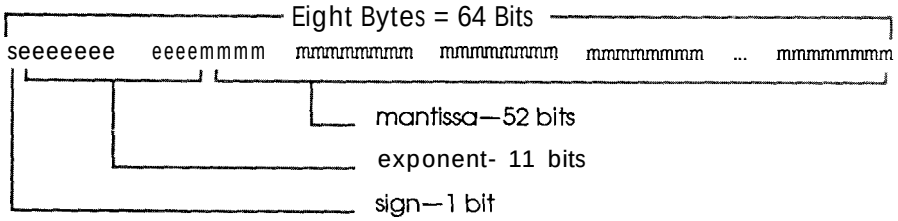


Figure 1 -This *is the double-precision floating-point format defined by the IEEE. The range of the format is* $\pm 1.7 \times 10^{\pm 308}$. *The* precision is *15–* 16 decimal digits with the *exponent* biased *by* + 1023 *(03FF* hex). *The* mantissa of this format has an implied high-order "*1*" bit, and all *negative* numbers *are represented in signed-magnitude notation.*

Each floating-point number has three fields distributed over eight bytes, as shown in Figure 1. It's easier to understand the fields by comparing them with a format that you are more familiar with: the scientific notation used to denote very large or very small physical quantities. For example, the speed of light is about 186,000 miles per second, which is 1.86×10^5 miles/second in scientific notation.

For this number, the sign is positive, the mantissa is 1.86, and the exponent is 5. The "x 10" is standard baggage to indicate we are working in base 10. The trick behind scientific notation is to convert all numbers, regardless of size, to something you can count on your fingers.. .the mantissa must be between 1 and 10!

While scientific notation is useful for those of us with 10 digits, computers have only two: zero and one. The first step in representing 186,000 as a binary floating-point number is a conversion to base two: 2D690 hex, or 0010 11010110 10010000 binary. You can imagine a "binary point" after the last bit on the right.

By definition, the mantissa of a binary floating-point number is greater than or equal to 1.0 and less than 2.0 (decimal), so it must have a "1" bit just to the left of the binary point. The number of places you shift the binary point to get a suitable mantissa determines the value of the exponent. This process is called "normalization."

Then, rather than waste storage on that high-order "1" bit, it is simply omitted. There are some cases where the range of the exponent won't allow normalization to be done, but I'm going to ignore those issues here. The final result is a 52-bit binary value

with the binary point on the left end of the number.

The normalized mantissa in our example is

6 B480 0000 0000

with the binary point now located to the left of the high-order bit. Because the leading 1 bit is suppressed, the highest bit is a zero: 6 hex is 0110 binary. The binary point was shifted 17 (decimal) bits to the left. The exponent field records the number of left shifts, but there must be some provision for right shifts as well: otherwise how would we represent numbers less than 1.0 (decimal)?

Rather than introduce a second sign bit for the exponent, the IEEE format adds an offset of 3FF to the shift amount. In our case, the shift was 11 hex, so the exponent field becomes 410 hex (11 + 3FF hex). For right shifts, subtract the shift from 3FF to get the exponent value.

The sign bit shown in Figure 1 is "1" when the mantissa is negative. Unlike the two's complement numeric format used in PC assembly code, mantissas use signed-magnitude notation, so the normalized mantissa corresponding to -186,000 is same as for +186,000 with a "1" sign bit.

Putting the normalized mantissa, the biased exponent, and the sign bit together, the floating-point equivalent of 186,000 turns out to be 4106 B480 0000 0000 hex. Figure 2 summarizes the bit manipulations needed for the conversion.

Almost Right, Mostly

You have surely converted decimal integers into binary integers. In fact, calculators have reduced base conversions to the same inscrutability as trig functions: "I don't know how it works, but this button right here does it just fine." Those buttons don't handle decimal or binary fractions, though, so an explanation is in order.

Table 1 provides equivalents for some simple binary fractions. Just as with integer values, you add the weights for each "1" bit to get the corresponding decimal value. Conversion is simple as long as you don't lose track of the bits!

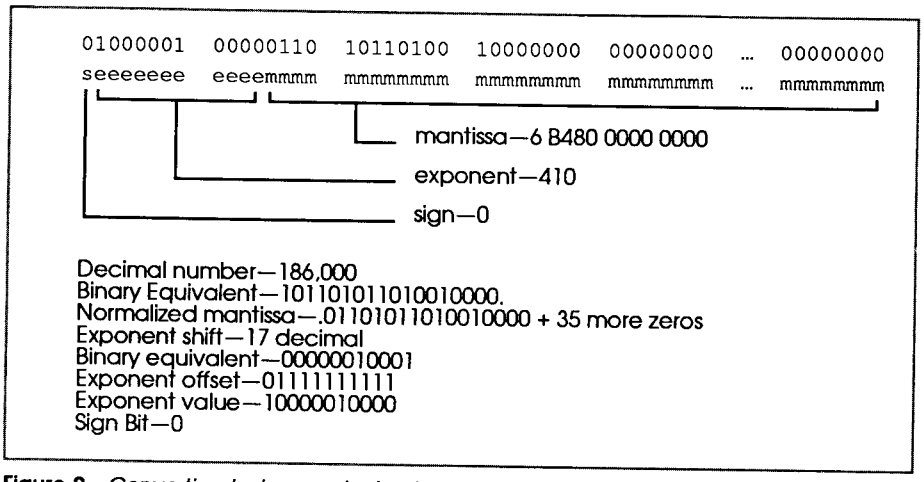


Figure 2—*Converting between decimal and binary representations involves "normalizing" the binary point to achieve a mantissa between 1.0 and 2.0. An explanation of the additional shifts required by the IEEE format is found in the text.*

In the Mandelbrot Engine's numeric format, shown in Figure 3, the binary point lies between the integer and fraction bits as usual, but it does not occupy an actual bit in storage. You can see that the Mandelbrot Engine format is optimized for small values: all numbers must fit between –8.0 and +8.0!

Most numeric applications will require more bits to the left of the binary point for a greater range of integers and can probably get along with far fewer bits in the fraction on the right. The key point is that you can pick the numeric format to suit the application as long as you are willing to write the math routines to manipulate the numbers.

However, even with 60 fractional bits there are some decimal numbers that do not have exact binary equivalents. Table 2 gives binary fractions (to 20 bits) for some common decimals. As you can see, the binary values do not settle down to an infinite string of zeros as the decimals did in Table 1.

Putting in the Fix

Converting between the binary floating- and fixed-point numeric formats is reasonably straightforward as long as you don't worry too much about where the extra bits in the fixed-point numbers should come from or go to. Listing 1 shows how to get from floating to fixed point.

The `DOUBLEBITS` union is the heart of the conversion. It allows the same eight bytes of storage to appear as either an ordinary floating-point number or a bit field that isolates the individual bits in each part of the number.

Because the binary point in the Mandelbrot Engine format is always located after the first four bits, the code in Listing 1 must denormalize the mantissa bits by shifting them until the exponent is zero (recall that the exponent is offset by +3FF hex). It starts by transferring the mantissa to a 16-byte work area and inserting the implied high-order bit to get all 53 bits.

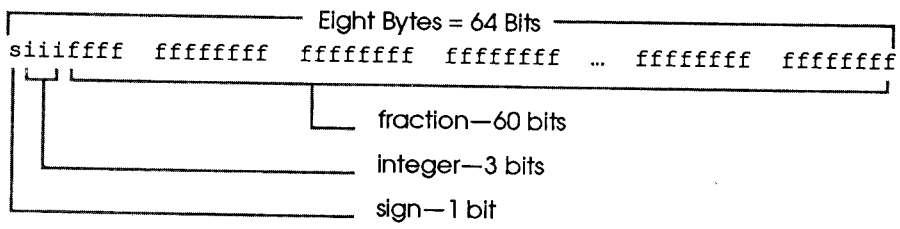


Figure 3—*The Mandelbrot Engine's numeric format is similar, but not identical, to the IEEE format. If you don't mind writing your own math routines, you can set the numeric format to suit the application.*

Binary	Fraction	Decimal
0.10000	1/2	0.50000
0.01000	1/4	0.25000
0.00100	1/8	0.12500
0.00010	1/16	0.06250
0.00001	1/32	0.03125
0.11000	1/2 + 1/4	0.75000
0.10100	1/2 + 1/8	0.62500

Table 1 -The *binary, fractional, and decimal equivalents for some common values. As with integers, the weights for each "1" bit may be added to get a corresponding decimal value.*

Incidentally, this conversion will fail if the floating-point number is already denormalized. In principle, this "can't happen here" because the normal 80x87 settings force zero for values that would ordinarily underflow, but it is worth worrying about for mission-critical code..

