ISSN # 0748-9331

THE COMPUTER JOURNAL®

For Those Who Interface, Build, and Apply Micros

Issue Number 14

$2.50 US

Editor's Page

The Future is Bright in Some Microcomputer Markets

We spend so much time thinking about the companies that are laying off their help or filing for Chapter 11 that we lose sight of those that are doing well. This month we will concentrate on the areas which we expect to grow and prosper.

I feel very strongly that the next real growth in microcomputers will be in managing the real world. This includes, but is not limited to, robotics, measurement, control, manufacturing, automotive, scientific research, medical, household, hobby, and thousands of other applications. The computer industry has been so busy developing the business, home, and personal computer markets that there has been relatively little work done on the real world applications.

We already have microprocessors in watches(????), microwave ovens, cars, stereos, and factories, but the expansion is being limited by a shortage of people with technical knowledge and imagination who can conceive and implement new applications. We need people experienced in using microprocessors to make things happen instead of using them to process data as in business programs. Until now most of the work on computer controlled applications has been done on mainframes, but now more of this work is being done on micros and the impact of using micros for controlling the real world will affect our life styles at least as much as the use of micros in the home and office already has.

The pioneers in this area (yes, there are still pioneers in the micro industry) are largely individuals familiar with hardware and software who are very curious about how things work and who can foresee how microprocessors can be made to perform a necessary function. We can't supply the curiosity—the need to know why—but we do intend to supply the information required to figure out how to accomplish the idea once it is formed. And perhaps we can prod the curiosity a little by showing what others are doing and presenting questions and challenging ideas.

One example of the type of applications people are working on is the monitoring of the water level in a standpipe or well. How do you determine the fluid level? How do you store and safeguard the data in a remote area with a hostile environment far from power or phone lines for extended periods of time? How do you acquire the data and reset the device without risking the loss of the data? All this must be done on a limited budget, so solar powered satellite microwave links are out.

The people working on projects like this are forced by necessity to be both hardware and software hackers who can work with a soldering iron as well as with a keyboard. There are many people and organizations who can fine tune and polish the device once a prototype has been built, but there is a definite shortage of people who can see the need and devise some way to accomplish it.

A person has to be proficient in many different skills to develop projects in this field. Some of the areas which we intend to cover are: understanding and using compilers, assemblers, linkers, and loaders; choosing and using utility libraries with standard languages; special languages such as FORTH, LISP, SAVVY, etc.; assembly language programming; developing public domain subroutines and programs for our area of interest; sensors for measuring position, distance, force, temperature, pressure, etc.; using PROMs, EPROMs, and EEPROMS; interfacing between the micro and the real world; developing stand-alone microprocessor controllers; and much more.

It is impossible to write at a level which will satisfy all of our readers. What is way over the heads of some will be too

continued on page 5

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

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

Entire contents copyright © 1984 by The Computer Journal.

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

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

HARDWARE TRICKS

by Bill Kibler

In the first article on "Tricks of the Trade" (Vol.II, No. 12), I explained how I do some of the programming for system integration, specifically, installing new I/O drivers in a BIOS. Let's look now at how to handle the hardware aspect of setting up a system. We'll start at the beginning—with the power supply.

Power Supplies

No system can run without power, so let's look at some concerns in the category of power supplies. For S-100 systems a large but non regulated supply is needed for the bus supplies (the main supply is non-regulated because there are regulators on each board). Three voltages of +8, +16, and -16 are used by the individual cards to get +5, +12 and -12. For any -5 volt needs the -12 will be used. Current draw on the +8 can be excessive, but the others will draw nominal amounts. For single board systems, supply voltages are +5, +12, and (sometimes) -12. Single board systems will need better regulated supplies than S-100, but both will need fairly clean outputs.

Most new systems use switcher supplies for better efficiency at a lower price. However, these physically smaller units have one drawback—mainly their noise passing abilities. To understand this, one must understand the two types of designs. The older "boat anchor" style of design, which has been around for many years, uses a large transformer to provide the desired voltages. A bridge rectifier is used off of each winding of the transformer for each voltage. Large amounts of capcitance, near one farad, are used to smooth out the ripple. This large amount of capacitance also removes noise, and provides a boost type of reserve for line variations. Having used several systems with this type of supply, I prefer them in environments which are noisy or have poor AC sources.

The switcher is a rather new power supply design, using cheaper and lighter components. The voltages are obtained from an oscillator (switching frequency) generated signal that is rectified to produce the desired voltage. The higher the frequency of oscillation the lighter the transformers become, and also the higher their efficiency. This is why aircraft systems use 400hz instead of 60hz; it provides a great reduction in size and weight, a real concern in aircraft, not computers. In theory, the only difference is the source of the supply—60 hz versus 400hz, but in actuality, there is more. Some switcher supplies will put noise into disk drives if not properly shielded. Noise spikes seem to pass through switcher supplies much more readily (lower output capacitance), and some supplies will not work properly if the load varies too much from their designed specifications.

The choice of which type of supply to use is normally not yours to make. Most manufacturers have gone to switchers to save money and sell noise filters. When building your own power supply, I would recommend using the older style "boat anchors," especially in noisy areas. Cost is not usually a consideration as non-switcher parts are more readily available than switcher parts. Also, it is generally easier to over-design a boat anchor than it is to buy an over designed switcher. Over-designing becomes more important as you add extra boards or circuits. Extra amounts of power (sometimes 100% extra), are needed to handle that initial surge of computing or disk accessing. Some rather strange problems have been traced to poor source supplies which were unable to handle that extra surge. An example was a disk supply with a marginal 24V supply. If stepped once it would work fine, but when returning to track zero, it would fail. When the supply was checked with a scope it showed that the voltage dropped under demand to less than 18V. Changing the design and using larger capcitors solved the problem.

UIP

The newest scam is the UnInterruptible Power source (UIP). These units are being sold to prevent loss of data during a power failure. Although I have as yet to see a power failure while computing, I suppose it does happen to some people. My personal feeling is that unless you are doing life threatening activities, proper backup procedures will provide the needed protection. Taking a second to dwell on what those procedures are, I would like to say that anyone doing computing with orginal disks or unbacked-up files deserves what can and will happen to them.

Loss of data is no laughing matter, and losing several hours of work can be both costly and frustrating. As a writer, I stop every 15 to 30 minutes and do a ◄CTRL K► S with WORDSTAR, which saves my additions to the file and gets me back to where I left off. This means that if something should happen, I would not lose more than a few minutes of work. However, this is still not foolproof, as a power failure can destroy the disk. A second backup disk, one which is unloaded from the machine when not transferring data, is an absolute necessity. I know of one case in which a power cord was accidentally kicked out. The users's backup disk was kept in the fourth drive with the door closed. After rebooting with a new boot disk, it turned out that all four disks had been damaged. Some disk doctor type programs may save data from a damaged disk, but the usual procedure is just to reformat and start over.

For those people who use boat anchor type supplies, there is a way to protect yourself. It can take many machine cycles for the supply voltages to drop to unusable levels in supplies with rather large amount of capacitance. We are not talking minutes but rather 1 to 10 sec or more (depends on size of supply and number of circuits). This time is actually enough to sense the power failure and save all of memory to disk. The procedure would be to use a circuit that forces a NON-MASKABLE interrupt should the line voltage fail. This special interrupt routine then saves all registers and memory banks to disk under a special file name. After power is returned a second program is run to reset the machine. I know of one such system and have planned on doing that to my own system but as yet have not had a reason to need it.

Should you determine that you actually need a UPI, you will then be faced with the problem of what type to get. Where I work we have several backup supplies, mainly to provide power for emergency two-way radio systems. A computer system that monitors critical temperature alarms has not only backup power but a second computer on line. The last system is a number controlled machine which requires a half hour or more to load the primary program paper tapes. As in any business, these units are there becuse time is money, big money, and the several thousand extra dollars for the backup is usually less than five percent of the overall price (quite often less than one percent). The expense and size of the system is based on the design and the components used. The big expense is in the very high quality batteries, which are usually some form of GEL-CEL type unit. These batteries will take nearly a hundred complete discharges over about a two to three year period before replacement is needed. The gelatin-like electrolytic eliminates the need to replace any lost water or worry about the acids eating away the wiring.

The electronics on a cheap unit can cause noise, or worse yet, still glitch the system when the power fails. The better units can be switched back and forth between the AC lines and batteries without any deterioration of the AC quality. What must be watched for is the type of output from the supply—it should be near a true 60 cycle sine wave. The cheap units will output square waves that can have too high a duty cycle for some switcher power supplies. The design is such that you charge the batteries from the line, and run an inverter from the batteries. Those inverters and switch-over relays are where the problems start. Should you really want to understand the inner design aspects of UIPs, contact one of the manufacturers of the bigger types (GOULD) as they will sometimes have a tutorial type sales pitch that will answer all your questions. Should you decide you must have one, build it yourself as part of your power supply. By this I mean design it so that the batteries are across the main output caps (or in place of them) so that your regular boat anchor is also the battery charger. Very small batteries will do, as you should only need them to save to disk. The power should then be turned off as quickly as possible (about 3 to 5 minutes), which is not what the advertisers show their people doing.

Buses

In S-100 as in other products, a bus is used to communicate between different cards or parts of the system. One subject which often leads to controversial discussions is that of terminators. In the new IEEE standard, terminators are generally specified. This is done to insure that a signal will need to be pulled up by some defined source. This is to say that the specified signal should be pulled high only at a certain given location. Doing so at other locations will produce multiple pullups and may hurt the signal. Termination, however, also involves the use of active voltages on most all signal paths to prevent ringing and help reduce other forms of noise. It has been my experience that using termination as a standard rule of thumb is not a good practice.

One of the shortcomings of the S-100 system is the rather arbitrary assignment of the signals. These assignments are not always the best choices for minimizing noise. Because of this, many systems have had considerable troubles, more often those with large, 20 or more slotted buses. These long signal paths have caused a lot of the headaches. Most good buses now have ground paths between each line to help solve the problem. The use of pullup and pulldown resistors also improved some of the problems. While working on a single board system at Micropro, I had a chance to see first hand what termination could and could not do. The first design had a considerable amount of ringing in the address lines, and required both series and shunt resistors to bring the noise to usable levels. The second design of the board, with different length signal lines, required no resistors at all. In my own work, I have a removable terminator board, and will look at my signals with a scope and try the board. In each test that I have made (and I have made many!) the signals were far worse with the terminator than without it.

When designing a system, always check the signals with a scope and be sure they need terminating or pullups or pulldowns. When extending the address system to contain more than 16 address lines, the use of pulldown resistors will be necessary. Outputs tied to devices that can tristate will need some form of resistor. This resistor will keep the input of a device from floating above the on state. This on state varies from device to device but is usually considered to be about 0.8V. Another way of looking at the problem is to consider that any signal between the range of 0.8 and 3.2V will take anybody's guess as to what state the device will assume. A zero signal is 0V and a one is 4.0V—anything else can cause problems. In reality, there is room for variation but my point is to watch your signal lines; too much variation from the normal will cause problems.

Hacking and Cutting

While talking about signals, I should refresh your memories about the PDBIN problem with Z80 processors. PDBIN should remain low until PSYNC goes low, which is the 8080 norm. The Z80 system will not run if the memory boards do their refresh during this time. To correct this I usually AND the two signals so that an output is not possible unless PDBIN is low. Physically, this involves finding an unused AND gate and doing some hacking and cutting.

When I decide that a signal needs to be rerouted or changed, I have no fears about doing it. Most users consider their boards to be sacred, but not I. My boards may have as many wires running around the board as devices. Typical changes are to upgrade the memory or change an address of the board. Wherever possible I use existing traces, but adding wires generally does not increase the noise on the board. The biggest problem I have with modifications is that the wiring often gets caught when I remove or insert the board. The use of glues to hold wires down can help the loose wire problem, but I may move the wire several times before I am happy with the results.

When starting the project, I find that making a list of spare devices is absolutely necessary. Finding the normal signals which you will need (PSYNC, PDBIN, I/O), and checking them with a scope first, will help familiarize you with the devices before getting started. The most important first step is to check the card in the normal operating mode to see if it works before changing it (as in 16K memory upgrades to 64K—does it work as 16K?). In several of my memory upgrades, I forgot to check them first and had to find troubles after the upgrade. Unless you have checked the card first, any problems you run into afterwards may get you questioning your changes.

Checking the board over many times and using a light to try and see under chip sockets will help find the most appropriate places to change. If you are careful, you can remove the plastic part of the socket. This is possible for socket types that do not have solid bottoms or solder shields. I have done this several times both for changes and to find out where signals went. When I had to remove about 25 sockets (⅔ of the board), I did mess one of them up, which was easy to change with a solder sucker. When replacing sockets or chips, first decide that it needs to be replaced and thrown away. This clears the mind for the next step, which is to cut the device out. For connectors or large sockets, the objective is to mash it in some way so that the individual pins are all that is left. Removing the individual pins can now be done quickly and cooly, with little damage to the board. I have seen too many people try to get the device out in one piece, and in so doing practically destroy the board. A good soldering iron and a plunger type solder sucker seems to work best for me. Many people now say that solderwick type products are needed for CMOS devices because otherwise the static charge caused by the suction will destroy the devices. I have tried solderwick and found it to be far inferior to my large sucker. I have found that small suckers will bounce more and are harder to use, even though they fit nicely in small tool boxes.

For cutting the traces, use a small, sharp EXACTO knife. These sharp cutting tools work best, as they will keep you from using too much force and slipping, cutting both yourself and the surrounding traces. Usually two small cuts will be enough to open up a circuit. Be sure to make it as obvious as possible so that you can find it later, when you've forgotten what you did. Actually, you should document all changes on separate sheets of paper and on the main schematic. As in other parts of computing, documentation is most important. For completing the new circuit I use wire wrapping wire—this small, cheap wire is ideal for adding new signal paths. Should you be changing power signals, use slightly larger wire as this thin wire will not handle much current before the voltage drop becomes detrimental. My usual technique is to get a small bead of solder to flow around the pin and then stick the end of my wire in it. This technique is best, as it keeps me from getting too much solder on the pin. Remember that a little solder can go a long way. Use very small rosin core solder, not that which comes with propane torches and is intended for mending car bodies. Really, the finer a solder you can get, the easier it will be to control the flow. Experiment to see how little you can use before the wire comes off when pulled. I think most people will be surprised at just how little solder is really needed to provide the proper connection.

