

# THE COMPUTER JOURNAL

For Those Who Interface, Build, and Apply Micros

Vol ii, Mo 7

issue Humber 11

\$2.50 US

Putting the CP/M IOBYTE To Work page 2

Write Your Own Threaded Language:  
Part Three: Secondary Words page 7

Build A 68008 CPU Board For  
the S-1 Bus page 11

Writing and Evaluating  
Documentation page 24

Electronic Dial Indicator:  
A Reader Design Project page 28

# Editor's Page

## Where Do You Get Support When The Supplier Goes Bankrupt?

This week we received information about two more software companies which are closing their doors. This makes the problem of support hit close to home, since we had just acquired an accounting package from one of these companies. We have not paid too much attention to the forecasts of an impending shake-out in the microcomputer industry. We felt safe because the problems caused by these failures always happened to someone else, but now we wonder what is in store for the users during the next six to twelve months.

The industry has grown too large too quickly, and can not continue to support the me-too companies with look-alike products. There have already been some problems in the hardware market (Osborne and Franklin), and there are reports that several more manufacturers are about ready to file for Chapter 11. The problems are even more severe in the software field, where we expect a large number of failures by next spring. Other high risk areas are peripherals (drives, printers, boards, etc.) and IBM clones. There will be screaming and gnashing of teeth (and lawsuits) by users who can not get support for products which they have purchased.

Our readers will not be affected as severely as other microcomputer users because we are interested in knowing how our computers work and are able to solve many of our own problems—but we are in the minority. The largest growth area in the past two years has been in the area of business, where micros have been marketed as appliances which are as easy to use as a typewriter or a telephone. The businessperson uses the computer as a tool, and is not at all interested in understanding how it works. Such a user is attracted by the marketing image of a one-piece, plug-it-in and turn-it-on "appliance." Unfortunately, micros as they are sold today are not simple, one-piece packages. They are an assembly of boards, drives, interfaces, and software from many different vendors, any one of which could close their doors tomorrow, leaving the user without support.

The most notable exception to this is IBM. I do not feel that their machine is the best (and I do not like the fact that it is becoming the defacto standard because this curtails new developments), but at least a business user can get a complete package, including the software, from one source which is likely to remain in business. Another choice would be the Apple II because there are so many in use that the hardware would be supported by third parties, and the more

popular software programs for the Apple would either have user support groups or could be replaced.

We are concerned about the problems faced by users of other systems, and are attempting to acquire documentation from hardware and peripheral manufacturers in order to establish a permanent reference file for out-of-production equipment. The useful life of micros greatly exceeds the marketing life, so this file will include older models from companies still in business as well as information from companies which have gone out of business. If you have documentation from equipment which you no longer use, we would appreciate this literature as a donation to start our library.

The forthcoming shake-out will provide a number of opportunities for those of our readers who are able to interface, integrate systems, do some software patching, and perform minor hardware repairs or modifications. These readers will be able to start a business (either part time or full time) supporting the orphans, or pick up some real bargains because other people are not able to work with these unsupported systems.

*The Computer Journal* will help you cope with the support problems caused by bankruptcies by establishing a library of documentation and acting as a clearinghouse for information. Your input as letters, notes, sources, and articles is needed to make this successful.

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

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

*Entire contents copyright © 1984 by The Computer Journal*  
*Postmaster: Send address changes to: The Computer Journal*  
*P.O. Box 1697, Kalispell MT 599031697.*

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

# PUTTING THE CP/M IOBYTE TO WORK

by E. Brooner

Among the other marvels of CP/M is a single byte, 8 bits, located at address 3 in the standard version. This is known as the "IOBYTE." You may not have given this much thought if you have a rather straightforward, factory-configured system; if you do your own interfacing, though, it can be very useful. It is the means by which you can reconfigure your system from the keyboard. You might want someone at the other end of your modem to operate your system, for example, or you may want to be able to switch printers without any hassle.

CP/M assumes that you have four logical peripherals called Console, Tape punch, Tape reader, and List device. Control normally takes place from the console (CON) and printing is automatically routed to the list device (LST). In the good old days everyone had a paper tape punch and reader, we are lead to believe. Nowadays it is more common to find a video terminal and a single printer of some kind.

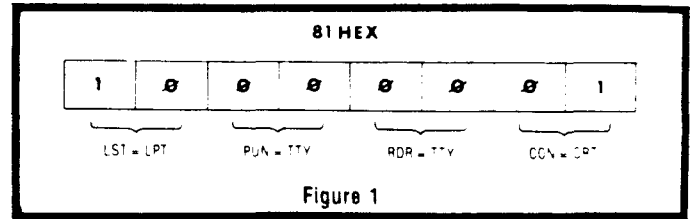
But what if you have more than two peripherals or even more than those four basic devices? Or what if you have two printers? Or two terminals? (Some people do.) Perhaps, too, you have a modem or some other device for input and/or output. You can have these extras by various means; the IOBYTE is one way to go about it. The other methods are hard-wiring, changing cables, or using an expensive switching arrangement.

## What the IOBYTE Does

Let's assume that you have a CP/M based computer at hand. You're probably familiar with the use of the STAT command to check disk space and file lengths. You may not have tried the STAT DEV: command. If you will enter that one you will get something like the following:

```
CON is TTY
RDR is TTY
PUN is TTY
LST is TTY
```

All this really says is that there is a 0 in the IOBYTE. This is the default value in a completely naked CP/M. The system simply read address three and, finding each bit a zero, concluded that all peripherals were TTY. It didn't care what they actually were, it simply described the configuration in its own terms. Looking back a few years this makes sense, because not too long ago a TTY (Teletype) was about the only I/O device available to hackers. All of your peripherals (if you had more than one) were probably TTYs. Actually many teletypes did have a built in punch and reader as well as the printer and keyboard, so they could be any of the four logical I/O devices.



Many CP/M systems default to a 95 hex in the IOBYTE, which will yield:

```
CON is CRT
RDR is PTR
PUN is PTP
LST is LPT
```

By entering the command STAT VAL: you get something like this:

```
CON:TTY: CRT: BAT: UC1:
RDR:TTY: PTR: URI: UR2:
PUN:TTY: PTP: UP1: UP2:
LST:TTY: CRT: LPT: ULI:
```

Which means that the CON(sole) *could* have any of the four assignments listed on the same line; any of four devices could be the RDR, and so on. Notice that some of the possible device names are input devices, some are output, and some (like the TTY) are capable of both. Thus there are some limits to the reassigning you can do: you can't, for example, 'read' input from a printer! The BAT (batch) function is really not a device as such; it is a way of routing continuous input from one device to another and has to be handled a bit differently than would a more conventional peripheral operation. So although it looks like 16 devices can be accommodated, this is not really true.

Now try the command STAT CON:UC1:

Entering STAT DEV: will now tell you that CON is UC1 rather than the CRT or TTY that was its previous designation. It does this because the previous command changed the value of the IOBYTE. If you change the assignment of the LST: device by means of another STAT command, the number changes again and the result of STAT DEV: changes accordingly.

But unless the IOBYTE has been implemented in your system nothing else will happen. The only 'real' connection between the logical I/O devices and the actual peripherals has to take place in software. This is sometimes referred to as linking the *logical* and *physical* devices, and the IOBYTE is the way it happens.

## How It Works

To follow the rest of this article you should have some familiarity with assembly language programming and CP/M; not a great deal, but just enough to find and modify your driver routines, which are a part of the BIOS.

If you have ever modified or customized your CP/M system you will be familiar with how these steps are performed. If not, it's a good exercise for learning about your operating system and the details are very likely buried somewhere in your documentation.

First, take the IOBYTE apart. If you'd like, you can use DDT to examine the contents of address 3 which will be the hex number stored in the IOBYTE. The 8 bits are divided into 4 2-bit groups, one group for each logical device. It really makes more sense to consider the byte as four separate '2-bit bytes.'

Ones and zeros in a 2-bit format can represent four values —0 through 3. A zero in any group means that device has been assigned as a TTY for purposes of the STAT commands. A zero in the CON group (the two least bits in the byte) means TTY, a one means CRT, and so on. In the highest bits, taken separately, zero again means TTY and one means CRT, two means LPT, and three some other device.

Figure 1 shows how the byte is divided in four portions for this purpose.

If we put a two in the high group and a one in the low group and then read the byte as one number, it will be 81 hex, and it will tell us (via STAT) that we have a CRT as console and an LPT as the list device. The other two devices are TTY because their groups contain all zeroes.

If this is not clear to you, read the last few paragraphs again and (as a last resort) consult your CP/M documentaion.

But still, nothing happens unless our driver routines can read the IOBYTE and find out where to go for input or output, as the case may be.

Actually, most CP/M systems will default (unless they've been modified) to either a zero or 95 hex in the IOBYTE. Zero says everything is TTY; 95 makes some other assignment. If you want something else, there are ways to change it: the method we prefer is to make the byte you want load as part of your initialization routine, on cold boot. It should load with a value that configures your system as you normally use it; you can then make other assignments with the STAT commands. Another fairly practical way of manipulating the byte —reading it and/or changing it—is from a running program in BASIC or whatever language you have, that can "peek and poke."

At this point we might want to refer to a flowchart or two. Figure 2 is the flowchart of a driver routine which ignores the IOBYTE and simply routes console calls to the terminal and print calls to a single printer. Figure 3 shows a driver which will consider the IOBYTE and route the I/O accordingly. We'll assume that your initialization routine (another jump in the jump table) has also been written to insert some significant number in the IOBYTE or that you have arranged to live with what your system already defaults to. In this case we'll assume that we are dealing

only with the print output.

At the beginning of the so-called "user area" in the BIOS there is a jump table. A call to the printer, for example, is routed to the LST vector, which in turn is a jump to the routine that actually outputs the character. In Figure 2 we have shown the call going straight from the jump table to the driver routine.

In Figure 3, we show the printer call being routed through another logic block, which reads the IOBYTE and, based on what it finds there, re-routes the information to any of two or more printers. Once this routine is in place, we can choose our printer simply by changing the IOBYTE from the keyboard. We would do this by STAT LST:TTY:, or STAT LST:LPT:, or whatever.

The same sort of thing has to be done for each logical device (LST, CON, RDR, or PUN) to which we want to add this versatility. RDR and PUN, incidentally, don't have to be paper tape devices. The punch might be another printer, or RDR and PUN might be the input and output to and from a modem.

Our publisher, who is interested in using things other than terminals and printers with a computer, has pointed out that some of the possible devices might be sensors or control systems of some kind. That's entirely possible and practical. You might want to boot up in the normal operating mode and at some point switch your system to monitor or control some kind of external process.

Another good purpose could be accomplished by putting your equipment in the basement or closet, with an extension terminal in the living room that you could move to once the system is running.

In a business the first printer might be a high speed dot matrix and the second a daisy wheel for word processing. The first console might be on the secretary's desk, and the second in the boss's office. Note that this would constitute a

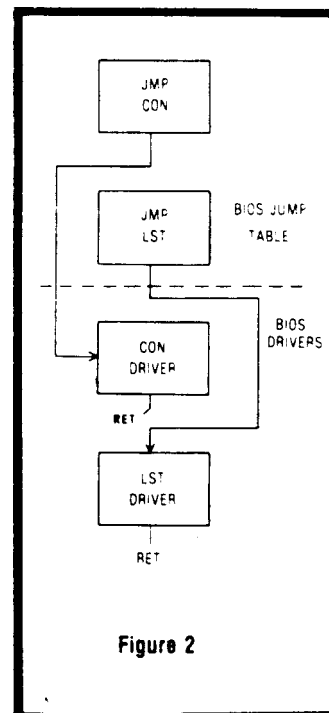


Figure 2

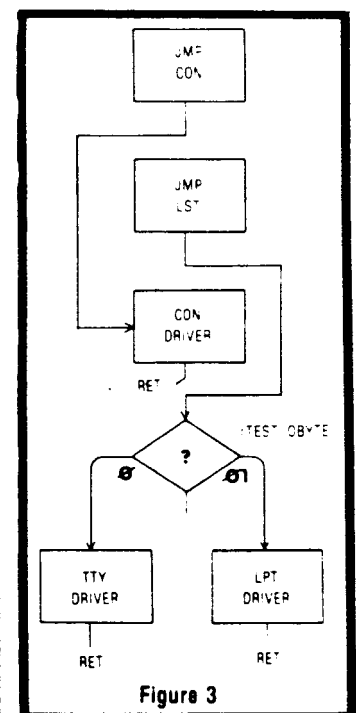


Figure 3

very cheap, simple multi-user system for two users. It actually only allows for one user at a time, but still does accommodate two users in some fashion.

### The Hardware/Software Changes

In order to do the things just discussed you need a port (serial or parallel as the case(s) might be) for each device to be connected. With an S-100 system you can plug in additional I/O boards until you run out of slots; some other computers might be more limited as to the number and type of available ports. That's the hardware, and of course each port has to be configured according to the documentation of the port itself and the device to be connected. This means setting the baud rate (for serial ports) and the status bits, and all that good stuff.

Once those things are established you will have to (carefully) modify or rewrite the driver routines in your BIOS. That's the software or fun part of the job, and the part we'll discuss next. Assume that we are dealing only with the printer or LST output for now and that you have a copy of the source listing for the user portion of the BIOS.

1. Locate the jump table. There will be a jump instruction which directs printed output to the print driver routine. This instruction will contain the address of the present printer driver.

2. Insert, in place of the driver, a routine to read and test the IOBYTE. One way to do this is to use the logical AND function to mask and look at only the appropriate portion of the byte; there are other ways that you might rather use. There are styles in programming and yours might be to rotate the byte through the accumulator or "line up" the bits you're checking.

3. Depending on the value of bits 7 and 6 in the IOBYTE, redirect the print output to one of (up to 4) new print routines.

4. At those addresses, put in the appropriate code for the port that will be used for each. To do this you will have to know the port address, ready bits, and so on, for each. They might all be the same or they might differ, depending on your hardware.

5. Reassemble the user portion of the BIOS and reinsert it into the CP/M system.

Perform the same steps for console or other devices that you might want to be able to reassign by use of the STAT commands. Always go directly from the jump table to a test routine, and from there to the appropriate driver.

Needless to say, when CP/M is implemented on a machine for which it was not designed, such as Apple or TRS-80, there will be some subtle differences. The object, of course, is to make the machine look the same as any other to CP/M application programs; to accommodate the foreign system, though, some compromises are usually made.

For example the Apple has no ports per se. The logical device drivers, then, must be caused to address expansion slots, or more likely, the memory addresses assigned to them. The peripheral cards are geared to 6502 conventions, i.e., input and output are by means of the load and store instructions rather than the IN and OUT instructions we

8080/Z80 people are used to. The nominal console (the keyboard and screen), have to be taken into account; some of the slots are dedicated and others should be available for reassignment. I am no Apple expert and welcome correction from readers if the following is in error.

As nearly as I can tell the Apple defaults to the "normal" console configuration unless an 80 column card is plugged into slot three. Slot three is nominally the "alternate console" and some magic in the system causes the machine to detect this addition and make the necessary patches to the drivers. This takes place automatically and does not change the IOBYTE.

Slot 1 is normally considered the LST device, or printer. An apparent difference here is that the Apple LST assignment seems able to handle both input and output rather than just output as is the normal case. There is also a persistent rumor that some of the available peripheral cards make direct calls to the 6502 ROM routines, as well as making other adjustments that are card-dependent. In other words Apple CP/M is based on the original version, but has some peculiar differences.

An unaltered Apple CP/M system (at least the one I looked at) has the IOBYTE filled with 95 hex, which presumably makes slot 1 the printer. Since there is an 80 column card occupying slot 3 it should be possible, somehow, to assign slot 2 to be another external device, and perhaps also one or more slots at 4 and above. The IOBYTE does not appear to be implemented in the one I examined, and I expect that some new driver routines would have to be written before it could be put to use.

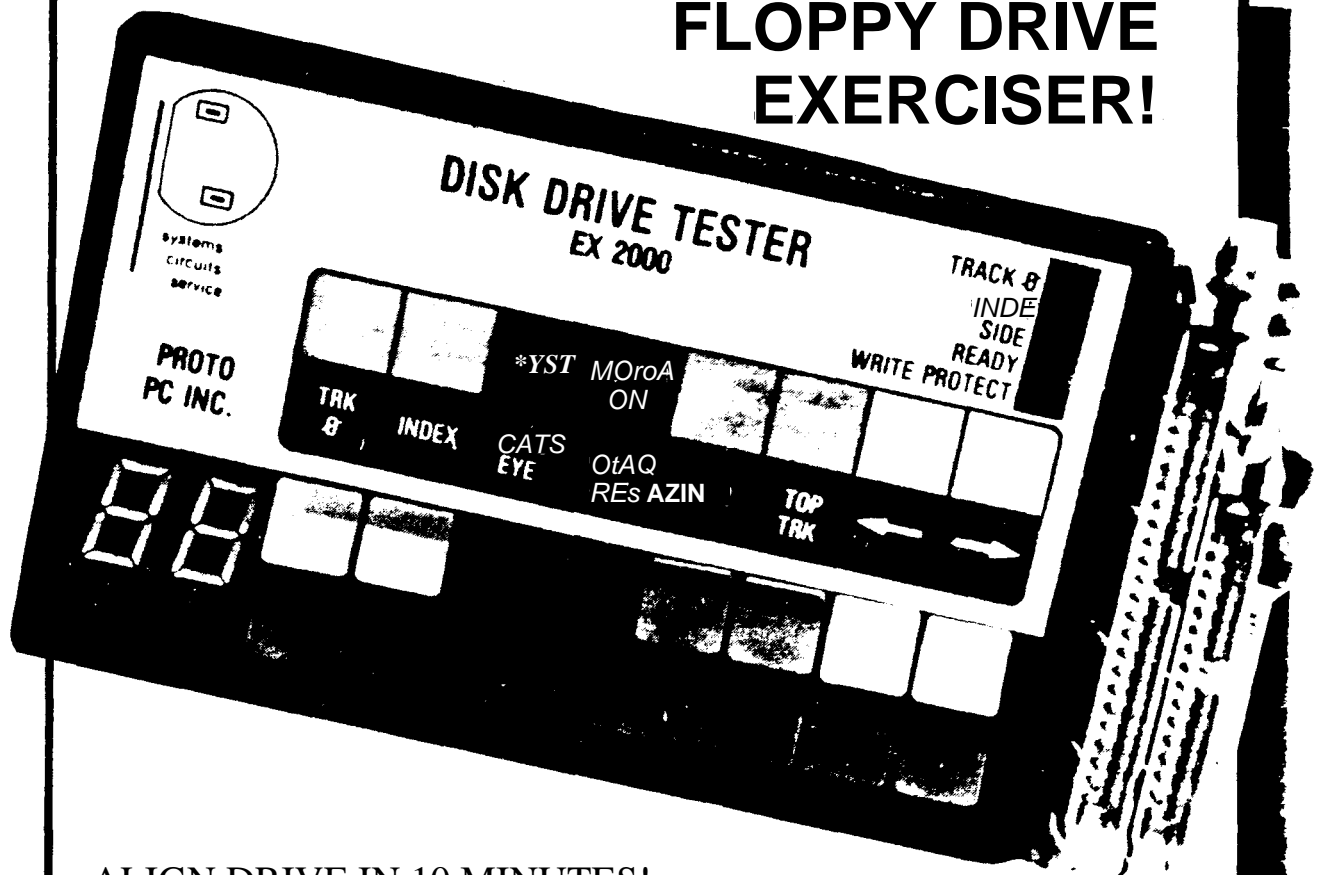
Without any more knowledge of the subject I can only advise care in re-assigning values to the Apple IOBYTE. It appears that the Microsoft documentation does go into some detail as to just what alterations you can and cannot make including the addresses for making alterations to the drivers. Have fun.

