ISSN # 0748-9331

THE COMPUTER JOURNAL®

For Those Who Interface, Build, and Apply Micros

Vol. [illegible] Issue Number 12 $2.50 US

Editor's Page

Choosing A Programming Language

One way to start a lively discussion in a group of computerists is to praise or criticize a popular programming language. Few people can discuss languages without becoming irrational and excited. It is unfortunate that there is not one language which is best for every possible application so that the choice would be simple, but the fact is that you will need different languages for different purposes—unless you are willing to limit yourself by trying to force the one language you are familiar with to do things for which it is not suitable.

High level languages (which we'll call HLL) are very popular with many people at the present time, but there are some who offer a good case against certain applications of HLLs. One of the most interesting proponents of using the right language for the right application is the editor of *DTACK Grounded*,* a publication which should be on your "must read" list if you are at all interested in what's going on with systems and HLLs in the marketplace.

Perhaps we should take a moment to describe what low level and high level languages are. An example of the lowest level language which we would normally encounter is assembly language, where we enter mnemonics which an assembler converts to hexadecimal code. Examples of HLL would be Pascal or C. In addition to the general purpose languages, there are also many special purpose languages such as LISP, Logo, FORTH, etc. which were developed for specific applications. I'm sure that I'll get in trouble with that statement because some people insist that one of these specialized languages is really the best language to use for everything.

The only way to decide which language to use is to become familiar with several and study how each one handles certain operations. For an example of a low level language (LLL) application, the source code for my CP/M BIOS is supplied as an assembly language text file which I can modify using a word processor, and then re-assemble. Assembly language seems the natural choice for modifying a program when the source code is supplied in assembler code, and I don't believe that it is practical to do it with any other language. On the other hand, if someone asked for a chart showing the circumference and areas of circles with diameters from one to a hundred, I'd use a short BASIC program to print out the chart. Any HLL would be better than assembly language for this simple one time job, but I'd use BASIC because I'm familiar with it and I can run the program without saving anything to disk, or compiling, or linking to a library. BASIC is great for a short, simple, seldom used program.

The choice is not alway as simple as in the previous two examples, and you'll have to consider programming time, run time, frequency of program use, code size, special requirements such as string handling, transportability between systems, utilities available, and much more before making your decision. The natural reaction is to use whatever language you are familiar with, but one of the first considerations should be whether the program will be for your own use or if it will be distributed to others. You can use anything you have for use on your own system, but you have to consider other users' systems if the program will be distributed. Again, I'll use two examples: if you are writing a Z-80 assembly language debugging program you can write it in Z-80 code because everyone will be using it on a Z-80 system. On the other hand, if you are writing a

continued on page 11

*DTACK Grounded, 1415 E. McFadden, Suite F, Santa Ana CA 92705. $15/10 issues.

Editor/Publisher	*Art Carlson*
Art Director	*Joan Thompson*
Technical Editor	*Lance Rose*
Production Assistant	*Judie Overbeek*
Contributing Editor	*Ernie Brooner*

The Computer Journal® is published 12 times a year. Annual subscription is $24 in the U.S., $30 in Canada, and $48 airmail in other countries.

Entire contents copyright © 1984 by The Computer Journal.

Postmaster: Send address changes to: The Computer Journal, P.O. Box 1697, Kalispell, MT 59903-1697.

Address all editorial, advertising and subscription inquires to: The Computer Journal, P.O. Box 1697, Kalispell, MT 59903-1697.

TRICKS OF THE TRADE:
Installing New I/O Drivers in a BIOS

by Bill Kibler

There are many shortcuts that I use when making modifications to equipment; I call them "tricks of the trade." These tricks help by-pass the normally long and tedious procedures for making things work. This article is the first in a series on "tricks of the trade," in which I hope to cover some of the quick and simple solutions to common system integration problems.

The Beginning

The most common problem, and the one we will start with, is that of installing new I/O drivers in a BIOS. Typically, the user is provided with a running system, but no printer device. Many of the newer systems now have install programs that set up the ports for you. As a system integrator, however, the task is your own—especially when the system is not standard. What you will need is a clear understanding of your system.

In a standard CP/M system there are already entry points in the BIOS for the extra routines. Printing out the listing of your current BIOS.ASM files will show how the current devices are handled. For some lucky persons, the routines for the manufacturer's standard installation may be present. For those, the task will be to change the routines to reflect the new devices. However, many will find a call to the terminal routine for the printer entry point. This echoes the output to your terminal until you do the modifications. Some manufacturers provide detailed instructions on how to change these conditions. For those who do not have that information, I will try and cover all that was omitted.

System Calls

There are two topics that must be understood before you can install a new I/O routine: first, the way the Disk Operating System (DOS) calls the routine, and secondly, the actual device's data handling procedures. For DOS calls, the manuals will have a detailed discussion explaining which registers of the CPU should contain what. The system is composed of a central processing unit (CPU), which is made up of temporary storage segments (registers) that can pass data between themselves, locations in memory, or input/output (I/O) locations. In CP/M this CPU is a Z80 or 8080 device which also has a separate set of locations for I/O. The Apple DOS is based on the 6502 CPU, which uses memory locations for I/O. In either case, they have their own way of talking to dedicated hardware with their registers.

Another way of saying this is that the information presented on your terminal screen is first loaded into a register of the CPU and then transferred to the I/O device at some location in the machine. Unfortunately, the device may or may not be ready for this information. The DOS also has to know which location or alternate location is being used. Programmers hate writing programs for each special case of output devices, so some standards have been set by the DOS makers to give stability to the problem. These standards are the BDOS and BIOS organization and conventions. The BDOS has an established entry point to the system which checks a set register of the CPU (8080's "C") to find out which procedure is to be performed. This also tells the system which other registers have information, or will recieve information. The BDOS (Basic Disk Operating System) will then take this information and call the proper BIOS (Basic Input and Output System) entry point or points to get the desired results.

To understand what your routine is doing, search the manual for a listing of BIOS or BDOS routines and you will find the values and registers used. For a printer, the routine is called "list" or "listing device," and is the sixth entry point of the BIOS. The BDOS system call will have a five in the "C" register of the function call. In the BDOS function call the "E" register has the character to print, which gets moved to the "C" register when calling the BIOS. That means the user/programmer puts the single character to be printed in the "E" register, puts a five in the "C" register, and then calls address "0005H," the entry point of the BDOS. The systems programmer will know that the same character will be in the "C" when the sixth entry to the BIOS is called by the BDOS, and the BIOS will then output it to the printer. Before outputting the character, the BIOS will test to see whether or not the printer is ready. If the printer is not turned on, or even not attached, most CP/M systems will wait forever for the printer to become "ready"; this is the hated "locked up system," and will require a master reset. You may be asking "why not output directly to the BIOS and skip the BDOS call?" This is just what a lot of IBM PC programmers have done to make the machine fast enough to use, but it also makes the system incompatible. As newer versions of the DOS are made, the entry points of the BIOS may change. So far, the entry points of the BDOS and the values in the "C" register have remained the same, which is why programs for CP/M 2.2 still work on CP/M 3.0 if done with proper calls to the BDOS.

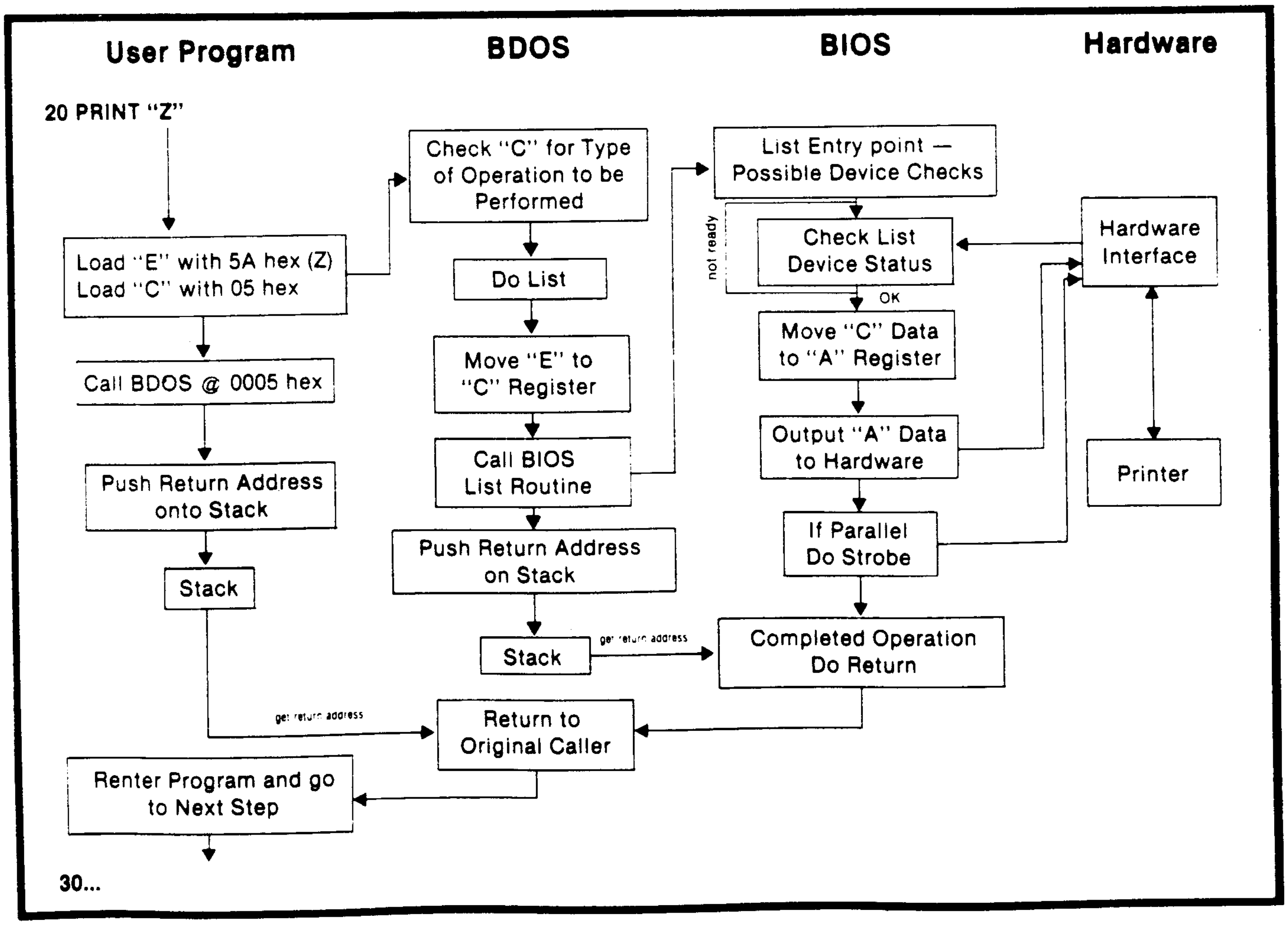
I/O Routines

Leaving the user and BDOS programming information for another time, let's start from the BIOS entry point. The BDOS will make a standard CALL into the BIOS. This means that a return address was pushed onto the stack, so that when a RETURN instruction is encountered at the end of the routine, the CPU will return to the address it takes off the stack. The stack, which may be short, will be set up in BDOS and BIOS. It will be almost fully used when entering the BIOS, so care is needed here that the stack is protected or not filled up with data. For simple routines like printer output this is seldom a problem, but for multiple nested calls in disk routines it can cause grief. You should also remember that the stack is just a place in memory, and as such can also be written into by mistake, so find out where it is and keep it in mind when programming. For complex routines the STACK will be protected by saving the entry stack address and putting your own in place of it; the reverse will happen when leaving the routine.

Normally, the procedure for the printer or list routine is to see if the printer is ready, and then output the data. A separate routine is available in the BIOS for checking list status—this is the 16th entry point, and it will return the status in the "A" register. With this in mind, our routine's first statment will be a call to the list status routine. The next step is checking the "A" register to see whether or not the status is ready. If not ready, recall the status check. If ready, continue on. Next, we will need to know if this is a serial or a parallel printer, as each has their own routine (and status check). For serial operation just load the "A" register with data from the "C" register and output it. For centronics style parallel printers the data will be loaded and then the printer will be strobed. Strobing signals the printer that a character has been loaded and is now ready for printing. After either routine a RETURN instruction is executed, returning the program to the BDOS and then to the user program.

Hardware

When you start looking at hardware information it is often helpful to draw a block diagram. Once the concept of the routine is understood, the specifics of the bits and addresses are next. This translates to finding the I/O device's manuals and seeing how the designers of the product intended the data I/O to be handled. For most systems this is straightforward. GODBOUT systems have another, interesting way of doing it. The standard way is to check the port addresses assigned for data in/out and status/command. These ports are usually tied directly to the actual device and will reflect the original chip maker's design parameters. In some cases it will be necessary to check the chip manufacturer's specification sheet to find out how to set up the device. In the case of GODBOUT you will first need to send a device activation number to the command port, as the system uses seven ports for several devices. This saves on I/O ports and can add some creative programming techniques that could speed up the I/O. However for most novice programmers this will simply add


User Program
BDOS
BIOS
Hardware
20 PRINT "Z"
Load "E" with 5A hex (Z) Load "C" with 05 hex
Call BDOS @ 0005 hex
Push Return Address onto Stack
Stack
Check "C" for Type of Operation to be Performed
Do List
Move "E" to "C" Register
Call BIOS List Routine
Push Return Address on Stack
Stack
get return address
List Entry point — Possible Device Checks
not ready
Check List Device Status
OK
Move "C" Data to "A" Register
Output "A" Data to Hardware
If Parallel Do Strobe
Completed Operation Do Return
Hardware Interface
Printer
get return address
Return to Original Caller
Renter Program and go to Next Step
30...

another level of possible error and confusion. When dealing with this type of problem, remember to add this step to your flowchart, and be sure you understand its operation before you start programming.

The Z80 SIO is a typical serial device. It is the companion to the Z80 CPU, but was designed for serial I/O. This device is a form of CPU in which the I/O and registers have been arranged to provide the desired type of operation. For the SIO it is two serial ports, and for the Z80 PIO it is two parallel ports. The devices have two modes of operation—a data mode and a command mode. As with a CPU, you load instructions into a register and then the CPU performs some task. These special devices work the same way except that you will most likely store the commands in the appropriate register only after resetting the system. This is called initialization of the device and is based on the original designer's specifications. These values stored at reset will have a single bit assigned for each function, "on" for function activation, and "off" if that function is not to be used. An example of this is the number of data bits used; this can be 5 to 8 bits and a value will be 0 to 3 to represent the data bit options.

A typical system will assign an I/O location for each of the functions of the device. This means that an address will be used for the command register, another for the "A" data I/O, and one for the "B" data I/O. Communicating with these I/O addresses will allow different functions to be performed, depending on whether the operation is a READ or a WRITE. For a more in depth study of how the serial device is programmed, you will need to buy the programmer's manual for the device. If you are like me, you deal with far too many devices to buy all that information. I usually just try to see what was done elsewhere in the system. The BIOS will most likely initialize the devices for some form of operation. Modifying that section of the BIOS for your application may be easier than starting from scratch. For simple tasks, just knowing what each value of the registers stands for will be enough once you have read one programmer's manual. Most of these devices all work the same—they just use different assignments for the commands.

The procedure then is to check that the device's registers are set up correctly at initialization time, and then to determine from the specs what status values are returned from the command mode in a READ operation. This will tell if the device is ready or not. In the serial mode, it tells if the data registers are empty or full. For the transmit direction we want the register to be empty, and for the recieve direction it has to be full of data. To find out whether the registers are empty or full, you read the status port, mask off the unwanted data bits, and check this remaining value with the desired value. This can be either a high or low (a "1" or a "0") and, depending on the processor, can be handled in different ways. An 8080 cannot do bit tests like the Z80, so if you are doing programs for unknown machines do not use a bit test. The assembler that comes with CP/M is setup for 8080 code and as such is usually the one used for assembling most BIOSs. This may not be the most efficient way, but it keeps you from buying a new assembler to handle Z80 code.