Logic

When trying to understand the output of a complex logic circuit, I usually try to draw it out. However, you will discover that rather large or complex diagrams can become almost impossible to follow, and just plain old trying it often works the best. Make the jumpers as best you can and check your results with a scope. This approach works well for simple jobs, and for those complex ones, just break the circuit down into smaller units. For socketed boards, bending the chip pin out and then reinserting the chip is used in place of cutting a trace. The pins left sticking straight out are quite easy to solder to, and can be wiped clean later to remove all traces of your experiments. And experiment I do—it is a vital part of understanding anything. I cannot stress that point enough, as so much of the circuit design cannot be explained except through experimentation.

For the real sticklers, a digitial analyzer is necessary. I have a design waiting to be assembled that will turn my S-100 system into an analyzer. Several people make them now. You could also rent an analyzer if needed (about $100 to $400 a month) for those really difficult problems. The analyzer will give you a picture of 8 or 16 or 32 data signals all at the same time. With such a device it is possible to see the true timing relationships between signals. By this I mean whether or not one signal is lasting a few nano-seconds longer than it should. These nano-seconds can actually be where many of the problems come up, and no form of scope work will ever give you that kind of resolution. This resolution is obtained by taking a timing snap shoot of maybe 500 nano-seconds. The analyser will make it possible to see graphically how several signals interact. You may discover that one device's signals are not cut off fast enough once every tenth operation—this is the kind of problem these devices were made to solve. The price of new units is coming down and it will not be long before all good service shops have one.

Last Word of Advice

The last word of advice is one that I have said before and which I will say again—DOCUMENTATION!!! Documenting yourhardware (and software) changes is an absolute necessity. Several of the articles that I have written are actually the documentation from the modifications I made. You never know who will eventually become the owner of your creative work. My systems are now in the hands of people I only see at club meetings, and if I had not documented all the changes properly, those system owners would have mugged me long before this. I consider my hide a most valuable asset, and documentation is the easiest way I know to save it. My previous article on documentation covers the topic quite well. To add to that is almost impossible, except to say that hardware will require lots of schematic drawings. Provide layouts of components before and afterward. Show the schematic before and afterwards. Make sure the documentation gives enough information to explain why it was necessary to make the changes, and what options are now availble. One option should always be that of returning it to the original state. If this is no longer possible, WHY NOT?

The last plug for good documentation is your own work. I have yet to meet anyone that can remember all the fine details of why this is that way one or two years down the road. There is still one of my systems that from time to time I get to work on. I've made several changes over a period of four years, mainly in software. Four years and six other systems can make it quite hard to remember what I did. My documentation, however, has made it possible for both myself and the owner to make many changes after a short review of what went on last time. The new owner also keeps the documetation up to date, mainly becuse I started doing it right the first time. To sum it up, nothing is fixed in stone. Accurate documentation will allow you to see the many changes that have occurred over the years and react accordingly.

Conclusion

What I have covered here are some tricks and indepth comments on how I personally build systems. I didn't give away all my secrets, but I hope I gave enough information to get you interested in doing some of your own work. Myself and the people at *The Computer Journal* want you to realize that these machines are not the untouchables some people want you to believe. However, caution and care, along with some prudent research on your part, are needed when attempting some of these projects. My advice is based on first hand experience, and although your personal experience will be different, my hope is that what I have said will keep you from having anything but a wonderful learning experience. ■

Editor's Page, continued

elemental and boring for others. Our goal is to present a balance of low, medium, and high level articles which will help the inexperienced get started without boring the experts. Even the experts in one area may be inexperienced in another area.

We will continue to publish material on hardware and software hacking in order to establish a foundation of information needed to implement new ideas, and with the demise of *Microsystems* we will add more coverage of the S-100 systems. We are not as language intensive as some publications, but we will concentrate on specialized languages, utilities, and programming techniques suitable for our area while leaving most general business type programming to others.

Database management is one type of program which is usually considered a business program, but which is also needed for experimental data handling. Most database programs are written for business office use (I'm not even satisfied with anything I've seen for business use) but they fail miserably when applied to experimental data. That's why we are presenting the series by E.G. Brooner on writing your own database program. The initial installments use a mail list for examples because it is easy for everyone to understand, but it is our intention to show how to develop a specialized program to do what the high-priced programs fail to do. We need your feedback, and are including a request elsewhere asking about your database needs which are not being met by existing programs.

The shake out in the microcomputer hardware, software, and publishing fields has just started, and it will get worse before it gets better, but there are tremendous opportunities for those who get in on the ground floor of real world applications. ■

COMPUTER® TRADER MAGAZINE

★ ★ ★ LIMITED TIME OFFER ★ ★ ★

BAKER'S DOZEN SPECIAL!

$12.00 for 13 Issues

Regular Subscription $15.00 Year

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

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

Classified Ads for Computer & Ham Radio Equipment

FREE CLASSIFIED ADS
for subscribers

Excellent Display and Classified Ad Rates
Full National Coverage

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

Controlling the Hayes Micromodem II from Assembly Language

by Jan Eugenides

In this article and the one that follows, we will be examining the Hayes Micromodem® II (or IIe), learning how to access and control it from assembly language, and in the process, writing a complete terminal program. Our program will have a large capture buffer (35K!), and will enable us to download text and programs from bulletin board systems (BBS) and other computers, and upload text and programs directly from floppy disk. This will allow uploading of programs of any length, as long as they will fit on a floppy!

You will need an Apple®][,][+, or][e, one disk drive, DOS 3.3, and a Hayes MicromodemII or IIe. An assembler of some kind would be very helpful. Source code in this article will be presented in the S-C Assembler format. For those without assemblers, I will give you a complete hex dump of the program, so you can just type it in from the monitor if you choose. This will allow you to use it as is, although modifying it would be difficult.

The program will be presented in sections, for the sake of clarity. As you examine Listing 1, you will notice some lines missing. These will be filled in later, in part two of this article. This time, we'll examine the dialing routine, the main program loop (incoming data and keyboard output), the command handler, and toggling the capture buffer.

I'm assuming a reasonable knowledge of assembly language in this article. If you find that it is over your head, I suggest the following references:

- **Assembly Lines: The Book** by Roger Wagner, pub. by Softalk Books.†
- **Assembly Cookbook** by Don Lancaster, pub by Howard Sams.
- *Apple Assembly Lines*, published monthly by S-C Software Corp.

I also write a monthly beginners' column for *Scarlett*, the Big Red Apple Club newsletter, so that might help you out.

Dialing the Phone

Before we can do anything, we must dial the phone, right? At the beginning of Listing 1, in line 1940, you'll see a JSR DIALUP. After program initialization (which we'll cover next time) this is the first routine. The routine itself is in lines 5710-6390. First, we clear the screen with a JSR HOME. This is the monitor routine commonly used, at $FC58. Then we tab down to line 5, using another monitor routine, VTAB. Then we JSR to our own PRINT routine (we'll examine it in detail next time) which grabs the ASCII in line 5760 and puts it on the screen. Then we JSR to our INPUT routine (also, next time) and get the phone number from the user. The number is stored in memory, and IPTR is left pointing to the address. Now we come to the heart of the dialup routine.

In order to pass commands to DOS, it is necessary to output a CTRL-D followed by the command at the beginning of a line. From assembly, this is easily accomplished by using the COUT routine built into the Apple monitor. Our print routine uses COUT, so that makes things easy. In lines 6250-6280 we use our print routine to print

◄CR► "CTRL-D PR#2" ◄CR►

(the ◄CR► means carriage return, $8D), which tells DOS to turn on slot 2 for output. DOS takes care of the details. Then we print "CTRL-Q", which is now output to the modem, and tells it to dial the upcoming number. These are simply the same commands you would issue from the keyboard if you were doing it manually. In my Hayes manual, this information is on page 12. Then in lines 6300-6340 we get the phone number from where we stored it in memory, and output it to the modem, which obligingly picks up the phone and dials the number. Easy, huh? Then we print a ◄CR► "CTRL-D PR#0" ◄CR► to turn the modem off for output (this does not hang up the phone!).

The program returns now to line 1950, where we JSR CDETECT to see if a carrier is present. If not, we start over. If we have a carrier, we continue to line 1960, where we enable the modem for input with

◄CR► "CTRL-D IN#2" ◄CR►.

Let's look at the carrier detect routine, in lines 7310-7370. Here we use one of the modem registers, CR1. Bit 3 is the "No carrier detect" flag, in other words, if bit 3 is set, no carrier is present. So, if we load the accumulator with CR1, and then AND it with #$04, the result will be zero unless bit 3 is set, in which case the BNE in line 7350 will branch over the RTS in 7360, and JMP to the start of the program. What are modem registers, you ask? Read on.

Modem Registers

Controlling the MMII is comparatively easy. There are two registers we need to be concerned with, called CR1 and CR2. CR1 is the status register, and gives us information on the incoming data, as well as various error conditions. For our purposes, it is only necessary to understand that if bit one of CR1 is on, then a character is ready in the input port, which is at CR1 + 1. CR1 is also used to detect the carrier, as we noticed above. CR2 is the modem control port, and controls whether the modem is on line or off line, and can be used to set the mode, turn on the transmitter, etc. We use CR2 to hang up the phone, in lines 3580-3610, by simply storing a zero there. Fortunately for us, the Micromodem firmware takes care of the rest. The actual memory locations for these registers are slot-dependent. I will present the program to you configured for slot 2, and I'll give you a configuration program which will allow you to modify it for any slot you choose (and also to select upper case only, or upper and lower case... see "A Note on Lower Case" below). For now, let's concentrate on CR1.

Incoming Data

In line 2040, the main loop begins with a JSR to a routine which will display the address of the top of the capture buffer as it fills with data. We'll look at that routine next time. Next, we check CR1 to see if a character has been received. The ROR in line 2070 puts bit one in the carry, thus if the carry is clear, we know there has been no character received, and we branch to the keyboard input routine. If the carry is set, a character has been received, and we branch to the character input routine. The program goes around and around in this loop, checking CR1 for input, checking the keyboard for output, and handling each character accordingly.

The character input routine (CHARIN) is in lines 2590-2890. First, we collect the character from the input port which is located at CR1+1. For the sake of the Apple, we then set the high bit. Next we check for linefeeds, and discard any we find. Otherwise all our lines would be double-spaced. In line 2640 we check the capture flag. This flag will be set by the "toggle capture" part of the program, and indicates whether or not we want to capture the incoming data in the Apple's memory. If the flag is zero, we want to capture it, and the program branches accordingly to the STORE routine in lines 2730-2890. Otherwise, we simply display the character on the screen in lines 2660-2720. Notice in lines 2660-2700 we are checking for lower case characters, which will be greater than $E0 (remember the high bit is set). If lower case is found, it is converted to upper case by subtracting $20. Then the Apple COUT routine is called to output the character to the screen. Next is the STORE routine, which simply stores the character in the memory location pointed to by PTR and PTR+1 in the Apple zero page. Next, we go through the lower case checking routine again, and display the character on the screen as before. Then, in lines 2820-2850 we increment PTR in preparation for the next incoming character. Thus, all characters are simply stored sequentially in memory. In our final program, the buffer will extend from $800 to $9000, 35K of space in a 48K Apple, and 10K is for DOS! See why I like assembly? If you have a 64K Apple, you could get snazzy and use the extra RAM to create an even bigger buffer, up to 47K or so. Or if you have a 128K Apple... I leave it up to you.

Finally in lines 2860-2890, we check to see if the buffer is full, and if so, branch to a routine which prints a warning and stops the incoming data. Otherwise we simply return to the main loop. The buffer-full routine is in lines 2930-3020, and works as follows:

First we put a $3F at INVFLG. This results in inverse characters being printed to the screen. Then we print "BUFFER FULL" in inverse (clever!) and set INVFLG to $FF, which is back to normal. INVFLG actually is a mask value, which is ANDed with the character before it is displayed on the screen. Thus the $FF results in no change, while $3F strips off the two highest bits (Can you guess what would happen if we used $7F to strip off just the high bit? Right, flashing characters!). Next, we execute the subroutine XOFF, which sends a CTRL-S out through the modem. Most hosts recognize this character as a signal to stop sending data. Thus, everything stops with "BUFFER FULL" displayed on the screen, and the program waits for you to do something about it. In the complete program, you'll be able to either save the buffer to disk, or just zero it out.

Apple][
Monitor Subroutines and System Calls

One of the nice things about programming in assembly language on the Apple][is the fact that there are many useful subroutines in the monitor ROM which can be called from the assembly language program. An example is the familiar BASIC command HOME which can be called from assembly with a JSR $FC58. Some of the common subroutines are listed below to help BASIC programmers understand the assembly listings. Let us know if you would like to see articles on beginning Apple][assembly language.

$FC58 HOME Clears the screen and positions the cursor at the upper left hand corner of the screen.

$FDED COUT The normal character output subroutine. It calls the output subroutine whose address is stored in locations $36 and $37 (usually COUT1). The character to be output should be in the accumulator.

$FD1B KEYIN Reads the APPLE keyboard, and waits for a keypress. When a keypress is received, it removes the flashing cursor, and returns with the keycode in the accumulator.

[illegible] location [illegible] high bit [illegible] to be [illegible] indicates [illegible] accept [illegible] location [illegible] can be [illegible] the value [illegible] read [illegible] The [illegible] was [illegible] after [illegible]

A Note About Lower Case

As you no doubt noticed in the previous code, all lower case letters were converted to upper case for display on the Apple screen. This is to make Big Buff compatible with regular Apple]['s and][+'s. If you have a][e, or if you have a lower case chip installed, these routines are unnecessary. In my efforts to make this program useful to the most people, I decided the best way to handle it was to write the program so that it always converts lower case to upper case. Then, a simple configuration program can be written which will modify the program to use lower case on those machines which can handle it, and incidentally, modify it for a MicroModem in any slot. I'll give you the configuration program next time.