For some reason I have the feeling that the TRS-80 versions of CP/M are even stranger than the Apple's. Nufi said about that.

Listing my own setup would serve no purpose. I have, as do many hackers, a weird mixture of mismatched equipment (basically vintage S-100) that suits me just fine, and will until I decide something has to be changed or added. I will say that I have two console options, two printer options, a modem, and one each parallel and printer ports that are as yet not assigned to any practical purpose. These will no doubt be my "reader and punch" someday.

I feel that being able to change these assignments from the keyboard is a worthwhile feature. Regardless of what is available to you, it can all be put to work by the proper juggling of the IOBYTE—and in the process you get a little closer to the heart and soul of CP/M.

# 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**

# WRITE YOUR OWN THREADED LANGUAGE

## Part Three: Secondary Words

by Douglas Davidson

The last article presented the main features of input and output, as well as something of dictionary management. The time has come to explore the structures of secondary words, and to present the routines needed to maintain these structures. As has been mentioned before, a secondary word will consist mainly of a sequence of calls to other words, and thus will be mostly pure machine language; however, there are a number of features for which special provision must be made. Many of these features will actually appear in the code of a secondary word in a somewhat different form from that which the user will see. Most commonly, they will be represented in code by a call to a supporting word, generally given a name with parentheses —e.g., (IF1 For now only the embodiment in code needs to be dealt with, so that is what will be presented here. How this code comes to be constructed will be covered in the next article. The first area to cover now is that of control structures.

In order for this language to be useful, there must be provision for secondaries to employ a full variety of control structures. Some structuring is already provided by the device of threading, as distinct functions can be made into distinct words, but this in itself is not enough. Both branches and loops are needed within a secondary, and it is useful to have both indexed and non-indexed loops. Here all of these will make use of the microprocessor's branch instructions, but these need a little help to serve the language's purposes. Conditionals will be handled by constructions of the following form:

(condition\* IF (true part) ELSE (false part) THEN.

The ELSE and false part are optional; true to the language's RPN base, the instructions follow their operands. An ELSE will be represented in the actual code by an unconditional branch instruction to the point represented by the THEN; the THEN itself will leave no trace in the code (see Figure 1). The IF is the tricky part, for it must make a decision based on a flag in TOS, but the microprocessor's branch instructions can judge only microprocessor status flags. Therefore a routine is needed to change a stack flag into a microprocessor flag, in this case the 6502 carry flag. The IF will be represented by a call to (IF) followed by a BCS (branch on carry set) instruction branching to beyond the ELSE or (if there is no ELSE) to the THEN.

(IF) This word takes a flag off the stack and sets or resets the carry flag accordingly. It calls 0, then loads the lower byte of TOS, drops TOS, and shifts the low bit of the byte into the carry flag; note that a false flag will set the carry and a true flag will reset it.

Non-indexed loops will be handled very much like

conditionals (see Figure 1); they will take one of these two forms;

BEGIN ... (condition\* UNTIL ... END  
BEGIN ... (condition\* WHILE ... END.

BEGIN, like THEN, serves only to mark a location in memory: END is represented in code by an unconditional branch to the point marked by BEGIN. The UNTIL and WHILE are much like IF, and indeed they will use (IF\*, followed by a conditional branch branching to just beyond the END: for UNTIL a BCC (branch on carry clear', for WHILE a BCS (branch on carry set).

Indexed loops require a somewhat different approach, as they involve more than a simple testing of a condition (see Figure 1). They also will take two different forms:

(parameters) DO ... LOOP

(parameters) DO ... (increment\* + LOOP

(Parameters\* refers to the initial and final values of the index, and (increment) to the amount by which the index should be incremented each time. Note that as + LOOP is presented here, the increment may well vary, for it is taken from the stack at each pass through the loop. The indexed loops will make use of the R-stack: the value of the index will be placed on top of the R-stack, and the index's final value will be second. This allows for the nesting of loops quite simply; note also that the word I brings the value of the most recent loop's index to the main stack. DO will be represented in code by calls to routines transferring the parameters from the main stack to the R-stack; we already have the >R routine for this. LOOP and + LOOP will be represented by calls to their respective supporting routines, followed by conditional branches back to the start of the loop, followed by a call to a routine dropping two values from the R-stack (see below).

(**LOOPI**) This word handles an indexed loop with constant increment. It increments the top value on the R-stack, then compares the top value with the second value. If the top value is greater, the carry flag is set; otherwise the carry flag is reset.

(**+LOOP**) This word handles an indexed loop with variable increment. It takes the TOS and adds it to the top value on the R-stack. If the sign of the TOS is positive, the carry flag is returned as with (LOOP); if the sign is negative, the carry flag is set if the top value is less than the second value.

Secondary words may have occasion to print out character strings; these will be represented in code by a call to a printing routine, followed by the string to be printed, in the familiar length-first format.

(.‘) This word prints a string of characters. It first pulls

the location from which it was called off of the microprocessor stack; since the 6502 JSR instruction stores the location of the byte before the location to be returned to, the index register pointing from that address will always be one greater than might be expected. The routine gets the one-byte length, adds one to it to compensate for the last-mentioned difficulty, then prints characters in order, as many as the length specifies; the length plus one is added to the initial location and this value is pushed back on the stack.

There will also be a need for secondaries to have immediate numerical values available; these will appear in code as a call to the routine that loads them, followed by the two-byte value.

**(LIT)** This word gets a numerical value imbedded in the secondary code. It pulls the calling location off the microprocessor stack, loads two bytes from just beyond it, pushes them into TOS, increments the location by two, and pushes it back onto the microprocessor stack.

Some secondaries will also need to recognize error conditions, and to abort back to the main executive when such conditions occur. **ABORT"** takes a condition flag from the stack. If it is true, an error is recognized and the last recognized command is printed, followed by the string stored with the **ABORT"**. The instruction is skipped entirely if the flag is false. **ABORT"** is embodied in code by a call to (**ABORT"**) followed by the error message in the same format as that of (").

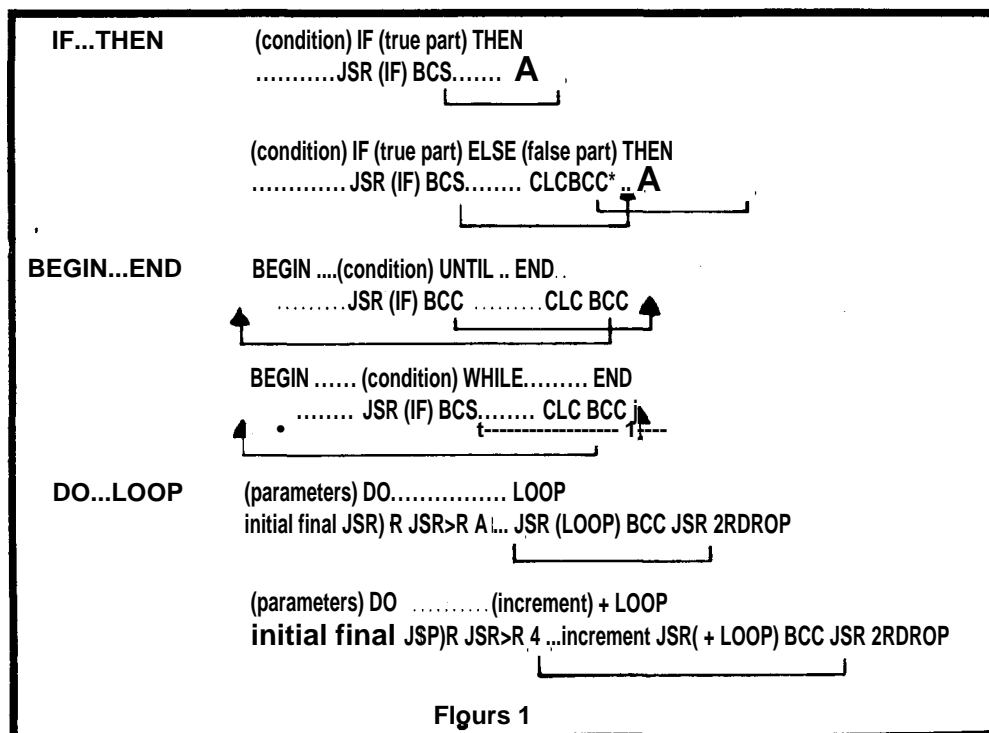
**(ABORT"**) This word conditionally executes an error abort. It first pulls the calling location off the microprocessor stack, then takes a flag from TOS; if the flag is true, the string of characters preceded by its length

starting at H is printed. Then the length byte is loaded from the calling address, as **if"**) and that many characters are printed. **7FORGET** is then called to destroy any word that is in the process of being created; finally a jump is made to the main executive (which will be covered in the next article; it is useful, for debugging purposes, to make this particular jump initially a jump out of the language, perhaps to a monitor). If the flag is false, then the length byte plus one at the calling address plus one is added to that address, and the address is pushed back onto the microprocessor stack.

Variables will require special handling. With a stack-oriented system, there is less need for variables, but they still can be very useful. Variables in this language will be semi-permanent; that is, each variable will actually be a word with its own name and place in the dictionary. Variables will actually be executable words; each will consist of a header, a call to a special routine, and some storage space. What the special routine will do is place the address of the storage space on the stack; then **@** and **I** can be used to call up or store the value of the variable. Akin to variables are constants; a constant will have the same form as a variable, but it will call a different routine. The constant routine will bring the value of the constant, not its address, to the stack; thus constants will be unalterable. Facilities will also be provided for the high-level definition of routines such as the ones implementing variables and constants, for use in special-purpose applications; the use of these requires concepts not yet covered, and will be discussed more fully in the next installment, but the words (**DOES >**) and **DODOES** are needed to implement them.

**(DOES >)** This word is the embodiment in code of **DOES >**; a call to it marks the end of the compile-time portion of a defining word. Essentially it replaces the call that **CREATE** put into the word being defined with a call to the location just after the call to (**DOES >**). The routine pulls the calling location off of the microprocessor stack and places it at **CURRENT+7** and **CURRENT + 8**.

**DODOES** This word is the beginning of the run-time portion of a defining word; the call to it immediately follows that to (**DOES >**). The routine pulls off the microprocessor stack and preserves the location (in the defining word) from which it was called. It then pulls off the microprocessor stack the location (in the defined word) from which the defining word was called; this address it increments and pushes onto the main stack. The first return address is then returned to the stack.



Figurs 1

**(CONSTANT)** This word is called by every constant to load the value of that constant. It pulls off the microprocessor stack the location in the constant from which it was called, and then pushes the value at that address plus one onto the main stack. **(VARIABLE)** This word is called by every variable to load the address of the storage area of that variable. It pulls off the microprocessor stack the location in the variable from which it was called; this address is incremented and then pushed onto the main stack.

Certain words may need to call other, unspecified words, or perhaps machine-language subroutines.

**EXECUTE** This word calls another word or an address; it expects the PF A of another word to be in TOS. It takes the TOS, puts it in a particular location, then executes an indirect jump to that location; it thus jumps to the address given in TOS.

A certain class of words, called defining words, will be needed; these words are distinguished by the fact that they create other words. They all have **CREATE** in common, for **CREATE** creates a header, using the next string of characters in the input line as the name of the word.

**CREATE** This word creates a dictionary header. It calls **WORD**, giving a space as the separation character; this puts the name for the header, in the proper form, at the proper place—at H. It stores the value of **CURRENT** for the link at H + 4 and H + 5, then places a call to **(VARIABLE)** at H + 6, H + 7, and H + 8. H is then moved to **CURRENT**, and H is incremented by 9.

It will be remembered that every input line was required to end with several termination characters, in this implementation, return characters. This is chiefly so that the termination character may be recognized as the name of a word, the word referred to for convenience as **NULL**, which is used to exit from several endless loops that interpret input lines. **NULL** is not the name of the word; the name of the word is simply the termination character.

**NULL** This word escapes from words with endless loops. It simply pulls one return address off the microprocessor stack and discards it.

**2RDROP** This word, used after the completion of each indexed loop, discards two values from the R-stack. It simply increments R by four.

Various storage locations have been presented as necessary to the operation of the primary words. It is also desirable that secondaries should have access to these words; therefore they are made into variables. They are not, however, ordinary variables, for their actual locations will most likely be in some privileged place; in this implementation they are stored in page zero. What will be placed in the dictionary will actually be constants, with values equal to the addresses of these storage locations. S, R, H, > IN, BASE, CURRENT, and SO should be treated this way.

Finally, all the information has been presented that is necessary to write an initialization routine. The above-mentioned storage locations —S, R, H, > IN, BASE, CURRENT, and SO—are those that need to be initialized. The initial value of >IN should be zero, that of BASE probably ten or sixteen, and that of S should be two less than whatever SO is. The values of H and CURRENT will be dictated by the state of the dictionary, and those of SO and R by the memory map.

**STARTUP** This headerless routine initializes all of the above-mentioned storage locations with values stored just above it; it then jumps to the main executive.

This completes the list of necessary primary routines. The words should be tested in isolation as far as is possible, to make sure that each works properly before they are combined. The next article will present the kernel of secondary words and wrap everything up. For those not exactly following the machine code presented, the following optional primary words may prove useful; they are not necessary for anything presented here, but coding them may be a good exercise.

**0 A** constant with name 0 and value zero.

**1 A** constant with name 1 and value one.

— **1 A** constant with name - 1 and value \$FFFF = - 1.

**1+** This word increments TOS by one.

**2+** This word increments TOS by two.

**1—** This word decrements TOS by one.

**2—** This word decrements TOS by two.

**2\*** This word multiplies TOS, taken as a signed integer quantity, by two.

**2/** This word divides TOS, taken as a signed integer quantity, by two.

**RDROP** This word drops one stack element from the R stack; it increments R by two.

**KEY** This word waits for a key to be pressed, then returns the (one-byte) ASCII value of the key pressed in TOS.

**?DUP** This word duplicates TOS only if TOS is non zero.

**I'** This word copies the second value on the R-stack and pushes it onto the main stack; it gets the limit of the current loop.

**J** This word copies the third value on the R-stack and pushes it onto the main stack; it gets the index of the loop exterior to the current loop.

**LEAVE** This word permits a premature exit from an indexed loop; it sets the second value on the R-stack equal to the first value.