My favorite time saving trick is to use DDT to check my routine for proper masking and options. I either look at other code and see how they did it, or make a stab at what I think it should be. This usually means doing an "ANI" with some mask value and then doing a "JNZ" to recheck the status. If you have been unable to determine the right status information for your device you can also use DDT to try and find out what happens when. The manuals will tell you most of what you need to know about DDT, so I'll only cover it briefly.

DDT, ASM, and MAC

DDT is the CP/M debugger. It will allow you to check the operation of a program or find out where it is going astray. Normally this is done only after a program has been written and is not working. I use mine for checking out my ideas of what code should be like. In our current example, that means using the assembler portion to write the printer output routine, complete with status checks. Then use the examine mode to set the desired registers with the correct values. The trace function will allow you to display all the registers and the program steps. This display will show exactly what data is going where, and whether or not you are masking correctly. There are several other functions that may help programmers, but a more complex debugger is necessary, such as SID or ZSID, which for about a hundred dollars will add some useful features. The best way to understand these is to get the manual and experiment until you fully understand the new commands.

Should you have a program where the code exists for another device, using the DDT functions to subsitute new I/O addresses can sometimes save reassembling a whole new BIOS. In the case of setting new speeds for a serial port, or reassigning an I/O function for a parallel port, DDT will do it much easier than anything else. Good manuals will list just such procedures for some of their standard modifications, so look at their sample for a "how to" approach.

When assembling a new BIOS many problems may crop up. One of these is the Z80 code, where the orginal BIOS is in MACROS and will require MAC to assemble it. The BIOS in Teletek systems is written using macros and must have a new version of Microsoft's M80 to be assembled. In cases of minor code changes, use of DDT is the only way out. Write to Teletek to get a copy of the customization notes that I wrote for some of their standard changes. Not all manufacturers will have such support, so using a wordprocessor to recombine all the files is one way of getting macros assembled. For real insanity, DRI (makers of CP/M) uses macros for their CP/M 3.0, in both 8080 and Z80 code. You will find that most of the simple statements are in 8080 and only the faster block moves are listed in Z80. Now this is not as bad as you might think—for those using a Z80 CPU the macro definitions allow the use of Z80 routines and their increased speed. If you have an 8080 however, you can rewrite the macro definitions to reflect the 8080 procedure for the same function. I find, however, that it is better to change an "LDIR" definition to a "CALL NLDIR", a new routine which can be called at any time. Otherwise the assembler will add 10 or 15 lines of code each time you put in an "LDIR" instruction.

Do not let macros scare you away from making changes in a BIOS—they are much like subroutines in a BASIC program. What you will need however, is to make a lot of flow charts and cross reference charts. Without this written information you will become lost quickly as you move from one macro to another. There are really two type of macros; those that are definitions and get put in library modules, and those that are complete functions (like a disk handling routine). The library listing for DRI's Z80 macros is mostly define byte or word ("DB") statements, which is what I do sometimes for a simple Z80 code insertion while using DRI's ASM. What this means is that you insert the actual hex codes in the program for the Z80 machine code. This is fine until you use DDT, which will not know what the code means (you will need ZSID for Z80 code). This procedure also takes time, as you will have to look up each instruction and get the hex values needed.

The second type of macros consist of complete routines in which the input and output data come from other programs. This shared data is listed at the beginning of the macros, and tells the macro assembler what data to include from other files. When assembling these files, they will be done one at a time and then linked together in some order. Linking passes the addresses between macros for those open variables and sets the order of routines within the main program. Library routines are used during the assembler time when a statement is listed whose definition is in the library (the libraries used are declared at the begining of the macro.)

Putting It Together

Lets start tying this up by getting to the "checking-it-out" phase. This is where I take the newly changed code (either DDTed or reassembled) and make it run. Hopefully the manual will give you the procedure to be followed for reassembling code. For ASM programs this is rather straightforward; assemble the program, use DDT to attach it to the old system file, and reload it on the system tracks. If you DDTed it, you need only put it on the system tracks and try it. For those who are completely confused about these steps, lets review the system's makeup. You are supplied with a complete system in several ways. The main boot disk has the system tracks already loaded with a system. To get at it, you use "SYSGEN.COM" with a "SAVE xx." This will save the necessary file for later DDTing, and you will need to check the manual for size of file. For non-CP/M systems, check the manual—it will list some procedure or program to get the system's track information. Newer systems, and those with more information than can fit on the system tracks, will put a simple loader on those tracks, and load a file called "CPM3.SYS" (mainly the .SYS). These system files are handled much like any other COM file or program file. In CP/M 3.0 there is a separate program for generating a new system, called "GENCPM." This takes the linked output files and generates a new "SYS" file.

MOVCPM is a program that has a relocatable source of the system, and is what is used when generating a new memory size system. I generally do not modify this program but use the results for my specific memory size. In other words, generate a new system file, save it as indicated by the MOVCPM program, add the new BIOS with DDT, and SAVE again but with a new name and maybe even a new file size. One thing to remember about MOVCPM is that it checks for serial numbers. These numbers are easy to find and change (only two locations in CP/M 2.2), and as long as you don't intermix systems you will not see a "SYNCRONIZATION ERROR" and have to reboot.

The next step is putting this all together into a running system, so let's look at a typical installation of a new BIOS on a 64K system using normal size programs.

1) Get the system: MOVCPM 64 * (generate a 64K system and get it ready for a save)
   ready for "SYSGEN" or
   "SAVE 36 CPM64.COM"
2) Save system: A> SAVE 36 CPM64.COM
3) Make new BIOS: A> ASM BIOS (use .AAZ if no prn file)
4) Correct any possible errors that the assembler may have found, check PRN file for possible memory conflicts.
5) DDT in new BIOS: A> DDT CPM64.COM
   2400 0100 C3FF
   D1F00 (make sure of BIOS begining) (look for string of jumps"C3")
   FIF80 4000 00 (clear out old BIOS)
   HIF80 F200 (do hex addition)
   1180 2D80 (get OFFSET)
   IBIOS.HEX
   R2D80 (loads BIOS at proper OFFSET)
   D1F00 (check to see if loaded and get end of file location)
   G0 (back to system)
6) Save new system: A> SAVE 36 CPM64X.COM
7) Put on system tracks: A> SYSGEN
   SOURCE NAME OR SKIP (skip as source is still in memory)
   DESTINATION DISK b (put files on spare disk in B drive)
8) Test system by putting disk in A and rebooting.

Now that doesn't look too hard, but I am sure a lot of people got lost at OFFSET. When you assemble the BIOS, it will be for its final location in memory, in this case F200hex. Using DDT you can find your current BIOS by looking at page 00 or 0000 hex to 0100 hex. At 0000hex there is a jump to the first entry of the BIOS, but remember that it is low address first then high value (remember, the jump is a C3). The disk read routine checks the addresses assigned to the program and tries to load it there. If you just have DDT load the program, it will go to the assigned address or F200. Because we are replacing the BIOS in our old program we want it to be in a location we can save it from, or in the 0100 hex region (which is where SYSGEN leaves it). To do this, we must get an OFFSET value that DDT adds to the address the disk controller supplies, and compute a new memory location at which to load the file. We do this by using the hex function of DDT and supplying the desired location and the source location. The H function will give use of the hex addition and subtraction values; use the subtraction value. This value is added to the R command and should load the program at the proper location.

Conclusion

If all went well, you should now have a new printer function routine for your BIOS. If not, you may have missed a few steps or had problems with your routine. I could give you more listings and instructions, but to do so would remove some of the challenge and learning steps needed to become a good programmer. Remember some of the words of wisdom I gave in previous issues, about trying it over and over till it all works. Plan to have problems, and do not be surprised when they occur—that is part of the challenge, and adds to the thrill of success when it all goes well. ■

CT COMPUTER® TRADER MAGAZINE

★ ★ ★ **LIMITED TIME OFFER** ★ ★ ★

BAKER'S DOZEN SPECIAL!

$12.00 for 13 Issues

Regular Subscription $15.00 Year

Foreign Subscription: $55.00 (air mail)
$35.00 (surface)

Articles on MOST Home Computers, HAM Radio, hardware & software reviews, programs, computer languages and construction, plus much more!!!

Classified Ads for Computer & Ham Radio Equipment

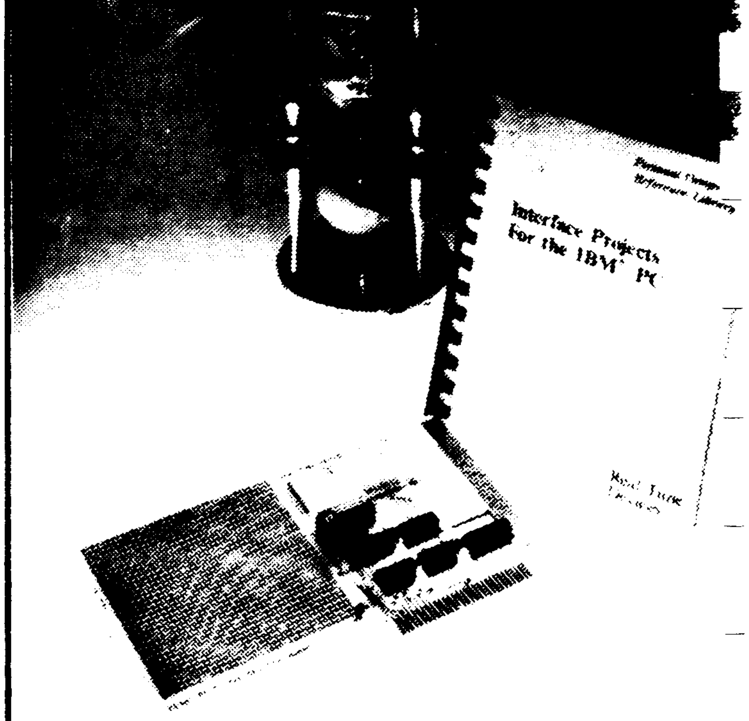
FREE CLASSIFIED ADS
for subscribers

Excellent Display and Classified Ad Rates
Full National Coverage

CHET LAMBERT, W4WDR
1704 Sam Drive • Birmingham, AL 35235
(205) 854-0271
Sample Copy $2.50

IBM PC OWNERS
DON'T WASTE YOUR TIME. . .

prototyping the same circuitry when you can quickly and easily implement your original design with Real Time Devices PD100 Hardware Development Board



With the PD100, we've done most of the difficult work for you. The PD100 contains a buffered data bus, switchable address decoder, prototyping area and easily available wire wrap posts. All that needs to be done is to make simple connections to the wire wrap posts and you have a unique design implemented in minutes rather than days. Not familiar with interfacing? Our comprehensive, 116-page manual "Interface Projects for the IBM PC" includes an introduction to interfacing and details implementing and programming A/D, D/A converters, I/O ports, connection of transducers and dozens of useful circuits.
The board and manual are invaluable aids to engineers, hobbyists, students and anyone seriously interested in expanding the power of the IBM PC. The PD100 will make your prototyping a lot easier . . . we guarantee it!"

MANUAL TOPICS
- **Introduction to Interfacing**
- **Prototype Construction Techniques**
- **Simplest I/O Devices**
- **I/O Software Example Commands**
- **Real World Interfacing**
- **Example Projects**
- **Analog Interfacing and Analog Signal Conditioning**
- **PD100 Schematic and Specifications**

BOARD FEATURES
- **1600-hole on board wire wrap area accommodates up to 40 DIP sockets**
- **Easily accessible buffered data bus, control signals, power supply, wire wrap posts**
- **Four switch selectable addresses; no contention with existing IBM PC peripherals**
- **Gold plated edge connector**

ORDERING INFORMATION
PD100 WITH MANUAL - $99.00 PLUS $3.50 P&H
MANUAL ONLY - $20.00 POSTPAID
PENNSYLVANIA RESIDENTS ADD 6% SALES TAX
MASTERCARD AND VISA ACCEPTED. SEND CHECK OR MONEY ORDER TO
REAL TIME DEVICES
1930 PARK FOREST AVENUE
P.O. BOX 906
STATE COLLEGE, PA 16801
PHONE (814) 234-8087
DEALER INQUIRES WELCOME

Searching for Useful Information?

The Computer Journal is for those who interface, build, and apply micros. No other magazine gives you the fact filled, how-to, technical articles that you need to use micros for real world applications. Here is a list of recent articles.

Volume 1, Number 1:
- The RS-232-C Serial Interface. Part One
- Telecomputing with the Apple][: Transferring Binary Files
- Beginner's Column, Part One: Getting Started
- Build an "Epram"

Volume 1, Number 2:
- File Transfer Programs for CP/M
- The RS-232-C Serial Interface, Part Two
- Build a Hardware Print Spooler, Part One: Background and Design
- A Review of Floppy Disk Formats
- Sending Morse Code With an Apple][
- Beginner's Column, Part Two: Basic Concepts and Formulas in Electronics

Volume 1, Number 3:
- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for the Apple][
- ASCII Reference Chart
- Modems for Micros
- The CP/M Operating System
- Build a Hardware Print Spooler, Part Two: Construction

Volume 1, Number 4:
- Optoelectronics, Part One: Detecting, Generating, and Using Light in Electronics
- Multi-user: An Introduction
- Making the CP/M User Function More Useful
- Build a Hardware Print Spooler, Part Three: Enhancements
- Beginner's Column, Part Three: Power Supply Design

Volume 2, Number 1:
- Optoelectronics, Part Two: Practical Applications
- Multi-user: Multi-Processor Systems
- True RMS Measurements
- Gemini-10X: Modifications to Allow both Serial and Parallel Operation

Volume 2, Number 2:
- Build a High Resolution S-100 Graphics Board, Part One: Video Displays
- System Integration, Part One: Selecting System Components
- Optoelectronics, Part Three: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Volume 2, Number 3:
- Heuristic Search in Hi-Q
- Build a High-Resolution S-100 Graphics Board, Part Two: Theory of Operation
- Multi-user: Etherseries
- System Integration, Part Two: Disk Controllers and CP/M 2.2 System Generation

Volume 2, Number 4:
- Build a VIC-20 EPROM Programmer
- Multi-user: CP/Net
- Build a High-Resolution S-100 Graphics Board, Part Three: Construction
- System Integration, Part Three: CP/M 3.0
- Linear Optimization with Micros
- LSTTL Reference Chart

Volume 2, Number 5:
- Threaded Interpretive Language, Part One: Introduction and Elementary Routines
- Interfacing Tips and Troubles: DC to DC Converters
- Multi-user: C-NET
- Reading PCDOS Diskettes with the Morrow Micro Decision
- LSTTL Reference Chart
- DOS Wars
- Build a Code Photoreader

Volume 2, Number 6:
- The FORTH Language; A Learner's Perspective
- Build an Affordable Graphics Tablet for the Apple][
- Multi-user: Some Generic Components and Techniques
- Make a Simple TTL Logic Tester
- Interfacing Tips and Troubles: Noise Problems
- Write Your Own Threaded Language, Part Two: Input-Output Routines and Dictionary Management
- TTL Reference Chart

Volume 2, Number 7:
- Putting the CP/M IOBYTE To Work
- Write Your Own Threaded Language, Part Three: Secondary words
- Interfacing Tips and Troubles: Noise Problems, Part Two
- Build a 68008 CPU Board for the S-100 Bus
- Writing and Evaluating Documentation
- Electronic Dial Indicator: A Reader Design Project

Back issues: $3.25 in the U.S. and Canada, $5.50 in other countries (air mail postage included.) Send payment with your complete name and address to The Computer Journal, PO Box 1697, Kalispell, MT 59903. Allow 3 to 4 weeks for delivery.

Interfacing Tips and Troubles

A Column by Neil Bungard

Noise Problems, Part Three

This month in "Interfacing Tips and Troubles" we will conclude our discussion on noise problems associated with interfacing. In parts one and two of this series we looked at power supply noise and noise associated with data transmission lines. Part three addresses noise generation from within the interface circuit, and noise induced from outside sources.

Noise Generated Within The Interface

Last month we eliminated all possible noise associated with the interface connecting cable. —But you say that you're still having problems? The noise may be originating inside the interface circuit itself. All the standard symptoms like latches dropping bits, flip flops arbitrarily changing states, counters spontaneously counting, buses locking up, etc., are malfunctions which alert you to the fact that you may have noise problems. In this section we will look at techniques which are designed to eliminate noise within the interface circuit itself.