Sending From the Keyboard

The keyboard input routine is in lines 2140-2580. For upper case only, we simply skip over all the conversion stuff. However, for Apple]['s or][+ 's with the popular shift key modification, we can do some conversion. This modification simply connects a wire from the shift key to one pin of the game port. This pin can be read from assembly language at location $C063. If the value is less than $80, the shift key is being pressed. With this information and a little knowledge of ASCII, it is quite easy to write an upper-lower case routine. This routine is presented in Listing 2. The configuration program will poke this into the proper locations for those users who have lower case capabilities. Notice that in lines 2160-2180 of Listing 1 we have left space for this purpose.

Table 1 is the Apple ASCII Character set. It differs from regular ASCII in that all the high bits are set. Thus a capital A, which would normally be $41, becomes $C1. This

Hex	$8Ø	$9Ø	$AØ	$BØ	$CØ	$DØ	$EØ	$FØ
$Ø-	nul	dle		Ø	@	P		p
$1-	soh	dc1	!	1	A	Q	a	q
$2-	stx	dc2	"	2	B	R	b	r
$3-	etx	dc3	#	3	C	S	c	s
$4-	eot	dc4	$	4	D	T	d	t
$5-	enq	nak	%	5	E	U	e	u
$6-	ack	syn	&	6	F	V	f	v
$7-	bel	etb	'	7	G	W	g	w
$8-	bs	can	(	8	H	X	h	x
$9-	ht	em	)	9	I	Y	i	y
$A-	lf	sub	*	:	J	Z	j	z
$B-	vt	esc	+	;	K	[	k	{
$C-	ff	fs	,	<	L	\	l	|
$D-	cr	gs	-	=	M	]	m	}
$E-	so	rs	.	>	N	^	n	~
$F-	si	us	/	?	O	_	o	rub

Table 1: The ASCII Character Set.

arrangement is due to the fact that the Apple displays characters in three ways: normal, inverse, and flashing. Regular ASCII appears as flashing characters on the Apple screen (Remember, we discovered this in the "Buffer Full" routine, above). In Big Buff, I have chosen to set all the high bits on incoming characters, and then deal with them thereafter as "high" ASCII.

Looking at Listing 2 now, in lines 1130-1150, we check for a keypress. If there is one, we go ahead and get the character, otherwise we just jump back to the main loop. In lines 1160-1190, we first clear the keyboard strobe, check to see if the shift key is pressed, and then branch accordingly. Lines 1200-1250 check for the three special cases on the Apple keyboard: shift-M, shift-N, and shift-P, which normally output], ^, and @ respectively. Since we want normal capital letters, these three must be converted. The rest of the capitals come out normally.

In lines 1270-1330, the lower case conversion takes place. Since we only want to convert letters, we first check to make sure the character is between A and Z. If it is, we add $20 to it, which makes it lower case.

In lines 1340-1390, the three special cases we mentioned earlier are converted. Finally, in lines 1400-1450, we check for a back-arrow. If we have one, we call a monitor routine at $FC10, which obligingly backspaces for us, and we then output a blank to clear the screen of the unwanted character. Big Buff uses combinations of the ◄ESC► key and one other key for all its commands. This keeps all commands on line. In lines 1460-1490 we check for an ◄ESC► key, and if it has been pressed, we branch to the command handler routine. Otherwise, we store the character at location $778, called CHAR.

Included in the MMII firmware is a routine called OUTA. Like the registers, its location is slot-dependent, and we'll go into that along with the configuration program later. OUTA takes whatever character is in $778 and outputs it through the modem. All we have to do, then, is put our character at $778, and JSR OUTA. We will also use OUTA in the uploading routine. Finally in line 1500, we return to the main loop.

The Command Handler

Next, we come to the command handler, in lines 3210-3400. As we noted before, control branches here if ◄ESC► is pressed. First, the XOFF is sent to stop incoming data. Then we check the next key pressed. Big Buff has seven commands, as follows:

◄Esc► Z	Zeroes the buffer
◄Esc► X	Hangs up the phone
◄Esc► S	Saves the buffer to disk
◄Esc► B	Toggles the buffer on or off
◄Esc► U	Uploads files from disk
◄Esc► C	Catalogs the disk
◄Esc► R	Reviews the buffer

Thus, we simply check for each of these keys, and branch accordingly. If the key pressed is none of these, the XON is sent to start incoming data again, and we return to the main loop. By the way, if you don't like the letters I chose, just plug in letters you like better. I will go into how to use each command next time, when I describe how to use Big Buff.

The Capture Buffer

In the initialization part of the program, PTR is set up to point to $800, the beginning of the capture buffer. Then, as we saw above, as incoming data is stored in the buffer, PTR is incremented so that it always points to the next available memory location. Then, all we need is LDY #0 : STA (PTR),Y to store the incoming character. In lines 3480-3540, the zero buffer routine simply resets PTR to $800, effectively "forgetting" all the data.

ESC-B takes us to line 3660, which toggles the capture buffer on and off. First, we check CAPFLAG. If it does not contain $FF, capture is on, so we go turn it off in lines 3760-3810 by simply storing $FF there. This value is checked by the incoming data routine, as we saw before, and if $00 is found there, the character is stored. In lines 3690-3750, the opposite condition is true, so we turn on the capture by putting a $00 there. Notice in lines 3860-4030, two short routines to first save the current cursor position, then move to the top of the screen, where we print "OFF" or "ON" depending on what's happening, and then to restore the cursor to its original location. This is so the user can easily tell the status of the capture buffer.

You may have noticed also that each routine ends with JSR XON : JMP MLOOP. The X-ON character is CTRL-Q, which tells the host to begin sending data again. Remember, when ESC was pressed, we sent the X-OFF, so before returning the the main loop, we have to send X-ON.

Next Time

Next time, we'll examine the disk access routines and develop a way to save the buffer to disk, and to upload files directly from disk. We'll also add the initialization, a way to review the buffer, input and print routines, and wrap the whole thing up. I'll give you a complete listing, both source and hex, and describe how to use the program. I will also be happy to supply you with the program and complete source code on disk for $10.00, to save you some typing. Send it to me at 1543 N.E. 123 St., N. Miami, FL 33161 ■

Listing 1

```
  Initialization goes here...

1910 *---------------------------------
1920 *Get number to call
1930 *---------------------------------
1940 START  JSR DIALUP
1950        JSR CDETECT     Check for carrier
1960        JSR PRINT
1970        .HS 8D84
1980        .AS -"IN#2"     Turn on modem for input
1990        .HS 8D00
2000 *------------------------------
2010 *Main program loop
2020 *------------------------------
2030 *
2040 MLOOP  JSR SHOWBUF     Show buffer filling up
2050        LDA CR1         Modem register
2070        ROR             Test for bit one on
2080        BCC KEYIN       If not, no character at input
port
2090        JSR CHARIN      Get character from input port
2100 *---------------------------------
2110 * Lowercase keyboard input
2120 *---------------------------------
2130 *
2140 KEYIN  LDA $C000       Keyboard input
2150        BPL MLOOP       No key has been pressed
2160        NOP             Leave space
2170        NOP             for lowercase
2180        NOP             Routine
2190 GOTONE BIT $C010       Clear strobe
2200        JMP OUTPUT      Skip lowercase routine -
configure program pokes
2210 *                      new code in here for lowercase
action
2220 *---------------------------------
2230 *Lowercase routine (first part
2240 *gets poked in by configure program
2250 *---------------------------------
2260        CPX #$80        Pressed?
2270        BCS LOWER       No, so lower case
2280        CMP #$DD
2290        BEQ CAP         Capital M
2300        CMP #$DE
2310        BEQ CAP         Capital N
2320        CMP #$C0
2330        BEQ CAPP        Capital P
2340 OUTPUT JMP BACK
2350 LOWER  CMP #$C0        Set range from A-Z only
2360        BCC OUTPUT      <A, no conversion
2370        CMP #$DB
2380        BCS OUTPUT      >Z, no conversion
2390        CLC
2400        ADC #$20         Convert to lower case
2410        JMP OUTPUT
2420 CAP    SEC              Special cases "N" AND "M"
2430        SBC #$10
2440        JMP OUTPUT
2450 CAPP   CLC              Special case "P"
2460        ADC #$10
2470        JMP OUTPUT
2480 BACK   CMP #$88
2490        BNE ESC
2500        JSR $FC10
2510        LDA #$A0
2520        JSR COUT
2530        LDA #$88
2540 ESC    CMP #$9B        Escape?
2550        BEQ ESCAPE      Yes-go to command handler
routine
2560        STA CHAR        Set up to output char thru
modem
2570        JSR OUTA        Output character
2580        JMP MLOOP
2590 CHARIN LDA CR1+1       Get char from input port
2600        ORA #$80        Set high bit
2610        CMP #$8A        Linefeed?
2620        BEQ SKIP        Skip linefeeds
2630        STA TEMP        Save character
2640        LDA CAPFLAG     Check capture status on or off

2650        BEQ STORE
2660        LDA TEMP        Retrieve character
2670        CMP #$E0        Lowercase?
2680        BCC DISP        No-display on screen
2690        SEC
2700        SBC #$20        Convert to upper
2710 DISP   JSR COUT        Display
2720        RTS
2730 STORE  LDA TEMP        Retrieve character
2740        LDY #$00
2750        STA (PTR),Y     Save character in buffer
2760        LDA TEMP
2770        CMP #$E0        Lowercase?
2780        BCC CHARTN      No-display on screen
2790        SEC
2800        SBC #$20        Convert to upper case
2810 CHARTN JSR COUT        Print to screen
2820        INC PTR         Increment buffer counter
2830        LDA PTR
2840        BNE SKIP
2850        INC PTR+1
2860 SKIP   LDA PTR+1
2870        CMP #$90        Buffer full?
2860        BCS FULL        Yes-print warning
2890        RTS             Done
2900 *-----------------------------
2910 *Full buffer routine
2920 *-----------------------------
2930 *
2940 FULL   LDA #$3F        Inverse
2950        STA INVFLG
2960        JSR PRINT
2970        .AS -"BUFFER FULL"
2980        .HS 8D00
2990        LDA #$FF        Normal
3000        STA INVFLG
3010        JSR XOFF        Stop incoming data
3020        RTS
3030 *-----------------------------
3040 *SEND X-OFF
3050 *-----------------------------
3060 XOFF   LDA #$93     Ctrl-S
3070        STA CHAR
3080        JSR OUTA     Send X-OFF
3090        RTS
3100 *---------------------------------
3110 *SEND X-ON
3120 *---------------------------------
3130 XON    LDA #$91     Ctrl-Q
3140        STA CHAR
3150        JSR OUTA     Send X-ON
3160        RTS
3170 *-----------------------------
3180 *Escape command handler
3190 *-----------------------------
3200 *
3210 ESCAPE JSR XOFF
3220        LDA $C000    Get keyboard input
3230        BPL ESCAPE
3240        BIT $C010
3250        CMP #$DA     "Z"
3260        BEQ ZBUFF    Zero buffer
3270        CMP #$D8     "X"
3280        BEQ HANGUP   Hangup phone
3290        CMP #$D3     "S"
3300        BEQ SAVE     Save buffer
3310        CMP #$C2     "B"
3320        BEQ BTOGL    Toggle capture
3330        CMP #$D5     "U"
3340        BEQ UPLOAD   Upload files
```

Listing 1, continued

```
3350          CMP #$C3          "C"
3360          BEQ CAT           Catalog disk
3370          CMP #$D2          "R"
3380          BEQ REV           Review buffer
3390          JSR XON
3400          JMP MLOOP
3410 UPLOAD   JMP READ
3420 CAT      JMP CATALOG
3430 REV      JMP REVIEW
3440 SAVE     JUMP SAVEBUF
3450 *-------------------------------
3460 *Zero buffer-reset pointer to $800
3470 *-------------------------------
3480 *
3490 ZBUFF    LDA #$00
3500          STA PTR
3510          LDA #$08
3520          STA PTR+1
3530          JSR XON
3540          JMP MLOOP
3550 *-------------------------------
3560 *Hangup phone
3570 *-------------------------------
3580 HANGUP   LDA #$80
3590          STA $D8
  Ø0          LDA #$00
              STA CR2        Hang up modem
              JMP SAVEBUF    Last chance!
        ---------------------------
        oggle capture on and off
     *-----------------------------
3650 *
3660 BTOGL    LDA CAPFLAG       Get flag
3670          CMP #$FF          Off?
3680          BNE CAPOFF        No-turn it off
3690          LDA #$00          Turn it on
3700          JSR CURS          Save cursor position
3710          JSR PRINT         Change indicator
3720          .AS -" ON"
3730          .HS 00
3740          JSR RCURS         Restore cursor position
3750          JMP CONT1
3760 CAPOFF   LDA #$FF          Turn it off
3770          JSR CURS          Save cursor position
3780          JSR PRINT         Change indicator to "OFF"
3790          .AS -"OFF"
3800          .HS 00
3810          JSR RCURS         Restore cursor position
3820 CONT1    JSR XON
3830          JMP MLOOP
3840 *
3850 *-------------------------------
3860 CURS     STA CAPFLAG
3870          LDA CV            Vertical cursor position
3880          STA VTEMP
3890          LDA CH            Horizontal cursor position
3900          STA HTEMP
3910          LDA #$02          VTAB 2
3920          STA CV
3930          JSR VTAB
3940          LDA #13           HTAB 13
3950          STA CH
3960          RTS
3970 *---------------------------------
3980 RCURS    LDA HTEMP
3990          STA CH
4000          LDA VTEMP
4010          STA CV
4020          JSR VTAB
4030          RTS
```