**PAGE** This word clears the screen; it will be implementation-dependent.

**MAX** This word returns the maximum of TOS and NOS, considered as signed integer quantities.

**MIN** This word returns the minimum of TOS and NOS, considered as signed integer quantities.

**MONITOR** This word exits the language and returns control to a monitor routine; it will be implementation-dependent.

**MOVE** This word moves a string of bytes. It takes the length from TOS, the destination address from NOS, and the

source address from the third entry on the main stack. Note that it can make a difference whether the bytes are moved from the beginning of the string first or from the end first; it may be wise to have two words, MOVE and <MOVE, of which the first starts from the beginning and the second from the end.

```

*
** <LF) **
*
108D: 04 A8 C9 C6 F3 0F
1093: 20 39 09 J SR 0=
1096: A0 00 LDY ase0
1098: B1 00 LDA (S),Y | get zero or one
109A: 20 09 08 JSR DROP
109D: 4A LSR | shift into carry
109E: 60 RTS ; falsecarry set

*
** (LOOP) **
*
109F: 06 A8 CC CF BD 10
10A5: A0 00 LDY s*1
10A7: 81 02 LDA <R),Y ; increment
10A9: 18 CLC ; loop index
10AA: 69 01 ADC 4401
10AC: 91 02 STA (R),Y
10AE: 85 18 STA SCRL ; and store it
10B0: C8 INY ; in zero page
10B1: 81 02 LDA (R),Y
10B3: 69 00 ADC eseo
10B5: 91 02 STA (R),Y
10B7: 85 19 STA SCRH
10B9: C8 COMPARE INY ; is index
10BA: 38 SEC ; greater than
10BB: 81 02 LDA (R),Y ; limit?
10BD: ES 18 SBC SCRL
10BF: C8 INY
10C0: 81 02 LDA (R),Y
10C2: E5 19 SBC SCRH
10C4: 50 02 BVC OK
10C6: 49 80 EDR 0480 ; shift answer
10C8: 0A OK ASL ; into carry,
10C9: 60 RTS ; set=greater

*
** (+LOOP) **
*
10CA: 07 A8 AB CC 9F 10
10D0: A0 00 LDY 0400 ; add TOS
10D2: 18 CLC $ to loop index
10D3: 81 00 LDA (S),Y
10D5: 71 02 ADC (R),Y
10D7: 91 02 STA (R),Y
10D9: 85 18 STA SCRL
10DB: C8 INY
10DC: 81 00 LDA (S),Y
10DE: 71 02 ADC (R),Y
10E0: 91 02 STA (R),Y
10E2: 85 19 STA SCRH
10E4: 81 00 LDA (S),Y ; is increment
10E6: 30 06 BMI NEG ; negative?
10E8: 20 89 10 JSR COMPARE ? no, do as in
10EB: 4C 09 08 JMP DROP 1 (LOOP)
10EE: C8 NEG INY ( yes, make
10EF: 38 SEC ; reverse comparison
10F0: A5 18 LDA SCRL
10F2: F1 02 SBC (R),Y
10FC: ce INY
10F5: A5 19 LDA SCRH
10F7: F1 02 SBC (R),Y
10F9: 50-02 BVC OK
10FB: 49 80 BOR 0080
10FD: 0A ASL } and shift into carry
10FE: 4C 09 08 JMP DROP ( setsindex is less

*
** ee <.(. **
*
1101: 04 A8 AE A2 CA 10
1107: 68 PLA | pull calling location
1108: 85 18 STA SCRL
110A: 68 PLA
110B: 85 19 STA SCRH
110D: A0 01 LDY 0001 ; get length
110F: 98 TYA $ plus one
1111: 71 18 ADC (SCR),Y
1113: 85 16 STA ACC.C2L
1115: C8 LOOP INY
1116: 81 18 LDA (SCR),Y i get each character
1118: 20 ED FD JSR COUT ; and print it
1118: C4 16 CPY ACC.C2L 1 until done
111D: D0 F6 BNE LOOP
111F: 18 CLC
1120: 98 TYA ( adjust calling
1121: 65 18 ADC SCRL ; location
1123: AA TAX
1124: A5 19 LDA SCRH
1126: 69 00 ADC 0000
    
```

```

1 128: 48 PHA $ and push it back
1 129: 8A TXA
1 12A: 48 PHA
1 12B: 60 RTS

*
** (LIT) * #
*
1 120: 05 A8 CC C9 01 11
1132: 68 PLA ; pull calling location
1 133: 85 16 STA SCRL
1135: 68 PLA
1136: 85 19 STA SCRH
1138: E6 00 INC SL 1 decrement S
113A: C6 00 DEC SL
1 13C: D0 02 BNE OKI
113E: C6 01 DEC SH
1140: C6 00 OKI DEC SL
1142: D0 02 BNE DK 2
1144: C6 01 DEC 3H
1146: C6 00 OK2 DEC SL
1148: A0 02 LDY *402 ; get two bytes
1144: B1 18 LDA (SCR),Y ; and store in TOS
114C: 88 DEY
114D: 91 00 STA (S) . Y
114F: B1 18 LDA < SCR >,Y
1 151: 88 DEY
1152: 91 00 STA (S) , Y
1154: A5 18 LDA SCRL ( adjust calling
1156: 18 CLC ( location
1157: 69 02 ADC *402
1159: AA TAX
115A: A5 19 LDA SCRH
1 150: 69 00 ADC *400
115E: 48 PHA $ and push it back
1 15F: 8A TXA
1160: 48 PHA
1161: 60 RTS

*
** (ABORT ") **
*
1162: 08 A8 C1 C2 2C 1 1
1168: 68 PLA ) pull calling location
1169: 85 18 STA SCRL
116B: 68 PLA
116C: 85 19 STA SCRH
116E: 20 39 09 JSR 0* | get flag
1171: A0 00 LDY *400
1173: B1 00 LDA (S) , Y
1175: 20 09 08 JSR DROP
1178: AA TAX 1 was it true?
1179: F0 0F BED ABORT
1178: CB INY 1 no, continue
117C: B1 18 LDA (SCR),Y ; simply adjust
117E: 38 SEC ; calling location
117F: 65 18 ADC SCRL
1 181: AA TAX
1182: A5 19 LDA SCRH
1184: 69 00 ADC *400
1186: 48 PHA $ and push it back
1 187: 8A TXA
1188: 48 PHA
1189: 60 RTS
118A: B1 04 ABORT LDA (H) , Y ; yes, abort
118C: 85 16 STA ACC.C2L ? print string at H
118E: C8 LOOP1 INY
118F: B1 04 LDA (H),Y ; get each byte
1191: 20 ED FD JSR COUT | and print it
1194: C4 16 CPY ACC.C2L ; until done
1196: D0 F6 BNE LOOP1
1198: A0 01 LDY *401 | print string
119A: 98 TYA I at calling location
1198: 18 CLC I get length plus one
119C: 71 18 ADC (SCR),Y
119E: 85 16 STA ACC.C2L
11A0: C8 LOOP2 INY
11A1: B1 18 LDA (SCR),Y I get each byte
11A3: 20 ED FD JSR COUT ; and print it
11A6: C4 16 CPY ACC.C2L S until done
11A8: D0 F6 BNE LOOP2
11AA: 20 21 0F JSR FORGET ; kill any stray word
11AD: 4C 73 14 JMP QUIT ; and restart

*
** (DOES> ) **
*
11B0: 07 A6 C4 CF 62 11
1186: A0 07 LDY *407 | replace the call
1188: 68 PLA ; that CREATE made
1189: 18 CLC $ with whatever called
1 1BA: 69 01 ADC *401 | this routine
11BC: 91 0A STA (CURRENT),Y
11BE: CB INY
11BF: 68 PLA
1 1C0: 69 00 ADC *400 ; and skip rest of
11C2: 91 0A STA (CURRENT),Y
11C4: 60 RTS ; whatever called this

*
** DODOES *
*
11C5: 06 C4 CF C4 B0 11
11CB: E6 00 INC SL I decrement S
    
```

```

1 ICD: C6 00      DEC SL
11CF: D0 02      BNE OKI
11D1: C6 01      DEC SH
11D3: C6 00      OKI  DEC SL
1 1D5: De 02      BNE OK 2
1107: C6 01      DEC SH
11D9: C6 00      OK2  DEC SL
11DB: 68          PLA          1 pull calling location
11DC: 85 18      STA SCRL     i and save 11
11DE: 68          PLA
11DF: AA         TAX
11E0: A0 00      LDY *600
11E2: 68          PLA          I pull whatever called
11E3: 18          CLC          1 that and increment
11E4: 69 01      ADC *601     ; it, then store it
11E6: 91 00      STA (S),Y   ; into TOS
11E8: C8          I NY
11E9: 68          PLA
11EA: 69 00      ADC *600
11EB: 91 00      STA (S),Y
11EC: 8A          TXA          { and restore
11ED: 48          PHA          1 calling location
11EF: A5 18      LDA SCRL
11F2: 48          PHA
11F3: 60          RTS
    
```

```

(CONSTANT) * <
11F4: 0A ee C3 CF C5 11
UFA: E6 00      INC SL          ; decrement S
11FC: C6 00      DEC SL
11FE: D0 02      BNE OKI
1200: C6 01      DEC SH
1202: C6 00      OKI  DEC SL
1204: D0 02      BNE OK 2
1206: C6 01      DEC SH
1208: C6 00      OK 2  DEC SL
120A: 68          PLA          ; pull calling address
1208: 85 18      STA SCRL
120D: 68          PLA
120E: 85 19      STA SCRH
1210: A0 02      LDY *602     ; get value stored
there
1212: B1 18      LDA (SCR),Y
1214: 88          DEY          1 and put it in TOS
1215: 91 00      STA (S),Y
1217: 81 18      LDA (SCR),Y
1219: 88          DEY
121A: 91 00      STA (S),Y
121C: 60          RTS          ; don't return to it
    
```

```

** (VARIABLE) 4 4
121D: 0A A8 06 C1 F4 11
1223: E6 00      INC SL          ; decrement S
1225: C6 00      DEC SL
1227: D0 02      BNE OKI
1229: C6 01      DEC SH
1228: C6 00      OKI  DEC SL
122D: D0 02      BNE OK 2
122F: C6 01      DEC SH
1231: C6 ee      OK 2  DEC SL
1233: A0 00      LDY *600
1235: 68          PLA          ; pull calling address
1236: 18          CLC          ; increment it
1237: 69 01      ADC *601
1239: 91 00      STA (S),Y   $ and store it in TOS
1238: C8          I NY
123C: 68          PLA
123D: 69 00      ADC *600
123F: 91 00      STA (S),Y
1241: 60          RTS          I don't return to it
    
```

```

** EXECUTE 2 **
1242: 07 CS D6 C5 ID 12
1248: A0 00      LDY *600
124A: 81 00      LDA (S),Y   | take TOS
124C: 85 18      STA SCRL     1 and store it
124E: C8          I NY
124F: 81 00      LDA (S),Y
1251: 85 19      STA SCRH
1253: 20 09 08   J SR DROP
1256: 6C 18 00   J MP (SCR)  I then jump to it
    
```

```

CREATE
1259: 06 C3 D2 C5 42 12
125F: A0 00      LDY *600
1261: A9 A0      LDA 06A0    | call WORD
1263: E6 00      INC SL      t Ml th space as
1265: C6 00      DEC SL      1 separation character
1267: D0 02      BNE OKI     | leaving room for
1269: C6 01      DEC SH      ; WORD to put H
1268: C6 00      OKI  DEC SL | on stack
1260: De 02      BNE OK 2
126F: C6 01      DEC SH
1271: C6 00      OK2  DEC SL
1273: 20 CC 0D   JBR WORD2
1276: E6 00      INC SL      | discard H that
    
```

```

1278: D0 02      BNE OK 3      ; WORD left on stack
127A: E6 01      INC SH
127C: E6 00      OK3  INC SL
127E: D0 02      BNE OK 4
1280: E6 01      INC SH
1282: A0 04      OK4  LDY 6604    ? store CURRENT as link
1284: A5 0A      LDA CURRENTL
1286: 91 04      STA (H),Y
1288: C8          I NY
1289: A5 08      LDA CURRENTH
128B: 91 04      STA (H),Y
128D: C8          INY          ; store cali
1286: A9 20      LDA 6620
1290: 91 04      STA (H),Y
1292: C8          INY
1293: A9 23      LDA 9623    i to (VARIABLE)>61223
1295: 91 04      STA (H),Y
1297: C8          INY
1298: A9 12      LDA <612
129A: 91 04      STA (H),Y
129C: A5 04      LDA HL      ; H -> CURRENT
129E: 85 0A      STA CURRENTL
12A0: 18          CLC          1 and increment H by 9
12A1: 69 09      ADC <609
12A3: 85 04      STA HL
12A5: A5 05      LDA HH
12A7: 85 08      STA CURRENTH
12A9: 69 00      ADC es00
12AB: 85 05      STA HH
12AD: 60          RTS
    
```

```

* NULL **
12AE: 01 8D A0 A0 59 12
1284: 68          PLA;          d i s c a r d a d d r e s s
12B5: 68          PLA          1 of whatever
12B6: 60          RTS          ; cal led thi s
    
```

```

* 2RDROP **
1287: 06 82 92 C4 AE 12
12BD: 18          CLC          1 increment R by 4
12BE: A5 02      LDA RL      ; to drop used
12C0: 69 04      ADC <604    ; loop index and limit
12C2: 85 02      STA RL
12C4: 90 02      BCC OK
12C6: E6 03      INC RH
12C8: 60          Ok  RTS
    
```

```

S
1209: 01 D3 A0 A0 B7 12 20 FA 11 00 00
** R *
12D4: 01 D2 A0 A0 C9 12 20 FA 11 02 00
** H
12DF: 01 C8 A0 A0 D4 12 20 FA 11 04 00
** >IN
12EA: 03 BE C9 CE DF 12 20 FA 11 06 00
** BASE *
12F5: 04 C2 C1 D3 EA 12 20 FA 11 08 00
** CURRENT
1300: 07 C3 D5 D2 F5 12 20 FA 11 0A 00
** S0
1308: 02 D3 80 A0 00 13 20 FA 11 0C 00
    
```

```

* STARTUP **
No Header
1316: A0 0D      LDY <60D
1318: 89 24 13 LOOP LDA STARTUP+SGE, Y
1318: 99 00 00      STA SL,Y     f move image to
131E: 88          DEY          | zero-page
131F: 10 F7      BPL LOOP    | storage locations
1321: 4C 73 14   JMP QUIT    | and start
1324: FE 88 FE 91 4B 18 00 00 0A 60 18 18 00 BC
    
```

## AUTHORS WANTED!

The Computer Journal is interested in technical articles. Query with SASE or send for our Author's Guide.

PO Box 1697, Kalispell, MT 59903

# Interfacing Tips and Troubles

A Column by Neil Bungard I

## Noise Problems, Part Two

This month I will present part two of a three part series on noise problems associated with interfacing. In part one we discussed the problems created by power supply noise. We looked at symptoms associated with power supply noise, and reviewed some possible methods for minimizing it. In parts two and three we will look at noise problems associated specifically with the interface circuit. For the purpose of discussion I have divided interface noise into three categories. We will look at each category in terms of what symptoms you might expect to encounter, and the possible solutions for reducing or eliminating the noise problems. In addition, a section on noise reduction using ferrite beads has been included. Finally, we will create a troubleshooting chart which can be used as a quick reference when approaching troubled interfacing projects. This month we will discuss noise associated with data transmission lines. In part three of this series we will cover noise generated within the interface circuit, and noise induced from outside sources.

Before we begin our discussion on specific noise problems encountered in interfacing circuits, let's look at the basic elements which are required for noise interference to exist (See Figure 1). The "noise source" can be a logic transition, a local oscillator, a wire vibrating in a magnetic field, a relay or solenoid deactivating, etc. The "noise coupler" can be a common impedance, a stray capacitance, a mutual inductance, or a magnetic field. The "noise receiver" can be an op amp input, a TTL input, an IC power supply input, a voltage reference, etc.

In order to solve any given noise problem, one or more of the three basic elements must be reduced or eliminated. As we look at specific noise problems, think about how the situations correspond to the model in Figure 1. Hopefully this will be helpful in understanding the purpose behind the solutions used in the noise reduction problems.

### Noise Associated With Data Transmission Lines IR Drop

Unless an interface is connected directly to a computer via an edgcard connector, it will be connected to the computer through a cable. The cable can be a ribbon cable or simply a bundle of wires, but regardless of the cable configuration there are considerations which must be taken into account when sending logic signals over transmission lines.

Always limit the length of cables carrying TTL level

signals to one meter or less if possible. There are several reasons for this. To begin with, signal deterioration occurs as a result of driving logic signals over long pieces of wire. This is because there is a resistance associated with the wire which increases with an increase in length. As the resistance increases, the resulting voltage drop along the wire increases. This is known as IR drop. The relationship between resistance and voltage drop can be expressed using Ohms law:

$$V = I \times R$$

According to the above expression, relatively high current situations (e.g., when logic transitions occur) can create voltage drops along the transmission line, which can become large enough to cause false triggering on logic inputs. The solution to these particular noise problems is simple: just keep the cables as short as possible.

### Line Inductance

In addition to resistance increase, there is an inductance increase associated with increases in line length (approximately 0.8 micro-henry per meter). The inductance associated with a length of wire can cause noise voltage spikes in a line where current values are suddenly changing. This condition can exist whenever a logic transition occurs. Typical values that might be observed in a logic transition are changes in current of 30 to 50 milliamps with signal rise times of 10 to 30 nanoseconds. The following equation expresses the relationship between variables in an inductive noise problem:

$$dV = L \times di / dT$$

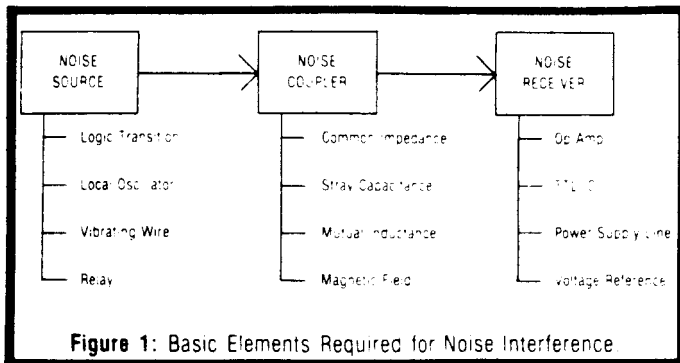
where: dV - magnitude of noise voltage spike  
L = inductance of signal carrying conductor  
di = sudden change in current  
dT - rise time associated with a logic transition

For example, If you had a gate signal with a varying current of 30 milliamps, and a rise time of 10 nanoseconds, the magnitude of a noise voltage spike which could be generated in a one meter line would be:

$$dV = 0.8 \times 10 \times 30 \times 10 / 10 \times 10$$

$$dV = 2.4 \text{ volts}$$

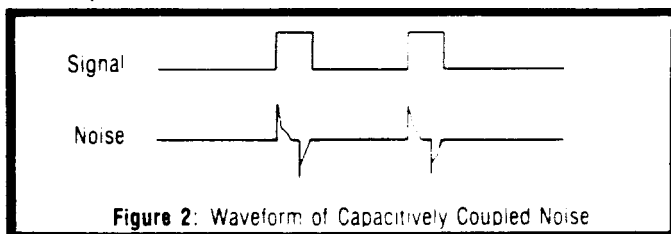
This value would be slightly decreased because of capacitive effects and other loading in the circuit, but in general this sort of noise can cause real problems if your connecting cables become too long.



### Line Capacitance

Another problem which can arise in a transmission cable is capacitive coupling. Capacitive coupling occurs when conductors containing signals with fast rise times (like a device code pulse) or signals with a high frequency content (like a local oscillator) are in close proximity to other signal lines. Stray capacitance between lines in the cable couples the fast edges of a signal from a source line to adjacent lines. Making the transmission lines as short as possible reduces stray capacitance and thus decreases the chance for coupling problems, but short lines alone may not ensure trouble free operation. Figure 2 shows the characteristic waveform associated with capacitively coupled noise. If the frequency of the noise waveform can be determined, the source of the noise can probably be located. This is possible because in a capacitively coupled circuit the noise source operates at the same frequency as the noise itself. If the noise appears to be noncyclic however, the source of the noise may be difficult to locate.

We have just talked about three possible causes of transmission line noise: IR drop, mutual inductance, and stray capacitance. But how do we know when the noise problem is related to the transmission lines, and assuming that we do know, how do we eliminate the problems?



### Finding and Solving Transmission Line Noise Problems

Symptoms associated with cable injected noise vary. In some cases data seems to disappear. The signals transmitted simply do not end up at the proper locations on the interface board. In addition, data integrity may be poor. This means that the information sent by the computer is not always the

same as the information received by the interface circuit. Other symptoms include incorrect or multiple device code generation within the interface circuit, data reception latches dropping bits, control flip flops toggling uninstructed, bus "lock-ups" resulting from false triggering of tristate devices, etc.

One indication that a problem is directly related to the transmission lines is that the problem is occurring in an IC directly connected to a line. However, this is not always the case because noise can pass through ICs (like buffers and effect other components deeper in the circuit. If this happens, there should be a logical path from the malfunctioning IC back to the noisy transmission line.

In the case of noise resulting from an IR drop in a transmission line, the solution was to keep the connection between the computer and the interface as short as possible. A cable length of one meter or less is recommended. In cases where noise results from mutual inductance or stray capacitance, the problem becomes a little stickier.

