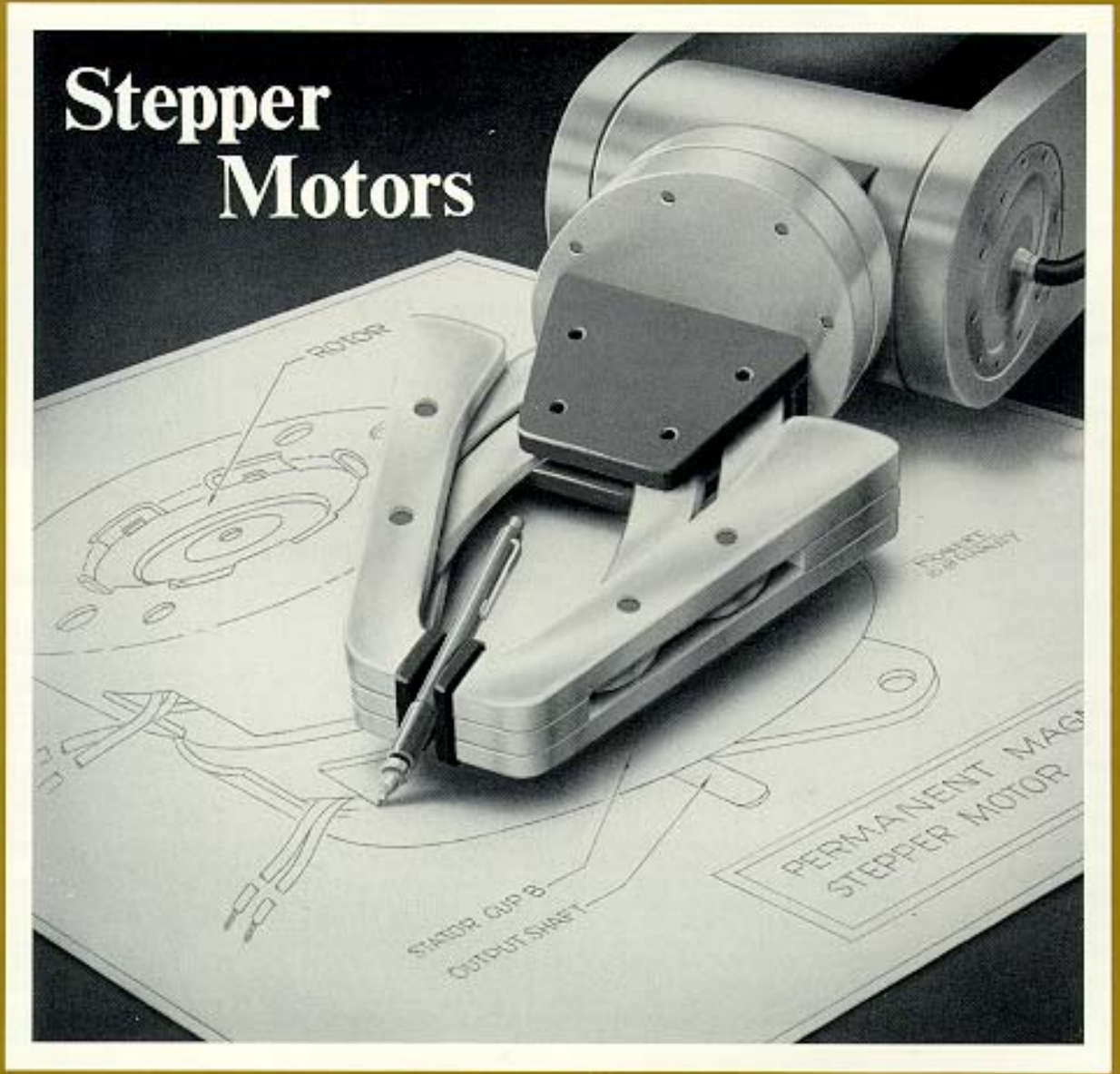
$3.95

Circuit Cellar INK

Ctrl

THE COMPUTER APPLICATIONS JOURNAL



July/August 1988 -- Vol. 1, No. 4

PUBLISHER
Stephen J. Walters

EDITORIAL DIRECTOR
Steve Ciarcia

EDITOR-in-CHIEF
Curtis Franklin, Jr.

EXECUTIVE EDITOR
Harv Weiner

TECHNICAL EDITORS
Ken Davidson
Jeff Bachiochi

CONTRIBUTING EDITORS
Thomas Cantrell
Edward Nisley

CIRCULATION DIRECTOR
Jeannette Dojan

CIRCULATION ASSISTANT
Diane Morey

CIRCULATION CONSULTANT
Gregory Spitzfaden

PRODUCTION MANAGER
Tricia Dziedzinski

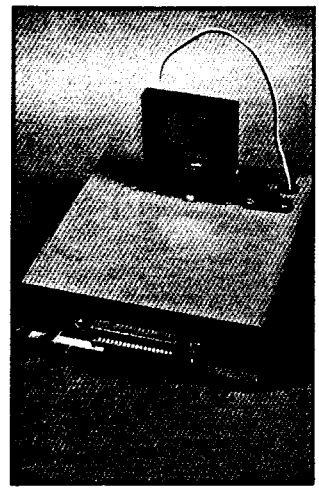
BUSINESS MANAGER
Daniel Rodrigues

STAFF RESEARCHERS
Northeast
Eric Albert
William Curlew
Richard Sawyer
Robert Stek
Midwest
John Elson
Tim McDonough
West Coast
Frank Kuechmann
Mark Voorhees

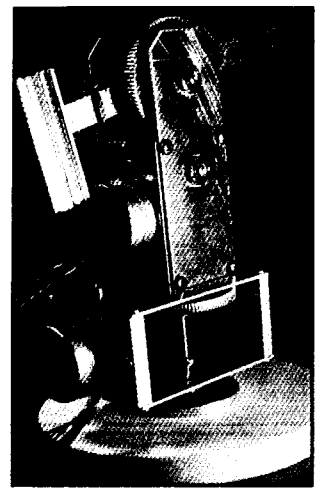
Cover Illustration by Robert Tinney

CIRCUIT CELLAR INK

THE COMPUTER APPLICATIONS JOURNAL



page 5



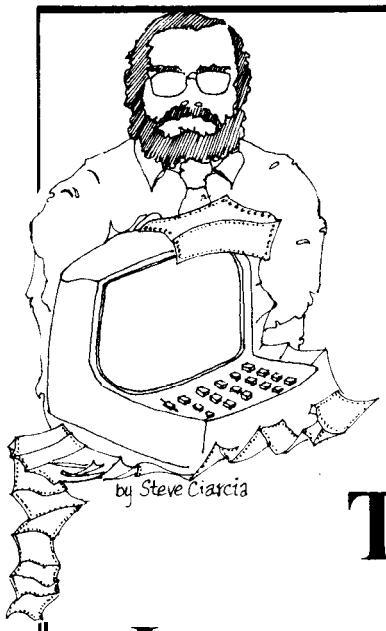
page 49

Circuit Cellar BBS - 24 Hrs. 300/1200/2400 bps, 8 bits, 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 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) charter subscription rate U.S.A. and possessions $14.95, Canada $17.95, all other countries $26.95. All subscription orders payable in US funds only, via international postal money or check drawn on US bank. Direct subscription orders to Circuit Cellar INK, Subscriptions, 12 Depot Square, Peterborough, NH 03458-9909.
POSTMASTER: Please send address changes to Circuit Cellar INK, Circulation Dept., 12 Depot Square, Peterborough, NH 03458-9909.



Ctrl

EDITOR'S INK

The Market-Driven Myth

I'm told by many in this industry that the marketing manager and volume buyer are king. I'm told that the microcomputer industry has "matured" to a point where all the innovation (not to mention the money) comes from end users, not engineers and programmers. I'm even told that there aren't enough people who still build hardware and write software to make it worthwhile for a magazine to concentrate on meeting their needs.

Hogwash.

Certainly, there are many productive microcomputer and microprocessor users who haven't the foggiest notion of how their machines work, but that doesn't mean that no one remembers what registers, counters, timers and pull-up resistors are for. Instead, it points out that there are two groups of people in this field, and their needs and interests are rapidly diverging.

The first group is made up of appliance buyers. They need lots of information about which appliance to buy, but they don't want to be confused by a lot of technical details. If they program, it's in a spreadsheet macro language. Connecting a printer or (for the adventurous) installing an internal modem runs them to the limit of their hardware expertise. Marketing managers and media buyers like these folks, becaused they don't use a lot of jargon and they tend to have needs that can be met with feature updates instead of real innovation. There are a lot of big, glossy magazines that are devoted to helping the buyers find the appliances while giving them a light smattering of technology.

The second group is harder to pin a label on, but it includes people who build their own boards and boxes, people who program in assembly language and C and folks who don't like trusting in "magic" black boxes, but who love making magical things happen with applications they put together. More powerful processors and more sophisticated applications make it more difficult, and more important, for the members of this group to keep up with innovative techniques and creative thinking in application design. That's where we come in. Our sub-logo is "The Computer Applications Journal" and we take that very seriously. Let others talk about how to buy appliances; we want to help you build them.

In the real world, of course, being able to market what you build is important. It's also important to recognize that without the expertise and ability to build applicatons, marketing schemes are just so many empty promises. I've seen the appliances, I've seen the builders, and it's not hard to tell who's really driving the innovation in the industry.

Ctrl

READER'S INK

Letters to the Editor

Dear Steve,

Stupendous! Great magazine! I thought my friends and myself were the only ones offbeat enough to think of stuff like launching 2-liter [plastic] bottles (and instrumenting the gantry no less!). I'm glad I subscribed. Really.

I also have an article request. It's an electronic astronomy suggestion. Please bear with me. Telescopes use clock drives. Clock drives are made by humans. Humans have accuracy limitations (mostly due to production costs). Clock drives, therefore, have to be constantly corrected to make up for imperfections and eccentricities in the gears when making astrophotos. This is known as guiding. It is, needless to say, a pain (especially 1 -hour and longer exposures). They do have one redeeming feature: they use synchronous motors (selsyns). This is what makes a drive corrector so easy to build. It's merely an adjustable (within limits) square-wave inverter.

It occurs to me that one could time the errors (for example, a friend has a 5-minute periodicity in the worm gear that drives the polar gear) and write a machine routine for a small Z80 (or other) controller board to clock the inverter at a user-defined rate (maybe a look-up table). The user could then, using a set of shorthand formulas, create the table, burn an EPROM, and plug it in (i.e., compensate for known errors in his own drive), This would eliminate the need for a keyboard. The whole package need not be very large or complex at all. We have discussed this at considerable length and have not come up with a means of feedback (photomultiplier or whatever) that seems feasible. So, it seems that the error-cancellation-type controller might be best.

If you or your crowd can think of something we didn't, I'd like to hear about it. Anyway, it would make an interesting article. I'm sure there are a lot of amateur astronomers out there that would be interested in such a beast.

One last piece of input you may not have. Due to atmospheric refraction, this would not work near the horizon (unless you have feedback from the image), but it would work fine near the zenith. It would at least cut the strain of the guiding chore (maybe eliminate it depending on magnification used and exposure time). Also, a ROM socket and routine to add in corrections for moon or planetary motions might be nice. So much for article suggestions.

Once again, I have to say, your magazine is great reading (and useful too). Oh, yes, an egg has a ***yolk*** not a ***yoke*** (it still makes a mess of the turntable regardless .). Had you ever considered trapping the yolk in the hole in a 45 record? What effect would the spindle (or lack of one) have on the integrity of the sac? Would the concentric ridges in a turntable mat make the white spread outward in jumps? It'd make a good article for "The Amateur Scientist" in ***Scientific American ...***

C. Sherman
Hesperia, CA

Some things are better left to Scientific American. ***Sorry about the spelling mistake but we is all enjeneers 'round hea <sic>. Please note that it says "editorial director" next to my name and not editor. Ain't my job man. Actually, I'll bet our hit*** rate ***in copy editing is better than the New*** York Times. ***There is a conscientious staff here at CCINK. Last time I checked in their office, however, they were trying to trap egg yolks in the center holes of 45-RPM records. Wonder where they got that idea?***

-- Steve

(We thought he was talking about ox eggs! Ed.)

Dear Steve,

I have read your projects since you first started writing for BYTE. After I scanned issue #2 of INK, I could not wait to get to the computer to write this note. I had to subscribe and just could not wait. I wish you

and your staff the very best, and if issue #2 is any indication, I believe that the best is now and also yet to come. Keep up the good work.

C. Powell
Tampa, FL

Dear Steve,

I read with interest the discussion on Audio Digitizing in "ConnecTime" of Circuit Cellar INK [Vol. 1, No. 2]. It happens that our "Digital Voice Card" (DVC) may provide some of the answers to the discussion there. The Digital Voice Card is of low cost ($95.00) and it produces good-quality digital voice. It can be easily connected to audio equipment for recording and playback.

Houng I-Yuan
President
Computer Age Ltd.
Box 730
Nobleton, Ontario
Canada IOG 1N0

Thanks for letting us know about your product. While I believe much of the discussion dealt with high-precision recording, low-cost digitized speech has its niche. Coincidentally, there is a digitized audio board in the works as a future CCINK project.

-- Steve

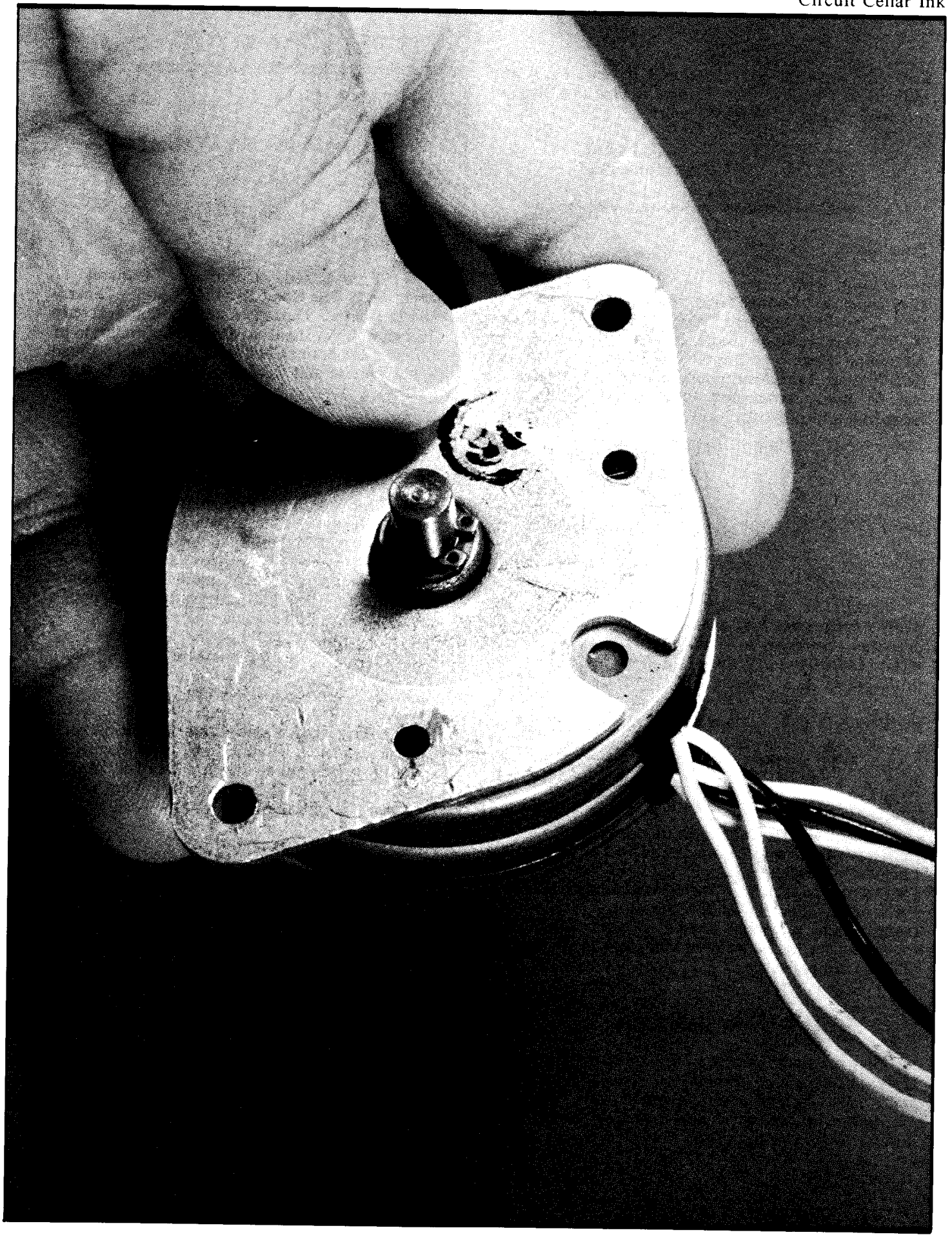
Dear Steve,

I am also very interested in receiving FAX on a shortwave radio and displaying it on an Amiga 1000. I am a research meteorologist studying hurricanes and I would love to set something up in my office to receive FAX maps. Later I would like to try receiving the satellite data. I am also interested in dumping RTTY transmissions (again, weather data) into the Amiga and drawing simple maps.

P. Dodge
Miami, FL

This is an area that is a bit alien to me but perhaps some budding author/inventor will be motivated enough to provide us all with an article on doing this. Regarding the Amiga in general, I would love to publish some articles which use it.

-- Steve



Circuit Cellar

Stepper Motor Scanning Sonar Sensor

A Look at the Logic and Control of Stepper Motors

by Steve Ciarcia & Ed Nisley

Ultrasonic sensing is not a new topic in Circuit Cellar. I've presented three projects (Nov. '78, Nov. '80, and Oct. '84) on sensing distances using ultrasonics. These past projects consisted mostly of hardware simply because I tend to do things that way when left to my own devices. Such projects are quite educational for already-computer-savy readers but sometimes leave people who think that computers and toasters are close relatives looking for the "rest." The "rest" is application software.

With Ed's software help I decided to redesign one of these former projects, incorporate present day technology, and provide some of the missing application software to broaden its appeal. The Circuit Cellar Stepper Motor Scanning Sonar Sensor (sonar scanner for short) is a computer peripheral which does a 360-degree scan of a room and creates a map of the objects in it.

The sonar scanner consists of an ultrasonic transducer mounted on the shaft of a stepper motor controlled by a standard parallel printer port. At each step the ultrasonic ranger records distances to all objects within a range of 1.2 feet to 35 feet and plots a real-time "radar" display on an IBM PC monochrome or CGA display. Software for controlling the stepper motor and driving the display is written in Turbo Pascal but easily converted for use on other machines. Complete source code can be downloaded from the Circuit Cellar BBS.

I'm taking a slightly different tact this time, however. Since I have explained the basics of finding distances using ultrasonic ranging many times, and that function in this project is provided by an off-the-shelf device, I will not dwell on that activity. Instead, I want to use this opportunity to describe more about how stepper motors work and logic used to drive them. With the addition of a little Nisley software magic added to the stepper motor, the result will be a sonar scanner!

Finding the Range

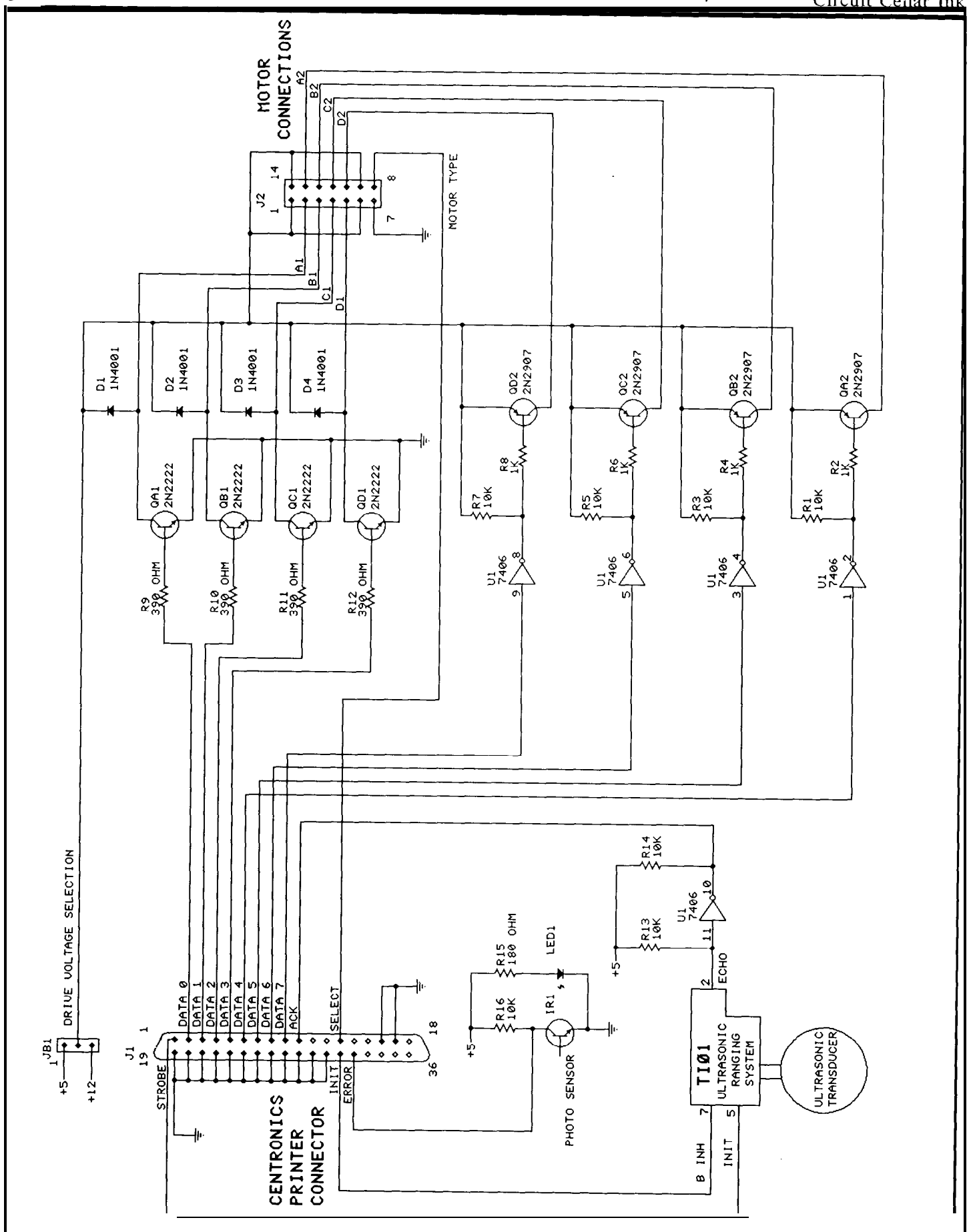
The basic principle of acoustic range finding is simple enough: make a noise and listen for the echoes. As with most projects, actually getting a circuit to work is a lot more difficult. Rather than building a circuit from discrete parts, I'll use the TI01 rangefinder board and ultrasonic sensor as a component and continue from there with the design of a complete scanning system. The TI01 was described in detail in "An Ultrasonic Ranging System," Ciarcia's Circuit Cellar, BYTE, October 1984 (reprinted in Ciarcia's Circuit Cellar Volume V).

Given the rate at which electronic components become obsolete you'd expect that the Texas Instruments ultrasonic rangefinder board I used in 1984 would be long gone by now. But it seems that nothing better has been offered to the market in all that time and the TI01 it is still the most cost-effective and readily available ultrasonic ranging unit.

Schematic 1 shows the complete schematic of the sonar scanner. The above photo shows the assembled prototype. It consists of a rather simple connection of the TI01 ultrasonic ranger board and a stepper motor to an IBM PC (or compatible) parallel printer port.

Most rangefinders usually only detect the distance to the nearest object. In an application intended to mimic a real radar we must scan an area and display a map of everything "seen" in each direction, including overlapping objects at many distances. Such a new requirement does not mean a complete redesign of previous hardware, however. While I might never have mentioned it before, the TI01 has the ability to register "multiple echoes" already; the "B INH" (which stands for "blanking inhibit") input allows the ECHO output to be reset after each pulse is detected.

The time between the INIT signal and each ECHO return is proportional to the round trip distance between the ultrasonic transducer and the objects causing the echoes. When each ECHO is detected, the B INH input is pulsed



Schematic 1 – Complete schematic of the sonar scanner.

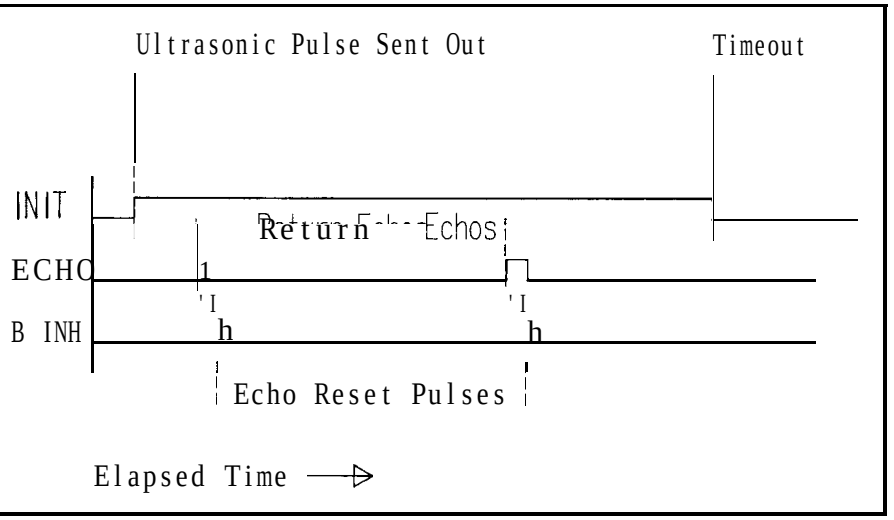


Figure 1 - Pulse time relations between sonar input and output.

Listing 1 - Loop for timing round-trip travel of the sound waves.

```
; Trigger the rangefinder

            in      al,dx                   ; get current bits
            and     al,NOT blanking         ; blanking off
            and     al,NOT trigger          ; trigger on!
            out     dx,al                   ; ... timing starts now
            dec     dx                      ; point to status again

; Run the timing loop until timeout or return array is full

looper      label   near
            in      al,dx                   ; get status bits
            test    al,echobit              ; check if all done
            jz      gotecho                 ;  ... 0 = echo

;--- no echo, check for timeout

            mov     ax,TODlsw               ; get current ticks
            cmp     ax,bx                   ; hit timeout value?
            jb      looper                  ; no, do again
            jmp     goback                  ; yes, bail out

;--- got an echo!
gotecho     label   near
            mov     dx,timeport             ; get Timer1 value
            in      al,dx                   ; . . . lsb
            xchg    al,ah
            punt                            ; delay a bit
            in      al,dx                   ;  ... lsb
            xchg    al,ah
            sub     ax,[bp].basetime        ; convert to abs time
            neg     ax                      ; . . . flip sign
            stosw                           ; and save into array

            mov     dx,[bp].rangeport;  blip  blanking
            inc     dx                      ; point to control
            in      al,dx                   ; set blanking bit
            or      al,blanking
            punt                            ; delay a bit
            out     dx,al
            and     al,NOT blanking         ; clear blanking bit
            punt                            ; delay a bit
            out     dx,al
            dec     dx                      ; point to status bits

            loop    looper                  ; count this echo
```

high and low to reset the ECHO output. Even though the reset is being handled in software it's fast enough not to miss any echoes from closely spaced objects. The program Ed wrote to measure the ranges can store up to ten echoes for each output pulse.

Figure 1 shows the timing relationships between the inputs and outputs. Remember that the speed of sound is about 1100 feet per second and that the time shown in the figure is for the round-trip distance. Because the maximum range of the TI01 is about 35 feet, the maximum round-trip time is about 64 ms (milliseconds). One inch of range (not round-trip) distance corresponds to about 150 us (microseconds).