Other stuff goes here...

```
5660 *--------------------------------
5670 *
5680 *Get number and dial phone      *
5690 *--------------------------------
5700 *
5710 DIALUP JSR HOME
5720 P1     LDA #$05
5730        STA CV         VTAB 5
5740        JSR VTAB
5750        JSR PRINT
5760        .AS -"DIAL WHAT NUMBER?"
5770        .HS 00
5780        JSR INPUT      GET NUMBER
5790        JSR PHNUM
5800        RTS
5810
```

Other stuff goes here...

```
6220 *------------------------------
6230 *DIAL PHONE
6240 *------------------------------
6250 PHNUM JSR PRINT            Print CTRL-D"PR#2"
6260       .HS 8D8D84
6270          .AS -"PR#2"       Turn on modem for output
6280          .HS 8D9100        Print ctrl-Q
6290          LDY #0            Print phone number
6300 NUMOUT LDA (IPTR),Y
6310          BEQ DONE
6320          JSR COUT
6330          INY
6340          JMP NUMOUT
6350 DONE     JSR PRINT         Print CTRLD"PR#0"
6360          .HS 8D8D84
6370          .AS -"PR#0"       Turn off modem for output
6380          .HS 8D00
6390          RTS
```

Other stuff goes here...

```
7310 *------------------------------
7320 CDETECT
7330          LDA CR1
7340          AND #4      check bit 4
7350          BNE NOCAR
7360          RTS         Carrier detected, resume
7370 NOCAR    JMP START No carrier, start over
```

Listing 2

```
1000 *-----------------------------
1010 *Main program loop
1020 *-----------------------------
1030 *
1040 MLOOP    JSR SHOWBUF      Show buffer filling up
1050          LDA CR1          Modem register
1060          ROR              Test for bit one on
1070          BCC KEYIN        If not, no character at input
port
1080          JSR CHARIN       Get character from input port
1090 *---------------------------------
1100 * Lowercase keyboard input
1110 *---------------------------------
1120 *
1130 KEYIN    LDA $C000        Keyboard input
1140          BMI GOTONE       A key has been pressed - go get
it
1150          JMP MLOOP        No key has been pressed
1160 GOTONE BIT $C010          Clear strobe
1170          LDX $C063        Game port
1180          CPX #$80         Shift key pressed?
1190          BCS LOWER        No, so lower case
1200          CMP #$DD
1210          BEQ CAP          Capital M
1220          CMP #$DE
1230          BEQ CAP          Capital N
1240          CMP #$C0
1250          BEQ CAPP         Capital P
1260 OUTPUT JMP BACK
1270 LOWER    CMP #$C0         Set range from A-Z only
1280          BCC OUTPUT       <A, no conversion
1290          CMP #$DB
1300          BCS OUTPUT       >Z, no conversion
1310          CLC
1320          ADC #$20          Convert to lower case
1330          JMP OUTPUT
1340 CAP      SEC               Special cases "N" AND "M"
1350          SBC #$10
1360          JMP OUTPUT
1370 CAPP     CLC               Special case "P"
1380          ADC #$10
1390          JMP OUTPUT
1400 BACK     CMP #$88          Back arrow?
1410          BNE ESC           No, check for command
1420          JSR $FC10         Backspace one
1430          LDA #$A0          Print a space
1440          JSR COUT
1450          LDA #$88          Restore accumulator
1460 ESC      CMP #$9B          Escape?
1470          BEQ ESCAPE        Yes-go to command handler
routine
1480          STA CHAR          Set up to output char thru
modem
1490          JSR OUTA          Output character
1500          JMP MLOOP
1510 CHARIN LDA CR1+1           Get char from input port
1520          ORA #$80          Set high bit
1530          CMP #$8A          Linefeed?
1540          BEQ SKIP          Skip linefeeds
1550          STA TEMP          Save character
1560          LDA CAPFLAG       Check capture status on or off

1570          BEQ STORE
1580          LDA TEMP          Retrieve character
1590          CMP #$E0          Lowercase?
1600          BCC DISP          No-display on screen
1610          SEC
1620          SBC #$20          Convert to upper
```

continued on page 26

FOR YOUR APPLE IBM, COMMODORE 64 OR CP/M COMPUTER

Problems with your Lover?

Tell ELIZA!

Meet ELIZA,
the computer psychotherapist.

Created at MIT in 1966 to run on a large mainframe, ELIZA has become the world's most celebrated artificial intelligence program. ELIZA will analyze any statement you enter and respond in true Rogerian fashion - and her remarks are often amazingly appropriate!

Unlike the stripped down versions you may have seen, our ELIZA has retained the full power and range of expression of the original mainframe program.

Best of all, ELIZA comes with the complete Source Program (written in BASIC) ... anyone, even a beginner, can easily customize ELIZA's responses.

So next time people ask you what your computer can do, bring out ELIZA and show them!

"Much more than a mere game ... you'll be impressed with Eliza...a convincing demonstration of artificial intelligence" - PC magazine

"Delightful entertainment ... an ideal medium for showing off your system." - Microcomputing magazine

ELIZA is only $45 and available in the following formats:

- Apple II, II plus, IIe, IIc, III
- IBM PC and all compatibles
- Commodore 64 (disk or cassette)
- 8" disk for all CP/M systems
- 5¼" disk for most CP/M systems (specify make and model)

Please add $3.00 for shipping and handling (California residents, add 6½% sales tax.)

Checks, MasterCard, and VISA accepted

Artificial Intelligence RESEARCH GROUP

921 N. La Jolla Ave., Dept. C, Los Angeles, CA 90046 • (213) 656-7368 • (213) 654-2214

S-100 8 TO 16 BIT RAM CONVERSION

by Lance Rose

Background

With the temptation these days to switch over to 16-bit processors, I know that many of us hesitate because of the limitation of our older 8-bit wide RAM boards. While some of the new processor chips are available with an 8-bit external data bus such as the 8088 and the 68008 (see "Build a 68008 CPU Board for the S-100 Bus" in Vol. 2, No. 7 of *The Computer Journal*), some of the speed advantages are lost without the ability to do full 16-bit wide data transfers. There are, of course, a number of 16-bit wide RAM cards marketed for the S-100 bus but in most cases the price of these is pretty steep. In addition, what do you do with your old 8-bit RAM cards?

Well, there is a solution to these problems. If you are fortunate enough to have some 64K RAM boards that lend themselves to this type of conversion, you can use them in pairs to accomplish 16-bit wide data transfers. The actual modification depends, of course, on the exact RAM card you have. In my case it is the Digital Research Computers 64K CMOS static RAM, and this is the board this article is based upon. If you don't have this type of board perhaps you can still glean enough of the philosophy from here to accomplish the conversion for your own particular type of RAM board.

Modifications

To understand what the necessary modifications are to get full 16-bit data transfers, it is first necessary to look briefly at the way data transfers are made on the S-100 bus. When an 8-bit RAM board is selected with address lines A0-A23 (extended addressing) it either presents data for input on the 8 data-in lines DI0-DI7 or accepts data for output on the 8 data-out lines DO0-DO7. The original designers (if the S-100 bus can actually said to have been "designed") unknowingly did us a great favor by separating the data-in lines from the data-out lines. Some other buses use bidirectional data lines and thus don't have the capability for this type of expansion.

The logical thing to do, and what in fact was actually done when creating the IEEE 696 standard, was to gang together the data-in and data-out lines to create a 16-bit wide bidirectional data bus to be used when devices on the bus are capable of 16-bit transfers. In this case, the data-out lines DO0-DO7 are used for the even-addressed data byte and the data-in lines DI0-DI7 are used for the odd-addressed data byte regardless of which way the transfer occurs.

Knowing this, we see what must be done to change a pair of 8-bit wide RAM boards to what appears as a single 16-bit wide RAM board. First of all, it turns out to be easiest to separate even-addressed data from odd-addressed data, using one board for each. This can be accomplished fairly easily by exchanging the A0 and A16 data lines on the board. Secondly, an additional bus driver or receiver must be added to each board to achieve bidirectional capability on either the DI or DO lines, depending on whether the board is even or odd. Thirdly, the board must respond to the sXTRQ* status signal and in turn generate the response signal SIXTN* indicating that the device selected is capable of 16-bit transfers. If a bus master asserts sXTRQ* and the slave device (a RAM board in this case) can't do 16-bit transfers, it doesn't respond with SIXTN* and the master must make the read or write with two single bytes instead of a 16-bit word.

Circuit Modifications

Figure 1 shows the necessary circuit modifications to the above-mentioned RAM boards to make them capable of the 16-bit transfers. All references to IC numbers in this figure correspond to the numbering convention used in the

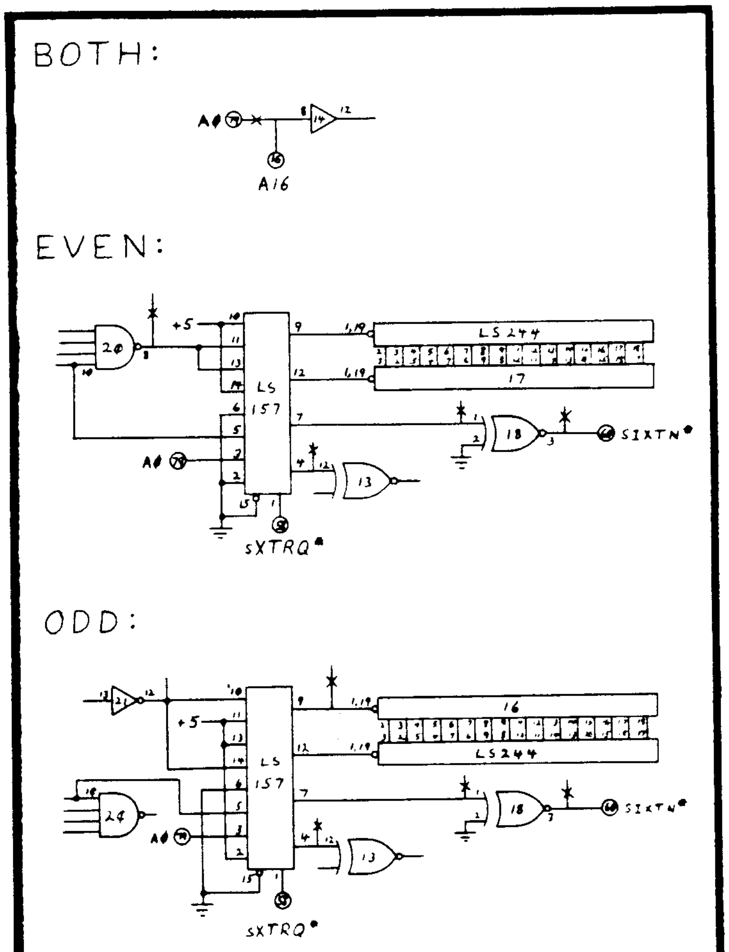


Figure 1

schematic diagram provided by the manufacturer with the board. A16 is moved to the position previously occupied by A0 while the A0 input is fed to one part of the 74LS157 multiplexer. When sXTRQ* is high (not asserted), the A0 input is passed along to what used to be the input of A16 to the board select circuity. This makes one board of the pair respond to even addresses and the other to odd addresses. When a 16-bit operation occurs and sXTRQ* is asserted, a fixed level signal is fed to the select circuitry so that both boards respond at the same time.

Two additional sections of the multiplexer are used to select the proper inputs to use for turning on the bus drivers and receivers. These are slightly different for the even and odd boards. The last part of the multiplexer is used to generate the SIXTN* response signal by borrowing part of the select logic normally used to reject any bus cycle where sINP is true (I/O operations). Since this is already done with sMEMR here, the sINP is superfluous and we can borrow part of the 74LS266 open-collector XOR gate to drive the SIXTN* bus line.

Technique

Figure 2 shows a photograph of one of the modified boards. Each of the two additional chips (a 74LS244 and a 74LS157) is "piggybacked" onto one of the existing chips already on the board. They are supported in mid-air by short pieces of wire-wrap wire which also serve to make the power and ground connections to the additional chips. The connections are made by carefully wrapping the necessary wires around the legs of the IC package using a fine pair of needle nose pliers. While some people have told me they use a wire wrap tool for this, I still prefer the needle nose pliers for better control. The two connections to the bus are made by soldering (using very little solder!) the wires to the upper part of the board fingers. In two other cases (the connections to A0 and A16) the connections are made by inserting one end of the wire into the socket pin normally occupied by an IC leg, said IC leg being previously bent out of the socket. These two connections can also be made by soldering.

Some helpful hints are in order here. First of all, the wires connecting the piggyback chips to the supporting chips should be cut to about an inch and a half long with approximately a half inch stripped from each end. This keeps the new chips up far enough to avoid shorting to the lower IC pins but close enough to prevent interfering with adjacent boards in the bus (at least in the case of every other slot being installed).

Another tip is to pre-cut and connect all the wires to the upper package first. Then, holding both packages in one hand, use the other hand to wrap the wire around the leg of the lower IC. About 4 or 5 turns around the upper legs is about right. With the lower chip, if you are connecting to a pin that is bent out of the socket the same number of turns is fine. If you are connecting to a leg that will remain in the socket, try to limit it to about two turns. Much more and the chip won't seat back in the socket far enough and the connection might not be adequate. My own experience is

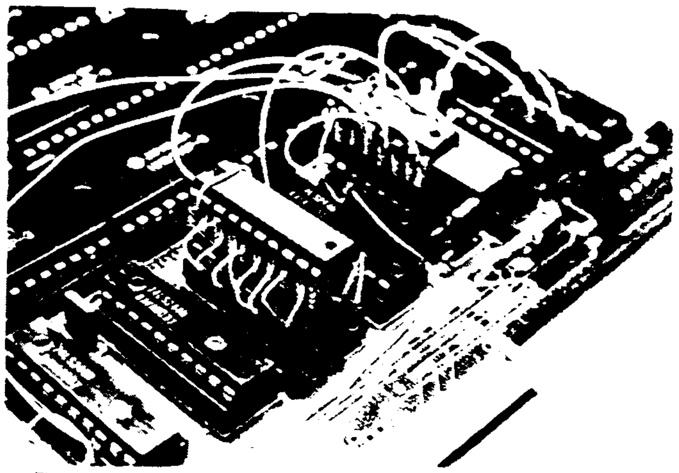


Figure 2: View of installation of additional parts on even board.

that this makes for a completely reliable system if done with a little care. It may lead to some eyestrain, so a few shorter sessions might be better than doing it all at once.

