THE COMPUTER JOURNAL®

For Those Who Interface, Build, and Apply Micros

Vol. II, No. 6 Issue Number 10 $2.50 US

Editor's Page

We are often asked, "Why do you want to publish a computer magazine when there are already too many on the market?" Other frequent questions are "Why can't I find your magazine on the newsstand?" and "Why don't you run full color ads like the other magazines?"

The answers to these questions are not simple, but I'll attempt to explain our reasons for publishing *The Computer Journal* and answer these questions.

Most of the established magazines are now being run by large corporations who are primarily interested in the annual profit, and they are engaged in a battle for the almighty advertising dollar. In order to attract advertising they must have a huge circulation base (many are running 400,000 to 700,000) which requires that they publish general interest articles that will appeal to the largest possible audience. Right now, the largest audience is the people just getting started in using micros for business aplications.

These large magazines have the advertising income to use full color, and the wide appeal needed to appear on the newsstands, but they can't use much editorial space for the nuts-and-bolts technical information we need. Instead, they publish mostly reviews on their advertisers' products and introductory articles for those who will buy their advertisers' products.

The purpose of *The Computer Journal* is to provide information for people who want to write the programs and build the hardware needed to interface their computer to the real world. We are adding some advertisers with products of specific interest to our readers, but our primary purpose is service the reader and not the advertiser. This means that our reviews will be limited to items which we feel are of interest to our readers, and not just those which will bring in advertising dollars. It also means that we can't afford full color printing at this time because we don't have much advertising income.

Our readers are a special kind of people—they're not the general newsstand audience. For that reason, you won't find *The Computer Journal* on the rack in your local drugstore, but we are interested in appearing on specialized, technical book and magazine racks. We'd appreciate your suggestions on people we should contact.

In order to provide a place where you can trade ideas and work with others on common problems, we are starting a series of reader design projects where we will present projects for our readers. This will be a forum where many people can contribute their knowledge so that everyone doesn't have to try to reinvent the wheel. We encourage you to suggest projects for this series, and we hope that you will take the time to send your ideas in response to others' projects.

We would also like to start a discussion on establishing a technical and scientific program exchange. These might be in the form of listings, disks, or possibly a special bulletin board. The purpose of this exchange would be to provide a base of public domain specialized programs and sub-routines developed by our readers. If we establish a bulletin board, we could also provide some of the programs from our articles so that you don't have to key them in. The programs on a bulletin board would have to be limited to those of special interest to our readers which are not available on most of the other bulletin boards. We would appreciate feedback from you regarding the programs you feel we should have, and what programs you could contribute—**no pirated copies of commercial programs please!**

The Computer Journal is published for YOU, and it will only be as good as YOU make it. Your input in the form of ideas, suggestions, and articles will be greatly appreciated ■

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.*

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.

THE FORTH LANGUAGE:
A LEARNER'S PERSPECTIVE
by Mahlon G. Kelly

FORTH is a strange programming language. Many who have learned conventional languages such as Pascal, BASIC, Fortran or Cobol find Forth hard to learn. Yet Forth has ardent, fanatic admirers. Some Forth programmers almost seem to be members of a cult, arguing that Forth is the ultimate language, the only one that should be used, or the first that should be taught to students. Early versions of Forth were given away, just to encourage use of the language.

If its admirers are so ardent why hasn't Forth achieved the popularity of BASIC and Pascal? And why do some find it so hard to learn? I found it fairly easy to learn but I had to ignore all of my preconceptions about how a program is written. I had to forget about subroutines, GOTO statements, variables as I had known them, and algebraic notation embedded in program lines. And I had to learn about such things as stacks, Reverse Polish Notation, dictionaries and word definitions. Nevertheless, Forth is logical and simple; the re-learning process was not difficult.

A good Forth program is an elegant and beautiful creation like a profound mathematical proof. It may even be worth learning Forth simply for the aesthetic satisfaction. I have known Forth programmers for several years, but their missionary zeal was such that I ignored their proselytizing. A few weeks ago I was waiting in a friend's home and playing with his TRS-80 Model I. He had loaded Forth and left the manual open to the glossary and a program on the screen. I started looking at the listing and the manual, and I had a wonderful "Aha!! experience." I suddenly saw how the program worked—not the details, but the general idea. And it was beautiful. I don't know why it was beautiful, but then, I don't know why an elegant proof of the Pythagorean theorum is beautiful.

And Forth is efficient. The source code for a Forth program normally resides on disk, taking virtually no memory, and is normally very compact in a listing. The compiled code typically takes a small fraction of the memory used by a compiled BASIC, Fortran or Pascal program. And a Forth program will run 20 or more times faster than an interpreted BASIC program and several times faster than a compiled one. Using Forth with an 8087 coprocessor in an IBM can produce more than a hundred-fold speed improvement. Usually the only way to beat the speed of a Forth program is to use assembly or machine language (and assembly language can be easily embedded into most Forth programs).

I had to learn Forth. I didn't know if I would use it much for my application programs (I now use only Forth) but I had to learn it. I had to find out more about the language that could be so conceptually simple and yet do so much with such elegance. I ordered MMSFORTH (one of the most powerful and fastest Forths) for my LNW-80, along with the book **Starting Forth** by Leo Brodie. But I started with trepidation—there were so many stories about how hard it is to learn Forth.

Much to my surprise I was comfortable with Forth and writing meaningful programs within four days. Forth is not a hard language to learn—you just have to think of programming in a completely different way. You have to completely re-learn how to interact with your computer. I find that it's more of a shock to put the Forth disk into my computer than it is to move from my LNW or Columbia with BASIC to our University's CDC with Fortran. But I found learning to use Forth to be a lot of fun, probably because I started with an open mind.

A Forth Program

Forth is different in many ways, but probably most confusing to a neophyte is reading a program. Conventional programs proceed from top to bottom. Instructions at the beginning are executed first and execution continues to the end, perhaps looping and perhaps branching to subroutines. But unless a conventional program is haphazardly written, with too many GOTOs and so on, it's read by the computer in the way that we read a page of text. Forth programs don't work that way. Lines in Forth programs define words that do various things. The words are put together in phrases to form new words, and at the end there will be a "punch line" that puts everything together into a single word that defines the program.

Let's look at an example. (Modified and extended from an example in the book **Starting Forth** by Leo Brodie.)

```
(Things within parentheses are comments.)
(This program will wash a load of clothes.)
(A phrase started by : and ended by ; defines a word.
The first word in the phrase names the word. The
phrase may span several lines.)
(We must assume that some words have been
previously defined in terms that the computer can
understand— those are the words that I've defined in
lower case. You wouldn't do it that way in a real
program.)
: START turns on the power to the washer ;
: LOAD sounds a bell to tell the operator to put in the
clothes ;
: WATER fills it with water to a high-level switch ;
: SOAP dispenses a slug of soap ;
: AGITATE does what it says for n minutes ;
```

```
: SPIN also does what that says for n minutes ;
: DRAIN lowers the water to a low-level switch ;
: UNLOAD tells the operator to take the clothes out ;
: WASH SOAP WATER 5 AGITATE DRAIN 2
SPIN ;(define WASH: 5 and 2 are the number of
minutes.)
: RINSE WATER 3 AGITATE DRAIN 2 SPIN;
(define RINSE; 3 and 2 are minutes.)
:WASHER START LOAD WASH RINSE RINSE 3
SPIN UNLOAD ;
```

If the program had been loaded (either from the keyboard or from a disk) and you type the word WASHER the machine will do your laundry. WASHER is interpreted and uses the words surrounded by :-;, which were compiled when the program was loaded. If you typed RINSE the clothes would only be rinsed. If you typed WASH the suds would be left. WASHER uses all of the defined words. Those words were compiled, or converted, to short machine language programs when the main program was loaded. WASHER then jumps from one little machine language program to another. Forth is called a Threaded Interpretive Language (or TIL) because programs are composed of words, each of which represents another program, and each of which is connected to another by a thread. You can imagine a Forth program as an intricate spider's web connecting various routines that are stored in memory. When the program is run it jumps around in memory, following the web. That's one of the reasons it's so fast.

It's usually easiest to understand a Forth program by reading it from the bottom up and then figuring out what each word does. If the words are chosen to have clear English meanings and are well commented, a Forth program is largely self-documenting. It's particularly helpful if each word has some special purpose and it's especially helpful if the word can be used several times.

Forth programs may be put into the computer from the keyboard or from disk, although the disk would normally be used for anything but the simplest program. Forth programs are stored on disk in blocks of 1024 bytes. For example, a 40 track, double-sided disk on an IBM, using MMSFORTH, will store 395 blocks. If I typed "45 LOAD" and the WASHER program was on block 45, it would be read from disk just as if I had typed it from the keyboard. That block could then have loaded another block. If a colon-definition is typed from the keyboard the word is compiled as soon as the semicolon is typed and "enter" is hit. Forth doesn't care where the input comes from. And Forth programs are easy to debug. Each word can be individually tested by typing it from the keyboard. This ability to quickly write and debug intricate programs is one reason Forth is popular for controlling complex machines such as radio telescopes and robots.

Forth programs may also be precompiled. That means they may be made part of the Forth system as it is loaded when the disk is booted. For example, in MMSFORTH, if I had loaded the WASHER program and then "customized" the disk (by typing CUSTOMIZE) the washer words would have been saved to disk as machine code and would be loaded when the disk was booted; they would become part of my particular version of Forth, always there when needed. Forth is a language that grows, and after a few months of use, my version of Forth will probably be different from yours, as each of us adds personal features.

Precompilation also allows production of versions of Forth dedicated to particular tasks. For example, FORTHWRITE is a text editor written in MMSFORTH for the IBM PC and TRS-80 machines. The user receives source code (four disks in the case of the TRS-80 Model I, two for the IBM) that are precomplied to machine language and merged with MMSFORTH to produce a dedicated word-processing system. When the precompiled FORTHWRITE disk is booted the user need not even know that Forth is being used, but the source code is always there to be re-precompiled with customized changes. FORTHWRITE, although written in Forth, is similar to, just as fast as, and much, much more flexible than Radio Shack's widely acclaimed Scripsit, which was all written in machine language. FORTHWRITE equals or betters any word-processing system written for any "normal" operating system, including such items as the widely used Wordstar. Precompilation allows the user to produce what are practically new operating systems. FORTHCOM (a smart-terminal program) and DATAHANDLER (a data-base management system) are similar dedicated programs. Forth has even been used to write a BASIC interpreter, and it was nearly as fast as the equivalent machine language version.

In summary, Forth programs comprise a group of predefined functions or subroutines, called words, that are entered from the keyboard or by loading a disk block. Words are made from more primitive words that are defined when you boot and load a disk. For example, I could type in a word "PLUS" like this:

```
: PLUS + ;
```

What I have done is create a duplicate definition for a word that's part of the original language. The new word, PLUS, can now be used interchangeably with the original word, "+."

Although there's a 1979 standard for Forth, most implementations have added many non-standard words (and a 1983 standard has recently come out). For example, MMSFORTH for the TRS-80 and IBM PCs includes separate utilities that support such things as high resolution vector graphics for the IBM PC as well as floating point math with transcendental functions and imaginary numbers, including support for the 8087 math co-processor chip (it may be the only language to support imaginary numbers on a microcomputer). And you will find yourself adding to your own version of Forth. Within a week I had written words for Hex-Binary-Decimal conversion and for a screen-oriented memory dump.

But you won't start learning Forth by writing whole-block programs. You'll start at the keyboard by interactively playing with the machine.

Learning Forth

Because Forth doesn't care if it receives input from a disk or from the keyboard, it may be the most completely

interactive compiled language there is. I learned Forth by typing very short "programs" directly from the keyboard. It's fun to learn because the results come out immediately. But to understand what's happening you have to understand a peculiar (and very efficient) feature of Forth: the *stack*.

If you type "55 6 – ." the result on the screen is 49. You have used two words, " – " ("minus") which says subtract the two numbers, and ".", ("dot") which says print the result. But how did " – " know what to work on? When you typed 55, that number was put on the top of a stack; 6 was then put on top of the 55, pushing the stack down by one (actually the computer first looked at 55 to see if it was a word, found it wasn't, then found it was a legal number, and so put it on the stack). Picture the stack as several numbers, like plates stacked on a table. When you typed " – " the computer took the 6 off, then the 55, then subtracted the 6 from the 55 and put the result (49) back on the stack. "." then pulled the 49 off and put it on the screen. Understanding how the stack works is very important in learning Forth. If you have used a Hewlett-Packard calculator you'll be used to this stack procedure, often called Reverse Polish Notation (RPN) or a FILO (First In Last Out) stack. Most Forth words use the stack, and that saves memory. Other languages require that numbers be assigned to variables, and those variables each require memory. In Forth, memory doesn't have to be permanently allocated to variables that are little used. Using the stack *does* mean you must keep track of what is pushed on and popped off. Keeping track of the stack can be confusing, but careful definition of words, so that each does something very specific and predictable to the stack, helps. So does practice. Actually, I should mention that there's another Forth stack that is used internally to keep track of where a program is in its execution (that stack may also be *carefully* used for number storage). Also, if floating point numbers are being used with an 8087 math co-processor chip on an IBM PC the 8087's stack actually becomes the main stack, which is a very, very fast way of doing things and makes MMSFORTH ideal for fast number crunching.

There are many words that operate on the stack. " – ", " + ", "/", and "," do what you would expect, and "MOD" divides two numbers and puts the remainder (the modulus) on the stack. There are also words for double precision and floating point operations, although the latter are not standard. You can try all of them from the keyboard. Some words rearrange the stack. For example, suppose the stack contains 6 with 55 above it and you want to subtract 6 from 55; " – " would give you – 49. But if you typed " SWAP – . ", 6 would be swapped with 55, subtracted, and 49 would be printed. Other words for manipulating the stack are OVER (3 2 OVER would leave 3 2 3 on the stack and ROTate (1 2 3 ROT would produce 2 3 1). (The left-hand number goes onto the stack first; the right-hand number is on the top of the stack). The first thing you will learn about Forth is how the stack works and that's easiest to do by experimenting at the keyboard. Forth is confusing at first; I couldn't have learned it without this kind of experimentation.

Forth allows the use of constants and variables, but the

beginner should beware; they can teach lazy habits and use more memory and time. Variables and constants are actually words, but they are special words that do special things. If you type

2 CONSTANT SPIN-TIME

the following happens: 2 goes onto the stack and CONSTANT creates a place in memory (in the dictionary) for the word SPIN-TIME and puts 2 there. Then when you type SPIN-TIME it is seen as a word that says take 2 and put it on the stack. If you type "55 SPIN-TIME -." you see 53 on the screen. If SPIN-TIME SPIN had been used in our washing machine program the washer would have spin-dried for 2 minutes. Variables are similar but a bit more complicated. If you type

VARIABLE AGITATE-TIME

space is set aside in the dictionary for the variable AGITATE-TIME. Typing AGITATE-TIME puts the *memory location* (address) for the variable on the top of the stack, *not* the value itself. The word " ! " ("store") says to take the value that's second on the stack and store it at the locatiion pointed to by the top of the stack. Thus

5 AGITATE-TIME !

would store 5 in the memory location pointed to by AGITATE-TIME.

AGITATE-TIME @

would put the value of AGITATE-TIME on the top of the stack. "@" ("fetch") is a word that says "take a memory value from the top-of-stack and fetch its contents onto the top of the stack." "AGITATE-TIME @." would print 5 on the screen; so would "AGITATE-TIME ?" because "?" is simply defined as "@.".