Since Turbo Pascal doesn't offer a standard way to measure time intervals that short, Ed decided to use one of the 8253 hardware timers available on the IBM PC to time the pulses. Timer 0 in the 8253 is used for the time-of-day clock; it counts at the rate of two ticks in 838 ns (for an obscure reason it counts by twos rather than ones, so it goes through a full 16-bit count in 54.9 milliseconds). The PC BIOS updates a word at 0040:006C every 54.9 ms (18.2 times per second) as part of keeping the time-of-day clock running, so we simply note the starting value of that word and check the current value when receiving an echo. Four counts (220 ms) are allowed before timing out.

The sonar plot program uses an assembly language routine to handle the high-speed timing during each scan. Listing 1 shows the core of this code. You'll notice that it spends most of its time watching that BIOS clock location! The Turbo Pascal code calls the assembly code whenever it needs to measure ranges.

That explains how we measure distance but not how we aim the

scanner. As I mentioned earlier, the ultrasonic sensor is mounted atop the shaft of a stepper motor. To aim the scanner around a room, we drive the stepper motor in the appropriate direction in discrete increments.

An ordinary electric motor is designed to rotate smoothly when power is applied to it. This sort of motor is found in everything from washing machines to car windows, wherever a constant rotary motion is required. These motors can be driven by either AC (alternating current) or DC (direct current) power supplies depending on the application (and, of course, the motor's design!).

There are some uses that demand motion measured in discrete steps. A classic example is the motor in a diskette drive that moves the heads exactly one track per step. Another is your printer's paper advance, moving the paper in exact steps as the head scans the page.

A stepper motor features a very simple electrical interface that's easily adapted for computer control. In fact, it's so simple that steppers are often used in applications where you'd normally expect a "smooth" DC or AC motor. If the steps are small enough the resulting motion is smooth enough for many purposes. You wouldn't want to have one driving your stereo turntable, but moving the tone arm in or out to specific locations might be reasonable.

The motors I'll be describing are ones designed to connect directly to power supplies in the 5- to 15-volt range and draw less than an ampere of current. These are used in low-power applications like printers and simple robots. The larger industrial steppers draw many amps from a 24-volt supply and require a lot more knowledge to incorporate their use than I can convey in an introductory article.

Although specialized stepper motor driver ICs make designing a system quite simple, it's often difficult to buy them in small quantities. For this project I used simple transistor circuits so that you can build them from readily available components. The tradeoff is that all of the timing and control is handled by software.

Figure 2a shows the simplest possible stepper motor: a bar magnet between two electromagnets. If the electromagnets are energized so that their North and South poles are aligned, the bar magnet will seek a stable position.

To "step" the motor, simply reverse the direction of the current flow in the electromagnets as shown in Figure 2b. The bar magnet will move to a new position aligned 180° from its previous direction. Thus the motor has two 180° steps per revolution.

The problem is that there's no way to control which direction the bar magnet will turn. The motor can rotate either way when you reverse the current, or it might just flop back and forth instead of making a complete revolution.

Figure 3a shows the simplest possible stepper motor that's actually useful: the same bar magnet

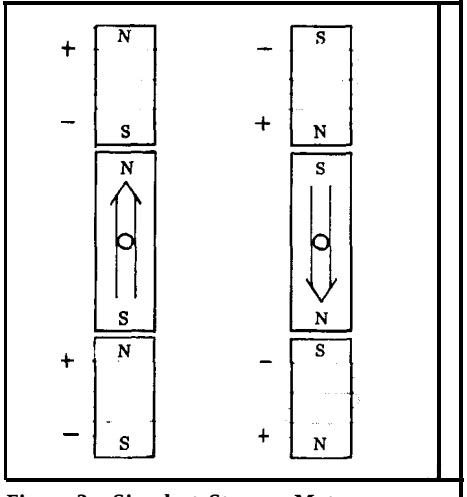


Figure 2 - Simplest Stepper Motor

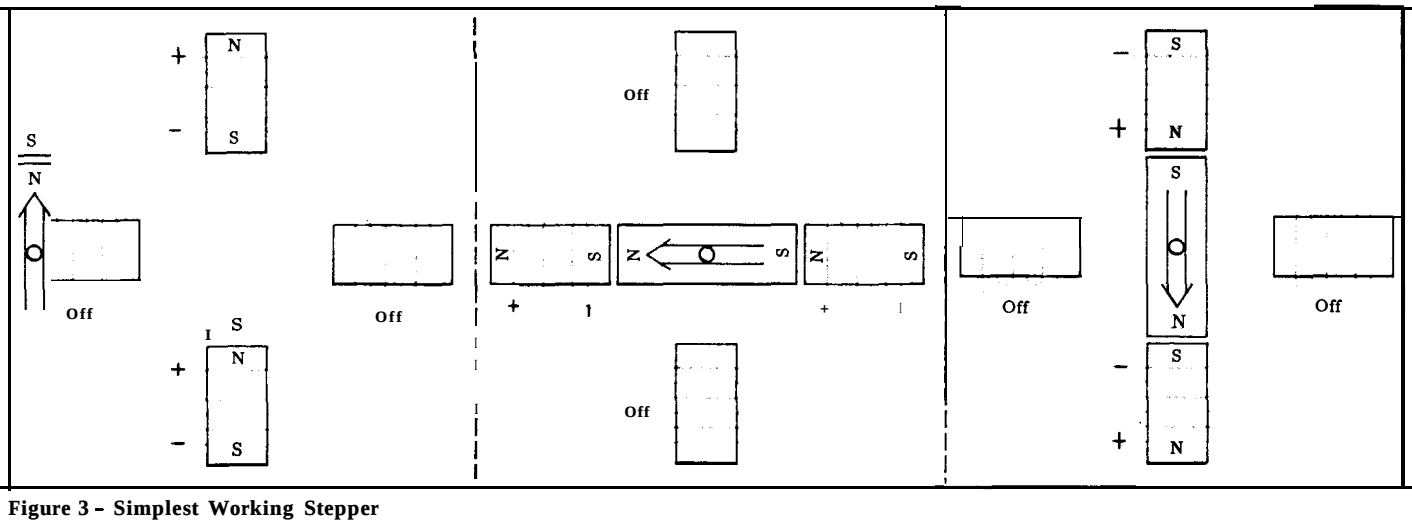


Figure 3 - Simplest Working Stepper

flanked by two pairs of electromagnets. The starting condition is the same, with the upper and lower electromagnets energized. By turning those two off, then energizing the left and right electromagnets as shown in Figure 3b, the bar magnet will rotate 90° CCW (counterclockwise).

Next, turn off the horizontal magnets and energize the vertical ones as in Figure 3c to make a second CCW step. The magnet is aligned just as it was in Figure 2b, but now we know exactly how it got there. This motor has four 90" steps per revolution.

Although there are four electromagnets shown in Figure 3, they are always used in pairs to provide both a North and South pole to align the bar magnet. To reduce the number of wires coming from the "motor" you can connect the two magnets in each pair together as shown in Figure 4. You must provide a way to reverse the current in each circuit as well as to turn it off completely at exactly the right times.

As you might expect, real stepper motors are a bit more complicated. The bar magnet is replaced by a cylindrical rotor and the electromagnets become complex multipole stator windings. But the principle remains the same: you drive a current through a stator coil and the rotor aligns with the resulting magnetic fields.

Figure 5 diagrams a (simple) motor with a ten-pole rotor and a twenty-pole stator, as seen looking along the rotor shaft. Remember that magnetic poles come in pairs of North and South poles, so both the rotor and stator must have an even number of poles. (For those of you new to motors, the stator is the outside frame, with the rotor being the central core attached to the shaft; just remember that the stator is stationary and the rotor rotates and you'll do all right.)

Despite the number of stator poles, there are only two stator windings: each creates ten of the twenty stator poles. Adjacent poles are connected to different windings. One set of ten poles is shown in gray boxes, the other in gold. The schematic diagram (Figure 5f) shows how this motor would be represented in a circuit.

Figure 5a shows the starting conditions: winding 1 (gray) is energized so that there is a South pole at the top of the stator. The rotor's North and South poles are directly opposite the stator's poles. Winding 2 (gold) is not connected, so half the poles are inactive.

Winding 1 is turned off and winding 2 energized so that new South poles are produced counterclockwise from the previous Souths. The rotor moves one step counterclockwise to align with the new set of poles as shown in Figure 5b.

Next, winding 2 is turned off and winding 1 turned on with the current running in the opposite direction. This produces a North pole at the top of the stator, where there was a South pole originally, but also produces South poles counterclockwise from the previous ones. The rotor makes another counterclockwise step to align with these poles, and Figure 5c shows the result.

By turning winding 1 off and winding 2 back on with the current in the opposite direction, the rotor steps once again to the position shown in Figure 5d.

Finally, winding 2 goes off and winding 1 on with current in the same direction it started with. This returns the stator poles to the same positions they started from, but the rotor is displaced four steps counterclockwise as shown in Figure 5e. Look carefully at the sequence of events in Figure 5 to verify that there's no magic involved.

The four steps can be summarized in Table 1. Each line of the table indicates one rotor step, so moving through the table one line at a time from top to bottom will step the rotor counterclockwise. When you get to the bottom of the table, just start over at the top again.

What happens if you reverse directions and read the table from bottom to top? It's simple -- the

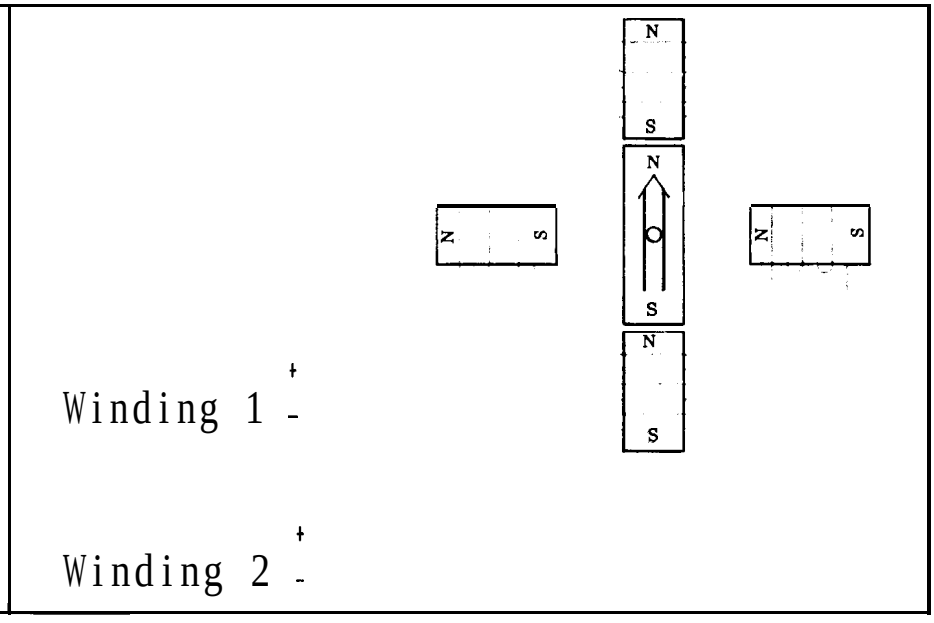


Figure 4 - Pole winding connections for a simple stepper motor.

rotor turns clockwise, one step at a time! When you get to the top of the table, start at the bottom.

What happens if both windings are energized at once? There will be twice as many poles on the stator as there are on the rotor, so the rotor poles will align toward the "average" of two adjacent stator poles. In effect, this doubles the number of rotor positions; naturally, it's called half-stepping rather than double-stepping to confuse the unwary.

Figure 6 shows what's needed to half-step the motor. It starts with the rotor and stator in the same state as Figure 5a. Instead of turning winding 1 off and winding 2 on as before, both are energized simultaneously to create a pair of South poles at the top of the stator. Figure 6b shows the rotor aligned midway between the stator poles. When winding 1 is then turned off, the rotor moves to the position shown in Figure 6c, which is identical with the full step shown in Figure 5b.

Table 2 summarizes the half-stepping process for a four-wire motor, while Tables 3 and 4 show the stepping in a half-stepping sequence for a six-wire motor. There are now eight half-steps required to bring the stator winding currents back to the starting condition. The rotor moves through the same angle, taking eight steps rather than four. As with Table 1, you can reverse the rotation direction by reading the table from bottom to top.

The Pitter Patter of Little Steps

There are, predictably, several types of small stepper motors. I've described one of the simplest: a two-winding, four-wire motor. It has a variant with only three wires; two are connected internally and brought out together. Another type has four windings and either five, six or eight wires depending on the internal connections. Figure 7 shows schematic diagrams for these motors.

The advantage of a two-winding motor is that it's somewhat simpler to build because there are only two wires inside. The disadvantage (as we'll see) is that the circuitry required to drive it is more complex.

A four-winding motor has simplified driver circuitry, but requires more complicated motor construction. However, because the motors are produced in large enough volume, it turns out that it's

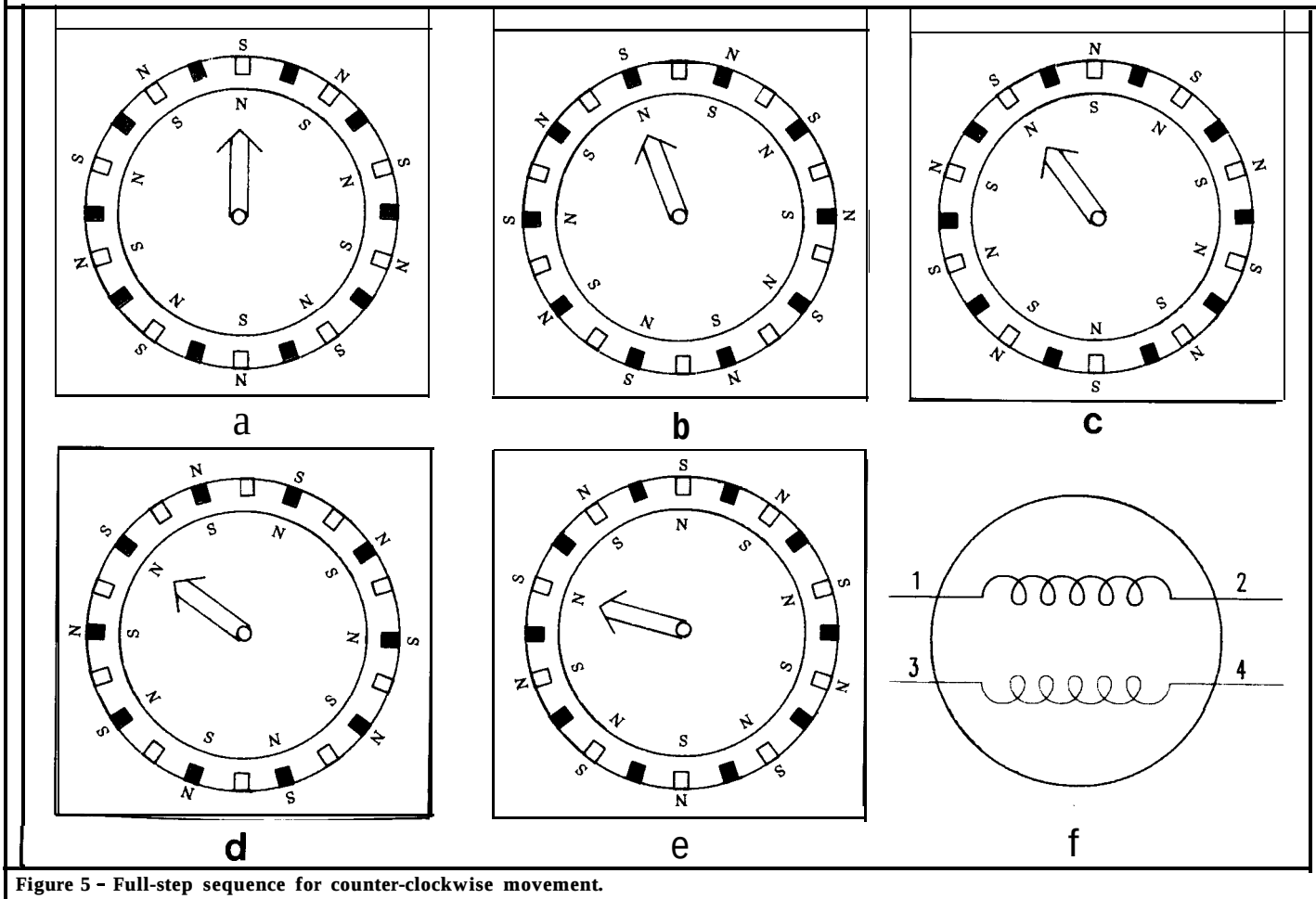


Figure 5 – Full-step sequence for counter-clockwise movement.

Table 1 - Four-wire full-step sequence

Full	--- Winding ---	
Step	Gray	Gold
1	+	off
2	off	+
3		off
4	off	

more economical overall to have complex motors with simplified electronics.

The motor drive circuitry for the Sonar Scanner will work directly with either four-, five-, or six-wire motors by changing only driver connections and the software. The most common motors have four or six wires, so those are the names I've used in the descriptions. You can convert an eight-wire motor to a six-wire one by connecting two of the leads together as shown in Figure 7d. A five-wire motor is just a six-wire one with the two common leads connected together.

You can buy stepper motors directly from vendors with catalogs listing the exact specifications, but for experimental applications it's simpler to get them from a surplus dealer and check them out to see if they do what you want. The prices are around $5-$10 in low quantities, and most of the motors you'll find in surplus outlets will have four windings.

To get started with a "strange" motor, you need some guidelines to figure out the supply voltage, the type of windings, and so forth. In many cases the supplier will be able to give you some hints, but it's always a good idea to check it out on your own.

If you look closely at a 6-wire stepper motor you'll see that it is divided into two halves, with three leads going to each part. Compare that with Figure 7d and it's easy to decide which leads are which.

The next step is to measure the electrical resistance between pairs of wires until you know which one are connected to which coils. Th resistances will also give you som idea of the supply voltage that th motor was designed to use. All o the windings will have resistance that are the same to within a fe percent, but don't expect an exac match.

For example, a four-lead moto has two independent windings, s you'll find two equal resistances. A six-lead motor has two pairs o windings, and the lead that's corn mon to a pair will have equa resistances to the other two wire! If you start by measuring the wind ings of a pair in series, the remain ing resistance will be half of tha and it'll be obvious which wire i which.

It's helpful to make a table tha shows the values for all wire pair: Table 5 gives the typical resistance measured for a 6-wire motor. Yo should measure the resistance bot

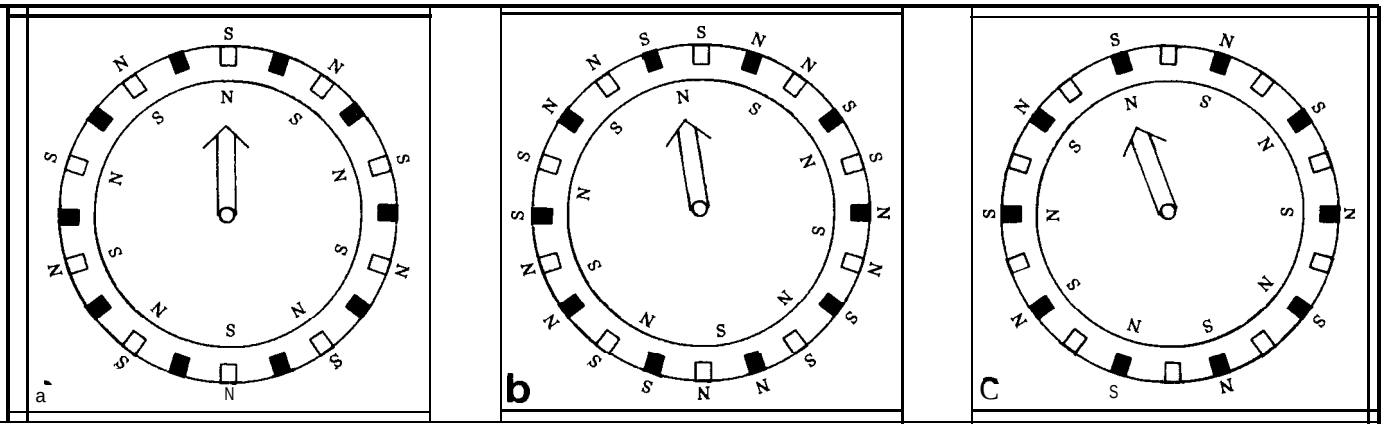


Figure 6 - Four-wire motor half-step sequence.

Table 2 - Four-wire half-step sequence

Half	--- Winding ---	
Step	Gray	Gold
1	+	off
2	+	+
3	off	+
4		+
5		off
6		
7	off	
8	+	

ways for each pair to make sure that the motor doesn't have diodes integrated into the windings -- rare, but possible.

Some motors will have a safety ground connected to the case. Usually this lead is green with a yellow stripe and is not electrically connected to any of the windings. It ensures that the case can't become hazardous if it's shorted to a supply voltage, even when the motor is removed from the chassis.

The next step is to identify the winding groups if they weren't obvious from the motor's construction. Table 5 shows two sets of three wires that don't connect to each other, so it's reasonable to assume that they are two separate groups.

Now you can identify the connections in each group. Each group in Table 5 has two resistance values: 65 and 130 ohms. The fact that the yellow wire has a 65-ohm resistance to the black and white wires, while the black-to-white resistance is 130 ohms tells you that the yellow wire is the common lead for the two windings. Use Figure 7 as a guide to the possible wiring connections, but expect the unexpected!

Finally, you can make a guess at the operating voltage. Typical motors will draw 100 to 400 mA (milliamperes) per winding. The motor in Table 5 has a winding resistance of 65 ohms, so the operating voltage should be around 13 volts (65 ohms times 200 mA). A more reasonable voltage would be 12 volts because that's more common. Try supplies of 5, 12, and 15 volts to see what currents they produce.

The power delivered to the load is proportional to the product of the supply voltage and total winding current, so a motor designed to run on a lower voltage will draw a higher current than a similar higher voltage motor. If you measure very low resistances, you can assume that the intended voltage is around 5 volts. (Note: Very powerful industrial steppers have resistances down around 1 ohm, so if you find a motor in that range you've got a different set of problems!)

A simple way to verify your guess is to connect half the windings to the supply voltage for a while and see if the motor overheats; if so, use a lower voltage. If not, try to turn the shaft by hand; if you can, use a higher voltage. By

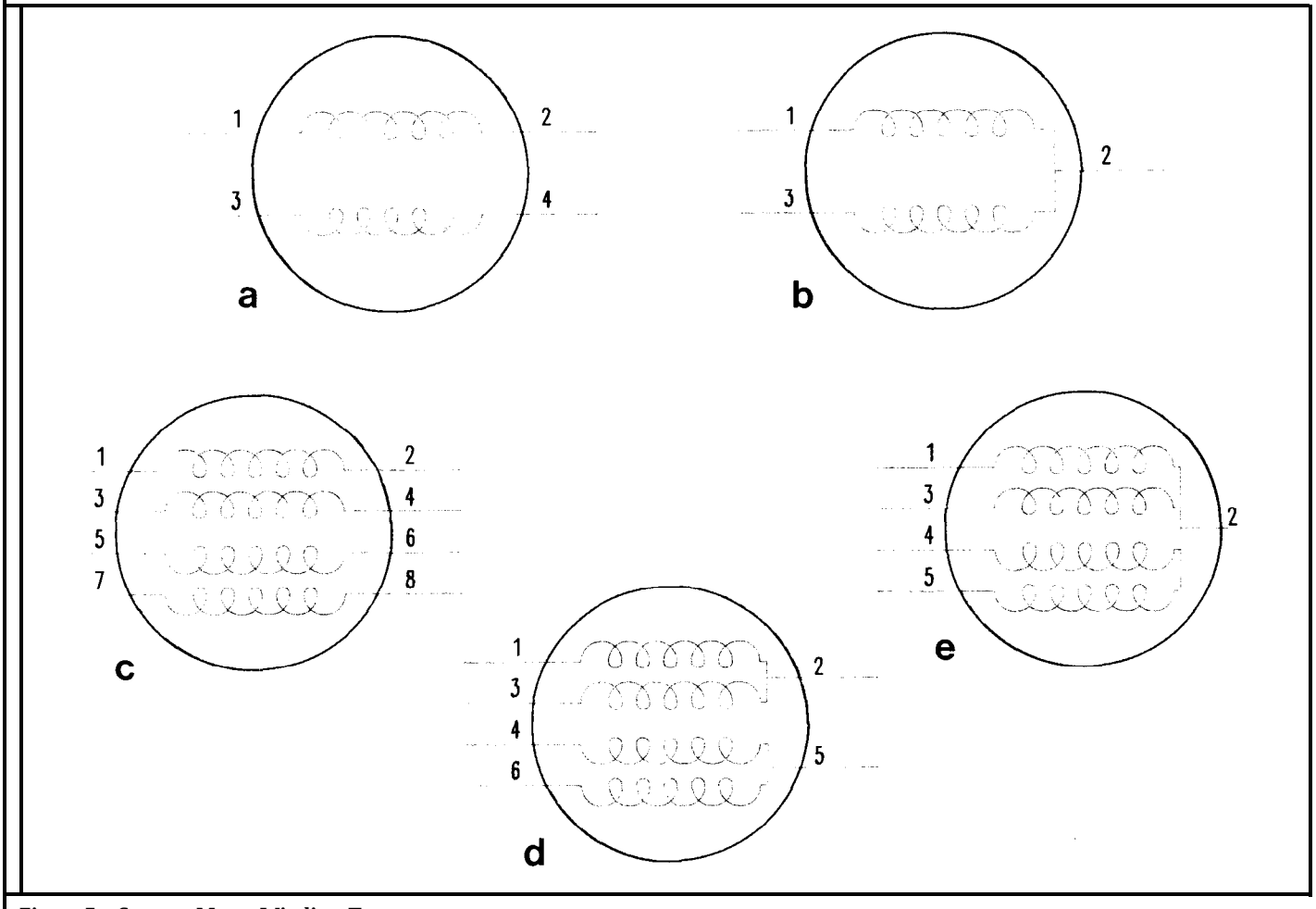


Figure 7 - Stepper Motor Winding Types

Table 3 - Six-wire full-step sequences

See Figure 7d for wire numbers. Wires 2 and 5 are connected to the motor supply voltage. Wires 1, 3, 4, and 6 areconnected to ground as shown.

Single winding active for reduced power consumption

Full Step	--- Wire numbers --- 1	3	4	6
1	gnd	off	off	off
2	off	off	gnd	off
3	off	gnd	off	off
4	off	off	off	gnd

Two windings active for increased torque

Full Step	--- Wire numbers --- 1	3	4	6
1	gnd	off	gnd	off
2	off	gnd	gnd	off
3	off	gnd	off	gnd
4	gnd	off	off	gnd

Table 4 - Six-wire half-step sequence

Half Step	--- Wire numbers --- 1	3	4	6
1	gnd	off	off	off
2	gnd	off	gnd	off
3	off	off	gnd	off
4	off	gnd	gnd	off
5	off	gnd	off	off
6	off	gnd	off	gnd
7	off	off	off	gnd
8	gnd	off	off	gnd

the way, I define "overheating" by the rule ofthumbthatifit'stoo hot to put your thumb on it, it's too hot!

Measure the steady-state current through the windings and pick transistors that can handle two or three times that current. For example, a 2N2907 transistor can carry up to 600 mA and the 2N2222 up to 800 mA. Of course, when using any power transistors in high-current applications, make sure that the transistor base connections don't draw excessive current from the TTL logic chips controlling them. 2N2907s and 2N2222s are relatively high-gain devices requiring low base currents.

There are also other considerations that enter into transistor selections but for small motors with low currents you can ignore most of them. One detail you shouldn't ignore, however, is protecting the transistors from transient voltages. That's what diodes D1 through D4 are doing in the Sonar Scanner.

Because the motor windings are inductors, the current flowing through them cannot be turned off instantly. If the switch transistor is not protected by a diode the current can force a high voltage at the collector, perhaps causing a failure. The diode conducts whenever the collector voltage exceeds the supply voltage, dumping the current into the supply where it can recirculate back into the winding without doing any damage.

Stepping in Sequence

Now that you know which wires go to which windings and roughly what the right voltage may be, it's time to make the motor go around.