Once all the connections are made, you can try checking the boards out. In order to check them completely you will have to beg, borrow, steal, purchase or build a true 16-bit CPU board. If you're one of the few people using front panels these days, you can make some preliminary tests using that. Successively stepping through addresses with the EXAMINE NEXT switch should cause the select LEDs on the pair of boards being tested to alternate. Briefly(!) shorting sXTRQ* to ground should cause both LEDs to come on at the same time. Do this with both an even and an odd address selected to make sure that both boards in the pair are responding to sXTRQ*.

The next step is to try the boards with your current 8-bit CPU. Since it should keep the sXTRQ* signal high (negated), the boards should operate in their 8-bit mode as before. If not, look for a wiring error before proceeding. Finally, try using a true 16-bit CPU board with the newly-converted RAM cards. Be sure to check and make sure that the SIXTN* line on the CPU board is pulled up to 5 volts with a resistor (the Compupro CPU-68K seems to lack this). The boards should function properly as before except that programs will now run quite a bit faster. How much faster? I tried some short benchmark programs to see. For code that is memory intensive, you should get a speedup of 60 over the 8-bit equivalent. Register intensive code is improved less (around 35) due to the fact that a higher proportion of clock cycles are spent doing internal operations in the microprocessor. As a rule of thumb I would say that a typical program will run 40-50 faster than before, not a bad improvement for about $2 worth of parts per board and an afternoon's work (eyestrain provided at no extra charge).

Although not everyone is using the 64K CMOS RAM card I talk about here, there are many varieties of such cards and others are probably open to similar modifications. The specifics may be different but the principles will be the same. The biggest problem I have since the conversion is how to keep the 16-bit CPU I borrowed a little bit longer. ■

TIME-FREQUENCY DOMAIN ANALYSIS

by Bert P. van den Berg, President, BV Engineering

Time Domain Signals and the Fourier Transform

In engineering analyses, the behavior of systems can be described as a function of time or as a characterization of the frequency content of the output signal. Both methods have advantages and the choice of which method to use depends heavily upon the system itself, whether the system is linear or nonlinear, and what the results are going to be used for. The frequency domain is the most convenient for linear analysis, while the time domain is most suited for nonlinear analysis. The time domain and frequency domain are related through the Fourier transform.

Any time domain signal S(t) can be represented as a mathematical series of sinusoids which, when evaluated as a function of time and summed together, yields the original signal S(t) as shown in equation 1. The Fourier transform of a time domain signal decomposes the signal into these constituent frequency components. Each of these components consists of a sinusoid of a fixed frequency with its associated magnitude and phase.

$$S(t) = \sum_{i=1}^{n} A_i \operatorname{Sin}(2 \Pi F_i t + B_i)$$

The A_i represent the magnitude of the ith component whose frequency is F_i Hertz and whose phase angle is B_i radians. The Direct Current (DC) component of the signal S(t) is obtained when $F_i = 0$ Hertz. The collection of these component frequencies, called the "spectra" of S(t), completely describes the signal S(t). The individual frequencies making up the spectra are called the "spectral components".

Errors and the Fast Fourier Transform

The number of spectral components needed to accurately represent the signal S(t) depends upon the waveshape of the time domain signal. The waveshape and period of the signal determine the frequency content of that signal and a sufficient number of spectral components need to be evaluated in order to prevent errors in the reconstitution of S(t). In turn, the number of spectral components that can be calculated with the Fourier transform depends upon how much we know about the time domain signal.

When using a computer, the time domain signal S(t) is "sampled." Unless we have infinite storage capability we can store only limited information about the signal. We might divide the signal into 100, 200, or even 500 samples, but no matter how many samples we store, we are losing some information. When the signal S(t) contains frequency components higher than one-half the sampling frequency, errors due to "aliasing" appear. Aliasing results in errors in reconstruction of the original time domain waveform due to a lack of sufficient frequency information.

Figure 1 shows a sine wave which is sampled at rates of 3 and 20 times per period (the triangular and square legends respectively). There is no way of differentiating the high frequency sine wave from the lower frequency sine wave when only the triangular samples are used. This graphically demonstrates the problem of aliasing, a loss of information of the desired (higher frequency) signal. The use of the samples represented by the square legends results in no loss of information. Aliasing can also be caused by other phenomena and is discussed further in the next section.

For simple signals with little harmonic content, the signals may be accurately described with just a few spectral components. Time domain signals with discontinuities and/or

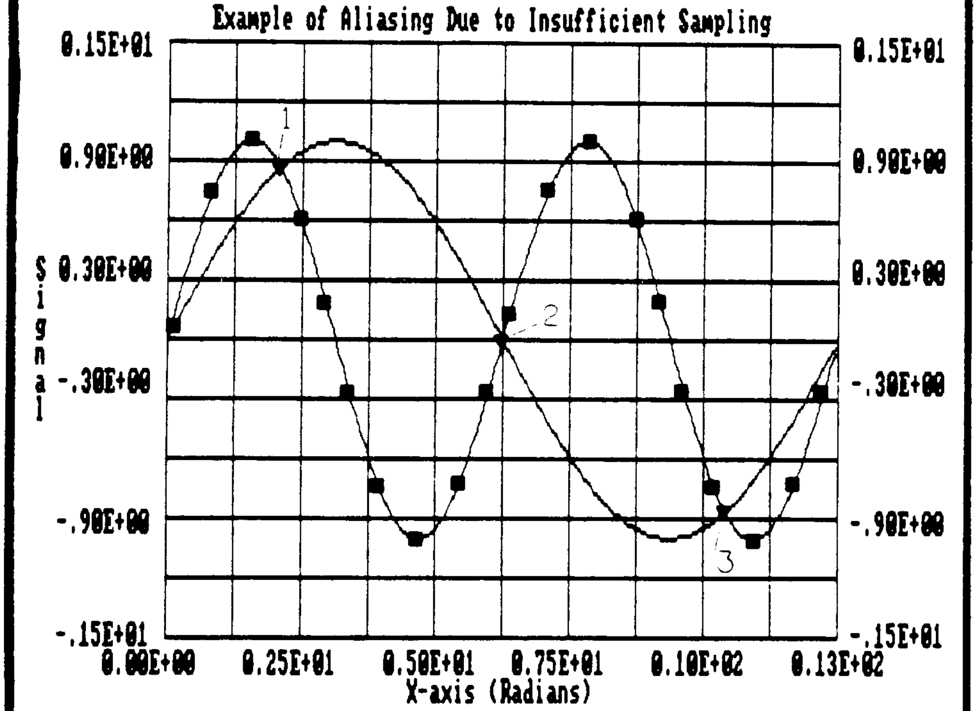


Figure 1: Aliasing Errors Due to Insufficient Sampling

vary complex waveshapes require more components to adequately describe them. As an example, let's take a signal S(t) with a period of 0.2 seconds and reconstruct S(t) using a varying number of spectral components to show the effect on the fidelity of the results. The results of evaluating the first 3, 5, and 256 components of the signal's spectra are plotted in Figure 2 along with the original signal. The original and reconstructed signals are shown on different scales to prevent overlap on the graph. It is impossible to differentiate the original signal from the reconstructed signal when 256 components are used.

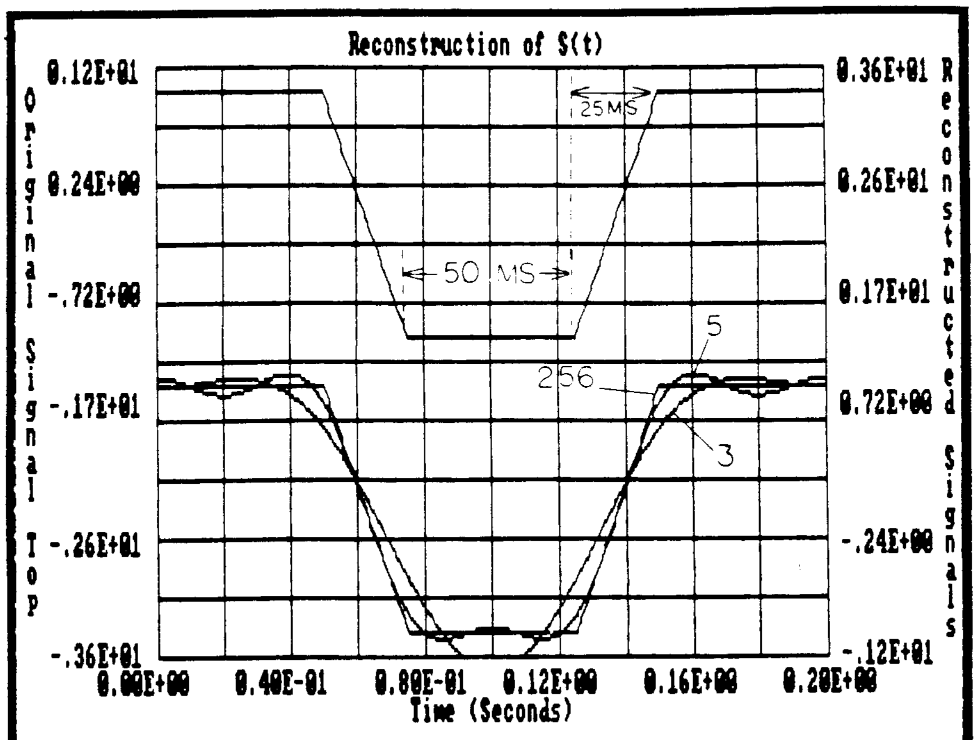


Figure 2: S(t) and Reconstructed S(t)

The Fourier transform was used to decompose the original signal (t) to its basic sinusoidal components. The Fast Fourier Transform (FFT) is a method of computing the Fourier transform faster by removing some of the calculations which result in redundant information. With the advent of the FFT described by Cooley and Turkey in 1965 and the availability of inexpensive computers, the use of the Fourier transform has been so simplified as to make this analysis technique very attractive to today's engineers. Besides being fast and easy to compute, the FFT is reversible. That is, the Inverse Fast Fourier Transform (IFFT) exists and may be used to derive a time domain signal from its constituent spectra.

The FFT and IFFT operations provide powerful problem solving tools in the analysis of system and circuit response to time domain stimuli. A number of techniques exist for computing the transient response of a system to time domain inputs. Included in these techniques are Fourier and LaPlace transforms, Z transforms, and time domain convolution of the system impulse response. FFT techniques are the most popular due to the systematic way that the problems can be set up and the speed with which we can arrive at a practical solution.

Computing System Response Using the FFT

Now that we have the tool to decompose a signal into its constituent frequency elements, it is a simple matter to compute the transient response of a system given its transfer function G(s). Consider the general form of the transfer function G(s):

$$G(s) = \frac{n(0) + n(1)s + n(2)s\ 2 + n(3)s\ 3 \ldots \text{etc.}}{d(0) + d(1)s + d(2)s\ 2 + d(3)s\ 3 \ldots \text{etc.}}$$

Where n(0) is the zeroeth order numerator coefficient, n(1) is the first order coefficient of s in the numerator, etc. and $s = jw = j2\pi f$.

The response of the system G(s) for each of the individual spectral components of the signal S(t) described by equation 1 may be computed by direct evaluation. The results of this "filtering" operation is to pass each component frequency of the spectra with appropriate magnitude and phase "through" the transfer function G(s). The transfer function will affect each of the spectral components differently depending upon the transfer function characteristics. The operation we are performing is that of multiplying the spectra of a signal S(t) by the spectral response of the transfer function G(s).

Both the magnitude and phase of each spectral component will be modified by the transfer function. The results of this "filtering" operation is a new spectra, O(*), different from the original spectra of S(t). When the IFFT of O(*) is computed we get the time domain output waveform of the system O(t). O(t) is the transient response of G(s) to stimuli S(t).

Let's take the signal S(t) of Figure 1 and compute the transient response of the transfer function G(s) given by equation 3.

$$G(s) = \frac{s}{s + 3400s + 3.4E5}$$

The results are shown in Figure 3 where both the input and the output transient response of the system G(s) are plotted on the same graph.

Impulse Response and Errors Due to Aliasing

The FFT by its nature works only for periodic

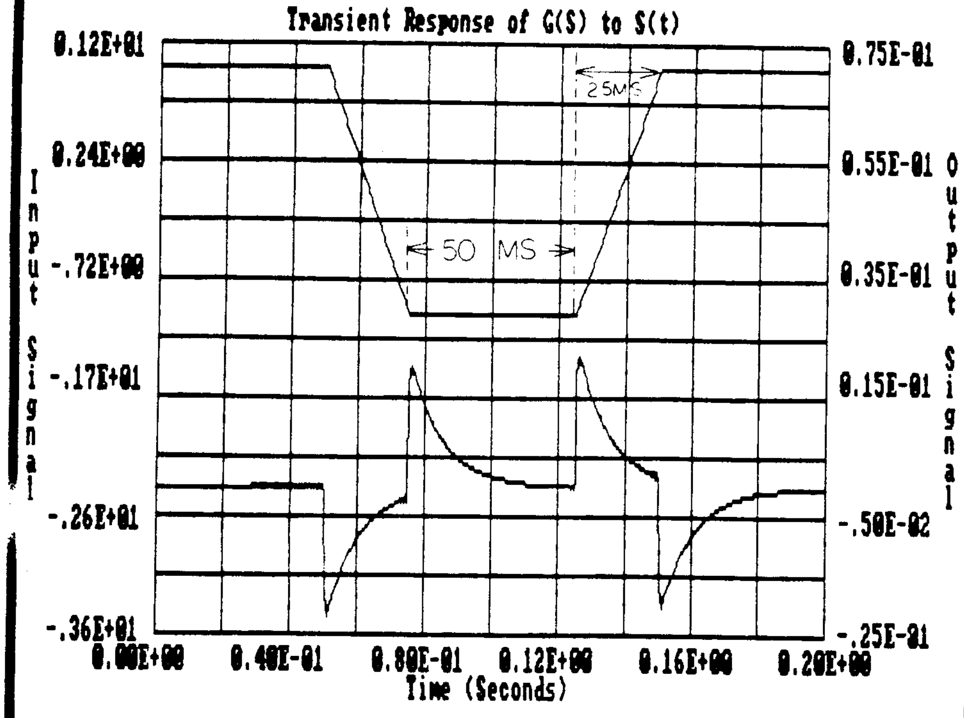


Figure 3: Transient Response of G(s) to S(t)

signals—signals that repeat themselves continuously. This does not mean that non-periodic signals may not be analyzed, but that we must be smart about how we define the signal to be processed. If a linear system G(s) has a non-zero impulse response of T_i seconds, then any waveform which is static for a time greater than T_i seconds will have resulted in the system having arrived at a "steady state." Once a system has reached a steady state condition, it does not "remember" whether the input signal was periodic or not.