One noise reduction technique that can be accomplished with no added expense and with little effort is physical separation. The rule is to separate all lines into groups according to the type of signals they are transmitting. In other words, do not run AC power lines with digital data lines, do not run DC power lines with lines that carry current for switching relays and solenoids, etc. At the very minimum, AC power lines, high current control lines (for relays, solenoids, heater elements, etc.), and DC power lines should be run in separate cables from the computer to the interface circuit. Physical separation is an effective noise reduction technique, but interference can occur even between lines of similar signal types. One very effective method for reducing interference due to capacitance and inductive coupling is to place lines physically close to a ground plane. In the case of a transmission cable, special ribbon cable can be purchased which has a ground plane molded into the cable assembly. These cables are less flexible than the cables without the ground plane. They are also considerably more expensive, and are harder to find. However, there is a trick which will allow you to use non-ground-plane cable and still reduce coupling effects in the cable. First, buy a ribbon cable with twice the number of lines which are required by the interface circuit. Alternately space the signal lines so that you have an unused line between each signal line. Ground all of the unused lines to a central ground point on the computer end of the cable (see Figure 3). This technique appreciably reduces noise because of two separate effects. First of all, by alternating signal and ground lines the physical separation between signal lines has actually been increased, which reduces coupling effects. Secondly, by placing a grounded line close to a signal line, you are effectively coupling the noise to ground. If the noise is being coupled to ground, it will not be available to interfere with other signal lines. The combination of these two effects makes this a very efficient method of reducing cable noise — personally, it is my favorite method.

There may be times when you cannot alternate signal and ground lines, or there may be times when this method alone

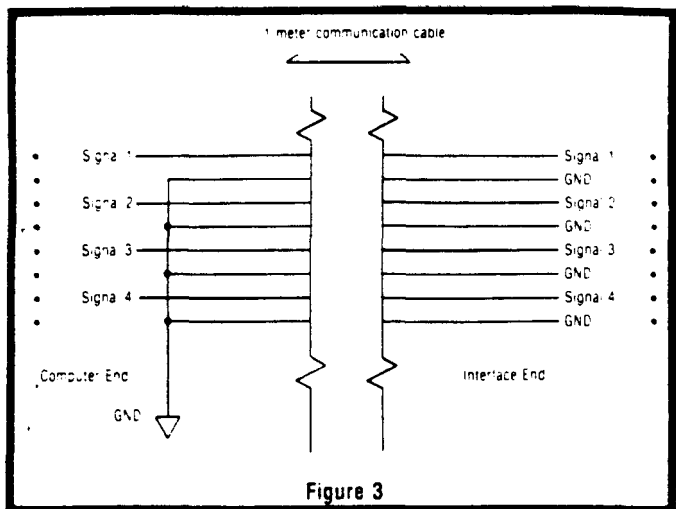


Figure 3

will not be sufficient to reduce noise below acceptable levels. In cases like these you can employ termination techniques to reduce noise. In fact, it is good practice to incorporate termination into the initial design of your interface project. Termination techniques reduce noise by changing line impedance and/or adding an element of filtering to the transmission lines. I have looked at a number of schematics of popular computers and have noticed specific similarities in the termination techniques used by many manufacturers.

To begin with, ICs that drive the signals between a computer and the interface circuit come in two flavors; regular TTL level output, and TTL open collector output. You will need to consult a technical manual for your particular computer to determine which configuration it uses. For lines driven by open collector ICs, the termination scheme in Figure 4a can be used to reduce noise. The idea is to match the driver impedance to the impedance of the connecting cable, thereby transmitting maximum signal strength with minimum distortion. In addition, terminating a signal line with a low resistance actually decreases the capacitive coupling which is the primary culprit in cable induced noise. The circuit in Figure 4b is also used on open collector driven lines. This circuit will decrease ringing caused by fast rise time signals, and will help filter high frequency and fast transient noise problems. The resistor and capacitor in Figure 4b act as a low pass filter and, in effect, short high frequency AC signals to ground. The 7414 is a Schmidt trigger and ensures that voltage fluctuations

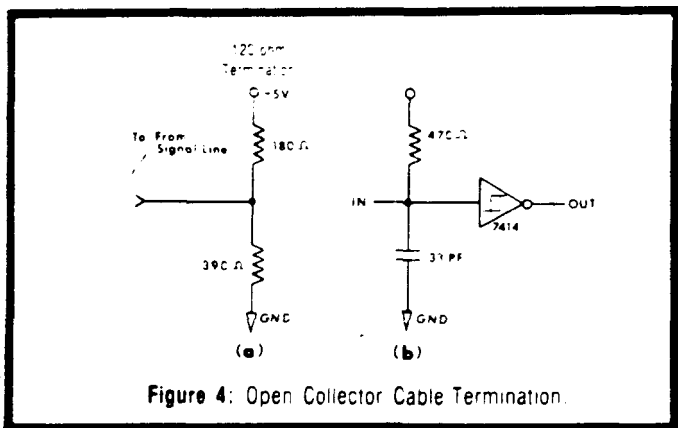


Figure 4: Open Collector Cable Termination.

below its threshold are not passed on. IBM uses this circuit in their PC products.

For regular TTL driven lines, I have found that the termination networks in Figure 5 work very well. Both circuits act as low pass filters, shorting high frequency noise and fast transients to ground. The steady state DC drive of the lines terminated by these circuits is not affected, so that signal deterioration due to loading is not a problem.

Probably the simplest, most effective, and (in my opinion) most elegant solution to cable noise problems is the use of 74L series ICs as a buffer between the interface circuit and the transmission cable. The signal lines must come into the interface circuit through ICs anyway, so using the 74L series does not add additional hardware. The 74L series ICs are more expensive, but well worth the cost. I was sold on this particular noise reduction technique when I was working on an interface project for the Commodore 64. I was having terrible noise problems in an address decoder. I tried grounding lines, grounding planes, and termination, but the interface circuit would not behave. Finally, out of desperation, I was looking through a TTL design manual and noticed the specifications on the 74L series ICs. Propagation delay time through the 74L series is terrible—just what I needed! I replaced the 74LS154 with a 74L154 as the address decoder and my problems disappeared. The low speed of the 74L series is the characteristic that makes it well suited for noise reduction applications. High frequency noise and fast transients simply do not get past the 74L series ICs. Unfortunately their speed is also their limitation. If your interface is "timing critical," the 74L series may slow the signals down enough to cause problems. Consult a TTL design manual for delay times if you suspect that your circuit is timing critical.

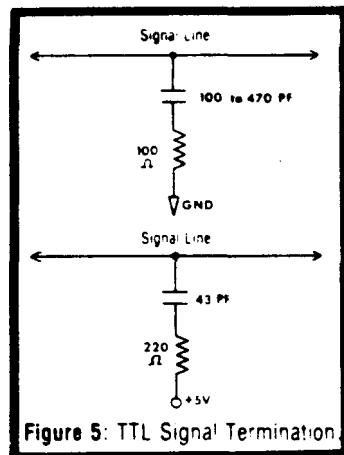


Figure 5: TTL Signal Termination.

This concludes our discussion on noise associated with data transmission lines. Next month we will look at noise problems associated with the interface circuit itself, and noise problems created by outside sources. Also, the "strange" noise reduction story that I promised this month has been added as a conclusion to next month's "Interfacing Tips and Troubles." If you have any personal questions concerning the information that I have covered up to this point, drop me a line, care of *The Computer Journal*. I will be glad to address any questions that you may have.

Dear Neil Bungard;

I would like to adapt the circuit from your article "Build a VIC-20 EPROM Programmer" (TCJ Vol II No. 4) for a Timex TS2068. It has 48K RAM from 4000h to FFFFh and the BASIC is somewhat different from the VIC-20. Where should I start, and how should I go about it? Is the circuit in

continued on page 37

# BUILD A 68008 CPU FOR THE S-100 BUS

by Lance Rose, Technical Editor

One of the nicer features of the S-100 bus, or any bus for that matter, is the ease with which the system can be gradually upgraded. Any one of the new 16-bit microprocessors which are coming into focus today can be adapted to the S-100 bus. The Motorola 68000 series has been largely ignored in this regard, even though it is probably the most powerful microprocessor chip available on the market today. In this article, I will describe how to build a CPU board for the S-100 bus using the 68008.

The 68008 is a version of the 68000 with an 8-bit external data bus; otherwise, it has the same instruction set as the 16-bit data bus 68000 chip. The relationship of the 68008 to the 68000 is the same as that of the 8088 to the 8086. The advantage in using the 8-bit data bus version is that it can be easily integrated into a system that has 8-bit wide memory boards, which includes my own system.

Before proceeding with the details of the design and construction, let's look at why the 68008 might be a good choice of processor as opposed to some others:

**1. Large linear addressing space.** The 68008 has 20 address lines (as opposed to 24 for the 68000\* thus allowing a megabyte of memory to be directly addressed. This should be enough for almost anybody's use in the near to moderate future. In addition to this, the addressing is treated as one long string of addresses with no segmentation or segment registers to manage. This makes it easier to write programs that require more than 64K of either program or data without the need to be constantly worrying about what segment you're currently in.

**2. Large number of registers.** The 68008 has 8 address registers and 8 data registers, all of them 32 bits wide. This permits many of the results of intermediate calculations to be held in registers without the need for storing them in memory or on the stack, a slower process than register to register operations.

**3. Complete instruction set.** In addition to the usual complement of ADD, AND, MOVE and so forth, the 68008 has a variety of bit manipulation instructions, hardware multiply and divide (both signed and unsigned), multiple register MOVES and so forth. Most instructions give you a choice in operand size of either byte (8 bits), word (16 bits), or long word (32 bits).

**4. Variety of addressing modes.** Operands can be addressed in many ways. Besides the usual direct addressing method, there are also address register indirect, indirect with either postincrement or predecrement, indexed with offset, base plus index plus offset, PC relative, and PC relative with index. Enough to make it easy to access operands once you master the principles of all these

various modes.

**5. Symmetrical register set.** As opposed to some processors, notably the 8086 family, there are no dedicated registers save the program counter and status flags. For example, any address register can be used for indirect addressing and any data register used for multiply and divide. This saves the programmer from having to do a lot of data shuffling to get at the single register that has to be used for, say, base addressing.

**6. Speed.** The 68008 runs at 8 MHz in the slowest version. 10 MHz and 12.5 MHz versions are available, though more costly and a bit harder to get. This increases throughput and, especially with the generous number of on-chip registers, makes the external 8-bit data bus less of a handicap than it might be.

With all these pluses, it might seem surprising that the 68000 family is not used more. Hopefully this article will help remedy that situation.

## Circuit Description

Figure 1 (pp. 16-17) shows a schematic diagram of the 68008 circuit, and Figure 2 gives a list of parts which can also be used to cross-reference the IC numbers with the part types in the discussion below. I will go through the circuit one section at a time, explaining how it all works.

**Clock circuitry.** U1a and U1b form a conventional crystal oscillator with X1, C1, R1 and R2. The output is buffered by U1c to prevent loading from affecting the oscillation of the circuit. One thing to note here is that you must use a 7404. A 74LS04 is okay for slower clocks but it's hard to get it to oscillate at 8 MHz or above due to the saturation characteristics, input loading, output drive, etc. of the Schottky version as opposed to the ordinary TTL 7404.

The clock signal is fed directly to the 68008 and to several other ICs. Its phase is inverted before being placed on the S-100 bus line, the reason being that most important state transitions of the 68008 occur on a falling clock signal, while most of the S-100 bus state transitions occur on a rising clock edge. U25b inverts the on-board signal to help the compatibility.

U9a and U9b form a divide-by-4 counter to reduce the clock to 2 MHz which is then buffered through U25b onto the S-100 CLOCK line for use as a reference frequency for UARTs, timers and so forth.

**Control bus.** U11a and U11b act as a delay circuit to cause pSYNC to be generated at the proper CPU state for the 68008. The output is fed through noninverting buffer U18a to the S-100 pSYNC line. U2a combines this signal with the

proper phase of the clock to generate pSTVAL\*. U8a is held in the reset state until a combination of read and data strobe from the 68008 releases it. On the falling edge of pSYNC, it then clocks high, beginning the pDBIN signal. The reason for delaying the start of pDBIN is to conform more closely to the IEEE-696 standard.

For pWR\*, a straightforward combination of the write and data strobe from the 68008 through U7a is used, since the write signal is already sufficiently delayed to meet the specification. U7b combines pWR\* and sOUT which is buffered through part of U26b to generate the MWRITE signal. This is switch selectable by SW1-4 in case you have a system with a front panel which generates MWRITE already.

The S-100 HOLD signal is fed directly to the 68008 bus request pin. The bus grant signal is inverted by U1d and driven onto the S-100 PHLDA line by U12a.

All control output signals are disabled when CDSB\* goes low and, after inversion through L'5a, turns off all the bus drivers U18a and U12a.

**Wait state circuitry.** UIOa and UIOb are set up as a generator of one or two wait states. Both are cleared by pSYNC; UIOa is clocked high on the next clock pulse after pSYNC goes low, and UIOb one clock cycle after that. The output of this flip-flop goes to three pairs of DIP switches, SW5. U15 is a multiplexer whose output is a function of whether the bus cycle accesses RAM, the on-board ROM, or an I/O device (addresses \$FFF00-\$FFFFF). Depending on which is selected, the output of the multiplexer corresponds to the number of wait states selected for each of the above conditions. Note here that for the pairs of switches, only one of each pair should be on at a time, thus selecting either one or two wait states but not both at once. If no wait states are desired, both switches of the pair should be left off.

The output of the multiplexer is connected to one input of U4a. Other inputs to U4a come from both the RDY and XRDY bus lines, thus allowing external boards to generate any number of wait states. The output of U4a is connected to DTACK\* on the 68008 so that either external devices or the on-board wait state generator can force the processor to wait for slow memory or the like.

**Interrupt circuitry.** Since the 68008 handles interrupts on a priority basis, some means must be found to prioritize both INT\* and NMI\*. Since NMI\* is supposed to be non-maskable, its assertion causes both interrupt inputs of the 68008 to be pulled low. This is accomplished directly for IPL2/0\* and indirectly through U2b and U5b for IPL1\*. This generates a level 7 interrupt which is non-maskable. For INT\*, only IPL1\* is pulled low, generating a level 2 interrupt. In order to enable ordinary interrupts, the interrupt mask should be set at less than 2 (say 1). To disable interrupts, the mask value should be 2 or greater (say 6).

**Reset and halt circuitry.** The fact that the halt and reset pins of the 68008 are bidirectional requires some special

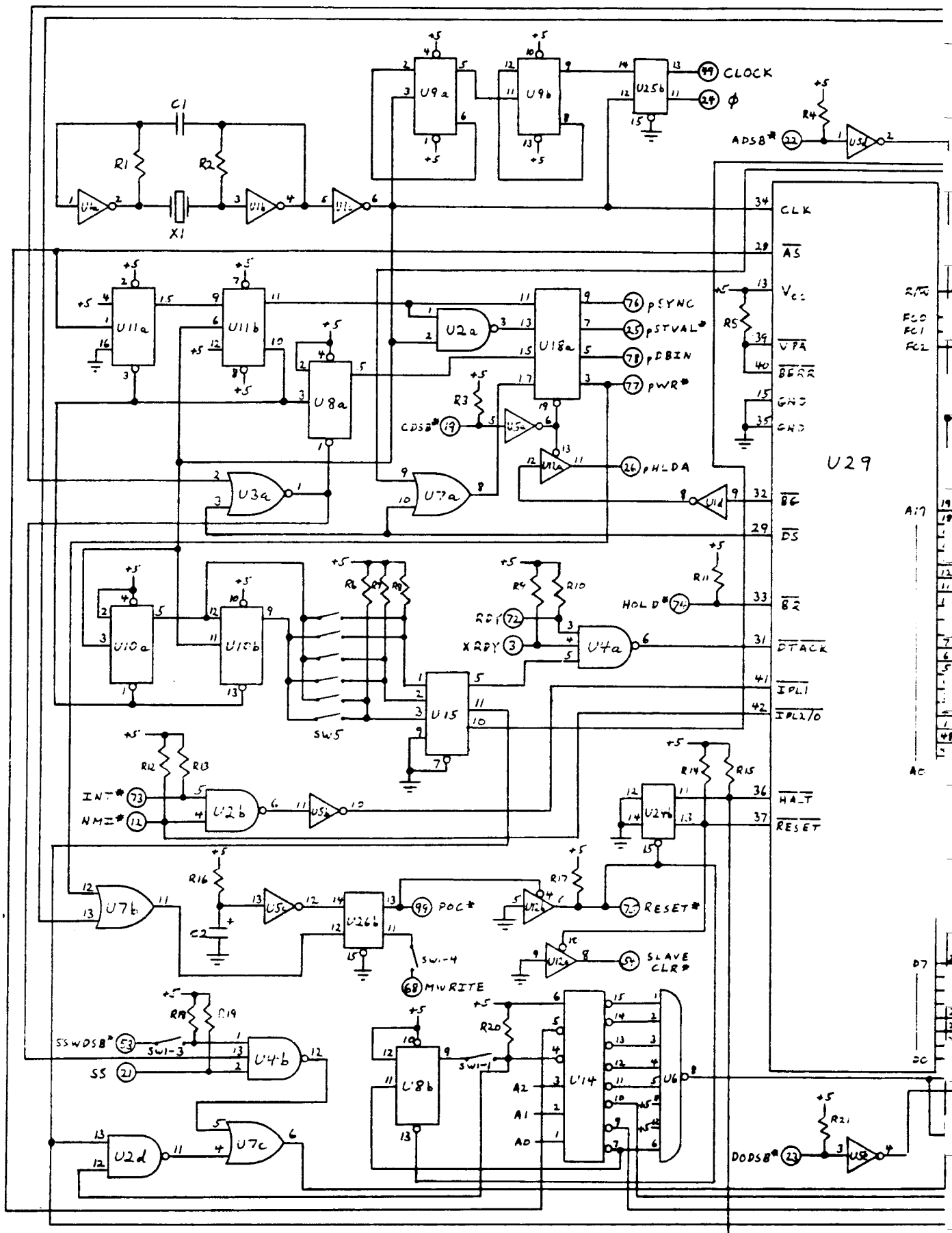
considerations in the reset circuit so that the 68008 will not be trying to drive the pins low when the external circuitry is trying to drive them high. R16 and C2 generate a power-on time constant of approximately 400 msec. This allows plenty of time for the processor to initialize itself before being allowed to run. U5c is a Schmidt trigger input hex inverter so that the 68008 run signal will be a fast-rising edge. Its output is driven through U26b onto the bus as

POC\*. The same signal activates tri-state buffer U12b whose output is the actual RESET\* bus signal. This is also applied to the control pin of U24b causing both the RESET\* and HALT\* pins to be driven low, a necessary condition for a complete 68008 reset. Note that a RESET instruction of the 68008 will also generate a low on the 68008 RESET\* line. This has the effect of driving the output of U12c low, which is connected to the S-100 SLAVE CLR\* line. This line is thus driven low on both external and internal resets, while the RESET\* line is driven low on external resets only.