Physical Separation

Once again, physical separation is the easiest noise reduction technique to implement. When you are laying out the interface project for placement of parts, keep as much distance between oscillator, power switching, and logic sections as possible. The more distance you have between these sections the better your chances of eliminating interference.

Decoupling Capacitors.

If you do not remember anything else from this series on noise reduction, remember that the use of decoupling capacitors is probably the single most effective noise reduction technique that you can use. Figure 1 shows how a decoupling capacitor should be connected to an IC. Decoupling capacitors counteract the effects of self inductance when an IC switches logic states. Self inductance occurs because when an IC switches states it draws a relatively high current for a few nanoseconds. These high currents generate large magnetic fields propagating from the power supply lines that feed the IC. The magnetic fields in turn induce a counter current back into the very lines that are generating them. The counter current opposes the supply current and the net result is a current deficit at the IC for a few nanoseconds. With no current applied to the IC, the IC attempts to turn off. If the IC is a latch, it looses its data. If the IC is a flip flop, it changes states. If the IC is a counter, it resets. Decoupling capacitors alleviate these problems by acting as small storage batteries during times of current deficits. For the few nanoseconds that current is not being supplied to the IC, the capacitor dumps its charge back into the IC's power supply inputs, thus ensuring that the IC is always being supplied a current.

Decoupling capacitors are typically 0.01mfd ceramic disk capacitors. When installing them on the IC, keep the leads as short as possible to reduce the chance of self inductance in the leads of the capacitor itself. The rules of thumb for applying decoupling capacitors to your interface circuit are:

- Use one 10mfd capacitor where DC power enters the interface board.
- Use one 0.01mfd capacitor on every latch, counter, flip flop, oscillator IC, tristate IC, memory IC, and multiplexer/demultiplexer IC.
- Use One 0.01mfd capacitor for every five "gate" ICs (7400, 7404, 7432, etc).

Frankly you cannot use too many decoupling capacitors, so use them liberally.

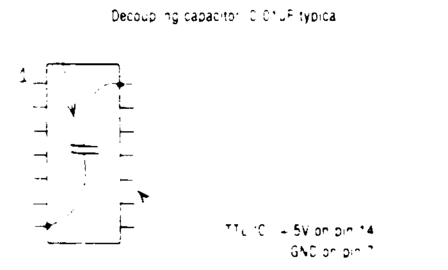


Figure 1: Decoupling Capacitor - IC Connection

Inductive Spike Suppression

If you have inductive, high current devices on your interface board (like relays, coils, solenoids, etc.), you will need to take precautions against inductive voltage spikes. These spikes will not only cause logic circuits to malfunction, they will actually destroy ICs and transistors. Guarding against inductive voltage spikes is a relatively straightfoward task. Figure 2 shows three schemes for reducing inductive voltage spikes. Figure 2a uses a silicon diode to prevent flyback currents from destroying switching circuits when they are turned off. Figure 2b uses a diode, resistor, capacitor network to absorb flyback currents and

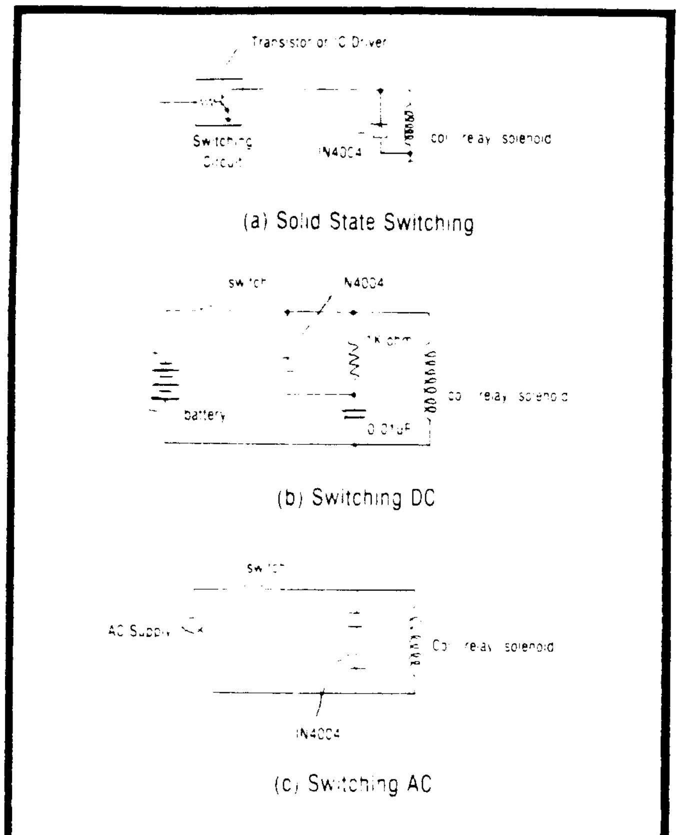


Figure 2: Inductive Spike Suppression

prevent arcing across the switch when it opens. Arcing sends a full spectrum of noise dancing through logic circuits, which can produce unpredictable malfunctions. Figure 2c uses back-to-back diodes in a suppression network to guard against flyback currents when switching inductive loads with an AC supply. Always use the appropriate suppression network when you use inductive devices. It will save you headaches (and maybe a few parts) in the long run.

Alternate Ground Paths

One relatively subtle source of noise which can cause extreme problems (especially if you are using operational amplifiers) is common impedance noise. Figure 3 shows a typical circuit situation where common impedance noise can cause problems. The switching circuit (a) can generate current fluctuations in its ground return path as it switches on and off. The ground return path has some impedance associated with it, and small voltage fluctuations may result. If an OP amp uses the same ground return path for its reference terminal, noise will be generated in the OP amp output. It is rarely obvious from the wiring diagram that this problem is likely to occur. Keep common impedance in mind when wiring the circuit, and try to wire reference junctions as close to the power supply terminals on the interface board as possible.

Noise Induced From Outside Sources

Now we come to "black magic noise." I say that because sometimes it seems as though this type of noise comes from nowhere. Of course this is not true. Noise will always be generated from a logical source, but the source can be hard to locate. Two sources which I will discuss are magnetic fields and electromagnetic interference (EMI).

If you are operating in a environment where permanent magnets and/or electromagnets are present, keep the computer and the interface as far from the magnetic fields as possible. Sometimes however, devices which generate magnetic fields must be mounted on the interface board itself (in the form of a relay, solenoid, motor, etc.). In this case, mount the device as far from circuit wiring as possible. The problem is that if the magnetic field fluctuates or if a wire is allowed to move in the magnetic field, a current will be induced into the circuit. This current is an unwanted signal, and can cause problems, especially if there are analog devices on the board. Moving electromagnetic devices off the interface board is the best technique for eliminating this noise problem. If permanent magnets are being used in the vicinity of the interface circuit, ensure that wires are not vibrating or moving near the magnetic fields.

Electromagnetic interference (EMI) can originate from radio transmitters, extremely high current switching devices, motors, gas discharge devices, etc. This energy will travel through air, it will ride on power lines, and it can be received by anything that acts like an antenna. This type of noise problem is usually found in an industrial environment or possibly next to a radio or TV station, but it is a good idea to be aware that EMI problems do exist, and know a few techniques for combatting them.

EMI will most easily enter a circuit through long wires that act as an antenna. Long power lines fall into this category, as do lines to/from an interface which monitor or control external devices. One simple and effective way to combat EMI is to run the signal lines and/or the power lines in a twisted pair. A twisted pair of wires in a varying magnetic field will have induced voltages in successive twists that oppose each other. The net interference effect in the cable will be the sum of the individual induced voltages which should be about zero. When using a twisted pair line never ground both ends of the line. Figure 4 illustrates the proper use of a twisted pair line. Grounding both ends of a twisted pair line can set up grounding loops which may cause worse problems than the interference from the EMI.

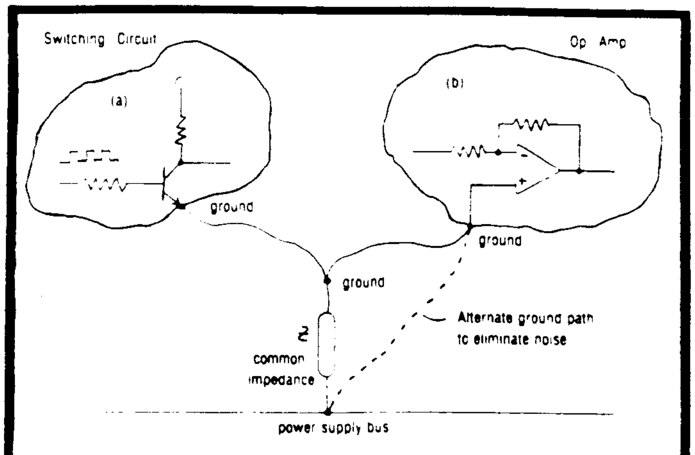


Figure 3: Common Impedance

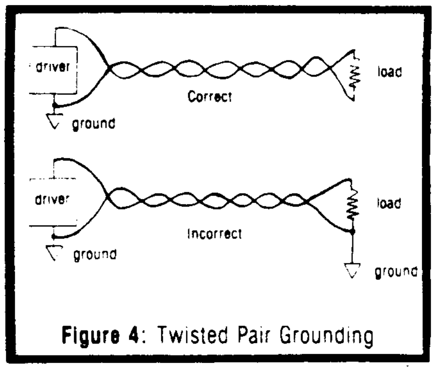


Figure 4: Twisted Pair Grounding

Another noise reduction technique which is extremely effective in reducing EMI is shielding. Shielding is a subject addressed by entire books, therefore, I will not attempt to cover thv subject in great detail. Instead, I will give an example of where shielding might help eliminate EMI, and will give a list of do's and don'ts concerning shielding techniques.

Low level analog signal measurement is one area where shielding techniques are used regularly. For instance, you may want to measure temperature at a location which is several feet from the measurement interface. The detector may output millivolt level signals which must be sent, via a cable, to an operational amplifier on the interface board. To eliminate the possibly of EMI interfering with the detector signal, you can shield the signal line. Figure 5 illustrates this measurement scheme. The following list of suggestions will aid you in applying shielding techniques in situations like the one shown in Figure 5:

- Do connect the shield to the reference potential of the signal contained within the shield. This is illustrated in Figure 6a. Note: Grounding the shield is useless if the signal is not referenced to ground.
- Do connect the shield conductor of the shielded cable directly to the reference potential at the signal-reference node. This principle is also illustrated in Figure 6a. Figure 6b illustrates a shield not connected at the signal reference node.
- Don't run more than one signal line in each shield. If more than one signal line must be sent to the interface, a shield for each line is required.
- Don't connect both ends of the shield to ground. Grounding both ends of a shield can generate ground loop currents in the shield which may be induced into the signal line.
- Don't allow a shield current to exist. This can be insured by grounding the shield at only one point.
- Don't allow the shield to be at a voltage with respect to the reference potential. Any potential on the shield may be capacitively coupled to the signal line.

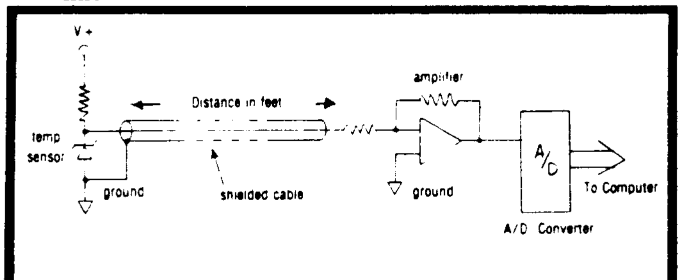


Figure 5: Remote Measurement Scheme

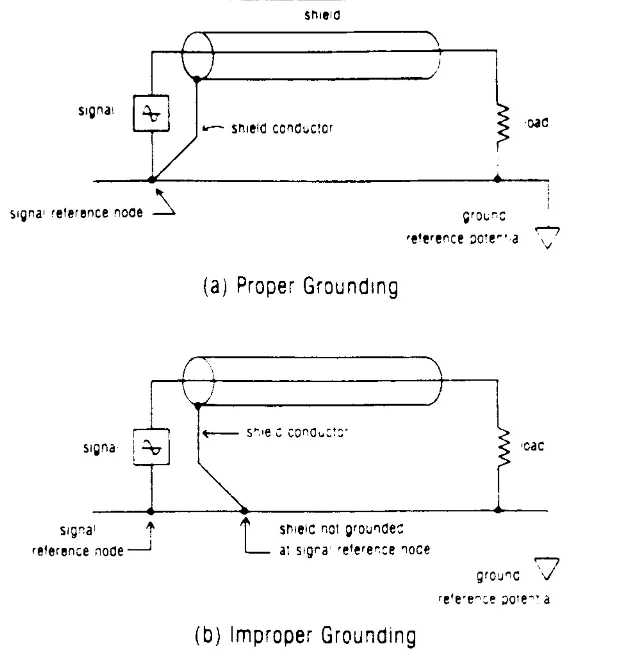


Figure 6: Shield Grounding

Noise Reduction Using Ferrite Beads.

In reviewing the schematics of several popular computers, I noticed the use of ferrite beads on the power supply inputs of several ICs, on the plus and minus 15 volt power supply lines of several OP amps, on cables extending from the CPU board to the keyboard, and on supply lines connecting front panel LEDs. My curiosity was aroused concerning the use of ferrite beads, so I did a little research and acquired some interesting information that I will share with you. Ferrite beads act as non-reactive energy absorbers. This is a rather unique characteristic in that the other electrical elements used for absorbing energy in a circuit are reactive. For instance, inductors will absorb energy from a circuit. However, the energy is actually stored as a magnetic field, and you can bet that you will get a large percentage of the stored energy back when the field collapses. The same is true for capacitors. A capacitor will extract energy from a circuit, but the energy is actually stored as an electric field on the plates of the capacitor. A ferrite bead, on the other hand, absorbs energy from a wire through inductive coupling and does not return it to the line. This makes the ferrite bead a perfect noise suppression element. These beads can be purchased from Mouser Electronics (11433 Woodside Ave. Santee, California 92071), and they are extremely easy to use. Just slide the bead over a wire before connecting the wire to your circuit and the bead is in place. If anyone knows a lot about the use of ferrite beads, or knows of literature that address this subject, drop me a line and I'll share the information with our readers.

A Noise Reduction Bedtime Story.

In the first part of this noise reduction series I promised you a story which I thought that you might find amusing.

It's a short story, but it's a story with a moral. I was employed by a research lab to build an amplifier to boost the signals from an experimental solar cell. The job wasn't bad; the signals were DC and I was using a monolithic instrumentation amplifier to accomplish the task. To simulate the sun's frequency spectrum we were using a huge Zenon light source and focusing the light down on the solar cell. I finished the amp and tested it by applying the same DC voltage levels that the solar cell would produce. Everything worked fine until I aligned the Zenon lamp and switched it on. ZAP—my amp went up in smoke (at 25 dollars apiece)! At first I thought that I had overdriven the amplifier inputs. So I decided to turn the light source away from the solar cell, turn it on, and move it onto the cell slowly as I watched the voltage level on the input of the amp. I installed a new instrumentation amp, rotated the light source and switched it on. ZAP—more smoke! Now this is serious noise!! Needless to say, I was irritated. I installed isolation transformers on everything, incorporated grounding and shielding techniques, decoupled my amps, and moved the light source to the other end of the room. All of these precautions did not save me from yet another smoking amplifier, but while staring at my third non-operational amplifier the solution came to me. Turn the amplifier circuit on *after* the Zenon lamp! Now I'm not particularly proud of the sophistication of my solution, but it works—and after all, that's the bottom line. The moral is: sometimes avoiding noise is a better solution than eliminating it.

Good luck solving (or avoiding) your noise problems. I hope that this series on noise reduction will be an aid to you in combatting noise problems. I have concluded this article with a noise reduction checklist which may prove helpful if you encounter noise problems in your interface.

Checklist for Reducing Noise Problems

The Power Supply:
- ☐ Keep power supply lines short.
- ☐ Route supply lines around oscillator and power switching circuits.
- ☐ Never "daisy chain" supply lines.

If noise persists:
- ☐ Be sure that you are using an adquate supply.
- ☐ Check supply for adequate filtering capacitors.
- ☐ Check for oscillating voltage regulators.
- ☐ Place ferrite beads on supply lines going to individual circuit elements.