In addition to variables and constants, arrays may be defined. And variables, constants and arrays may be of a variety of types: single byte, single precision (2 bytes or 16 bits), double precision (4 bytes), or strings. In MMSFORTH there are enhancements to use floating-point real and imaginary numbers as well, including exponential notation. MMSFORTH also supports QUANs, which combine the properties of both constants and variables.

You should learn how all of these things work by trying them interactively while sitting at the keyboard. It's much easier to understand how Forth works by trying it than by reading about it.

Forthrightly Simple

At first Forth may seem to be such a simple language that not much can be done with it. The main things that one does are define words, manipulate the stack, declare constants, and change the values of variables. (Actually, we can also do string manipulation but I've said little of that.) Forth may seem simple, but it gives a programmer power over a microcomputer that's rivalled only by assembler language.

In a sense, it is hard to program in Forth. You have to be clever and plan ahead. A program must be well thought out in advance and each word must be defined with care, being cautious for example, not to unintentionally leave something on the stack. But Forth's interactive nature and power makes up for this. And there are some very important words and features that I have not mentioned yet.

Suppose you want to work with strings of letters. Letters can be represented by numbers. The EMIT will put the letter represented (in ASCII) by the number on the top of the stack to the screen. Thus 89 EMIT would print a Y (if you are in DECIMAL; Forth also supports other number bases). There's also a sequence in memory starting at a location called PAD where strings can be stored as ASCII equivalent numbers and manipulated. There are many more string manipulation words, including "TYPE," which puts a string starting at a specified memory location and with a specified length onto the screen. String manipulation in the 1979 and 1983 standards of Forth is not very good, but enhancement is fairly easy and is included in many implementations. MMSFORTH has particularly good string-handling abilities, mimicking many of the functions of BASIC.

There are also all kinds of conditional operations that let you do the equivalent of branching to subroutines. For example there are comparators such as "<", ">","=", "<>", and "0=". They each return a value of "true" (1) or "not-true" (0) to the stack. "5 3<" would return "not-true" or 0 while "5 3>" would return "true" or 1. Then there's the complex word IF, which must be followed later by THEN. If IF sees a true (non-zero) on the top of the stack it will execute the words between IF and THEN. If it sees a not-true (0) it will skip to what follows THEN. Thus the words in the middle act like subroutines. The construct "BEGIN–WHILE–REPEAT" executes the words in the first half and loops back as long as a non-zero number remains on the stack. There are other WHILE constructs in most Forths. "DO–LOOP" works differently. It makes a number of loops specified by the top two numbers on the stack "15 5 DO I . LOOP" would produce "5 6 7 8 9 10 11 12 13 14" on the screen. "DO–LOOP" actually uses another stack. "15 5 DO I . LOOP" does the following: it puts 15 and then 5 from the main ("user") stack to the top of the second ("return") stack, gets the 5 from there to the user stack (with the word "I" for increment), then prints the 5. "LOOP" then adds 1 to the 5 on the return stack, looks to see if it is less then 15, and if it is, execution goes back to "DO." If 15 was equaled or exceeded, whatever followed "LOOP" would be executed.

Most of these more complex words are made up from the simpler stack manipulation and memory manipulation words. Even more complex words can be constructed, for example D+ and D- for adding and subtracting double-precision numbers, or SIN, TAN, and LOG for doing transcendental functions. But they're mostly made up by simple manipulation of the stack, words, memory, variables, and constants. If, however, a word cannot be defined by using previous words, or if the definition is too slow (both of which are uncommon) then a word can be defined by a sequence of assembler language mnemonics. Assembler language can easily be written directly within a Forth program. Forth words may be made up of assembler instructions. Forth can have several vocabularies; for example MMSFORTH includes assembler instructions for

the 8080, Z80, 8088, and 8087 chips.

Learning Forth is really just a matter of becoming familiar with the way words are used, how the stack works, how programs are written, stored, and called from disk, and, perhaps with most difficulty, what the important pre-defined system words do. There's a sort of knack to learning to use the stack and defining appropriate words, but once you have the knack, Forth becomes a very "natural" language. In a way, Forth asks you to think like you did back when you were learning addition and subtraction and before you were corrupted by the terminology and formalisms of algebraic notation. (If you use a Hewlett-Packard calculator, you'll understand the attraction of this.) It took me only four days to become comfortable with Forth, which is more than I can say for Fortran, Cobol, or Pascal.

But What's It Good For?

Forth was originally developed by Charles H. Moore at the National Radio Astronomy Observatory (Kitt's Peak) during the early 1970s. The language was used for controlling radio telescopes. It was intended to rapidly and efficiently control equipment with a minimal use of memory and time, but without the programming effort that would have been needed for a machine language program. Forth's early applications were for controlling large machines and equipment.

But the constraints of time and memory make Forth particularly useful for microcomputers. Is Forth the "natural" language for microcomputers? Should it be the first learned by novices? Should it replace other languages? All of these have been argued convincingly by Forth enthusiasts.

Without question, Forth writes faster code than any other "high-level" language. And Forth source code is shorter. Forth object code is also usually shorter than the compiled code from other "high-level" languages. Why? Because Forth is a TIL (Threaded Interpretive Language). Unlike an interpreted language such as BASIC, the source code need not remain in memory. And while most compilers pile up redundant machine language instructions (macros) on top of each other, duplicating every function each time it's defined, Forth defines the machine code only once for each word. Forth is quick because machine language jump instructions take very little time. And, of course, machine language and assembly language instructions can be embedded directly in the definition of Forth words. In some ways Forth isn't really a higher level language. It combines the advantages of machine, assembly, compiled and interpreted languages. Forth can do anything that your hardware allows, and by using assembler mnemonics it can do it as fast as is theoretically possible.

Forth's speed, efficient use of memory, and combined levels of programming make it a "natural" high-level language for use in microcomputers. Does that mean that Forth should be used by everyone learning to program microcomputers? People are oriented towerd a spoken language, and programming languages like BASIC are easier for them. For example it's easier to understand

```
10 A = 5
20 PRINT A
```

than it is

```
5 A ! A @.
```

although both assign a value to a constant and print the constant. If everyone had to learn Forth it might be that fewer people would program their own computers.

But there's another side to the argument. BASIC encourages you to sit down at the keyboard and interactiverly write a program. That means if you write a long program you will eventually be tempted to patch it up by using a lot of GOTOs. Two months later you won't understand the program, and it will certainly not be efficient in terms of either time or memory. A Forth program would have required much more forethought (Forthought?) and planning. The discipline would have produced a better program. You would have broken the problem down into tasks and sub-tasks, each defined by a word, and the end result would be very carefully designed and efficient. Forth may be the ultimate structured language. The problem is that many don't want to bother with such discipline when they're writing a short program that will be used a few times and forgotten. Forth may be the best way to teach future programmers, but BASIC may be preferred by many casual users of personal computers.

There's a more important problem with Forth. A "pure" Forth language is a self contained entity. There is no separate operating system, only an operating "environment." "Pure" Forth won't run under CP/M, MSDOS, Newdos, or whatever. And it won't read disks from those operating systems. There are Forths that work under conventional operating systems (MMSFORTH is available in a PC-DOS version for example) but such versions entail compromises produced by the speed and memory overhead of the operating system, and sacrifice some of the advantages of the language. But is the "stand-alone" nature of a "pure" Forth really a disadvantage? Forth is very transportable. I can take a source program written on my LNW with MMSFORTH and run it on a TRS-80 Model I, III, or IV, or on an IBM PC or compatible machine (using a completely different microprocessor). The problem isn't really that Forth doesn't work with other operating systems, but that few IBM PC owners, for example, have Forth. If Forth had come first to the microcomputer world we might find the usual "higher-level" operating systems and languages written in Forth (it has actually been done, and quite effectively). If most programmers used Forth we would be able to use assemblers, applications programs and other languages as we wished. But they would be written in Forth and we would be using Forth.

When I'm working with my LNW microcomputer I prefer to use Newdos 80 Version 2 as the operating system, LNWBASIC as an interpreted BASIC, either ZBasic or ENHBAS as my compiled language, and NewScript as a word processor (although I may soon convert to FORTHWRITE). There are many other programs that interact with those. In other words I can use a variety of programs and languages, and because of the operating

system and microcomputer I can let them interact with each other. That's much more difficult, if not impossible, with most versions of "pure" Forth. But *if* I had started with Forth and *if* those other languages and programs had been written in Forth there would be no problem. It's easy to understand why Forth users are active missionaries. If more microcomputers and more programmers worked with Forth, the lack of compatibility between different machines, languages and hardware would be greatly reduced.

And I'm pretty sure that we would have better software if it was all written in Forth. Forth provides so much control of the machine, requires so little memory, and is so fast that well written Forth programs have an inherent advantage.

In Summary

Forth is a very different programming language. Its use of the stack, words, and Reverse Polish Notation all ask that you think differently when you write a program. But it is **not** hard to learn—not if you're willing to forget your preconceptions about what a programming language should be like. And learning Forth can be very exciting and satisfying. Forth programs are like works of art; they give an inherent satisfaction that is a reward in itself. But Forth is not for those who are intentionally lazy. Forth has been called a language that uses a human as a precompiler. It requires that you think out very carefully what you want to do and that you be very careful about what your word-definitions mean. Writing a program in Forth, even a simple one, requires brain work. But that doesn't mean it takes longer to write a Forth program. Any program should be well thought out at the start—it will decrease debugging time and result in a better piece of software. The time spent "up front" in planning a Forth program will be regained at the end by a decrease in debugging effort. And perhaps most importantly, Forth words can be separately debugged, preventing long and tedious changes to the whole program.

The greatest disadvantage of efficient and "pure" implementations of Forth is that they are hard to use with data and files produced on other operating systems. An efficient Forth is its own operating system. Somehow the cart has come before the horse here: Forth probably could have been used to write most of the presently popular operating systems, and if it had, I suspect that many of the incompatibilities between IBMs, TRS-80s, Apples and Ataris wouldn't exist. But that's wishful thinking.

Forth writes code that is very fast and uses relatively tiny bits of memory. That fact in itself will allow Forth to continue as the language of preference for certain applications.

Who cares what Forth is good for? I don't ask about the practical use of the Mona Lisa or Blue Boy. I don't knock the efforts of those producing computer art. And I don't argue that pure mathematics should be eliminated from the university curriculum. Forth is beautiful. It is a very satisfying creative medium, and it can be used to produce programs that are works of art (of course, it can be used to produce rubbish as well). We respect painters, poets and novelists for the works they produce. We will eventually respect prgrammers who work in Forth (or some very similar language) for the same reasons. ■

An Affordable Graphics Tablet for the Apple][

by James E. Oslislo

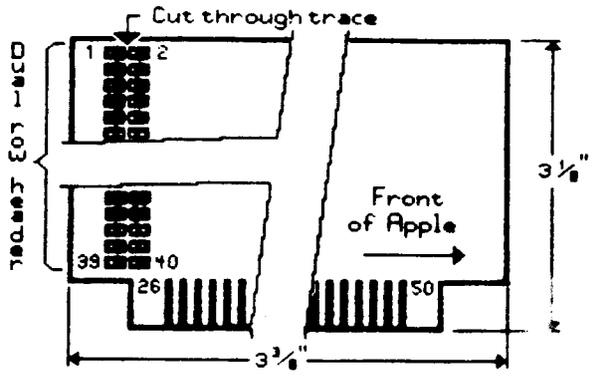


FIGURE 1: INTERFACE BOARD LAYOUT

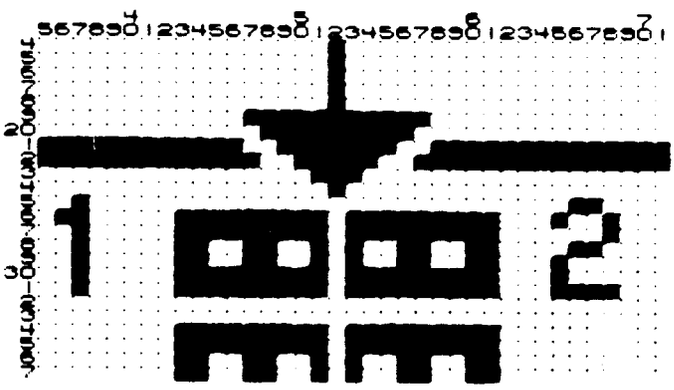


FIGURE 2: ENLARGED CORNER OF INTERFACE BOARD LAYOUT

A relatively low cost graphics tablet can be obtained for the Apple II computer by converting the Radio Shack Color Computer graphics tablet with a simple interface card. Although there is a certain amount of work involved in the conversion, no modifications need to be made to the graphics tablet.

As with any project, there are a number of pros and cons associated with it. A few I have found so far are as follows:

Pros:

1. Ease of use. A graphics tablet allows the fastest and most accurate inputting of graphical information of any of the current popular devices such as mice, joysticks and touch pads.
2. Ease of interface. The interface card is not very difficult to build, and all of the parts necessary can be purchased at Radio Shack.
3. The low price. I purchased my tablet during a sale for $199; I have seen other graphics tablets for the Apple which cost four times that much.

Cons:

1. The warranty. Although the conversion makes no direct modifications to the tablet, an error could creep into construction which could ruin the tablet—it might take some fancy talking to get it repaired under warranty. Even though no modifications need be made to the graphics tablet, the mere use of the tablet with a computer other than one of Radio Shack's undoubtably voids the warranty.
2. Hardware incompatiblities. My tablet has difficulty accessing some of the extreme values in its range. Hence, some points on the edge of the graphics screen are accessible only after software manipulations.
3. There is no support software. Although the book of software for the Radio Shack computer that comes with the tablet could be converted for an Apple, it did not appear that it would be worth the effort. I preferred to start from scratch and write my own software.

Interface Operation

To function correctly, the graphics tablet needs thirty connections to an Apple peripheral slot. Of these connections, sixteen go to the Apple address bus, eight go to the Apple data bus and four go to the Apple power supply (+5v, +12v, -12v and ground). Of the two remaining connections, one goes to the R/W line and the other to the phase zero clock. One problem that arises is due to the difference in system clocks. The Color Computer has a clock rate of 0.89Mhz while the Apple works at 1.028Mhz. I suspect that this slight difference may be the cause of the difficulty in accessing some areas of the tablet. Another problem is the difference in memory usage between the two computers. The tablet is accessed with the addresses $FF60, $FF61 and $FF62. This would seem to be a problem because the Apple system monitor requires these addresses. The solution lies in switching the interface card address lines A4 with A12 and A7 with A13. This change converts the previous addresses to $CFF0, $CFF1 and $CFF2, a range of addresses which falls nicely into the area set aside for the Apple's I/O needs.

Interface Construction

The four items required for construction are as follows:

A 50 position plug-in interface board (RS# 276-166)
A 40 position card edge connector (RS# 276-1558)
A 40 position dual-row socket jumper (RS# 276-1542)
A 40 position dual-row header (RS# 276-1540)

Due to the transitory nature of Radio Shack parts, these recently available parts may have to be replaced with other parts found elsewhere.

If the Radio Shack board is used it must be sawed as shown in Figure 1 in order to fit into the Apple's enclosure. The copper traces shown on the sketch must be cut in half with a thin saw before the dual row header can be soldered to the board. Care should be taken so as not to accidentally peel the traces off the board. The purpose of the dual row header is to remove the strain on the soldered joints that would occur if the ribbon cable was soldered directly to the board.

The major task in completing the interface is making sure that the connections go to the right places. In a connection that goes first from the Apple's slot, then to the board, then to the header, then to the ribbon cable, then to the edge connector, and finally to the graphics tablet board, errors can easily occur in the assembly if extreme care is not practiced.