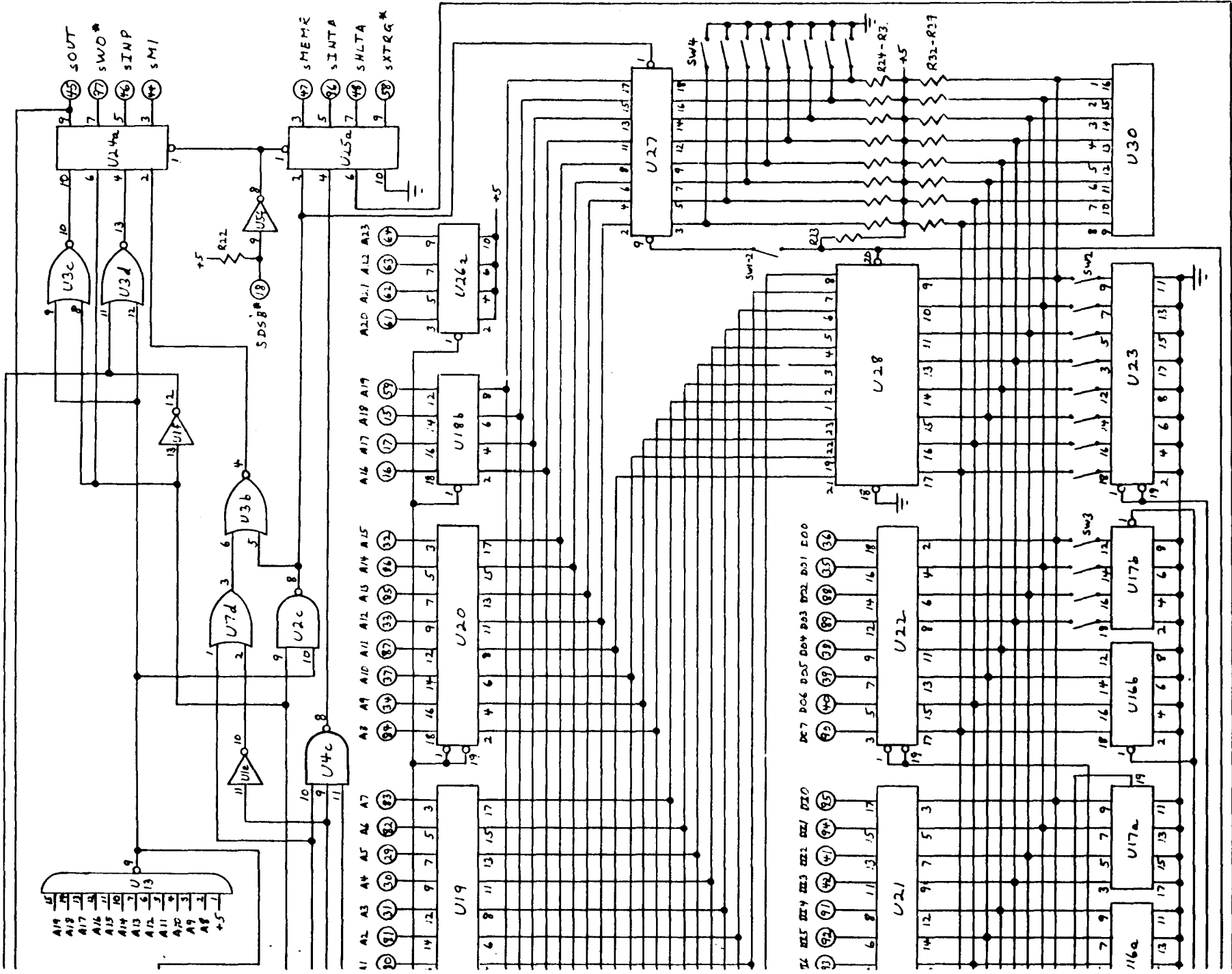
**Data bus.** On an output cycle, the data lines from the 68008 are buffered onto the S-100 data out lines by U22. For external device TMA accesses, U22 can be disabled by asserting DODSB\* through U5e and turning off the tri-states. For input cycles, U21 drives the 68008 data lines with data from the S-100 data in lines. Its disabling is somewhat more complicated than the data out buffers. Since I use this board in an IMSAI with a front panel, it was necessary to stay as compatible as possible with its operation. The IEEE-696 standard assigns bus line 53 to ground. However, the front panel uses the same line to disable the data bus in buffers in the case of data coming from the front panel instead of the bus proper. To allow for both cases, SW1-3 selects SSWDSB\* as an input for front panel machines, or allows it to be ignored for systems that have no front panel. The SS (single step) line has been left undefined by the IEEE-696 standard and may be left as is. It is pulled up so that in the case of front panel-less machines, it will be ignored. The third input to U4b is from the 68008 read and data strobe lines combined through U3a. This turns on the input buffers during read cycles only. At all

U1	7404
U2	74LS00
U3	74LS02
U4	74LS10
U5	74LS14
U6	74LS30
U7	74LS32
U8-U10	74LS74
U11	74LS76
U12	74LS125
U13	74LS133
U14	74LS138
U15	74LS151
U16, U17	74LS241
U18-U23	74LS244
U24	74LS367
U25, U26	74LS368
U27	25LS2521
U28	2732
U29	68008
U30	data socket
R1-R39	2.2k 1/4 watt
C1	.01 ceramic
C2	220 electrolytic
C3-C6	4.7 tantalum
C7-Cn	.1 ceramic
XI	8 Mhz
sw1, Sw3	4-position DIP
sw2, Sw4, SW5	8-position DIP
VR1, VR2	7805

Figure 2: Parts List



copyright by the author, 1984



other times they are turned off. A data socket, U30, is provided for the actual connection to the front panel should one be present.

As an aside, I should mention that the design here is compatible enough with the IMSAI front panel (as modified to decode the lower 8 address lines instead of the upper) that almost all the features of the panel are still usable. Run/stop, single step, examine next, deposit and deposit next all will function. Only examine, which depends on the processor recognizing \$C3 as a jump instruction, will not work.

**Address bus.** The 20 address lines, AO-A19 of the 68008, are buffered through U19, U20 and U18b before driving the S-100 address lines. Since the 68008 does not implement A20-A23 as the 68000 does, these four lines are held at ground by U26a. All address buffers can be disabled by the ADSB\* bus signal through U5d.

**Status bus.** Many of the status signals for the bus are derived from the processor status lines FCO-FC2. When these are all high, an interrupt acknowledge cycle is being executed and U4c will respond to this. The sHLTA signal is simply a copy of the 68008 HALT\* line. Sixteen bit transfer signal sXTRQ\* is held high since the 68008 performs only 8-bit transfers.

Since the S-100 I/O space is reserved at memory locations \$FFF00-\$FFFFFF (in this design, the 68008 itself makes no distinction), bus read cycles that access this space must not activate sMEMR. U13 is a NAND gate whose output goes low only when the I/O space is accessed. It is combined with the 68008 R/W\* line via U2c to generate sMEMR. All the status lines up to this point are driven onto the bus through U25a.

To generate the closest thing to MI, sMEMR is combined with the processor status outputs that indicate either a user program or supervisor program access. This is accomplished with U1e, U7d and U3b. User data or supervisor data accesses do not cause MI to become active.

The 68008 R/W\* line itself is used for the sWO\* output. Both sOUT and sINP are the logical combination of the output of U13 with either the R/W\* line or its inverse, generated by U1f. These are combined through U3c and U3d and fed to the bus through U24a. All status signals can be disconnected from the bus by asserting SDSB\*, thus turning off U24a and U25a.

**Power-on jump circuit.** This part of the design is optional. If you are content to allow the 68008 processor to always fetch its initial stack pointer and program counter from location \$00000, you don't need this.

The output from U8b is used as a reset status bit. During power-on or reset, it is cleared. Its output is optionally fed through SW1-1 to one of the enable inputs of decoder U14. Another enable input of U14 comes from AS, of the 68008 to prevent false selection. For the first eight read cycles after a power-on or reset, the data read by the 68008 will come from tri-states U16, U17 and U23 if the power-on jump

switch is on. The output of U8b is fed through U2d and U7c to prevent the bus data in buffers from turning on during this period and causing a conflict.

Since the address inputs of the decoder U14 come from the lowest three address lines, each output in turn of U14 is enabled during the first eight read cycles. For the first five cycles, all that happens is that the output of U6 goes high, enabling U16a and U17a to drive a zero onto all the data lines. This corresponds to all four bytes of the initial stack pointer and the most significant byte of the initial program counter. For the sixth cycle, U16b and U17b are enabled, which drives a zero onto the high nybble of the byte and a value onto the low nybble that is a function of SW3 switch settings. This is the second byte of the initial PC. For the seventh cycle, a similar operation occurs with U23, all of whose outputs are switch selectable. Finally, for the eighth and last byte, another zero is placed on the data bus. This procedure allows a jump to any 256-byte boundary in the address space.

One last thing which occurs at the end of the POJ cycle is that pin 7 of U14, after going low to generate the last byte, returns high, which clocks the POJ flip-flop high and deselects the reset state. At this point, execution continues normally.

**On-board ROM.** This too is optional. If you have a ROM somewhere else in the system, you don't have to add it to your CPU board. However, it is handy to have a small monitor in ROM on the board to use for debugging purposes, something I found quite useful when trying to bring up CP/M-68K. See below for a source listing of such a monitor.

U28 is a 2732 4K x 8 EPROM. Address lines A0-A11 connect directly to the EPROM, while A12-A19 go to U27, an 8-bit comparator. When the highest eight address lines match the pattern set with EPROM address switches SW4, the comparator output goes low and selects the ROM whose data outputs connect directly to the on-board data bus. As with the power-on jump circuit, the bus data-in buffers are disabled through U2d and U7c during ROM read cycles. The ROM feature can be disabled by SW1-2 if desired. SW4 can be used to set the ROM address at any 4K boundary.

## Construction

Most of my general construction philosophy is available elsewhere (see "Build a High Resolution S-100 Graphics Board, Part 3" in Volume 2, No. 4 of *The Computer Journal*/ so I won't repeat it here. There are just a couple of things I would add to that. For power supply and ground lines. I used 22 gauge solid wire set up in a bus structure. Each individual chip socket then has a short piece of wire wrap wire soldered to the bus lines on one end and wrapped around the appropriate pins on the other end. See Figure 3 for a view of the back side of the board and Figure 4 for close-up detail of power supply and ground connections. The above-described method provides for a lower impedance from power supply to ground, and better noise immunity than using wire-wrap wire daisy chained from one socket to another. I have had no trouble whatsoever with improper

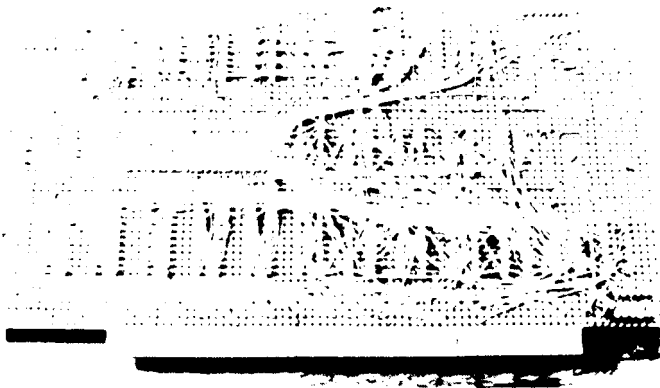


Figure 3

board operation using this technique and I would recommend it, even though it's a little more tedious than using all wire wrap connections. All signal connections, of course, are still made with wire wrapping.

Using two 7805 regulators, I split the power supply load roughly in half. In my own version, I supply the 68008, power-on jump circuit and on-board ROM from one regulator and everything else from the other. Both regulators run only moderately warm to the touch and you should have no problem there.

Figure 5 shows a front view of the board. I won't show a detailed parts layout for the board since a lot depends on whether or not you decide to include the power-on jump and/or on-board ROM options. The exact layout will be a function of this. Some general suggestions are just to keep the bus interface tri-states down at the bottom of the board near the S-100 fingers and the front panel data socket and DIP switches near the top where they're easy to get at.

### Software

The weak link once again. One thing that I'm sure has slowed down acceptance of the 68000 series is the lack of software. For operating system choices, you have either the UCSD p-system or Digital Research's CP/M-68K. I prefer the latter since I'm already running CP/M-80 and CP/M-86

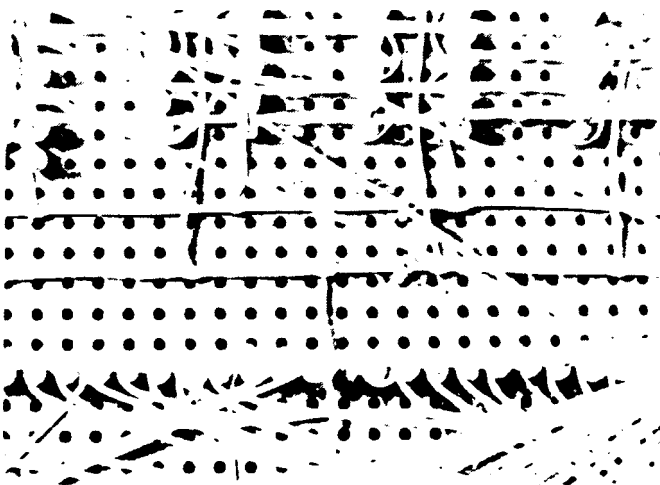


Figure 4

with a dual processor board. If you're fortunate enough to have a hardware configuration for which a CP/M-68K BIOS has already been written, you should have no problem other than the financial one. If you have a custom hardware configuration with homebrew boards and the like, your job will be much harder. The first step in getting some kind of disk-based system running is to have some kind of ROM monitor on the CPU board, first to verify that the hardware is functioning properly, and secondly, to use as a debugging tool for a disk-based system using a gradual bootstrapping process.

Figure 6 is a program listing of a ROM monitor I wrote to get started in this. Its basic commands emulate a subset of the DDT debugger, so those familiar with that will have no trouble using the monitor. One difference is that the "go" command to execute a program returns to the monitor with an RTS instruction so that you have to use the monitor's stack for your programs and avoid resetting the stack pointer (A7). The listing was made by one of two versions of a 68000 cross-assembler I wrote in order to bootstrap up to the 68008 from my 8085/8088 system. One cross-assembler runs under CP/M-80, the other under CP/M-86.

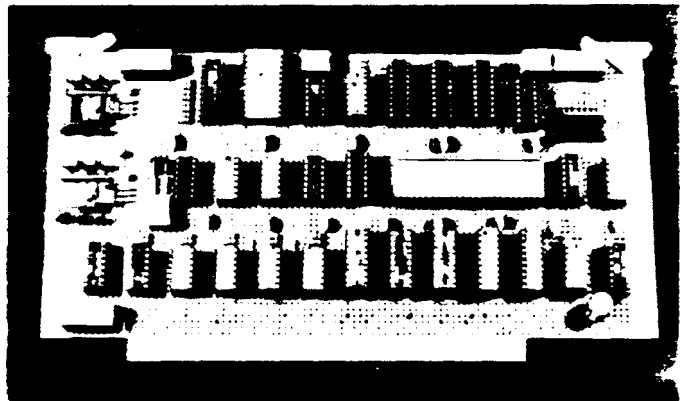


Figure 5

### Comments

The 68008 is definitely a high-performance processor. At 8 Mhz you can do a lot of operations in a short time. I also like the comprehensive instruction set, lots of registers, and built-in multiply and divide. Even with CP/M-68K running, there aren't a lot of languages or applications programs available for the 68000, although I expect this to change in the next year or two. If you really want to do a lot of things with it, you'll have to be prepared to do a lot of your own programming. The way I look at it though, that's really part of the fun of computing.

I would encourage you to build this CPU board. The power of the 68008 belies the simplicity with which it can be added to an S-100 system. Operation is speedy and reliable and the cost is quite nominal (about \$100 for parts). Once built, you'll probably not want to go back to your older CPU except to run application programs you don't have yet for the 68K.

*Listing 6 follows. References for this article are found on page 23.*

```

0001 ; Monitor for 68008 (emulates DDT)
0002 ; Version of 6/21/84
0003 ;
0004 STACK EQU $40000 ;256K RAM
0005 IOSPC EQU $FFF00 ;I/O space
0006 SIOADR EQU IOSPC+$00 ;SIO base address
0007 DATA0 EQU SIOADR+$00 ;SIO0 data port
0008 STAT0 EQU SIOADR+$01 ;SIO0 status port
0009 IBIT EQU 1 ;SIO input flag bit number
0010 OBIT EQU 0 ;SIO output flag bit number
0011 DTRBIT EQU 7 ;DTR flag bit
0012 LINLEN EQU 64 ;Maximum input line length
0013 ;
0000000 00020000 0014 ORG $20000
0015 ;
00020000 2E7C00040000 0016 MON68K: MOVE.L STACK,SP
00020006 207C0003F000 0017 MOVE.L SGNMSG,A0 ;Clear screen and display message
0002000C 610002E2 0018 BSR DSPLY
00020010 2E7C00040000 0019 MON: MOVE.L STACK, SP ;Reset stack pointer
00020016 702D 0020 MOVEQ '-',D0 ;Display prompt
00020018 6100030E 0021 BSR COUT
0002001C 61000224 0022 BSR GETLIN ;Get command line
00020020 67EE 0023 BEQ.S MON ;Empty line
00020022 0C000044 0024 CMP.B 'D',D0
00020026 672C 0025 BEQ.S DUMP ;Dump memory
00020028 0C000053 0026 CMP.B 'S',D0
0002002C 670000DE 0027 BEQ SET ;Set memory
00020030 0C0000 46 0028 CMP . B 'F',D0
00020034 6700011A 0029 BEQ FILL ;Fill memory
00020038 0C00024L 0030 CMF . B 'M',D0
0002003C 6700014A 0031 BEQ MOV ;Move memory
00020040 0C000047 0032 CMP.B 'G',D0
00020044 67000186 0033 BEG GO ;Go command
00020048 207C0003F012 0034 ERROR: MOVE.L ERRMSG,A0
0002004E 61000290 0035 BSR DSPLY ;Display '?' for errors
00020052 60BC 00 36 BRA.S MON
00020054 61000186 0037 DUMP : BSR GETARG ;Get first argument
00020058 2241 0038 MOVE.L DI , A1
0002005A 0C10000E 0039 CMF . B * $0L, (AC )
0002005E 6612 0040 BNE.S DUMPO ;More than one argument
00020060 2009 0041 MOVE.L A1 , D0
00020062 0680000000C0 0042 ADD.L $C0,DE
00020068 0280FFFFFFF 0043 AND.L ?FFFFFFF, DC
0002006E 24 40 0044 MOVE.L D0 , A2
00020070 6016 0045 BRA .S DUMP 1
00020072 61000256 0046 LUMPC : BSR G ETC OH ;.Check for separator
00020076 61000164 0047 BSR GETARG ;Get second argument
0002007A 2441 0048 MC\ 2 . ID1, A2
0002007C 61000166 0049 BSR GETEND ;.Check for end of line
00020080 B5C9 00 50 CMP . L A1 , A2
00020082 6402 0051 BHS.S DUMPOC ;Second argument is same or greater
00020084 2449 00 5 2 MOVE.L A1 , A2 ;Only display one byte
00020086 528A 0053 rumPee: ADIC. L >1, A2
00020088 2F09 0054 DCMF 2 : MOVE.L A1,-(SF) ;Save address on stack
0002008A 61000272 0055 BSR C SPA DR ;Display address
0002008E 1019 0056 DVMP2: MOVE.E (A1) + , D0
00020090 6100027E 0057 BSR HOUT ;.Display hex value
00020094 70 20 0058 MCVECL ' ',D0
00020096 61000290 0059 BSP COUT ;Insert a space
0002009A B5C9 0060 CMP.L A1 , A2
0002009C 6708 0061 BEC. S DUMP 3 ;Last value
0002009E 20 09 0062 MOVE.L A1 , DC
000200A0 0200000F 0063 AND. E $0F,D0
000200A4 66E8 0064 BNE.S DUMP2 ;Not an even 16 yet
000200A6 225F 0065 DUMP3: MOVE.L (SP)+,A1
000200A8 1019 0066 DUMP4: MOVE.B (A1) + , D0
000200AA 0C000020 0067 CMP.B ' ', D0
000200AE 6506 0068 BLC.S DUMP5 ;Non-printing
000200B0 0C00007F 0069 CMP. B $7F,D0
00020084 6502 0070 BLO. S DUMP6 ;Printing
000200B6 702E 0071 DUMP5 : MOVEQ ' ',D0 ;. Substitute '.' for non-printing
000200B8 6100026E 0072 DWMP6: BSR COUT
000200BC B5C9 0073 CMP.L A1, A2
000200BE 67 08 0074 BEQ.S DUMP 7 ;Last value
000200C0 2009 0075 MOVE.L A1 , De
000200C2 0200000F 0076 AND. B $0F,D0
000200C6 66E0 0077 BNE.S DUMP4
000200C8 207C0003F00F 0078 DUMP7: MOVE.L CRLF,A0
000200CE 61000210 0079 BSR DSPLY ;Dc a CRLF
000200E2 B5C9 0080 CMP.L A1, A2
000200D4 6700FF3A 0081 BEQ MON ;Last value
000200D8 08390001000FFF01 0082 BTST IBIT,STAT0 ;See if key pressed
000200E0 6700FFA6 0083 BEQ DUMPI ; No
000200E4 1039000FFF00 0084 MOVE.B DATA0,D0 ;Get the character
000200EA 0C000013 0085 CMP.B $13,D0
000200EE 670C 0086 BEQ.S DUMP 8 ; 'S