The Connecting Cables:
- ☐ Limit cable length to one meter.
- ☐ Physically separate transmission lines according to the types of signals that they carry.
- ☐ Place a grounded line between each signal line in the cable or use a cable with a grounding plane incorporated.
- ☐ Terminate all cable lines.
- ☐ Use 74L series ICs as cable line buffers.

The Interface Circuit:
- ☐ Separate oscillator, power switching, and logic sections on the project board.
- ☐ Use decoupling capacitors liberally.
- ☐ Use spike supression on EVERY inductive circuit element.
- ☐ Always ground circuit sections to a single common point.
- ☐ Keep wires and circuits away from magnetic fields.

Outside Sources:
- ☐ Always run two conductor lines in a twisted pair.
- ☐ If twisted pair lines do not eliminate noise, use shielding techniques.
- ☐ Place a ferrite bead on lines going to remote circuit elements. ■

Editor's Page, continued

business office program for wide distribution you had better make sure that it runs on an IBM PC.

We have to think of our readers' needs when preparing articles, and the large number of systems and languages in current use makes it difficult for the editors at *The Computer Journal* to choose the language which will be of interest to the largest number of readers. We want to concentrate on covering the areas of interfacing, measurement, control, robotics, new devices, etc. without becoming computer or language specific, and will try to use assembly code where it is indicated (such as the monitor for the 68008 board in the last issue), or one of the more popular languages in other cases.

Unfortunately, the authors and the editors cannot become proficient in all the languages so that we can provide every program in several languages. In order to alleviate this problem we encourage our readers to share any routines which they have rewritten for their own use. We would also be interested in publishing comparisons of code size and run time between different languages.

The Computer Journal's main emphasis is on using computers rather than writing large involved business oriented programs, but our readers are interested in learning about assembly language, utility program design, program debugging, operating system design and modification, programming for new CPU's such as the 68000, control oriented languages such as FORTH, and how to choose and use a language for a certain application. We need more input from the people in the field telling about their experiences with actual applications. Send us a short article or outline on why you feel your favorite language is the one they should use. ■

FLOPPY DRIVE EXERCISER!

ALIGN DRIVE IN 10 MINUTES!
Use with scope and alignment disk (SS $49, DS $75)

- SINGLE KEYSTROKE FOR ALL ALIGNMENT TRACKS
- JOG KEYS-MOVE TO ANY TRACK
- INCLUDES "OSBORNE" TYPE POWER HOOKUP
- RUNS ANY STANDARD 34 PIN (5") OR 50 PIN (8") DRIVE
- SHOWS SPEED AND SPEED AVERAGE!
- HYSTERESIS CHECK BUILT IN
- SELECT 5" 48, 96, 100 TPI, OR 8" 48, TPI
- POWER "Y" CABLE=$10 DRIVE DATA CABLE=$20

USED BY: IBM, ARMY, NAVY, RCA, ETC...

EX 2000 **$299**

FREE Air Freight on Prepaid Orders. COD: Add $5 Plus Shipping

PROTO PC inc. **CALL NOW!** **612-644-4660**
2439 Franklin, St. Paul, MN 55114

Beginner's Project: 555 Timer Breadboard

by Art Carlson

Most measurement and control applications of microcomputers involve some use of a timing signal, and the 555 timer chip is frequently used to generate these signals. This simple breadboard circuit demonstrates how easy it is to use this chip.

Prior to the development of the 555 integrated circuit timer chip in the early 1970s, building even a simple timer circuit involved working with dozens of discrete components (transistors, capacitors, resistors, etc.) along with their associated sockets, wires, and connections. This took a lot of time and space. Now you can make a timer circuit with just the one chip, a capacitor, and two resistors. The 555 can be used in a wide variety of ways, including monostable or astable multivibrators operating over a frequency range of microseconds to hours. For this project we will use it as a free-running astable multivibrator.

This is a simple breadboarding project for first time builders. It is intended to provide some experience in building prototype circuits and in using the 555 timer, the 74LS90 BCD counter, the 74LS47 BCD to seven segment decoder/driver, and common anode LED display chips. The circuit in Figure 1 was constructed on a Radio Shack breadboard as shown in Figure 2. These breadboards are ideal for experimental circuits because you can make changes and salvage the components without any soldering or damage to the parts. All of the material used in this example was purchased from Radio Shack, except for the wire and the potentiometer, which came from the junk box. These are common, easy-to-get parts which can be obtained from Jameco, Digi-key, and many other sources. I purchased these from Radio Shack because we are in a remote area and Radio Shack is the only local source where we can pick them up without the delay and shipping expense involved in mail order. A parts list is found in Table 1.

Circuit Description

The 555 is connected as a free-running astable multivibrator. This means that it is not stable in either state, and keeps changing back and forth, giving us a square wave output. A monostable circuit would be stable in one state, would switch when it received a trigger signal, and would then return to the stable state until another trigger was received. The frequency for the circuit we are using is determined by the values of R1, R2, and C1, according to the formula in Figure 3.

The values of R1, R2, and C1 in Figure 1 were chosen to produce a one count per second display on the LED, and we used a potentiometer to adjust the value of R2 to obtain the

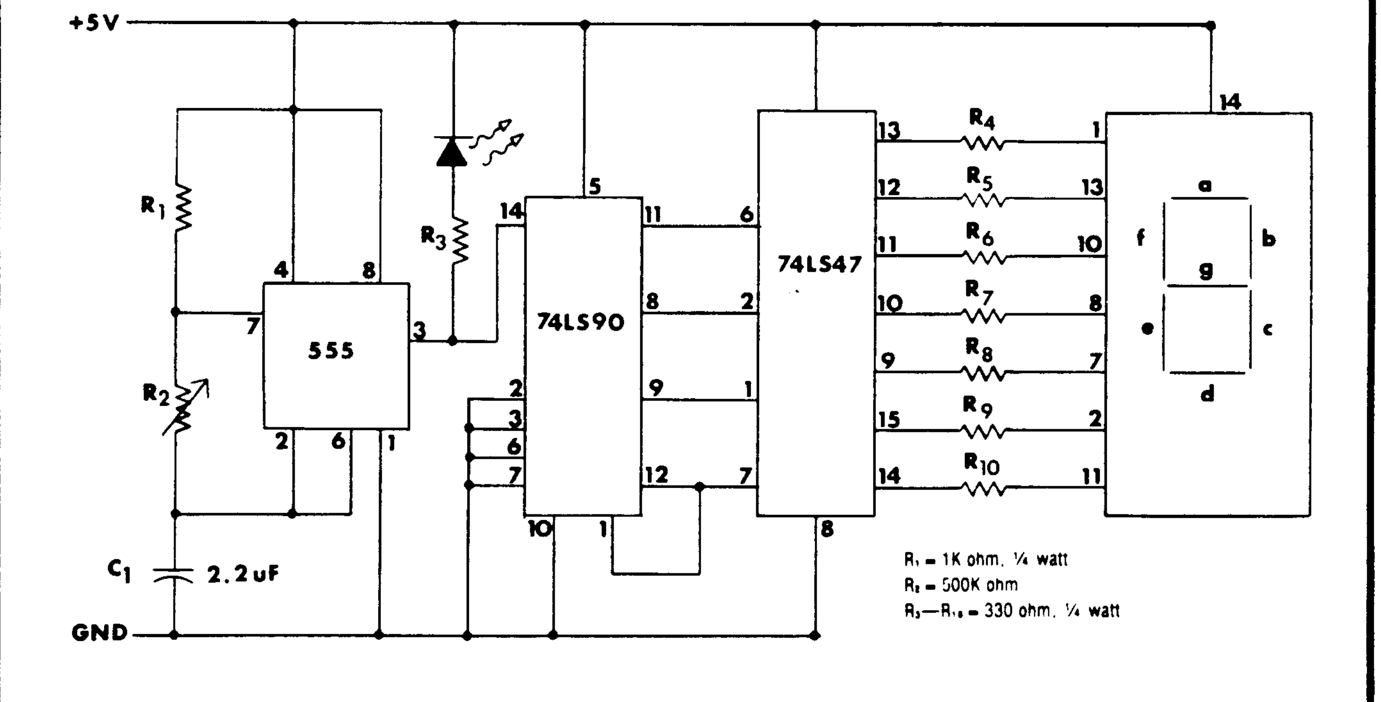


Figure 1: Schematic

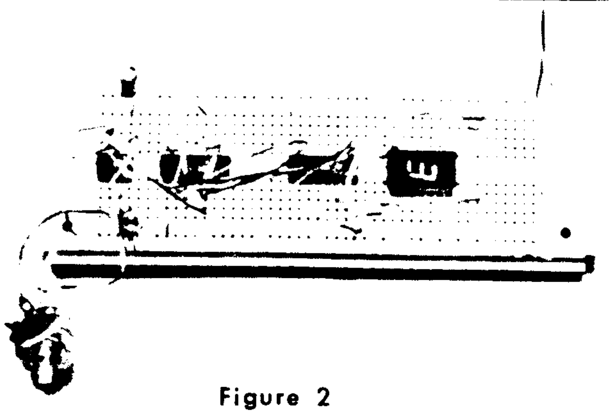


Figure 2

correct frequency because the parts are not exactly the nominal value. I have also included an LED on the output of the 555 to monitor its operation. Note that an LED requires a current limiting resistor because the LED is a diode which would pass enough current to destroy itself when it conducts. An incandescent lamp of the proper voltage could be connected at this point without requiring a current limiting resistor.

The output from pin three of the 555 is connected to pin 14 of the 74LS90. This is the clock 1 input of a divide by two counter. The output from this section on pin 12 is connected to pin 1, which is the clock input for the divide by 5 section. The standard 8-4-2-1 weighted BCD (Binary Coded Decimal) output is available from the counter on pins 12, 9, 8, and 11, with the least significant count on pin 12, the second bit on pin 9, the third bit on pin 8, and the fourth bit on pin 11. See the "BCD Math" sidebar if you are not familiar with the BCD method for encoding a decimal number from 0 through 9 in a four bit binary code. Note that the zero set pins 2 and 3 and the nine set pins 6 and 7 must be grounded.

The outputs from the 74LS90 are connected to the inputs of the 74LS47 with the least significant bit to pin 7, the second bit to pin 1, the third bit to pin 2, and the fourth bit to pin 6. The 74LS47 converts the BCD input to the proper outputs required by the LED seven segment display as shown in Table 2. Note that since this display is an LED, a series resistor is required on each segment, just as for the LED connected to the 555 output.

Part Description	Radio Shack #	Price
Breadboard	276-174	$11.95
555	276-1723	$1.19
74LS90	276-1808	$1.09
74LS47	276-1805	$1.59
7 segment LED display	276-053	$1.79
LED	276-021	2/$.79
500K Potentiometer	271-221	$.59
2.2uF Capacitor	272-997	$.79
330 ohm, ¼ watt Resistor	271-1315	5/$.39
1K ohm, ¼ watt Resistor	271-1321	5/$.39

Total Price $20.10

Table 1: Parts List

$$\text{Total Time (seconds)} = 0.693\,(R_1 + 2R_2)\,C$$

$$\text{Frequency} = \frac{1.443}{(R_1 + 2R_2)C}$$

$$\text{Duty Cycle} = \frac{R_1 + R_2}{R_1 + 2R_2}$$

In this particular application we have made R_2 large with respect to R_1 in order to obtain a square wave with a duty cycle of almost 50%, although the counter triggers on a short pulse and it would work with a much lower duty cycle.

Figure 3: 555 Timer Design Formulas.

Construction

Building this circuit using the solderless breadboard is simple. All you have to do is to cut and strip the wire and plug the parts into the board. You should not have much difficulty even if you have never built anything before! Follow the next few steps if this is your first project, otherwise skip this section.

Figure 2 will give you an idea of where to place the parts, but you don't have to follow the exact layout. I just started with the 555 section and it grew from there. You may notice that it does not look as neat as the illustrations you usually see in magazines. That's because this is the way it turned out when built as a test project, and I didn't rebuild it for the photo.

Start by inserting the 555 into the board. If you have never used a chip, there is a notch or circle on the chip near pin one, and the pins are numbered counterclockwise looking at the top of the chip (as shown in the pinouts of these chips in the Reference Chart). Add the capacitor, the three resistors, and the LED, then use short pieces of 24 gauge wire with ¼" of the insulation stripped from each end to make the connections. It is helpful to make a photocopy of the circuit and mark each connection with a colored pencil as it is completed. Don't connect pin 3 to the 74LS90 yet. Connect a five volt power supply with the polarity as shown, and the LED should start flashing. If the LED does not flash, check all of the connnections, especially the polarity of the LED (you do have a visible light LED and not an infrared LED, don't you?). If you can't find anything wrong, connect a 330 ohm resistor in series with the LED and try it across the five volt supply to be sure that you have the right polarity and that the LED really does work.

After you have the 555 section working you can disconnect the power supply and complete the rest of the circuit. This circuit is designed for a common anode display, but common cathode displays are also available, so check carefully if you have substituted a different part number.

BCD Math

Computers use binary numbers, but humans prefer to enter information such as dates, time, or dollars and cents in the decimal numbers which we normally use. BCD (Binary Coded Decimal) is one method of encoding decimal numbers in binary fashion. BCD is similar to hexadecimal except that only the digits 0 through 9 are used for BCD instead of adding the letters A through F as in hexadecimal. BCD uses more memory than hexadecimal because four bits in BCD can only represent a maximum value of nine, while four bits in hexadecimal can represent a maximum value of 15. BCD is very useful when we want to read or display decimal information, because four bits always equals one decimal digit. BCD numbers can easily be converted to decimal by treating groups of four digits as shown in the following example:

8	4	2	1
0	1	1	1

4 + 2 + 1 = 7

In BCD, the first digit has a value of 8, the second has a value of 4, the third a value of 2, and the fourth, 1. Hence, the decimal number 7 would be represented by the BCD number 0111.

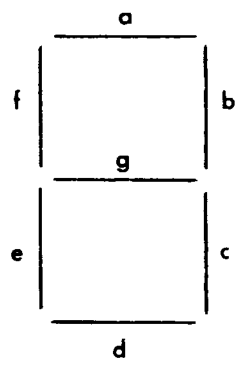
If binary representation of decimal numbers is new to you, practice converting the decimal numbers 1 through 9 to BCD and compare your results with the data in Table 2.

Now for the real test! When you connect the power supply, the seven segment display should count up to nine, then roll over and repeat. If there is nothing on the display, check to see if the LED on the 555 is flashing (it should be, because you made sure that it was working earlier). Assuming that the LED is flashing, go back over the circuit diagram and check off the connections with a different colored pencil. If the LED flashes, the display remains black, and you can't find any errors on the breadboard, you get the chance to do some troubleshooting. Sometimes I find it helpful to go do something else for a while and recheck it again later.

In order to troubleshoot the circuit you will have to be able to check various points to see if they are high or low. We usually use a logic probe for this type of testing, but you can use a voltmeter or simple LED indicator on this circuit. You can make an indicator by connecting a 330 ohm resistor to the cathode of an LED. Connect the other end of the resistor to ground and touch the anode lead of the LED to a connection to see if it is high or low. This indicator is not equivalent to a regular logic probe which can detect high speed pulses, but it will work for low speed or static circuits such as this.

You know that the 555 is working because the LED on pin 3 is flashing, so you can start there to double check your probe and follow the signal through the circuit to find the trouble. The truth tables in Table 2 will help you determine which pins should be high at any point in the count. After the circuit is working you can adjust R2 so that the count is approximately one per second.

DEC.	BCD Equivalent				Coding scheme for 7-segment display						
					a	b	c	d	e	f	g
0	L	L	L	L	L	L	L	L	L	L	H
1	L	L	L	H	H	L	L	H	H	H	H
2	L	L	H	L	L	L	H	L	L	H	L
3	L	L	H	H	L	L	L	L	H	H	L
4	L	H	L	L	H	L	L	H	H	L	L
5	L	H	L	H	L	H	L	L	H	L	L
6	L	H	H	L	L	L	H	H	H	H	H
7	L	H	H	H	L	L	L	H	H	H	H
8	H	L	L	L	L	L	L	L	L	L	L
9	H	L	L	H	L	L	L	H	H	L	L



These truth tables will help you understand the relationships between the signals from the 74LS90 counter, and the 74LS47 seven segment decoder. The 555 produces pulses, which are received by the 74LS90 counter. It outputs the BCD number to the 74LS47, which decodes the signals to drive the LED display.