One possible trouble area is the numbering of the card edge fingers in the black cartridge which contains the electronics for the graphics tablet. You can see the numbering scheme etched on the PC board inside if the cartridge is opened up, but to do so you must break the warranty seal which covers a screw holding the box together. If you choose to leave the box closed, the numbering scheme is as follows:

1. Assume that the top of the cartridge is the side with NO slots in the case.

2. While looking straight in at the card edge fingers with the top side of the cartridge up, the odd numbered fingers labeled one to thirty nine occur from left to right on the top side of the circuit board.

3. With the cartridge in the same position as in step two, the even numbered fingers labeled two to forty occur from left to right on the bottom side of the ciruit board.

4. When numbered correctly, the next even numbered finger should sit directly underneath the odd numbered finger which preceded it, i.e.; finger #2 should be directly beneath finger #1, finger #4 should be directly beneath finger #3, etc.

5. An observant individual will notice that the card edge connector to be installed on the ribbon cable has its connections already numbered on the outside of its case. This should simplify things immensely, but for some reason the numbering is **NOT** the same as per the card edge fingers and **should be disregarded!**

On the new interface board, run thin gauge wire (such as stripped apart ribbon cable) from the card edge connector to the dual row header according to the wiring arrangement in Table 1 and the photos in Figures 5 and 6.

Before any testing is done under power, all of the wiring should be double checked with a continuity tester from the fingers on the interface card to the contacts inside the card edge connector. After you are positive that everything is perfect, plug the interface into your Apple but NOT into the cartridge of the graphics tablet. Power up the Apple and use a voltmeter to check for correct power supply voltages at the card edge connector (+12v, −12v and +5v) with respect to ground. Recheck all wiring if a problem is found. Remember never to connect or disconnect any part of the project while the Apple is turned on; serious electronic problems could occur in either the Apple or the graphics tablet. After all the supply voltages check out, turn off the Apple and connect up the graphics tablet through the cartridge. Repower the Apple and look for any obvious problems such as no picture present on the monitor or no "beep" from the speaker. If anything out of the ordinary occurs, immediately switch off the Apple and recheck the wiring, while keeping a look out for solder bridges.

One possible problem which may occur is due to a conflict of I/O usage by expansion cards. The easiest solution is to remove any unnecessary cards when using the graphics tablet.

If everything seems to be all right, the system can be checked out with the following Basic program:

```
1 PRINT PEEK(53232), PEEK(53233), PEEK(53234)
2 GOTO 1
```

After you type RUN to start the program, three columns of numbers should be printed across the screen. The first

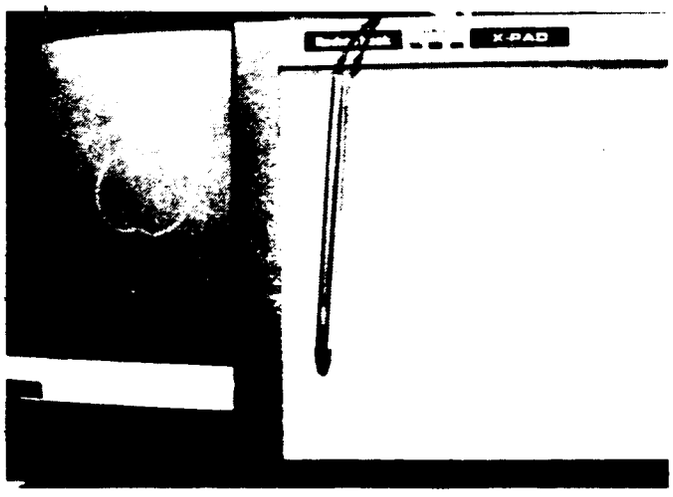


Figure 3

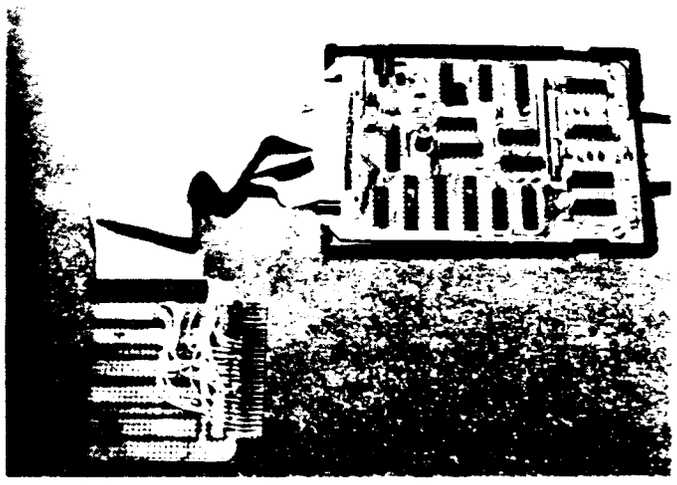


Figure 4

column should contain a value between 0 and 255 corresponding to the horizontal position of the pen on the graphics tablet, 0 being far left and 255 being far right. The second column should contain the vertical position of the pen, 0 being at the top of the tablet and 191 being at the bottom. The third column should contain a value of 0, 2 or 3 depending upon the proximity of the pen to the tablet. If the pen is farther than about an inch from the tablet, a value of zero is printed. If the pen is within an inch but not pressing down on the tablet, a value of two is printed. Lastly, if the pen is pressed down on the tablet, a value of three is printed. As discussed before, the vertical and horizontal values may not reach the extreme values in their ranges. It should be noted that the horizontal range of the graphics tablet is from 0 to 255, while the high resolution screen of the Apple contains 279 horizontal points. Therefore, a direct point to point relationship is not possible between the tablet and the screen. When you tire of the string of numbers, type <CTRL><C> to exit the program.

A trivial Basic program to see a graphical representation of the tablet in action is as follows:

```
1 HGR: HCOLOR = 1
2 X = PEEK(53232)
3 Y = PEEK(53233)
4 Z = PEEK(53234)
5 IF Z < > 3 THEN 2
6 HPLOT X,Y
7 GOTO 2
```

When you run this program, dots should be printed on the computer's screen corresponding to whatever is drawn on the tablet. Nothing will happen unless the pen is in contact with the tablet. To exit the program <CTRL><C> must be used.

I found that when the graphics tablet is repeatedly accessed at high speed, a glitch will occasionally occur in the outputted pen coordinates. It is therefore necessary to generate a software timing delay of about seventeen milliseconds between accesses to maintain data integrity.

TABLE 1

COLOR COMPUTER CARTRIDGE CONNECTOR SIGNALS		APPLE II PERIPHERAL CONNECTOR SIGNALS	
Pin #	Descrip.	Pin #	Descript.
1	-12V	33	-12V
2	+12V	50	+12V
3	unused		
4	unused		
5	unused		
6	E, cpu clock .89 Mhz	40	Phase Zero clock 1.03 Mhz
7	unused		
8	unused		
9	+5V	25	+5V
10	Data, D0	49	D0
11	Data, D1	48	D1
12	Data, D2	47	D2
13	Data, D3	46	D3
14	Data, D4	45	D4
15	Data, D5	44	D5
16	Data, D6	43	D6
17	Data, D7	42	D7
18	R/W	18	R/W
19	Address, A0	2	A0
20	Address, A1	3	A1
21	Address, A2	4	A2
22	Address, A3	5	A3
23	Address, A4	14 ####	A12
24	Address, A5	7	A5
25	Address, A6	8	A6
26	Address, A7	15 ####	A13
27	Address, A8	10	A8
28	Address, A9	11	A9
29	Address, A10	12	A10
30	Address, A11	13	A11
31	Address, A12	6 ####	A4
32	unused		
33	GND	26	GND
34	unused		
35	unused		
36	unused		
37	Address, A13	9 ####	A7
38	Address, A14	16	A14
39	Address, A15	17	A15
40	unused		

denotes different address than expected

Applications and Software Considerations

As the name implies, a graphics tablet is used primarily in graphics applications. Some frequent uses are in computer art and computer aided engineering design. My software was directed along the lines of developing a package which would allow for the rapid creation of a high resolution picture for an unspecified end use. Figures 1 and 2 were created with my program as a demonstration of the practical use of the graphics tablet. Figure 2 shows the enlarging feature of the program which permits fine

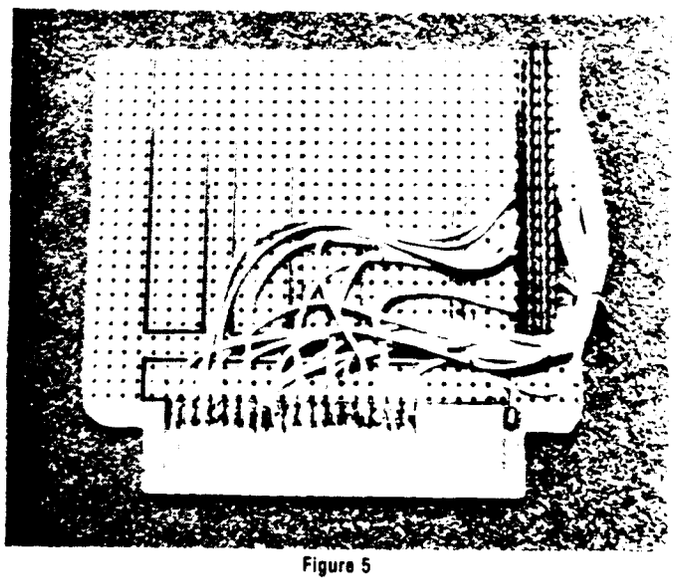


Figure 5

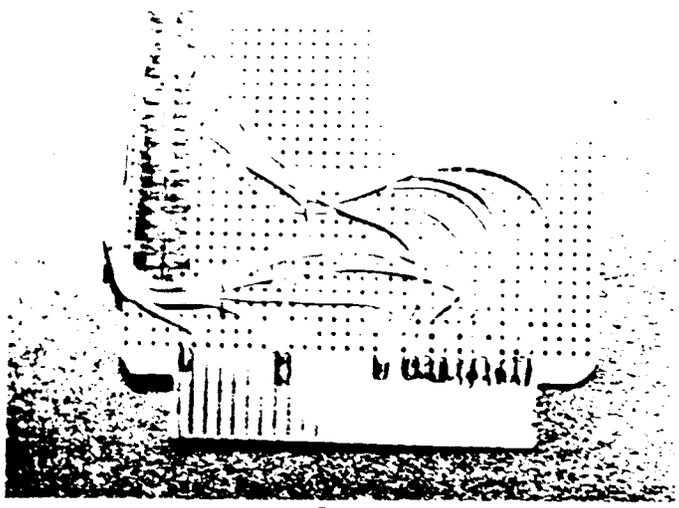


Figure 6

modifications to the screen image. Some of the features which I found could be easily incorporated in the software are as follows:

1. Direct display of image on Apple hi-res screen.
2. Non disturbing cursor showing current pen position on screen.
3. Use of only two colors, white and black for monochrome monitor use.
4. Plotting only while pen is in contact with tablet.
5. Reversal of pen color to "unplot" unwanted areas or the entire screen image.
6. Enlarging desired screen areas to allow for easier dot by dot manipulation of the image.
7. Labeling coordinates around the enlarged screen area so exact dot locations may be verified.
8. Point to point drawing of lines.
9. Saving of current screen to disk or loading of previously created screens.
10. A separate program is used to dump screen images to a printer for hard copies.

Many of the above features could probably be acceptably accomplished with a BASIC program, but to make drawing on the tablet as painless as possible, I wrote an assembly language program to do all the dirty work. Unfortunately, the source code listing is too long (about twenty pages) for publication in this article. However, for a nominal charge, a floppy disk containing the program and listing can be sent to anyone who wishes to see it.

Conclusions

In my opinion, the Radio Shack graphics tablet conversion is a reasonable alternative to the high cost of an Apple tablet and the heartache of using a joystick for developing computer graphics. For a small investment in time and money you can experiment with the same type of device used in state-of-the-art computer aided design equipment. ■

References

1. Apple Computer Inc. Apple II Reference Manual, 1978

2. Radio Shack Inc. Color Computer Technical Reference Manual, 1981

Interfacing Tips and Troubles

A Column by Neil Bungard

Noise Problems, Part One

As I have stressed in previous articles on interfacing, there is more to completing a circuit design than generating the logic diagrams, wire wrapping the circuit, and applying power. Many times you will encounter problems which cannot be solved by studying the logic diagrams or by checking your wiring. The next couple of installments of "Interfacing Tips and Troubles" are going to address an area of hidden but nonetheless real (and frustrating) problems; the area of noise problems. Noise is any undesired signal which interferes with a signal purposely generated. An undesirable signal can be anything from voltage fluctuations in a power supply to electromagnetic interference in the atmosphere. In this article we will look at noise problems which frequently cause trouble in the development of interfacing circuits. For the purpose of our discussion, we will consider noise as being generated in one of three broad categories. (1) Noise associated with the power supply. (2) Noise associated with the interface. (3) Noise associated with outside sources. I will use actual examples of construction and noise reduction techniques which I have used to eliminate noise problems in my own designs, and explain symptoms which were encountered when constructing interface circuits. Finally, we will investigate how and why the solutions to the noise problems were implemented.

A project that I have been working on recently incorporates the use of counters, flip flops, and latches as do most interface projects. These particular devices operate at logic transitions, so they can be considered "edge triggered" devices. The problem with edge triggered devices is that they are not particular about the source of the "edge" which triggers them. In addition to normal triggering by the desired signal, the device can be triggered by fluctuations in the power supply's output voltage, signals induced by a nearby oscillator, current deficits resulting from gate transitions on the board, etc. In addition, the edge triggered devices are fast. This means that they will respond to signals as short as a few nanoseconds. When we consider that the frequency spectrum which effects these devices extends from DC to hundreds of megahertz, it becomes clear that noise can be (and is) a real problem. Unfortunately, it is not easy to determine whether or not noise is the problem in a circuit that is misbehaving. Furthermore, if noise *can* be identified as the culprit in your circuit, the solution to the noise problem may not be obvious. It is my hope that this article will help you gain some insight on identifying and eliminating various noise related problems.

Noise Problem Prevention

In interfacing, as in many other endeavors, an ounce of prevention is worth a pound of correction. Therefore, there is no substitute for sound circuit construction techniques. When wire wrapping interfacing projects it is best to take a modular approach to constructing the circuit. This means that the circuit is wired and tested a section at a time instead of constructing the complete circuit and attempting to test it in its entirety.This allows you to more easily pinpoint problems within a particular section of your design.There is nothing more frustrating than wire wrapping 25 ICs on a piece of vectorboard, powering up the board, and having nothing happen. If you are constructing the circuit a section at a time, you will know exactly where to start looking for a particular problem.

The first step in constructing a wire wrap circuit is running power to all of the ICs. Even though you are not going to wire wrap the logic on all of the ICs, you will want to fully load the power supply to determine that it will not give you problems as you add more ICs to the circuit. You can ensure this by running power to all of the IC sockets and inserting the ICs from the very beginning. Doing this also impels you to formalize the logic diagrams so that you know exactly which ICs you are going to use in your design and completes (except for design changes) the board layout. When running power lines to the ICs, some very important rules should be followed:

1. Make all power supply lines as short as possible.
2. Do not route power supply lines through oscillator sections on the board.
3. Never "daisy chain" power supply lines.

There are three reasons for keeping the power supply lines short. First, even a piece of wire has some resistance. The resistance of the wire is associated with a voltage drop along its length: ($V = I \times R$). If the wire is a power supply line, a small percentage of the actual supply voltage will be lost. The longer the supply line, the greater the resistance and the higher the voltage losses. Since wire has a very low resistance, you might be inclined to think that the losses would be negligible. Under normal circumstances this is true, but in a situation where supply voltages are low, supply currents are high, and power supply noise is

marginal, the length of the supply lines can make a difference. So keep the power supply lines as short as possible.