Schematic 2a shows the circuitry required for a two-winding motor. Because the two pairs of transistors for each winding look somewhat like the letter "H" the configuration is sometimes called an "H bridge." In normal use only one transistor on each side will be turned on: QA1 and QD2 to pass current left-to-right, then QA2 and QD1 for the other direction.

Turning on both QA 1 and QA2 (or any of the three similar pairs) will short the power supply to ground through the transistors. You should make certain that your software cannot make this happen!

Schematic 2b shows the driver circuitry for a four-winding motor. The common leads are connected to the supply voltage and the other leads of each pair are switched to ground using QA1 through QD1. Unlike the two-winding motor, turning on two transistors at once will have no ill effect other than doubling the current drawn from the power supply.

Because drive current flows in both directions through a two-winding motor, this type is also known as a "bipolar" stepper. Because a complete circuit requires twice as many transistors, resistors,

and so forth, it's more expensive than that for a "unipolar" four winding motor.

The Sonar Scanner hardware includes a pair of H bridges that can drive a bipolar motor, but the four lower transistors can also drive unipolar motors. The software uses an input bit from the motor connection socket to determine which type of motor is connected and select the appropriate values to send to the output port. Assuming that you've not told any lies to the software, either motor will work correctly.

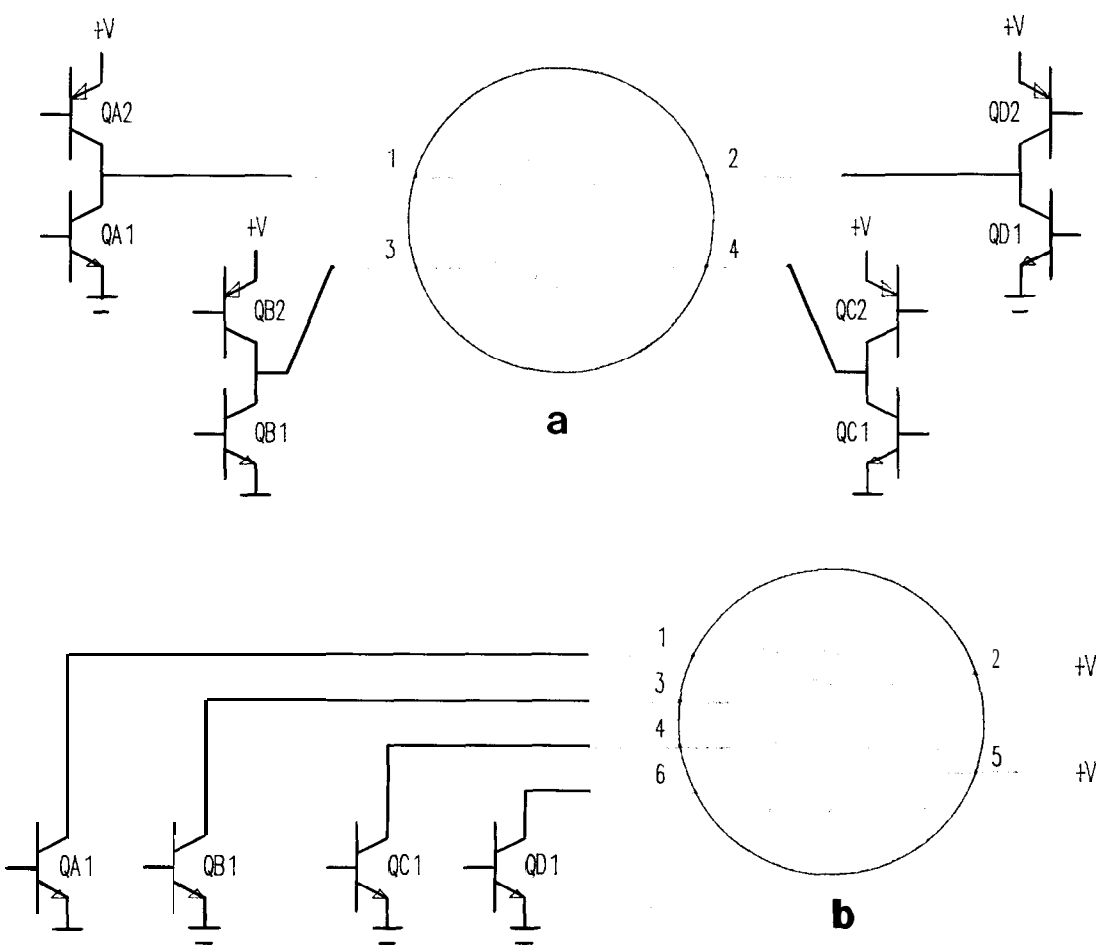
Contrary to what you might think, the consequences of interchanging two or more motor leads aren't tragic. The motor will simply not rotate uniformly: it will jerk

Table 5 - Six-wire motor resistance table

Column and row headings identify wires:
B1 = black, winding 1
Y1 = yellow, winding 1
W1 = white, winding 1
B2 = black, winding 2
Y2 = yellow, winding 2
W2 = white, winding 2

resistance values are in ohms

From	To:	B1	Y1	W1	B2	Y2	W2
B1			65	130	nc	nc	nc
Y1		65	-	65	nc	nc	nc
W1		130	65	-	nc	nc	nc
B2		nc	nc	nc	-	65	130
Y2		nc	nc	nc	65	-	65
w2		nc	nc	nc	130	65	-



Schematic 2 - Motor connections for (a) four-wire and (b) six-wire stepper motors.

Listing 2 - Step Control Code

```
{-------------------------------------------------------}
{ Motor definitions                                     }

CONST

 motors    : mtype = (
            (phaseseq  :  ($18,$3C,$24,$A5,$81,$C3,$42,$5A);
             phaselim : 8;
             holding  : TRUE:
             title    : 'Bipolar 4 wire half step'),
            (phaseseq : ($18,$24,$81,$42,0,0,0,0);
             phaselim  :4;
             holding  : TRUE:
             title     : 'Bipolar 4 wire full step'),
            (phaseseq  :  ($08,$02,$04,$01,0,0,0,0);
             phaselim : 4;
             holding  : TRUE:
             title    : 'Unipolar 6 wire full step'),
            (phaseseq : ($08,$0a,$02,$06,$04,$05,$01,$09);
             phaselim : 8;
             holding  : TRUE:
             title    : 'Unipolar 6 wire half step'),
            (phaseseq : ($0a,$06,$05,$09,0,0,0,0);
             phaselim : 4;
             holding  : TRUE:
             title    : 'Unipolar 6 wire full step dual')
              );

{-------------------------------------------------------}
{ Core of motor step routine                            }
{ "mID" selects the appropriate motor definition        }
{ "phase" selects a value from the "phaseseq" array     }

   CASE dir OF
    cw  : BEGIN                  ( turn clockwise       }
           IF phase < motors[mID].phaselim
            THEN phase := phase + 1
            ELSE phase := 1;
           Port[mport] :=  motors[mID].phaseseq[phase];
          END;
    ccw : BEGIN                  { turn counterclockwise }
           IF phase > 1
            THEN phase := phase - 1
            ELSE phase :=  motors[mID].phaselim;
           Port[mport]  :=  motors[mID].phaseseq[phase];
          END:
    ELSE ;                       { nothing if stationary }
   END:
```

first one way, then the other, as the rotor tries to align itself with the stator's magnetic fields. You simply interchange pairs of wires until the motor rotates correctly!

To simplify this process Ed wrote a control program called SEQUENCE.PAS that produces the correct sequences for several types of motors. When experimenting with an unknown motor, simply hitch it up to the Sonar Scanner's driver transistors (after figuring out how many windings it has, of course) and run SEQUENCE. Eventually the program will hit on the right combination of leads to make it run.

It's a good idea to keep a written record of the lead combinations you've tried. Because most motors will have several leads of the same color, you might want to attach a numbered label to each lead so you can keep things straight. Always interchange leads within a winding, not between windings.

Listing 2 shows the motor definition constants used by SEQUENCE. The phase sequence array (phaseseq) determines which transistors are turned on for each motor step. The phase limit value (phaselim) indicates the number of valid entries in phaseseq. The boolean variable "holding" controls decides if the transistors are left on after the step is completed, or turned off to reduce power dissipation. Finally, the title string is used to describe the particular motor so you know what the output is supposed to be doing.

For example, the second type is the familiar 4 wire bipolar motor described in detail above. If you match the values in phaseseq ($18, $24, $81, $42) with Schematic 1 you'll find that the outputs are being turned on in exactly the order needed to make it rotate clockwise.

Incidentally, SEQUENCE includes the ability to vary the step rate. On an 8-MHz AT it can produce up to 1000 steps per second, well beyond the ability of some small motors. A "ramp" function was added to slowly increase the speed to make sure the motors didn't get confused by sudden changes.

The Sonar Scanner program's motor definitions include additional information that describes the basic motor step size in degrees, as well as the number of motor steps in a single sonar scan. The program can format the screen display to match the capabilities of the motor without further assistance.

Typical stepper motors have step sizes of 18°, 7.5°, and 15°. The stepper motor I used on the sonar scanner is an Airpax #82227 which has 18° per step. The Sonar Scanner software drives the 18° motor with two half-steps for each scan to smooth the scan motion. If you attach a 7.50 motor, it will respond with two single steps per scan.

Photo 2 shows the Sonar Scanner program in action on a monochrome monitor (it displays in color on a CGA). The 1 8° step size gives 20 scans per revolution. The tick marks indicate 5-foot intervals.

In order to position the screen displays correctly the Scanner software needs to know which way the ultrasonic transducer is pointing. Because stepper motors have no position feedback, we added an optical sensor to tell the software when the transducer was located at 180°. Photo 3 shows the sensor as well as the metal tab attached to the ultrasonic transducer. The output of the sensor is returned to the PC through an input bit in the printer port.

The Scanner software always initializes the transducer's position by rotating it counterclockwise until the sensor reports that the tab has been seen. To make sure that the alignment is as close as possible, the software uses the smallest step size that the motor is capable of executing. After the transducer is aligned, the software may use a larger step size to give about 20 step/scans per revolution.

While I used an optoelectric sensor, you could also use a reed switch with a magnet attached to the rotation ultrasonic sensor. The basic idea is just to have one position during the rotation generate a signal.

There are no mechanical stops to prevent the software from twisting the wires attached to the transducer into a tight knot. This is an example of "fail-dangerous" design: if the software runs amok, the hardware can be damaged. In a serious application you would want to add hardware or software to prohibit such occurrences.

In addition, since the Sonar Scanner uses a standard printer port, it's possible that you might inadvertently print a listing directly to the scanner. Bizarre things can occur; if you're using a bipolar motor, some of the characters will surely turn on both of the transistors on one side of the H-bridge circuits. If you don't catch it soon enough it may burn out the transistors.

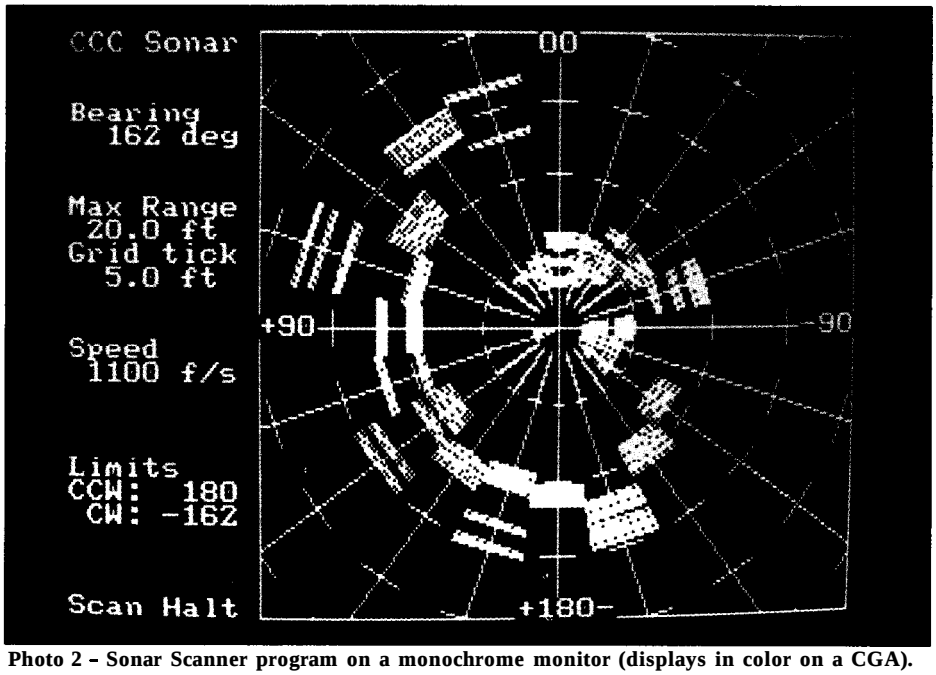


Photo 2 – Sonar Scanner program on a monochrome monitor (displays in color on a CGA).

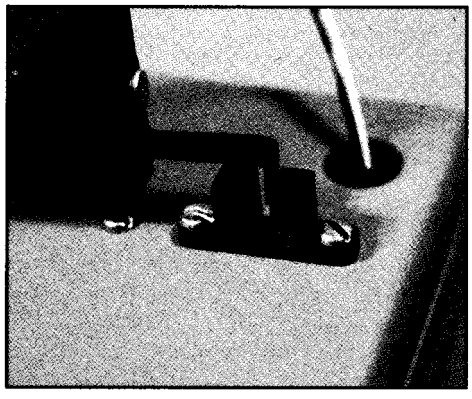


Photo3 – Sensor and metal tab attached to the ultrasonic transducer.

Unfortunately, the PC BIOS contributes to this problem and there wasn't much that Ed could do to fix it. During the power-on initializations and tests, true-blue IBM PCs (and, presumably, most clones) check the printer ports by writing $AA to the data ports and reading it back. This value turns on four of the transistors in exactly the wrong way for bipolar motors.

It turns out, however, that the Micromint UPS1 1 power supply that I was using goes into current limiting when it's overloaded. No damage was done because the transistors can handle the current until the power supply shuts down. This problem could be fixed by some additional hardware, but I don't think it was worth complicating this relatively simple design. The solution is to use some intelligence and to not plug in the Sonar Scanner box until the program is started; the program's first function is to write $00 to the output port to shut off all the transistors. Of course, if you are building this sonar scanner as a "real" project, you might want to take this situation into account and perhaps add hardware current limiting on the drivers.

Listing 3 presents the function key assignments for the Scanner program. You can change the maximum range displayed on the screen, restrict the scan to a small arc, and fine-tune the speed of sound value. Internally, all of the echo times are stored in an array that could be saved on disk or analyzed by another program to "see" what's in the room.

For example, you could add software to compare two successive scans and decide if something in the room has moved. The program

Listing 3 - Sonar Scanner Program Function Keys

		Default
F1	-	
F2	beeps and boops on/off	off
F3	set speed of sound	1100 ft/sec
F4	erase traces, refresh display	
F5	decrease max range (40/20/8 feet)	20 ft
F6	increase max range (8/20/40 feet)	
F7	set CCW scan limit	180 degrees
F8	set CW scan limit	-180+1 scan
F9	-	
F10	start/stop scanning	

To exit from the program, tap the ESC key.

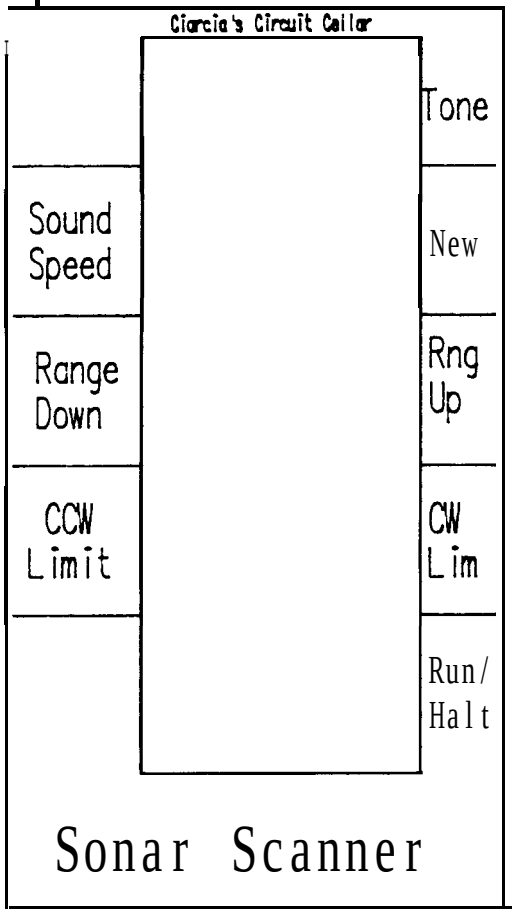
Sonar Scanner connector wiring

The computer connection is a standard Centronics parallel port as used by the IBM PC. The cable has different pin numbers on each end for some signals, due to the "standard" IBM PC cable:

PC end	Scanner end	Signal function
1	1	+Init to ultrasonic board
2	2	+A1 (NPN transistors to ground)
3	3	+B1
4	4	+C1
5	5	+D1
6	6	+A2 (PNP transistors to +V)
7	7	+B2
8	8	+C2
9	9	+D2
10	10	-Echo from ultrasonic board (inverted)
13	13	+motor select
15	32	+limit switch (+ on detect)
16	31	+blanking inhibit to ultrasonic board

Pins 16,17, and 19-30 at the sonar connector are ground.
Pins 18-25 at the PC connector are ground.
Not all printer cables connect all the ground pins together.

IBM PC Funtion-Key Temolate



could then change the maximum range and restrict the scans to a few degrees on either side of the "target" to get more data on it at a faster rate.

A mobile robot using the scanner would have to correct the alignment of the stored scans to compensate for its own movement. Imagine that the robot turned 90" between two scans: unless the data was adjusted, it would appear that everything had moved around the robot.

Of course, all this is a simple matter of software that I prefer to leave for others . . .

Ctrl

I N K

Dear INK,

I have a Tandy 1000 SX and use several IBM compatibles. My problem is with sound. I have Let's C, Utah COBOL, Microsoft Macro Assembler and COBOL, and BASIC. I primarily use the COBOLs. I can find everything you would want to know about everything except sound. With the exception of the BASIC which has sound commands, I can't seem to find anything on the subject. I can find lots of books and articles on color, but nothing about sound.

I know there has to be something on the subject as almost every program I get anymore has both color and sound incorporated.

I would appreciate any information you could give on where to get information on how to get sound into my COBOL programs. If I need to access the assembler to do this, that would be fine. I bought the assembler with sound in mind, but as with all of Microsoft's manuals, it was no help unless you already knew the answer you were looking for.

D. McLean-Cheyenne, WY

Dear Daniel,

IBM PCs have a primitive sound generator that can handle one tone: software sets the frequency, turns it on, then turns it off. There's no volume control. You can do some remarkable tricks, but it takes a lot of programming to make it happen.

One book we've used for a while is Bluebook of Assembly Language Routines for the IBM PC and XT ***by Christopher Morgan (Waite Group, ISBN 0-452-25497-3). It's fairly old, but has a wealth of routines and tips for all those gritty little assembly routines. There's a 30-page dissertation on sound, as well as another 30 on basic graphics and colors. The remaining 200 pages cover math, strings, I/O, and so forth with a lot of sample programs and examples.***

-- INK

Dear INK,

I am faced with a problem that may be common to many of your readers. I am using a PC at home in more and more situations. My application is primarily word processing, however I expect to move into desktop publishing applications. I hope to be able to eventually capture screen images from television and print them to the page.

My problem is with selecting a monitor. I would like to find a terminal which not only adapts to EGA, CGA, and the other multitude of standards which seem to abound, but also to find one which allows the use of images I have saved to video disc and video tape. Rather than purchasing a monitor which is inherently limited, is it possible to select one which has sufficient resolution for word processing and desktop publishing in both color and black and white, but that also would double as a television/VCR screen both for computer applications and in other situations?

B. Millar-Hanna, Alberta, Canada

Dear Brian.

A monitor that is compatible with CGA, EGA, VGA, PGA, and several other higher resolution display cards is not hard to find. Any of the several multifrequency monitors like the NEC MultiSync, the Sony Multiscan, or the Magnavox Professional will accept analog input and do an excellent job of displaying color pictures at normal TV scan frequencies. The problem is that most of these monitors require separate red, green, and blue signals as input, while most home video equipment send out only a composite video signal. There are, however, at least three monitors available that accept composite video as well as digital and analog RGB signals. These are the Mitsubishi Diamondscan AUM1371A, the Thompson Ultrascan 4375M, and the Taxan Multivision 770 Plus. These monitors are all capable of displaying graphics with resolution of at least 640x480 pixels. The Thompson and Mitsubishi are good for 800x560. These

monitors might have some advantage over a good-quality television for displaying VCR and TV pictures, but remember that, to a very large extent, the picture quality is limited by the bandwidth restrictions on the standard NTSC composite signal. The 30-MHz video bandwidth of these monitors is mostly wasted on a composite signal with a bandwidth of about 3 MHz on a good day.

-- INK

Dear INK,

I am an amateur electronics/robotics enthusiast in great need of help. My problem is that my projects seem to be limited by the wall of nonautomation (no computer brain to monitor and control). I have gone to countless libraries in search of a book that can help me interface my projects to an 8080/8085-based microcomputer, with no luck. This book has to teach me how to build a microcomputer based on the 8080/8085 CPU, program, and interface it with the real world. The computer I need to know how to build needs no special programming language, except assembly language. I would like to create my own system monitor.

My question to you is, do you know of any books in print that can help me achieve this task, and could you send me a list?

J.Stratman-Westland, MI

Dear Joe,

Several of the books in the Blacksburg Continuing Education Series, published by Howard Sams, cover 8085 hardware and software. 8085A Cookbook ***by Titus, Larsen, and Titus (ISBN O-672-21697-3) is a good place to begin, followed by the two books on 8085 so jtware design. The cookbook covers design o fan 8085 computer in all important particulars, and may be exactly what you need.***

These books may be available in some libraries, but the most certain way to obtain them is through a book store or electronics supplier that carries books. They can also be obtained directly from the publisher, Howard W. Sams (4300 W 62nd Street, Indianapolis, IN 462681. Ordering in formation and current prices can be obtained from the publisher.

-- INK

Dear INK,

I am currently trying to design a proportional controller based on Signetics TDA-1023 chip. This controller is to accept an input signal of O-100 Hz at 100 ms P.W. from a flow sensor to control proportionally an AC motor at 120 VAC 6.6 amps. I prefer to stay away from CPU-based systems which is why I incorporated a TDA- 1023 TRIAC controller. Any help you can offer in my project would be appreciated.

Keith J. Donadio-Frostproof, FL

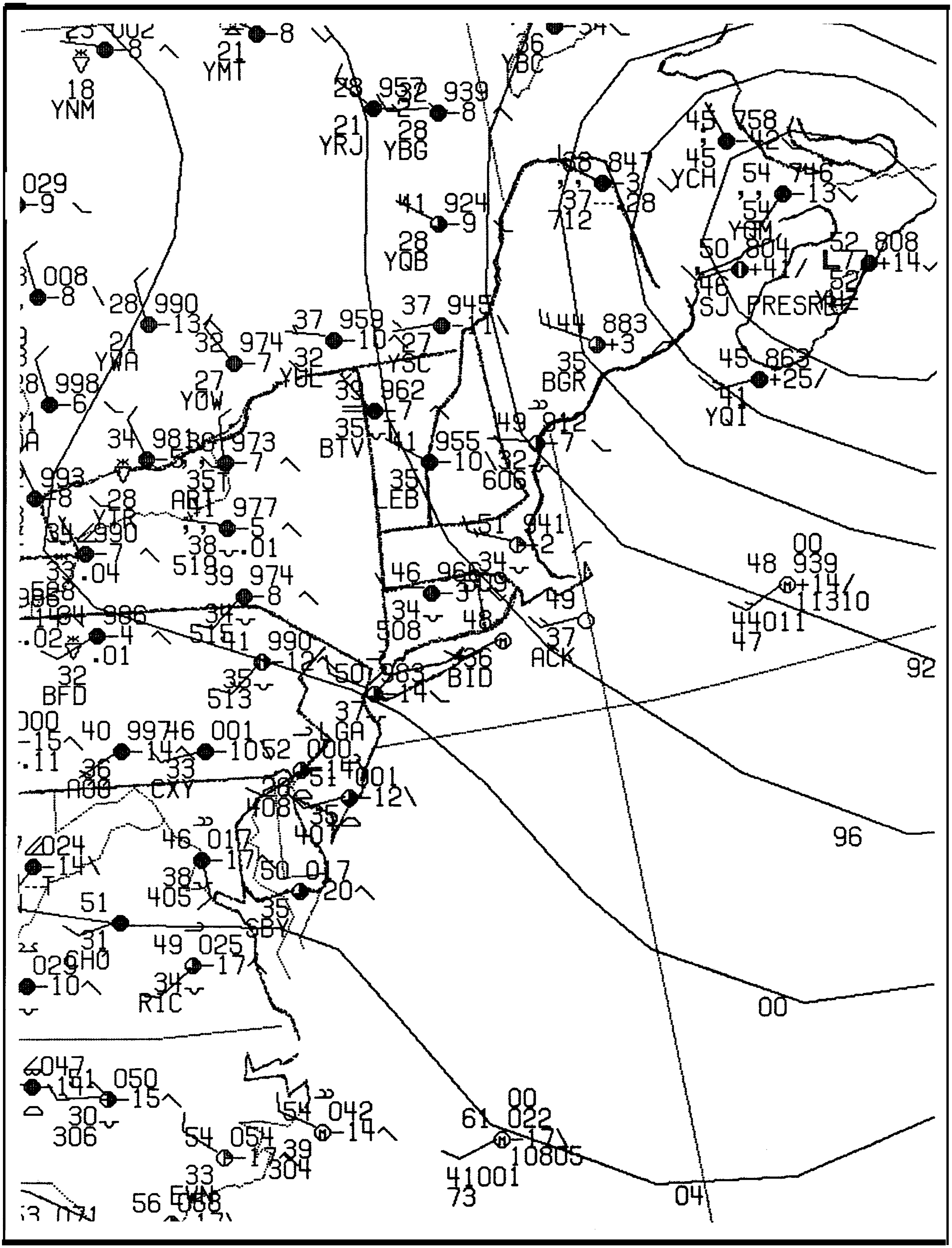
Dear Keith,

Microprocessor-based control systems possess great versatility, but in simpler systems, are neither necessary nor cost-effective. Your system seems simple enough that a dumb controller is more than adequate. While your pupose is sound, I'm not sure that the TDA-1023 is the most appropriate integrated circuit to use in your design. The TDA-1023 contains an internal comparator and pulse-generating circuitry. It is normally con figured using external variable resistances such as thermistors to determine the ON-OFF state. The October 1987 issue of Modern Electronics ***features construction of a darkroom temperature controller using the TDA- 1023 with a thermistor as the temperature-sensitive element. I know of no practical means of triggering it using pulses from an outside source. Applications data from Signetics illustrate possible modes of operation for the circuit.***

Since you already have a source of pulses, a simple TRIAC triggering circuit such as an MOC301 I (zero-crossing detector built-in) would seem more appropriate. You might also find buffering necessary using a 7406 or 7407 package, and/or perhaps a one-shot (e.g. 74LS123) to get precise control over the triggering process. With a pulse width of 100 us, considerable stretching is likely to be required for reliable triggering if the zero-crossing trigger circuit is used. Sample circuits for the TRIAC trigger can be found in applications data available from manufacturers such as Motorola.

--INK

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, Circuit Cellar INK, Box 772, Vernon, CT 06066. All letters and photos become the property of CCINK and cannot be returned.



YNM
YMT
YBC
YRJ
YBG
YCH
YQM
YQB
YSJ
BGR
YWA
YOW
YUL
YSC
BTV
LEB
ART
YTR
ACK
BID
BFD
LGA
CXY
SBY
RIC
YQI

The Home Satellite Weather Center

Part 4

Dial-Up Databases and a 68000 Peripheral Processor

by Mark Voorhees

As we all tire of summer, with the associated lawn mowing, allergies, and assorted summer ills, hopefully some of you have had time to download the first group of IBM PC software modules for our Home Weather Center system. If you've also had time to subscribe to one of the weather databases (and experiment with its use), you probably are already reaping some benefits from your efforts in increased knowledge of storm activity in your area, or just knowing when it's going to be nice at the beach.