With the counter at zero, the four BCD outputs from the counter are all low, and the decoder sets LED segments a through f low and g high to display a zero. When the counter recieves the first pulse it outputs the binary number 0001 and the decoder sets b and c low to display the number one.

Table 2

Going Further

This simple project was intended to encourage you to start building hardware projects, and to familiarize you with some basic chips. The zero to nine second counter isn't very useful, but we will use it as a building block for more involved projects. In the next step we will add more digits to the output so that we can count to larger numbers, and use an AND gate between the 555 and the 74LS90 so that we can count the number of pulses from another circuit which occur in one second. Why don't you work ahead and try to develop this on your own before we publish the next article? ■

For Further Reading: **The 555 Timer Applications Sourcebook, With Experiments,** by Howard M. Berlin. Pub. by Howard W. Sams, 1982.

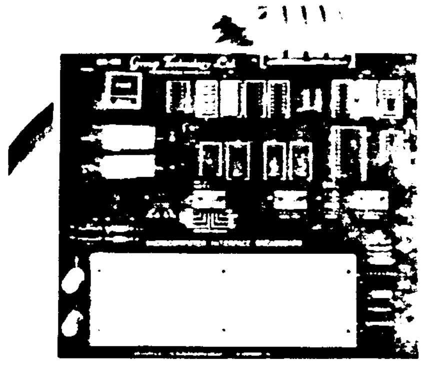
LEARN MICROCOMPUTER INTERFACING
VISUALIZE SCIENCE PRINCIPLES

Using GROUP TECHNOLOGY BREADBOARDS with your
APPLE® ...COMMODORE 64® ...TRS-80® ...TIMEX-SINCLAIR® ...VIC-20®

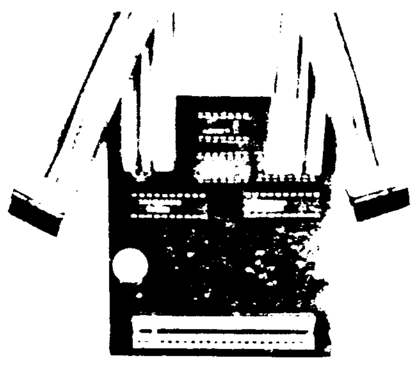
Versatile breadboards and clearly written texts with detailed experiments provide basic instruction in interfacing microcomputers to external devices for control and information exchange. They can be used to provide vivid illustrations of science principles or to design interface circuits for specific applications. Fully buffered address, data, and control buses assure safe access to decoded addresses. Signals brought out to the breadboards let you see how microcomputer signals flow and how they can be used under BASIC program control to accomplish many useful tasks.

Texts for these breadboards have been written by experienced scientists and instructors well-versed in conveying ideas clearly and simply. They proceed step-by-step from initial concepts to advanced constructions and are equally useful for classroom or individual instruction. No previous knowledge of electronics is assumed, but the ability to program in BASIC is important.

The breadboards are available as kits or assembled. Experiment component packages include most of the parts needed to do the experiments in the books. Connecting cables and other accessory and design aids available make for additional convenience in applying the boards for classroom and circuit design objectives. Breadboard prices range from $34.95 to $350.00



The INNOVATOR® BG-Boards designed by the producers of the highly acclaimed Blacksburg Series of books have gained wide acceptance for teaching microcomputer interfacing as well as for industrial and personal applications. Detailed, step-by-step instructions guide the user from the construction of device address decoders and input/output ports to the generation of voltage and current signals for controlling servo motors and driving high-current, high-voltage loads. BG-Boards are available for the Apple II, II+, IIe; Commodore 64 and VIC-20; TRS-80 Model 1 with Level II BASIC and at least 4K read/write memory, Models III and 4. The books, *Apple Interfacing* (No. 21862) and *TRS-80 Interfacing Books 1 and 2* (21633, 21739) are available separately.



The FD-ZX1 I/O board provides access to the Timex-Sinclair microcomputer for use in automated measurement, data acquisition, and instrument control applications. A number of science experiments have been developed to aid teachers in illustrating scientific principles. The operating manual contains instructions for constructing input and output ports. A complete text of the experiments will be available later in 1984. The FD-ZX1 can be used with Models 1000, 1500, 2068, ZX81, and Spectrum.

The Color Computer Expansion Connector Breadboard (not shown) for the **TRS-80 Color Computer** makes it possible to connect external devices to the expansion connector signals of the computer. Combined with a solderless breadboard and the book *TRS-80 Color Computer Interfacing, With Experiments* (No. 21893), it forms our Model CoCo-100 Interface Breadboard providing basic interfacing instructions for this versatile computer. Experiments in the book show how to construct and use a peripheral interface adapter interface, how to input and output data, and how digital-to-analog and analog-to-digital conversion is performed.

Our new Spring Catalog describes the interface breadboards, dozens of books on microcomputer interfacing, programming, and related topics including the famous Blacksburg Continuing Education Series, a resource handbook for microcomputers in education, and a comprehensive guide to educational software; utility software for the TRS-80, scientific software for the Apple II, and other topics. We give special discounts to educational institutions and instructors. Write for the catalog today.

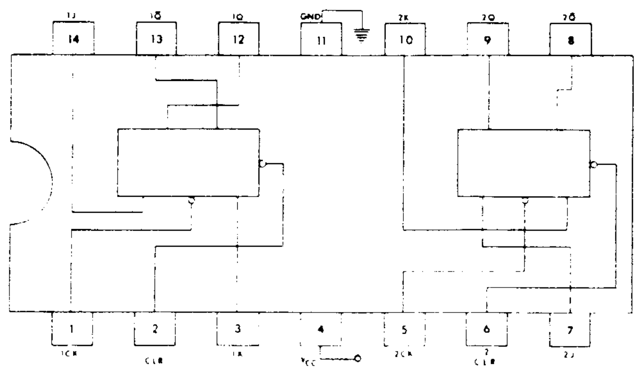
Apple II, II+, and IIe are registered trademarks of Apple Computer Inc., Commodore 64 and VIC-20 are registered trademarks of Commodore Business Machines; TRS-80 is a registered trademark of Radio Shack, a Tandy Corporation; Timex/Sinclair is a registered trademark of Timex Computer Corporation

PUTTING HANDS AND MINDS TOGETHER

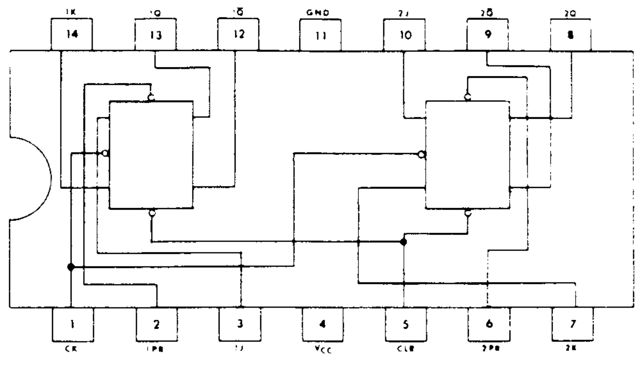
Group Technology, Ltd.
P.O. Box 87N
Check, VA 24072
703-651-3153

LSTTL Reference Chart

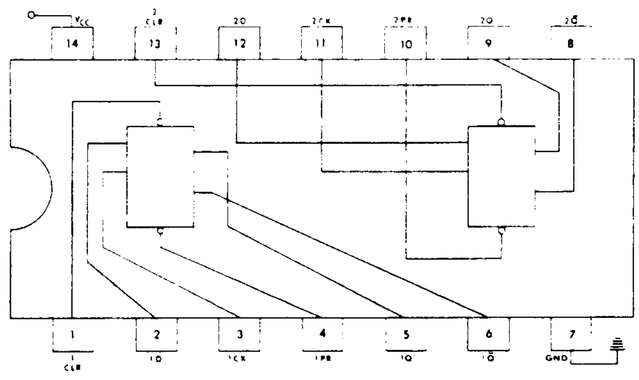
74LS73
Dual JK Negative Edge Triggered Flip-Flop



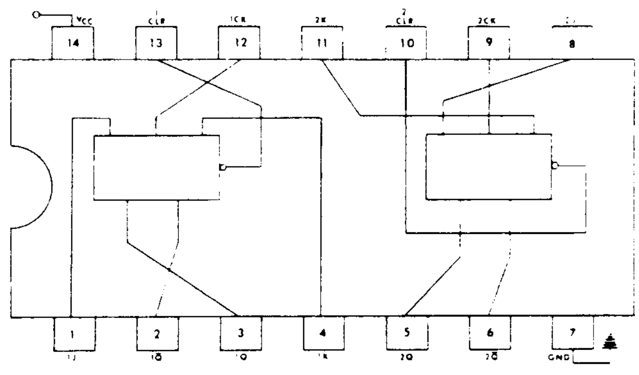
74LS78
Dual JK Negative Edge Triggered Flip-Flop



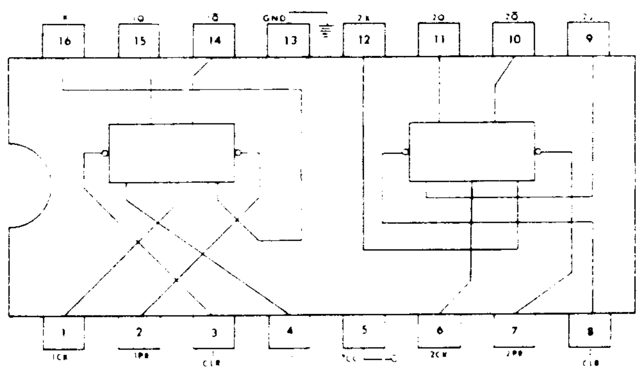
74LS74
Dual D Positive Edge Triggered Flip-Flop



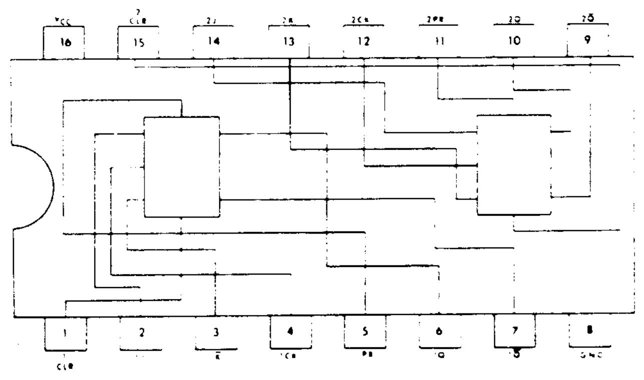
74LS107
Dual JK Negative Edge Triggered Flip-Flop



74LS76
Dual JK Negative Edge Triggered Flip-Flop

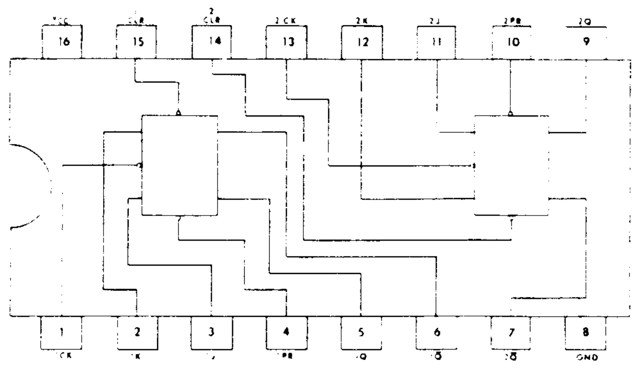


74LS109
Dual JK Positive Edge Triggered Flip-Flop

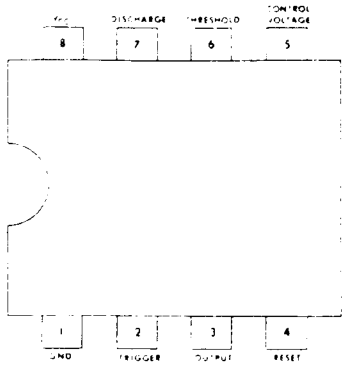


74LS112

Dual JK Negative Edge Triggered Flip-Flop

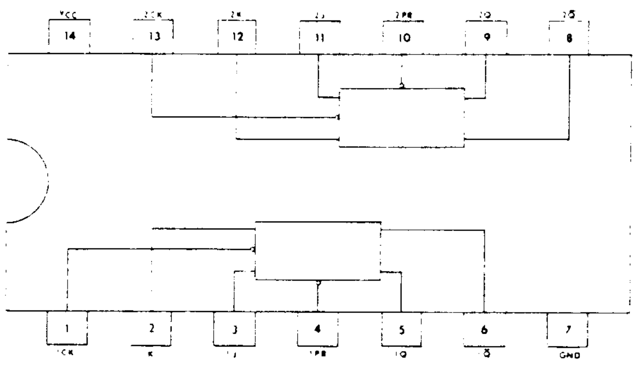


555 Timer



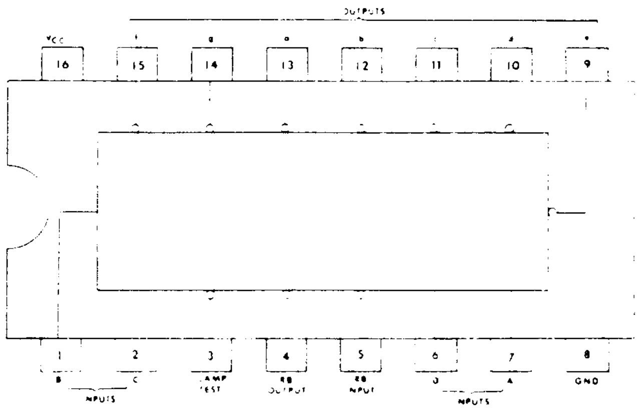
74LS113

Dual JK Negative Edge Triggered Flip-Flop



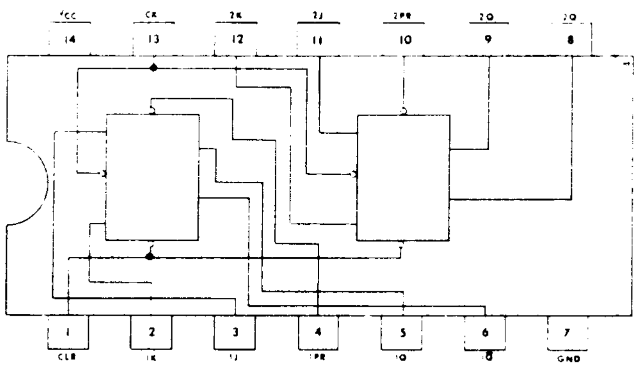
74LS47

BCD to 7-segment Decoder/Driver



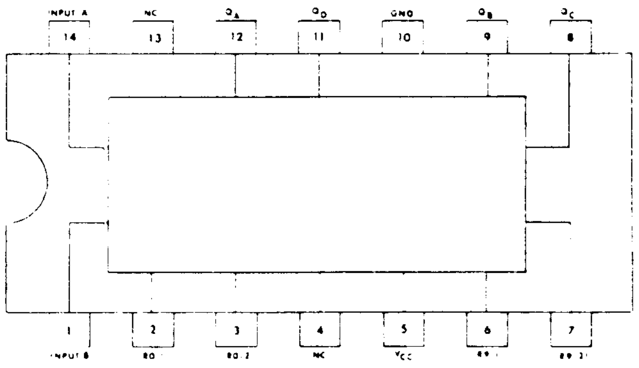
74LS114

Dual JK Negative Edge Triggered Flip-Flop



74LS90

Divide by 2 or 5, BCD Counter



Multi-user

A Column by E.G. Brooner

In previous editions of this column we have discussed a variety of multi-user concepts such as time sharing, networking, and multi-processors. A few particular systems have been described to illustrate the various ways in which several users can be tied together, either for communication or for resource sharing.

One aspect which has been relatively untouched in this column is that of the physical connection between the devices. We have just assumed that some sort of wiring exists and that it is adequate for the purpose. There is a little more to it than that.