```

```

000200F0 0C000003 0087      CMP. B    $03,D0
000200F4 6700FF0A 0088      BEQ      MON68K ;Break
000200F8 6000FF16 0089      BRA      MON ;Go back to monitor on all others
000200FC 610001EE 0090      DUMP8:   BSR      CIN ;Wait for another character
00020100 0C000003 0091      CMP.B    #$03,D0
00020104 6700FEFA 0092      BEQ      MON68K
00020108 6000FF7E 0093      BRA      DUMP1 ;Continue with display
0002010C 610000CE 0094      SET:     BSR      GETARG ;Get address
00020110 2241      0095      MOVE.L   D1, A1
00020112 61000122 0096      BSR      GETEND ;Make sure EOL is next
00020116 610001E6 0097      SET1:    BSR      DSPADR ;Display current address
0002011A 1019      0098      MOVE.B   (A1)+,D0
0002011C 610001F2 0099      BSR      HOUT ;Display current contents
00020120 70 20     0100      MOVEQ    #' ', D0
00020122 61000204 0101      BSR      COUT ;Space
00020126 6100011A 0102      BSR      GETLIN ;Fetch new value if any
0002012A 67EA      0103      BEQ. S   SETI ;Skip to next location
0002012C 0C0000 2E 0104      CMP.B    ' ',D0
00020130 6700FEDE 0105      BEQ      MON ;End with '.'
00020134 5348      0106      SUBQ.W   1, A0 ;Point at beginning of line again
00020136 610000A4 0107      BSR      GETARG ;Get the value
0002013A 610000FA 0108      BSR      GETEND ;Make sure no more
0002013E 0C8100000100 0109      CMP.L    $100,D1
00020144 6400FF02 0110      BHS      ERROR ;Out of bounds
00020148 1341FFFF 0111      MOVE.B   DI,-1(A1) ;Modify value
0002014C 6000FFC8 0112      BRA      SETI ;See if more
00020150 6100008A 0113      FILL:    BSR      GETARG
00020154 2241      0114      MOVE.L   D1, A1 ;First address in A1
00020156 610000D2 0115      BSR      GETCOM
0002015A 61000080 0116      BSR      GETARG
0002015E 2441      0117      MOVE.L   D1, A2 ;Second address in A2
00020160 610000C8 0118      BSR      GETCOM
00020164 61000076 0119      BSR      GETARG ;Byte in D1
00020168 610000CC 0120      BSR      GETEND
0002016C 0C8100000100 0121      CMP.L    $100,DI
00020172 6400FED4 0122      BHS      ERROR ;Byte must be <=255
00020176 B5C9      0123      CMP.L    A1,A2
00020178 6500FE96 0124      BLO      MON ;First address is greater
0002017C 528A      0125      ADDQ.L   1, A2
0002017E 12C1      0126      FILL1:   MOVE.B   D1,(A1)+ ;Fill the byte
00020180 B5C9      0127      CMP.L    A1, A2
00020182 66FA      0128      BNE.S    FILL1 ;More to fill
00020184 6000FE8A 0129      BRA      MON
00020188 61000052 0130      MOV:     BSR      GETARG
0002018C 2241      0131      MOVE.L   D1, A1 ^Starting address in A1
0002018E 6100009A 0132      BSR      GETCOM
00020192 61000048 0133      BSR      GETARG
00020196 2441      0134      MOVE.L   D1,A2 .•Ending address in A2
00020198 61000090 0135      BSR      GETCOM
0002019C 6100003E 0136      BSR      GETARG
000201A0 2641      0137      MOVE.L   D1, A3 /*Destination address in A3
000201A2 61000092 0138      BSR      GETEND
000201A6 B5C9      0139      CMP.L    A1, A2
000201A8 6500FE66 0140      BLO      MON .•Ending address must be greater
000201AC 220A      0141      MOVE.L   A2,D1
000201AE 92 89     0142      SUB.L    A1, D1 ;Byte count in D1
000201B0 B7C9      0143      CMP.L    A1, A3
000201B2 63000010 0144      BLS      MOV 2 ;Destination is lower or same
000201B6 D7C1      0145      ADD.L    D1,A3
000201B8 B70A      0146      CMPM.B   (A2)+,(A3) + .•Increment both pointers
000201BA 1722      0147      MOV1:    MOVE.B   -(A2),-(A3)
000201BC 51C9FFFC 0148      DBRA     D1,MOV1
000201C0 6000FE4E 0149      BRA      MON
000201C4 16D9      0150      MOV 2:   MOVE.B   (A1)+,(A3)+
000201C6 51C9FFFC 0151      DBRA     D1,MOV2
000201CA 6000FE44 0152      BRA      MON
000201CE 2F3C00020010 0153      GO:      MOVE.L   MON,-(SP) ;Put return address on stack
000201D4 61000006 0154      BSR      GETARG ;Get go address
000201D8 2F01      0155      MOVE.L   D1,-(SP) ;Put on stack
000201DA 605A      0156      BRA.S    GETEND ;Make sure EOL is next
000201DC 4281      0157      GETARG:  CLR.L    D1 ;Use D1 as accumulator
000201DE 1218      0158      MOVE.B   (A0)+,D1
000201E0 04010030 0159      SUB.B    '0',D1
000201E4 6500FE62 0160      BLO      ERROR ;We need at least one real digit
000201E8 0C01000A 0161      CMP.B    10,D1
000201EC 6512      0162      BLO.S    GETARI ;Digit is 0-9
000201EE 5F01      0163      SUBQ.B   7,D1
000201F0 0C01000A 0164      CMP.B    10,D1
000201F4 6500FE52 0165      BLO      ERROR ;Illegal character
000201F8 0C010010 0166      CMP.B    16,D1
000201FC 6400FE4A 0167      BHS      ERROR . Doesn't qualify for hex digit
00020200 1018      0168      GETARI:  MOVE.B   (A0)+,D0
00020202 04000030 0169      SUB.B    '0',D0
00020206 65 IE     0170      BLO.S    GETAR3 ;Delimiter
00020208 0C00000A 0171      CMP.B    10,D0
0002020C 6512      0172      BLO.S    GETAR2 ;Digit is 0-9

```

```

0002020E 5F00 0173 SUBO.B 7, D0
00020210 0C00000A 0174 CMP.B 10,D0
00020214 65000010 0175 BLO GETAR3 .•Delimiter
00020218 0C000010 0176 CMP.B 16,D0
0002021C 64000008 0177 BHS GETAR3 .•Delimiter
00020220 E981 0178 GETAR2: ASL.L 4, D1 ;Multiply accumulator by 16
00020222 D200 0179 ADD.B D0,D1 ;Add in digit
00020224 60 DA 0180 BRA.S GETARI ;Go look for more digits
00020226 5348 0181 GETAR3: SUBQ.W 1,A0 ;Back up pointer
00020228 4E75 0182 RTS
0002022A 1018 0183 GETCOM: MOVE.B (A0)+,D0 .•Check for ', ' next
0002022C 0C00002C 0184 CMP.B ',',D0
00020230 6600FE16 0185 BNE ERROR
00020234 4E75 0186 RTS
00020236 1018 0187 GETEND: MOVE.B (A0)+,D0 .•Check for CR next
00020238 0C00000D 0188 CMP.B $0D,D0
0002023C 6600FE0A 0189 BNE ERROR
00020240 4E75 0190 RTS
00020242 207C0003F016 0191 GETLIN: MOVE.L LINBUF,A0 ;Storage pointer
00020248 4201 0192 CLR.B D1 ;Length counter
0002024A 610000A0 0193 GETLI : BSR CIN ;Input a character
0002024E 0200007F 0194 AND.B $7F,D0 ;Mask out parity
00020252 0C000003 0195 CMP.B 503,D0
00020256 6700FDA8 0196 BEQ MON68K
0002025A 0C000018 0197 CMP.B $18,D0
0002025E 6762 0198 BEQ.S CANCL .•Cancel the line
00020260 0C00000D 0199 CMP.B $0D,D0
00020264 6760 0200 BEQ.S EOL ;CR is end of line
00020266 0C00000A 0201 CMP.B $0A,D0
0002026A 67 5A 0202 BEQ.S EOL ;LF can also end line
0002026C 0C000008 0203 CMP.B #$08,D0
00020270 672C 0204 BEQ.S BS ;BS is allowable
00020272 0C00007F 0205 CMP.B $7F,D0
00020276 67D2 0206 BEQ.S GETLI ;Ignore DEL
00020278 0C000020 0207 CMP.B ', ',D0
0002027C 65CC 0208 BLO.S GETLI ;Ignore non-printing characters
0002027E 0C010040 0209 CMP.B LINLEN,D1
00020282 64C6 0210 BHS.S GETLI ;No more room in line
00020284 610000A2 0211 BSR COUT ;Echo the character
00020288 0C000061 0212 CMP.B 'a',D0
0002028C 650A 0213 BLO.S STORE ;Not lower case
0002028E 0C00007A 0214 CMP.B 'z',D0
00020292 6204 0215 BHI . S STORE . Not lower case
00020294 0200005F 0216 AND.B $5F,D0
00020298 10C0 0217 STORE: MOVE.B D0,(A0)+ ;Store character
0002029A 5201 0218 ADDQ.B 1,D1 ; Increase byte count
0002029C 60AC 0219 BRA.S GETLI
0002029E 143C0001 0220 BS: MOVE.B 1, D2
000202A2 4A01 0221 BS1 : TST.B D1
000202A4 6 7A4 0222 BEQ.S GETLI ;Already at beginning
000202A6 700E 0223 BS2: MOVEC. *$08, D0
000202A8 6100007E 0224 BSR COUT ;Move cursor back one
000202AC 7020 0225 MOVEC. ', ', D0
000202AE 61000078 0226 BSR COUT .•Overwrite with ' '
000202B2 7008 0227 MOVEQ >>$08,D0
000202B4 61000072 0228 BSR COUT ;Go back again
000202B8 5348 0229 SUBC.W 1, A0
000202BA 5301 0230 SUBQ.B 1, D1
000202BC 5302 0231 SUBC . B 1, E2
000202BE 66E6 0232 BNE.S BS2 ;Court is in D2
000202C0 6088 0233 BRA.S GETLI
000202C2 1401 0234 CANCL: MOVE.B D1,D2 .•Back up DI spaces
000202C4 60 DC 0235 BRA.S BS1
000202C6 10BC000D 0236 EOL: MOVE.B $0D,(A0) ;Store CR at end of line
000202CA 207C0003F00F 0237 MOVE.L CRLF,A0 .•Echo a CRLF
000202D0 610E 0238 BSR.S DSPLY
000202D2 207C0003F016 0239 MOVE.L {LINBUF,A0 ;Point at beginning of line
000202D8 1018 0240 MOVE.B (A0)+,D0 .-Get first character
000202DA 0C00000D 0241 CMP.B $0D,D0 ; See if CR
000202DE 4E75 0242 RTS
000202E0 1018 0243 DSPLY: MOVE.B (A0)+,D0 .•Display string
000202E2 4A00 0244 TST.B D0
000202E4 6704 0245 BEQ.S DS PLY 1 ;Zero terminates string
000202E6 61 40 0246 BSR.S COUT ,-Send the character
000202E8 60F6 0247 BRA.S DSPLY
000202EA 4E75 0248 DSPLY1: RTS
000202EC 08390001000FFF01 0249 CIN: BTST >>IBIT,STAT0 ;Test status bit
000202F4 67F6 0250 BEQ.S CIN ;Wait for data received
000202F6 1039000FFF00 0251 MOVE.B DATA0, D0 ;Read the byte
000202FC 4E75 0252 RTS
000202FE 2009 0253 DSPADR : MOVE.L A1,D0 .•Display current address
00020300 4840 0254 SWAP D0
00020302 610C 0255 BSR.S HOUT ;Do highest of 3 bytes
00020304 E1 98 0256 ROL.L 8,D0
00020306 6108 0257 BSR.S HOUT ;Next byte
00020308 3009 0258 MOVE.W A1,D0

```

# I COMPUTER\*

## T TRADER MAGAZINE

\* LIMITED TIME OFFER !  
**BAKER'S DOZEN SPECIAL!**

\$12.00 for 13 Issues

Regular Subscription \$15.00 Year

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

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

Classified Ads for Computer & Ham Radio Equipment

**FREE CLASSIFIED ADS**

for subscribers

Excellent Display and Classified Ad Rates  
Full National Coverage

**CHET LAMBERT, W4WDR**

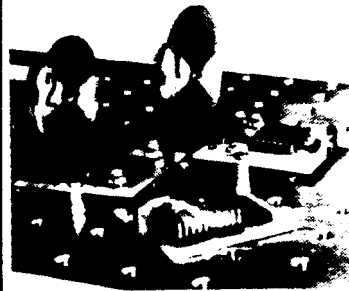
1704 Sam Drive • Birmingham, AL 35235

(205) 854-0271

Sample Copy \$2.50

## Low Cost, Modular Encoders

from a priori let  
you easily build  
and control:



- ROBOTS
- XY TABLES
- VEHICLES
- MOTORS
- CONVEYORS  
and more.

This modular approach to position and direction sensing of rotary motion lets the OEM or robotic designer fit the encoder to the job. Instead of fitting the job to the encoder PU-2/DISC and IM-1/UNIDISC form a flexible, low cost, precision dual-track encoder. Both packages output a 50 cycle/rev quadrature TTL square wave signal OSPB processes these signals and gives up/down pulses at 1, 2 or 4 times encoder resolution. A unique collet design lets DISC/UNIDISC mount on standard shaft sizes of .187", .250", .375" and .500".

**1. DISC (shaft size) 2-25" encoder used with PU-2 \$29.95**

**2. UNIDISC (shaft size) 1.85" encoder used with IM-1 \$24.95**

**3. PU-2** Retroreflective IR pick-up electronics for DISC. **\$29.95**  
Includes 4-pin connector. 2" by 2" pc board.

**4. IM-1** Through-beam IR pick-up electronics for UNIDISC. **\$24.95**  
Includes 4-pin connector 2" by 2" pc board.

**5. OSPB-1X, -2X or -4X** use with PU-2, IM-1 or **\$29.95**  
any dual track encoder. Outputs: UP/DOWN and PULSE, or  
UP and DOWN (add DC) Includes 4 and 5 pin connectors.

Call or write for brochure. It describes the above parts, plus linear encoders, sprockets and thin, flexible metal belts for drive systems.

Terms: Purchase Order or check Add \$3.00 shipping and handling Wash, residents add 7.9% sales tax

a priori

21318 125th S.E.

Kent, WA 98031

(206) 630-2144

```

0002030A 6104          0259          BSR.S  HOUT          ;Last byte
0002030C 70 20         0260          MOVEQ  #' 1 , D0
0002030E 6018         0261          BRA.S  COUT          ;Fol Low with a space
00020310 1F00         0262          HOUT:  MOVE.B  D0,-(SP) ;Output ASCII for hex byte
00020312 E808         0263          LSR.B  <<4,D0        ;Get high nybble
00020314 6102         0264          BSR.S  HOUTI
00020316 101F         0265          MOVE.B (SP)+,D0
00020318 0200000F     0266          HOUTI: AND.B  $0F,D0    ;Mask nybble
0002031C 06000030     0267          ADD.B  ' 0 ', D0    ;Add offset
00020320 0C000039     0268          CMP.B  >> ' 9 ',D0
00020324 6302         0269          BLS.S  COUT          ;0-9
00020326 5E00         0270          ADDQ.B 7,D0          ;A-F
00020328 083900000000FFF01 0271          COUT:  BTST  OBIT,STATO ;Test status bit
00020830 67F6         0272          BEQ.S  COUT          ;Wait for holding register empty
00020332 083900007000FFF01 0273          BTST  #DTRBIT, STATO
0002033A 67 EC         0274          BEQ.S  COUT          ;Wait for DTR high
0002033C 13C0000FFF00    0275          MOVE.B D0,DATA0     ;Write the byte
00020342 4E75         0276          RTS
00020344 0003F000     0277          ;
0003F000 1B2A3638303038204D4F 0278          ORG    STACK-$1000
0003F00F 0D0A00         0279          ;
0003F012 3F0D0A00     0280          SGNMSG: DC.B  S1B,'*68008 MONITOR'
0003F016 00000042     0281          CRLF:  DC.B  '$D,$A,@ '
0003F058 00000000     0282          ERRMSG: DC.B  '$2$0C,$0A,0
0283          LINBUF: DS.B  LINLEN+2
0284          ;
0285          END

```

### References:

MC68008 16-bit Processor with 8-bit Data Bus by Motorola Semiconductor (#ADI-939).