Secondly, a power supply line can act as an antenna, picking up noise from any source that is emitting an electromagnetic wave. As a matter of fact, I have even seen local radio station signals imposed on a power supply voltage. Typically though, problems occur from signals induced by local oscillators on the wire wrap board or on an adjacent board. For this reason you should attempt to route supply lines around oscillator sections on the board. Also, if possible, always run supply lines perpendicular to wires containing oscillator signals. It is much easier for parallel lines to induce signals into one another than lines which run perpendicular. Wire wrap boards that have grounding planes also help to reduce induction of oscillator signals into supply lines. These boards can be purchased from most electronic distributors that sell regular vector board, but are considerably more expensive. Thirdly, we come to the major reason for keeping power supply lines short. As devices on our wire wrap board go through logic transitions they will draw varying amounts of current. As the current through a wire varies, it creates a condition known as *self inductance*. Self inductance generates a counter current which opposes the original driving current in a wire. This can cause current deficits which will starve the ICs and create all manner of problems on your board. This self inductance problem may not strike you as being that important, but I can assure you that it is a major consideration in interface and digital circuits. I am not finished with the problems related to self inductance—you will be hearing more on this subject in the discussion on decoupling capacitors. But remember, keep your power supply lines short to reduce the possibility of self inductance.

Daisy chaining is the practice of running power supply lines from IC to IC. The preferred method is to connect each IC to a central power supply point (see Figure 1). Daisy chaining is a cardinal sin in circuit construction and should be avoided at all cost. There are a number of reasons why this practice is particularly bad. Daisy chaining effectively extends the length of power supply lines.Even worse, the supply lines are long pieces of 30 gauge wire wrap wire, and the lines may have several wrap junctions between the power supply and an IC. Each wire wrap junction constitutes an added resistance which can be responsible for losses in the supply voltage. But more importantly, each wire wrap junction is a noise generator and can be a major source of power supply noise problems. There should be only two wire wrap junctions between any IC and the power supply; one at the power supply bus and one at the IC.

Power Supply Noise

So far we've talked about techniques for applying power supply voltages to your circuit which will help reduce power supply noise problems. But what are the symptoms of power supply noise, and, assuming that you have taken all of the precautions mentioned above, what could the problems be?

Some of the common problems generated by power supply noise are flip flops that change states without being instructed, counters which increment or decrement without being given count pulses, latches which lose information, and even computer crash situations caused by tristate devices malfunctioning. A number of other conditions can cause this same set of symptomatic malfunctions but the power supply is probably the quickest circuit element to check, therefore I always look for problems there first. Two tests can usually verify the condition of the power supply. First, with the power supply fully loaded, use a DC voltmeter and check the power supply's output voltage. Ideally (for TTL circuitry) this value should be 5 volts, but I have operated between the ranges of 4.5 and 6 volts without any problems. Once you are satisfied that the power supply has sufficient output, check the noise component on top of the DC voltage level. This noise component will be AC, so you won't be able to see it when you make the output voltage measurement with a DC voltmeter. Ideally, you would like to use an oscilloscope to make the noise component measurement so you can see the amplitude and the frequency of the supply noise. If the amplitude (peak to peak) of the noise component is 0.2 volts I would consider using a different power supply or correcting the problem in the supply that you are currently using. If the frequency of the noise component is 120Hz the problem could easily be insufficient power supply filtering and may be corrected by adding additional filtering to the power supply output. If the frequency is higher (in the kHz range) I would suspect that the voltage regulator is oscillating. Monolithic voltage regulators (LM78XX series) have a tendency to oscillate, but this problem can sometimes be corrected by placing 0.22F capacitors on the input and the output of the regulator. If the regulator already has these capacitors installed, try replacing the voltage

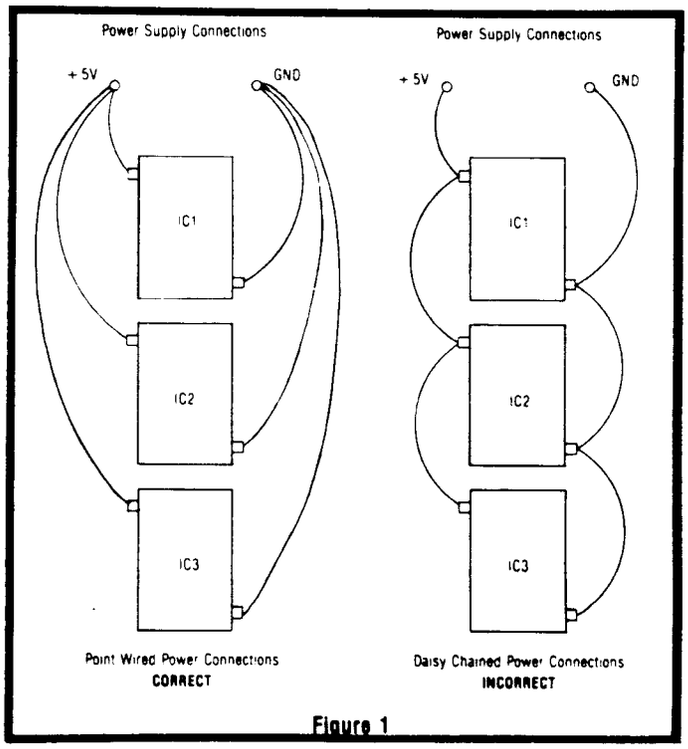


Figure 1

continued on page 26

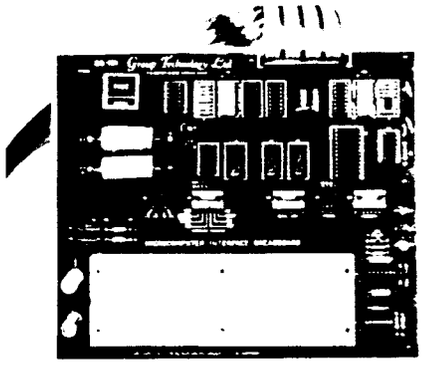
LEARN MICROCOMPUTER INTERFACING
VISUALIZE SCIENCE PRINCIPLES

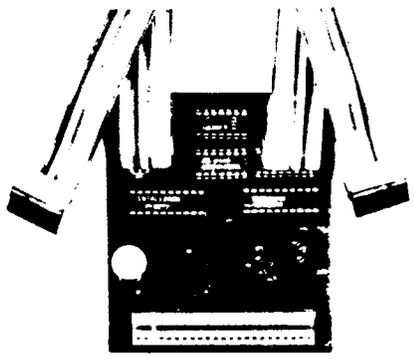
Using GROUP TECHNOLOGY BREADBOARDS with your
APPLE® ...COMMODORE 64® ...TRS-80® ...TIMEX-SINCLAIR® ...VIC-20®

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





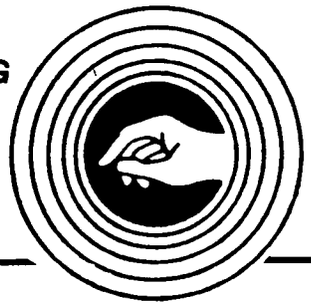
The INNOVATOR™ BG-Boards designed by the producers of the highly acclaimed Blacksburg Series of books have gained wide acceptance for teaching microcomputer interfacing as well as for industrial and personal applications. Detailed, step-by-step instructions guide the user from the construction of device 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.

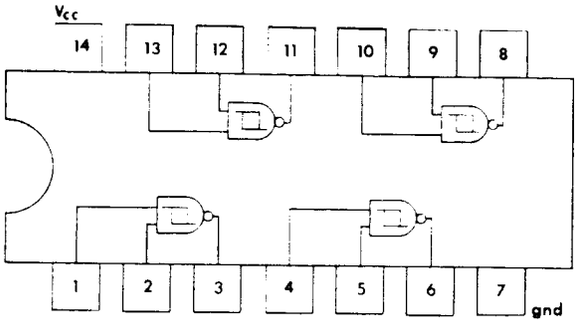
PUTTING HANDS AND MINDS TOGETHER



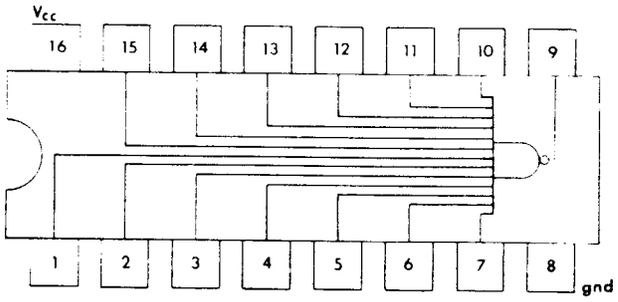
Group Technology, Ltd.
P.O. Box 87N
Check, VA 24072
703-651-3153

LSTTL Reference Chart

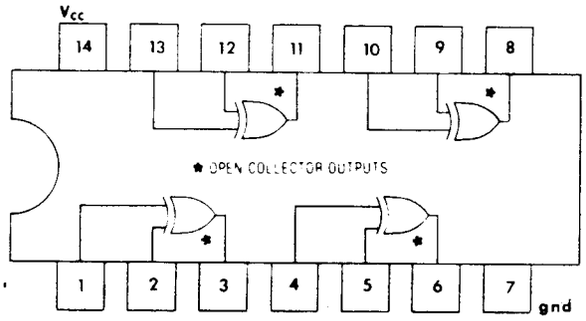
74LS132
QUAD 2-INPUT SCHMITT TRIGGER NAND GATE



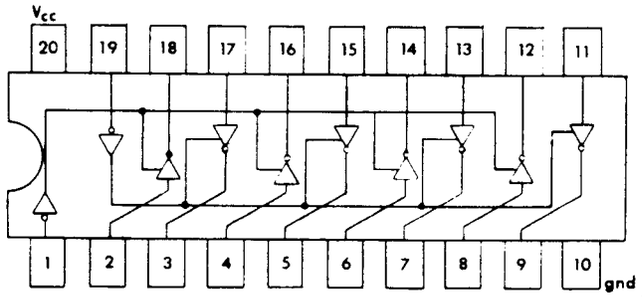
74LS133
13-INPUT NAND GATE



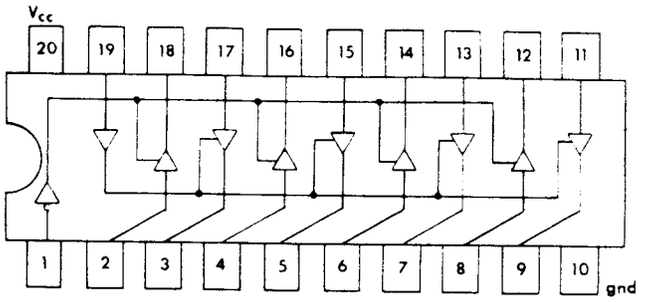
74LS136
QUAD 2-INPUT EXCLUSIVE-OR GATE



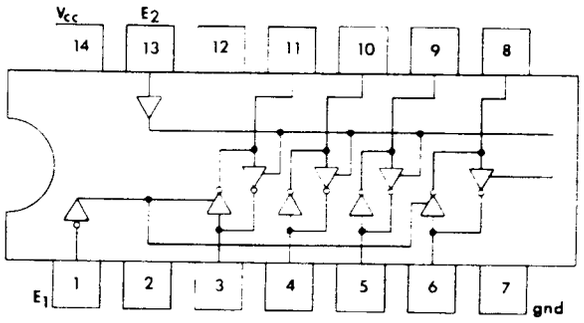
74LS240
OCTAL BUFFER/LINE DRIVER WITH 3-STATE OUTPUTS



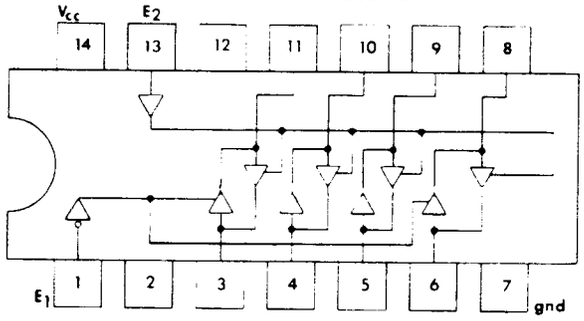
74LS241
OCTAL BUFFER/LINE DRIVER WITH 3-STATE OUTPUTS



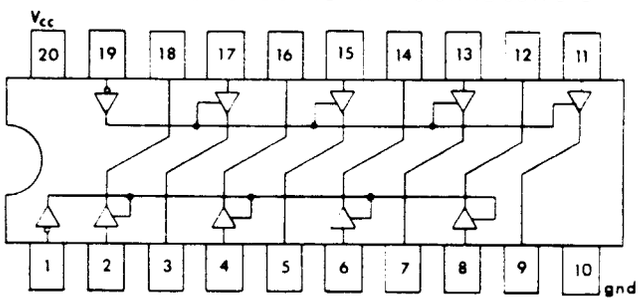
74LS242
QUAD BUS TRANSCEIVER



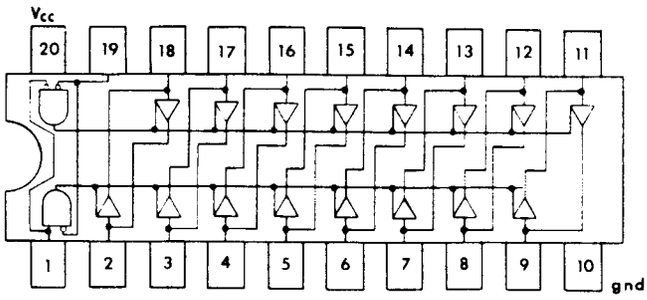
74LS243
QUAD BUS TRANSCEIVER



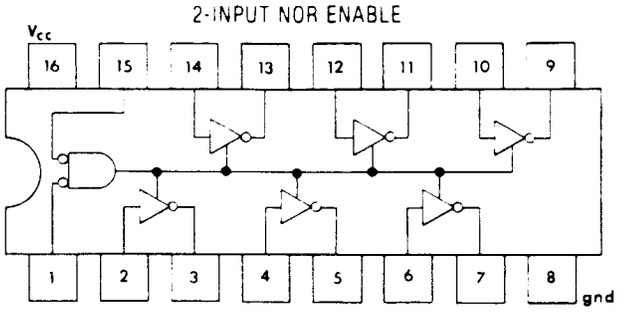
74LS244
OCTAL BUFFER/LINE DRIVER WITH 3-STATE OUTPUTS



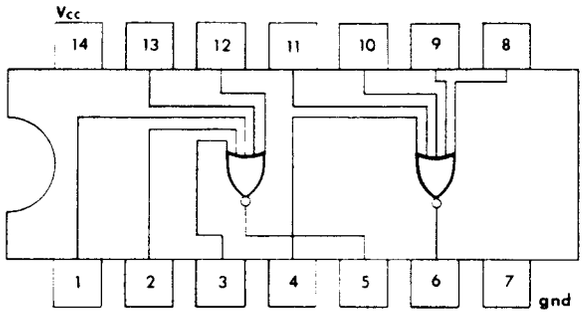
74LS245
OCTAL BUS TRANSCEIVER



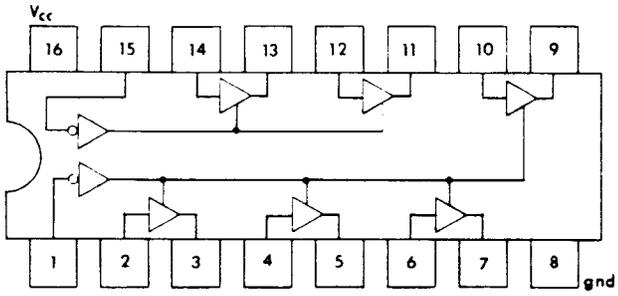
74LS366
HEX 3-STATE INVERTER BUFFER WITH COMMON 2-INPUT NOR ENABLE