There are many ways to wire a conglomeration of computer equipment. One of the terms that must be understood is *topology*, which describes the physical layout of the wiring. The topology depends a lot on the kind of multi-user installation that is involved, as well as what kind of devices and interfaces are to be used.

In spite of the different names that are used, network and other multi-user topology falls into three general categories. These are *bus*, *star*, and *ring*. A bus arrangement connects everything essentially in paralllel and might be compared, in some ways, to electrical wiring or a group of phone extensions. Bus topology is the easiest to understand and implement. With this kind of arrangement the failure of one device does not usually disrupt the remaining operation.

Star connections involve separate cabling between some central device (such as the computer in a time sharing system) and each user device. The distinction of a star system is that each individual cable only serves one user, and all of them terminate at a common point which exercises some control over them. The failure of the central device can put the entire system out of service.

Ring configurations describe a circle with user points spaced along the periphery. Some rings are just a bus tied end-to-end, while others place the devices in series so that the complete circuit can be interrupted at any device. Rings can be thought of as a hybrid between buses and stars, with some of the advantages and disadvantages of both.

There are special uses and reasons for each of the many forms of topology. The configuration for any given network is predetermined by the system design and manufacturer, and is thus a limit that cannot easily be changed.

There are also a multitude of cables that might be used for the various purposes. We are all familiar with most of them in some form; there are ribbon cables such as those that connect our disk drives and are sometimes used with parallel printers; coaxial cables, twisted pair cables, and last but not least, those used with RS-232 devices. RS-232 is actually a 21 wire interface, but we sometimes implement it (or part of it) with as few as three wires. Since there are not many three wire cables around, we usually end up using two pair telephone cable or a ten conductor cable.

Ribbon cables are used mainly with parallel devices which require a large number of connections. We don't generally run them very far for some obvious reasons—loss and noise pickup are two of them. Cables with fewer wires in them are more reasonably priced, less cumbersome, and can usually be run farther without problems; in the case of RS-232 the specs say 50 feet but a couple hundred may be practical. The RS-449 convention used for very similar purposes has an even longer range. RS-449 permits a trade-off between speed and distance: high speed over short distances, and slower performance for up to several hundred or thousand feet.

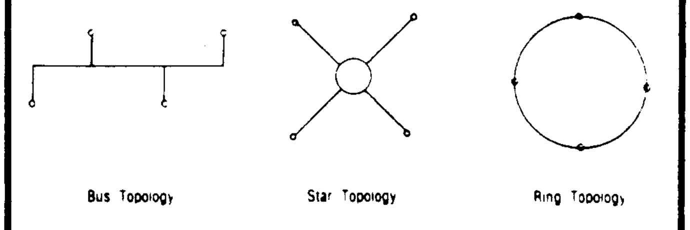
Coaxial cables can be installed to cover thousands of yards, but there usually has to be some kind of interface to get the signals on and off the cable, and some cost is always involved. The coax itself may also be more expensive than some simpler kind of wiring. Fiber optics provide an even higher quality, and more expensive, wiring alternative.

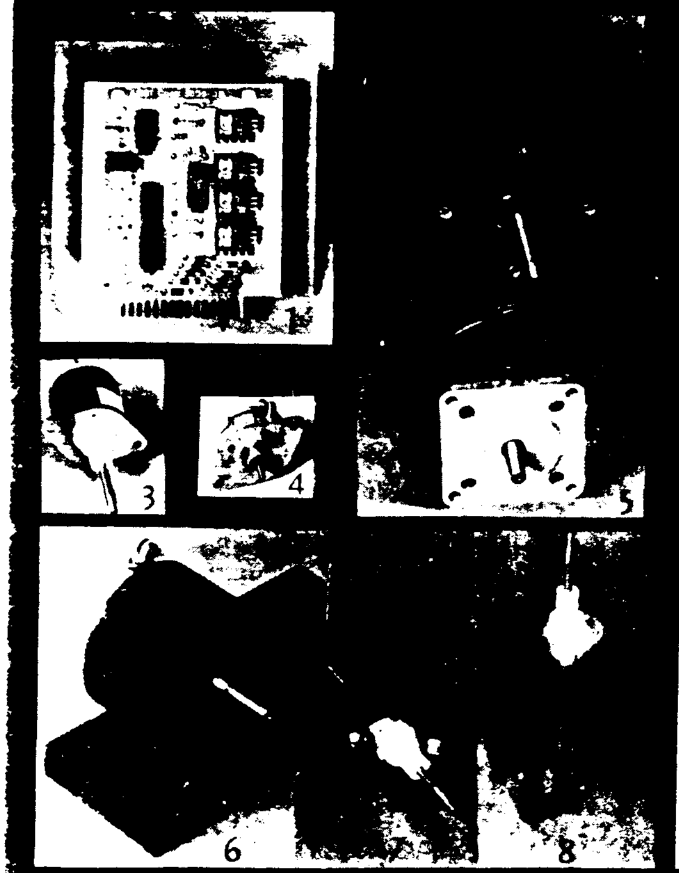
Low-end systems most often employ some derivation of the RS-232 or 449 systems, often using a simple shielded twisted pair cable for the purpose. (If one conductor is grounded, a shielded pair can sometimes be considered a three-wire cable.) The higher cost, higher speed, longer distance networks use coaxial cable. We can see, then, that we need to have the final version of our system fairly well worked out before we start building cables into the wall.

We generally find some version of the RS-232 cable used with time sharing systems and multiprocessors, twisted pair for the cheaper LANs, and coaxial cable (or fiber optic paths) where high-performance networks are involved.

One of the most interesting cabling/interface systems, from the viewpoint of the interfacer, is the IEEE-488 bus. Unfortunately, it is seldom used in business or home computing; it was designed for control purposes and might actually be the best available system for the tinkerer. It is much used by industry, and at one time Commodore made an attempt to adapt it to their micros. One of their early models drove both the disk system and the printer via IEEE-488.

The 488 bus is a parallel system (many wires involved)





COMPUTER CONTROLLED ROBOTICS

1. DRIVER BOARD 5005 DB $75 *
4.5" x 3.8" x 0.5", TTL CMOS COMPATIBLE.
OPTICALLY INSOLATED FOR 4 PHASE MOTORS 2AMPS 50 VOLTS
2. LINEAR ACTUATOR 601 AM $75
12V 12W, 16 OZ. .001" STEP SIZE
19 LBS HOLDING FORCE, 3 IN TRAVEL
3. LINEAR ACTUATOR 501 AM $43
12V 3.5W, 1.5 OZ. .002" STEP SIZE
40 OZ HOLDING FORCE, 1.88 IN TRAVEL
4. STEPPER MOTOR 201 SM $16
5V 2W, 1.0 OZ. 15° STEP SIZE
0.8 OZ IN HOLDING TORQUE
5. STEPPER MOTOR 301 SM $59
12V, 21.5 OZ, 1.8° STEP SIZE
80 OZ IN HOLDING TORQUE
6. MOTOR MOUNT FOR 301 SM $25
7. MOTOR MOUNT FOR 501 AM $12
8. MOTOR MOUNT FOR 501 AM $13

* EDGE CONNECTOR $3.50

AMSI CORP. (516) 361-9499
BOX 651, SMITHTOWN, L.I., N.Y. 11787

with some unusual features. Take topology: there is one and only one kind of connector and cable. Each connector is both male and female. This means that if you have two or more devices to connect you still need only one port on the central device, i.e. your computer. The second cable plugs into the back of one of the existing connectors. The third plugs into any available position—there is always room for one more. Each time you add another cable you automatically add the socket for the next one!

If you connect several devices "end-to-end" it looks like a typical bus topology. You could as easily connect devices two through four, or five, or whatever, directly to the first device by "piggy-backing" the connectors, in which case the layout would resemble a star configuration. Electrically, they will still all be in parallel. The topology then is "bus" even though it might physically look like a star or ring. You might have connected, on the same port, a printer and disk drive and several sensors, control devices, or items of test equipment. Theoretically anything that functions with 8-bit data can be accommodated by the system, and can be connected to the same port.

You can forget things like baud rate; data can be exchanged over the same bus by different kinds of devices and at different rates of speed. The 488 is really a sophisticated communication system which is under control of the central computer, which "knows" the address of each device and its capabilities, and controls their input and output accordingly. There is a common data bus and a control bus that knows how to access and control all connected devices.

We don't often encounter this system in micros because its initial cost is high, compared with a simple Centronics or RS-232 port. There are, though, plug-in 488 controller boards available for the S-100 machines. (Pickles & Trout, Goleta, CA., or D&W Digital, Hathaway, CA.)

With IEEE-488 a central device (usually a computer of some kind) acts as system controller. The other devices can, to use the jargon, "talk," "listen," or both, according to how they have been configured. There is a complete handshaking system between the controller and each talker or listener. Data to or from a high speed device, or a low speed device, passes over the common data bus at the rate that has been determined for each. If data is sent to more than one device at a time the slowest one effectively controls the rate—thus devices of different speeds can be accommodated on the same bus.

The bulk of the cable, as in the case of the ribbon cables we are familiar with, imposes a practical limit on the distance and the configuration.

Cabling is an important part of any multi-user system. The kind of cable used, the topology, the cost, and all the other limitations are determined at the time the system is designed. Once installed it becomes practically invisible and trouble free, but it is nevertheless one of the important ingredients. ■

For More information, see the following books:

Computer Communication Techniques, Howard W. Sams, #21998

The Local Area Network Book, Howard W. Sams, #22254

WRITE YOUR OWN THREADED LANGUAGE

Part Four: Conclusion

by Douglas Davidson

The last article finished the presentation of the primaries. All that is needed now to get the language up and running is a few secondaries. We will be able to define most of the secondaries by using the facilities of the language itself, but certain of the most important secondaries must be hand-compiled; once they have been defined, the language will support its own growth. Enough information has already been presented about the coding of secondaries to write all of those that must be hand-compiled. The high-level definitions of most of these are given in Figure 1. Coding from a high-level definition is pretty straightforward: most words are represented simply by calls to them, while the special constructions presented in the last article are handled as specified there.

The main functions performed by secondaries are the outer interpreter and the compiler. The outer interpreter is the part of the language that is normally being executed; it accepts input lines, parses them, and executes them. This function is performed by QUIT and its subprogram INTERPRET. The compiler is used to create new secondaries. It is invoked with : , which first uses CREATE to make a header, then takes over the job of parsing with its subprogram] . This is not enough, however, for all the functions of the compiler. Certain words, called immediate words, are designated so that the compiler does not compile them when they appear in its input stream, but instead executes them immediately. The high bit of the length byte of the header is high for an immediate word. Immediate words are used to implement all of the control structures described in the last article. Most immediate words will also be compiling words; a compiling word is one that adds bytes to the word currently being defined. Compiling words can be created with the word COMPILE; immediate words can be created with the word IMMEDIATE. The word (COMPILE) forces an immediate word to be compiled instead of executed. The word (') allows the address of another word to be used. One note: the word NULL must be an immediate word for an exit from] to be possible without leaving : . With NULL an immediate word, word definitions can extend over several lines. *(Ed. note: see correction on page 26.)*

The other special class of words is the defining words; defining words create entries in the dictionary. Defining words can be created using CREATE and DOES>. CREATE makes a dictionary header. DOES> designates the portion of a definition that follows it to be the code that is run when a word created with the defining word is executed. The defining word : uses the routine SMUDGE to flag words in the process of definition so that if an error occurs in the process of definition the half-created word will be removed from the dictionary, and so that the half-created word cannot be referenced while it is being defined. This is so that words can be redefined with the new definition referring to the old one (note that this does not change any words already defined with references to the old definition).

ALLOT This word reserves space in the dictionary. It simply adds the TOS to H.

, This word adds a value to the end of the dictionary; it is the basis for most of the creation of words. It simply stores the TOS at the location pointed to by H, then ALLOTs two bytes.

C, is the single-byte version of , . It simply stores the low byte of TOS at the location pointed to by H, then ALLOTs one byte.

QUERY This word gets an input line at S0. It uses S0 as input to EXPECT, then resets >IN to zero.

NUMBER converts a string of characters at H to a number, with error-handling. It requires H to be in TOS when it is called; it swaps a zero into NOS, then calls >BINARY. It subtracts H from the address returned by >BINARY of the first non-convertable character, then subtracts from this the length of the string, stored at the location pointed to by H. From this result one is subtracted; if the final value is non-zero, (ABORT") handles the error by printing the offending string followed by a question mark. Otherwise just the number is returned.

INTERPRET This word interprets an input line. It consists of an endless loop where each pass casues a string of characters separated by spaces to be taken off the input line. -' checks to see if the string is the name of a word; if it is not, NUMBER tries to convert it to a number and aborts

```
Secondaries to be hand-compiled.
Standard format is  : name definition ; .
All numerical values are given in hexadecimal; 20=JSR,
60=RTS, four-digit values are PFA's: 1132=(LIT)  1344=,
1358=C, .  The routine to reset S mentioned in QUIT is the
machine code equivalent of S0 @ 2 - S ! .  The pseudo-word
AGAIN merely signifies an uncondition (i.e., infinite) loop
back to the BEGIN statement.
: ALLOT H + ! ;
: , HERE ! 2 ALLOT ;
: C, HERE C' 1 ALLOT ;
: QUERY S0 @ EXPECT 0 >IN ! ;
: NUMBER 0 SWAP >BINARY HERE - HERE C@ - 1 - ABORT" ?" ;
: INTERPRET BEGIN -' IF NUMBER ELSE EXECUTE ?STACK ABORT"
STACK EMPTY" THEN AGAIN ;
: ' -' ABORT"  ?" ;
: (COMPILE) ' 20 C, , ; IMMEDIATE
: COMPILE 1344 20 1358 20 DUP 1132 20 ' 1132 20 C, , , C, ,
, C, , C, , ; IMMEDIATE
: (') ' 1132 20 C, , , ; IMMEDIATE
: QUIT (here is a routine to reset S) ." READY" BEGIN CLEAR
CR QUERY INTERPRET ."  OK" AGAIN ;
: ] BEGIN -' IF NUMBER 1132 20 C, , , ELSE DUP 7 - @ 0< IF
EXECUTE ?STACK ABORT"  STACK EMPTY" ELSE 20 C, , THEN THEN
AGAIN ;
: : CREATE -3 ALLOT SMUDGE BEGIN ] QUERY AGAIN ;
: ; 60 C, SMUDGE POP POP ; IMMEDIATE
: . DUP ABS <# #S SIGN #> SPACE ;
```

Figure 1

if it isn't one. If it is the name of a word, the word is EXECUTEd and the stack is checked for underflow.

' This word finds the address of the word whose name follows the ' in the input line (e.g., ' DROP), with error-handling. It simply calls -', then (ABORT").

(COMPILE) This immediate compiling word forces the compilation of the immediate word following it in the input line. It calls ' , then compiles a call to the address returned.

COMPILE This immediate compiling word is used to create a compiling word. It is used in a sequence that compiles the word whose name follows the COMPILE in the input line.

(') This immediate compiling word places the address of the word whose name follows the (') in the input line as a literal in the word in which (') is used. It uses ', then compiles a call to (LIT), then compiles the address recovered by '.

QUIT This is the main executive. QUIT is essentially an endless loop, CLEARing the stacks and then calling QUERY and INTERPRET. First QUIT clears the main stack by setting S to S0-2 (as implemented here, this demonstrates the flexibility of the language in mixing machine code with secondaries. See Figure 1 and the assembly listings), then it prints a system message. It then starts the endless loop, which consists of a CLEAR of the other stacks, a carriage return, a QUERY, an INTERPRET, and a printing of the message " OK".

] This word is the equivalent of INTERPRET for compilation; it parses and compiles a single input line. It is an endless loop; in each pass of the loop a string separated by spaces is taken off of the input line. -' checks to see if the string is the name of a word; if it is not, NUMBER tries to convert it to a number, and aborts if it is not one. When a number is successfully converted, a call to (LIT) is compiled followed by the number. If the string is the name of a word, that word is checked to see if it is an immediate word. If it is an immediate word it is executed and the stack is checked for underflow. If the word is not immediate, a call to it is compiled.

: This defining word is the equivalent of QUIT for compilation; it compiles a word. It first calls CREATE to make a header, using the string next on the input line as the name of the word. The call to (VARIABLE) that CREATE put at the PFA is deALLOTed, and the length byte is SMUDGEd. An endless loop then calls] and gets another line.