With the mantissa in place, the code simply shifts it left or right as dictated by the exponent. The rather complex code in the shifting loops shows what you must go through to makeaone-bit shiftacrossamultibyte quantity.

The mantissa uses signed-magnitude notation, so the code converts it to two's complement if the sign bit is one. The two notations use the same representation for positive numbers, so no change is needed in that case.

The final step is to copy the first eight bytes from the work buffer into the result variable. The bytes left behind may also have bits from the floating-point number, but those bits are lost forever because there is no room

Decimal	Binary
0.1000	0.0001 1001 1001 1001 1001 ...
0.0100	0.0000 0010 1000 11110101 ...
0.0010	0.0000 0000 0100 00011000 ...
0.0001	0.0000 0000 0000 0110 1000 ...
0.5000	0.1000 0000 0000 0000 0000 ...
0.2500	0.0100 0000 0000 0000 0000 ...
0.2000	0.00110011 0011 00110011 ...
0.1100	0.000111000010 1000 1111 ...

Table *2-Binary equivalents of 'simple' decimal fractions. The binary fractions do not terminate.*

in the floating-point format for them. Recall that the tradeoff was higher precision over a smaller dynamic range and you'll see why this must be so.

Conversions from fixed point to floating point are handled by the code in Listing 2. The process runs more or less in reverse from the previous routine: copy to the buffer, convert to signed-magnitude, shift to normalize, suppress the leading bit, copy the fraction to the mantissa, and set all the fields appropriately. If there were more fractional bits than fit into the mantissa (which is quite likely) they are simply discarded.

A Blizzard of Bits

Those two conversion routines suffice to convert between doubles and fixed-point numbers. You may have an application where the decimal number exists **only as** a text string; for example, a keyboard input specifying a servo table position. Converting the ASCII text to a decimal fraction, then to a binary floating-point number, then to binary fixed-point number may lose more bits than you care to discard. In that case a direct conversion to fixed point is in order.

Because the decimal number exists as ASCII text thereisnobitpattern we can copy directly into the fixed-point number. However, by treating each character of the string as a numeric digit you can extract as many bits as you need with no errors at all. The trick involves mechanizing the method you would use if you were forced to convert the number by hand (you could convert it by hand, right?). In effect, we treat the string as a binary-coded-decimal (BCD) number with each ASCII character as a digit.

Figure 4 shows how this works. Each multiplication by two cooks a single bit out of the value; if the decimal product exceeds one, the corresponding binary bit is one. The multiplications must be performed on a decimal number using decimal arithmetic because the number may not have an exact binary equivalent: that's the whole reason for this using method!

```
/*- galactic constants                                              */

#define NUMLEN 8          /* bytes per extended precision num   */
#define DBLPREC 16        /* total significant digits in double */

/* Extended precision numbers look like this:                      */
/*  +000.0000|00000000|NUMLEN-3 bytes|00000000                     */
/*   ^   ^                                                         */
/*   |   binary  point                                             */
/*   sign bit                                                      */
/* Negative numbers are in two's complement form                   */
/* The overall length is 64 bits, of which 60 are fractional      */
/* Overflow occurs for positive nums >7.999 & negative <-8.000    */
/* but we blithely ignore all that...                              */
/* Extended-precision numbers are stored with least-significant*/
/* byte first, in standard Intel format...                         */

typedef unsigned char EXTNUM[NUMLEN];  /* ext. precision format*/

/* "double" variables look like this:                              */
/*  +0000000|00000000|5 bytes|00000000                             */
/*  ^|-----------++-------------------|                            */
/*  |  exponent          mantissa                                  */
/*  sign bit                                                      */
/* Negative numbers are in signed-magnitude form                   */
/* Exponent is biased by +1023 (decimal)                           */
/* Note that the exponent is split across a byte boundary          */
/* ...remember these are stored with LSB in lowest addr. byte      */

struct dbits {
  unsigned char mantissa[6];  /* mant[0] is lowest order byte */
  unsigned int highmant:4;    /* high-order bits of mantissa   */
  unsigned int exponent:11;   /* excess-1023 exponent          */
  unsigned int signbit:1;     /* sign for sign-mag mantissa    */
};

typedef union {               /* map double & bits to same spot*/
  double dbl;
  struct dbits bits;
} DOUBLEBITS;

/*-------------------------------------------------------------------*/
/* Convert a double-precision value to extended fixed point    */
/* Catches overflows into sign bit, sticks at max value        */
/* Double values have only 53 bits of precision; force low 0s  */

void CvtDbl2Ext(unsigned char *extval, double dblval) {

   int index;
   int expshift;
   unsigned int accum;
   unsigned int saccum;
   DOUBLEBITS d;

   d.dbl = dblval;                    /* put it into the union */
   expshift = d.bits.exponent - 1023; /* conv exp to shift amt */

   if (expshift > 2) {                /* catch overflow        */
      memset(ExtAccum,'\xff',NUMLEN-1);
      ExtAccum[NUMLEN-1] = 0x3f;      /* force to max value    */
      expshift = 0;                   /* and do no shifting    */
   }
   else {
      ExtAccum[0] = 0;                /* fill unused bits      */
      memcpy(ExtAccum+1,d.bits.mantissa,6); /* trans low mant. */
      ExtAccum[NUMLEN-1] = (unsigned char) d.bits.highmant;
      ExtAccum[NUMLEN-1] |= 0x10;     /* add implied "1" bit   */
   }
```

listing 1 *-A program to convert from floating point to fixedpoint begins by denormalizing the mantissa until the exponent is zero. Performing one-bit shifts across multiple bytes requires* the *complex loop shown here.*

```
    while (expehift > 0) {              /* handle left shifts */
       saccum = 0;
       for (index = 0;index < NUMLEN; index++) {
          accum = (unsigned int) ExtAccum[index];
          accum = (accum << 1) | saccum:
          ExtAccum[index] = (unsigned char) accum;
          saccum = accum >>a:
       }
       expshift--;
    }

    while (expshift < 0) {              /* handle right shifts */
       saccum = 0;
       for (index = NUMLEN-1; index > -1; index--) {
          accum = (unsigned int) ExtAccum[index];
          accum _= rotr( (accum | saccum),1);
          ExtAccum[index] = (unsigned char) accum;
          saccum = (0x8000& accum) >> 7;
       }
       expshifttt;
    }

    if (d.bits.signbit) {
       NegExt (ExtAccum+NUMLEN,ExtAccum); /* compl. &set retn */
       memcpy(extval,ExtAccum+NUMLEN,NUMLEN);
    }
    else {
       memcpy (extval,ExtAccum,NUMLEN);   /* just set retn val */
    }

    return:
}
```

listing 1—*(continued)*

Theonlyarithmeticoperationyou must perform on the string is a multiplicationby two, or, alternatively, add the number to itself. There may be a carry from the preceding digit position and the operation may produce a carry for the next position as well.

Listing 3 illustrates one way to double an ASCII fraction without intricate program logic. A loop examines each character in the string right to left. At each character the code isolates the low-order four bits and adds 0x10 if there was a carry from the previous character. The result becomes an index into the `Dvalue` and Dcarryarrays,whichprovidethenew result digit and the carry for the next position. The ASCII string is updated in place so it is ready for the next pass.

The arrays have 32 elements, of which only 20 are used. I could have used 20-element arrays, but wasting a little space made debugging easier by separating the effect of the digits and carries on the array index.

The return value is simply the carry from the high-order (leftmost)

CIRCUIT CELLAR INK'S

ADVERTISER'S INDEX

Reader Service Number	Advertiser	Page Number
101	2000 AD Software	24
102	AISI Research	C4
103	Alpha Products	21
104	AVOCET	C2
105	B&B Electronics	39
106	Berry Computers	49
107	Binary Technologies	54
108	Cabbage Cases	66
•	Chrysalis	79
•	Ciarcia Design Works	79
109	Circuit Cellar	17
110	Circuit Cellar	62
111	Circuit Cellar	17
112	Computerwise	47
113	Contact East	70
114	Cottage Resources	73
117	Covox, Inc.	73
118	DATArx	34
119	Dycor	35
120	Eng. Collaborative	46
121	Environmental Optics	70
122	Express Circuits	6
123	Galacticomm	C3
124	Grammar Engine	54
125	GTEK, Inc.	7
126	Hazelwood Computer	68
127	[illegible]	14
128	Hogware	22
129	Innotec	15
130	Introl Corp.	29
115/116	JDR Microsystems	23
•	Lear Labs	79
131	Logical Systems	77
132	LTS/C Corp.	38
133	Meredith Industries	79
134	MetraByte	15
135	Micro Resources	34
136	Micromint	73
137	Micromint	51
138	Micromint	57
139	Micromint	64
140	Ming Eng.	76
141	NOHAU Corp.	66
142	Paradigm Systems	15
143	Programmer's Shop	53
144	PseudoCode	51
145	Quinn-Curtis	32
146	R&D Electronics	68
147	Softaid	71
•	Systronix	79
148	Thinking Tools	7
149	Timeline	4
•	Tinney	54
150	x-10	9

IRS

INK Rating Service

How useful is this article?