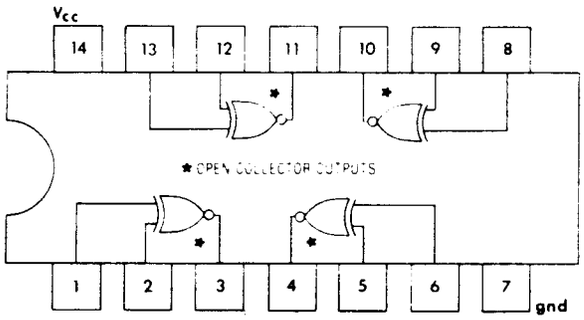
74LS260
DUAL 5-INPUT NOR GATE



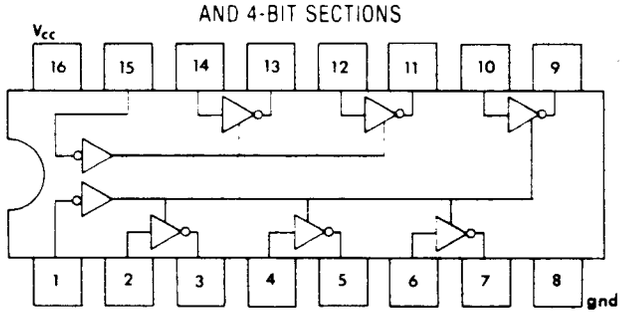
74LS367
HEX 3-STATE BUFFER. SEPARATE 2-BIT AND 4-BIT SECTIONS



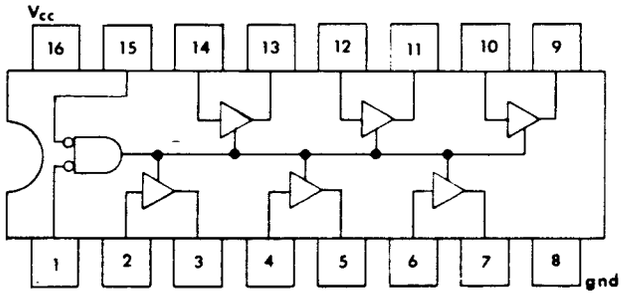
74LS266
QUAD 2-INPUT EXCLUSIVE NOR GATE



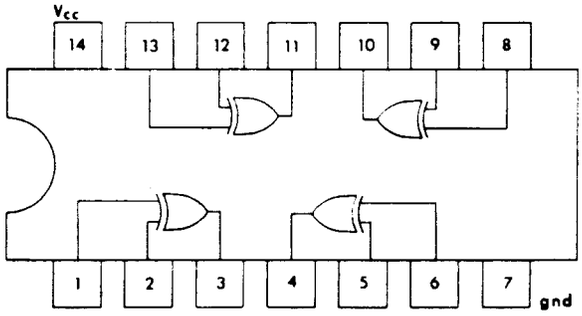
74LS368
HEX 3-STATE INVERTER BUFFER WITH SEPARATE 2-BIT AND 4-BIT SECTIONS



74LS365
HEX 3-STATE BUFFER WITH COMMON 2-INPUT NOR ENABLE



74LS386
QUAD 2-INPUT EXCLUSIVE-OR GATE



Multi-user

A Column by E.G. Brooner

Some Generic Components and Techniques

In previous columns we discussed various kinds of multi-user systems such as time sharing, multi-processing and networks, and we described some particular systems.

This month's column seems like a good place to talk about some individual components that are not part of any particular, complete, integrated multi-user or network system. This category includes various items of hardware and software that can be used as "building blocks" to construct customized multi-user systems or functions. These building blocks permit what might be called "the poor man's multi-user" to be hacked together. They also have value in some kinds of interfacing, which is one of the main thrusts of this magazine.

Some Generic Products

Probably the simplest multi-user device, one with which almost everyone is familiar, is the printer switch. A lot of small installations have two microcomputers and only one printer. A good printer costs as much as a computer, and it seems foolish to buy one that will only be used part time.

The most commonly encountered printer switch is a two-user RS-232 switch. This is a small box with one cable to the printer and one to each computer. The change-over is accomplished by simply turning a knob.

The prospective user is shocked by the price, typically around $150 plus the cost of the cables. What we forget is that RS-232 fully implemented uses 23 lines of the cable, and that's a pretty big switch. The parallel switch for Centronics type printers is available, but less necessary because parallel printers are less costly. Again, parallel interfaces can use up to 30 or more lines, so the switch may cost even more than the RS-232 type. Switches are available for four users if you have such a need, but they cost even more. Gender-changers, which permit the mating of otherwise "wrong-ended" cables, are another generic product.

Protocol converters form another class of equipment. There are RS-232-to-parallel converters and vice-versa. "Modem eliminators" or "null-modems" are sometimes necessary to mate two pieces of equipment, one of which must seem to be DTE and the other DCE. This involves merely reversing some of the transmit/receive lines. Some protocol converters are dedicated computers that translate between ASCII, for example, and another code such as IBM's EBCDIC. When vastly dissimilar equipment has to be run together, such items are an absolute necessity. Another conversion sometimes needed is from synchronous to asynchronous transmission, or from one baud rate to another.

Actually, a modem can be considered a protocol converter, as it converts between two dissimilar communication formats, audio and digital. Several users can share a single phone line by means of a modem-sharer, while "port expanders" provide a similar service for other input/output devices.

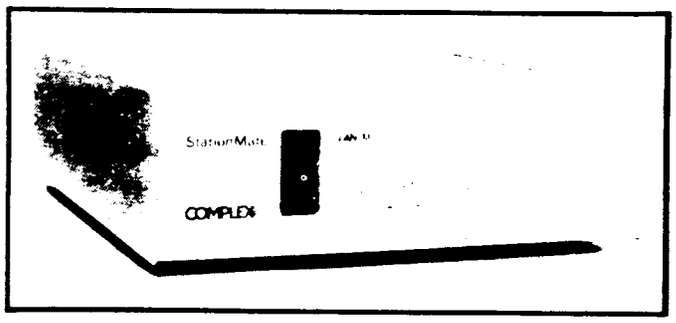
Some of the conversion and other interfacing tasks are performed by hardware, some by software, and some by a combination of both. It is typical for the more complicated devices to be dedicated computers with the functions performed by built-in software. The software for such a device may be programmable or it may be in the form of a plug-in PROM that can be replaced for different functions.

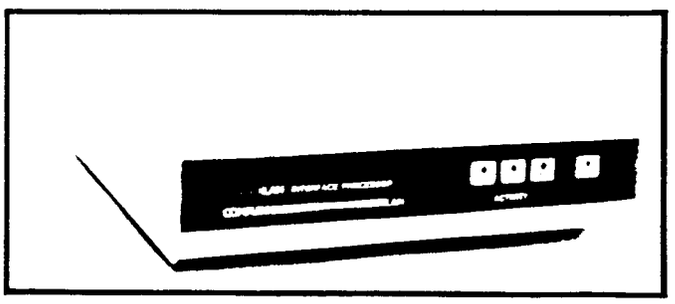
A great many of the devices just mentioned can be obtained fron a single source, Expandor Incorporated, 400 Sainte Claire Plaza, Pittsburgh, PA 15241. Their "Black Box" catalog is worth asking for.

Complexx StationMate and Xlan Products

Complexx Systems, Inc. makes products to convert a variety of different computer equipment into a local area network. Although somewhat lacking in technical detail, their brochure does indicate that the products are quite versatile in their interfacing abilities. Complexx was good enough to send us photos of their major products, which are reproduced here with their permisssion.

Xlan appears to be a standalone multi-user device which can interconnect three RS-232 devices. It contains





continued on page 22

WRITE YOUR OWN THREADED LANGUAGE

Part Two: Input-Output Routines and Dictionary Management

by Douglas Davidson

In the first part of this article, the basic elements of threaded languages were introduced, along with the simplest words. The reader was introduced to the main stack and the R-stack. Pointers were assigned to the stacks, and words used to manipulate the pointers were written. From now on, the two bytes starting at S will be called TOS (top-of-stack) and the two bytes starting at S + 2 will be called NOS (next-on-stack). By convention each word must destroy its operands but nothing else; thus, if a routine takes its input from TOS it is understood that the stack counter is later incremented to drop this value, and if a routine leaves its result in TOS, it is understood that the stack pointer is first decremented. Now the important problems of input and output, as well as more of the dictionary, must be dealt with.

Input in general will be taken in lines; a line will consist of sequences of characters separated by spaces. A line must be stored somewhere in memory, and the place used here will be above the main stack; two bytes are needed, to be called S0, to store the location of the start of the main stack. Each line will be followed by several termination characters to denote the end. Each sequence of characters in the input line will usually be either the name of a word or a number. Sequences of characters separated by spaces will be taken off the input line one at a time, with the current relative position in the input line stored in two bytes to be called >IN. A single sequence of characters must be stored somewhere; for reasons that will be clearer later, the place will be just above the dictionary. Two bytes, to be called H, will point to the first byte above the dictionary.

One further note before proceeding—what is presented here was developed on an Apple II, and Apple's use of ASCII code is somewhat peculiar in that each byte representing a normal character has its high bit set. This fact is used in several places in this presentation.

EXPECT This word gets an input line and places it in memory starting at the location referred to by the TOS. This routine will, in general, be implementation-dependent, but it must put several (three to be safe) termination characters ($8D = carriage return, in this version) at the end of the input line.

CR outputs a carriage return (and linefeed, no doubt, if the distinction is important). This will be implementation-dependent.

SPACE outputs a single space.

WORD This word takes a sequence of characters off of the input stream and places it at the location pointed to by H. The TOS is used as the separation character (usually $A0 = space); this means that WORD starts at S0 + > IN and searches memory for the first character which is not the separation character. It then takes the section which starts there and ends at the next separation character or termination character and places this block, preceded by its (one-byte) length, at the location pointed to by H. >IN is modified accordingly, and the value of H is returned as the TOS. (See Figure 1.)

One thing more is needed to complete the dictionary management: two bytes, to be called CURRENT, are required to store the NFA of the last and highest word o the dictionary. Also, a note is needed here about the thir and final stack to be used. The 6502, and no doubt man other processors as well, keeps its own stack, on which i stores return addresses from subroutine calls. These retur addresses will be used directly in several ways; here the

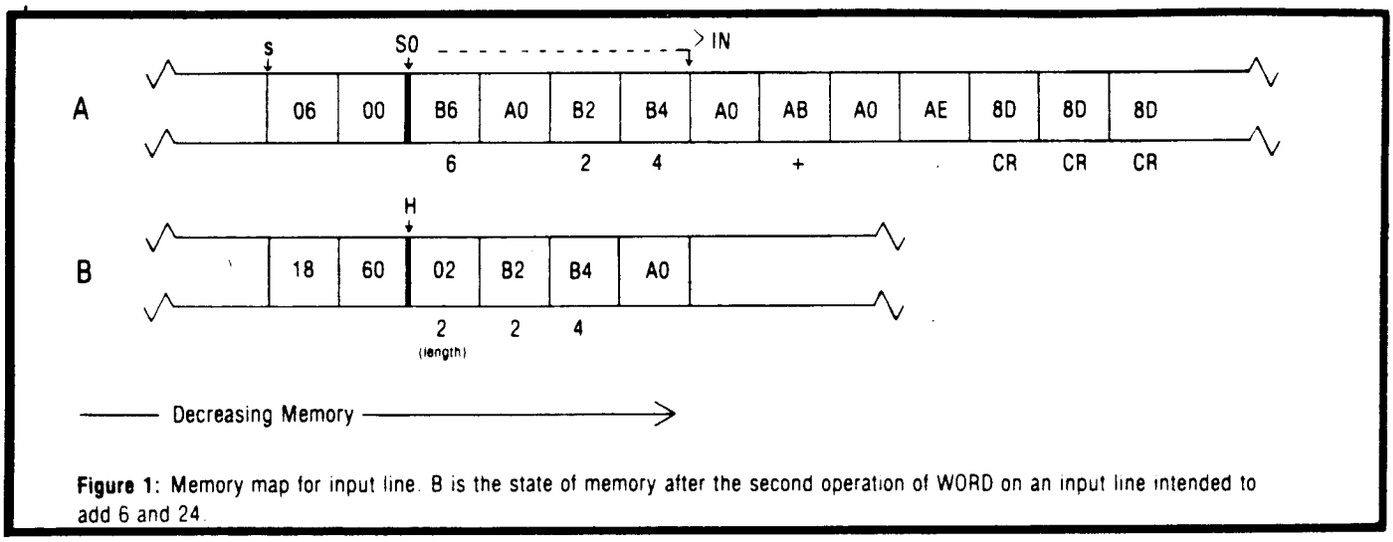


Figure 1: Memory map for input line. B is the state of memory after the second operation of WORD on an input line intended to add 6 and 24.

will be assumed to be stored as two bytes, low byte on top, pointing to the byte before the one to which they are to return control.

—' This word searches the dictionary for the word named by the next sequence of characters in the input line. It first calls WORD, giving a space as the separation character, and from then on deals with the sequence of characters stored, preceded by its length, at the location pointed to by H. The value of H left on the stack by WORD is destroyed. The search starts with the word pointed to by CURRENT and proceeds down link by link until it reaches a zero link. Up to the first three characters of each word are compared with the characters at H for length (ignoring the high bit; the high bit of the length byte is reserved for a special purpose). If a match is found, a false flag is returned as TOS and the PFA of the matching word as NOS. If no match is found, a true flag is returned as TOS and the value of H as NOS.

?STACK compares the value of S with that of S0; if the former is greater than or equal to the latter, a true flag is returned as TOS, otherwise a false flag. This word will be used every so often to check for stack underflow.

HERE returns the value of H as TOS.

CLEAR clears both the R-stack and the microprocessor stack; it sets R and the microprocessor stack pointer to their initial values. This routine will be called before each input line is processed.

SMUDGE toggles the sixth bit of the length byte of the word on top of the dictionary. This will later be used when a word is being created, to make sure that a word cannot be used while it is still half-finished.

?FORGET checks the sixth bit of the length byte of the word on top of the dictionary. If the bit is low, it does nothing. If the bit is high, that word is "forgotten"; CURRENT is set to the link of the word, and H is set to CURRENT. This effectively cuts the word out of the dictionary and frees the space it took up.

POP, when called by another word, allows that other word, when it is done, to return control not to the word that called it but to the word that called that word. That is, it removes from the microprocessor stack not the *top* two bytes (since those refer to the word that called POP), but the *next* two bytes.

Number output is a special process unto itself. At least one byte, to be called BASE, will be needed to store the base in which number input and output is to be conducted. In number output the main stack will be used to store each digit to be output. Routines are needed to supply each digit; essentially, the number will be repeatedly divided by BASE, with the remainders being the digits. The word /MOD is (not entirely accidentally) ideal for this; it leaves the remainder as NOS and what is left of the number as TOS. The digits may be interspersed as desired with ASCII characters to be output. A routine is finally needed to output all the digits and characters.