Periodic waveforms can be treated as non-periodic with the proper selection on R, the waveform period. There are some restrictions on the choice of T or another type of aliasing error is introduced. If the input signal S(t) is chosen such that the transfer function G(s) has not had time to steady-state before S(t) repeats, then the output waveform 0(t) will be corrupted. The error due to this phenomena is sometimes called "leakage."

The error introduced by leakage can be graphically demonstrated by repeating the FFT-Filter-IFFT operation of the previous example with the waveform shown in Figure 4. Figure 4 shows an input waveform S(t) identical to that of Figure 3 but with the leading and lagging edges of the waveform shortened to such an extent that they no longer meet the criteria for steady state conditions imposed by the impulse response of G(s). Note that the output 0(t) of Figure 4 is different than that of Figure 3. Even though the input waveform has the same slope as before and the width of the pulse is the same, the value of the negative peaks of the output 0(t) have different values.

Multiple Transfer Functions and Spectra Multiplication

In the last section we saw what happened when we multiply the spectra of a time domain signal S(t) by the spectral response of a transfer function G(s). What happens when we multiply the spectral response of two transfer functions or the spectra of two time domain signals? Multiplication of two spectra is a linear operation, that is, if G(*) and H(*) are the spectral responses of two transfer functions G(s) andH(s), then GH(*) = G(*)H(*). It matters not whether we multiply the transfer functions together first, then compute the resulting spectral response, or whether we compute the spectral response of each transfer function separately and then multiply them.

In practical terms what this means is that when a system is described by more than one transfer function block it is not necessary to compute the IFFT and FFT between each transfer function block. It is not necessary to return to the time domain between each transfer function and then return to the frequency domain to continue to compute the transient response of a system with more than one transfer function. The two operations are identical:

S(t)—FFT—G(s)—IFFT—FFT—H(s)—IFFT—0(t)

S(t)—FFT—G(s)—H(s)—IFFT—0(t)

There is another consideration when using a computer to perform spectra multiplication; that of the dynamic range of the mathematics involved. When multiplying spectra of transfer functions it is very easy to exceed the dynamic range of the math package of most small machines. The range of all mathematical operations must fall between 10 to the −38th and 10 to the +38th power. Consider two transfer functions with a combined power of s to the 8th; at any frequency greater than 50KHz or so the computer math package will over or under-flow. If such errors are encountered the use of the first form of the above operations may prevent grief.

What about the meaning of multiplication of the spectra of two time domain signals S(t) and R(t)? Multiplication in the frequency domain is the same as convolution in the time domain. An often-used means to obtain the transient response of a system G(s) is to convolve the input waveform by the impulse response of transfer function G(s). If you know the transfer function but not its impulse response, simply perform the IFFT of the spectral response of the transfer function. The IFFT of the spectral response of a transfer function (or electronic circuit) is the impulse response of that transfer function.

If a very fast way to convolve two time domain waveforms is needed, take the FFT of each signal and multiply their spectra followed by an IFFT operation. You get the convolution of two time domain signals using FFT techniques! These operations take only seconds to perform as opposed to hours if done in the time domain on a small computer.

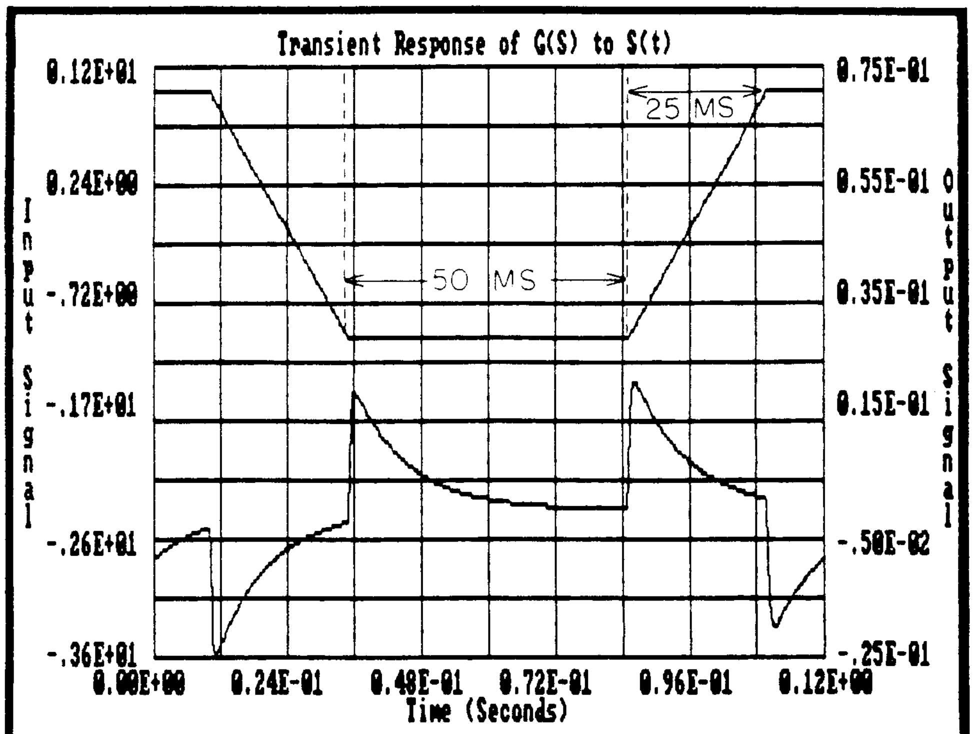


Figure 4: Demonstration of Error Due to Leakage

Non-linear Operations

The fact that we can use the FFT and IFFT to rapidly switch between the time domain and the frequency domain arms us with an analytical tool to analyze systems involving both linear and non-linear elements. The following example demonstrates how the response of a system containing a full wave rectifier, a saturated amplifier with unequal clipping levels, several transfer functions, and a multiplier can be easily analyzed.

Figure 5 shows the block diagram of the system to be analyzed. It is desired to compute the waveforms at all nodes in the block diagram in order to establish acceptance criteria for a circuit to be tested on a volume basis. The analysis performed for this example uses "nominal" values to establish the normal operating conditions of the system. In actual practice, parameters would be modified to represent a worst case situation in order to prevent good circuits from being rejected and bad circuits from being passed.

The input waveform is a sinusoidal "bundle" shown in Figure 6. G(s) is a bandpass filter described by:

$$G(s) = \frac{1250\ s}{s^2 + 1250\ s + 4.0E^7}$$

The stage following the bandpass amplifier saturates for values of output above 10 volts in the positive direction and below −6 volts in the negative direction. Following the saturating amplifier is a full wave rectifier followed by a simple lowpass filter H(s):

$$H(s) = \frac{1}{1 + 0.001\ s}$$

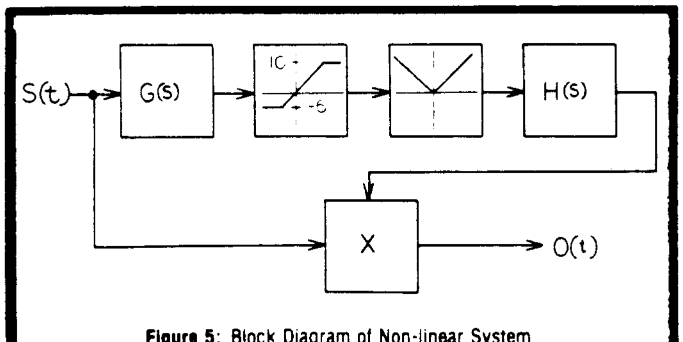


Figure 5: Block Diagram of Non-linear System

The output signal from the lowpass amplifier is multiplied by the original input signal S(t) to form the total system output, O(t). In order to arrive at these waveforms, the following operations are performed:

a) Perform an FFT on the input signal S(t).

b) Multiply the spectra of S(t) by the spectral response of G(s).

c) Perform an IFFT to get back to the time domain; the resulting wave form is that at the output of the bandpass amplifier, G(s).

d) In the time domain, clip all signals greater than +10 volts and less than −6volts.

e) Full wave rectify (take the absolute value) of the resulting waveform.

f) Perform an FFT on the resulting signal from step e).

g) Multiply the spectra of step f by the spectral response of the lowpass filter H(s).

h) Perform an IFFT to obtain the time domain signal at the output of the lowpass filter.

i) Multiply the output of the lowpass filter point-by-point with the input signal S(t) to obtain the output signal O(t).

Figures 6 through 8 show the waveforms at each of the nodes of the system as would be seen by an oscilloscope connected to these nodes. As has been demonstrated, it is possible to solve a complex non-linear signal processing

QUALITY SOFTWARE AT REASONABLE PRICES

CP/M Software by

Poor Person Software

Poor Person's Spooler **$49.95**
All the function of a hardware print buffer at a fraction of the cost. Keyboard control. Spools and prints simultaneously.

Poor Person's Spread Sheet **$29.95**
Flexible screen formats and BASIC-like language. Pre-programmed applications include Real Estate Evaluation.

Poor Person's Spelling Checker **$29.95**
Simple and fast! 33,000 word dictionary. Checks any CP/M text file.

aMAZEing Game **$29.95**
Arcade action for CP/M! Evade goblins and collect treasure.

Crossword Game **$39.95**
Teach spelling and build vocabulary. Fun and challenging.

Mailing Label Printer **$29.95**
Select and print labels in many formats.

Window System **$29.95**
Application control of independent virtual screens.

All products require 56k CP/M 2.2 and are available on 8" IBM and 5" Northstar formats, other 5" formats add $5 handling charge. California residents include sales tax.

Poor Person Software
3721 Starr King Circle
Palo Alto, CA 94306
tel 415-493-3735

CP/M is a registered trademark of Digital Research

FREE SOFTWARE

RENT THE PUBLIC DOMAIN!

User Group Software isn't copyrighted, so there are no fees to pay! 1000's of CP/M and IBM software programs in .COM and source code to copy yourself! Games, business, utilities! All FREE!

CP/M USERS GROUP LIBRARY
Volumes 1-92, 46 disks rental—$45

SIG/M USERS GROUP LIBRARY
Volumes 1-90, 46 disks rental—$45
Volumes 91-176, 44 disks rental—$50
SPECIAL! Rent all SIG/M volumes for $90

104 FORMATS AVAILABLE! SPECIFY.

IBM PC-SIG (PC-DOS) LIBRARY
Volumes 1-200, 5¼" disks $200

Public Domain User Group Catalog Disk $5 pp. (CP/M only) (payment in advance, please). Rental is for 7 days after receipt, 3 days grace to return. Use credit card, no disk deposit. Shipping, handling & insurance—$7.50 per library.
(619) 914-0925 information,
(619) 727-1015 anytime order machine
Have your credit card ready! VISA, MasterCard, Am. Exp.

Public Domain Software Center
1533 Avohill Dr.
Vista, CA 92083

problem with the FFT, IFFT, and a few simple time domain operations. What makes this analysis technique possible is the speed with which these operations can be performed. ■

Bert P. van den Berg is president of BV Engineering, which produces the SPP (Signal Processing Program) and PLOT-PRO (Scientific Graph Printing Program) software which were used to analyse the data and print the graphs used in this article. These programs plus ACNAP (Electronic Circuit Analysing Program) and DCNAP (DC Network Analysis Program) are available from BV Engineering, 2200 Business Way, Suite 207, Riverside, CA 92501.

Note: Figures 7 and 8 can be found on page 22.