At the end of each article and some features there are three 3-digit numbers by which you can rate the article or feature.

Please take the time to let us, at Circuit Cellar INK, know how you feel our material rates with you. Just circle the numbers on the attached card.

```
/* Convert an extended fixed-point value to double floating   */
/* Returns the double value. Some bits will get dropped.       */

double CvtExt2Dbl(EXTNUM extval) {
   int index:
   int expshift;
   unsigned int accum;
   unsigned int saccum;
   DOUBLEBITS d;

   memset(&d.dbl,'\0',NUMLEN);       /* clear the decks          */
   if (extval[NUMLEN-1] & 0x80) {    /*get positive in acc.      */
      d.bits.signbit = 1;
      NegExt(ExtAccum,extval);
   }
   else {
      memcpy(ExtAccum,extval,NUMLEN);
   }
   accum = 0;                        /* if zero, we're done!     */
   for (index = 0; index < NUMLEN; index++) {
      accum |= ExtAccum[index];
   }
   if (accum != 0) {
      expshift = 0;                  /* normalize first 1 bit loc*/
      while (ExtAccum[NUMLEN-1] & OxEO) { /* right shift norm */
         saccum = 0;
         for (index = NUMLEN-1; index > -1; index-) {
            accum = (unsigned int) ExtAccum[index];
            accum = _rotr((accum| saccum), 1);
            ExtAccum[index] = (unsigned char) accum;
            saccum = (Ox8000 & accum) >> 7;
         }
         expshifttt;
      }
      while (!(ExtAccum[NUMLEN-1]& 0x10)) { /* lft shift norm*/
         saccum = 0;
         for (index = 0;index < NUMLEN; index++) {
            accum = (unsigned int) ExtAccum[index];
            accum = (accum << 1) | saccum;
            ExtAccum[index] = (unsigned char) accum;
            saccum = accum >> 8;
         }
         expshift-;
      }
      d.bits.exponent-expshift  + 1023; /* get excess 1023 exp */
      d.bits.highmant=0x000f&(unsigned int) ExtAccum[NUMLEN-1];
      memcpy(d.bits.mantissa,ExtAccum+1,6); /*xfer useful bits*/
   }
   return d.dbl;
}
```

Listing 2-Converting *from fixed point to floating point is roughly the reverse of the process shown in Listing 1. Excess fractional bits in the mantissa are simply discarded by this program.*

Decimal	x2
0.54321	1.08642
0.08642	0.17284
0.17284	0.34568
0.34568	0.69136
0.69136	1.38272
0.38272	0.76544
0.76544	1.53088
0.53088	1.06176
0.06176	0.12352
0.12352	0.24704

0.10000101100 binary

Figure 4-Converting *an ASCII representation to a numeric format requires treating the string as a binary-coded decimal (BCD) number where each ASCII character is a single digit.*

```
/* The original dec. number is stored as ASCII text string as */
/*  +d.ddddddddd...ddddd                                       */
/*  |||    up to 60 fractional digits                          */
/* ((actual decimal point (0x2e)                               */
/* |integer part of number (less than 8)                      */
/*  sign (+ is 0x2b, - is 0x2d)                                */
/*-------------------------------------------------------------*/
/* Double the decimal value represented in the ASCII string    */
/* Returns 'carry" from highest character (either 0 or 1)      */
/* The string must not contain anything other than dec. digits*/
/*  (so strip out things like decimal points and signs first) */
int TimesTwo(char *Value,int NumDigits) {
static char *Dvalue = "0246802468......"
                      "1357913579......";
static int Dcarry[32] = {00,00,00,00,00,16,16,16,16,16,99,99,
   99,99,99,99,00,00,00,00,00,16,16,16,16,16,99,99,99,99,99,99};
   int  Binary;
   char *Digit;
   int  Counter;
   int CarryIn,CarryOut;

   Digit = Value + NumDigits - 1;  /* point to low-order digit*/
   CarryIn = CarryOut = 0;
   for (Counter = NumDigits; Counter > 0; Counter--) {
      Binary = 0x0f & *Digit;
      CarryOut = Dcarry[Binary+CarryIn];
      *Digit = Dvalue[Binary+CarryIn];
      CarryIn = CarryOut;
      Digit--;
   }
   return (CarryIn != 0);
}
```

Listing 3—*This* program uses array indexes to double *an ASCII* fraction *without resorting to complicated logic. The program begins by isolating the low-order four bits of each ASCII digit, and ends by returning the carry from the high-order* character.

character which would become the integer part of the number if you were doing this by hand. Because the string does not include the integer position, it does not need to be modified for the next pass.

The program CONVDEC illustrates how to use this routine. Simply type "CONVDEC 0.1" to get the 60-bit binary equivalent of 0.1 decimal. CONVDEC is limited to positive values between 0.0 and 7.9999... because I didn't want to implement negation and general integer conversions for ASCII strings.

Bits from Thin Air

So now we have two methods to convert decimal fractions into binary fixed-point numbers. One starts with a double-precision floating-point number, but is therefore limited to the 53 bits available in the mantissa. The other starts with an ASCII text string, but can produce as many binary bits as you need with complete accuracy. Wouldn't it be nice to be able to combine the two methods and get an infi-

nite number of bits from a floating-point number?

Well, it would be nice to have perpetual motion, too.

It turns out you can wring more bits from the floating-point number than are present, at least for some numbers. The method is similar to that used in Listing 3: cook decimal digits out of the floating-point representation, then convert them to binary fractions with as many bits as you like. This will work as long as you don't try to cook more decimals out than the floating-point number encodes.

Figure 5 shows how this works. First, you need a table of the binary equivalents of each decimal place, taken to as many bits as appropriate. Next, you break the floating-point number into separate decimal places, extract the equivalent of each place from the table, and multiply them by the respective digits. Finally, when you add all the products together, the result is a reasonable approximation of the exact binary value, even if the floating-point number didn't have that many bits! Listing 4 illustrates the code required for this conversion.

The accuracy of the result could be improved by adding a few more bits to each table entry and rounding after the final summation. Even better results would come from expanding the table to include equivalents for all ten possible digits in each of 15 decimal places; this entirely eliminates the multiplications, at the cost of much more storage for the table entries.

You don't get something for nothing, of course. The magic conversion method doesn't know what bits were discarded when you (or the C library routines) created the floating-point number, so it cannot re-create the original value from thin air. In effect, it is somewhat increasing the precision of the equivalent number by assuming that the decimal values are exactly correct. This is not always so, but sometimes it's useful.

The EXTMATH routine converts a decimal number to fixed point (and back again) using both methods and displays all of the bit patterns so you can see what goes on. Type "EXTMATH 0 .1" to see what happens.

```
        Decimal      Binary

        1.00000  =  10 00 00 00 00 00 00 00
        0.10000  =  01 99 99 99 99 99 99 99
        0.01000  =  00 28 F5 C2 8F 5C 28 F5
        0.00100  =  00 04 18 93 74 BC 6A 7E
        0.00010  =  00 00 68 DB 8B AC 71 OC
        0.00001  =  00 00 OA 7C 5A C4 71 B4

   Binary point ----------------------------

        0.54321  =  5 X 0.10000  =  07 FF FF FF FF FF FF FD
                    4 X 0.01000  =  00 A3 D7 OA 3D 70 A3 D4
                    3 X 0.00100  =  00 OC 49 BA 5E 35 3F 7A
                    2 X 0.00010  =  00 00 D1 B7 17 58 E2 18
                    1 X 0.00001  =  00 00 OA 7C 5A C4 71 B4

                    Exact value = 08 BO FC F8 OD C3 37 17
```