< # This word starts the process of converting a number to be output. It requires that the sign to be used be in NOS, and that the absolute value to be output be in TOS. It takes the sign of NOS and places it in a special location, then replaces NOS by $FFFF (note that this violates the usual stack conventions). From the time this word is called until the time # > is called, the stack will contain a $FFFF, then a sequence of digits and ASCII characters, rightmost digit deepest in the stack, with whatever is left of the number to be output remaining on top of the stack.

converts one digit (moving from right to left). It places the value of BASE on the stack and then jumps to /MOD; /MOD leaves the remainder of the division, which is the digit, as NOS, and keeps what is left of the number being output on top of the stack.

#S converts digits until there are no more left to convert. It calls # (at least once) until TOS is zero.

 # > ends the number output process by printing the number. It drops the TOS, then prints the new TOS: it takes the lower byte of TOS; if its high bit is set, the byte is printed as ASCII; if the high bit is low, the byte is considered as a digit and is converted into ASCII, then printed. The process is repeated until the $FFFF entry left by <# is reached; this $FFFF is dropped.

SIGN places a negative sign into the output conversion stream if the number being converted is negative. It takes the sign from where # left it; if it is positive, nothing is done. If it is negative, a negative sign ($AD) is put on the stack, then swapped into NOS.

HOLD takes a given single byte from TOS, considers it as an ASCII code, and places it into the output conversion stream. It simply jumps to SWAP; SWAP will leave the ASCII byte (from TOS) in NOS, and what is left of the number being output (pushed into NOS by the ASCII byte) back on top of the stack.

These routines may be combined as desired to produce various forms of number output; perhaps it will clarify matters to note that the sequence used for standard output is DUP ABS <# #S SIGN #> SPACE. The sequence between the <# and the #> may be regarded as a reversed picture of the output. Thus, for example, the sequence DUP <# # # $AE HOLD #S $A4 HOLD #> would print a number in dollars and cents format: # # gives two digits after the decimal point, $AE HOLD gives a decimal point, #S gives the rest of the digits, and $A4 HOLD gives a dollar sign.

Number input is somewhat less complicated, but it does involve a single complex routine. A sequence of characters, when taken off an input line, will first be checked to see if it is the name of a word; >BINARY will then be used to try to convert it into a number.

> BINARY takes a string of characters stored, preceded by its length, at the location given as TOS, and tries to convert it to a number. This number is then accumulated with the number, usually zero, given in NOS. It first places a space, which can never be taken for a digit, at the end of the string of characters. It then checks to see if the first character is a negative sign; if it is, it is saved. Each character is checked in turn; if it is not a digit in the BASE system, conversion is stopped. If it is a digit, the previously accumulated total is multiplied by BASE and the digit is

added on. When conversion is done, the accumulated total is given the proper sign and returned in NOS; the address of the first unconvertable character is returned in TOS.

The use of some of these routines may seem unclear now, but it should become clear when the secondary words that use all of the above are presented. Again, each of these words can be tested in isolation to make sure that it handles its inputs and outputs as specified. The next installment of this article will deal with the words required to implement some of the special functions used in secondary words.

```
                *
                ** EXPECT **
                *
0D82: 06 C5 D8 D0 68 0D
0D88: A0 01             LDY #$01    ; put TOS in SCR
0D8A: B1 00             LDA (S),Y
0D8C: 85 19             STA SCRH
0D8E: 88                DEY
0D8F: B1 00             LDA (S),Y
0D91: 85 18             STA SCRL
0D93: 20 09 08          JSR DROP
0D96: 20 6F FD          JSR GETLN1  ; get an input line
0D99: 8A                TXA         ; starting at $200
0D9A: A8                TAY         ; with length in X
0D9B: C8                INY
0D9C: A9 8D             LDA #$8D    ; put carriage returns
0D9E: 91 18             STA (SCR),Y ; at the end
0DA0: C8                INY         ; (one already there
0DA1: 91 18             STA (SCR),Y ; plus two more)
0DA3: 88                DEY
0DA4: 88        LOOP    DEY
0DA5: B9 00 02          LDA $0200,Y ; move input line
0DA8: 91 18             STA (SCR),Y ; to where SCR points
0DAA: 98                TYA
0DAB: D0 F7             BNE LOOP
0DAD: 60                RTS
                *
                ** CR **
                *
0DAE: 02 C3 D2 A0 82 0D
0DB4: 4C 8E FD          JMP CROUT
                *
                ** SPACE **
                *
0DB7: 05 D3 D0 C1 AE 0D
0DBD: A9 A0             LDA #$A0    ; output space
0DBF: 4C ED FD          JMP COUT
                *
                ** WORD **
                *
0DC2: 04 D7 CF D2 B7 0D
0DC8: A0 00             LDY #$00
0DCA: B1 00             LDA (S),Y   ; get separation
0DCC: 85 18     WORD2   STA SCRL    ; character
0DCE: 18                CLC
0DCF: A5 0C             LDA S0L     ; start at
0DD1: 65 06             ADC >INL    ; S0+>IN
0DD3: 85 16             STA ACC.C2L
0DD5: A5 0D             LDA S0L
0DD7: 65 07             ADC >INH
0DD9: 85 17             STA ACC.C2H
0DDB: 18                CLC
0DDC: A5 04             LDA HL      ; move the
0DDE: 69 01             ADC #$01    ; string to H+1
0DE0: 85 14             STA ACC.C1L
0DE2: A5 05             LDA HH
0DE4: 69 00             ADC #$00
0DE6: 85 15             STA ACC.C1H
0DE8: A2 00             LDX #$00
0DEA: A1 16     LOOP    LDA (ACC.C2,X)
0DEC: C5 18             CMP SCRL    ; is it not a
0DEE: D0 0C             BNE NOSPACE ; separation character?
0DF0: C9 8D             CMP #$8D    ; is it a termination
0DF2: F0 1D             BEQ END     ; character?
0DF4: E6 16             INC ACC.C2L ; it is a separation
0DF6: D0 F2             BNE LOOP    ; character, proceed
0DF8: E6 17             INC ACC.C2H
0DFA: D0 F2             BNE LOOP
0DFC: 91 14     NOSPACE STA (ACC.C1),Y
0DFE: C8                INY         ; we hit a non-
0DFF: F0 10             BEQ END     ; separation character
0E01: E6 16             INC ACC.C2L ; start transfer
0E03: D0 02             BNE OK1
0E05: E6 17             INC ACC.C2H
0E07: A1 16     OK1     LDA (ACC.C2,X)
0E09: C9 8D             CMP #$8D    ; is it a termination
0E0B: F0 04             BEQ END     ; character?
0E0D: C5 18             CMP SCRL    ; loop if not a
0E0F: D0 EB             BNE NOSPACE ; separation character
0E11: 98        END     TYA         ; we're done
0E12: 81 04             STA (H,X)   ; store length
0E14: 38                SEC         ; at beginning
0E15: A5 16             LDA ACC.C2L ; correct >IN
0E17: E5 0C             SBC S0L
0E19: 85 06             STA >INL
0E1B: A5 17             LDA ACC.C2H
0E1D: E5 0D             SBC S0H
0E1F: 85 07             STA >INH
0E21: A0 01             LDY #$01    ; return H
0E23: A5 05             LDA HH
0E25: 91 00             STA (S),Y
0E27: 88                DEY
0E28: A5 04             LDA HL
0E2A: 91 00             STA (S),Y
0E2C: 60                RTS
                *
                ** -' **
                *
0E2D: 02 AD A7 A0 C2 0D
0E33: A5 00             LDA SL      ; we must return
0E35: 38                SEC         ; two stack items
0E36: E9 04             SBC #$04
0E38: 85 00             STA SL
0E3A: B0 02             BCS OK1
0E3C: C6 01             DEC SH
0E3E: A9 A0     OK1     LDA #$A0    ; use space for
0E40: A0 00             LDY #$00    ; separation character
0E42: 20 CC 0D          JSR WORD2   ; and get next string
0E45: B1 04             LDA (H),Y   ; take min(length,3)
0E47: C9 04             CMP #$04
0E49: 90 02             BCC OK2
0E4B: A9 03             LDA #$03
0E4D: 85 18     OK2     STA SCRL
0E4F: A5 0A             LDA CURRENTL
0E51: 85 16             STA ACC.C2L ; start at CURRENT
0E53: A5 0B             LDA CURRENTH
0E55: 85 17             STA ACC.C2H
0E57: A5 17     MAIN    LDA ACC.C2H
0E59: F0 38             BEQ NOPE    ; is link zero?
0E5B: A4 18             LDY SCRL    ; compare at most
0E5D: B1 16     LOOP    LDA (ACC.C2),Y
0E5F: D1 04             CMP (H),Y   ; three characters
0E61: D0 21             BNE NEXT
0E63: 88                DEY
0E64: D0 F7             BNE LOOP
0E66: B1 16             LDA (ACC.C2),Y
0E68: 29 7F             AND #$7F    ; ignore high bit
0E6A: D1 04             CMP (H),Y   ; when comparing
0E6C: D0 16             BNE NEXT    ; length
0E6E: 98                TYA         ; we have a match
0E6F: 91 00             STA (S),Y   ; return a false flag
0E71: C8                INY
0E72: 91 00             STA (S),Y
0E74: C8                INY
0E75: A5 16             LDA ACC.C2L ; and the PFA
0E77: 18                CLC
0E78: 69 06             ADC #$06
0E7A: 91 00             STA (S),Y
0E7C: C8                INY
0E7D: A5 17             LDA ACC.C2H
0E7F: 69 00             ADC #$00
0E81: 91 00             STA (S),Y
0E83: 60                RTS
0E84: A0 04     NEXT    LDY #$04    ; no match here
0E86: B1 16             LDA (ACC.C2),Y
0E88: AA                TAX         ; so take link
0E89: C8                INY         ; as new address
0E8A: B1 16             LDA (ACC.C2),Y
0E8C: 85 17             STA ACC.C2H
0E8E: 86 16             STX ACC.C2L
0E90: 18                CLC
0E91: 90 C4             BCC MAIN
0E93: A0 00     NOPE    LDY #$00    ; no match anywhere
0E95: A9 01             LDA #$01    ; return a true flag
0E97: 91 00             STA (S),Y
0E99: 98                TYA
0E9A: C8                INY
0E9B: 91 00             STA (S),Y
0E9D: C8                INY
0E9E: A5 04             LDA HL      ; and H
0EA0: 91 00             STA (S),Y
0EA2: C8                INY
0EA3: A5 05             LDA HH
0EA5: 91 00             STA (S),Y
0EA7: 60                RTS
                *
                ** ?STACK **
                *
0EA8: 06 BF D3 D4 2D 0E
0EAE: 38                SEC         ; compare S
0EAF: A5 00             LDA SL      ; with S0
0EB1: E5 0C             SBC S0L
0EB3: A5 01             LDA SH
0EB5: E5 0D             SBC S0H
0EB7: E6 00             INC SL
0EB9: C6 00             DEC SL
0EBB: D0 02             BNE OK1
0EBD: C6 01             DEC SH
0EBF: C6 00     OK1     DEC SL
0EC1: D0 02             BNE OK2
```

```
0EC3: C6 01             DEC SH
0EC5: C6 00    OK2      DEC SL
0EC7: A0 00             LDY #$00    ; return appropriate
0EC9: 98                TYA         ; flag
0ECA: 2A                ROL         ; (true=stack
underflow)
0ECB: 91 00             STA (S),Y
0ECD: 98                TYA
0ECE: C8                INY
0ECF: 91 00             STA (S),Y
0ED1: 60                RTS
                *
                ** HERE **
                *
0ED2: 04 C8 C5 D2 A8 0E
0ED8: E6 00             INC SL
0EDA: C6 00             DEC SL
0EDC: D0 02             BNE OK1
0EDE: C6 01             DEC SH
0EE0: C6 00    OK1      DEC SL
0EE2: D0 02             BNE OK2
0EE4: C6 01             DEC SH
0EE6: C6 00    OK2      DEC SL
0EE8: A0 00             LDY #$00
0EEA: A5 04             LDA HL      ; return H
0EEC: 91 00             STA (S),Y   ; as TOS
0EEE: C8                INY
0EEF: A5 05             LDA HH
0EF1: 91 00             STA (S),Y
0EF3: 60                RTS
                *
                ** CLEAR **
                *
0EF4: 05 C3 CC C5 D2 0E
0EFA: A9 FE             LDA #$FE
0EFC: 85 02             STA RL      ; restore R
0EFE: A9 91             LDA #$91
0F00: 85 03             STA RH
0F02: 68                PLA         ; preserve
0F03: A8                TAY         ; return to
0F04: 68                PLA         ; whatever called this
0F05: A2 FF             LDX #$FF    ; otherwise restore
0F07: 9A                TXS         ; microprocessor stack
0F08: 48                PHA
0F09: 98                TYA
0F0A: 48                PHA
0F0B: 60                RTS
                *
                ** SMUDGE **
                *
0F0C: 06 D3 CD D5 F4 0E
0F12: A0 00             LDY #$00
0F14: B1 0A             LDA (CURRENT),Y
0F16: 49 40             EOR #$40    ; toggle sixth bit
0F18: 91 0A             STA (CURRENT),Y
0F1A: 60                RTS
                *
                ** ?FORGET **
                *
0F1B: 07 BF C6 CF 0C 0F
0F21: A0 00             LDY #$00
0F23: B1 0A             LDA (CURRENT),Y
0F25: 29 40             AND #$40    ; check sixth bit
0F27: F0 14             BEQ NOPE
0F29: A5 0A             LDA CURRENTL
0F2B: 85 04             STA HL      ; it's high
0F2D: A5 0B             LDA CURRENTH
0F2F: 85 05             STA HH      ; CURRENT->H
0F31: A0 04             LDY #$04    ; link->CURRENT
0F33: B1 0A             LDA (CURRENT),Y
0F35: AA                TAX
0F36: C8                INY
0F37: B1 0A             LDA (CURRENT),Y
0F39: 85 0B             STA CURRENTH
0F3B: 86 0A             STX CURRENTL
0F3D: 60                RTS
                *
                ** POP **
                *
0F3E: 03 D0 CF D0 1B 0F
0F44: 68                PLA         ; preserve
0F45: AA                TAX         ; return to
0F46: 68                PLA         ; whatever called
0F47: A8                TAY         ; this,
0F48: 68                PLA         ; destroy next
0F49: 68                PLA         ; reference
0F4A: 98                TYA
0F4B: 48                PHA
0F4C: 8A                TXA
0F4D: 48                PHA
0F4E: 60                RTS
                *
                ** <# **
                *
0F4F: 02 BC A3 A0 3E 0F
0F55: A0 03             LDY #$03    ; get sign
0F57: B1 00             LDA (S),Y   ; of NOS
0F59: 85 FF             STA SIGN
0F5B: A9 FF             LDA #$FF    ; replace NOS
0F5D: 91 00             STA (S),Y   ; with $FFFF
0F5F: 88                DEY
0F60: 91 00             STA (S),Y
0F62: 60                RTS
                *
                ** # **
                *
0F63: 01 A3 A0 A0 4F 0F
0F69: E6 00             INC SL
0F6B: C6 00             DEC SL
0F6D: D0 02             BNE OK1
0F6F: C6 01             DEC SH
0F71: C6 00    OK1      DEC SL
0F73: D0 02             BNE OK2
0F75: C6 01             DEC SH
0F77: C6 00    OK2      DEC SL
0F79: A0 00             LDY #$00
0F7B: A5 08             LDA BASEL   ; perform the
0F7D: 91 00             STA (S),Y   ; equivalent of
0F7F: 98                TYA         ; BASE @ /MOD
0F80: C8                INY
0F81: 91 00             STA (S),Y
0F83: 4C 16 0C          JMP /MOD
                *
                ** #S **
                *
0F86: 02 A3 D3 A0 63 0F
0F8C: 20 69 0F LOOP     JSR #       ; perform #
0F8F: A0 00             LDY #$00
0F91: B1 00             LDA (S),Y   ; until zero
0F93: D0 F7             BNE LOOP
0F95: C8                INY
0F96: B1 00             LDA (S),Y
0F98: D0 F2             BNE LOOP
0F9A: 60                RTS
                *
                ** #> **
                *
0F9B: 02 A3 BE A0 86 0F
0FA1: A0 00             LDY #$00
0FA3: 20 09 0B LOOP     JSR DROP    ; drop TOS
0FA6: B1 00             LDA (S),Y   ; have we hit
0FA8: C9 FF             CMP #$FF    ; the $FFFF?
0FAA: F0 13             BEQ DONE
0FAC: C9 00             CMP #$00    ; no, output
0FAE: 30 09             BMI OK      ; is high bit set?
0FB0: 18                CLC         ; no, convert to ASCII
0FB1: C9 B0             ADC #$B0
0FB3: C9 BA             CMP #$BA
0FB5: 90 02             BCC OK
0FB7: 69 06             ADC #$06
0FB9: 20 ED FD          JSR COUT    ; output
0FBC: 18                CLC
0FBD: 90 E4             BCC LOOP
0FBF: 4C 09 0B DONE     JMP DROP    ; done, drop $FFFF
                *
                ** SIGN **
                *
0FC2: 04 D3 C9 C7 9B 0F
0FC8: A5 FF             LDA SIGN    ; is it negative?
0FCA: 10 1D             BPL NOPE
0FCC: E6 00             INC SL      ; yes, HOLD a "-"
0FCE: C6 00             DEC SL
0FD0: D0 02             BNE OK1
0FD2: C6 01             DEC SH
0FD4: C6 00    OK1      DEC SL
0FD6: D0 02             BNE OK2
0FD8: C6 01             DEC SH
0FDA: C6 00    OK2      DEC SL
0FDC: A0 00             LDY #$00
0FDE: A9 AD             LDA #$AD    ; $AD="-"
0FE0: 91 00             STA (S),Y
0FE2: 98                TYA
0FE3: C8                INY
0FE4: 91 00             STA (S),Y
0FE6: 4C 69 0B          JMP SWAP    ; swap into NOS
0FE9: 60       NOPE     RTS
                *
                ** HOLD **
                *
0FEA: 04 C8 CF CC C2 0F
0FF0: 4C 69 0B          JMP SWAP    ; HOLD=SWAP
                *
                ** >BINARY **
                *
0FF3: 07 BE C2 C9 EA 0F
0FF9: A0 01             LDY #$01
0FFB: B1 00             LDA (S),Y   ; location at TOS
0FFD: 85 17             STA ACC.C2H
0FFF: 88                DEY
1000: B1 00             LDA (S),Y
1002: 85 16             STA ACC.C2L
1004: B1 16             LDA (ACC.C2),Y
1006: A8                TAY
1007: C8                INY
1008: A9 A0             LDA #$A0    ; put space at end
100A: 91 16             STA (ACC.C2),Y
100C: E6 16             INC ACC.C2L
100E: D0 02             BNE OK1
1010: E6 17             INC ACC.C2H
1012: A0 00    OK1      LDY #$00
```

```
1014: B1 16             LDA (ACC.C2),Y
1016: C9 AD             CMP #$AD      ; do we have a
1018: D0 07             BNE PLUS1     ; negative sign?
101A: E6 16             INC ACC.C2L   ; yes, push a zero
101E: E6 17             INC ACC.C2H
1020: 98        OK2     TYA
1021: 48        PLUS1   PHA           ; push something anyway
1022: A0 00     MAIN    LDY #$00
1024: B1 16             LDA (ACC.C2),Y
1026: 38                SEC
1027: E9 B0             SBC #$B0      ; convert from ASCII
1029: C9 0A             CMP #$0A
102B: 90 02             BCC OK3
102D: E9 07             SBC #$07
102F: C5 08     OK3     CMP BASEL     ; is it a digit?
1031: B0 32             BCS DONE
1033: 48                PHA           ; yes, multiply
1034: A5 08             LDA BASEL     ; the accumulated
1036: 91 00             STA (S),Y     ; number by BASE
1038: 98                TYA
1039: C8                INY
103A: 91 00             STA (S),Y
103C: 20 F2 0B          JSR *
103F: 68                PLA           ; and add the digit
1040: 18                CLC
1041: A0 00             LDY #$00
1043: 71 00             ADC (S),Y
1045: 91 00             STA (S),Y
1047: 98                TYA
1048: C8                INY
1049: 71 00             ADC (S),Y
104B: 91 00             STA (S),Y
104D: E6 00             INC SL
104F: C6 00             DEC SL
1051: D0 02             BNE OK4
1053: C6 01             DEC SH
1055: C6 00     OK4     DEC SL
1057: D0 02             BNE OK5
1059: C6 01             DEC SH
105B: C6 00     OK5     DEC SL
105D: E6 16             INC ACC.C2L
105F: D0 C1             BNE MAIN      ; and loop
1061: E6 17             INC ACC.C2H
1063: D0 BD             BNE MAIN
1065: 68        DONE    PLA           ; no more digits
1066: D0 19             BNE PLUS2     ; was it negative?
1068: A0 03             LDY #$03      ; yes, negate it
106A: B1 00             LDA (S),Y
106C: 49 FF             EOR #$FF
106E: 91 00             STA (S),Y
1070: 88                DEY
1071: B1 00             LDA (S),Y
1073: 49 FF             EOR #$FF
1075: 18                CLC
1076: 69 01             ADC #$01
1078: 91 00             STA (S),Y
107A: C8                INY
107B: B1 00             LDA (S),Y
107D: 69 00             ADC #$00
107F: 91 00             STA (S),Y
1081: A0 01     PLUS2   LDY #$01      ; return address
1083: A5 17             LDA ACC.C2H   ; of first
1085: 91 00             STA (S),Y     ; unconvertible
1087: 88                DEY           ; character
1088: A5 16             LDA ACC.C2L
108A: 91 00             STA (S),Y
108C: 60                RTS                              ■
```