This time around, we are going to cover several topics ranging from more advanced use of the phone access software modules to the first descriptions of the peripheral processor system.

Communication with Dial-up Databases

In the last issue, we discussed the relative merits of some commercially available weather database services, focusing on the products they provide to help you gain further knowledge about the weather phenomena occurring around you. At that time, we also covered the first PC software modules (these modules are available on the Circuit Cellar BBS or by mail from me).

The modules included in the first batch were the main configuration and menu program (WEATHER.EXE), the hardcopy module (PRINT.OVL), and the four weather service communications modules (CAD.OVL, CSS.OVL, CNOAA.OVL, and CWB.OVL).

Each of the communications modules contains the ability to use what I've called an "auto-access" file. The program always performs the sign-on and sign-off functions automatically (even for a manual session); the "auto-access" mode allows you to create a script-type file for often-used commands. For instance, if you normally want to get radar data for the U.S., the computer forecast for the next 24 hours for your region, and your state's current temperatures, you could create a file to handle those requests automatically. At the end of the file, the program signs off (or, if you have selected an "auto/manual" session, returns control to the keyboard).

Using the auto-access files can be tricky unless you are familiar with your service's command syntax. Whether you are a novice user or an "old pro," it is always best to manually perform the commands to be sure that you get what you want before committing them to an auto-access file. Certain service-generated errors are almost impossible to trap effectively, and if you find yourself "hung" within the auto-access modes due to an error, your only solution may be to reboot the machine.

Each communications module contains a complete auto-access file editor, so creating, editing, or deleting a file is easy. The editor for a given service module will only access, delete, or edit files for that service, thus minimizing the potential 'for corrupting files for another service. Each module can address up to 255 auto-access files (each file up to 100 lines long), and a hard disk is highly recommended!

Using short examples, I'll show you how an auto-access file might look for each service:

```
Accu-Data:
    RADU * 1
    MOSP @AZ
```

The first line calls for unaltered radar data (RADU) for all U.S. reporting stations (*) for the last hour (1). The second line requests plain-language computer forecast data (MOSP) for Arizona (@AZ). This command mode is called the "Advanced User Option" by AccuWeather, and their manual describes the possible commands fully. Although you can use the menu command mode with auto-access files, it is generally easier for this and other services to use direct-access-type commands.

```
CompuServe:
    SD CKL, LZK, SAC, LIC
    PBI, TBW, MMO
    <2 spaces>
```

The first line is, again, specifying radar report (SD) data, this time from four specific stations: CKL (Centerville, AL), LZK (Little Rock, AK), SAC (Sacramento, CA), and LIC (Limon, CO). The

second line continues the station requests for SD information for PBI (West Palm Beach, FL), TBW (Tampa, FL), and MMO (Marseilles, IL). The third line, consisting of two spaces, causes CompuServe to return to the command mode and wait for another data-type request or the OFF command.

In the case of CompuServe, the sign-on sequence of our communications program navigates to the AWX- 1 page before activating the auto-access file (in auto and auto/manual modes) or releasing command to the keyboard (in manual mode). Thus, the auto-access file need only address the actual weather commands. The first line must specify the data type (in this case, SD), followed by the reporting stations required. The following lines may specify the station names within the requested data type, and may number as many as necessary. When you desire to change data types or end the auto-access file, you must execute the line containing two spaces to tell CompuServe to return to command mode within AWX- 1.

Weatherbank:

```
NWS 2,PHX
RADAR/D
```

The first line of our file requests the National Weather Service forecast for Phoenix, AZ (code PHX); the second line requests the raw radar data file for the U.S. radar reporting stations.

The Weatherbank communications program has a slightly different operational feature in manual mode. While the auto-access function handles the sign-on routine just as it does with the other modules, the manual mode requires that you press a key at the command prompt to complete the sign-on process. Weatherbank allows the use of help files without charge before the sign-on process, so, you can browse these help screens before signing on. Note that the capture file for the session (explained later) will capture the screens you look at prior to sign-on in this module only.

NOAA:

```
1
7
```

Since the NOAA system only operates in a menu mode, you must enter the selection number for the desired report. In the first line, we are requesting the Climate Rankings for selected regions of the U.S.; the second line requests a listing of the "help" file. It is not necessary to include the "0" selection for sign-off in your file since it is taken care of automatically by the module.

I want to take a moment at this point to advise you of some problems I've noted recently while using the NOAA system. I've had several instances of poor line connections or apparent modem failures which cause either their system or my PC to lock up. In contacting NOAA, I've learned that an upgrade of the system (apparently both hardware and software) is "in the works," and problems can be expected until that is completed. I was also advised that NOAA plans to start charging for connect time for this service, although rates and transition dates have not been set. If and when changes are made to this service, 1'11 advise you of them in one of these articles. If changes are required in the NOAA communications module, a new version will be released at that time also.

The above modules have many functions in common and all are "menu driven,*' requiring little, if any, explanation. Note that a primary feature of all of the comm modules is the creation of a capture file for each session, consisting of all the data following the sign-on function (except for Weatherbank, as noted above) and the completion of the sign-off routine. The file is named using the following format:

MMDDHHNN.SS

Where:

MM is the month (leading zero ignored)
DD is the day (leading zero ignored)
HH is the hour in 24-hour format (leading zero ignored)
NN is the minute (leading zero not ignored)
SS is the service:
AD = Accu-Data
CS = CompuServe
WB = Weatherbank
NO = NOAA

All time and date references are to the time the session is started. It's obvious that you must make sure your PC's clock and calendar are set (or use a clock/calendar accessory to do this for you) to make effective use of this built-in session reference. This filename format makes it very difficult to accidentally overwrite a session file.

The session files will be useful to other modules in the future, but, for now, they allow you to get a hardcopy printout of the session using the hardcopy option from WEATHER's main menu. The printer function supports any DOS-driven printer; Epson-compatible command codes are used for any internal feature selection (none of which are used for the standard hardcopy text output).

Again, the hardcopy module is menu driven and straightforward in operation. You have the ability to select the file from a list of all session files for that service on the disk. You can then print or delete the selected file.

The printout itself will show a header listing the service, the filename printed, and the date of the printout. Continuous-form-format printing is used, with a form-feed issued at the end of the file.

As noted previously, these modules will be subject to revision to increase features or repair flaws. The executable programs will always be available from the Circuit Cellar BBS.

Should you experience any problem with a module, please report it to me either by Email on the Circuit Cellar BBS, CompuServe (70566,777), or by regular mail at the address shown in the sidebar. While I've tested these routines extensively, errors can occur, so I appreciate any feedback that the users can provide.

Radar Display Module

Our new software overlay for this issue is the radar data display module, which will create a map using the raw station data acquired from your weather database service.

The use of radar data requests in the above auto-access-file examples was not accidental. This module searches the selected session capture file for the raw data reference:

RADU	on Accu-Data
SD	on CompuServe
RADAR/U	on Weatherbank

It then creates a data table from the received data and plots the information from the table over a U.S. map which is grid-calibrated for the stations' locations. The base map is in the GIF format, and maps with the data plots can be saved for future use (saving is also in the GIF format). The module is menu driven and self-explanatory.

Minireview of the Heath ID-5001 Weather Computer

Having finally found time to complete the ID-5001 and install it (where do the hours go?), I'm now able to give you an idea of its capabilities.

Overall, this is probably one of the better instruments that Heath has produced. The kit is definitely not for the novice; the tight layout and close trace spacing is similar to that found on boards in some of their computer kits. Precision soldering is a must since a trace short could easily occur with the use of too much solder.

The main unit functions as an indoor/outdoor thermometer, barometer, and wind direction and speed indicator (with a pole-mountable anemometer and direction vane). Current time and date are maintained, with memory to show the day's high and low temperatures, barometric pressures, and high wind speed. All functions are protected by a battery back-up power supply, so memory can be maintained in case of power failure. The display is a professional-looking back-lit LCD panel, and commands are entered from a front-panel keypad. The entire unit is housed in a metal case (with partial wood-grain trim). The user may set alarm functions for an audible or visual (or both) alert of abnormal conditions; some of these alerts depend on the presence of the relative humidity accessory.

There are three accessories available: a rain gauge, an indoor/outdoor relative humidity sensor set, and an RS-232 communications port.

The rain gauge is a self-dumping, calibrated unit with some generic-style mounting hardware. The gauge is the same unit provided with the earlier, separate rain gauge display. Since I had that unit as well, I had already solved the mounting problem. My approach was to use a l-foot length of 1" galvanized pipe with a pipe flange screwed onto one end. I drilled the pipe flange for three machine screws spaced at 120-degree increments, and then used the provided mounting straps to attach the gauge to the flange. Then, using two 2-foot lengths of 1" square steel tubing, I mounted the pipe/gauge assembly to my tower with 1.25" U-bolts. I should note here it is imperative that the gauge not be mounted under a tree limb or other overhang that could cause the readings to be in error.

The humidity sensors (which are not the same as the sensors used in the earlier Heath Relative Humidity Unit) were easy to construct. The outdoor sensor (along with the outdoor temperature sensor) can be mounted under the eaves of your house, but I recommend that you mount them instead in a suitable instrument shelter to minimize errors in readings. Shelters are available from several suppliers, one of which is:

Wind and Weather
P.O. Box 2320-WW
Mendocino, CA 95460

They will provide a free catalog on request.

The RS-232 port should be acquired if you plan to use the ID-5001 with our peripheral processor since we will support its protocols as well as those of the earlier (and less sophisticated) ID-4001 weather computer.

I recommend that you acquire all of the desired accessories at the time you purchase the main unit to allow you to calibrate everything (and, in the case of the RS-232 port, install the extra parts on the main board) at the same time.

My only complaint with the unit is the procedures used to calibrate some of the sensors. I've never been a fan of "cut-and-try" methods, and the procedures used for the barometer, temperature, and humidity adjustments seem to, at least in part, fit that mold. It is important for proper operation that you follow the calibration procedures precisely, and redo them several times to allow for fine tuning. (I actually allowed the unit to "burn in" for several days to

stabilize the components before recalibration. I also repeated the procedures three days in a row, and, even at that, I'm not satisfied that the system is tightly calibrated.) I plan to investigate the possibility of scope (or other instrument) calibration procedures to make alignment more precise. If I can produce such procedures, I'll provide them in a future article.

The preceding problem aside, this unit appears to be an excellent buy for the features provided and in comparison to other commercially available instruments of similar precision.

The Home Weather Center Peripheral Processor

As I said before, summer is on the wane, so I suspect we're all ready for a hardware project. Let's get started with our peripheral processor unit.

Our Home Weather Center system is designed to provide the following:

- Monitoring of either the Heath ID-4001 or Heath ID-5001 weather instruments, sampling and holding their data at a preselected sample rate (e.g., every 5 minutes). When an IBM PC is used to access the peripheral processor, the data is downloaded for processing in a database within the PC's software.

- Digitizing and recording of **WEFAX** satellite information in memory for later download and display by the PC's graphics card. Based on the normal configuration, the basic unit will store the most recent four **WEFAX** images in memory.

- Control of an integral **WEFAX** receiver.

- Additional peripheral support (disk? modem? direct graphics?) to be determined by user interest.

The unit will operate 24 hours a day, with an optional battery backup to maintain memory in the case of power failure. The intention of the peripheral processor is to handle the full-time data housekeeping needed for full-time operational system without dedicating a PC to these chores. The PC's graphics support, communications, and data storage and processing, are used to minimize the cost of the system.

In designing the peripheral processor and its accessories, Figure 1, I tried to consider any possible problem areas. For instance, we know that a **WEFAX** image takes about 200 seconds to be received. Our system must control the sample rate, trigger an A/D converter, take the digital value from the converter, and place it in memory. Given that we sample an image 800 times in each line of FAX data (800 pixels x 800 lines is the specified **WEFAX** resolution), samples are approximately 312 microseconds apart. Allowing 100 us for an A/D conversion, our processor has 200 us to handle its part of the routine. It's obvious that the above would not be difficult unless you consider that the system overhead must also include the worst-case scenario of the processor handling console I/O, instrument sampling, and **WEFAX** sampling at the same time. Our system needs a clock speed of about 9 MHz to be efficient.

The FAX image will need to be stored. If we store four 4-bit pixel levels in a 16-bit memory location, we'll need to have 160K of RAM for this purpose. To store four images, 640K will be needed. The requirements of storing instrument data can be met by about 300K of memory since the instruments are sampled every 5 minutes and data is downloaded at least every three days. Hence, we need a processor capable of addressing more than 1 megabyte of memory and operating on 16-bit numbers.

My choice for the processor was a Motorola MC68000 microprocessor running at 10 MHz. It fulfills the requirements stated above and keeps the price down as well. The possibility of disk system requirements, an internal graphics card, and other accessories also figured in my choice since I wanted

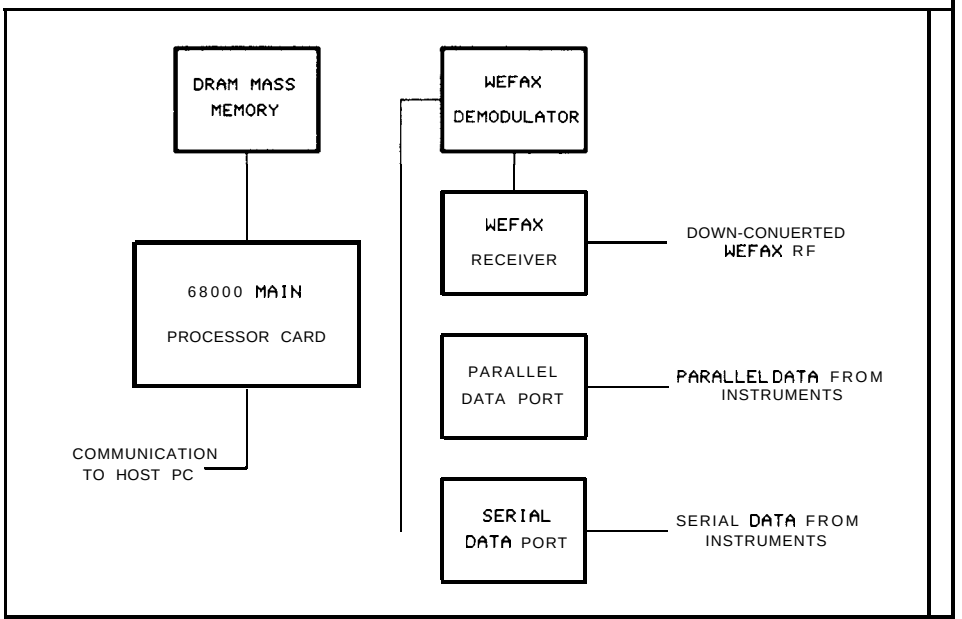
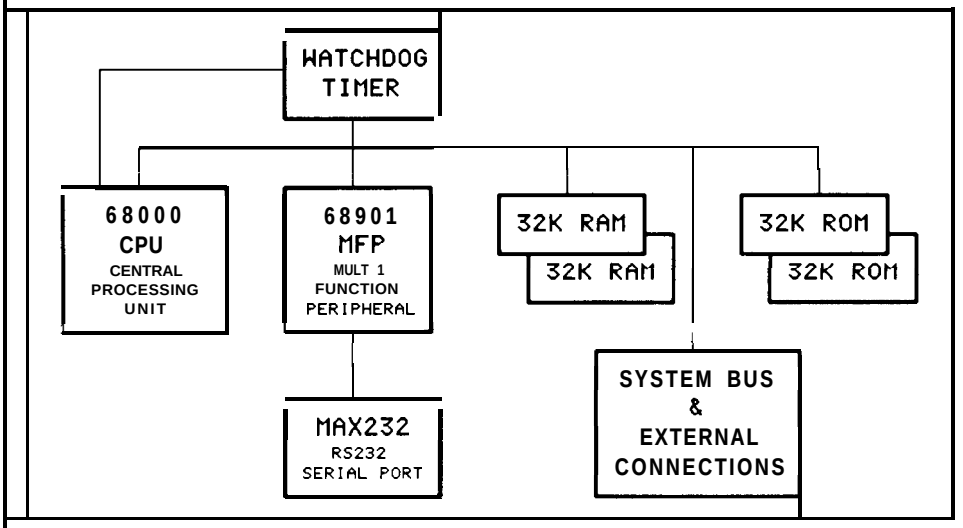


Figure 1 - Peripheral Processor and its accessories



'igure 2 - Evolution of the peripheral processor unit.

a microprocessor which had a large device support base.

With that selection made, the peripheral processor unit began its evolution to the final design, which is shown in the block diagram of Figure 2. The CPU card contains the processor, clock, and processor memory, as well as a multifunction peripheral used primarily as the console communications port (communications with the PC). This card also controls the front panel display and monitors front panel switches.

The data memory consists of a l-Meg x 16-bit RAM array with control. An optional card expands data memory to 2 Meg x 16 bits.

The parallel and/or serial ports are provided to connect instrumentation to the peripheral processor system. Additional ports can be installed if needed.

The **WEFAX** demodulator and A/D converter comprise another bus device, as does the planned integral VHF receiver for the downconverted **WEFAX** signal (more about this when we talk about the **WEFAX** unit in a future article).

All of the devices will be connected to a common ribbon cable bus and will conform to the dimensions necessary for stack-type mounting in your cabinet. (I'll discuss my recommendations for a power supply and cabinet in the next installment.)

The CPU Card

The CPU card schematic is shown in Figure 3. I've attempted to minimize board density and cost by using Programmable Array Logic (PAL) devices when they were most practical -- especially in the device selection logic, where their use replaces several **ICs.**

(For those of you who may think these custom devices carry a large price tag, don't worry. I'll provide the devices preprogrammed at a reasonable price [about $3.50 each]. If you have access to a programmer, the CUPL PAL compiler files will be available on the Circuit Cellar BBS. The individual parts, as well as complete kits, will be offered in the next issue.)

Briefly, **U101** is our MC68000 processor, which is the "brains" of our unit. It is clocked by a IO-MHz clock generator (which also supplies a phase-2 clock for use by bus devices). The operating system is contained in two 27256 EPROMs **(U103** and **U104),** appearing to the bus as a 32K x 16-bit array. The processor also has a 32K x 16-bit static RAM array **(U105** and **U106)** for on-board stack and configuration table use.

U102 is the multifunction IC (MC68901), which handles interrupt **prioritization** and communications with the console unit (your PC). **U108** creates the necessary **150-ms** reset pulse width; **U107** operates as a "watchdog" circuit, causing a reset if the processor should halt for some reason.

CN1 A/ 1 B is the bus connection to the other cards in the unit and provides all of the needed signals to and from the processor, as well as reset and selection information. Some of the bus lines have been left open, reserved for data between other cards in the unit.

We'll hold the discussion of other details until the next installment, when I'll provide the schematic for the memory card, discuss the firmware, and provide ordering information for the parts and kits.

Until then, remember to drop me a line if you have input as to other possible accessories for this unit. I've already received comments requesting some form of mass storage, an integral video display card, and a frame buffer for high-resolution **EGA-to-**NTSC translation. What are your ideas? ■

Software for Circuit Cellar INK projects is available on the Circuit Cellar BBS and can be downloaded free of charge. If you would prefer more personal service and would. like to receive the WDPS software on disk, I will eend it to you for $6.00 ($5.00 copying/disk fee, $1.00 P&H). Source code will be handled in a manner similar to "shareware," in that you will send a registration fee of $36.00. This payment will cover source for all modulw in the system and all updates or fixes. Software will be released in module groups. You will receive the appropriate disk shortly after it is published.

To order software or source code on disk, send check or money order to:

Mark Voorheee
P.O. Box 27476
Phoenix, AZ 85061-7476

Allow 30 days for shipment.

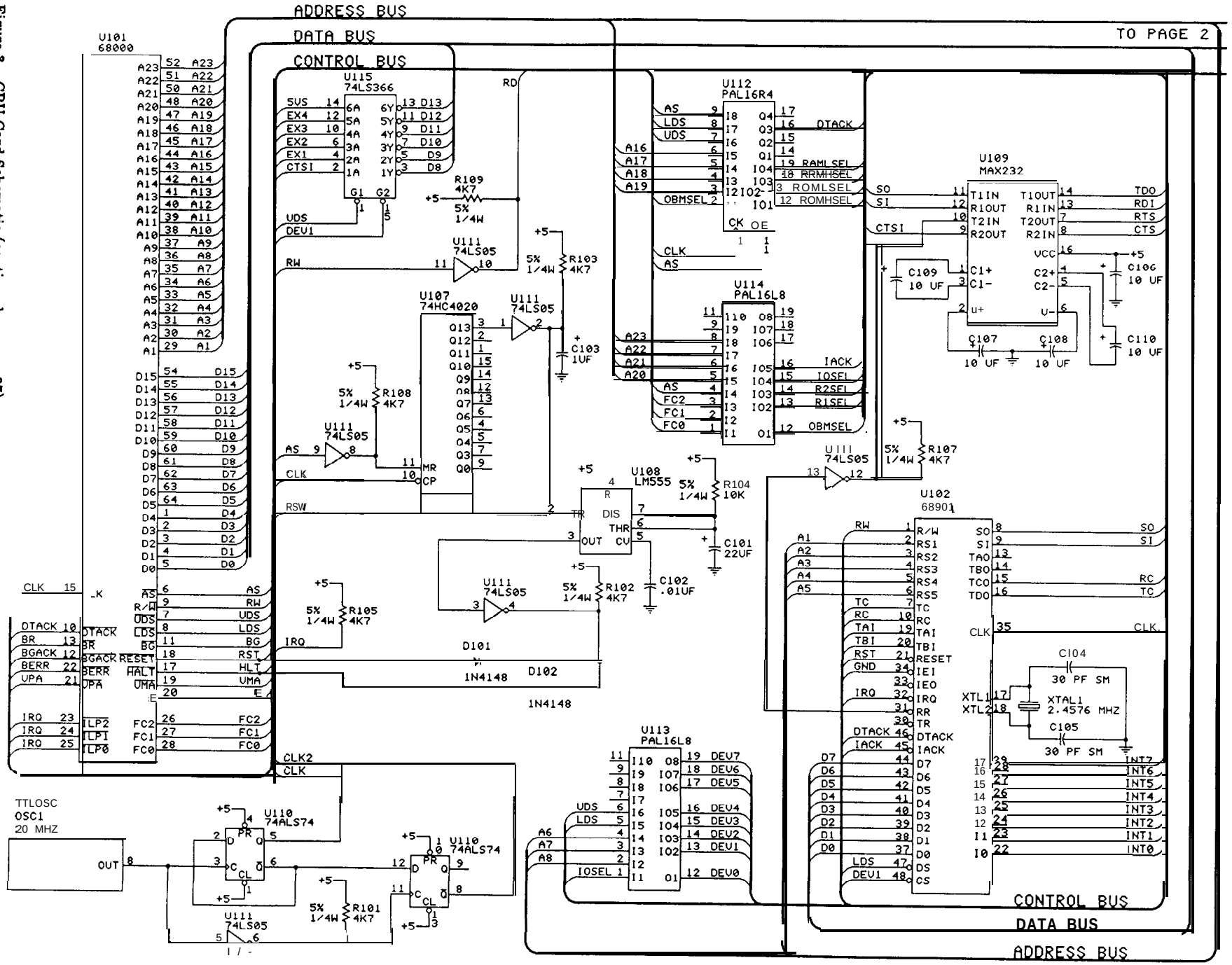
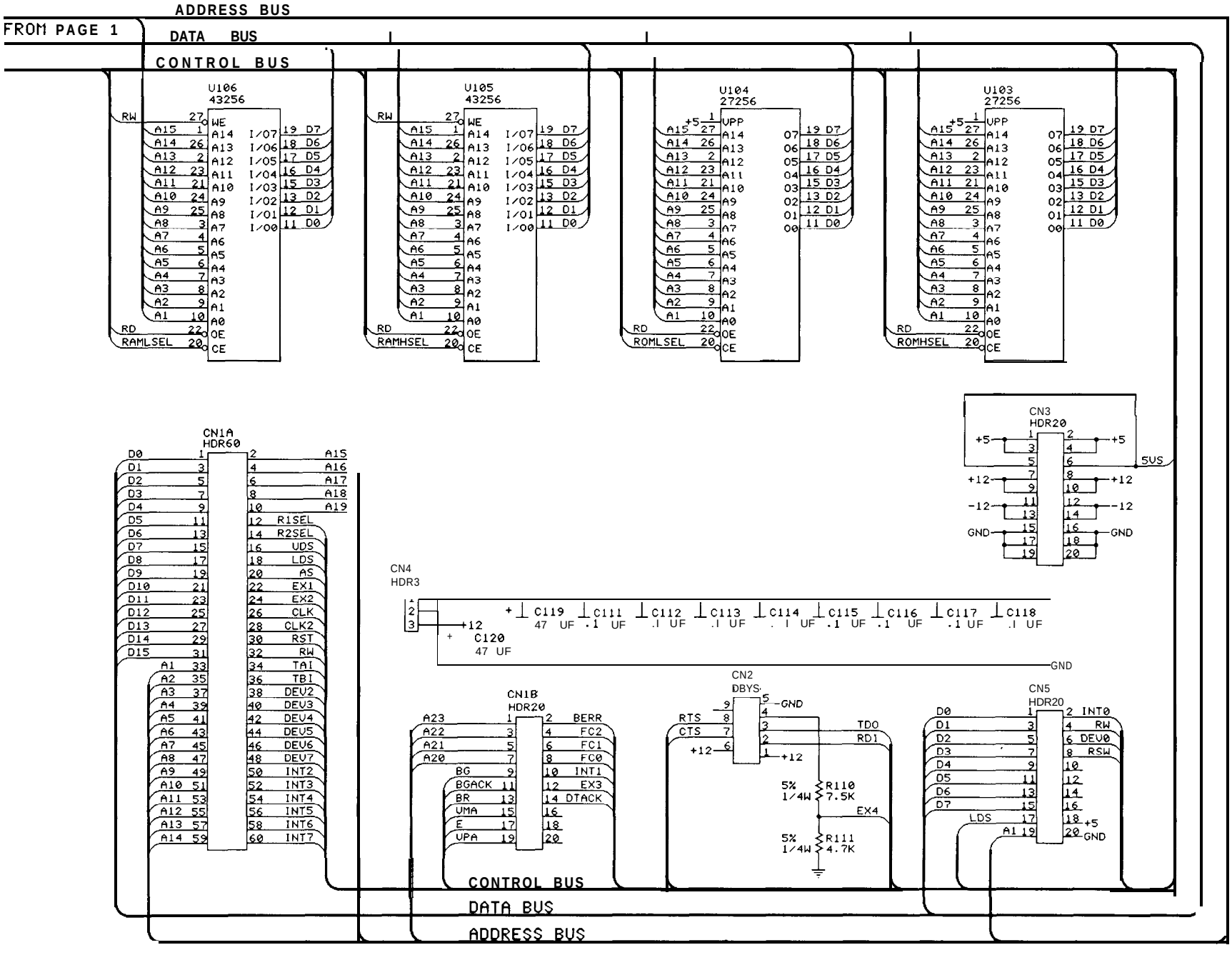


Figure 3 - CPU Card Schematic (continued on page 27)



(Figure 3 continued from page 26)

Ctrl *Guest Editorial*

INK SPOT

BIGGER is not neccessarily better

by Ezra Shapiro

The computer industry is finally being trapped by its own marketing strategy, and the result is more grief ahead, both for the consumer and the industry itself. We're being buried in a landslide of over-powered, redundant products that will make the jump into the next generation of computing incredibly difficult to accomplish, at least psychologically if not physically. The syndrome is most obvious in the software arena, but there are going to be repercussions in the hardware market as well.

The problem is that the mass market has been convinced that computing power can best be measured by cost, size, and newness rather than efficiency, elegance, and suitability. Buyers stampede to their dealers the second any major company announces an upgrade to an existing product. Why? Because they believe that if they fail to move up, they'll be missing out on "powerful new features." Sometimes those features are desperately overdue, but often as not, the motivation for buying has less to do with need than with the desire to stay at the forefront of the technological revolution.