Figure 5—*In some cases. you can wring more bits out of a number than were originally present. Adding the positions in the table allows you to get a reasonable approximation of the equivalent. It's important that you not try to cook out more decimals than the floating-point number can encode.*

```
/*-------------------------------------------------------------*/
/* Extended precision equivalents for decimal fractions       */
/* Each entry is the ext-precision equiv. of the corresponding*/
/* decimal fraction 1.0, 0.1, 0.01, 0.001, and so forth       */
/* Stored high-or&r byte first and flipped in SetupMath       */
/*  ...because flipping during transcription is hard to do    */
/*      and they're easy to mistype if transcribing by hand   */

unsigned char binfracts[DBLPREC][NUMLEN] = {
  {'\x10','\x00','\x00','\x00','\x00','\x00','\x00','\x00'},
  {'\x01','\x99','\x99','\x99','\x99','\x99','\x99','\x99'},
  {'\x00','\x28','\xf5','\xc2','\x8f','\x5c','\x28','\xf5'},
  {'\x00','\x04','\x18','\x93','\x74','\xbc','\x6a','\x7e'},
  {'\x00','\x00','\x68','\xdb','\x8b','\xac','\x71','\x0c'},
  {'\x00','\x00','\x0a','\x7c','\x5a','\xc4','\x71','\xb4'},
  {'\x00','\x00','\x01','\x0c','\x6f','\x7a','\x0b','\x5e'},
  {'\x00','\x00','\x00','\x1a','\xd7','\xf2','\x9a','\xbc'},
  {'\x00','\x00','\x00','\x02','\xaf','\x31','\xdc','\x46'},
  {'\x00','\x00','\x00','\x00','\x44','\xb8','\x2f','\xa0'},
  {'\x00','\x00','\x00','\x00','\x06','\xdf','\x37','\xf6'},
  {'\x00','\x00','\x00','\x00','\x00','\xaf','\xeb','\xff'},
  {'\x00','\x00','\x00','\x00','\x00','\x11','\x97','\x99'},
  {'\x00','\x00','\x00','\x00','\x00','\x01','\xc2','\x5c'},
  {'\x00','\x00','\x00','\x00','\x00','\x00','\x2d','\x09'},
  {'\x00','\x00','\x00','\x00','\x00','\x00','\x04','\x80'}
};

/*-------------------------------------------------------------*/
/* Convert a double-precision value to extended fixed point   */
/* Catches overflows into sign bit, sticks at max value       */
/* Converts at maximum precision in the extended buffer,      */
/*  using decimal equivalent of each digit in double value... */

void CvtDbl2FullExt(unsigned char *extval, double dblval) {

   int digit;
   int reps:
   int dblsign;
   DOUBLEBITS d;
   double dblint,dblfract;
```

listing 4—*This is the program equivalent of Figure 5. The program increases the precision by assuming that the decimal values are precisely correct. If this assumption is wrong, your results could begin deviating from correct by an ever-wider margin.*

```
    d.dbl = dblval;          /* set up union with double value */

    dblsign = d.bits.signbit;      /* force double to positive */
    d.bits.signbit = 0;

    if (d.dbl >= 8.0) {                    /* catch overflow     */
       memset(ExtAccum,'\xff',NUMLEN-1);
       ExtAccum[NUMLEN-1] = 0x3f;          /* force to max value */
    }
    else {
       memset(ExtAccum, '\0',NUMLEN);      /* zap accumulator    */
       for (digit = 0; digit < DBLPREC; digit++) {
          dblfract = modf(d.dbl,&dblint);
          for (reps = 0; reps < (int) dblint; reps++) {
             AddExt(binfracts[digit]);
          }
          d.dbl = 10.0 * dblfract;         /* extract next digit */
       }
    }

    if (dblsign) {
       NegExt(ExtAccum+NUMLEN,ExtAccum); /* compl & set retn    */
       memcpy(extval,ExtAccum+NUMLEN,NUMLEN);
    }
    else {
       memcpy(extval,ExtAccum,NUMLEN);  /* set return value    */
    }
    return;
}
```

listing 4—*(continued)*

I've put together a pair of test programs using the routines shown in this article so you can experiment with the different conversion methods. ***[Editor's Note:*** *Software for this article is available for downloading from the Circuit Cellar BBS and on Circuit Cellar INK Software On Disk #9. For information on downloading and ordering software, see page 78.]* If you spend enough time with them, you'll see where they work, where they fail, and ways to improve them for your projects. . . which is, after all, what this column is all about.

Ed Nisley is a member of the CIRCUIT CELLAR INK engineering staff and enjoys making gizmos do strange and wondrous things. He is, by turns, a beekeeper, bicyclist, Registered Professional Engineer, and amateur raconteur.

IRS

225 Very Useful
226 Moderately Useful
227 Not Useful

UPDATE:
Build an 87xx Programming Adapter

by Jeff Bachiochi

Programmable microcontrollers require less board space than nonprogrammable microcontrollers because they don't need external address latches and EPROMs. The problem is, not only do programmable microcontrollers cost two or three times as much as the nonprogrammable versions, they don't fit into a normal EPROM programmer.

Steve Ciarcia presented an article describing a Serial EPROM Programmer (SEP) in the October 1986 issue of BYTE. The design gives the user complete control over the parameters of the ZIF programming socket through its use of DATA statements within the part of the operating system written in BASIC. These programmable parameter statements make it possible to create a simple, low-cost adapter for programming 87xx microcontrollers.

Projects are often created out of necessity. For example, the Mandelbrot Engine required up to 256 8751 microcontrollers to be programmed.

(Editor's Note: See the October, November, and December 1988 issues of BYTE for details of this project.)

Before the Mandelbrot Engine, we had no need for an EPROM programmer that would also program microcontrollers. When work started on the Engine, we quickly reached a point where we could not continue without a way to program the EPROM on an 8751: We looked around for a programmer that we could either buy or modify, and found that the SEP seemed perfect for the job. We began the process by building a small plug-in adapter to adapt a 40-pin ZIF socket to the 2%pin programming ZIF socket of the SEP.

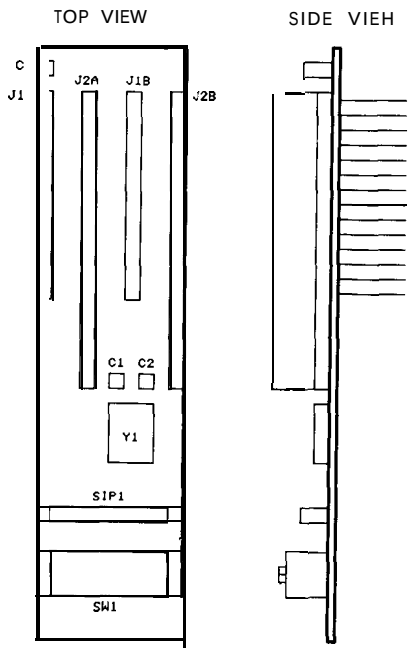


Figure 1—*The* ten *components required for the adapter lay out very compactly on a piece of vector board.*

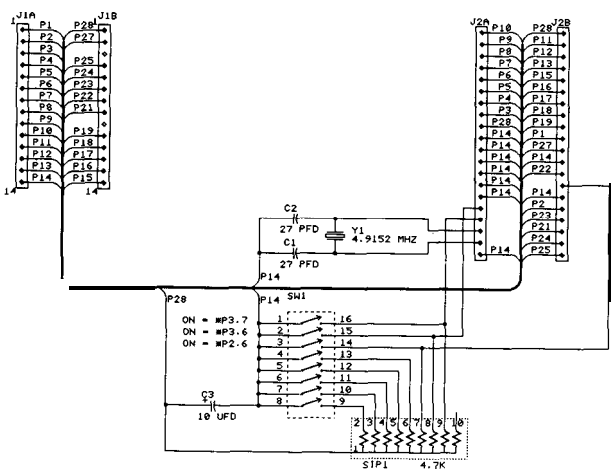


Figure 2-The *schematic for the modification* shows a *4.9152-MHz* crystal in *place. Note that any crystal between 4 and 6 MHz may be used for the project.*

The 8751 programming adapter is made on a 1" x 4" piece of vector board. Cut the centers out of a 28-pin wire-wrap and a 40-pin socket to make individual strips. Alternately mount the sections on 0.300-inch centers as shown in Figure 1 (pins l-14,0.300" space, pins l-20, 0.300" space, pins 15–28, 0.300" space, pins 2140). A 28-pin wire-wrap socket's long leads extend through the vector board to make the connection to the SEP's ZIF socket. (A second socket may be stacked downward if additional clearance between the programming adapter and the SEP is required.) A 40-pin ZIF socket maybepluggedinto the standard 40-pin socket strips later to make insertion and removal of the microcontroller easier.