68000 Microprocessor Handbook by Gerry Kane (Osborne #41-1).

68000 Assembly Language Programming by Kane, Hawkins and Leventhal (Osborne #62-4).

# WRITING AND EVALUATING DOCUMENTATION

by Bill Kibler

Over the last several months, my series of articles on computer integration had a lot to say about documentation; most of it was not flattering. This is due to the common misconceptions about what documentation is and does. I recently took a course on software documentation which dealt with how to write good supporting documentation, and which brought many other topics to my mind. Criticizing without providing help is not a good way to write, so I have decided that this follow-up article would provide just such help. This discussion will also give users some idea of how to tell good documentation from bad.

## Documentation

In writing support documentation the first step is understanding the user. In the computer industry this has meant deciding whether your user is a beginner, an intermediate, or an expert. The inexperienced writer will assume one type of user and write for that audience. Even in my class that was the preferred method. I differ from that idea only in its implementation. Personally, I feel that all users pass through all levels of ability when using documentation. With any new equipment or software the first time use is that of a beginner. An expert will approach the information and scan the beginner section for important facts. However, the expert may take only ten or fifteen minutes to become familiar with it, whereas the beginner may take several days.

Because all users pass through different levels of experience when using documentation, it should contain many levels of help. The beginner is "lead by the hand" through the needed operations. Fear is still a problem and sample procedures are used to eliminate it. Pictures and drawings provide the most help for beginners and will point out the important bits or facts that even the experts should know. Intermediates will read the beginner section much like a beginner but more quickly, looking for warning statements. At the intermediate level what is needed is tutorial information. The tutorials provide an in-depth understanding of the product. This understanding is achieved through instructional examples and learning projects. The only difference between users is how long and how much they use the tutorial. A beginner may spend a year, an intermediate two to four months, and the expert a few days to a couple of weeks.

When people start feeling confident about their abilities, they will then start using the documentation only for reference. At this point such items as glossaries and indexes become important. In software the use of sections in which

the topics are listed and then broken out by function or use, seem to work the best. Short explanations with a sample of the routine procedures, all limited to one page, are ideal. This advice on understanding how people use their documentation is important to keep in mind when writing documentation, whether it be for software or hardware.

## Organization

The organization of a document should be the same throughout a company's product line. This advice is also good for both hardware and software. Technical readers expect to find the information in standard formats, and may be unable to use the product properly if the format is unusual. An organization that fits the levels of experience and at the same time meets the normal user's expectation is:

- Table of contents, both for the book and for each chapter.
- An overview or introduction section.
- The beginner section
- The intermediate or tutorial section
- The expert or reference information
- Appendixes
- Glossary
- Indexes (by as many ways as possible)

**Tables of Contents** give the user a quick overview of what to expect, and are a help in finding a certain topic. Tables at the start of each chapter can be expanded to show subtopics that would otherwise make main tables too large. Bleeding the index location to the edge of the page improves the documents much in the same way that index tabs help separate the chapters.

**An Overview** of the entire document's objective, intended users, and scope is needed at the beginning. Overviews at the start of each chapter further help explain what is going on and can help set the user's level of expectation for the next chapter.

**The Beginner Section** is not only for beginners, but for all first time users. This should have the 'how to turn it on' information as well as any warnings and dangerous situations to watch out for. When used by more advanced users this will be skimmed over, so large drawings and highlighted information should be used here.

**The Intermediate Section** starts the in-depth explanation of the product. This is generally done by starting out with simple topics and progressing to more advanced subjects as the user's skill level advances. A hardware document will have the theory of operation in this area, and the software document will provide information here about the structure

of the program. In either case, this section explains how everything fits together.

**The Expert Section** contains all the previous information arranged alphabetically, by topic or by use. Only one topic is covered per page (one topic per heading, if more than one page is needed). A typical format here would be a description of the topic first, some details, and then an example. The information will be used in a random method and therefore should be self-supporting.

**Appendixes** contain item-specific information or reprints. For hardware documentation, reprints of hard to get information are excellent, and diagrams of any custom ICs are necessary. For software users a step-by-step guide of some special or one-time-only procedures are found here. A good example would be the install routines of software programs and the hardware specific routines for system generation.

**Glossaries** are very important, as they can list words beginners have not yet fully understood. All levels of users can get confused at times, and glossaries will be the quick answer to their needs.

**Indexes** are very important for the more advanced users, as they can speed up their search for help while sitting at the terminal. Poorly designed indexes can be almost useless, so subtopics or descriptions should be used where possible.

### Content

When writing the document, the content of each section is actually more important than its layout. If the index makes it possible to find the information rapidly, but the information is not accurate, who cares about the index? This fact was recently brought home to a computer maker whose systems were being returned for failures at a rather large rate. The problem, however, was caused by incorrect instructions in the documentation, and not by component failures.

There are many styles of document writing currently vying for prominence. Simply, the best style will be the one that conveys the needed information clearly and quickly. I personally feel that locking into a certain style for an entire book doesn't work. An example is the Epson Grafrax manual. Except for a few appendixes, the writing style is intended for beginners and will insult anyone else. Although this tone may be appropriate at the beginning, it grows old quickly. Changing the slant of the writing as the skill level of the user increases would be a much better approach.

When changing the slant, it is important not to change the usage. This common problem is caused by the use of a thesaurus in technical writing. Whatever term is used to describe an action or function should be used ever more. You may have noticed that I have used the word 'document' consistently, even though 'manual' would mean the same for most readers. The careful avoidance of synonyms is absolutely necessary to prevent disastrous results as the level of complexity increases, and is especially important if the user is new to computers. This consistency works not only for text but for graphics as well.

The use of graphics cannot be overstressed. We all know the saying about one picture being worth a thousand words, but in documentation, it's worth time and money to Speed in using the document can be gained by using 'key' symbols to denote functions or topics. Flowcharts are graphics and can be used in many ways other than just program organization.

### Updating and Management

A major problem with most manuals is their inability to be updated. As quickly as things change in our industry, this can be an important aspect that some companies overlook. Those documents that have pages that can be replaced make it easier. A recent style is the bound book in which all information is done in an overview method, with specific information contained in other pamphlets. Updating is simplified by replacement of the smaller pamphlet.

Updating and support go hand in hand; both are ultimately a result of management decisions. Technical support is most often a division of sales, and as such, updates may be aimed more at selling you the latest than at fixing the bugs. Management is probably the largest stumbling block to effective documentation. We have all heard the saying that documents are a necessary evil. This attitude is changing, but so slowly that it could cause the downfall of some companies before it gets corrected.

I believe that a good company is a lot like our country—it needs a balance of power to operate efficiently. In the industry we have three groups; a manufacturing group, a sales group, and a management group. The relationship is something like this:

- sales finds out what clients want
- sales tells manufacturing what to make
- manufacturing designs it and makes it
- sales sells it
- sales supports it when it fails
- management checks and watches sales and manufacturing relationships
- management controls funds and thus quality of product
- when product sells well sales gets glory
- when product fails manufacturing gets funds cut by management

These are some of the many different cross paths of a typical business with a lot of unequal relationships. I feel that the relationship should be more like our own government (see Figure 1). Sales is a lot like the president, and should be telling manufacturing (congress) what it needs. "Congress" produces products to sell, but the "supreme court" controls whether it passes inspection or not. For me, the support organization is much like the supreme court and should be able to tell manufacturing when something needs to be changed. As with our government, the hub of this wheel is the people, or management, who have the final say in all matters.

The reason for this division is the ever-increasing needs of the user. As the systems increase in number and complexity, so too will the need for good support

documentation. The current problem is the lack of support for maintenance, the next boom industry.

### Maintenance Support

As more and more units are bought, more and more failures will occur. Warranties will handle some problems at first, but later on it will be repair sites and users that will handle the problems. For this to be possible, the documents will have to provide all the necessary support. Currently the information needed by independent repair people is not supplied in most users' manuals. It is not clear whether this is the sales department's way of getting you to buy, or just a result of the lack of knowledge about what users need and do. I personally feel that the fear of losing trade secrets may have the largest bearing on the subject. This may become a moot point as more sources for support start appearing, such as SAMS photofacts for computers.

### What It All Means

In the case of maintenance support, the user should check his documents at the time of purchase to see whether they contain enough information for him to do his own maintenance. For programs, this means that enough information should be supplied so that you can use the program for what you want it to do. I check the Table of Contents to make sure the topics I am interested in are there. Quite often I start from the back and go forward, checking the indexes, glossary, and reference sections. My quickest rule of thumb is to look at the appendixes—they give me a pretty good idea of how thorough those preparing

the document were. Lastly, however, there is still nothing to compare with an educated buyer. This statement holds true whether you are buying a system, a book, or a document.

### Conclusion

By no means have I covered all there is to preparing documents. If you are involved in writing documents, I hope that this article will get you to reflect on what you write. A considerable number of people who got their training in other disciplines are now finding themselves writing. They got their training by reading very dry and hard to read documents. This is reflected in their own dry and boring writing style. Those who start out as writers often try to write technical material in fancy prose, only to lose their audience through absolute confusion. What this means is that there is a need for special training in technical writing.

As I said at the beginning, I just completed such a course, and can clearly see now how this is true. Most managers know how important proper skills are for managing a company. The use of management seminars is becoming very commonplace, but writing seminars are not yet as common. If you write, take my advice and try one of the many writing courses available—they are well worth it. For those who only write in-house, or have penny pinching managers, join one of the technical writing organizations, and get into the habit of trading your work with others for comments. Nothing surpasses the field testing of documents for finding out if you did a good job or not. The end user is still the best test of a document's usability.

For us end users who get the written work inside the

*continued on page 29*

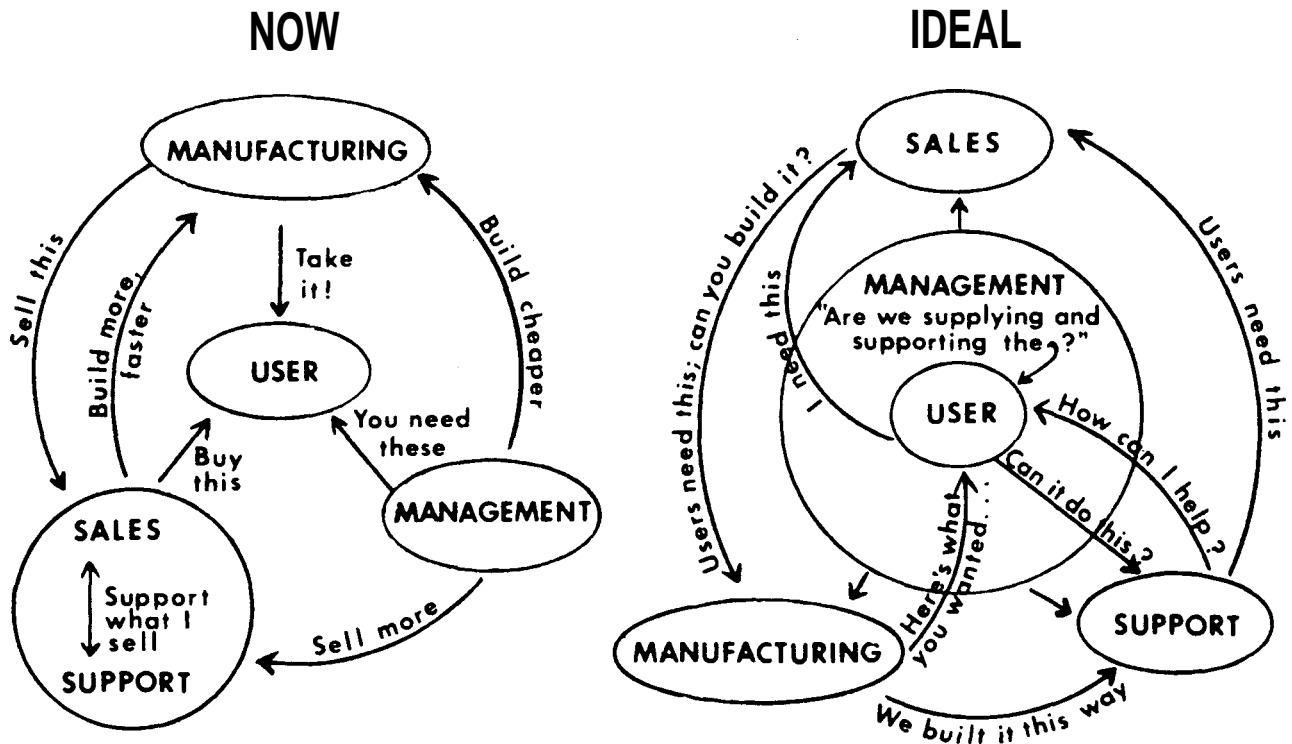


Figure 1: Manufacturing Management Structure.



# COMPUTER CONTROLLED ROBOTICS

1. DRIVER BOARD 5005 DB \$75 \*  
4 5' <38x05 ni CMOS COMPATIBLE  
OPTICALLY INSOLATED FOR 4 PHASE MOIORS.2AMP: - VO.15
2. LINEAR ACTUATOR 601 AM \$ 75  
12V 12W 16 OZ 00' STEP S ZE  
19 LBS HOLDING FORCE 3 IN IDAVE
3. LINEAR ACTUATOR 501AM \$43  
12V 3 5W 15OZ 002 STEF SIZE  
40 02 HOLDING FORCE 1 88 IN TRAVEL
4. STEPPER MOTOR 201 SM \$ 16  
5V 2W 1 O 02 15 - STEP SCE  
08 0? IN HOLDING IORGE
5. STEPPER MOTOR 301 SM \$ 59  
12V 21 5 OZ 1B STEF SIZI  
80 OZ IN HOLDING TORQUE
6. MOTOR MOUNT FOR 301 SM \$25
- 7 MOTOR MOUNT FOR 501 AM \$ 12
- S. MOTOR MOUNT FOR 501 AM \$ 13

\* EDGE CONNECTOR \$ 3.50



**AMSI cORp** (\$61361-3499)  
BOX 651, SMITHTOWN, L.I., N.Y. 11787



*Interfacing Tips. Continued from page 13*

Figure 1 on page 3 directly couplable to the TS2068, or is some modification needed? What software modifications would I need to make, such as PEEK and POKE statements and RAM memory allocation? Also, what data on the TS2068 would you need to know to advise me on how to adapt the circuit for the VIC-20 EPROM Programmer to the Timex TS2068 in both software and hardware considerations?

Enclosed is an addressed, stamped envelope. Any advice would be helpful.

Bryan Lepkowski, New York

Dear B^yan;

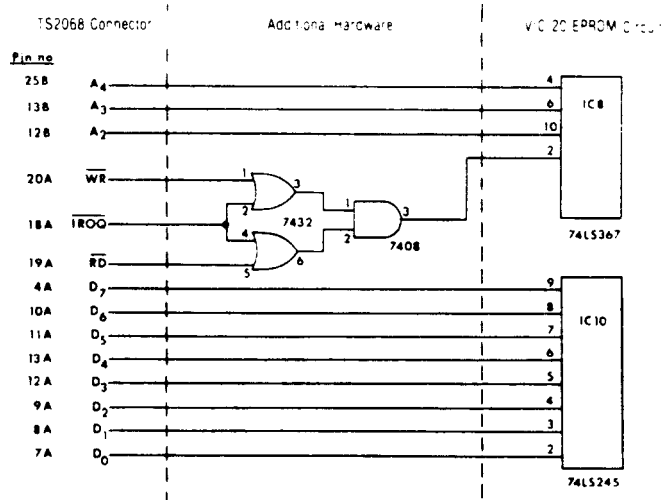
Your question about interfacing the TS2068 to the VIC-20 EPROM Programmer is an interesting one. The hardware changes which are required are not extensive, but the software differences are rather involved.

The Commodore machines (VIC-20 and Commodore 64) use memory mapped input/output (MMIO) for interfacing, but the Sinclair machines (ZX81, TS1500, TS2068) cannot use MMIO because of idiosyncrasies in ther address decoding scheme. This leaves accumulator input/output (I/O) as the Sinclair's means of interfacing, which requiries a different software approach than the Commodore's MMIO. To make matters worse, there is no way to execute the accumulator I/O instructions from the Sinclair's BASIC language. You must write the interface "drivers" in machine code and execute them as "USR" subroutines from BASIC.

The answer to your question is interesting enough that I am going to address it directly in my column with an installment on interfacing the Sinclair machines. It will be November before this information will appear in The Computer Journal, so I will send you enough information to interface the EPROM programmer to the TS2068 as soon as I have all the details worked out.

I have enclosed the required hardware changes in case you want to attempt the software for the interface yourself. If not, I will be sending you additional information shortly.

Thank you for your interest,  
Neil Bungard



## Reader Design Project:

# AN ELECTRONIC DIAL INDICATOR

by Art Carlson, Editor/Publisher

Now that microcomputers are well established in the business world, the next area of rapid growth will be that of using micros to control the real world—and many of these applications will require measuring and controlling the position of a mechanical device. There is a problem, however, in that much of the technology needed for these applications does not exist on an affordable level.

Computer machinists and robotics experimenters need to convert the mechanical position of an object to electric signals for the purpose of measurement or control. For several years I have thought that I would like to use a computer to control my lathe. I am sure there are thousands of applications for the electronic equivalent of the dial indicator commonly used in machine shops, but I have not been able to locate a low-priced, hacker oriented transducer, nor have I been able to come up with a good design myself. These devices do exist for companies such as General Motors who have large budgets, but they are too expensive for the amateur machinist or the educational experimenter.

Since one of the purposes of *The Computer Journal* is to encourage the sharing of knowledge and ideas for solving common problems, we are presenting the electronic dial indicator as a reader design project. I'm sure that our inventive readers can come up with some good design ideas.

There are so many possible applications that no one design will be suitable for all of them. Our goal is to design electronic indicators for several of the most frequent uses. This information will be placed in the public domain for use by all, and should include sufficient background information so that others can customize the designs for their individual

applications.

The purpose of this project is to design a transducer to convert the position of an object to an electric signal which can be displayed on a digital readout and/or used by a computer for control purposes. The device should interface through standard parallel or series interfaces so that it is not computer-specific. For convenience in writing, and to use a consistent term, we will refer to the device as the *indicator* and define this to mean the portion which converts mechanical position to an electric signal. There will be several classes of indicators for different combinations of range and sensitivity, as most devices either measure extremely small variations over a small range, or large variations over a large range. Perhaps our readers will surprise us with an indicator which can respond to small variations over a large range (and still be affordable by a hardware hacker).