The manufacturers chuckle all the way to the bank. Fiscal projections based on exponential growth need artificial mechanisms to continually pump up revenues. Upgrades are a perfect way to inject new buying enthusiasm into markets already saturated. This is simply a rule of economics.

As an example, look at the marketing ramifications of OS/2 and the Micro Channel Architecture: IBM will sell new boxes, board vendors will sell new cards, and software companies will rewrite existing programs. Lemmings in the marketplace will cheerfully purchase the lot, with no ***immediate benefit,*** merely because they've been told to do so. It may be years before OS/2 results in meaningful advances, if ever, but boy, does it produce revenue in the meantime!

Consequences? First, manufacturers in this highly competitive business now feel intense marketing pressure as well as technological pressure, and a lot of products are released before they're fully debugged. A couple of software firms have gone so far as to sell programs to consumer "beta testers," thereby disclaiming responsibility for errors while still generating appreciable cash flow.

Second, the rush to market encourages sloppy design. The solution for many companies is to throw engineers or programmers at a project with more concern for schedule than quality. Production models of computer systems are sold with ROMs that change weekly; you've got to know the exact date of manufacture to determine software compatibility. Boards are shipped with jumpers of wires soldered to the underside to correct flaws in the layout. On the software side, you get memory-hungry, crash-prone applications written by hordes of semiliterate C programmers.

Third (and this is the point that's really scary), we're seeing a phenomenon in software design that can best be called "creeping integration." In an effort to expand their markets, developers take perfectly good applications and graft on features from other disciplines. Word processors sprout spreadsheet functions, spreadsheets become desktop publishing engines, databases add telecommunications capabilities, and so on. These overlapping hybrids are invariably huge and less robust than their forebears. They use unique data structures and effectively short circuit operating system design.

In the coming era of multitasking, multiprocessing, networked hardware, these hulking monstrosities represent an approach exactly opposite from what is needed. RAM and CPU time will become even more precious than they are today, and overlapping functions will be an expensive annoyance. The goal should be trim, fast, efficient software that's talented in data exchange and sharing. After years of convincing consumers that bigger is better, how will the industry reverse itself to promote downscaled programs? Will today's leaders collapse, to be replaced by firms that understand good design? Or will the enigma be ignored, forcing hardware vendors, operating systems engineers, and computer users into costly, inefficient workarounds?

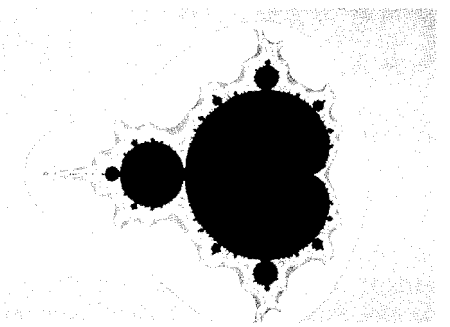
The answer, of course, lies in another basic principle of economics. If we, as consumers, vote against unnecessary bigness with our checkbooks, by consciously buying only those tools that are right for the tasks at hand, we might be able to stem the tide. If that happens, the next generation will mean easier computing for everyone. ■

Ezra Shapiro writes about computers for Byte, MacWeek, and MacUser.

CONNECTIME

THE CIRCUIT CELLAR BBS
300,' 1200/2400 bps
24 hours/7 days a week
(203) 871- 1988 -- 4 incoming lines
Vernon, Connecticut

Sysop: Ken Davidson



In the October, November, and December 1988 issues of BYTE ***magazine, Steve will be presenting a Circuit Cellar project called a "Mandelbrot Engine." (I might add that these will be the last Ciarcia's Circuit Cellar construction articles ever to appear in*** BYTE.) ***There has been lengthy discussion about the engine on the Circuit Cellar BBS lately, so I'm starting this month's column with that discussion.***

For those unfamiliar with the Mandelbrot set, or have heard all the uproar and wondered what the commotion was all about, its popularity probably started with an article written by A.K. Dewdney for the "Computer Recreations" column in the August, 1985 issue of Scientific American. ***In it, Dewdney describes a set of numbers which, when passed through a deceptively simple equation repeatedly and plotted, create marvelous pictures containing spirals, whorls, and "black holes." When a small portion near the edge of the set is magnified, just as much beauty and detail emerges as the new picture is calculated. Limits to how far one can zoom in on the set are created only by the precision of the computer calculating the picture. The picture above is a black-and-white representation of the entire set and its surroundings, but it doesn't really do the set justice. Only on a full-color, high-resolution screen can you appreciate the elegance.***

Based on what is known as "fractal geometry," the set is named after Benoit B. Mandelbrot of IBM's Thomas J. Watson Research Center. Mandelbrot's research in geometry led to the development of fractal geometry, or "the mathematical study of forms having a fractional dimension."

"Chaos theory" is also related to fractals and the Mandelbrot set, and a discussion about chaos is what led into the Mandelbrot discussion. The chaos thread would fill another whole column, but is still on-line for those interested. Anyone interested in finding out more about chaos theory, fractals, and the Mandelbrot set should look into the book entitled Chaos: Making a New Science ***by James Gleick, published by Viking Penguin Inc., as well as the*** Scientific American ***article cited above and, of course, Steve's upcoming*** BYTE ***articles.***

Msg#: 3149 *GENERAL*
From: BEN TREMBLAY
To: ALL USERS
Subj: CHAOS

Scientific American's "Computer Recreations" section is a superior source, but the thin edge of the wedge for me is Art Matrix. From Homer Smith's place you can not only connect with like-minded folks but source FORTRAN code and paradigmatic video. Happy **frac**-tailing!

Msg#:3158 *GENERAL*
From: HOMER SMITH
To: ALL USERS
Subj: CHAOS

Hello people, this is Homer Wilson Smith of Art Matrix. Art Matrix is a legal Partnership of myself and Jane Elizabeth Staller, both of Ithaca, NY and Cornell University. We work with John **Hamal** Hubbard and the Cornell National Supercomputer Facility exploring chaos, fractals, and other dynamical systems. I do all of the supercomputer programming for Dr. Hubbard and in the past three years have produced a number of interesting products for the beginner (and not so beginner) **who is** interested in chaos. **We intend** that chaos be taught in the 9th through 12th grades within a few years and are trying desperately to get thin material into the hands of anyone who can help with this goal either on a small level or on a planetary level.

Cur main products in present time are:

1.) A postcard set in color of 36 different images from varioue equations.
2.) A slide set of 140 **slides** from the work we have done.
3.) A do-minute video (noon to be 120 minutes) containing **zooms** into the Mandelbrot set set to music.
4.) We also sell floppy disks with the FORTRAN code for each iterated equation we have studied and the coordinate sets for the slide nets.

The video is $26 for the first copy and $10 for each additional. Don't let the price fool you; this cost us over 100 hours of supercomputer time to create. The slide set is $30 and the postcard set is $10 for two packs. The floppies are $15 for one and $20 for both.

Art Matrix in a cooperative arrangement between the Cornell National Supercomputer Facility, IBM, and the National Science Foundation. We fully intend to make chaos known to the world and if you would like to help or just get more information please call (607) **277-0959** 24 hours a day or write Art Matrix, PO Box 880, Ithaca, NY, 14851-0880. Thank you.

Msg#: 3171 *GENERAL*
From: STEVE CIARCIA
To: HOMER SMITH
Subj: MULTIPROCESSOR MANDELBROT ENGINE

Well, speak of the **devil!** I've been looking for you guys. Let me explain what I've got going (I will be posting much more on this design in the next few days). **I'm doing** a project for BYTE on multiprocessing in the fall. Since this is usually the kind of article where everybody talks about it but nobody can demonstrate it, Ed Nisley and I are actually building an array processor as the project. Each processor is a **single-**chip microcontroller containing the Mandelbrot calculation. There are 1 to 8 processors on a board **and 1 to 8 boards in the system (actually we believe you can go to 32 boards). An IBM PC/AT assigns coordinates to the array depending upon how many processors are available. The results are transferred back from the array to the PC and plotted on an EGA display (eventually it will be VGA).**

Now for the good news. Five processors (each running 12 MHz) collectively run at the same speed as an AT (about 3.5 processors at 16 MHz). Our demonstration unit has 64 processors and runs about 13 times faster than an AT. If we use a CMOS processor that runs at 16 MHz, we can make it 17 times faster. Finally, if we go to 256 processors we are talking about 68 times the speed of an AT in a box the size of an AT!

At those speeds we **can finally deal with fractals without having to wait days or using supercomputers.**

I'll be posting more on this Mandelbrot engine in the next few days. My prototype uses 8751 processors that can cost about $40 each. If we perceive enough interest by you and others on this BBS, however, we would consider masking an **80C52** CMOS 16-MHz chip (minimum order 2000 chips) and making PC boards to create your own **1-to-n-**sized array. We estimate that a complete 8-processor unit would be less that $200 (kit) if we mask the chip. The **64-chip** prototype probably cost about $6000 by comparison.

I am very interested in your video tape and slides for use in my article. I will call you to discuss it.
-- Steve

Msg#: 3190 *GENERAL*
From: KEVIN SCHNEIDER
To: STEVE CIARCIA
Subj: REPLY TO **MSG#** 3171 (MFG)

The thought of building an array processor which can be attached to the IBM PC is very exciting. Especially at the prices you speak of. It seems that your project may be limited to handling fractals at the moment. While this is fine (for now) I hope that you will discuss at least casually how one might begin to consider programming the array processor for other applications. My personal interest lies with mechanical engineering applications (**finite** element analysis). I think a project like this could be quite instructive.
Best wishes,
--Kevin

Msg#: 3184 *GENERAL*
From: RICHARD ANDREWS
To: STEVE CIARCIA
Subj: REPLY TO **MSG#** 3171 (MFG)

Soundz like a great project. I, for one, would be very interested in an array processor with an eye towards image processing. Perhaps you could use the **ImageWise** board set together with the array processor, although it would be nice if the I/O was something other than RS-232.

Any idea when this is likely to be published?

Msg#:3201 *GENERAL*
From: STEVE CIARCIA
To: RICHARD ANDREWS
Subj: REPLY TO **MSG#** 3184 (MFG)

The articles will be in October, November, and December 1988 issues BYTE. However, we may have something about it in INK as well. As I said, unless you implement the multichip version of each controller element, this is not reprogrammable. It is designed for Mandelbrot and fractals. Also, a bit of a correction. Since neighboring processors don't "talk" to each other, I think this is better referred to as a star configuration rather than a true array processor. You have to see this thing to believe how fast it cranks!
-- Steve

Msg#: 3179 *GENERAL*
From: STEVE CIARCIA
To: ALL
Subj: MULTIPROCESSOR MANDELBROT ENGINE

The following is excerpted from some recent correspondence elsewhere and I'd like your opinion:

Since I don't have room for long humorous intros in BYTE these days, the real story about why this Mandelbrot generator is being built may never come out unless I say something. You see, I have this entertainment room where I go to sit and relax: listen to music, etc. I have a 6.5-foot Kloss 2000 projection monitor and it has a RGBI (as well as composite) video input. It was designed for computers.

Eversince **I got** the damn thing1 have thought about puttingsome computer-generated pattern on it (I've got a few other ideas in mind) while I listen to good music. Fractals have always interested me, so why not? You know the rest.

Well, as we started looking at the architecture of our multiprocessor project we came to some realizations. While I'll probably discuss all the variations of multiple processor connections, the specific Mandelbrot generator I am demonstrating is a branched tree with a single parent with all the rest offspring.

This is a very inefficient multiprocessing approach for general tasks but very efficient for THIS specific use. Since fractal calculations take so long and we are only communicating a few bytes of information each time, there is no need to implement another multiprocessor architecture whose major benefit is faster internal communication (and a LOT more complexity). In our Mandelbrot Engine all subprocessors talk to a single master processor, an IBM PC/AT.

The PC first scans the Multiprocessor Fractal Generator (MFG) to determine how many subprocessors there are (1, 5, 16, 255, etc.). Depending upon how many processors are available, the PC divides up the coordinate spectrum and sends each subprocessor a set of coordinates tocompute on. **Every** subprocessor contains **the same** computing program and all **grind away in parallel. I guess we could call it an array processor of sorts.**

In any case, as it is running, the AT polls the individual subprocessors and accumulates the data.
(Continued next message)

Msg#: 3180 *GENERAL*
From: STEVE CIARCIA
To: ALL
Subj: MFG (CONT)

Each subproceseoris acomplete microcomputer: processor, RAM, EPROM, serial, and parallel I/O. Because I want to finish this project some time this year, I have chosen to use single-chip microcomputers to implement the subprocessors and a more expanded computer for the parent. Anyone wishing to use this system for other applications might

consider just expanding each subprocessor block to be a more generalized computer with more memory. The basic polling software and parent-to-PC communications software would still be of considerable value.

When we started, we felt that 16 processors would do fine but, you know how hardware people are .

Looking at the communication speeds required between the parent and the subprocessors, the parent apparently has plenty of time to deal with more than 16 children. In fact, we think its limit may be 256!

So, instead of 16 processors, the Circuit Cellar MFG prototype will be implemented with 64 processors! And, we'll try for 64 MIPS! (Let's call these LMIPS for "little" MIPS. The 8751 processor does indeed execute 1 instruction per microsecond but I'd hardly compare it to a '386. But then it is more fun saying 64 MIPS!) :-)

The way it is designed the speed is dependent upon the number of processors (as it should be in a multiprocessor syztem). All subprocessors are connected on a serial party line to the parent host processor. If there is only one subprocessor then the speed will be 1S (one snail). If there are four subprocessors then it will generate fractals at 4S.

We estimate that five subprocessors (@ 12 MHz) are roughly equivalent (if you don't mind comparing apples to bananas) to an 8-MHz 80286.

So, after we analyzed the 8751 a little, we decided it made the most cost-effective multiprocessor and it would be achievable at a reasonable cost (I figure this 64-processor prototype boxwillcost about $6000 with labor) for a one-of-a-kind prototype. Of course, now we are discussing masking a low-cost CMOS version for production.

You can use anywhere from one subprocessor to 256; they are daisy-chained. In my prototype, we are putting eight processors on a card and eight cards. The schematics will be presented showing 8-processor increments. So, yes, by design you can use increments of 8 (or 7, or 3, or . . .)

Msg#: 3181 ***GENERAL***
From: STEVE CIARCIA
To: ALL
Subj: **MFG** (CONT)

To anticipate a few of your questions, let me say the following: This Mandelbrot Engine array is designed specifically to solve fractal/chaos problems and has a few other tricks. However, it is not a general array processor (only the 8751 version can be reprogrammed for other things) and it is not a substitute for a PC or a mini VAX. The article is not about just this Mandelbrot array processor but rather multiprocessing in general. Most of the discussion about the fractal computations or the Mandelbrot program in general should take place on this BBS.

Msg#: 3182 *GENERAL*
From: STEVE CIARCIA
To: ALL
Subj: **MFG** (CONT)

(Note: provided that you like my concept of array processing but want some programmability, that is easily remedied. The two preceding articles are for an 8051 development system. When you look at my array processor, merely replace the one 8751 chip with an 8031-based 4-chip computer with RAM and EPROM. Still relatively small, such a configuration would be programmable. With only about four chips you could have an 8031 and 32K each of EPROM and RAM. I would have considered making this modular CPU approach myself but it is relatively expensive unless you have some application in mind. Given my specific application, 8751s were a quick, albeit still not cheap, solution to demonstrating array processing. Remember, the project isn't to build a replacement for the Cray X-MP.)

The PC software starts by plotting the standard Mandelbrot set and, like most other fractal generators, allows you to specify the number of iterations and the coordinates to "zero" in on specific sections of the scene for "closer" looks. Of course, at the speed of the MFG, we don't have to wait long. Presently the display is 16-color EGA. There may be an upgrade (only to the PC software, no changes to the chip) to 256-color VGA later.

So tell me what you think. Just don't ask me if it will run Lotus faster. :-)
--Steve

Msg#: 3215 *GENERAL*
From: BEN TREMBLAY
To: STEVE CIARCIA
Subj: REPLY TO MSG# 3202 (MANDELBROT ENGINE)

Elegant. Tell me, though, how many digits precision? As Homer said, not so much of a worry for end results but I find it frustrating not being able to zero right down on the point I want.

Msg#: 3231 *GENERAL*
From: STEVE CIARCIA
To: BEN TREMBLAY
Subj: REPLY TO MSG# 3213 (MANDELBROT ENGINE)

I'd rather Ed Nialey answer that question. He's around here some place.
-- Steve

Msg#: 3239 *GENERAL*
From: BEN TREMBLAY
To: STEVE CIARCIA
Subj: REPLY TO MSG# 3231 (MANDELBROT ENGINE)

Sounds good. When I ran two NEC 6001As together (ancient 16K Z80-based orphans) I found that, unless the two coordinates had very similar "sizes," there had to be wait states introduced -- and that with maximum iterations of only 100. My question is this: with maximum iterations of something larger, and X number of processors, how would you manage "data ready" without always having to wait for the "worst case" coordinate to pop up? Could the AT handle housekeeping with an asynchronous "catch as catch can" system?

Msg#: 3249 *GENERAL*
From: STEVE CIARCIA
To: BEN TREMBLAY
Subj: REPLY TO MSG# 3239 (MANDELBROT ENGINE)

In the present 8751 version, the AT does have to wait for the worst coordinate. Fortunately, since there is usually a string of these 100 or so iteration points usually in a row, by the time the AT has waited for the first point, the following points have finished and no further waiting is required. Or, at least it's something close to that.
-- Steve

Msg#: 3271 *GENERAL*
From: DAVE MILLER
To: STEVE CIARCIA
Subj: REPLY TO MSG# 3249 (MANDELBROT ENGINE)

Steve, will the code or algorithm be available? I would like to try this with an 18-MHz 8796 microprocessor.

Msg#: 3272 *GENERAL*
From: STEVE CIARCIA
To: DAVE MILLER
Subj: REPLY TO MSG# 3271 (MANDELBROT ENGINE)

An elaborate explanation of the processor code will be provided and executable code will be available here for download, but no source. The complete source and executable code for the PC's communication and display will be provided, however.
-- Steve

Msg#: 3243 *GENERAL*
From: ROBERT WELKER
To: STEVE CIARCIA
Subj: MULTIPROCESSING ENGINE

Your multiprocessor Mandelbrot engine sounds pretty darned interesting.

Observation one: there are probably enough people exploring the Mandelbrot set to justify a dedicated processor design. On the other hand, this would be a great project to do very open-ended; that is, once we eager experimenters build the main guts of our 8-, 64-, 128-, 256-level array processor we could get a lot of exciting and informative mileage out of it by reprogramming and adapting it slightly for other tasks. There are probably a half-dozen interesting projects you could initiate that your multiprocessor would be the heart of. Our hardware investment could be spread out over many more projects.

Observation two: you may even wish to use this project as the focus for an entire book. One of the things I've always wanted to see available (but seldom find) is a comprehensive text associated with electronic projects. When I say comprehensive I don't mean start from scratch with basic electrical theory, but rather a text which explains thephysical phenomenawe areexploringindepth, hand-in-hand with an explanation of how the electrical tools we use relate to it.

For instance, I could fix a temperature controller long before grasping that it worked as an analog computer modeling the behavior of the load. Not understanding the controller's place in the "big picture" wasn't a problem on the troubleshooting end (a bad cap is a bad cap), but process control was a mysterious subject.

(Note that observation two is a general comment on technical texts in general. I haven't read any of your books yet, and have only recently sent in for a CCINK subscription).

In all, it sounds like a great project -- worth it even to look at the results in green!
-- Bob

Msg#: 3251 *GENERAL*
From: STEVE CIARCIA
To: ROBERT WELKER
Subj: REPLY TO MSG# 3243 (MULTIPROCESSING ENGINE)

When you run a Mandelbrot set on a PC you might as well take a weekend vacation (unless it has a coprocessor). At least with our Mandelbrot engine it runs fast enough (a 64-processor unit plots the Mandelbrot set in 3 minutes) to actually see it plotting and not go to sleep.
-- Steve

Msg#: 3280 *GENERAL*
From: ED NISLEY
To: ALL USERS
Subj: MFG DETAILS

This thing is designed to do Mandelbrot calculations. It's not a general-purpose array processor. The calculations don't need any communication between the processors, so that's what we've got. A "real" array processor should have some communication, but there's no good way to pull it off.

The internal math is fixed point, with 60 fractional bits. That gives a dynamic range of -8.0 to +7.999, which is ideally suited to Mandelbrot calculations, but not much else. The routines support addition, complementing,multiplying, squaring, but not division. The multiply routine is a 300-line macro that expands into about 3K of code -- it's written without run-time loops! There are some interesting side effects, because some of the routines check for overflow and clamp the numbers to 3.9999 ...

With 60 fractional bits you can eoom in pretty much to your heart's content ... we lose precision at about 1.0E-12; you'll lose interest first. The iteration counts can range up to 64K-1, which ought to be enough for any picture you can get in your sights. With 8-byte reals there isn't enough RAM to hold all the values for Julia calculations ..., unless we go to an 8052-style processor.

The results stream out of the array in daisy-chain order, which means the AT has to wait for the slowest processor. The big advantage of this method is that the communications channel doesn't have to carry addressing information, so the data rate isn't a limiting factor for any practical image. We could have the AT poll the processors, but then the effective serial data rate would be under 1/3 the actual rate, which isn't a good idea either. We could have any processor that's not ready send a eero count, but then the AT has to maintain a map of which points are filled in, which are pending, and so forth ... this puts the AT smack into the critical path, which was exactly the reason we wanted an array processor in the first place!

Now, the good news:

This thing is a killer. Steve said that it takes about 3 minutes to compute the overall image with 64 processors. What he didn't say is that much of that is spent drawing the dots on the screen -- the Mandelbrot engine is waiting on the AT!

The initial image is the whole Mandelbrot set, centered on (-0.405,0) with a real axis siee of 3.59 and aspect ratio of about 1.33. The image has 19.7% black points (44154 pels) with an average 6.6 iterations/point and computes in 2.8 minutes. If overhead in transmission and dot drawing weren't a factor, it would take 1.9 minutes.

A better measure is to run the same scene with an iteration count of 128. It takes 6.2 minutes to display 37667 Mset points (16.8%), with an average of 15.0 iterations/point. The communications and display overhead accounts for about 1.8 minutes of that time.

A LONG computation with this thing set to 1024 iterations is maybe 45 minutes ...

I don't have comparative times, but it greatly outruns my 8-MHz AT/10-MHz 80287 running the official IBM MSET program. Benchmarking this thing is a problem because we don't have an apples-to-apples comparison with a '286 running the same code.

Ballpark performance: each iteration takes 5 to 7 ms per processor (including ALL overhead). Divide by the number of processors to get the average time per iteration for the whole array. For 64 processors it's about 94 us, including the data transmission and dot drawingtimes. Your mileage may vary, but that's a good starting point.

Multiply by the number of points on the screen (224,000 for an EGA) and multiply THAT by the average number of iterations per point (which depends on the scene). Divide by 60,000 to get seconds and send us a check!

From the esoterica of fractals, we now turn our attention to something a bit more mundane, but certainly no less important. Left-handed people often have a difficult time living in this predominantly right-handed world, even down to which mouse buttons to use while

running a CAD or drawing program. The following offers some suggestions for one such dilemma:

Msg#: 2810 *GENERAL*
From: RICHARD ANDREWS
To: ALL
Subj: LEFT-HAND MICE

Mice are really handy little gadgets but are somewhat difficult to use if you are left handed. It would be great if there was a utility that would allow you to swap the two buttons (or outer two) on the mouse so that the primary switch will be located under the index finger. Anybody out there know of anything?

Msg#: 2839 *GENERAL*
From: JEFF BACHIOCHI
To: RICHARD ANDREWS
Subj: REPLY TO MSG# 2810 (LEFT-HAND MICE)

Ever consider opening that little bugger up and swapping the wires from the outer buttons? The mouse would be somewhat personalieed, but the righties would then know what lefties have to go through in this right-handed world!
--Jeff
P.S. I'm right handed!

Msg#: 2845 *GENERAL*
From: RICHARD ANDREWS
To: JEFF BACHIOCHI
Subj: REPLY TO MSG# 2839 (LEFT-HAND MICE)

I've thought about doing that but the mouse would have to be used by those poor souls that had the misfortune of being born right handed. I would prefer to have a driver that I could load when I want to use the system. Thanks for the suggestion.
--Rich

Msg#: 2848 *GENERAL*
From: JOHN COOK
To: RICHARD ANDREWS
Subj: REPLY TO MSG# 2810 (LEFT-HAND MICE)

If you had the time and the patience to actually disassemble the mouse driver and if you knew the codes for the switches (I'm sure you could get the info if you asked but I haven't tried), you could then switch the codes in the program and reassemble it under a new name. Then, instead of loading the normal mouse driver, load your lefty driver.
--John

Msg#: 2992 *GENERAL*
From: RICHARD ANDREWS
To: JOHN COOK
Subj: REPLY TO MSG# 2848 (LEFT-HAND MICE)

I thought of doing that but I'm lasy and hoped that someone else had already done the work.
--Rich

Msg#: 2857 *GENERAL*
From: PETE CHOMAK
To: RICHARD ANDREWS
Subj: REPLY TO MSG# 2810 (LEFT-HAND MICE)

Why not install a small slide switch in the back of the mouse to swap the buttons around?
--Pete

Msg#: 2993 *GENERAL*
From: RICHARD ANDREWS
To: PETE CHOMAK
Subj: REPLY TO MSG# 2857 (LEFT-HAND MICE)

It's not my mouse, but rather one that is shared among several people. I use it just often enough to for the buttons to be inconvenient. Thanks.

Noise pollution is a big problem for those living in urban ***areas or close to major roadways. Active "sound deadening" devices offer the potential of canceling the offending noise altogether, resulting in a quieter environment. The following discussion centers around such sound deadening devices:***

Msg#: 3901 *PROJECTS*
From: GREG CROASDILL
To: STEVE CIARCIA
Subj: FUTURE PROJECT??

This is my first time on your board. I have called up looking for information on some sort of sound "deadener." I imagine that such a device could be made simply by reading in a sound, determining its average waveform, and then generating a sound that is 90 degrees off from that average. Do you have any projects planned for this sort of device? I just bought a house close to the freeway and would like to kill some of the truck rumble. Thanks for any info you can give.
--Greg

Msg#: 3938 *PROJECTS*
From: STEVE CIARCIA
To: GREG CROASDILL
Subj: REPLY TO MSG# 3901 (FUTURE PROJECT??)

Actually, I've been reading a lot about that topic lately and do have some interest in the subject. I can't say that there will be a specific project on sound deadening, but it may be a natural fall-out from some elaborate high-performace A/D project. Stick around. :-)
-- Steve

Msg#: 3940 *PROJECTS*
From: BOB PADDOCK
To: GREG CKOASDILL
Subj: REPLY TO MSG# 3901 (FUTURE PROJECT??)

Would a sound deadener have to project its SO-degree phase-shifted signal backwards in time (so that the two 0^{o} and 90^{o} waves arrive together) to get silence? The last apartment I lived in had a front door that would resonate every time an 18-wheel truck would go by (very annoying in the middle of the night). It might be anothersourceof noise -- sonic-induced ground tremors -- that causes the house itself to make noise.

Msg#: 3943 *PROJECTS*
From: ALEXANDER SCHNEIDER
To: BOB PADDOCK
Subj: REPLY TO MSG# 3940 (FUTURE PROJECT??)

You would simply need to generate the phase-shifted signal (actually 180 degrees, I believe) as close as possible to the source of the original noise.
-- Alex

Msg#: 3976 *PROJECTS*
From: GREG CROASDILL
To: ALEXANDER SCHNEIDER
Subj: REPLY TO **MSG#** 3943 (FUTURE PROJECT??)

Right. 180 degrees. I don't know what I was thinking (90 would just make more noise). Since the source of the noise is the other side of a 7' cement wall, I know where I'll put the speaker. Maybe I can get the DOT to pay for this ... Thanks.
--Greg

Msg#: 3985 *PROJECTS*
From: JEFF JENSEN
To: GREG CROASDILL
Subj: REPLY TO **MSG#** 3976 (FUTURE PROJECT??)