Multi-user, continued from page 17

"intelligence" which is user configurable for different purposes from the keyboard. The configuration is stored in non-volatile RAM and remains in effect until changed by the user. StationMate is essentially the same product with a built in modem. Complexx will provide brochures which list prices and illustrate several of the possible uses of the products. Contact Complexx Systems, Inc., 4930 Research Drive, Huntsville, AL 35805.

Multi-user Software

In an earlier issue we mentioned CP/NET, Digital Research's "software network." At another time we talked about North Star's multi-processor system. Both of these products utilize what might be called multi-user operating systems.

North Star uses TurboDos, a product by Software 2000, Inc, of Arroyo Grande CA. TurboDos is compatible with CP/M applications software, and is said to be more compatible with such programs than is the multi-user version of CP/M itself. In any event, it is an enhanced system which bears a strong resemblance to CP/M insofar as commands are concerned.

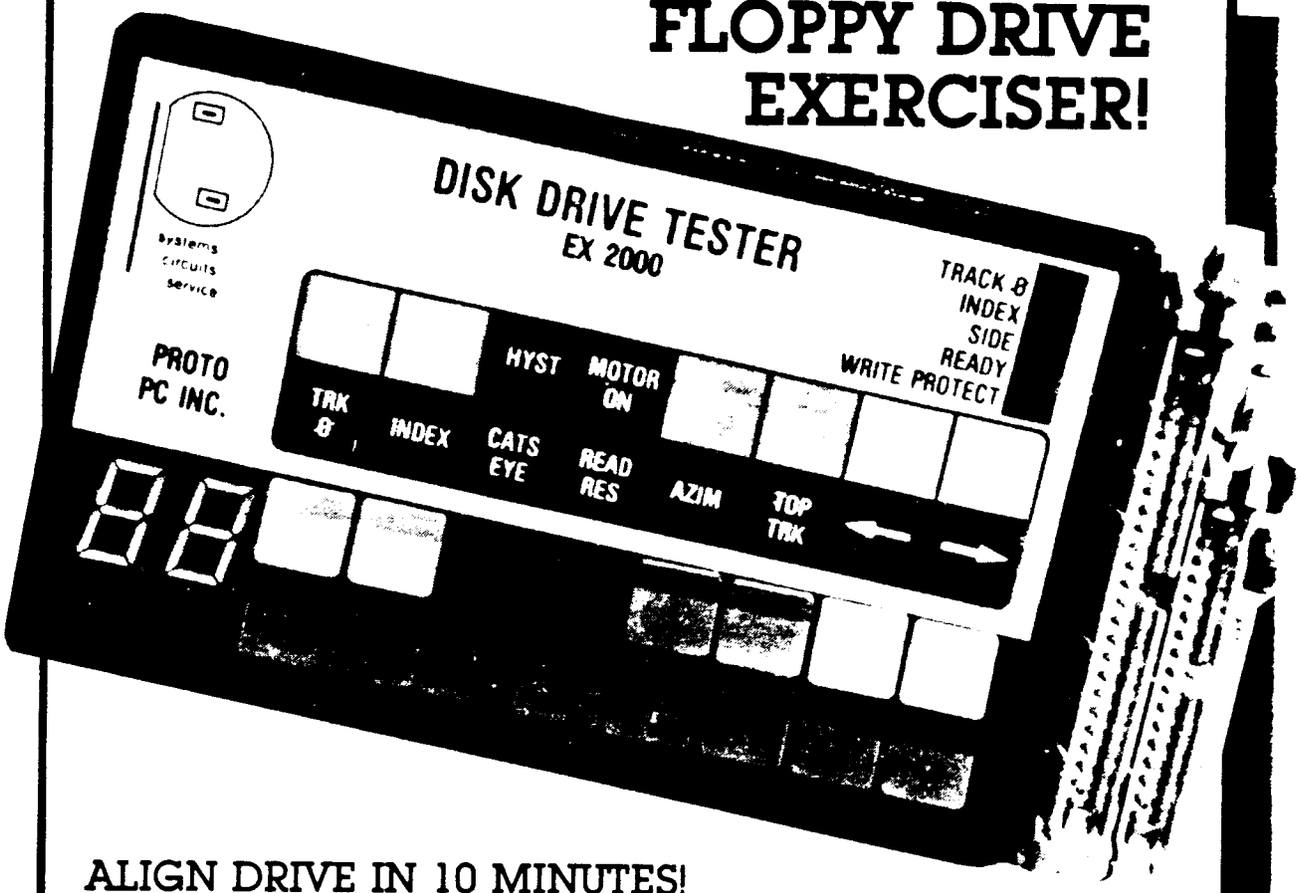
Software 2000 states that the product will work with Z-80 based systens as well as with the 8086/8088 family of CPUs. It supports bank-switching and up to 128K of memory in an 8-bit environment (proportionately larger in 16-bit memories). It was specifically designed for multi-user applications and the enhanced commands permit such features as record locking and linking with peripherals located elsewhere in the network.

Software 2000 does not sell directly to end users, but markets TurboDos through OEMs and distributors (you can obtain a list of distributors by writing the company). One of their distributors is North Star, who customized a special version for their own use with their new Horizon 8/16 system.

We've noticed an increase in advertisements for software packages that connect micros (such as Apples) to large mainframe computers (such as the IBM 370). Programs of this kind would necessarily be fairly complex, as they must involve protocol conversion as well as conversion of file formats. So far there has been no opportunity to review any of these products, which certainly fall into the multi-user category.

We are interested in hearing from readers who have had experience, either good or bad, with commercial or "homebrew" multi-user applications. We also welcome your questions, and will attempt to research and report on any systems about which you are curious. ■

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

MAKE A SIMPLE TTL LOGIC TESTER

by E. G. Brooner

In the "old days" we all knew that although a tube or transistor tester was no substitute for good troubleshooting, it could at least narrow the job down a bit. Strangely enough, a similar treatment of silicon chips has seldom gotten beyond the factory level; we all tend to check these devices by substitution. On many occasions I had wished for some kind of chip tester, and when I dropped 110 volts AC across my data lines one day I decided the time had come to build one. All of my boards are buffered, and I knew that a great many support chips had been damaged, if not completely destroyed. The little tester described in the following paragraphs is the result of that series of events.

Theory

A brief review of simple TTL gates will help to explain how the tester works. The discusssion ignores the more complex TTL chips and is confined to one-input gates (AND, OR, NAND, NOT, and XOR) and tristate buffers. These logic blocks make up a large majority of the chips we use, and are usually the interfaces between boards and buses. As a result, they are the ones that are most often questionable.

The one-input driver or inverter is the easiest block to understand and to test. The output is either a duplicate of the input (driver) or the inverse of the input (inverter). These gates usually come six to a chip. Two-input gates are slightly harder to understand, mainly because there are so many different kinds. The so-called "AND" circuit requires that both inputs be "true" to yield a "true" output. The "OR" circuit will give the proper output if either of the inputs is "true." Not-and (abreviated NAND) and not-or (NOR) are in effect the same devices with an inversion added. Exclusive-or (XOR) is only a little more complicated—it yields a true output if *either* of the inputs is true, but not if they are both true.

Tristate buffers/inverters are in effect single input gates which have a very high impedance if not "enabled." The enable pulse can be thought of as another input, whose absence essentially disconnects the gate(s) from the circuit. This third state prevents any input, either true or not-true, from affecting the output, and accounts for the name "tristate."

It can be seen that to test any of these logic blocks, we need a socket into which the chip can be plugged, a source of voltage, some means of applying input signals and "enable" signals, and a way of telling what state is present at the output. This is a simple concept, and the only real complications come from the fact that the socket connections differ from chip to chip.

Figure 1 shows what is commonly referred to as a "truth table." This particular table is for a 7400 NAND gate. A and B are the inputs and X represents the output. It can be readily seen (top row) that a 1 on both A and B will yield a 0 on X. All other combinations of inputs result in a 1 on the X output. Had this two-input gate been an AND gate, the output would not be reversed, and two 1s would have resulted in a 1 output. Similar tables can be drawn for all multi-input gates, and are often found in data handbooks. The truth table is the basis for logic design and, in our case, for the construction of the tester.

A	B	X
1	1	0
1	0	1
0	1	1
0	0	1

Figure 1: 7400 Truth Table.

The Basic Circuit

Figure 2 is the complete schematic of the tester. Before trying to trace the wiring, look for a moment at Figure 3. This is a simplified drawing which shows the connections that exist when the instrument is set up for the 7400 NAND gate described in the preceding section. The A inputs for all four of the gates, contained in one 7400 chip, are connected together. All of the Bs are likewise connected. The pull-up resistors cause a high or logic 1 to be applied to the inputs Looking at the truth table, we see that this should result in a low, or 0, on each output. The outputs can be connected one at a time, to the logic probe circuitry which will light one light (LED 3) if high and the other if low; by switching the outputs (with SW C) we can test row one of the truth table for each gate.

Now if we depress either the A or the B pushbutton switch, we can put a low, or 0, on the associated inputs. In this way we test the performance for the other three possible states, as they are listed in rows two, three, and four of the table.

Simple, isn't it? Single input gates are even simpler—you just plug the chip into the correct socket, and alternately press and release the A button while switching from one output to the next. Other two-input gates, as listed earlier, test in exactly the same manner as did our example 7400, always using the truth table until you have them all memorized.