A frequent application for a dial indicator in the machine shop is to measure the runout when setting up a piece in the lathe (see Figure 1), or to sense the position of a cutting tool for making cuts or holes to an exact depth (see Figure 2). An electronic indicator which could be coupled to either a digital readout or a computer would be very useful in the machine shop and for other mechanical positioning applications.

### Design Requirements

1) Resolution: The indicator must be able to resolve a displacement of 0.001 inch (one-thousandth); a resolution of 0.0001 (one-tenthousandth) would be better.

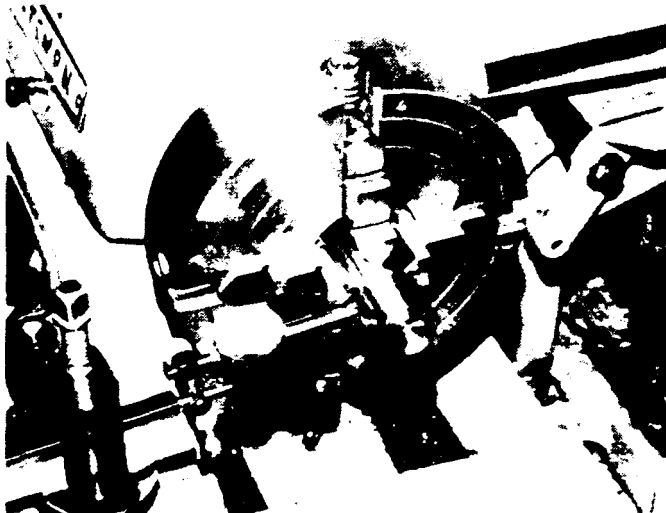


Figure 1

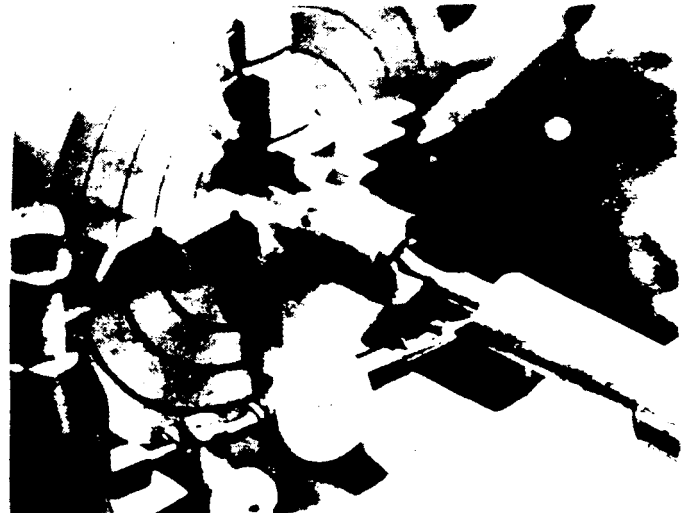


Figure 2

2) Range: The indicator must be able to respond over a range of at least 0.030 (thirty thousandths) of an inch; a range of 0.060 (sixty thousandths) or more would be better.

3) Overtravel Protection: The indicator must include a mechanical stop to avoid damage caused by displacement in excess of the range, or must be capable of mechanical displacement in excess of the useful range without damage to the indicator.

4) The indicator must work with any type of test piece (steel, brass, aluminum, plastic, etc), and should not depend on the characteristics of the test piece. It should also work with a test piece having a surface which may be smooth, rough, or dirty (covered with cutting oil).

5) Safety: The probe portion of the indicator should operate with low voltages of limited current in order to avoid the possibility of electrical shock or damaging high currents if the cable should be cut.

6) Output: The electrical output from the indicator should be suitable for input to popular computers. The reading should be stable and consistent.

7) Direction: The indicator must respond to the direction of travel, and indicate the absolute position if it is fluctuating above and below a certain value. 8) Interface: The indicator should interface to the computer through a standard interface such as the parallel Centronics® or series RS-232 interfaces so that it can be used with most common computers.

These are the minimum requirements for an indicator, and can be expanded for other applications.

### One Approach

As an example of a simple, but workable approach to this design problem, I'll outline some thoughts I have developed. This is not necessarily a good approach, but it is easy to do with readily available materials. I believe in using "cheap and dirty" methods for one-of-a-kind devices when the primary objective is to accomplish some other task. In this case, I want to make a useful indicator so that I can proceed

#### *Documentation, continued from page 26*

packing box, there is not much we can do about bad documentation except not buy. The lack of funds will stop many companies from producing poor documents. A more helpful way, however, is constructive criticism. Good documents will have a postage-free return card for your comments; use them when you see ways to change things for the better. Comments like "the document is terrible" help no one. Be precise and informative, and let them know that you would be willing to review their next version for free.

When buying equipment or books, you should keep in mind the points I covered above. Remember to check for indexes, glossaries, tutorials, and appendixes. Ask yourself first, "what do I really need from this document," "how do I plan on using the material," and "are my needs going to change?" Getting these points clear in your mind before checking out what is available will make things much easier. Above all else, become an educated buyer.

with the job of automating my lathe, rather than spending months developing an elegant indicator to be produced in large quantities.

The indicator shown in Figures 1 and 2 contains the mechanism to change a small motion of the tip to a large rotation of the pointer. One approach would be to fasten a fiber optic pickup on the pointer to sense the black gradations on the dial as the pointer rotates. It is likely that there would be a problem with the amount of force required of the pointer to move the fiber, so I thought of letting the pickup remain stationary while the marks rotate. One could draw the gradations on a thin piece of clear plastic, remove the pointer from the shaft on the dial indicator, and then fasten the plastic disk on the shaft so that the disk rotated in place of the pointer. The optoelectronic pickup would detect the lines on the rotating disk, and this signal would be used by the computer or the digital display. In practice, the pattern on the disk would have to consist of a simple pattern viewed by two detectors in order to determine the direction of motion. Otherwise a small back-and-forth motion around a fixed point would be shown as a large displacement.

### Conclusions

The use of an indicator on the lathe is a very simple example used here merely to illustrate the principles. There are many more involved applications for the indicator in robotics and computer controlled machining where there is a need to determine where something is located or measure how much something is moving under load. A good example would be measuring how far an internal part in a robotic device bends under load. Another example would be upgrading a robotic parts placement device for greater precision by using the signal from an indicator to control the final position of the part.

We encourage our readers to share their knowledge in this area with others, and to submit additional ideas for reader projects.

### Recommended Reading:

"Why Johnny Can't Document," Sandra Pakin & Associates Folio, 6007 N. Sheridan Road, Chicago, IL.

"Method For Designing Computer Support Documentation," Master's Thesis, Sept. 1983, Report no. LSSR 54-83, Department of Communications, AFIT/LSH, WPAFB, Ohio 45433.

"The New Playscript Procedure, Management Tool For Action," by Leslie H. Matthies.

**Software Psychology** by Ben Schneiderman. Wintrop Publishers, New York, 1980.

**The Elements of Friendly Software Design**, by Heckel.

Join the "Society for Technical Communications," 815 15th Street NW, Washington D.C. 20005.

## Letters From Our Readers

Dear Friends:

Here is my subscription renewal. I just want you to know that your magazine is one of the most interesting and useful to which I subscribe. I have subscribed to more than two dozen magazines over the past years, and some I have let drop, while others have ceased publication. But for my purposes the best three are yours, *Microsystems*, and *Computer Shopper*. I would have to put *Byte* fourth; I would drop it except for what I learn from reading the ads (they should give it away as a throwaway instead of charging for subscriptions).

An article (or series) I would like to see very much would be on uninterruptible power supplies. There is so much hype about UPS that I just can't sort any of it out. I think I need one, yet some have square-wave output and others have sine-wave or "sinusoidal" output (sounds like a respiratory disease, but I guess it means "almost sine-wave, but not quite"). One ad says "Not For Use With Linear Power Supplies," and another says "Note: do not use with refrigerators, capacitor-start or split phase motors, inductive reactive capacitive loads." That seems to let me out, with the big capacitors on my hefty Integrand linear power supplies. And why should they COST so MUCH? Why do they come with batteries included so that they weigh over the 70lb. shipping limit and so have to come by common carrier motor freight, when I can buy perfectly good batteries right here in town? If the square-wave output type will not work with (or will damage) my linear power supplies, could I still hook my terminal up to a square-wave UPS, plus maybe the AC motor drive line to my 8-inch drives? I had the idea of using a battery charger, a deep-cycle battery, to a DC-to-AC power inverter, then cleaning up the inverter output with one of the powerline regulators. That would be modular and much cheaper to buy and repair than the ready-made UPS units, but it is still too much money to buy the pieces just to see if they work together. At this point, I am totally bogged down.

I want to pass on a positive experience with Wyse Technology in San Jose; the makers of the Wyse-50 terminal I am using with my S-100 system. The keyboard layout and feel, the 16 x 2 user-programmable function keys (arranged in groups of four keys, so that four fingers fit right on top of them), and the easy-to-read 14" screen, give this terminal the best price/performance ratio for me that I have yet seen. One week after the three-month warranty expired, the terminal died. I called the dealer I bought it from (a "full-service full-price" store, not a mail-order outfit), and he tried to pretend he didn't know me. So I called Wyse directly, and they offered to repair the terminal in San Jose, and since it was an early serial number, they would fix it free and pay the shipping both ways, even though it was out of warranty.

I thought I would have the terminal running faster if a tech friend fixed it right here. Wyse said that would void the free repair offer. I replied that if I could get schematics and maybe a repair manual, I was willing to waive the free repair. They said the repair documentation was only for dealers and OEMs. I answered that everyone in Wyoming is an OEM, because out here we have to fix things ourselves when they break down, or else send them to some city and do without for a month. So Wyse got the schematics and manuals to me FOUR days after the phone call. In a field where there are so many broken promises, this impressed me greatly. It took us about an hour to troubleshoot the terminal and replace a dead diode, and now it works perfectly.

On the other hand, I recently bought a floppy drive from Priority One Electronics in Chatsworth. It arrived with configuration jumpers on the PCB, and the drive door open (although the manual says clearly that the drive door should be closed during shipment). The foam container had the name of a large computer manufacturer on the sealing tape. I called Priority One, and they tried to tell me that the drive had been "checked out" prior to shipment. How I was supposed to tell the difference between a "checked out" drive and a "used" drive, he was not able to explain. It does have some minor problems, and so appears to be a quality-control reject from the large California OEM. So, no more drives for me from Priority one. I know what a factory-sealed drive looks like, because I have bought other drives from other vendors. When I pay good money for a new drive, I doggone well want to be sold a brand spanking new drive!

Keep up the good work. Your magazine sure is needed. I am sick and tired of reading reviews of both hardware and software which are nothing but promotional hype. The *Info World* reviews have gotten worthless: maybe they always were. If a review doesn't find anything wrong with the product, I figure there is something wrong with the review, because I have never seen a computer product yet that didn't have problems or bugs. Besides, to make an informed buying decision, the consumer needs to know how a product compares in features and performance with other substitutable items.

P.L.G.

Wyoming

*Ed note: Uninterruptible power supplies are something which many people are interested in. Perhaps some readers with experience in this area could provide some helpful advice. Readers?*

Dear Sirs:

Your advertising literature for *The Computer Journal*

makes it clear that this is the magazine for me.

I am an assembly language programmer on micros and minis. I program in several higher level languages: C, Cobol, Basic, Pascal, etc.

My interests are in assembly languages, real world interfacing, hardware, peripherals, trouble shooting and servicing, and robotics. I feel that your magazine will hit the spot for me no matter which way you go, as long as it doesn't become yet another consumer oriented hardware/software review magazine. If I read one more review of Lotus 1-2-3 or of dBase II, I shall scream!

I would like to order all of the available back issues along with a one-year subscription. I look forward to receiving the first issue.

M.B.  
California

Dear Sirs:

Since subscribing recently, *The Computer Journal* has become my favorite magazine. Keep up the good work.

G.K.  
New York

Dear Friends:

Thank you so much for your series on Threaded Languages. I am having a ball with this one. Please keep the series coming—I can't wait for the next issue!

D.T.  
Washington

Dear Bill Kibler,

I enjoyed reading your articles on integrating systems, and particularly the one called "Dos Wars." It is long since time that someone wrote some true technical facts about some of the "hype" that is being put out about computers nowadays.

While all of us have personal prejudices regarding both hardware and software, it is refreshing to read an article that examines things in enough depth to point out some true facts.

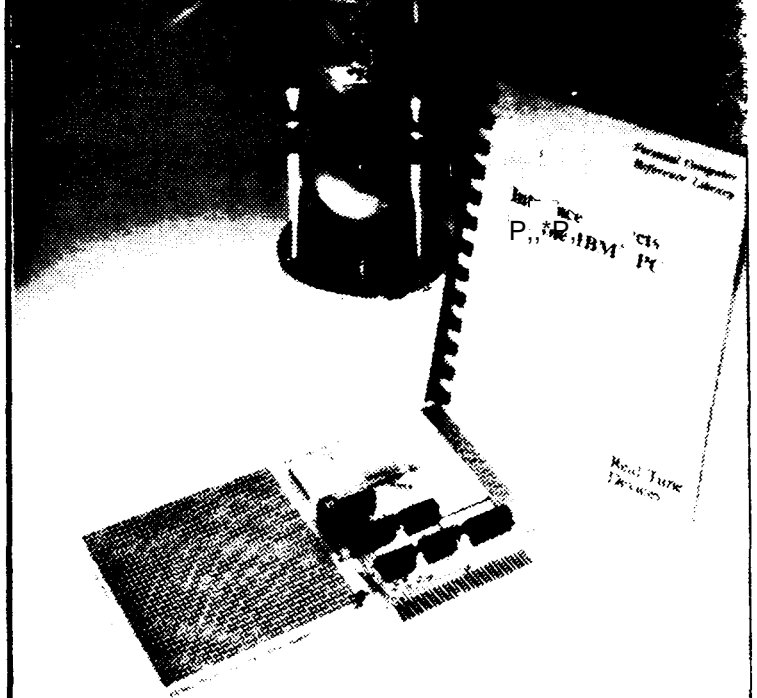
I recently retired my TRS-80 Model I, but not before I spent many hours programming in Z-80 assembly language. If one examines the Z-80 instruction set it is a wonder to me that the "hypesters" didn't call it a 16 bit processor. As you know, it does have 16 bit move and load instructions as well as 16 bit arithmetic instructions. (One minor pet peeve I have with guys like you is that I never learned to speak 8080 and resent all the code I see still kicking around written in 8080. Come now, an 8080 hasn't been used in a computer since MITS and Altair went out of business.)

My present machine is an Altos Model 5-15, and I have both MP/M II and CP/M 2.2. I needed the MP/M II for some multiuser consulting programming I have been doing, but as you pointed out in your article, try and get some technical information on it!! I was not particularly impressed with this machine when I first got it because the client required MP/M II running a version of Business Basic called B1280. Once that stuff was out of the way I got CP/M 2.2 and some good programming tools like Microsoft's Basic compiler,

*continued*

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

prototyping the same circuitry  
when you can Quickly and easily  
implement your original design  
with Real Time Devices PD 100  
Hardware Development  
board



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

### MANUAL TOPICS

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

### BOARD FEATURES

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

### ORDERING INFORMATION

PD100 WITH MANUAL - \$99.00 PLUS \$3.50 P&H  
MANUAL ONLY - \$20.00 POSTPAID  
PENNSYLVANIA RESIDENTS ADD 6% SALES TAX

MASTERCARD AND VISA ACCEPTED SEND CHECK OR MONEY ORDER TO

**REAL TIME DEVICES**  
1930 PARK FOREST AVENUE  
P.O. 80X906  
STATE COLLEGE, PA 16801

PHONE (814)234-8087

DEALER INQUIRES WELCOME

Microsoft's macro-assembler, and very recently Borland International's TURBO-PASCAL. (Another aside —TURBO-PASCAL is one of the best software packages I have had the good fortune to use, BAR NONE!!) I am just now beginning to appreciate what a really good Z-80 based machine can do. My terminal is a Televideo 950, and is truly a "professional" device.

I do have one small problem though. You spoke of public domain software on 8" disks. The Altos gives me a great disk capacity of over 1.5 megabytes on two 5" floppies, but unfortunately that requires 80 track double sided, double density formats. The logical block length in this version of CP/M 2.2 is 4K and the physical sector length is 512 bytes. Could you guide me to a source of public domain software in this format? As you may also know, the Altos is a single board machine and there are no provisions for adding another drive, otherwise I would simply add an 8" drive and be on my way.

Write some more fine articles. Next time why don't you take on the people promoting those stupid "mice" and cute little pictures! I think that to run a computer a person ought to be able to read and write!

L.S.P.  
California

Dear L.S.P.,

Thanks for your comments on my articles. Your question on "why 8080" programming should have been answered somewhat in my article on documentation, and in the upcoming one on programming tricks. To summarize, the makers of CP/M provide only an 8080 assembler, so to keep our readers from buying more software, I do it in 8080. Also, 8080s are not dead; my Z100 uses an 8085 and will not run Z-80 code. This problem will change soon, as National Semiconductor is making a "plug compatible" NSC800 to replace the 8085. The NSC800 is an 8085-looking device with the insides of a Z-80, all done in CMOS.

For public domain software on odd formats, try the Computer Shopper—several advertisers sell the disks in non 8" formats. Other than that, using a modem between other's systems or local nets is your only choice. I also have bought the TURBO-PASCAL and have yet to find fault with it. Not only is it a good product, but they have learned how to support the users' needs. Look for even better things from them in the future. And from us. Keep watching The Journal, as I am working as fast as I can on lots of needed topics. Got one I haven't covered yet?

Bill Kibler

Dear Computer Journal:

Please renew my subscription. Very good magazine —almost like *Byte* in the early years before it got too commercial and software based.

W.W.  
Illinois

## Searching for Useful Information?

Back issues of *The Computer Journal* make excellent reference material. Back issues: \$3.25 in the US and Canada. \$5.50 in other countries (air mail postage included). Send payment with complete name and address to PO Box 1697. Kalispell, MT 59903. Allow 3 to 4 weeks for delivery. Issues 1 and 2 of Vol. 1 have been sold out. The following back issues are still available:

### Volume 1, Number 3:

- Add an 8067 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
- Gemini10X. 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 Two 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: CPNet
- 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 PC DOS Diskettes with the Morrow Micro Decision
- LSTTL Reference Chart
- DOS Wars
- Build a Code Photoreader

### Volume 2, Number 6:

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