; This immediate compiling word ends the definition of a word and exits from : . A return code is compiled, SMUDGE is called to restore the length byte, and two calls to POP ensure that the return will be not to] or to : , but to INTERPRET.

. This word prints a signed number, using the number output words presented in the last article.

SET This last primary is not really part of the language, but is a very useful development tool. It essentially sets the language to remain in its present state ; that is, it moves the current values of H and CURRENT to the locations from which STARTUP takes them. It also leaves the current length of the language plus a little in TOS, for use in saving the current version of the language to disk. SET is used whenever the language needs to be able to be restarted in its current state; it can be used, for example, after every few definitions in the next list have been entered. One caution: always use SET after anything that has been SET into the dictionary has been forgotten.

```
Secondaries to be typed in.
All numerical values are given in hexadecimal, so BASE
should be set to $10.  In 6502 machine language,  18=CLC,
20=JSR, 90=BCC, B0=BCS; A2=" in ASCII.
: HEX 10 BASE ! ;
: DECIMAL A BASE ! ;
: VARIABLE CREATE , ;
: CONSTANT CREATE , (') (CONSTANT) CURRENT @ 7 + ! ;
: IMMEDIATE CURRENT @ DUP C@ 80 XOR SWAP C! ; IMMEDIATE
: FORGET ' DUP 6 - H ! 2 - @ CURRENT ! ;
: ADJUST 1 >IN +! ;
: ." ADJUST COMPILE (.") A2 WORD C@ 1+ ALLOT ADJUST ;
IMMEDIATE
: IFFER HERE 1 ALLOT ;
: GOUP HERE OVER - 1- SWAP C! ;
: GOBACK HERE - 1- C, ;
: IF COMPILE (IF) B0 C, IFFER ; IMMEDIATE
: THEN GOUP ; IMMEDIATE
: ELSE 18 C, 90 C, IFFER >R GOUP R> ; IMMEDIATE
: DO COMPILE >R COMPILE >R HERE ; IMMEDIATE
: LOOP COMPILE (LOOP) 90 C, GOBACK COMPILE 2RDROP ;
IMMEDIATE
: +LOOP COMPILE (+LOOP) 90 C, GOBACK COMPILE 2RDROP ;
IMMEDIATE
: BEGIN HERE ; IMMEDIATE
: WHILE COMPILE (IF) B0 C, IFFER ; IMMEDIATE
: UNTIL COMPILE (IF) 90 C, IFFER ; IMMEDIATE
: END 18 C, 90 C, SWAP GOBACK GOUP ; IMMEDIATE
: DOES> COMPILE (DOES>) COMPILE DODOES ; IMMEDIATE
: ABORT" ADJUST COMPILE (ABORT") A2 WORD C@ 1+ ALLOT ADJUST
; IMMEDIATE
```

Figure 2

Once these words have been entered, the language is ready to operate (pending debugging, of course). Debugging is probably easiest if breakpoints are placed at strategic locations along the path of execution—at the routines called by the main executive, to start out with. The most important storage locations can be checked at each breakpoint to make sure that things are running properly. Once the main executive is running, printout—the . routine—must be debugged. When . is working, most of the primaries can be tested directly from the main executive. Then the secondaries and particularly : can be debugged by the same techniques. Once all of these are working correctly, the language can be used to enter the rest of itself, the words which follow.

The rest of the language is embodied in the following secondary words. They include all of the words to implement the control structures; the important defining words VARIABLE and CONSTANT, the dictionary management words IMMEDIATE and FORGET, the string output word .", and a few others. They should be entered a few at a time, then tested, and SET into the language. Once all of them have been entered, the language is finished and ready for programming.

HEX This word sets the number input/output base to hexadecimal; it simply stores a $10 in BASE.

DECIMAL sets the number input/output base to decimal; it simply stores a $0A in BASE.

VARIABLE This defining word creates a variable with the string just after the VARIABLE in the input line for a name and TOS for an initial value. It calls CREATE to make a header and , to store the initial value and reserve space for it.

CONSTANT This defining word creates a constant with the string just after the CONSTANT in the input line for a name and TOS for its value. It calls CREATE to make a header, then replaces the call to (VARIABLE) that CREATE stored with a call to (CONSTANT). It then calls , to store the value and reserve its space.

IMMEDIATE This immediate word makes the latest and highest word on the dictionary into an immediate word (or changes it from an immediate word to a normal word). It simply toggles the high bit of the length byte.

FORGET This word removes from the dictionary the word whose name follows the FORGET in the input line, along with anything after that word in the dictionary. It uses ' to get the address of the word, then sets H to the word's NFA and CURRENT to the word's link.

ADJUST This word is useful in the definitions of ." and ABORT" ; it increments >IN by one, thus skipping over one character.

." This immediate compiling word compiles a sequence that prints the string following the ." in the input line, terminated by a quotation mark (e.g., ." HELLO THERE"). It calls ADJUST to skip the space that must necessarily follow the ." , then compiles a call to (."), then uses WORD with a quotation mark for the separation character, then ALLOTs one more byte than the length of the string, then uses ADJUST to skip the terminal quotation mark.

The words implementing the control structures use the stack to store address references; they are built from several utility words which do such things as compile a forward or a backward branch.

IFFER This word is useful in the definitions of some of the control structure compiling words. It puts the value of H on the stack, then ALLOTs the one byte necessary for a relative branch reference.

GOUP This word is useful in the definitions of some of the control structure compiling words. It subtracts the value in TOS from the value of H, subtracts one from this difference, then stores the (one-byte) result at the address which was in TOS.

GOBACK is useful in the definitions of some of the control structure compiling words. It subtracts the value of H from the value in TOS, subtracts one from this difference, and compiles the (one-byte) result into the dictionary.

IF This immediate compiling word marks the start of a conditional. It compiles a call to (IF), then compiles a BCS (branch on carry set = $B0) and calls IFFER.

THEN This immediate word marks the end of a conditional. It is just an immediate version of GOUP.

ELSE This immediate compiling word marks the separation between the two branches of a conditional. It compiles an unconditional branch, in this implementation a CLC and a BCC (clear carry = $18 and branch on carry clear = $90), then executes IFFER, temporarily shuffles aside the TOS and calls GOUP, then restores the TOS.

DO This immediate compiling word marks the start of an indexed loop. It compiles two calls to >R, then puts the value of H on the stack. Note that there is some choice here; as presented, DO will take the loop starting value from NOS and the final value from TOS. This convention is opposite to that of FORTH; to use that of FORTH, simply insert a COMPILE SWAP at the start of the definition of DO. It may be advantageous to define DO as in FORTH, and to define another word FOR with the definition given for DO.

LOOP This immediate compiling word marks one sort of end to an indexed loop. It compiles a call to (LOOP), compiles a BCC (branch on carry clear = $90), calls GOBACK, and compiles a call to 2RDROP.

OOP This immediate compiling word marks the other sort of end to an indexed loop. It is identical to LOOP, except that (OOP) is compiled in place of (LOOP).

BEGIN This immediate word marks the start of a non-indexed loop. It is just an immediate version of HERE; it places the value of H on the stack.

WHILE This immediate compiling word marks one sort of exit test for a non-indexed loop. It compiles a call to (IF), compiles a BCS (branch on carry set = $B0), and calls IFFER.

UNTIL This immediate compiling word marks the opposite sort of exit test for a non-indexed loop. It compiles a call to (IF), compiles a BCC (branch on carry set = $90), and calls IFFER.

END This immediate compiling word marks the end of a non-indexed loop. It compiles an unconditional branch, in this implementation a CLC and BCC (clear carry = $18 and branch on carry clear = $90), then SWAPs the TOS and NOS and executes both a GOBACK and a GOUP.

DOES> This immediate compiling word marks the end of the compilation-time portion of a defining word and the start of the run-time portion. It simply compiles calls to (DOES>) and DODOES.

ABORT" This immediate compiling word compiles a sequence that executes a conditional error abort. This sequence will print the string as an error message, terminated by a quotation mark, which follows the ABORT" in the input line (e.g., ABORT" NOW YOU'VE DONE IT"). ABORT" first calls ADJUST to skip the space that must necessarily follow the ABORT", then compiles a call to (ABORT") and uses WORD with a quotation mark as the separation character to get the string, then ALLOTs one more byte than the length of the string, and uses ADJUST to skip the terminal quotation mark.

There are a few more secondary words which, though not necessary, may be useful:

U. This word prints the TOS taken as an unsigned integer quantity. It is defined by : U. DUP <# #S #> SPACE ;.

? This word prints the (two-byte) value at the address given in TOS. It is defined by : ? @ . ; .

EXIT This immediate compiling word permits a premature exit from a word. It simply compiles a return instruction (= $60). It is defined by : EXIT 60 C, ; IMMEDIATE .

SPACES This word prints a number of spaces, the number being given in TOS. The definition is simply : SPACES 1 SWAP DO SPACE LOOP ; .

RECUR This word allows recursion. Normally the name of the word being defined is SMUDGEd so that it cannot be referred to; this is so that words can be redefined, with the new definition referring to the old. If a word actually wishes to call itself, the immediate word RECUR should be used both before and after the self-reference. RECUR is simply an immediate version of smudge, defined by : RECUR SMUDGE ; IMMEDIATE .

The definitions of many of these words may seem incomprehensible at first, but they should become clear with study and experimentation. They will also repay study, in the sense that an understanding of the workings of the words that make up the language will teach you how to program in the language. Now a short application should demonstrate some general principles. We develop a simple bubble sort and a quicksort (Figure 3).

First, we require an array of 50 random numbers. The word RANDOM uses a simple algorithm to produce pseudorandom numbers. The defining word ARRAY then is compiled; ARRAY defines an array of two-byte values of a given length. CREATE first creates a header, then 2 * ALLOT multiplies the length by two (the number of bytes in each entry) and allots that much space. DOES> closes off the main portion of the defining word. The rest of the code (SWAP 2 * 2 -) will be executed only when the array (in this case NUMBERS) is invoked. NUMBERS will be invoked with an index from 1 to 50 in TOS; DODOES will then place the PFA of NUMBERS on the stack. SWAP 2 * 2 - then adds twice the index to the PFA and subtracts 2 to get an address.

FILL simply fills RANDOM with 50 random numbers; PRINT prints out the values in RANDOM. SWITCH is a

```
Bubble sort and Quicksort

12345 VARIABLE SEED
: RANDOM SEED @ 16807 * DUP SEED ! ;
: ARRAY CREATE 2 * ALLOT DOES> SWAP 2 * + 2 - ;
50 ARRAY NUMBERS
: FILL 1 50 DO RANDOM I NUMBERS ! LOOP ;
: PRINT 1 50 DO I NUMBERS @ . CR LOOP ;
: SWITCH NUMBERS SWAP NUMBERS OVER OVER @ SWAP @ ROT ! SWAP
! ;
: BUBBLE 1 BEGIN
      DUP DUP 1 + NUMBERS @ SWAP NUMBERS @ <
          IF DUP DUP 1 + SWITCH 1 - DUP 0 =
              IF 1 + THEN
          ELSE 1 + THEN
      DUP 50 < WHILE END ." NUMBERS SORTED" ;

: PIVOT OVER OVER OVER OVER + 2 / NUMBERS @
      BEGIN
          BEGIN OVER NUMBERS @ OVER > WHILE SWAP 1 - SWAP
END
          ROT SWAP
          BEGIN OVER NUMBERS @ OVER < WHILE SWAP 1 + SWAP
END
          ROT ROT OVER OVER SWITCH
          OVER OVER 1 + > WHILE SWAP ROT END
      ROT DROP SWAP ;
: RECUR SMUDGE ; IMMEDIATE
: QUICK OVER OVER 1 - <
      IF PIVOT ROT RECUR QUICK QUICK RECUR
      ELSE OVER NUMBERS @ OVER NUMBERS @ >
          IF SWITCH
          ELSE DROP DROP THEN
      THEN ." NUMBERS SORTED" ;
```

Figure 3.

utility word to exchange two elements of NUMBERS, whose indices are given in TOS and NOS. Diagram the state of the stack to understand how it works.

BUBBLE starts at index 1; it takes the index and the index plus 1, obtains the corresponding values from NUMBERS, and compares them. If they are out of order, SWITCH is called in and the index is reduced by 1 – unless it is 1 already. If they are not out of order, the index is increased by 1. If the index is less than 50, the process repeats. At the end, a message is printed out.

Type in the words and variable definitions down to BUBBLE as shown; then execute FILL, PRINT, BUBBLE, and PRINT again. This should show something of the data-handling features of the language, the use of the stack, and the overall speed of the language, as well as the basic pattern of breaking problems up into several parts, each of which forms a word.

The routine PIVOT takes as input two indices from the stack, then shuffles the portion of NUMBERS between those two indices into an upper part all greater than some pivot and a lower part all less than the pivot; it returns with the lower index of the lower part, the upper index of the lower part, the lower index of the upper part, and the upper index of the upper part stored on the stack in that order. The first line finds the pivot value, arbitrarily that of the cell midway between the indices. The first inmost loop moves the upper index down until it hits a value less than the index; the second inmost loop moves the lower index up until it hits a value greater than the index. These values are SWITCHed, and then the outer loop repeats until the indices meet.

RECUR is the immediate word described above for recursion. QUICK is the quicksort routine; it takes as input the upper and lower indices between which it is to sort. If these two indices are not one apart, the space between them is PIVOTed and the resulting two parts are QUICKsorted. If they are one apart, the two values are placed in order. A message is then printed out. Type in PIVOT, RECUR, and QUICK in the order shown, then execute FILL and PRINT. To use the quicksort, type in 1 50 QUICK, then execute PRINT again. This should show something of the power of recursion.