There remain the tristate buffers/inverters. They test as do any of the other single input gates, with one exception: there is an enable switch (SW 2) provided so that we can test them in both the enabled mode and in the tristate

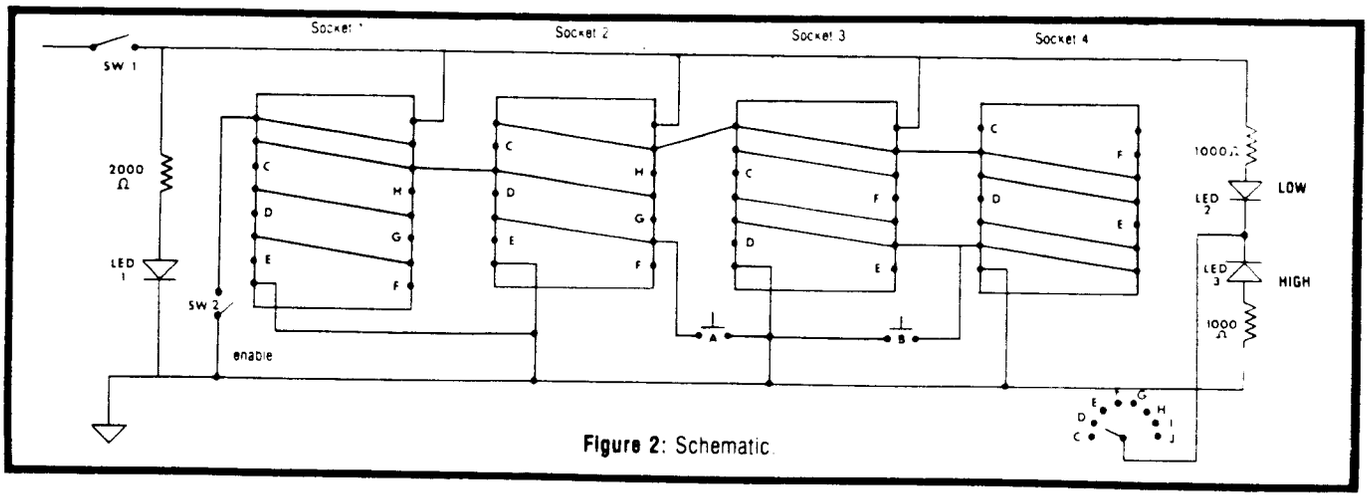


Figure 2: Schematic.

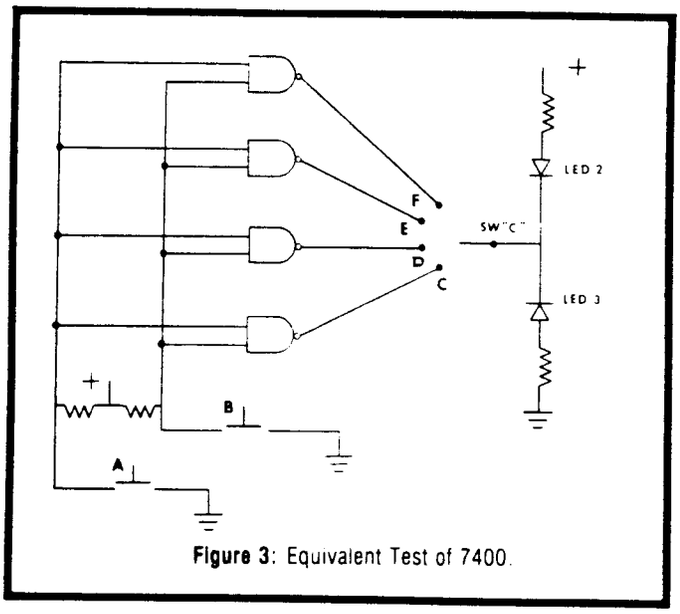


Figure 3: Equivalent Test of 7400.

mode. In the latter, of course, changing the inputs should have no effect on the outputs.

Now you can return to Figure 2 and see that the various sockets are all interconnected in some manner. The reason for this is that, unfortunately, manufacturers do not use the same base connections for all chips. Socket #1 fits all of the common tristates, and #2 fits all hex drivers and inverters. The two remaining sockets are for various two-input bases. The correlation between chips and sockets is shown by Figure 5.

Some Additional Comments

We have noted the variances between the "pinout" of various chips, and another caution is in order. The military 54XX series chips are usually compatible with the equivalent 74XX number. A data handbook should be consulted when there is any doubt.

Since the time that the photos were taken and the diagram drawn, a start has been made toward adding a test for the 8212 latch that is commonly used for interfacing. The photo shows a vacant space in the center of the panel which was reserved for this purpose. Since the heart of the 8212 is a group of eight tristate buffers, that part of the test is

3 each 14 pin wire wrap DIP sockets
1 each 16 pin wire wrap DIP socket
3 each low current LED. LED 1, 2, and 3
2 each SPST miniature toggle switch
2 each SPST pushbutton, normally open
1 each single pole, 6 or 8 position rotary switch
4 each 1000 ohm ¼ watt resistors
1 each 2000 ohm ¼ watt resistor
Box, panel, power source, and wirewrap materials

Figure 4: Parts list

easily accomplished by following the technique used for hex buffer/inverters. The task of testing all of the many enable options that go with this chip is not quite so simple; for this reason, the 8212 test will not be further elaborated and the prospective builder is encouraged to work that part out to his own satisfaction.

Construction

This prototype model was built in the easiest possible way, by wire-wrapping on a small piece of perfboard (see photos). The perfboard, in turn, had been cut to fit as the lid or panel on a small plastic box left over from another project. Construction could have taken almost any form, as size and layout are not critical.

There is also room for innovation in the power department. Three pen size batteries were used, simply because the holder was available and they fit the space in the box. Stability is more important than voltage, and the resulting 4.5 volts has proven adequate to test 5 volt chips. If desired, a builder could use an external, regulated AC supply, or take 5 volts from some existing piece of equipment. The current drain is negligible, though with small batteries it is not advisable to leave the tester turned on, with a chip inserted, any longer than necessary.

The photos are included to show the placement of the chip sockets and switches in this model; it should be understood that any layout that suits the builder is satisfactory. The construction parts are few and simple, and are listed in Figure 4.

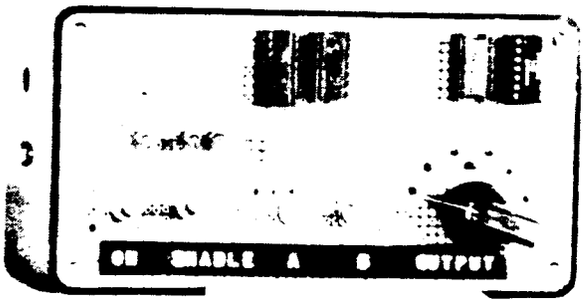


Photo 1: Front view of the tester. Some of the resistors shown here are not used in the working model.

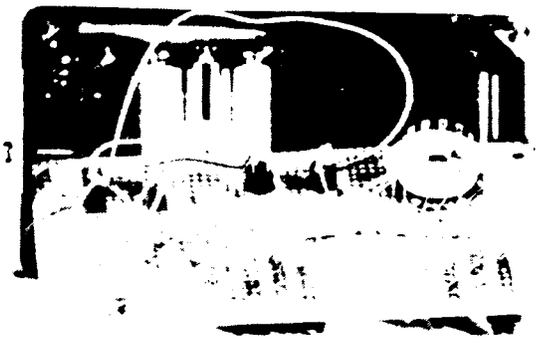


Photo 2: Inside view of the tester, showing the placement of major parts.

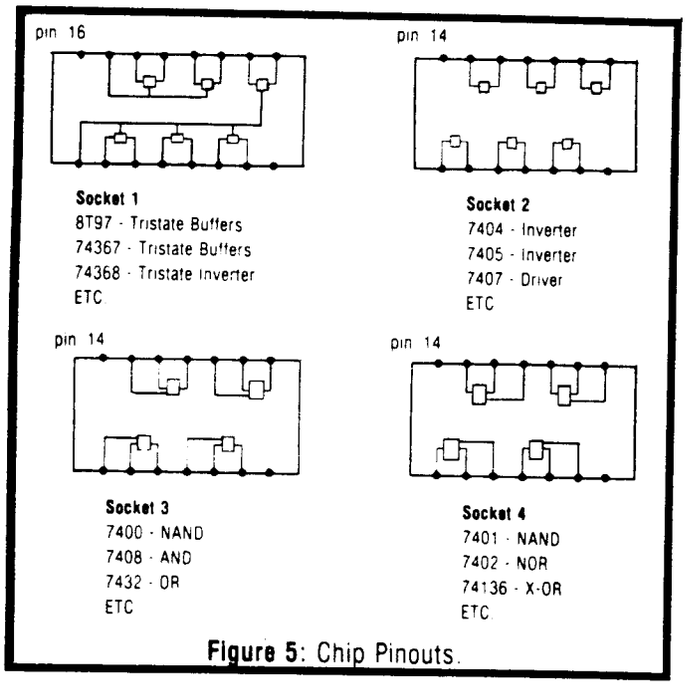


Figure 5: Chip Pinouts.

Operation

Rather than list all the chips that could conceivably be tested, we have duplicated in Figure 5 the "pinout" diagrams that go with the four sockets shown in the diagram, and listed "typical" chips that will fit that layout.

Notice that the chip diagrams differ from the actual layout that you will find in a data handbook. They have been drawn this way on purpose, and indicate only the *base connection* rather than the gates' *purpose*. To test a chip, look it up in a data handbook or working diagram, and match it to the proper socket. It does not matter, with the two-input gates, whether they are AND, NAND, OR, NOR or XOR. Remember, it is your interpretation of the logic states, from the truth table, that tests the chip. The tester is simply a convenient way of "getting it all together." If you know someone who is a bit hazy about how computer logic works (and that includes most of us), this device can provide an enlightening demonstration of the basic principles. ■

Interfacing Tips and Troubles, continued from page 13

regulator itself.

Summary

To summarize our discussion on power supply noise: always use proper construction techniques as a first attempt to reduce noise problems. 1) Make all power supply lines as short as possible, 2) route supply lines around oscillator sections in the circuit, and 3) never "daisy chain" power supply lines. If noise problems persist, conduct two measurements: output voltage and noise component. The output voltage should be between 4.5 and 6 volts. The amplitude of the noise component should be no more than 0.2 volts peak to peak. If the noise component is excessive, the problem may be alleviated by adding filtering capacitors, 0.22F capacitors on the regulator, or replacing the regulator. If all else fails, use a different power supply altogether.

In the next installment of "Interfacing Tips and Troubles," we will look at noise generated by, or because of, the interface and/or computer circuitry. We will also discuss noise problems created by outside sources. In addition, I will tell the story of a strange and unusual noise experience I had that I am sure you will find amusing. ■

SUBMIT YOUR IDEAS FOR A READER DESIGN PROJECT

The Computer Journal will be publishing design projects so that our readers can cooperate to solve common problems. We welcome your project suggestions.

Watch the next issue for the first Reader Design Project!

STAFF PRODUCT REVIEW: MicroSolutions' "UniForm"

With the proliferation of different 5¼" diskette formats nowadays, moving information from one machine to another can be quite a problem, even if they both use 5¼" soft-sectored floppy diskettes. UniForm is one of several programs that have appeared recently which attempt to ease the problem. It is designed to run on a specific computer (the particular version we reviewed was for the Morrow Micro Decision) and can format, read, and write a couple of dozen different CP/M machines' diskettes. When using the MD2 only single-sided diskettes can be manipulated, but if you have the MD3, UniForm supports double-sided versions of most of the formats.

As an added bonus, UniForm can format, read and write IBM PC diskettes in both single-sided and double-sided versions, both DOS 1.x and 2.x. This feature in itself is almost worth the price of the package if you need to move software on and off a PC or compatible.

UniForm consists of three programs—the main program, an overlay file, and an installation program that customizes the program for your particular terminal. The programs are menu-driven and the user should find them very easy to use, as we did.

The manual that comes with this is small but complete. It is also realistic in listing some of the different formats that, because of their uniqueness, may not work completely correctly with Uniform.

There is only one minor annoyance in an otherwise good sofware package. In copying files between drives, the copy is apparently made on a sector-by-sector basis, resulting in an continuous clicking as the disk drives are alternately selected. Our feeling is that UniForm would present a more professional appearance if it were to buffer a larger portion of the file being transferred in memory, resulting in quieter operation and less wear and tear on the drives.

Other than this minor point, we found the program to be easy to use and effective. If you are having problems moving between diskettes, UniForm should help overcome them. Versions are available for a number of different hardware environments. UniForm is $69.95 from MicroSolutions, 125 S. Fourth Street, DeKalb, IL 60115. ■

New Products

Multi-Function RS232 Breakout Box

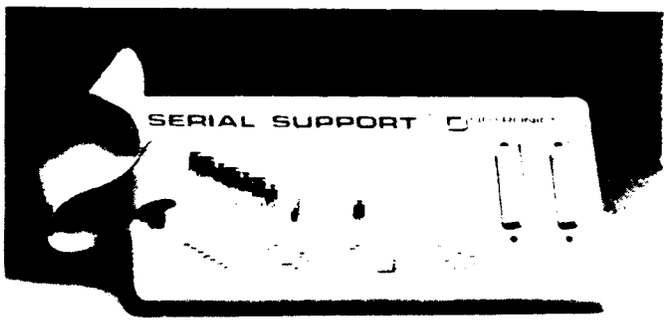
Optronics Technology has announced their Serial Support: a cable interface, a diagnostic tool and a port expansion in one new product.

Serial Support adapts output lines to any configuration with the unique "Versa-Matrix" system. Shunts are used instead of confusing and unreliable wires. Common jumper configurations are displayed on the front panel.

Tristate displays are used for true EIA RS232C signal voltage level validation, with green for greater than +3 volts and red for less than −3 volts.

The shared port feature allows two peripheral devices to share one port. A passive signal splitting design overcomes the need to manually switch from one peripheral to another.

Serial Support comes with a ribbon cable and is housed in a rugged case. It uses a low load terminal-powered design to eliminate the need for batteries. In stock for $133, including shipping, from Optronics Technology, 2990 Atlantic Ave., Penfield, NY 14526. ■

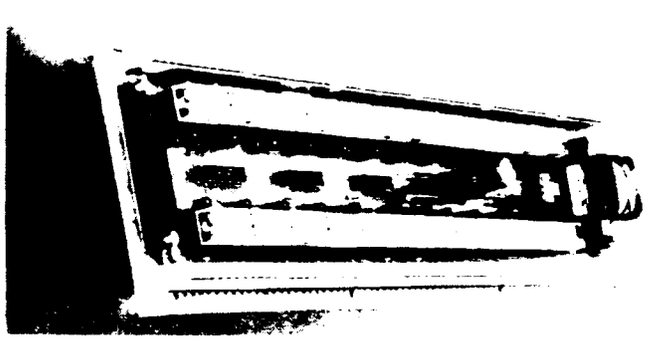


Measurement and Control with the IBM PC

MetraByte's 24 channel DPDT relay output accessory board for the IBM PC/XT offers the programmer 24 electromechanical, double-pole-double-throw relays for efficient switching of loads by programmed control. Each relay contains two N.C. and N.O. contacts for controlling up to a 3 amp resistive load at 120 Vrms per contact. Designed to be operated with MetraByte's 24 bit parallel digital I/O board, model PIO-12 ($97), it can also be operated on any digital output device providing TTL logic levels.

The relays offer the user zero leakage output currents, compared to the solid state relay alternative. There are 24 LEDs, one for each relay, which light when their associated relay is activated. The board contains its own power supply for relay driving and operates on 120/240Vac ±15% at 50½60Hz. Operating temperature is 0 to 70°C (32 to 158°F).

Single piece price, $395, in stock for immediate delivery from MetraByte Corporation, 254 Tosca Drive, Stoughton, MA 02072. ■



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"

SOLD OUT

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

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