About two weeks ago, National Public Radio ran a feature during "All Things Considered" about noise pollution and attempts at active noise cancellation. They spoke to some East-coast company about their efforts at reducing noise in a room or enclosed space, as well as reducing exhaust noise on vehicles. They also spoke with a company developing aviation headphones with active noise cancellation built into them.

In theory, you sample the sound and reproduce the noise 180 degrees out of phase at the point of the listener. In actuality, since the noise source and the cancellation source are not at the same place, phasing will change depending upon where the listener stands. Other problems like amplitude differences at the listener's location, attenuation of some frequencies, and the speed of sound also get in the way. The net of all the stories is, it sounds (pun not intended) simple to cancel sound, but in practice, the process is tricky.

In the noisy room, they were able to cancel sound only in a localized area (between a pair of speakers). The demo car had no muffler, and at a constant speed and load, was dramatically quieter than no muffler. It lost it when the car revved up or changed load, until the noise characteristics stabilized (volume, frequency components, and speed of the exhaust gasses through the tube.) It appeared that the approach this company took was a feedback loop, where the noise is sampled close to the source, a transducer creates the cancellation waves, and another microphone positioned near the listener monitors the results. Then the results are used to tweak (filter) amplitude, frequency attenuation, and phasing characteristics of the sampled noise to further improve the cancellation signal. Hence the problems keeping up with noise that changes characteristics.

The aircraft headphones seemed to be less sophisticated, possibly because the design limits didn't require dynamic changes to the sampled noise signal. The headphones either had a second pair of transducers or a method of mixing the noise-counteracting signal with the headphone signals. Phasing could be addressed in the placement of the sampling microphones, and frequency response and amplitude could be hard-wired into the circuit. Although the listener could move in relationship to the noise, the sampling microphone, the canceling transducer, and the listener stay in the same relationship.

This brings up memories of Maxwell Smart and the "Cone of Silence." What's that, Chief7
--Jeff

The Circuit Cellar BBS runs on a IO-MHz Micromint OEM-284 IBM PC/AT-compatible computer using the multiline version of The Bread Board System (TBBS 2.0M) 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) 871-1988. Set your modem for 8 data bits, I stop bit, and either 300, 1200, or 2400 bps.

UPDATE: *Additional information on previous articles*

Circuit Cellar Neighborhood Strategic Defense Initiative

Building the Bottle Launcher and Gantry

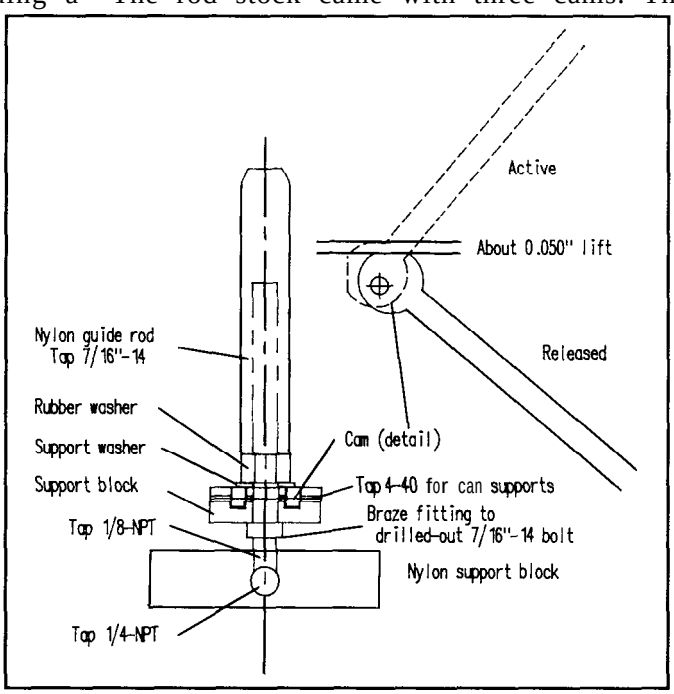
by Ed Nisley

Contrary to what some people think, the Bottle Rocket is a real gadget that actually works. After all, the Circuit Cellar wouldn't have it any other way! For those of you who are thinking of building a CCBM system, here are the construction details for the expanding seal launcher.

You'll need a lathe and milling machine to make most of the parts and some brazing or silver soldering to fit them together. Fortunately, none of the dimensions are particularly critical! You can use almost any materials that come to hand, although I'd suggest aluminum and brass if you're planning an extensive series of water launches.

The basic structure is a driveway sealer can, mounted upside down. The lid has a circle of 3/4" plywood bolted underneath to give it some rigidity and provide an anchor for three 3/8"-16 bolts that serve as feet. Centered atop that is a 3.5" wood cube cut from a 4x4 post, which I ran through the milling machine to make the top and bottom surfaces plane and parallel. The launcher proper mounts on the wood block and has three main parts: the guide rod, the expanding seal, and a nylon block base.

The guide rod was turned to about 0.845" dia from a length of nylon rod. It's 5 inches long, with a chamfer on the top that reduces the BANG when the bottle clears the end of the rod. An internal 7/16"- 14 thread screws onto a hollow bolt passing through the expanding seal. The rod stock came with three cams. The 0.250" thick cams were turned to 0.375" dia from brass rod stock, with eccentric holes drilled to provide about 0.050" lift in a 90-degree turn. They're brazed onto an iron wire handle bent to fit around the wood cube, with a launch string passing through a hole in the can lid and plywood.



Bottle Rocket Launcher Assembly

The steel block holding the cams is also drilled and tapped to hold the 7/16"- 14 bolt passing through the middle. The cams are spaced 0.950" on centers, which puts about half of each one under the washer supporting the seal. A pair of 1 -inch-long 4-40 screws tapped into the block support the cams.

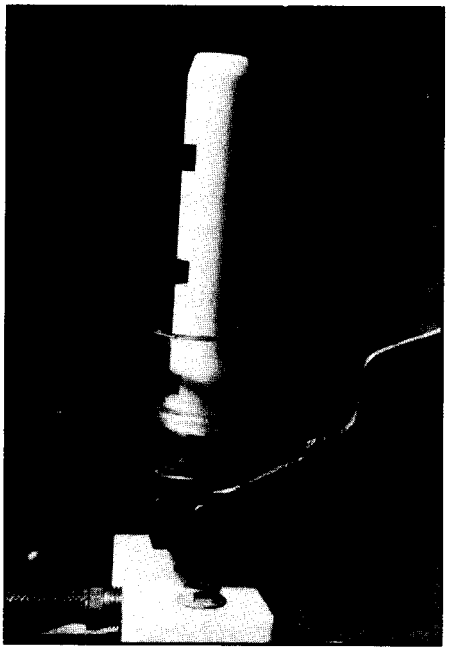
The central 7/16"- 14 bolt has an axial 0.250" hole drilled through its length to allow air into the bottle. The head has a l/8"-NPT brass fitting brazed to it, which screws into a tapped hole in the lower nylon block. The block simply connects the bolt to the air supply through a standard 1/4"-NPT hose barb adapter screwed into its side. There are two 1/4"-20 clearance holes to bolt the block securely to the wood cube on the base. The gantry is mounted on the

The seal is compressed between the nylon guide rod and a steel washer driven by a pair of brass notches milled across it, so I didn't have to drill any holes to get the air from the internal thread to the bottle.

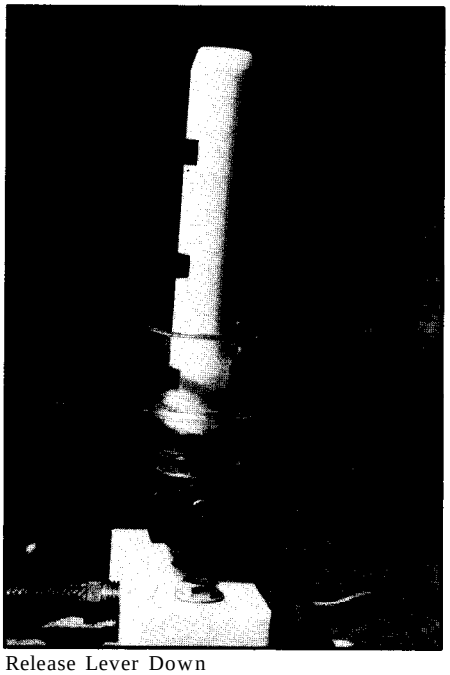
The expanding seal is made from a 1/2" thick silicone rubber shock mount, hand cut to about 0.875" dia and sanded down to 0.855" on the lathe. The ID is a tight fit on the body diameter of the 7/16"-14 bolt, which has the threads turned off behind the seal. The material is critical to the success of the whole launcher, but I'll admit that I simply stumbled across it after trying a lot of things that didn't work.

driveway sealer can, with a 3/4" plywood circle as a stiffener.

The gantry is made from three aluminum bookshelf brackets, mounted to top and bottom plates cut from 0.100" aluminum sheet with 4.5" diameter circles cut to clear the bottles. The fittings holding it all together were machined from 1" nylon plate.



Release Lever Up



Release Lever Down

FIRMWARE FURNACE

Using the ZBM PC Joystick Port

by Ed Nisley

he IBM PC started out with 16K bytes of RAM, a cassette tape recorder, and a joystick port. In the ensuing seven years the RAM grew to 16 megabytes and the cassette port vanished into history, but the joystick port remains a mystery to most programmers.

Enough is enough! It is now time to reveal the One True Way to read the joystick port and explain why the methods you may have seen elsewhere are, well, less than adequate for Real Firmware.

From the preceding Firmware Furnace columns you may have gotten the impression that all firmware is burned into EPROMs next to 8031 processors, but early on in the first column I staked a claim to my code that snuggles right up next :o the hardware, regardless of the ›rocessor. The fact that this col-ımn will use IBM PCs and a shudder!) high level language hould come as only a mild surprise.

The first step in any firmware ›roject is understanding the hard-vare, so I will start with an expla-ıation of the IBM PC's joystick ›ort, which is called the Game Control Adapter in Official .BMese.

Figure 1 shows the entire IBM Game Control Adapter schematic diagram. There are four main areas: address decoding, four switch inputs, four joystick inputs, and the buffer on the PC's data bus. Those four functions involve only five ICs, so there isn't too much complexity hidden on the board.

The Game Adapter has two different types of inputs: switches

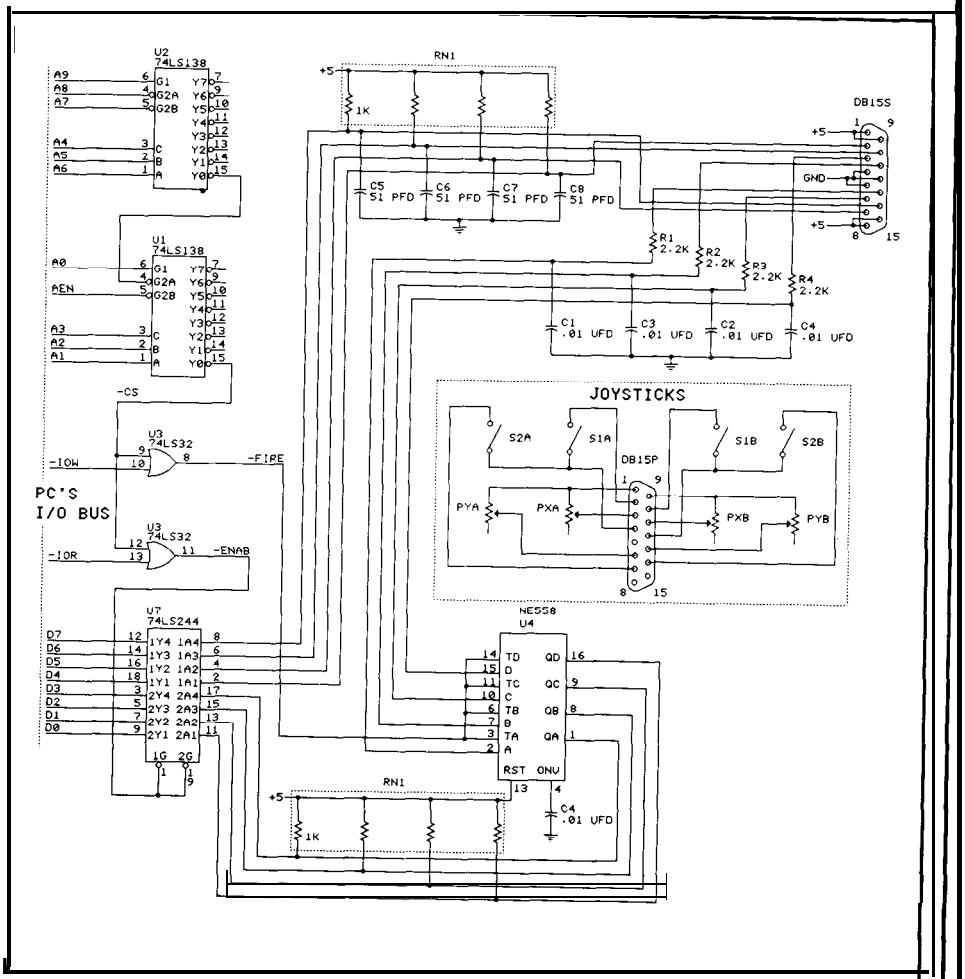


Figure 1 - IBM Game Control Adapter

and resistances. The four switches are normally-open connections to ground, usually simple pushbuttons. The four resistances are generally potentiometers attached to a pair of two-axis joysticks. With a little bit of trickery you can replace the switches and potentiometers with other circuits, but I'll save that for a later column.

The switch inputs are "real time" connections to the processor; there's no matrix scanning to interfere with the timings. The resistances, on the other hand, control pulse widths that must be measured by the program. The precision and accuracy of that measurement are what this column is all about.

The Game Adapter responds to only one I/O address, hex 201. In assembly language terms, an OUT to that address starts the timers and an IN returns the status of the timers and the switches. That's the entire program interface, so you can see why the hardware is so simple.

The pair of 74LS138 demultiplexers and the 74LS32 gates provide the address decoding. The 'LS138s produce an output whenever the processor executes an IN or OUT instruction to address 201h. The 'LS32 gates further refine the address decoding so that an OUT instruction produces a pulse on the -FIRE line and an IN instruction pulses -ENAB.

The -ENAB line activates the 74LS244 buffer which drives the PC's data bus. Because the Game Adapter's -ENAB line is active only during IN instructions from port 20 1 h, the 'LS244 provides isolation between the Game Adapter and the bus whenever the Processor isn't reading the board.

Data bus bits D7 through D4 come from the four switches connected to pins 2, 7, 10, and 14 on the Game Adapter socket. The switches are normally open, connecting the pins to ground only when pressed. Because those pins are pulled up by 1K resistors, the processor will read a "0" bit only when a button is pressed and a "1" bit at all other times. The 51 pF capacitors provide a small amount of noise filtering, probably to ensure that the board passes FCC testing.

As you might expect from the name, the -FIRE line triggers something on the NE558 chip. It turns out that the NE558 is actually four timers similar to the venerable NE555 on a single IC. The -FIRE line starts all four timers whenever the processor executes an OUT to 201 h.

Notice that the 'LS244 buffer is a one-way connection: the PC cannot write any data into the Game Adapter. An OUT pulses the -FIRE line, which triggers the NE558 timers without affecting anything else. This is truly an example of a "read only" board!

Figure 2 shows what happens when the -FIRE line triggers the four timers in the NE558. The output lines go to a "1" state immediately after the trigger pulse and return to zero after a time delay set by the resistance and capacitance. Because all four timers use 0.01 uF capacitors the pulse widths are controlled by the resistances connected to pins 3, 6, 11, and 13.

According to the IBM Technical Reference the pulse width is given by this formula:

pulse width T = 24.2 + (0.011*R) microseconds

where R is in ohms. The normal range for R is zero to 100K ohms, so T varies between 24.2 and 1124 us. Component tolerances may cause some differences, but the orders of magnitude are certainly correct.

The fact that the pulse width is directly proportional to the resistance is particularly important to the software. If the joystick potentiometers produce a resistance proportional to the joystick angle (as nearly all of them do), the pulse width is a direct indication of the angle. There's no need to go through complex calculations to linearize the output!

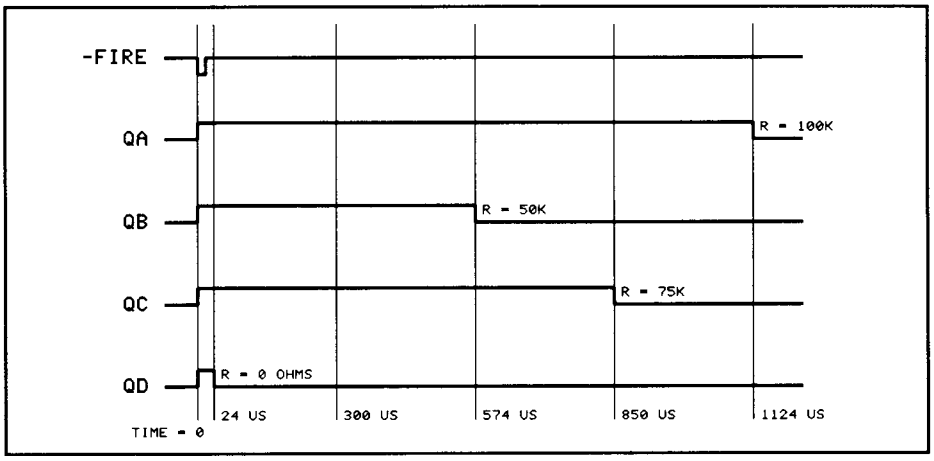


Figure 2 - NE558 Timing

Because a single IN instruction returns the values for all four joystick axes (and the switches) at once, the program must read the port repeatedly until all four joystick timer bits are zero. The time at which each bit changed from a "1" to a "0" is proportional to the joystick angle, so the program must identify the first "0" and record the corresponding time. It's also a good idea to have a deadman time so that the program doesn't hang up in a loop if one of the bits is stuck on.

Listing 1 - Program Loop Method

```
/* The wrong way to read the joystick port...    */

/*-------------------------------------------------- */
/* Program loop version of joystick port read routine   */

#define MAXLOOPS 1000

int  ReadJoyStick(int  JoyPort,unsigned  int  *pAxes) {

  unsigned  int  loops:            /* loop counter          */
  int  portval;                    /* current port value */
  int  lastport;                   /* previous port value ● */
  int  delta:                      /* changed axis bits     */

/*--- set up for loop                                */

  loops = 1;
  outp(JoyPort,0);                 /* trigger the hardware */
  portval = inp(JoyPort);
  lastport = portval;

/*--- run loop until all axes done or max loops run out */

  do {
     portval = 0x0f & inp(JoyPort);     /* get current bits */
     delta = portval ^ lastport;        /* any changes? */

     if (delta & 0x01)                  /* set current time     */
        *(pAxes+0) = loops:
     if (delta & 0x02)
        *(pAxes+l) = loops;
     if (delta & 0x04)
        *(pAxes+2) = loops:
     if (delta & 0x08)
        *(pAxes+3) = loops:

     lastport = portval;                /* remember axis bits   */

  } while ((loops++ < MAXLOOPS) &&
           (lastport != 0x00));

  return (0x0f & (~inp(JoyPort)>>4));   /* figure out switches */
}
```

Code the Wrong Way

Listing 1 shows a C function that reads the joystick port in the "traditional" way. I used C rather than assembler so that the method isa little more obvious. The code iswritten in Microsoft C 5.1, but you should have little trouble porting it to your favorite flavor of C

Although I described the Game A.dapter's operation with assembly language IN and OUT instructions, C also provides direct port I/O operations. The statement

```
outp(JoyPort,0);
```

issues the OUT instruction to trigger the NE558 timer, with a zero data byte that the board ignores. The statement

```
portval = inp(JoyPort);
```

executes an IN that sets the variable "portval" to the current state of the Game Adapter lines. In both cases JoyPort contains the port address.

The code computes the changes from the previous port value by exclusive-ORing the two values to produce a "1" bit whenever two corresponding bits change. The "if" statements examine each of the four timer bit positions for those "1" bits. Because more than one axis can time out at the same moment all four ifs must be executed every time through the loop.

When an axis changes, the code stores the current value of the loop counter in the output array. Because each pass through the loop takes the same amount of time the loop counter value is a direct measure of the elapsed time from the moment of triggering.

The code in Listing 1 is a subroutine that must be combined with the driver program in Listing 3 to produce a complete program. The resulting program is called JOYSLOW.C (JOYSLOW.EXE as the executable file).

Although the logic in JOYSLOW is correct, it is unworkably slow. Using Microsoft C 5.1 with all optimizations turned on, the maximum value for any axis is about 25 counts when run on an 8-MHz AT. Knowing that the joysticks time out in about 1 .1 ms you can compute that each iteration through the measuring loop takes 44 microseconds. For software this is OK, but it's not nearly good enough for firmware!

You have probably seen versions of Listing 1 written in assembly language, perhaps with the note that this optimizes the speed of the code. While that's true, it does not remove the fundamental problem inherent in a program timing loop: the effect of the processor clock speed.

The IBM PC

The IBM PC family includes processors ranging from the original 4.77-MHz 8088 to the latest 25-MHz 80386 screamers, covering more than an order of magnitude in performance. The code in Listing 1 will return values that vary from about 5 to over 50 depending on which PC you use, despite the fact that the joystick is at the same angle. Obviously, any program using this code must decide just what a particular value means before using it.

Fortunately, the PC provides

hardware to solve that problem, in the form of an Intel 8253 Programmable Interval Timer. This should sound familiar if you were paying attention to the Bottle Rocket article in the March/April issue of INK because we used it to time the rocket launches. The Firmware Furnace column in the last issue described how the **ImageWise** uses an 8254, which is an improved version of the 8253, to generate precision pulses. You may want to refer to those articles for more programming hints.

The PC's 8253 contains three separate timers, each driven by a clock derived from a **14.3-MHz** oscillator. Unlike the CPU clock, this oscillator is the same on all PCs, so each timer "tick" measures the same amount of time on any PC. The **14.3-MHz** signal is divided by 12 (first by six, then by two) to produce a 1.1 **9-MHz** signal with a period of 838 ns. The timers can measure up to 54.9 ms by counting 64K ticks before wrapping back to zero.

Timer 0 is dedicated to the BIOS real-time clock; although we used it in the Bottle Rocket project, it is best not to fiddle with "mission-critical" hardware without a very good reason. Timer 1 provides the RAM refresh timing in PCs and is missing in **PS/2** systems, so it is not a good choice either. Timer 2 normally controls the frequency of the tone fed to the PC's speaker (and the long-gone cassette port!), so there's no problem using it for a different purpose.

Figure 3 shows the circuitry involved in this discussion. Many connections are not shown because they're not relevant here; you can refer to the schematic in the IBM Technical Reference Manual for more details.

The BIOS sets Timer 2 to Mode 3, which produces a square wave output. The input clock frequency is divided by the value loaded into the timer, once for the high half of the wave and again for the low half. The speaker control circuitry provides a bit to enable the speaker and another to turn Timer 2 on and off; these bits are located in an Intel 8255 Programmable Peripheral Interface chip.

The corresponding IBM AT circuitry is similar, but replaces the 8255 with discrete **ICs** and uses an 8254 in place of the 8253. A quick look in the Tech Ref will show you that there are no differences visible to the code in this column, so the programs work unchanged on PCs or **ATs.**

Measuring time intervals instead of producing square waves requires changing Timer 2 to Mode

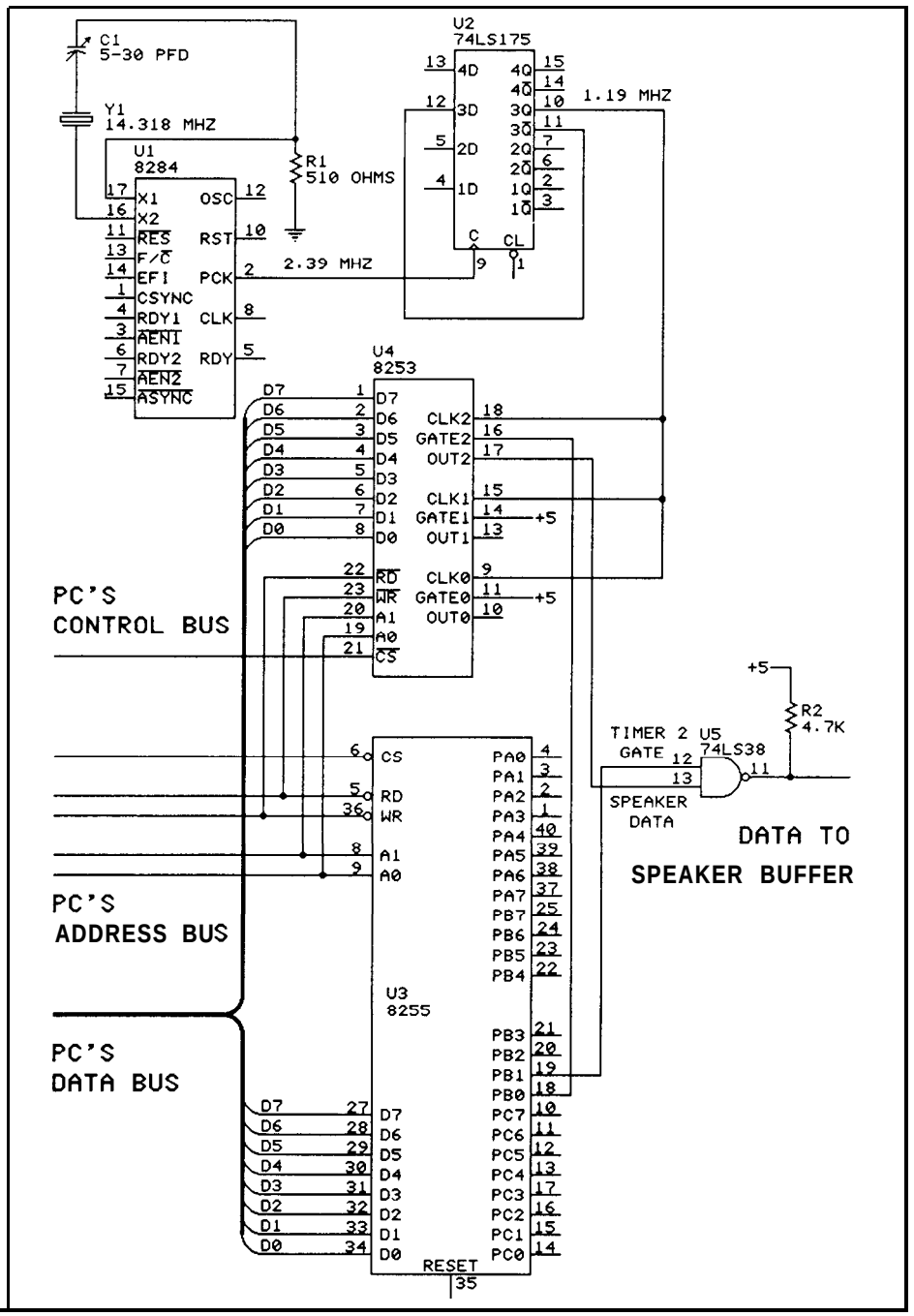


Figure 3 - IBM PC 8253 Timer Circuitry

Listing 2 - Hardware Timer Method

```
                PAGE    79,132
;
; Joystick port read routine
; Samples joystick port, returns timeout value for all axes
; Uses 8253 Timer 2 counter for CPU speed-independent timing
;
; C declaration:
;  extern int ReadJoyStick(int JoyPort,unsigned int near *pAxes);
;
; return value is binary equivalent of switches in bits 0:3
;               0 = open
;               1 = pressed
; JoyPort contains the joystick port address
; pAxes points to an array of FOUR unsigned integers to hold the
;               results must be FAR pointer to get-correct segment
;               address results are counts of 838 ns input clock
;
;IBM Game Adapter timing is 24.2 us + (0.011 us)*resistance
;  joystick resistance can range from 0 to about 100K,
;  so timing is between 24.2 us and 1.12 ms
;
; This code requires Microsoft MASM 5.0 or 5.1


;----------------
; Hocus pocus to start up the assembler

                DOSSEG
                .MODEL    MEDIUM

                .CODE

;----------------
; Constants

timeout         EQU       11900       ;10 ms of counts at 838 ns/count

I8255B          EQU       0061H       ; 8255 port B address

I8253C          EQU       0043H       ; 8253 command register
I8253T2         EQU       0042H       ; 8253 Timer 2 register

timer2          EQU       10000000b   ; bits 7-6 = timer number

latchcmd        EQU       00000000b   ; bits 7-6 = timer #
mode2cmd        EQU       00110100b   ; 76=timer, 54=r/w, 31=mode  O=bin

PBbits          RECORD    pbbz:6,spkrdata:1,spkrgate:1; I8255 port B
;----------------
; Stack  layout

args            STRUC
DIsave          DW        0                   ; saved DI
SIsave          DW        0                   ; saved SI
BPsave          DW        0                   ; saved BP
IPret           DW        0                   ; return address, offset
CSret           DW        0                   ;  ... segment
JoyPort         DW        0                   ; port address
pAxes           DW        0                   ; address of result array
args            ENDS

;----------------
; Output record structure
; This is an array of four integers at pArgs

results         STRUC
X1              DW        0
Y1              DW        0
x2              DW        0
Y2              DW        0
results         ENDS

                PAGE
```

(continued on page 43)

2, which is normally used to generate an output pulse after counting a specific number of input clock periods. The value read back from Timer 2 during counting indicates the number of periods since the timer started. In this application the output signal isn't needed, so the code simply disables the speaker circuitry using one of the bits in the 8255.