The microcontroller needs an external oscillator to transfer data internally. Any crystal from 4 to 6 MHz and two 27-pF capacitors are all the components necessary to drive the oscillator. In addition, 87xx programmable micros have various forms of protection against code theft. A set of rocker switches and some pull-up resistors comprise the circuitry required to allow access to all such protection features.

Place thepartson thevectorboard as shown in Figure 1. Figure 2 is the

In BASIC:

```
170 DATA 00001,000H,00,000H,000H,000H,000H,000H,0,00,0,00,0
```

In tokenized form in RAM:

```
(24C0H)
start hex ⬇                          start ASCII ⬇
 24C0H OOH AAH 9CH 30H 30H 30H 30H 31H - . . . 0 0 0 0 1
 24C8H 2CH 30H 30H 30H 48H 2CH 30H 30H - , 0 0 0 H , 0 0
 24D0H 2CH 30H 30H 30H 48H 2CH 30H 30H - , 0 0 0 H , 0 0
 24D8H 30H 48H 2CH 30H 30H 30H 48H 2CH - 0 H , 0 0 0 H ,
 24E0H 30H 30H 30H 48H 2CH 30H 30H 30H - 0 0 0 H , 0 0 0
 24E8H 48H 2CH 30H 2CH 30H 30H 2CH 30H - H , 0 , 0 0 , 0
                    (24F5H)
                   end hex ⬇          end ASCII •
 24F0H 2CH 30H 30H 2CH 30H 0DH 37H 00H - , 0 0 , 0 . 7 .
```

Figure 3— *When you begin modifying yoursystem EPROM, look for the starting addresses in the* setup data shown here.

schematic. A spot of super glue will hold things in place before you do any soldering, but be careful: the thinner versions of this glue like to wick up into IC sockets, gluing their contacts shut. Use small insulated wire (e.g., wire-wrap wire) to make all the interconnections on the bottom of the vector board.

WARNING: The SEP *can* damage any microcontroller *left* in *the ZIF socket uponinitialpower-uporsystem reset. Do not insert a microcontroller into the adapter ZZF socket until the system is initialized with the proper TYPE selection.* Do not *turn the* SEP *off until you have removed any microcontroller from the adapter ZIF socket.*

Programming 87xx microcontrollers on the SEP with the adapter

Turn on the SEP and make the proper TYPE selection. Insert the programming adapter into the SEP making sure pin 1 is aligned properly. The green light should be on. Insert the microcontroller (again, align pin 1). All programming should be done at Normal programming speed, *not* Intelligent or Fast modes.

To program code into a microcontroller:
- Move the code to be programmed into the SEP
- Set the adapter switches to CODE
- Select the Program function

To verify code in a microcontroller:
- Move the code to be verified into the SEP
- Set the adapter switches to CODE
- Select the Verify function

To read code from a microcontroller:
- Set the adapter switches to CODE
- Select the Read function

To program the security table:
- Move the mask code to be programmed into the SEP
- Set the adapter switches to P. Table
- Select the Program function (program first 32 bytes)

In BASIC:

```
260 DATA $(10)="USER 1 ":$(11)="USER 2 ":$(12)="USER 3"
```

In tokenized fonn in RAM:

```
                    (26C6H)
                   start hex ⬇        start ASCII ⬇
 26C0H 32H 2EH 35H 56H 22H 0DH 42H 01H - 2 . 5 V " . B .
 26C8H 04H 24H E0H 31H 30H 29H EAH 22H - . $ . 1 0 ) . "
 26D0H 20H 55H 53H 45H 52H 20H 31H 20H -   U S E R   1
 26D8H 20H 20H 20H 20H 22H 3AH 24H E0H -         " : $ .
 26E0H 31H 31H 29H EAH 22H 55H 53H 45H - 1 1 ) . " U S E
 26E8H 52H 20H 32H 20H 20H 20H 20H 20H - R   2
 26F0H 20H 22H 3AH 24H E0H 31H 32H 29H -   " : $ . 1 2 )
 26F8H EAH 22H 55H 53H 45H 52H 20H 33H - . " U S E R   3
                    (2707H)
                   end hex ⬇        end ASCII ⬇
 2700H 20H 20H 20H 20H 20H 20H 22H 0DH -             "
```

Figure 4— *The beginning addresses for the display data to be altered should appear as in* this *frame.*

ADDR	ORIGINAL HEX	ORIGINAL ASC	CHANGE TO HEX	CHANGE TO ASC
24C0H	00H		+	
24C1H	AAH		+	
24C2H	9CH	.	+	
24C3H	30H	.	+	
24C4H	30H	0	38	8
24C5H	30H	0	37	7
24C6H	30H	0	35	5
24C7H	31H	1	31	1
24C8H	2CH	,	+	,
24C9H	30H	0	30	0
24CAH	30H	0	31	1
24CBH	30H	0	30	0
24CCH	48H	H	+	H
24CDH	2CH	,	+	,
24CEH	30H	0	32	2
24CFH	30H	0	38	8
24D0H	2CH	,	+	,
24D1H	30H	0	37	7
24D2H	30H	0	38	8
24D3H	48H	H	+	H
24D4H	2CH	,	+	,
24D5H	30H	0	32	2
24D6H	30H	0	36	6
24D7H	48H	H	+	H
24D8H	2CH	`	+	,
24D9H	30H	0	30	0
24DAH	30H	0	43	c
24DBH	30H	0	45	E
24DCH	48H	H	+	H
24DDH	2CH	,	+	,
24DEH	30H	0	30	0
24DFH	30H	0	43	c
24E0H	30H	0	45	E
24E1H	48H	H	+	H
24E2H	2CH	,	+	,
24E3H	30H	0	30	0
24E4H	30H	0	32	2
24E5H	30H	0	33	3
24E6H	48H	H	+	H
24E7H	2CH	,	+	,
24E8H	30H	0	30	0
24E9H	2CH	,	+	,
24EAH	30H	0	35	5
24EBH	30H	0	30	0
24ECH	2CH	,	+	,
24EDH	30H	0	30	0
24EEH	2CH	,	+	,
24EFH	30H	0	30	0
24F0H	30H	0	30	0
24F1H	2CH	,	+	,
24F2H	30H	0	30	0
24F3H	0DH	.	+	

Figure 5—*These changes are made to the setup data shown starting in Figure 3.*

To program lock bit 1 or 2:

Set the adapter switches to Lock 1 or Lock 2

Select the Program function (program first byte)

Altering your system EPROM—Version 2.0

If you want to program an 8752 or an 87C252 (87C51FA), nothing special must be done since these devices program like 2764As. However, if you want to program 8751s, 8751BHs, or 87C51s then you must change a TYPE SELECTION (SEP ROM version 1.0-1.6) or add a new USER ENTRY (SEP ROM version 2.0).

[Editor's Note: There are *significant differences between version 1 .x and version 2.0* of *the* SEP system EPROM. If you *have version 2.x and would like more specific instructions on how to modify 2.x* for use *with the programming adapter, send a self-addressed stamped envelope fo:*

87xx Programming Adapter
Circuit Cellar INK
4 Park St.
Vernon, CT 060661

To alter the system EPROM you must first get the system EPROM's code into the buffer. Version 2.0 has a selection in the main menu for accomplishing this. This version also has six reserved entries, USER l-USER 6, for adding new device types. In this example, USER 1 will be eliminated to make an entry for 8751s.

Two areas of the system EPROM will be affected: Setup data and Display data. First locate the Setup data using the SEP's DUMP function and write down the starting and ending addresses as shown in Figure 3. Similarly, use the DUMP function to find the starting and ending addresses for the Display data area as shown in Figure 4.

Next, select the CHANGE function and enter the starting and ending addresses for the Setup data you found above. Then enter the changes that are shown in Figure 5. Likewise, change the Display data as shown in Figure 6.

ADDR	ORIGINAL HEX	ORIGINAL ASC	CHANGE TO HEX	CHANGE TO ASC
26C6H	42H	B	t	
26C7H	01H	.	t	
26C8H	04H	.	t	
26C9H	24H	$	t	
26CAH	EOH	.	t	
26CBH	31H	1	t	
26CCH	30H	0	t	
26CDH	29H	)	t	
26CEH	EAH	.	t	
26CFH	22H	"	t	
26DOH	20H		20	
26D1H	55H	U	38	8
26D2H	53H	S	37	7
26D3H	45H	E	35	5
26D4H	52H	R	31	1
26D5H	20H		20	
26D6H	31H	1	20	
26D7H	20H		20	
26D8H	20H		32	2
26D9H	20H		31	1
26DAH	20H		56	V
26DBH	20H		20	
26DCH	22H	"	t	
			(all+)	
2706H	22H	"	+	
27078	ODH	.	t	

Figure 6-Change the display data shown starting in Figure 4 as indicated in this table.