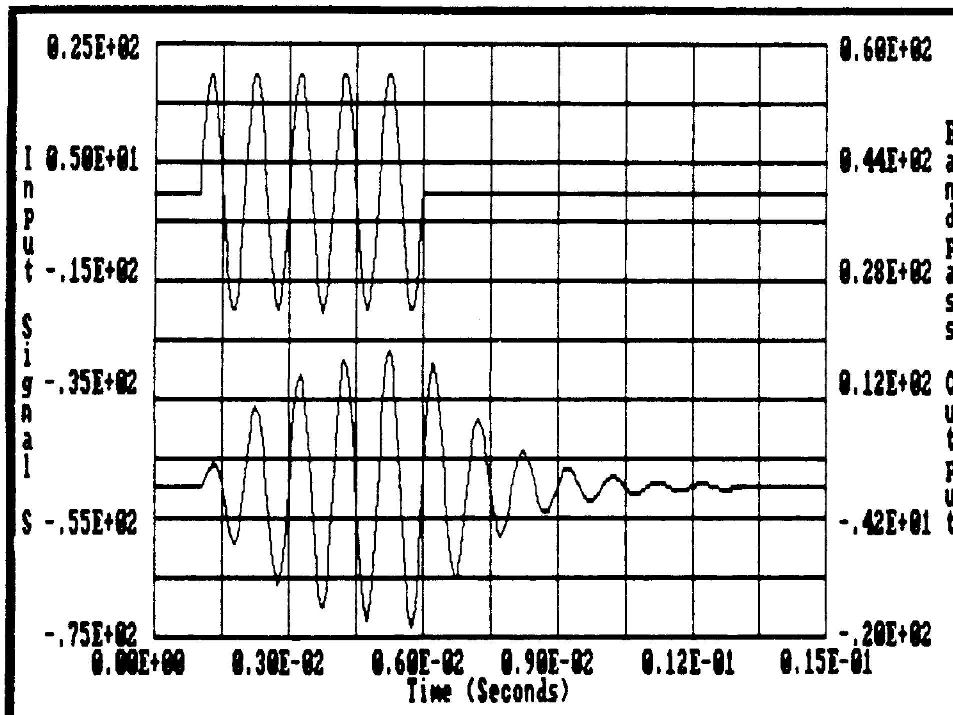


Figure 6: Input Signal S(t) and Bandpass Output

Letters From Our Readers

Dear Computer Journal:

I was intrigued by the listing on your "Author's Hot Sheet" in the current issue. I'm not qualified to write such an article, but I'm planning to build and instrument an earth-sheltered house with a minimum of 8' of earth cover. So your listing of soil temperatures at various depths over a period of years really hit my funnybone. The solar heating and light intensity were another pair of variables which I intend to instrument, since I'll be piping in light through large pipes from skylights at the surface.

Some other items you hit were pressure (strain on the concrete dome) and liqui . level (water supply in large buried tanks), not to mention the servos to control items such as light supply, air handling, and various items around the home.

The actual sensors and associated equipment aren't going to be my biggest problem (I'm an EE, but not really proficient in digital techniques)—the processing and storage of the data in the computer will be what I'll be looking forward to in the magazine.

Hope you get a lot of takers on your appeal to authors. Best of Luck.

L.S.
Arizona

Dear Lance Rose:

I recently bought some back issues of *The Computer Journal*, among which was Volume 1, Number 2, issued in October of 1983.

I was very interested in your article on CP/M file transfers and in Mr. Mosher's contribution on floppy disk formats. However, I'm afraid one statement of yours has me confused. In your article it says that in return for $15 a CP/M copy of the source files will be sent on an 8" single density floppy.

Have I missed something? How would you know what format to send? Do you have the formats for all computers? If it is not too much inconvenience I should be grateful for an explanation. I am relatively new to the world of computing and I am hoping to build a "Big Board" Z80 computer but this business of formats has got me spooked.

N.D.F.
Florida

Dear N.D.F.:

Let me try to clear up the confusion you mentioned in your letter. You are correct in assuming that I would have no way of knowing what type of floppy disk system you had. The programs were offered on a 8" floppy since that was the only format my system used at the time the article was written. I might mention that in spite of the plethora of 5¼" formats, the 8" single density CP/M format remains a standard for that size diskette.

Between the time that article was published and the present, I have acquired a second system that can make a variety of 5¼" soft-sectored diskette formats. I you would like the software on a minifloppy, let me know what format you are using and I will send it to you for the same price.

Lance Rose
Technical Editor

Dear Neil Bungard:

I was just reading your article on interfacing the Sinclair computers in "Interfacing Tips and Troubles" in issue number 13. In that article you state the "the Sinclair machines do not support memory mapped I/O—only accumulator I/O." This statement is not true, in fact, I have used the 8K to 16K block of memory which is not used by the Sinclair ZX81 ROM to interface an EPROM programmer and an IBM Selectric typewriter, for storing extended ROM operating programs, and for a speech generator and sound generator.

There have been numerous articles written in *SYNTAX* magazine, *SYNC* magazine, and in others on using this area for MMI/O In fact, because of the limitations in using AI/O (ie: having to use machine code) this technique is the only way I would interface to the Sinclair. All of my interfaces use the peek and poke commands from BASIC to facilitate the interface.

It is my understanding, however, that the TS2068, which I do not own, may indeed only support AI/O.

The trick to using MMI/O in the ZX81 8K free block is to make sure the "ROMCS" line on the Sinclair backplane is raised high to disable the ROM's mirror image in that memory range, when performing MMI/O.

L.E.D.
Michigan

Dear Computer Journal:

I recieved your subscription form through my father, who is a High School Physics and Physical Science teacher. I am a lab assistant for Advanced Computer Communications. I have noticed that the older, popular magazines are getting further away from hardware as time goes on. I want a hardware experimentor, learner magazine.

Here is my one year subscription plus enough, I hope, to cover all back issues through November. I realize it is a risk but I think you are going in the right direction. Here is my vote for the free enterprise system as well as its most basic underpinings.

J.H.
California

BASE
A Series on How To Design and Write Your Own Database
By E.G. Brooner

Why File Design is Important

In the first article of this series we used a telephone directory as example of how a database could be organized. We decided that it would be desireable to "find" a complete entry, or set of entries, by either the name, the address, the phone number, or some other identifier such as the type of business. When searching a list of names, too, it might be desirable to isolate the first and last names to facilitate alphabetical arrangements. Unfortunately, most directories are not organized for this kind of elaborate searching, but if we produced our own it could be. A typical printed directory is "keyed" to the last name only and the rest has to fall into place. This illustrates the basic difference between a database program and a simple store-and-retrieve filing application.

In a database (or any collection of data stored on a disk) the organization must provide some hierarchy, or framework into which the data is deposited. The most general assembly of data is the database itself; the next division is into (one or more) data files. Each file is further divided into records and then into fields. A complete name could be a field, or the first and last name could be separate fields. The entire address can be one field, but for many purposes it is best if the street and number, city, state, and zip code each comprise a separate field.

The phone number is yet another field, and the type of business or similar indicator of purpose would also be a useful classification. Other data collections might contain other classifications; a magazine subscription list, for example, might contain an expiration date. These considerations and more affect the structure of the database and the way information is stored in and retrieved from it. The ability to retrieve data in the order you choose is an important characteristic of any collection of data.

Another database example, a very simple one, is a checkbook. In this relatively simple collection of data you might want to locate a certain line item by check number or type of payment, or find those within a certain range of dates, and in addition, do some simple addition and subtraction along the way. In either case it would be impractical to store entire records or transactions, because each might be treated differently in the recovery or manipulation process. In each checkbook entry, therefore, the check number, date, item and amount all have to be considered as separate entities which are *related* to one another in some logical manner. Together they constitute a record, one in which different items might be significant at different times. We would call the significant item (date, check number, or whatever, the *key* to that record, or its *key field*.

For many purposes we only need one key field per record, at least at any one time. A more flexible database will allow us to specify several fields in each record as key fields - perhaps all of them. With a checkbook so organized, then, we could find all records (entries) that pertained to "rent" and occured between two specified dates.

Once the file structure is assigned all information going into it and coming out again has to rigidly conform to the original limits. You might not always want phone numbers as part of a name-and-address list but if they were not originally provided for they cannot be added later, hence you would probably allow a field for that information when the file is being designed.

How a Database is "Different"

A completely inflexible storage system cannot properly be called a data base. Almost every application program stores data in some manner, and if there is a minimum of flexibility in entering and retrieving the information, and in putting it to use, it is more properly referred to by some name like "file handler." To earn the title of database your system must let you specify, when the base is created, just what fields will be permitted, what type of data each will contain (numbers or text, for example) and how much information can be entered. A name field that will accomodate only 10 text characters is not much use, nor is a phone number field that has no room for the area code, or a zip code field that cannot accomodate the new, longer numbers or the Canadian zips that are mixed numbers and letters. By the same reasoning, it would be senseless to make every field 50 characters long when some of them will only hold a zip code.

Commercial database software usually lets you establish these parameters in a relatively simple manner such as from a menu, rather than having to write a new program for each new use. This emphasizes the main difference between a simple file handler and a database. A file handling application lets you do a limited number of things from the package, and the file structure is pre-determined. A database package lets you use the *same program* to perform *several different applications* which you, the user, design as need be. The effect is that the database program actually writes a new application to your specifications, within its inherent limits. It is a program that writes programs (or at least the file handling portion, which is often the most difficult part).

If you write your own software some similar arrangement has to be made. One of the tasks in designing your own database, then, is to anticipate the data structure and provide for more-or-less painlessly changing it as your needs change. In commercial software there is often some arbitrary limit to the choices you have in such matters as number and length of fields, completely aside from any limitations your operating system might impose. By designing your own system you can get around all of these obstacles.

This brings up one of the reasons that our examples will be in C-BASIC. There are BASICs which arbitrarily limit the record length and file structure; one of these with which I am familiar is the Microsoft BASIC used on the original TRS-80 Model I. When dealing with random access files (the only kind useful for data base use) it requires that each record be 256 bytes in length. You do not have to use all of that space, of course, but unused space is wasted and no one record can exceed 256 bytes. Again, some BASICs (North Star comes to mind) require that you reserve file space in advance—if your file outgrows the reserved space you are in real trouble. C-BASIC (and some others) lets you establish records of any length and lets the file "grow" as you add more entries, until the entire disk is full.

Random Files Defined

Let's review the foregoing just briefly: a field can be thought of as a box into which you can put a certain amount of information, of a certain type, which will occupy a specified relative position in some collection of related data. All of the related fields such as those pertinent to a single individual, or a single transaction, will usually be referred to as a record. A number of similar records will usually (but not always) make up one distinct file. Repeating the heirarchy as we will use it here, then, from the particular to the general, we will have fields, records, files, and databases. The database will be the entire collection; the file(s) will be collections of records, and the records will each be a collection of fields. There should be a provision for changing the file structure for a new application without re-writing the entire program. If we are too repetitious about this fact it is only because this concept of structure is so basic to an understanding of database use and design. Let's agree at this point that we will deal primarily with *random access* files, those in which we can read or change any record without slogging through and re-writing the entire file. We'll occasionally make use of the other major file type, *sequential files*, for some purposes. These files are easy to handle if we want to simply store an array or a list of numbers, all of which need to be read out at the same time anyway. I'll pause here to repeat some advice I gave earlier—familiarize yourself with the way your own system generates, reads and writes data files before going any further into the details. We will be especially interested in creating and defining files within a program without re-writing the program itself.

Start Thinking About Your File Design Now

You might give some thought, too, to the data types that you might want to handle. Some items will be text (string variables) while others will be numbers. Some programmers like to enter all information as text and let the program convert it to numbers when necessary. This is because a program that is expecting text will accept numbers but the reverse is not true. For example, you could enter the number "12345" as text and if you wanted later to perform an arithmetical operation you would use the feature (found in nearly all BASICs) to convert the text to its equivalent numeric value. Things like this are a matter of programming "style" and we all have our own preferences. At the very least, pick something like names and addresses and try your hand at designing a record—how many fields, how long each is, and whether it is text or numeric. You might also give some thought to the problem of inadvertently entering a longer name than you left space for; this comes under the heading of error-trapping, and a good program simply won't let you make an illegal entry. Again, this is a matter of programming style.

We'll cover the design of our databases' files and records and the entering of data as we go along. Perhaps the most important part of the design, though, is that which permits us to readily retrieve data after it has been entered. After all, that is the main reason for having a database! Finding data involves various means of sorting or ordering it, and searching through the files to find a particular item or set of items. That will be the subject of the next article in this series and it is an interesting programming exercise for those of us who enjoy that sometimes black art.

For now we can observe the opening module of the program; this will serve to set the stage for the next section, which is the creation of file structures. Actually,

APROTEK 1000™ EPROM PROGRAMMER

TECHNICAL BREAKTHROUGH NOW ALLOWS A PRICE BREAKTHROUGH

only $250.00

A SIMPLE, INEXPENSIVE SOLUTION TO PROGRAMMING EPROMS

The ***APROTEK 1000*** can program 5 volt 25XX series through 2564, 27XX series through 27256 and 68XX devices plus any CMOS versions of the above types. Included with each programmer is a personality module of your choice. Others are only $10.00 ea. when purchased with ***APROTEK 1000***. Later you may require future modules at only $15.00 ea., postage paid. Available personality modules: PM2716, PM2732, PM2732A, PM2764, PM2764A, PM27128, PM27256, PM2532, PM2564, PM68764 (includes 68766). (Please specify modules by these numbers)

APROTEK 1000 comes complete with a menu driven BASIC driver programmer listing which allows READ, WRITE, COPY, and VERIFY with Checksum. Easily adapted for use with IBM, Apple, Kaypro, and other microcomputers with a RS 232 port. Also included is a menu driven CPM assembly language driver listing with Z-80 (DART) and 8080 (8251) I/O port examples. Interface is a simple 3-wire RS 232C with a female DB-25 connector. A handshake character is sent by the programmer after programming each byte. The interface is switch selectable at the following 6 baud rates: 300, 1.2k, 2.4k, 4.8k, 9.6k and 19.2k baud. Data format for programming is "absolute code" (i.e., it will program exactly what it is sent starting at EPROM address 0). Other standard downloading formats are easily converted to absolute (object) code.

The ***APROTEK 1000*** is truly universal. It comes standard at 117 VAC 50/60 HZ and may be internally jumpered for 220/240 VAC 50/60 AZ. FCC verification (CLASS B) has been obtained for the ***APROTEK 1000***.

APROTEK 1000 is covered by a 1 year parts and labor warranty.

FINALLY — A Simple, Inexpensive Solution To Erasing EPROMS

APROTEK-200™ EPROM ERASER
Simply insert one or two EPROMS and switch ON. In about 10 minutes, you switch OFF and are ready to reprogram.
APROTEK-200™ only ***$45.00.***

APROTEK-300™ only ***$60.00.***
This eraser is identical to ***APROTEK 200™*** but has a built in timer so that the ultraviolet lamp automatically turns off in 10 minutes, eliminating any risk of overexposure damage to your EPROMS.
APROTEK-300™ only ***$60.00.***

APROPOS TECHNOLOGY
1071-A Avenida Acaso, Camarillo, CA 93010
CALL OUR TOLL FREE ORDER LINES TODAY:
1-(800) 962-5800 USA or 1-(800) 962-3800 CALIFORNIA
TECHNICAL INFORMATION: 1-(805) 482-3604
Add Shipping Per Item: $3.00 Cont. U.S. $6.00 CAN, Mexico, HI, AK, UPS Blue

this first section does create a pair of blank files the first time the program runs. The first is "BASES.DEF" and the second is "BASES.EXT." The program cannot be run unless these files exist.