The possibilities for expansion of the language are great; some form of mass storage is still needed, and a text editor would be a very nice addition. Beyond that, graphics, sound, and various sorts of output can be added. Whatever functions are required can be coded, either as primaries, as secondaries, or as a mixture of both. A floating point package is a possibility, as are various other sorts of number forms. The chief feature of the language is its flexibility – its ability to be molded into any desired form. ■

```
                         *
                         ** ALLOT **
                         *
1332: 05 C1 CC CC 0B 13
1338: 20 E5 12                H
133B: 4C 13 0D                +!
```

```
                        *
                        ** , **
                        *
133E: 01 AC A0 A0 32 13
1344: 20 D8 0E                    HERE
1347: 20 F3 0C                    !
134A: 20 32 11                    (LIT)
134D: 02 00                       $0002
134F: 4C 38 13                    ALLOT
                        *
                        ** C, **
                        *
1352: 02 C3 AC A0 3E 13
1358: 20 D8 0E                    HERE
135B: 20 38 0D                    C!
135E: 20 32 11                    (LIT)
1361: 01 00                       $0001
1363: 4C 38 13                    ALLOT
                        *
                        ** QUERY **
                        *
1366: 05 D1 D5 C5 52 13
136C: 20 11 13                    S0
136F: 20 53 0D                    @
1372: 20 88 0D                    EXPECT
1375: 20 32 11                    (LIT)
1378: 00 00                       $0000
137A: 20 F0 12                    >IN
137D: 4C F3 0C                    !
                        *
                        ** NUMBER **
                        *
1380: 06 CE D5 CD 66 13
1386: 20 32 11                    (LIT)
1389: 00 00                       $0000
138B: 20 69 08                    SWAP
138E: 20 F9 0F                    >BINARY
1391: 20 D8 0E                    HERE
1394: 20 B9 0A                    -
1397: 20 D8 0E                    HERE
139A: 20 6E 0D                    C@
139D: 20 B9 0A                    -
13A0: 20 32 11                    (LIT)
13A3: 01 00                       $0001
13A5: 20 B9 0A                    -
13A8: 20 68 11                    (ABORT")
13AB: 01 BF                       "?"
13AD: 60                          ;
                        *
                        ** INTERPRET **
                        *
13AE: 09 C9 CE D4 80 13
13B4: 20 33 0E BEGIN              -'
13B7: 20 93 10                    (IF)
13BA: B0 06
13BC: 20 86 13                    NUMBER
13BF: 18 90 F2                    ELSE
13C2: 20 48 12                    EXECUTE
13C5: 20 AE 0E                    ?STACK
13C8: 20 68 11                    (ABORT")
13CB: 0C A0 D3                    " S
13CE: D4 C1 C3                    TAC
13D1: CB A0 C5                    K E
13D4: CD D0 D4 D9                 EMPTY"
13D8: 18 90 D9                    AGAIN
```

```
                        *
                        ** ' **
                        *
13DB: 01 A7 A0 A0 AE 13
13E1: 20 33 0E                    -'
13E4: 20 68 11                    (ABORT")
13E7: 02 A0 BF                    " ?"
13EA: 60                          ;
                        *
                        ** (COMPILE) **
                        *
13EB: 89 A8 C3 CF DB 13
13F1: 20 E1 13                    '
13F4: 20 32 11                    (LIT)
13F7: 20 00                       $0020
13F9: 20 58 13                    C,
13FC: 4C 44 13                    ,
                        *
                        ** COMPILE **
                        *
13FF: 87 C3 CF CD EB 13
1405: 20 32 11                    (LIT)
1408: 44 13                       $1344
140A: 20 32 11                    (LIT)
140D: 20 00                       $0020
140F: 20 32 11                    (LIT)
1412: 58 13                       $1358
1414: 20 32 11                    (LIT)
1417: 20 00                       $0020
1419: 20 1C 08                    DUP
141C: 20 32 11                    (LIT)
141F: 32 11                       $1132
1421: 20 32 11                    (LIT)
1424: 20 00                       $0020
1426: 20 E1 13                    '
1429: 20 32 11                    (LIT)
142C: 32 11                       $1132
142E: 20 32 11                    (LIT)
1431: 20 00                       $0020
1433: 20 58 13                    C,
1436: 20 44 13                    ,
1439: 20 44 13                    ,
143C: 20 58 13                    C,
143F: 20 44 13                    ,
1442: 20 44 13                    ,
1445: 20 58 13                    C,
1448: 20 44 13                    ,
144B: 20 58 13                    C,
144E: 4C 44 13                    ,
                        *
                        ** (') **
                        *
1451: 83 A8 A7 A9 FF 13
1457: 20 E1 13                    '
145A: 20 32 11                    (LIT)
145D: 32 11                       $1132
145F: 20 32 11                    (LIT)
1462: 20 00                       $0020
1464: 20 58 13                    C,
1467: 20 44 13                    ,
146A: 4C 44 13                    ,
                        *
                        ** QUIT **
                        *
146D: 04 D1 D5 C9 51 14
1473: 38                          SEC
1474: A5 0C                       LDA S0L
1476: E9 02                       SBC #$02
```

```
1478: 85 ØØ                  STA SL
147A: A5 ØD                  LDA SØH
147C: E9 ØØ                  SBC #$ØØ
147E: 85 Ø1                  STA SH
148Ø: 2Ø Ø7 11               (.")
1483: Ø5 D2 C5               "RE
1486: C1 C4 D9               ADY"
1489: 2Ø FA ØE BEGIN         CLEAR
148C: 2Ø B4 ØD               CR
148F: 2Ø 6C 13               QUERY
1492: 2Ø B4 13               INTERPRET
1495: 2Ø Ø7 11               (.")
1498: Ø3 AØ CF CB            " OK"
149C: 18 9Ø EA               AGAIN
                 *
                 ** ] **
                 *
149F: Ø1 DD AØ AØ 6D 14
14A5: 2Ø 33 ØE BEGIN         -'
14A8: 2Ø 93 1Ø               (IF)
14AB: BØ 19
14AD: 2Ø 86 13               NUMBER
14BØ: 2Ø 32 11               (LIT)
14B3: 32 11                  $1132
14B5: 2Ø 32 11               (LIT)
14B8: 2Ø ØØ                  $ØØ2Ø
14BA: 2Ø 58 13               C,
14BD: 2Ø 44 13               ,
14CØ: 2Ø 44 13               ,
14C3: 18 9Ø DF               ELSE
14C6: 2Ø 1C Ø8               DUP
14C9: 2Ø 32 11               (LIT)
14CC: Ø7 ØØ                  $ØØØ7
14CE: 2Ø B9 ØA               -
14D1: 2Ø 53 ØD               @
14D4: 2Ø 58 Ø9               Ø<
14D7: 2Ø 93 1Ø               (IF)
14DA: BØ 19
14DC: 2Ø 48 12               EXECUTE
14DF: 2Ø AE ØE               ?STACK
14E2: 2Ø 68 11               (ABORT")
14E5: ØC AØ D3               " S
14E8: D4 C1 C3               TAC
14EB: CB AØ C5               K E
14EE: CD DØ D4 D9            MPTY"
14F2: 18 9Ø BØ               ELSE
14F5: 2Ø 32 11               (LIT)
14F8: 2Ø ØØ                  $ØØ2Ø
14FA: 2Ø 58 13               C,
14FD: 2Ø 44 13               ,
15ØØ: 18 9Ø A2               THEN THEN AGAIN
                 *
                 ** : **
                 *
15Ø3: Ø1 BA AØ AØ 9F 14
15Ø9: 2Ø 5F 12               CREATE
15ØC: 2Ø 32 11               (LIT)
15ØF: FD FF                  $FFFD
1511: 2Ø 38 13               ALLOT
1514: 2Ø 12 ØF               SMUDGE
1517: 2Ø A5 14 BEGIN         ]
151A: 2Ø 6C 13               QUERY
151D: 18 9Ø F7               AGAIN
                 *
                 ** ; **
                 *
152Ø: 81 BB AØ AØ Ø3 15
1526: 2Ø 32 11               (LIT)
1529: 6Ø ØØ                  $ØØ6Ø
152B: 2Ø 58 13               C,
152E: 2Ø 12 ØF               SMUDGE
1531: 2Ø 44 ØF               POP
1534: 2Ø 44 ØF               POP
1537: 6Ø                     ;
                 *
                 ** . **
                 *
1538: Ø1 AE AØ AØ 2Ø 15
153E: 2Ø 1C Ø8               DUP
1541: 2Ø ØØ ØB               ABS
1544: 2Ø 55 ØF               <#
1547: 2Ø 8C ØF               #S
154A: 2Ø C8 ØF               SIGN
154D: 2Ø A1 ØF               #>
155Ø: 4C BD ØD               SPACE
                 *
                 ** SET **
                 *
1553: Ø3 D3 C5 D4 38 15
1559: A5 Ø4                  LDA HL
155B: 8D 28 13               STA STARTUP+$12
155E: A5 Ø5                  LDA HH
156Ø: 8D 29 13               STA STARTUP+$13
1563: A5 ØA                  LDA CURRENTL
1565: 8D 2E 13               STA STARTUP+$18
1568: A5 ØB                  LDA CURRENTH
156A: 8D 2F 13               STA STARTUP+$19
156D: 2Ø D8 ØE               HERE
157Ø: 2Ø 32 11               (LIT)
1573: FØ Ø7                  $7FØ
1575: 4C B9 ØA               -
```

CORRECTION

In part three of "Write Your Own Threaded Language," found in issue number 11, there is an error on page 10. In the definition of NULL, the first byte of the header should **not** be 01, but 81 instead. This makes null an immediate word. This change needs to be made in both the assembly and machine code. Without this, word definitions are restricted to one line.

```
        *
        ** NULL **
        *
12AE: <81> 8D AØ AØ 59 12
```

Books of Interest

Soul of CP/M

by Mitchell Waite and Robert Lafore
Published by Howard W. Sams & Co., Inc.
4300 West 62nd Street
Indianapolis, IN 46268
382 pages, 7¼" x 9", $18.95

The CP/M operating system is a very powerful tool, but most books only show you how to use the built in commands such as DIR, REN, and ERA. The manual supplied by Digital Research is basically a list of commands without any instructions on how to modify them or use them from your programs. There are a number of books on CP/M, but most of them are beginners guides which tell you how to boot the system, change drives, use STAT and PIP, etc. There are a few good books which contain detailed information on rewriting CP/M for special applications, but most of these books start out on a high level and assume that you already understand how the CP/M functions work, that you are familiar with assembly language programming using ASM, DDT and LOAD, and that you know the mechanics of modifing the BIOS.

Soul of CP/M begins by describing the 8080, 8085, and Z-80 CPUs, and then explains the structure of CP/M. Next, it introduces DDT and 8080 assembly language with a step by step example of using DDT to enter and run a short program using the Console Output System call. After several examples of using DDT to enter and save short programs it shows how to use a word processor to write a text file assembly language program and assemble it with ASM.

The authors assume that the reader has no knowledge of assembly language programming or the CP/M system, and they very thoroughly explain each assembly language instruction and system call when they are first used. After establishing the basics, the authors continue with examples of utility programs using additional instructions and calls. The following list of contents will give you an idea of the broad coverage in this book.

•**Introduction.** Who Is This Book For, What This Book Will Teach You, 8080 8080A 8085 Z-80—What's the Difference?, What You Need To Know To Get the Most Out of This Book, How This Book is Organized.
•**Chapter 1 The Big Picture: How CP/M Is Organized.** What Is an Operating System Anyway?, What's So Great About CP/M?, The Parts of CP/M, 8080 Architecture, DDT: The Programmer's X Ray and Probe.
•**Chapter 2 One Toe In The Water: Console System Calls.** Console Output System Call, Get Console Status, Barber-Pole Display Program, Console Input, Executing Programs From CP/M, System Reset, A Warm Boot.
•**Chapter 3 Getting In Deeper: Advanced Console System Calls.** Print String, Read Console Buffer, Echo Program, Name Display Program, Direct Console I/O, List Output to Printer, Reader Input, Punch Output, Get I/O Byte, Set I/O Byte.
•**Chapter 4 Using The Assembler.** What's an Assembler Do Anyway?, What ASM Does, The DECIBIN Routine Reads Decimal From Keyboard, DECIHEX Program Converts Decimal to Hex on Screen, BINIHEX Binary to Decimal Conversion Routine, Using CP/M's Submit Utility.
•**Chapter 5 Disk System Calls.** Records - Files - Tracks - Sectors - Allocation Units - Extents - and Goodness Knows What Else, Talking to BDOS, Open File, The Problem With Where the DMA Is Located, Read Sequential System Call, Set DMA Address, TYPE2 Program Imitates the TYPE Command, LINES Program Prints Number of Lines in Text File.
•**Chapter 6 Writing to the Disk.** Make File, Write Sequential Record, Close File System Call, Program to Write a Sequential Record, STORE Program Stores Text in File, Delete File System Call, Random Records, Read Random System Call, Write Random System Call, RANDYMOD Program to Modify a Random Record, Compute File Size System Call, Set Random Record System Call.
•**Chapter 7 Soul Searching: Wildcards and the Disk Directory.** How CP/M Stores Files on the Disk, Search For First System Call, Wildcards, Search For Next System Call, Erased Files, Saving an Erased File, The Bit Map, WORDS Program Counts Words in Files and Uses Wildcards.
•**Chapter 8 Teamwork: Using System Calls From Basic.** Where Do We Put the Assembly Language Program in Memory?, How To Get the Assembly Language Program Where We Want It To Go, How Do We Transfer Control Between BASIC and the Assembly Language Routine?, How Do We Pass Arguments Between BASIC and the Assembly Language Routine?, BINEHEX2 Routine Called From BASIC, Other Ways To Put the Assembly Language Routine Into Memory, HEXIBIN2 Passing Arguments to BASIC From an Assembly Language Routine, Operating on Strings With an Assembly Language Routine.
•**Chapter 9 The Innermost Soul of CP/M: How To**

PRODUCT EVALUATION: MACINKER

by Art Carlson

Have you checked the copy from your printer lately? You may be surprised if you do—the print may have been nice and dark when you started with a new ribbon, but the image slowly gets lighter and lighter as you use it. The change is so gradual that you may not have noticed that the print is now grey instead of black.

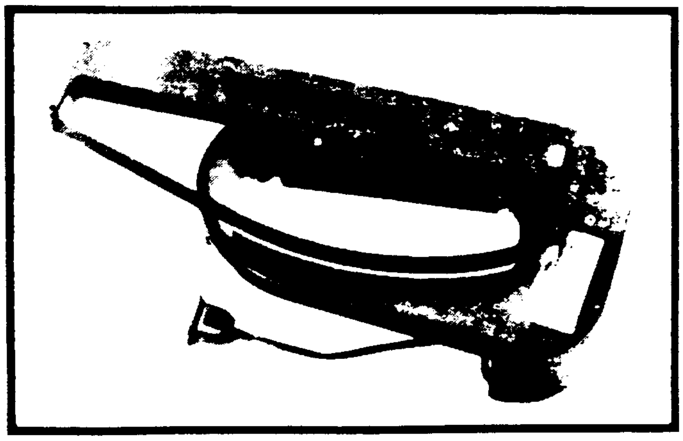
I didn't realize how bad the output from my Epson MX-80 was until I was checking a listing against an older copy, and could see the difference. The copy used for printing the magazine has to be black to ensure good reproduction. I purchased a new Epson type ribbon from an independent supplier, which temporarily corrected the problem. The ribbon then proceeded to go through the same routine again, letting the quality deteriorate until I happened to see new and old copy together. This time I bought three ribbons so that I could change on a regular schedule before the print quality got so bad, but the new ribbons were just as light as the old ribbon I was ready to throw out! That's when I got real interested in reinking ribbons.

I had originally felt that I couldn't justify the expense of an inker since I only had one printer, but a little figuring (I still use a slide rule for rough estimates like this because I don't have to carry approximations out to many digits) showed that in the long run it was cheaper to get an inker than to continue buying ribbons—especially if some of the new ribbons were marginal to begin with.

At this time Computer Friends (6415 SW Canyon Court, Suite #10, Portland, OR 97225) sent a MacInker for evaluation. We proceeded to ink the three defective new ribbons and several old ribbons. It took a little experimenting to get the amount of ink and time right, but the inker worked as advertised and we save money every time we reink a ribbon instead of throwing it away.

I also reink ribbons for friends who have Epson printers, but each model inker only works with certain ribbon types, so I can't ink ribbons from other printers. It takes about 45 minutes to ink an Epson MX-80 ribbon, but it doesn't require any attention after I fill the ink wells and turn it on. You can also purchase blank cartridges, and colored inks which can be mixed to match your logo or to create new colors.

The MacInker would be a worthwhile purchase for anyone who uses their printer a lot, and should be a natural for clubs or other groups who can share the inker. An unanticipated benefit is that you can change ribbons more frequently and give them a light reinking before the copy is grey. This way, the printer will consistently produce high quality copy instead of allowing it to vary from dark to light. To me, this alone is worth the $57.95 price of the inker even if I didn't save money. ■



Books of Interest, continued

- **Modify CP/M for Different Peripherals.** Why You're Reading This Chapter, What Is the BIOS Anyway?, Learning Your Way Around the BIOS, The Complete BIOS Listing, How to Modify Your Printer Driver, Installing the New Driver Into Your Bios, Inserting the New Driver Into the CP/M System, A Shortcut, Modifying BIOS for Different Control Characters.
- **Appendix A Hexadecimal Notation.** Why Use Hexadecimal Notation?, Binary Notation, Decimal Notation, Hexadecimal Notation, Converting Hex to Decimal, Converting Decimal to Hex.
- **Appendix B Utility Programs.** HEXDUMP, Micro Space Invaders, HEXIDEC, FILEDUMP.
- **Appendix C Summary of 8080 Instructions.** 8080 Architecture, 8080 Instructions, Assembler Directives.
- **Appendix D Tables.** ASCII Character Set With Hexadecimal Equivalents, Hexadecimal to Decimal Conversion, Multiples of 1K (1024) in Decimal and Hexadecimal, Decimal Hex and Binary Conversion.
- **Appendix E Summary of BDOS System Calls For CP/M 2.2.**
- **Appendix F Summary of DDT Commands.**
- **Appendix G Summaries of Programs Used and Locations of Instruction Descriptions.**

No book is perfect, and **The Soul of CP/M** does contain a few minor errors, such as referring to the Z-80 as "another chip from Intel" when it is actually from Zilog. The book is well laid out with liberal use of a second color and tint blocks and a comprehensive index. We recommend this for anyone who wants to learn how to use the hidden power of their CP/M system. ■