It is never safe to assume any particular setup for hardware that is not under your program's direct control, so Timer 2 must be set up every time it is needed. Consider what happens if a resident program should pop up and produce a "beep" when the joystick routine is using the same timer to measure the joystick. Obviously something bad is going to happen, but it's hard to tell which program will come out the worse for wear.

Because the longest joystick pulse is only about one millisecond long, the routine in Listing 2 takes the simple precaution of disabling all processor interrupts just before it sets up Timer 2. The interrupts are enabled after the measurements are complete. A watchdog counter (held in SI) ensures that the code will terminate even if there is no response from the Game Adapter, so there is no risk of locking up the PC.

The timers begin counting from the preset value and count downward toward zero. By loading FFFFh and complementing the value read from the timer it is easy enough to make them appear to count upward. The resulting value is a direct measure of the number of 838-ns counts since the timer started running.

The inner logic of Listing 2 is similar to that in Listing 1, with the difference that the value stored in the output array is the timer value instead of the loop count. The tests are much faster in assembly language, but the source code required

is much bulkier.

The driver routine in Listing 3 will also handle the assembly code from Listing 2. You must compile, assemble, and link the modules together; MAKEFILE.MAK in the downloadable files contains the commands I used to handle the process.

Figure 4 presents the results of running JOYFAST.EXE on three different processors. The maximum value is over 1000 counts in all cases, which is significantly better than the paltry 25 produced by the C-language loop. More important, the value is essentially independent of processor clock speed.

The "Increment" column in Figure 4 shows successive values starting at the minimum count, produced by teasing the joystick in small steps. The PC's step size is about 20 counts, while the two ATs step by four or five counts. Converting these counts into seconds gives the time for one pass through the assembly language loop based on the processor speed: a PC takes 16.8 us versus an AT's 3.5 us.

For comparison, recall that the C language code takes about 44 microseconds on an AT. Although the code is not quite identical, the performance improves by about a factor of ten in assembler language. That explains why nearly all firmware is written in assembler: it's not easy, but it's essential!

The loop time determines the number of different counts that can occur. For example, the PC makes only about 64 loops before all the axis timers finish (1250 counts divided by 20 counts/loop), while the AT will have about 300 loops (1200/4). The effect of this is

(Listing 2 - continued from page 42)

```
;----------------
; Force I/O recovery time on ATs

punt            MACRO
                LOCAL   L1
                JMP     SHORT L1                ; flush prefetch queue
L1              LABEL   NEAR
                ENDM

                PAGE
;----------------
; The Main Event

_ReadJoyStick PROC      FAR
                PUBLIC  _ReadJoyStick


                PUSH    BP                      ; save bystanders
                PUSH    SI
                PUSH    DI

                MOV     BP,SP                   ; set up frame pointer

;----------------
; Initializations

;--- get pointer to results and clear them to zero

                MOV     DI,[BP].pAxes
                MOV     [DI].X1,0
                MOV     [DI].Y1,0
                MOV     [DI].X2,0
                MOV     [DI].Y2,0

;--- get output port address

                MOV     DX,[BP].joyport

;--- set up timeout counter

                MOV     SI,0
;----------
; Set up Timer 2
; Sets Mode 2 for 64K counts, one per 838 ns input clock
; Sets FFFF reload value
; Clears "speaker enable" bit to prevent embarrassing noises
; Does not start timer, leaves "gate speaker" bit low
; Interrupts are OFF until the final axis times out in about a
   millisecond
;  ... or the timeout limit hits in about 10 ms

                CL1                     ; interrupts are OFF

                IN      AL, I8255B      ; get existing port B bits
                punt
```

(continued on page 44)

Clock Speed & Processor	Joystick Position	Count Value	count Increments
4.77 MHz IBM PC	minimum	65	65 84 104
	center	750	
	maximum	1250	
8 MHz IBM AT	minimum	38	38 42 47
	center	610	
	maximum	1200	
10 MHz CCAT	minimum	38	38 42 47
	center	8 0 0	
	maximum	1400	

Figure 4 - JOYFAST.EXE **performance tests**

(Lirting 2 - continuedfrompage 43)

```
                AND     AL,NOT MASK spkrgate ; turn off timer gate
                AND     AL,NOT MASK spkrdata ; turn off speaker
                OUT     I8255B,AL
                punt

                MOV     AL,mode2cmd+timer2
                OUT     I8253C,AL
                punt

                MOV     AL,0FFH          ; set reload value to FFFF
                OUT     I8253T2,AL
                punt
                OUT     I8253T2,AL
                punt


;----------------
; Start the joystick hardware and Timer

                MOV     CL,OFH           ; axis timeout mask

                IN      AL,I8255B        ; get existing port B bits
                punt
                OR      AL,MASK spkrgate ; turn on timer gate
                OUT     I8255B,AL        ; set bits out again
                punt

                OUT     DX,AL          ; trigger the joystick timers
                punt

;----------------
; Run the timing loop until all axes time out or we give up
; Axis bits are '1' until timed out

looper          LABEL   NEAR

                IN      AL,DX       ; get bits
                NOT     AL          ; flip so 0 = run, 1 = timeout
                TEST    AL,CL       ; any new axis timeouts?
                JNZ     gotone      ;  any 1 -> yes!

;--- decide if it's time to check for a timeout yet
;    we do this by entering axis set loop with no axes selected

                INC     SI
                CMP     SI,0
                JNZ     looper      ; nope

;--- got an axis timeout, grab Timer2

gotone          LABEL   NEAR

                MOV     BX,AX       ; save the new axis timeout bits

                MOV     AL,latchcmd OR timer2 ; latch current value
                OUT     I8253C,AL
                punt

                IN      AL,I8253T2      ; get LSB
                punt
                MOV     AH,AL
                IN      AL,I8253T2      ; get MSB

                XCHG    AH,AL           ; swap 'em around
                NOT     AX              ; flip bits to count upward

;--- save value in output array for each newly changed axis
;    several axes can change at once, so we've got to check 'em all
;    if we're here for a time check, nothing gets stored

                MOV     BH,BL           ; save raw axis bits again
                AND     BL,CL           ; isolate new bits
```

(continued on page45)

simple: a given joystick position will return the same value on any processor, but faster processors will be able to distinguish smaller position changes.

The values shown in Figure 4 used the same joysticks on each machine, so the different maximum counts show the component tolerances in the three Game Adapter boards. The IBM PC and the IBM AT have boards with precision capacitors, so their values agree to within about 4%. The board in the CCAT uses ordinary ceramic capacitors and the results differ from the other two boards by about 12%.

The Bottom Line

This type of code has application beyond the simple matter of reading joystick ports. Any program can measure brief time intervals accurately using the PC's timing facilities, with results independent of the processor clock speed.

Of course, these programs will not be burned into an EPROM, but I call the code firmware because it's so closely tied to the hardware. After all, you can't write the code without poring over the schematic!

All of the programs in this column will work on your PC if you've got a joystick port. You'll need Microsoft C 5.0/5.1 or QuickC 1.00/1.01 for the C code and MASM 5.0/5.1 to assemble the source code if you decide to make any changes. I used the Medium model for the C and assembler routines because QuickC depends on it, but you can change it to suit your needs. Both the source and executable files are available for downloading from the Circuit Celiar BBS; see the masthead for phone numbers and modem settings.

(Listing 2 continued from page 44)

```
                TEST    BL,01H
                JZ      not1
                MOV     [DI].X1,AX
not1            LABEL   NEAR

                TEST    BL,02H
                JZ      not2
                MOV     [DI]+Y1,AX
not2            LABEL   NEAR
                TEST    BL,04H
                JZ      not3
                MOV     [DI]+X2,AX
not3            LABEL   NEAR

                TEST    BL,08H
                JZ      not4
                MOV     [DI]+Y2,AX
not4            LABEL   NEAR


;--- update axis mask: if all done, bail out
;    if some axes left, decide if current time exceeds limit

                NOT     BH              ; timed out axis = 0 again
                AND     BH,OFH          ; remove unused bits
                MOV     CL,BH           ; reset mask
                JZ      goback          ; if zero, all axes are done

                CMP     AX,timeout      ; decide if timed out yet
                JA      goback

                JMP     SHORT looper    ; and keep on checking



;----------------
; Sample buttons and set return value
; The input value is "1" when buttons are not pressed, so we flip
  it over
; Interrupts are OK at this point because the critical timing's
  over
goback          LABEL   NEAR

                STI

                IN      AL,DX
                NOT     AL
                MOV     AH,0

                MOV     CL,4                    ; align in low bits
                SHR     AX,CL



;----------------
; Termination  cleanup

                POP     DI                      ; restore  bystanders
                POP     SI
                POP     BP

                RET

_ReadJoyStick   ENDP


;----------------
; hocus pocus to turn off the assembler

                END
```

Listing 3 - Driver Code for Joystick Routines

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>

#define JOYPORT 0x201           /* joystick address */
#define MAXAXIS 3                       /* last axis to use */

unsigned int Axes[MAXAXIS+1];   /* joystick counts */
int Buttons:                            /* buttons = 0:3 */


/*---------------------------------------------------------- */
/* This preprocessor if statement controls how the program */
/*  reads the port:                                         */
/*                                                          */
/*    use 1 with the assembly code version and link with    */
/*      Listing 2                                           */
/*    use 0 with the C program loop version and include     */
/*      Listing 1                                           */

#if 1
extern int ReadJoyStick(int JoyPort,unsigned int near *pAxes);
Yelse
        int ReadJoyStick(int JoyPort,unsigned int near *pAxes);
#endif


/*---------------------------------------------------------- */
/* The Main Loop                                            */

int main(void) {

int axis:

  while (!kbhit()) (

     Buttons  =  ReadJoyStick(JOYPORT,Axes);

     printf("Axis values: ");
     for (axis=0; axis<=MAXAXIS; axis++) {
        printf("%5u ",Axes[axis]);
     }

     printf("Buttons: %1.1X",Buttons);

     printf("\r");
  }

  return(0):
```

WRITE FOR INK!

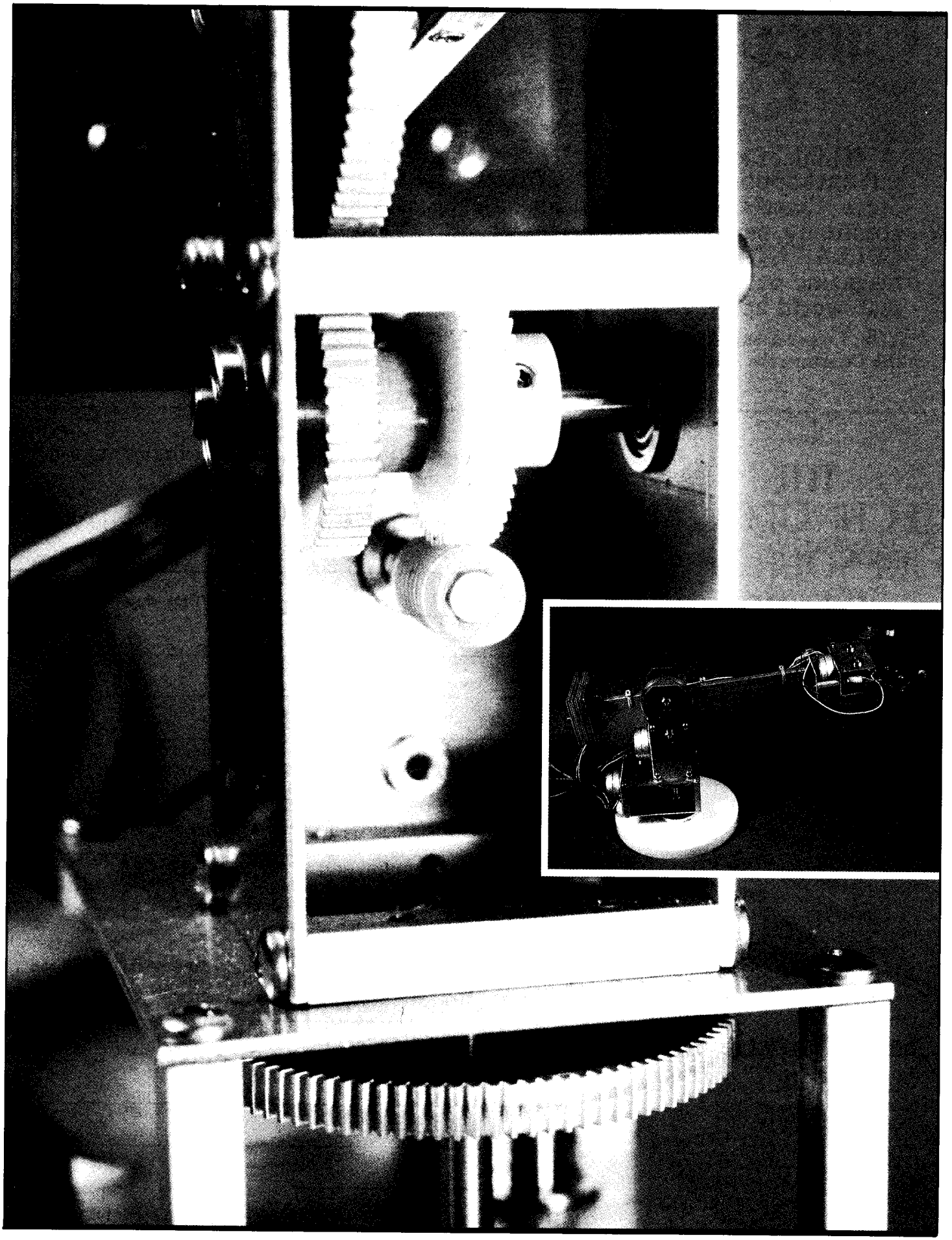
Writing technical articles may not make you rich and famous but it might be just the incentive to finish that 100-MIPS computer you started last summer. Or, if your expertise is software, perhaps it's time you presented your talents to the world.

Unlike most narrowly specialized publications, Circuit Cellar INK's charter is to cover a wide variety of hardware and software technology and ideas.

Send your project outline to:

Editorial Director
Circuit Cellar INK
P.O. Box 772
Vernon, CT 06066

or contact the Circuit Cellar BBS at (203) 871-1988.



Stepping Out

A Robot Arm that Demonstrates Microprocessor Control of Stepper Motors

by Tim McDonough & Dennis Grim

Controlling a process, whether it be an automated factory or a simple robot, can be broken down into two very broad steps: getting input from one or more sources, and making something happen in response to that input. Since the best education source is always experience, we decided to build a simple robotic arm as a learning vehicle which would allow us to experiment directly with these concepts.

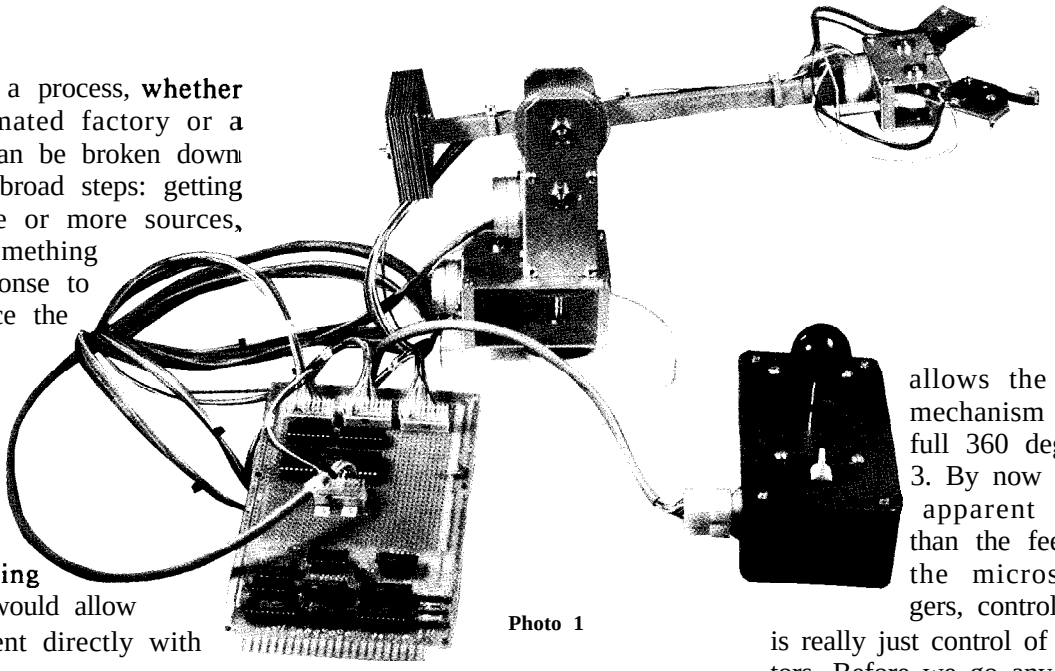


Photo 1

This project can be broken into three sections, based on the technology used in each. The first is the arm itself, a simple two-axis aluminum affair that uses unipolar stepper motors for motion. The second is the controller electronics for the arm. In our case, we used a BCC52 BASIC controller, but our techniques should transfer easily to any controller you want to use. Finally, there's the software that makes it all work together. The BCC52 gave us a choice of programming in BASIC or 8052 assembler, and we found that each language has its advantages.

The final configuration, Photo 1, is a robotic system that's absolutely useless for building Buicks or fabricating circuit boards, but is perfectly tailored for letting you directly experience microprocessor-based control.

The computer-controlled arm used for this project is approximately 7.5 inches tall and has an arm length of about 10.5 inches. You operate the arm via a small control pod with a joystick to control the two major axes of arm movement and a rocker switch to control the opening and closing of the fingers. The fingers on the hand are two common microswitches. They give the arm a crude sense of touch by providing feedback to the controlling software whenever the hand has gripped something.

Three stepper motors power the arm. The first is used to operate the mechanical fingers, Photo 2. Another is used to raise and lower the arm through an arc of about 45 degrees, Photo 3. The third motor allows the entire arm mechanism to rotate a full 360 degrees, Photo 3. By now it should be apparent that, other than the feedback from the microswitch fingers, control of the arm is really just control of stepper motors. Before we go any farther, you should know some stepper motor fundamentals.

A stepper motor is a particular type of motor ideally suited for microprocessor control. It converts pulses of electrical current into discrete rotational movements. There are two varieties of stepper motors: bipolar and unipolar. A bipolar motor uses a two-coil construction and requires that the driving circuitry be able to reverse the direction of current flow in the windings. A unipolar motor has a required to drive either type of motor is essentially the same. Within the two basic types of stepper motors, a broad variety of torque ratings and step graduations are available. The type of motor chosen for a particular application will depend primarily on the amount of torque required for the application and the number of "steps" in which the

motor completes one revolution.

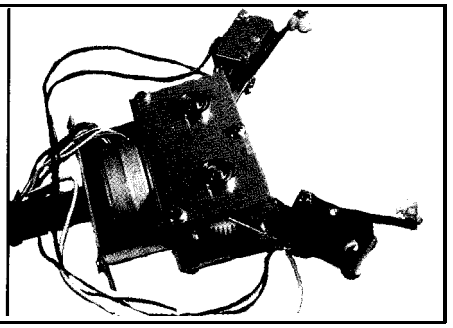


Photo 2 - Microswitches serve as fingers with "tactile" input

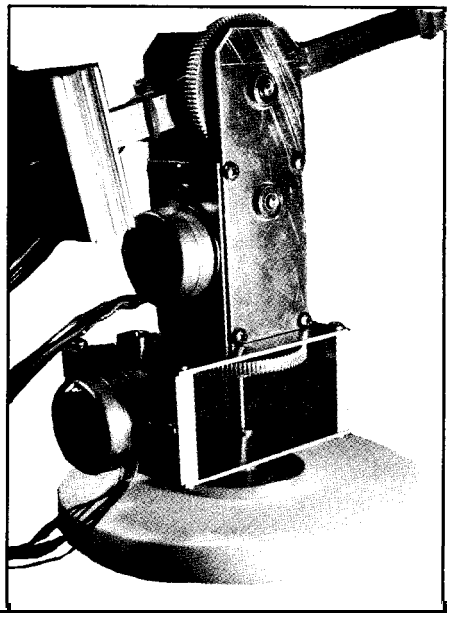


Photo3 - Stepper motors raise, lower and rotate the arm in precise increments

The Brains of the Operation

The mechanical arm and control pod both operate under the watchful eye of a Micromint BCC52 computer and a BCC-bus-compatible interface board, Photo 4. The combination can interface up to four stepper motors to the bus and sense the condition of the control pod switches. The BCC52 uses an Intel 8052AH-BASIC microprocessor and was originally presented in the August 1985 issue of BYTE magazine.

The stepper motor interface circuitry is a fairly simple affair. The interface was constructed on a BCC55 prototyping card (Figure 1) which contains all of the address decoding and buffering required to interface additional circuitry to the BCC bus. The stepper motor interface circuit diagram (Figure 2) shows only those portions of the circuitry which were added to the BCC55 board.

An 8255AC-5 Programmable Peripheral Interface is the core of the stepper motor/control pod interface card. The 8255 contains three 8-bit ports which we configured as two 8-bit output ports and one 8-bit input port. Ports A and B of the 8255 are connected to a pair of Sprague ULN2803As. The ULN2803 is a Darlington transistor package which is used to beef up the output of the 8255 to allow it to drive the stepper motors. Take note that in this particular application the 12-volt supply required by the ULN2803s is taken from the bus. When using larger stepper motors you may need to provide an alternate source of 12 VDC to avoid overloading the power supply or burning the traces on the motherboard.

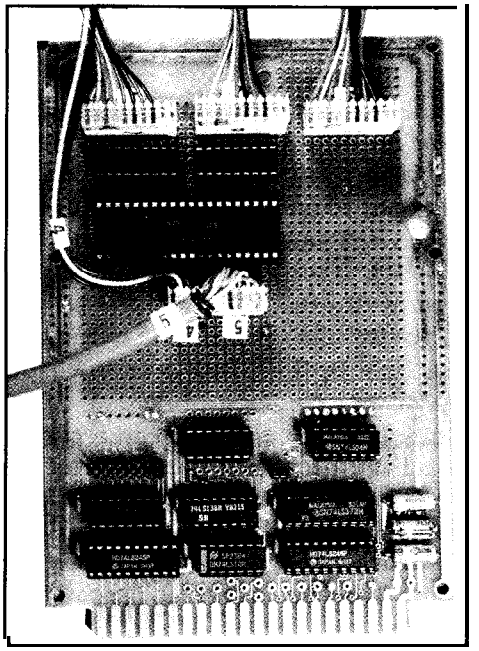


Photo 4 - The stepper motor driver circuitry shown in Figure 2 is mounted on a BCC55 Proto Board.

The stepper motor interface is strapped for a base address of EOOOH by positioning the jumper on JP2 of the BCC55. Pin 8 of J22 is tied to the chip-select pin of the 8255 PPI and pins 8 and 9 of the 8255 are tied to the buffered address lines A0 and Al. The result places the I/O ports and control port of the 8255 at addresses E800H through E803H.

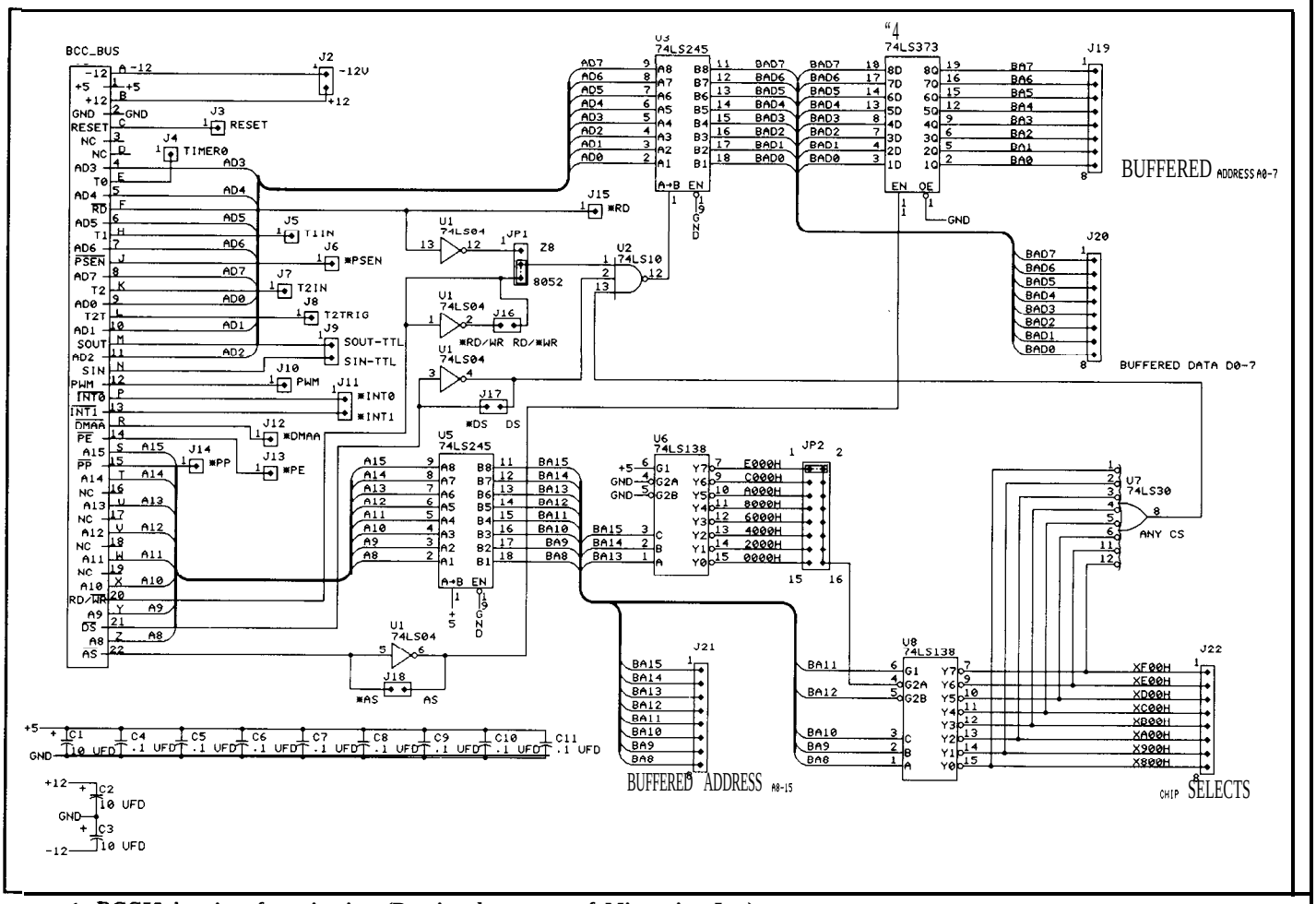
The schematic diagram of the control pod is shown in Figure 2. The control pod switches are connected to port C of the 8255, which is used as an 8-bit input port. The common side of each switch is tied to ground and the switched side is connected to one of six inputs. The remaining two input lines are used for the microswitches on the fingers. SIP1 contains 4.7K pull-up resistors so that in the absence of any switch closures all input pins are held high.

We connected each of the output pins on ports A and B of the 8255 to the TTL/CMOS-compatible base of a Darlington transistor driver in one of the ULN2803A transistor arrays. These arrays can supply as much as 500 mA of output current with the appropriate power supply and wiring.

One end of each stepper motor winding is connected to an output pin of a Darlington package. The common ends of the windings are connected to the 12-VDC supply. Anytime a particular output transistor is turned on by the output of the 8255, the switch provides a path to ground and the coil is energized.

The Software Dilemma

The MCS BASIC-52 available on the BCC52 is a powerful process-control programming language. Unfortunately, stepper motors can be pretty demanding when it comes to the optimum step rate to achieve smooth motion and maximum torque. We first tried



gure 1 - BCC55-bus interface circuitry (Reprinted courtesy of Micromint, Inc.)

sing BASIC-52 as the control language. We wrote the BASIC-52 ersion to test the arm in its early ages of development, but found at it was too slow to work well ith our stepper motors. Perhaps a ompiled BASIC would be fast tough to avoid delving into the nards of the processor but on our BCC52 system assembly code was the only way to go.

Listing 1 is the assembly language program that controls the arm. The first thing the software does is to initialize the 8052 and set up the 8255 on the arm's control board. The Program Status Word (PSW) is initialized to select register bank 3 on the 8052. Bank 3 was osen due to the availability of the n-board BASIC. BASIC-52 uses egister banks 1 and 2 as well as the majority of the 8052's internal memory. By restricting the current software to locations and registers which are not used by BASIC, we gave ourselves room for future expansion of the arm system software.

Next, registers RO through R2, which will be used as pointers into the motor phase table (end of Listing 2), are initialized. Their initial values are relatively unimportant as long as they point to some valid table offset, either 0, 1, 2, or 3. Once the register bank has been selected and the table pointers have been initialized, the address of the 8255 control register is loaded into the 8052 data pointer (DPTR). The value 89H is transferred into the accumulator and written to the control register to set up the 8255 on the interface board for output on ports A and B (the stepper motors) and input on port C (the control pod). The remainder of the program is a loop which tests the control pod switches, sets up the phase data to be output, and steps the motors.

Before the arm can move it has to know what the operator wants it to do. The first thing the main program loop does is read the data byte from port C of the controller board's 8255 and store it in internal memory location 20H. This memory location was chosen for two reasons. First, it is one of the few locations unused by BASIC-52 (remember, we're planning for the future), and it is one of several bit-addressable locations on the 8052.

The procedure for getting input from the operator and passing it to the individual stepper is essen-

tially the same for each motor, so only the tests for motor A will be explained. First, the phase pointer is transferred from register 0 to the accumulator. If bit 0 is a "1" then the switch on the pod was not closed and the program jumps ahead to *LBLZ* to test bit 1. If bit 0 was a "0" then the corresponding pod switch was closed, so the phase pointer is incremented (the motor needs to be turned clockwise) and the program jumps immediately to LBL2. The program doesn't have to check bit 1 if bit O's switch is closed since the physical construction of the joystick prevents opposing switches from being closed at the same time.

If bit 1 is a "0" when tested then the table pointer is decremented (the motor needs to be turned counter- clockwise). The only other possibility is that neither switch was closed. In this case the program jumps ahead to LBL3 (the next motor) and the phase pointer remains unchanged.

If the motor A phase pointer was changed in either direction, the new value in the accumulator is ANDed with the number 3. Since we only use the lower four bits of information to operate the stepper motor, the value of the phase pointer will always be a valid number by performing this operation. Regardless of which way the pointer moves, this logical AND will ensure that things keep running smoothly.

Once all of the control pod bits are tested and the phase pointers are updated, it's time to actually move the motors. Before the writing begins, however, the data which will be written to the output ports needs a bit of preparation. First, DPTR is loaded with the base address of the phase table. Next the offset into the table for motor A is added and the actual table value is copied from the table and stored in a scratch register, R3. Next, we get the motor B offset (the base address in DPTR is unchanged) and copy the table value for motor B into the accumulator. The high and low nibbles of the accumulator are then SWAPped. The data for motor B is now in the high nibble and once we perform a logical OR with motor A's data (which is tucked away in R3), we can then save the entire byte containing both motor A's and motor B's data back in R3. Motor C requires less code since our robot arm uses only three motors and motor C is the only one connected to port B. Its offset is retrieved from register R3 and we copy the proper data value from the table and place it in R4, another scratch

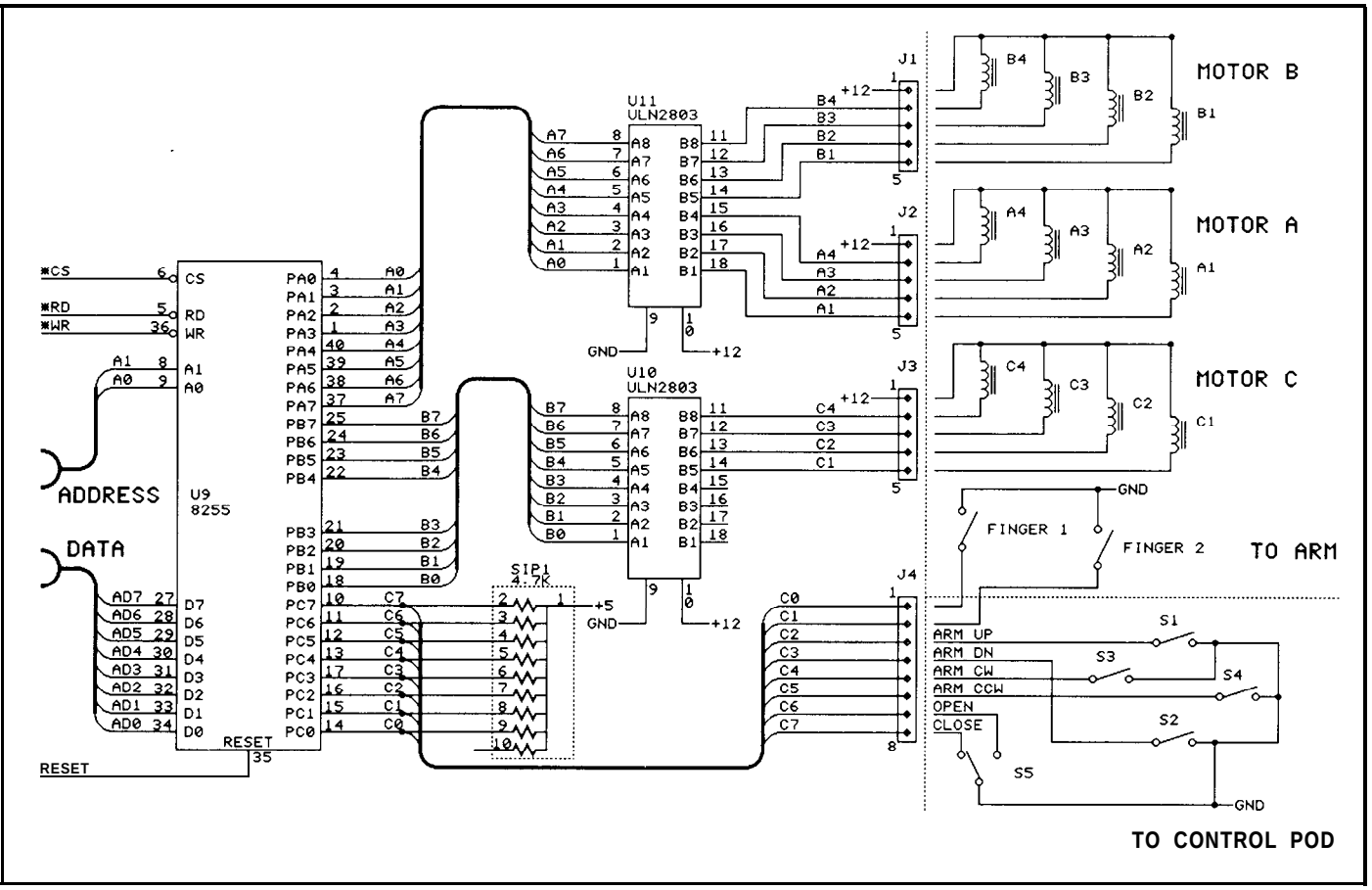


Figure 2 – Schematic of the circuits added to the BCC55 to complete the Stepper motor/control pod interface.

Listing1 -Mechanical Arm Control Software