For: 8751 Must add USER ENTRY

Function	SW3 setting
Code	ON
Lock 1	OFF

```
USER ENTRY for 8751:
  setup data
   DATA 8751,010H,28,78H,26H,0CEH,0CEH,023H,0,50,0,00,0
  Display data
   " 8751   21V"
```

```
For:  8752BH                Program as 2764A
For:  87C252BH (87C51FA)    Program as 2764A
For:  8751BH                Must add USER ENTRY
For:  87C51                 Must add USER ENTRY
```

Function	SW1	SW2	SW3
Code	OFF	OFF	ON
P. Table	ON	OFF	ON
Lock 1	OFF	OFF	OFF
Lock 2	ON	ON	OFF

```
USER ENTRY for 8751BH and 87C51:
  Setup Data
   DATA 8751,010H,28,78H,26H,0CEH,0CEH,025H,0,50,0,00,0
  Display data
   " 87C51  12.5V"
```

Figure 7—The 87xx Programming Adapter can be made to work with a variety of devices by changing the switch settings shown here.

Circuit Board
1 1"x4" piece of vector board (preferably with holes predrilled on 0.100-inch centersand individual pads on each hole)

IC Sockets
2 28-pin wire-wrap sockets (one cut into two 14-pin strips J1A & J1B, the second used as an extension for more height if necessary)
1 40-pin ZIF (Zero Insertion Force) socket
1 40-pin IC socket (must allow the ZIF to plug into it) (cut into two 20-pin strips J2A & J2B)

Discretes
2 27 pF capacitors
1 10 µF tantalum capacitor
1 4.9152-MHz crystal (anything from 4 to 6 MHz)
1 4.7k ohm pull-up SIP (or individual resistors)
1 B-position slide or rocker switch (only three positions are used)

Miscellaneous
Wire-wrap wire for interconnections
Cyanoacrylate glue for tacking down parts

Figure 8—The parts required for the adapter are simple, and will require a minimal cash outlay by most builders.

This completes the changes. Use the DUMP funtion to check that all changes were made correctly. Select the 27C128 (27128) EPROM type and insert a blank 27C128 (27128) in the SEPs programming socket. Program the whole EPROM and your new system EPROM should be ready to use. Remember to remove the EPROM from the ZIF socket before turning off the power to replace the system EPROM with the new one.

The adapter can also be used with other EPROM programmers to program 8752BHs and 87C252BHs (87C51FAs) by setting the programmer up for 2764As. The other processor types discussed requirealterations which we've accomplished through the flexiblity of the SEP, and not through the complexity of the hardware for the adapter. A list of programadapter switch settings for working with other devices is given in Figure 7. Figure 8 shows the programming adapter's parts list. Cost for the parts should be minimal.

It is possible, of course, to buy completely hardware-based adapters for programming programmable microcontrollers. Most such adapters will cost at least $100 and will be limited in their application to one family of micros. I like the fact that, in this case, hardware and software complement each other quite nicely. Neither is more important than the other, with thelowest-cost solutionbeing a 50/50 mix of both. ✣

IRS

228 Very Useful
229 Moderately Useful
230 Not Useful

CONNECTIME

Excerpts from *the Circuit Cellar BBS*

THE CIRCUIT CELLAR BBS
300/1200/2400 bps
24 hours/7 days a week
(203) 87 1- 1988 — 4 incoming lines
Vernon, Connecticut

The message base of *the Circuit Cellar BBS in now available on disk.* *See page 78 for details.*

With the upgrade to version 2.1M of TBBS *(The Bread Board System), one of the most obvious improvements was to the file system. I mentioned a few issues ago that I'd be implementing the new* file *system features, and have been quite* successful at *doing so since then. Using the old software, the only way to group related files into separate file areas was to define individual menus for each* of *those areas. Finding a certain file involved coursing through menus in an attempt to find the correct subarea. Plus, checking* for new *files involved a good memory (yours).*

The new software solves these problems as well as some others not mentioned. You start off at the main menu and select the <F>iles area. At the next menu you tell the system you want to <D>ownload a file. *Once you're into the download section, you arepresented with a list* of *14 file areas. Areas include files* for *articles which have appeared in CIRCUIT CELLAR INK, files related to Electrical Engineering, cross-development tools, and newly uploaded* files *which don't* have a *home yet.*

Upon selecting one of *the subareas, you are given a list of commands available. Commands include <L>ist the available files, <D>ownload a file, show a list of <N>ew files, <E>xamine the contents* of *an ARC file, and go to a* new *<A>rea.*

The <L>ist command allows you to include a partial or a complete file name to restrict the files shown. For example, type "L.ARC" to list just the ARC files available in* the *selected subarea.*

The <N>ew command *will actually show you all* files *uploaded after a certain date in all subareas. Used without any parameters, the <N>ew command will list all files uploaded since your last log-on. Used with a date, it will show* all files uploaded since that date.

Once you know what file you want to download, you'll encounter another nice addition that version 2.1M introduced: moreavailablefile transferprotocols. In addition to ASCII, XMODEM, and YMODEM (XMODEM/1K), we now have available YMODEM Batch (True *YMODEM), Kermit, and SEAlink.*

Finally, help is available by using the <H>elp command. Of course, I'm always available via *Email to answer any specific questions, so if you've been putting off trying to* download files because the thought of trying to figure *out just what a "protocol transfer" really is, give it a t y on your next call. It really is easy once you've tried it.*

The key to any successful hardware project is actually building the circuit and having it work. There are numerous methods that can be *used* for *building prototype circuit boards, each with its own good and bad points, as we find out from the first discussion.*

Msg#:11923
From: MARK BALCH To: STEVE CIARCIA

Steve, I will start building a project that I am designing in about a month or so. Can you tell me how it is best to prototype a simple 8-bit microprocessor circuit? Should I wire-wrap, breadboard, or solder it? In your opinion, what works best for Circuit Cellar projects like the ones you did back in 1980 and '83?

Msg#:11947
From: STEVE CIARCIA To: MARK BALCH

Most people like wire-wrap. Personally, I don't. I still point-to-point solder (using PC board sockets and wire-wrap wire) all my projects. That way they are neater and smaller.

Msg#:12165
From: MARK BALCH To: STEVE CIARCIA

Thanks. I guess I'll stay with soldering then. What do people see in wire-wrap? From what I see, it can get extremely messy with tangled wires. I remember seeing a back issue from 1983 or so when you addressed that topic and showed pictures of the backs of your boards. I just wanted your latest opinion.

Msg#:12535
From: BILL CURLEW To: MARK BALCH

I think the only reason Steve gets away with soldering is because God had intended him to be a brain surgeon. His boards look more like works of art than protoware.

I, on the other hand, am a big banana when it comes to point-to-point wiring. Wire-wrap has allowed me to continue dabbling without excess pain. Just for some perspective, in 1975 I decided to build an Altair 8800A. Since I was a poor boy at the time, I

bought the manuals for $30.00 and WIRE-WRAPPED THE WHOLE SYSTEM, INCLUDING THE FRONT-PANEL ASSEMBLY.

Msg#:12266
From: KEN DAVIDSON To: MARK BALCH

For doing small circuits with many discrete analog parts, point-to-point wiring with a soldering iron works the best. For wiring an alldigital circuit where you have mostly socketed chips, I find wire-wrapping is much faster, easier to do, and easier to change. True, you end up with a board that is a little fatter, but when you want to do a lot of playing with a circuit before casting it in copper, it works well. The first cut of the BCC180 was mostly wire-wrapped with solder connections to the 68-pin PLCC socket. The result was that the first prototype PC board worked on the first try. Wire-wrapping also works better if your soldering techniques leave your finished work looking like it was done using a blow torch.

Msg#:12273
From: BOB PADDOCK To: KEN DAVIDSON

Have you ever looked at Vero Speedwire stuff? It has the same advantages as wire-wrapping, but without the thickness.

Msg#:12278
From: KEN DAVIDSON To: BOB PADDOCK

One of the problems with the thickness of wire-wrapping is you usually can't plug a board into a backplane without taking up several slots. In a junk box, I found some sockets that could be glued onto the top of a board that had the pins bent around and sticking up on the same side of the board as the components. You couldn't fit as many parts on the board, and you had to channel the wires between the sockets, but it made a nice thin board.