```
;*****************************************************************
;
;       Mechanical Arm Control Software
;
;       Version 1.4 by Dennis Grim and Tim McDonough
;
;*****************************************************************
;
        ORG 6000H LOC 4000H      ; Assemble for 6000 Hex and save
                                 ;  at 4000 Hex
;
;       Equates
;
        PORT_A EQU      0E800H   ; 8255 Port A
        PORT_B EQU      0E801H   ; 8255 Port B
        PORT_C EQU      0E802H   ; 8255 Port C
        CONTROL  EQU    0E803H   ; a255 Control Port
        PAUSE   EQU     16       ; Value for delay loop
;
;       Initialization
;
        MOV     PSW, #18H        ; Set 8052 to register bank 3
        MOV     RO,#0            ;
        MOV     R1,#1            ;
        MOV     R2,#2            ;
        MOV     DPTR,#CONTROL    ; Load DPTR with 8255 control
                                 ;  register address
        MOV     A,#89H           ; Load control value into
                                 ;  accumulator
        MOVX    @DPTR,A          ; Write the value to the control
                                 ;  register
;
;       Read the switch positions on the control pod
;
LOOP:   MOV     DPTR,#PORT_C     ; Load DPTR with address of 8255
                                 ;  Port C
        MOVX    A,@DPTR          ; Read in the data byte
        MOV     20H,A            ; Move the value to a bit-
                                 ;  addressable location
;
;       Test Motor A Bits
;
        MOV     A,RO             ; Load the motor A phase pointer
                                 ; from reg. 0
        JB      0,LBL1           ; Jump to LBL1 if bit 0 is set
        INC     A                ; Increment the motor phase pointer
        SJMP    LBL2             ; Opposite switch can't be closed
                                 ;  so skip over
LBL1:   JB      1,LBL3           ; Jump to LBL3 if bit 1 is set
        DEC     A                ; Decrement the motor phase pointer
LBL2:   ANL     A,#3             ; AND the accumulator with 3 to
                                 ;  mask LSB
        MOV     RO,A             ; Store the updated pointer in
                                 ;  register 0
;
;       Test Motor B Bits
;
iBL3:   MOV     A,R1             ; Load the motor B phase pointer
                                 ;  from reg. 1
        JB      3,LBL4           ; Jump to LBL4 if bit 3 is set
        INC     A                ; Increment the phase Pointer
        SJMP    LBL5             ; Opposite switch can't be closed
                                 ;  so skip over
LBL4:   JB      2,LBL6           ; Jump to LBL6 if bit 2 is set
        DEC     A                ; Decrement the motor phase pointer
LBL5:   ANL     A,#3             ; AND the accumulator with 3 to
                                 ;  mask LSB
        MOV     R1,A             ; Store the updated pointer in
                                 ;  register 1
;
;       Test Motor C Bits
;
LBL6:   MOV     A,R2             ; Load the motor C phase pointer
                                 ;  from reg. 2
        JB      5,LBL7           ; Jump to LBL7 if bit 5 is set
        INC     A                ; Increment the phase pointer
        SJMP    LBL8             ; Opposite switch can't be closed
                                 ;  so skip over
LBL7:   JB      4,LBL9           ; Jump to LBL9 if bit 4 is set
        DEC     A                ; Decrement the phase pointer
```

(continued on page 54)

register.

Now, we can finally move the motors. The address for port A of the 8255 is placed in DPTR and the data byte for motors A and B is loaded into the accumulator. The accumulator data is transferred to external memory (the 8255) and voila! Motors A and B move if the pod controls have instructed them to do so. Motor C is moved in the same manner by changing the address in DPTR and outputting the data which was saved in register R4.

BASIC-52 was too slow for our purposes and 8052 assembly language turns out to be too fast. If our program jumps back to the top of the loop now, the motors will stand dead still and literally scream for mercy (actually it's more of a high-pitched whine but that doesn't sound as dramatic). A short delay provides an overall step rate of about 80 steps per second which corresponds to the maximum torque the Airpax K82201s used in the arm are capable of in this configuration.

The program delays by simply burning up a little CPU time. First the value 13 (decimal) is loaded into R3. Next, the value in R4 is decremented and tested for 0. Once it reaches 0, R3 is decremented and again tested. Eventually R3 will be zero and the program will jump back to the top of the loop to test the pod switches again. The two lines of the delay code cause register R4 to be counted down from 254 to 0 almost 16 times. The "almost" comes into play the very first time when the value in R4 is unknown and, for our purposes, unimportant.

So, what does the little mechanical arm do? Well if you're expecting it to clean the house, wash the car, or take out the trash you're going to be disappointed. As we indicated in the beginning, this whole thing was an experiment to

(Listing 1 continued from page 53)

```
LBL8:   ANL     A,#3            ; AND the accumulator with 3 to
                                ;  mask LSB
        MOV     R2,A            ; Store the updated pointer in
                                ;  register 2
;
;       Prepare the data to be output to the stepper motors
;
LBL9:   MOV     DPTR,#TBL       ; Load DPTR with the table address
        MOV     A,R0            ; Load acc. with motor A offset
                                ;  into table
        MOVC    A,@A+DPTR       ; Put the value from the table
                                ; into the acc.
        MOV     R3,A            ; Save the value in a scratch
                                ;  register
;
        MOV     A,R1            ; Load acc. with motor B offset
                                ; into table
        MOVC    A,@A+DPTR       ; Put the value from the table
                                ; into the acc.
        SWAP    A               ; Move the B value to the high
                                ; nibble of acc.
        ORL     A,R3            ; OR the motor A nibble with the
                                ;  B nibble
        MOV     R3,A            ; Save the byte in a scratch
                                ;  register
;
        MOV     A,R2            ; Load acc. with motor C offset
                                ;  into table
        MOVC    A,@A+DPTR       ; Put the value from the table
                                ;  into the acc.
        MOV     R4,A            ; Save the value in a scratch
                                ;  register
;
;       Move the motors one step
;
        MOV     DPTR,#PORT_A    ; Set DPTR to port A of the 8255
        MOV     A,R3            ; Load phase data for motors A & B
                                ; into acc.
        MOVX    @DPTR,A         ; Output to port A
;
        MOV     DPTR,#PORT_B    ; Set DPTR to port B of the 8255
        MOV     A,R4            ; Load phase data for motor C into
                                ;  acc.
        MOVX    @DPTR,A         ; Output to port B
;
;       Delay to provide optimum step rate
;
        MOV     R3,#PAUSE       ; Load scratch register with the
                                ; pause value
DLY:    DJNZ    R4,DLY          ; Decrement and jump to DLY if not
                                ;  zero
        DJNZ    R3,DLY
;
        AJMP    LOOP            ; Go check the switches again
;
TBL:    DB      05H,09H,0AH,06H ; Motor phase table
;                                 Winding
;                                 ABCD  Value
;                                 0101   05H
;                                 1001   09H
;                                 1010   0AH
;                                 0110   06H
```

test out some ideas. A few styro- weight items can provide hours of foam blocks, a couple of chess distraction from whatever you were pieces, or any other small, light supposed to be doing. ❑

Editor's Note: Steve Ciarciaoften refers to previous Circuit Cellar articles. These past articles are available in book form from Circuit Cellar Inc., 4 Park St., Suite 12, Vernon, CT 06066, (203) 875-2751.

Ciarcia's Circuit Cellar Volume I covers articles in BYTE from September 1977 through November 1978. Volume II covers December 1978 through June 1980. Volume III covers July 1980 through December 1981. Volume IV covers January 1982 through June 1983. Volume V covers July 1983 through December 1984. Volume VI covers January 1985 through June 1986.

Resources for Robot Arm Materials

Materials for constructing a project such as this are often hard to come by if you don't know where to look. While there are no doubt many companies who supply similar components, the mechanical items such as the aluminum sheet stock, gears, bearinga, and stainless-steelshaft material can be obtained in small quantities from the following vendors:

Small Parts, Inc.
6891 N.E. Third Ave.
P.O. Box 381736
Miami, FL 33238-1736
(305) 751-0856

Stock Drive Products
2101 Jericho Turnpike
New Hyde Park, NY 11040
(516) 328-0200

Winfred M. Berg, Inc.
499 Ocean Ave.
East Rockaway, NY 11518
(516) 599-5010

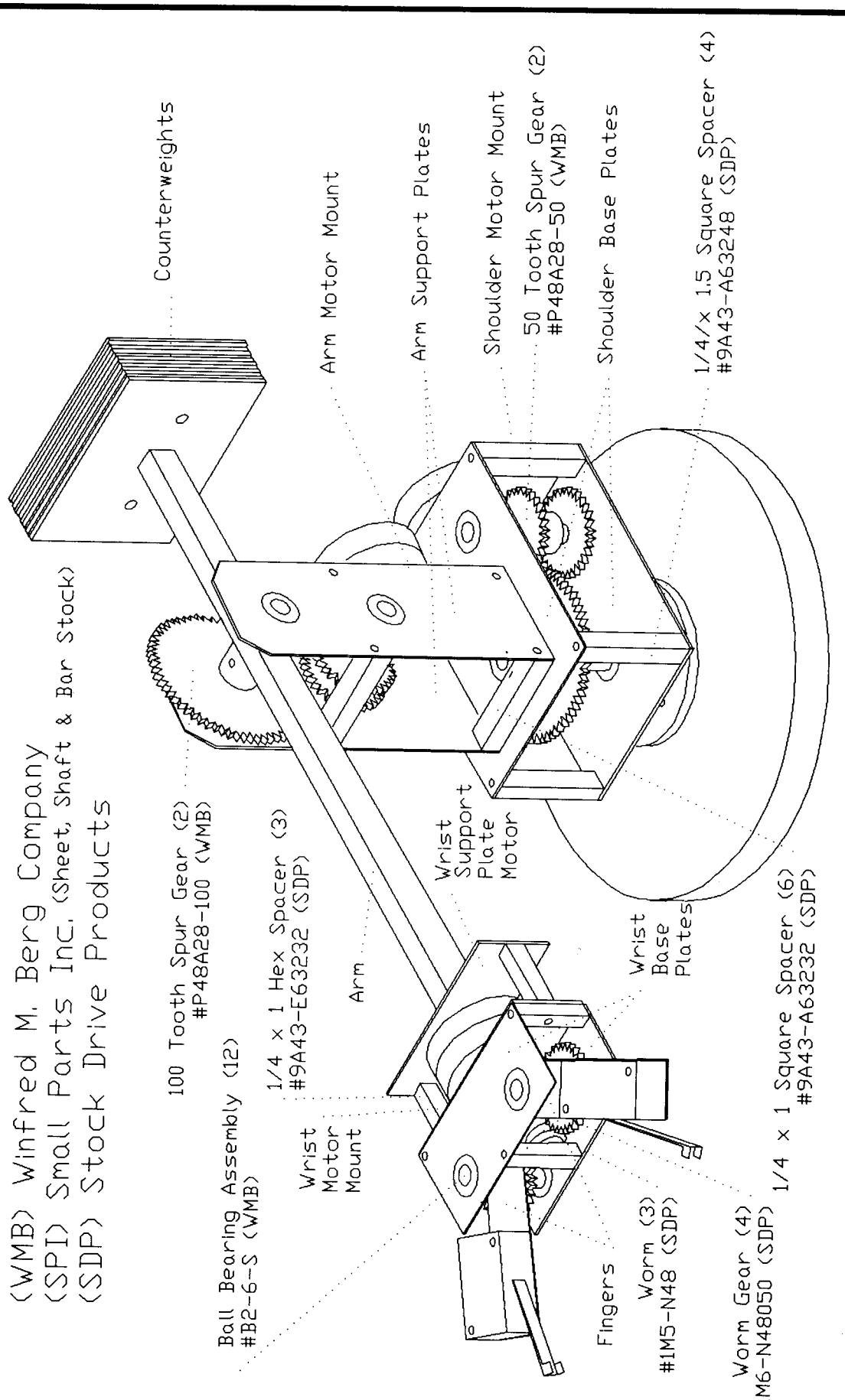
The following books and publications were useful in researching the mechanical design of the mechanical arm:

Airpax Stepper Motor Handbook
Publication No. MR-116-R4
Airpax Corporation
Cheshire Diviaion
Cheshire Industrial Park
Cheshire, CT 06410

Handbook of Industrial Robotics
Edited by Shimon Y. Nof
John Wiley & Sons, 1986
ISBN O-471-89684-5

Fundamentals of Robot Technology
D.J. Todd Halsted Press/
John Wiley & Sons, 1986
ISBN O-470-20301-3

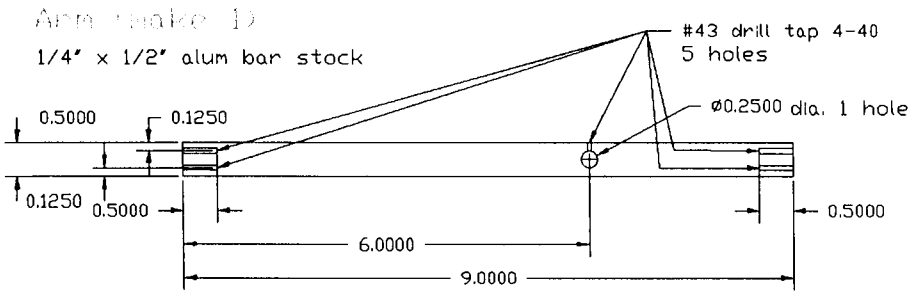
Robotics, An Introduction
D. McCloy and M. Harris
Halsted Press/
John Wiley & Sons, 1986
ISBN 0-470-20325-O

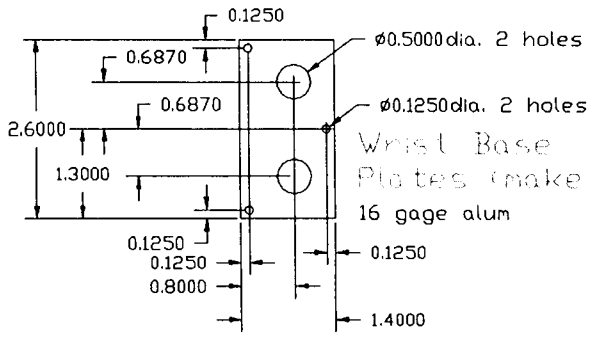


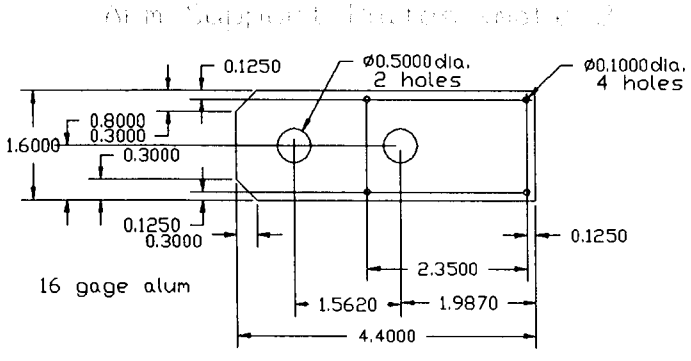
Winfred M. Berg, Inc.
499 Ocean Ave.
East Rockaway, NY 11518
(516) 599-5010

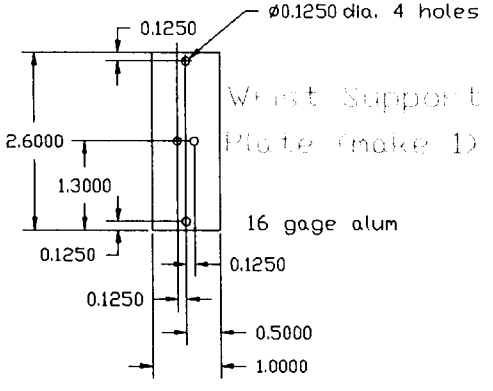
Small Parts, Inc.
6891 N.E. Third Ave.
P.O. Box 381736
Miami, FL 33238-1736
(305) 751-0856

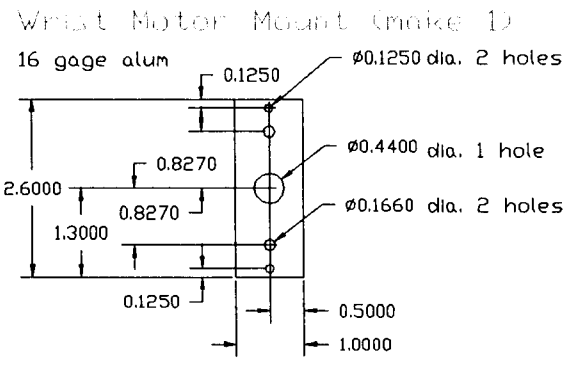
Stock Drive Products
2101 Jericho Turnpike
New Hyde Park, NY 11040
(516) 328-0200

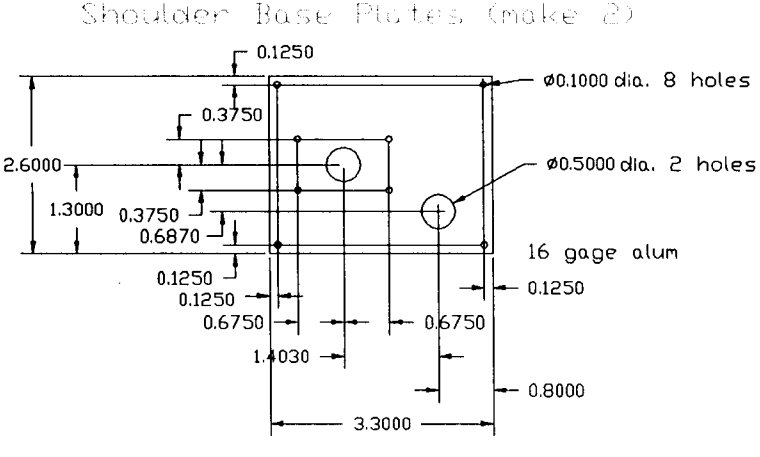


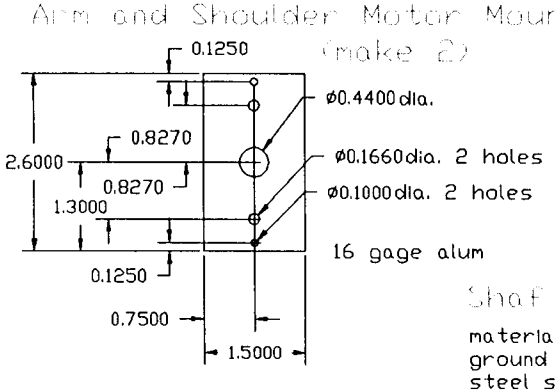


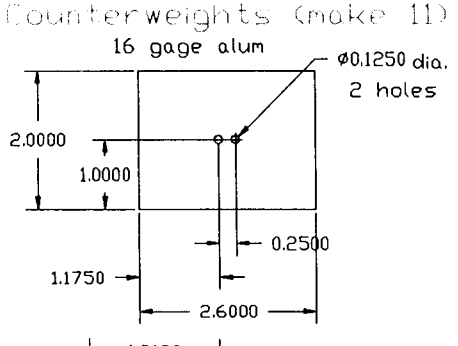


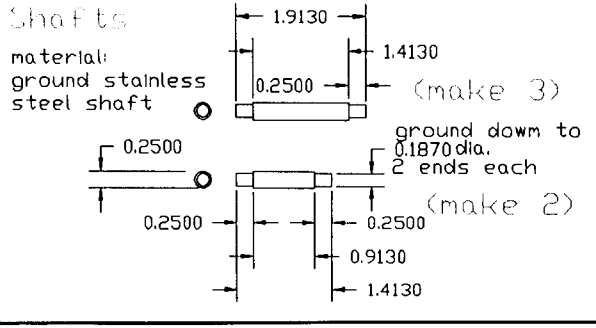












ROBOT ARM PARTS LIST