Msg#:12291
From: HENRY MINSKY To: BOB PADDOCK

I have also found that Speedwireis the way to go for prototyping. It seems like there ought to be something better, but Speedwire seems to be the best for quick prototyping. One thing that would be a good addition would be whole Speedwire sockets. I have only seen spools of individual pins, which you have to insert in the proto board yourself. Has anyone seen prefabricated Speedwire DIP sockets?

Msg#:12311
From: BOB PADDOCK To: HENRY MINSKY

Iknowwhatyoumean. We make our own Speedwire boards and buy the pins from Vero. Three cents per pin doesn't seem like much until you have to put 1500 of them on one board. Someone in our production department found an interesting way to install them. Instead of inserting the pins one at a time, they insert them a row at a time, then use our metal break to press them into the PC board; sure cuts down on construction time.

Msg#:12839
From: MARK BALCH To: BOB PADDOCK

But aside from Speedwire, would you say that wire-wrapping is the best to prototype? In terms of ease of use and speed.

Msg#:12877
From: BOB PADDOCK To: MARK BALCH

Wire-wrapping is great for digital projects, but for anything that has a lot of analog components it can be a curse. You have to solder some type of wrappable pin to the component so you can use **your** wire-wrap tool on it, so you are just as far **ahead** to solder the wire in the first place.

Don't use wire-wrap wire for your power bus. Use something like at least 22-gauge wire for a bus, then run power to your sockets using your red & black wire-wrap wire from these buses (something I learned the hard way!).

Msg#:12503
From: RON LEBLANC To: MARK BALCH

A little background. I have been designing and prototyping microprocessor-based systems since early 1974. The first computer I built was an 8008-based unit. Since then I have prototyped four or five dozen micro-based systems. In short, I've been doing it a while and know of what I speak. Everyone of those projects except three were all wire-wrapped.

You will find, and already have from what I can see of the other messages posted here, that there are two camps on what method to use. I use Vector perf board as a base, Vector bus strips for running power, T49 pins for mounting resistors, caps, etc., and J pins for test points and power connections. And of course, wire-wrap sockets for ICs. The address for Vector is:

Vector Electronics Company
12460 Gladstone Ave.
P.O. Box 4336
Sylmar, CA 91342-0336
(818) 365-9661

Call them; they will send you a catalog full of tools, sockets, plug boards, and so on. No, I'm not associated with them in any way except as a long-time customer. I believe if someone makes a good product you ought to tell others about it.

Why wire-wrap? I have found wire-wrapping to produce the cleanest, neatest, and most well-behaved prototypes (less crosstalk, much cleaner power). It is fast! At least for me much faster than point-to-point soldering. It is also very easy to modify. Connections are made at 20 or more points and therefore you don't get loose or flakey connections because of cold solder joints. It takes all of about 10 minutes to learn to do correctly. As to why commercial companies wire-wrap:

1) The connections are good for 20+ years. (Hint, the black corrosion on the exterior of the wire after a time DOES NOT have any effect. The connections are made between the pin corners and the wire,and theyareverygas tight and corrosion resistant.)

2) Ever try to design a complex project and wire the prototype point-to-point with solder? What a mess! Try making a significant modification to that! (Bell Labs has done extensive testing on this. Solder connections are only good for eight or so years.)

1 know, who's going to keep a project for twenty years? That's not the point. The point is the connections are much more reliable.

Bad points? As always, there are some. Extra cost is a big one. Wire-wrap sockets are not cheap.

Msg#:12516
From: MARK BALCH To: RON LEBLANC

Thanks a lot for your help. You have thoroughly answered my question. I already have Vector's catalog and have seen some of their stuff. One thing I don't like is that their boards seem very expensive. I'll check their wire-wrap equipment.

So do you think I should buy a cheap wire-wrap starter set and some sockets and wrap a "dummy" board to see if I like it?

Msg#:12541
From: RON LEBLANC To: MARK BALCH

You should rarely take anyone's word as the absolute answer to anything. Yes, by all means, experiment with wire-wrapping. There are some relatively inexpensive hand wrapping and unwrapping tools made by a company called OKI. These should not cost more than 5 to 10 dollars each. The only other things you need are wire-wrap sockets and a set of wire strippers. I would suggest that you get some wire strippers called T strippers. These are simple, cost less than $10, and work just fine. Forget spending any large amounts until you get the "feel" of wire-wrapping. You may or may not like it.

Of course, you'll need some wire-wrapping wire. Get at least three 100' rolls: one red for power, one black for ground connections, and another white for signal connections. You will find color coding the connections very useful when tracing the wiring. I always wire all of the power connections first and then proceed to do the signal wiring. Strip about 1" of insulation off the end of the wire. This will give you a good four to five complete turns about the pin. Use what is called a modified wrapping tool if you have a choice. The modified wrap puts about one turn of insulated wire about the pin at the base of the connection. I have found this to make the best connections.

Finally, do try to do another project using the point-to-point wire and solder method. You can then compare the two for yourself. I much prefer wire-wrapping, but not all people do. Experiment! That's what it's all about, isn't it?

Msg#:12548
From: MATT OLSON To: MARK BALCH

I would like to add my two cents on the topic of wire-wrapping. I have been wire-wrapping for a few years and nothing can beat prototyping with it for speed and flexibility. Granted, the profile will be higher, but surface area can be just as compact as a soldered board if not more. A few good techniques should be followed for a good clean board.

Plan the layout carefully, as should be done with any board. Use prestripped cut lengths of different colored wires, using only the length that will reach the two connections. This can eliminate the "rats nest." This wire can be purchased through a number of places (Digi-Key, Specialized Products). There are also a few low-cost hand-cut-and-strip-type products that workquite well, although that is more time consuming. Route wires carefully. Run wire down the center IC or between them, as opposed to in between their pins. Use different colored wire to identify address bus, data bus, control lines, I/O ports, and so on. Use wire-wrap ID labels and write part or reference numbers on them.

Use a "modified" type bit for your WW gun. This can eliminate broken wires at the base of the pin. I have been using the Radio Shack prototype board #276-188 for a number of projects. The board has a ground plane on the component side and individual solder pads on the back, along with an edge connector. I usually solder the pins of individual components and two pins of an IC to the pads so that nothing comes loose. Sprinkle a few ground and power pins throughout the board using a heavier gauge wire soldered to connect them. Route component connections for these to the closest pins, and avoid daisy-chaining power and ground. Ofcourse,useamplebypassanddecouplingcapacitors.

I did have an experience with a just-wrap slit-and-wrap-type gun, in which the wire inside the insulation was being broken, but the insulation was not. This turned out to be a total nightmare and the entire board and all the labor had to be scrapped. It may have been that the gun and bit I was using were faulty.

Msg#:12836
From: MARK BALCH To: MATT OLSON

Whew. Thank you very much for those ideas and hints. Interesting ideas that you have. I am sure they will help me to make my first project easier. It looksvery expensive, though. I'm gonna try using a manual wrapper for my first project because I will be giving wire-wrapping a "test run." I already have four colors of wire: two for power and two for different signals. I think I'll buy a fifth color to make the signals even easier.

Do you recommend buying one of those wire kits that has many precut lengths in assorted colors? And do you know of a good place for me to get a starter kit for wrapping? If I can't find another place, I will order from Jameco because they are very nice people and have good products (from what I have ordered from them).

Msg#:13018
From: BILL CURLEW To: MARK BALCH

On the subject of "kits" with different wire lengths: I have found that wrapping goes MUCH QUICKER if you have prestripped lengths available, but the commercially precut stuff is too darned expensive for me.

What I did was to figure out the three or four sizes I use most often. Then I took some of those DIP carriers (the plastic U-shaped tube things) and cut them to size. I take the spooled wire and wrap it around the form, cut the ends, and strip the resultant wires.

I usually make up a bunch after designing the circuit I'm going to build, and fill in as needed.

Msg#:12840
From: MARK BALCH To: KEN DAVIDSON

Is it really easy to modify a connection? It always looked hard because of the turns of the wire and the layers.

What do your boards look like when you finish them?

Msg#:12893
From: KEN DAVIDSON To: MARK BALCH

As long as you try to keep it to a limit of two wraps per pin, you won't end up with countless layers to unwrap should you want to move the connection on the bottom of the pin. And as long as you cut the wires close to their proper length, you won't end up with a board that looks like a rat's nest. Don't cut the wires so short that they are like guitar strings once you're done stretching them to make the connection, but don't leave so much extra that you have to route it three times around the board to take up the slack. Even though the finished board may look like a mess of wires going everywhere, it's actually very easy to trace a wire to make sure you have a proper connection. Just use a pair of pointed tweezers and you can easily follow any wire through its entire path.

We've all seen the large shoplifting detection systems installed in the entrances of stores. Such conspicuous presence *is bound to spark a good* electronics *experimenter's curiosity, as it did in this discussion.*

Msg#:12183
From: DALE REID To: ALL USERS

I'm not about to circumvent the various sensors that stores use to prevent their products from walking past the cash registers, but have some questions on the technology used to make them work.

There seem to be two kinds available. The first is a little strip that B. Dalton or Software Etc. has pasted on the back of every piece of stuff they have, and has to be "deactivated" in some way. I assume that this is magnetic, and it always bothers me to have them do that to anything I have purchased that has a disk in it. This seems to be like the little strips that the libraries put in the books to make sure you have checked them out. But if it is a magnetic strip, how do the little blades of the tunnel you have to walk through pick up the strip? Why doesn't my magnetic screwdriver&off everyone1 walkthough? It seemstobeagiant hall-effect detector, but can they be tuned to the strip somehow?

The other type is one is a tag a few inches long, has what appears to be a flat bronze strip in it, with an ordinary diode across it, and makes me think that it will change an RF field when passed through; something like a magnetometer.

Anyone know in general how these things work? As I say, it is like asking how a police radar works: someone will always be wondering why we want to know. I detest the "need to know" attitude of the military, and have always had a general curiosity of just how it works, much to my poor old mother's concern when I started tearing things apart. I'm sure about 95% of the users of this board have a similar curiosity. Thanks.

Msg#:12237
From: MARK LAMPKIN To: DALE REID

The systems you are curious about are two totally different technologies. The first system is magnetic. The strip is actually magnetized by a little machine that they have at the counter. When you purchase the item, they demagnetize the strip. The portal at the entry/exit is a large proximity switch (i.e., the portal is a big antenna that is part of an oscillator tuned to a specific frequency). When the mag strip is in the field of the antenna, the oscillator is detuned proportionally by the size of the strip's magnetic permeability. Those strips are not cut to random lengths; they are all approximately the same size and have the same magnetic permeability factor. The oscillator will deviate a specific amount from the center frequency in the presence of the strip (resulting in the alarm sounding).

The second system is a microwave system. The diode on the strip is a strip-line transmitter. The "tag" senses the presence of a certain frequency and retransmits the signal at a different frequency. The main antenna (i.e., transponder) detects the retransmitted signal and presto! Alarm.

Both are quite simple systems, but effective. The actual detection ratio is only approximately 72-78% accurate, but it's the deterrent effect of the system on the shopping public that makes it work. They are psyched out by the visual effect.

Sensors for measuring temperature, wind speed,and wind direction are relatively easy to *find when* constructing a *home weather center. However, humidty and pressure sensors are* much *more* difficult to *find, as we* find out *in the following discussion.*

Msg#:13275
From: JACK DILLON To: ALL USERS

I am building a home weather center and need information on a low-cost humidity sensor to measure relative humidity. I'm also looking for a way to measure barometric pressure. I am using an A/D converter with 0-5-volt inputs.

Msg#:13538
From: FRANK KUECHMANN To: JACK DILLON

Pressure transducers suitable for barometers are made by Sen Sym, 1255 Reamwood Ave., Sunnyvale, CA 94089. Sen Sym's "1989 Solid State Sensor Handbook" contains app notes, barometer circuits, design discussions, and so on using their line of sensors. App note SSAN-29 is a barometer design discussion.

Motorola has at least two pressure transducers similar to Sen Sym's parts (specs so similar I suspect Motorola buys them from Sen Sym rather than making them).

January '89 issue of ***Modern Electronics*** featured a barometer project using a Sen Sym transducer feeding into an ADC.

Humidity sensor: Mepco/Electra #232269190001 capacitive humidity sensor. Made by Philips Electronics (Netherlands). Call (817) 325-7871 for name/address of nearest distributor.

February '86 ***Radio-Electronics*** had a humidity sensor project.

A small kit of parts to make a variable-frequency humidity sensor using the Philips capacitor in an NE555 oscillator circuit can be obtained from Vernier Software, 2920 SW 89th St., Portland, OR 97225, (503) 297-5317.

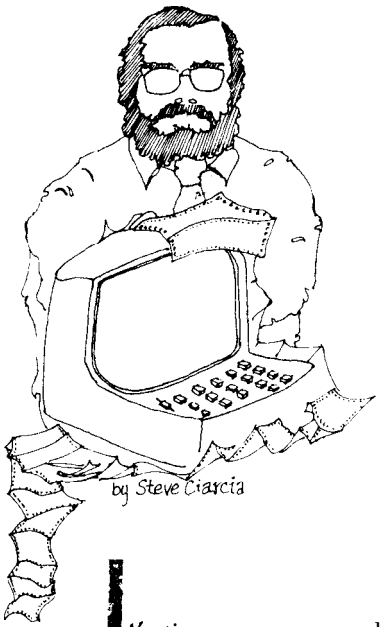
The kit includes the capacitor itself, NE555, and miscellaneous parts to hook up to an Apple II-series computer's gamecontroller port. The parts could easily be used to work with about any l-bit input port you can read with machine code.

Cost of the kit is something like $25 (that humidity-sensitive cap is pricey); contact Vernier. Vernier sells a book called 'How to Build a Better Mousetrap" with a humidity monitor (Apple II) as one of the projects; the discussion of measurement is very good. Worth it even if you don't have an Apple II.

The Circuit Cellar BBS runs on a 10-MHz Micromint OEM-286 IBM PC/AT-compatible computer using the multiline version of The Bread Board System (TBBS 2.1M) and currently has four modems connected. We invite you to call and exchange ideas with other Circuit Cellar readers. It is available 24 hours a day and can be reached at (203) 871-l 988. Set your modem for 8 data bits, 1 stop bit, and either 300, 2200, or 2400 bps.

IRS

231 Very Useful
232 Moderately Useful
233 Not Useful



Ctrl

STEVE'S OWN INK

The Good Old Ways

It's time someone spoke up in support of the old ways. I've been hearing a lot about the "new age" in everything from music to computer engineering, and I've come to the conclusion that traditional styles and methods need better public relations. I'm not going to spend a lot of time trying to convert you to my kind of music. Let's just say that the Doors were a great band and leave it at that. What I really want to get to is this new-fangled notion that you can't be a real engineer unless you're also a great programmer.

The arguments for engineer-as-programmer usually start out something like this: Modern microprocessors and microcontrollers have much more power and flexibility than the chips of yesteryear. They have so much more, in fact, that you can replace most of the silicon you used to need with a few (hundred) lines of simple code. Think of how much simpler (and smaller and cheaper) the design can be without all of those pesky peripheral chips to worry about. The arguments then proceed to their knockout blows: expense and flexibility. Someone, somewhere, decided that programming is faster than circuit design (they certainly didn't talk to me before reaching this conclusion), and it's "intuitively obvious" that using a full-bore processor allows you to make your hardware do new tricks with only a software change. Eight.

I'm going to counter the knockout blows first. Programming might be faster than designing a circuit if you're used to thinking in software; I'm not. I think in gates and resistors and flip-flops. If I'm forced to do so, I can translate from hardware into software, but it takes a long time and a lot of aggravation. For me, it's cheaper to just do most jobs in silicon and solder. It's also cheaper to not put flexibility you'll never use into every circuit. Now don't get me wrong, I like microcontrollers as much as the next guy. But if all you want to do is dim an LED, why not just use a simple resistor? Sure, a PWM controller would be more flexible and offer an infinite range of brightness, but at a cost of wasted time, money, and computing power. Finally, it's true that you can replace a lot of older hardware with assembly code. If I have to put a circuit into a very small box, I usually consider software in place of hardware. The fact is, most designs don't have to go into a matchbox, and I am more comfortable with the predictability and reliability of something I can touch and troubleshoot with a scope.

I don't want to sound like a throwback. I can program, and I work with some engineers who are great programmers. It's just that I resent the notion that nothing interesting is happening in hardware. There are many areas where software design is only beginning to catch up to hardware power, and many more areas where a design in silicon is simply the best way to get the job done. So hang on to your soldering irons. Be proud of your wire-wrap tools. And don't be afraid to remind your programmer friends that, without our hardware to run their software, they're left just writing bad poetry.

Steve Ciarcia