This action takes place between lines 1100 and 1150. If you are familiar with BASIC this portion is easy to follow, with the possible exception that C-BASIC needs line numbers only for lines that will be referenced from elsewhere in the program.

Once one or more data base files have been created, this portion is skipped. In that case, the program reads the two referenced files and prints, on the screen, the names of each data collection and the number of records each contains. How that information happens to be there will become apparent later. In any event, the final action in this module is the presentation of the main menu (line 1300) which ables you to choose the various program functions listed ere. The perhaps-strange C-BASIC syntax will be explained in the next article, as we take up the more-or-less automatic creation of data structures. For now we'll just mention the rather unorthodox use of the "WHILE END" loop that will show up in this program. It is used here as an endless loop, and the exit is in response to the previously executed "IF END" statement.

```
REM     PROGRAM BASE.COM TO PERMIT AUTOMATIC
REM     CREATION/MANIPULATION OF MULTIPLE DATABASES
REM     ****************************************

REM     BY E.G. BROONER VERS 1.01 DEC 22 1983

REM     ****************
REM     *OPENING MODULE*
REM     ****************

        DIM FIELD.NAME$(12),FIELD.LENGTH%(12),DATA$(12),FL%(12)
1000    GOSUB 9999        REM CLEAR SCREEN**
        PRINT "BASE, VERS 1.01 83/12/22":PRINT
        FOR X%=1 TO 12
                FIELD.NAME$(X%)=""
                DATA$(X%)=""
                FIELD.LENGTH%(X%)=0
                FL%(X%)=0
        NEXT X%
        IF END #17 THEN 1100
        OPEN "BASES.DEF" RECL 10 AS 17
        OPEN "BASES.EXT" RECL 5 AS 18
        GOTO 1150

1100    PRINT "NO DATA BASES CURRENTLY ACTIVE. WHEN MAIN MENU APPEARS,"
        PRINT" YOU MUST CHOOSE THE 'CREATE' OPTION BEFORE PROCEEDING"
        FOR X%=1 TO 1000
                NEXT X%

REM     BEGIN FILE OF DATABASE NAMES IF NONE EXIST
        CREATE "BASES.DEF" RECL 10 AS 17
        CREATE "BASES.EXT" RECL 5 AS 18
        PRINT #18;0
        GOTO 1300

1150    IF END #17 THEN 1200
        WHILE RECORDS%=EXIST%
                READ #17;BNAME$
                IF BNAME$="" THEN 1190
                PRINT "DATABASE ";
                PRINT BNAME$;" - ";
                FILE$="B"+BNAME$+".EXT"
                OPEN FILE$ RECL 6 AS 16
                READ #16,1; NBR.RECS%
                PRINT TAB(20); NBR.RECS%;" -";" RECORDS"
                CLOSE 16
1190    WEND
1200    PRINT
        PRINT "ENTER NAME OF BASE EXACTLY AS IT APPEARS"
        INPUT NAME$
        PRINT
REM     VERIFY THAT FILES EXIST
        IF END #17 THEN 1250
        FOR X%=1 TO 100
                READ #17,X%;BNAME$
                IF BNAME$=NAME$ THEN 1300
        NEXT X%
1250    PRINT"NAME DOES NOT MATCH ANY EXISTING DATABASE"
        PRINT
        GOTO 1150

1300    GOSUB 9999  REM CLEAR THE SCREEN
        PRINT "1) CREATE OR DELETE A DATABASE"
        PRINT "2) ENTER DATA IN ";NAME$
        PRINT "3) MODIFY DATA IN ";NAME$
        PRINT "4) SEARCH DATABASE ";NAME$
        PRINT "5) SORT ANY FILE ON ANY FIELD"
        PRINT "6) CHOOSE ANOTHER DATABASE"
        PRINT "7) PRINT FORMATTER"

        PRINT
1400    PRINT "CHOOSE BY NUMBER"
        INPUT CHOICE%
        IF CHOICE% <1 OR CHOICE% >7 THEN 1400
        ON CHOICE% GOTO 2000,3000,4000,5000,6000,7000,8000
```

■

Time-Frequency, continued from page 18

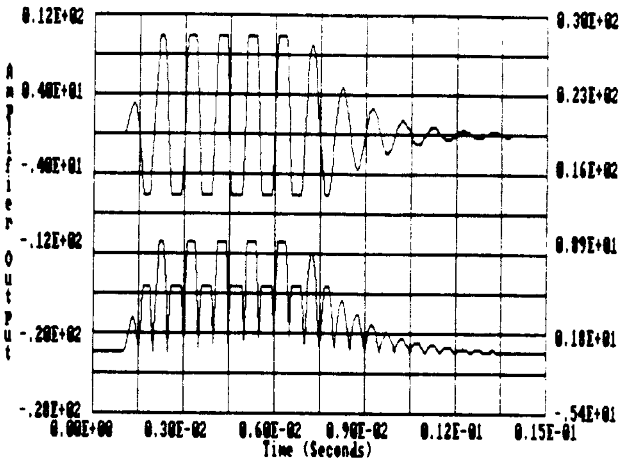


Figure 7: Ampligier and Rectifier Outputs

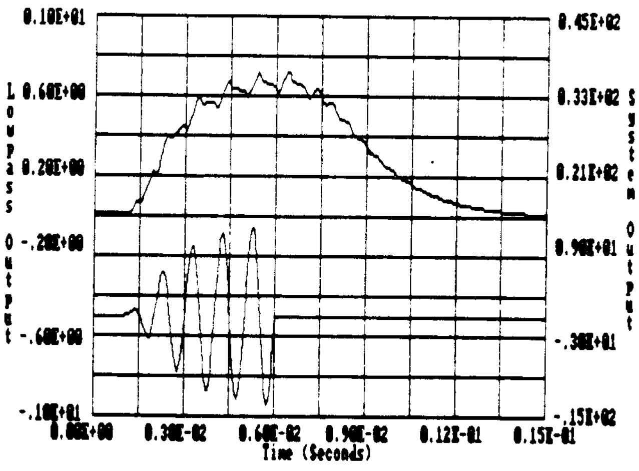


Figure 8: Lowpass and System Output [O(t)]

Books of Interest

The Free Software Catalog and Directory
by Robert A. Froehlich
Published by Crown Publishers, Inc.
One Park Avenue
New York, NY 10016
475 pages, 8½" × 11", softcover, $9.95

One of the advantages of using the CP/M system is the vast amount of public domain software which is available. Much of the new work is being done on sixteen bit systems, but eight bit CP/M still runs on more computers and types of computers from more manufacturers than any other operating system. Even many of the computers which were not intended to run CP/M can use this system with additional software and/or a Z-80 card. This book contains sources for the software and hardware to adapt the following computers to run CP/M: Atari; Apple; Commodore; Digital Equipment DEC, BAX, PDP11, ECLIPSE, and NOVA; IBM-PC and PC Jr.; Radio Shack TRS-80 models II, III, 4, 12, and 16; and the Hewlett Packard 85.

The book is primarily a catalog of the CPMUG and SIG/M libraries, but the authors have also added some very useful information which will help the beginner use the programs, including indexes cataloged by key word, language, author, and file name. The CPMUG disks are available in 8" format for $10; Northstar, Epson QX-10, Apple, and Kapro II disks are $15. The SIG/M disks are $6 for 8", and other formats are available. **Remember that these programs must be run on a computer which uses the CP/M operating system.**

The variety of programs available in these libraries is fantastic—there are games, utilities, finances, mathematics, languages and a lot more in many different languages. I found sixteen disks I need the first time I looked through the catalog, and I'm sure I'll find more next time. Alternatives to ordering the individual disks are to rent the entire library from National Public Domain Software Center, download the programs from a bulletin board, or copy them at a regional user's group library. In these cases you can save a lot of time by using this book to decide beforehand which programs you want.

Most of the programs will be useable as is, but even the ones which need minor modifications will serve as a good starting point and save you the work of reinventing the wheel. This book is recommended for all CP/M programmers and hackers. ■

What Would YOU Like to See in a Database Program?

Most database programs are designed for business use, and do not meet our readers' needs for scientific and experimental data handling.

We are publishing a series on writing your own database program for applications where the available commercial programs are not suitable. We need your comments on features that you feel should be included. This program will be placed in the public domain for use by the scientific and engineering communities.

Tell us what features you would like to see in order to accomplish tasks which are difficult or impossible with the business-oriented programs.

3.5" DRIVES

WHY GO 3.5"? THEY ARE SMALL, FAST, LIGHT, LOW POWER, COMPATIBLE WITH 5.25" DRIVES (ON THE SAME CABLE), AND THE DISKS AND DRIVES ARE MORE RUGGED AND RELIABLE. MITSUBISHI HAS A RECORD FOR BUILDING THE FINEST QUALITY DISK DRIVES.

COMPATIBLE WITH IBM PC, PC COMPATIBLES, RADIO SHACK, or ANY SYSTEM NOW USING STANDARD TANDON OR SHUGART TYPE 5.25" DRIVES

☐MF351 SS 360K w/manual, connectors, disk $215
☐MF353 DS 720K w/manual, connectors, disk avail soon
☐Case and Power supply (built in spike protection) avail soon
☐Cable (two drive ribbon cable) $15

MITSUBISHI HALF HEIGHT 5.25" DRIVES ALSO AVAILABLE
☐M4853 DS 720K $225 ☐M4854 DS 1.2 Meg $245

FREE SHIPPING, QTY DISCOUNTS, CA RESIDENTS 6% tax
TECHNICAL QUESTIONS WELCOMED!

MANZANA
935 Camino Del Sur
Isla Vista, CA 93117
CHECK, M.O., VISA, M.C.
1-805-968-1387

Interfacing Tips and Troubles

A Column by Neil Bungard

Last month in Interfacing Tips and Troubles we presented part one of a two part article on interfacing the Sinclair computers. In part one we looked at the hardware required for a simple interface, and explained how to protect memory space for storing machine language programs. This month in part two, we will present the software required to complete the interfaciing task. The minimum software required to accomplish an input or an output is shown in Figure 1. In the following sections, I will discuss the software in functional blocks and explain how each block is used to accomplish the interfacing task. Remember from last month that all information in brackets [ ] refer to the Spectrum and the TS2068 computers.

Reserving Space For the Machine Language Routines

When using the TS1000 and the TS1500, line one of the BASIC program should be: 1 REM 123456789012345678. This command reserves 18 locations of protected RAM memory for storing the machine language routines. The first eight characters following the REM statement (1 through 8) occupy addresses 16514D through 16521D which will be used to store the machine language output routine. The second eight characters (9 through 6) occupy addresses 16522D through 16529D which will be used to store the machine language input routine. The last two characters (7 and 8) occupy addresses 16530D and 16531D which will be used to pass values from the BASIC programs to the machine language programs and vice versa. For the color computers, line one of the BASIC program should be: 1 CLEAR 32129. This command reserves space above the BASIC routines. Address locations 32130D through 32138D will contain the machine language output routine. Locations 32139D through 32146D will contain the machine language input routine, and locations 32147 and 32148D will be used to pass values from the BASIC programs to the machine language routines and vice versa.

Loading the Machine Language Routines

Referring to Figure 1, lines 5 through 70 of the BASIC program are used to automatically POKE machine language instructions into the space reserved by the REM statement. Line 10 sets the beginning storage location to 16514D [32130D]. Line 20 inputs a machine instruction which is entered through the keyboard. Note that the machine instructions must be in hexadecimal format. Line 30 checks to see if the character which was entered was an "S," and if so, program execution stops. If not, line 40 converts the instruction to decimal format and POKES it in the appropiate memory location. Line 50 increments the machine language storage location and the entry process is repeated. You may be wondering why we enter the machine instructions in hex format since they are converted to decimal in line 40 before they are POKED into memory. The reason is that if we use the hex format, the instructions can conveniently be represented with two characters. Also, hex is the format that is most used by companies and authors that provide listings of machine language instructions. To execute the machine language entry program, type the command "RUN 5" and press the ENTER key. Once the program is running, input the following hex values: 06, 40, 0E, 92, 0A, D3, 00, 06, 40, 0E, 93, DB, 01, 02, C9. These are the machine language values shown in Figure 1. Note: Do not enter the commas, and be sure to press the ENTER key between each instruction. When the last instruction has been input, enter an "S" to cease program execution.

```
                    BASIC ROUTINES

  1 REM 123456789012345678
 [1 CLEAR 32129]

  5 REM THIS IS THE  MACHINE LANGUAGE ENTRY ROUTINE
 10 LET X=16514
[10 LET X=32130]
 20 INPUT A$
 30 IF A$="S" THEN STOP
 40 POKE X,16*CODE A$+CODE A$(2)-476
 50 LET X=X+1
 60 GOTO 20
 70 STOP

 100 REM THIS IS THE BASIC OUTPUT ROUTINE
 110 CLS
 120 PRINT "INPUT A VALUE BETWEEN 0 AND 256 AND PRESS THE ENTER KEY"
 130 INPUT X
 140 POKE X,16530
[140 POKE X,32147]
 150 LET A=USR 16514
[150 LET A=USR 32130]
 160 GOTO 110
 170 STOP

 200 REM THIS IS THE BASIC INPUT ROUTINE
 210 CLS
 220 PRINT "PRESS ENTER TO INPUT A VALUE FROM THE INTERFACE"
 230 INPUT X
 240 LET A=USR 16522
[240 LET A=USR 32139]
 250 A=PEEK (16531)
[250 A=PEEK (32148)]
 260 PRINT A
 270 GOTO 220
 280 STOP
```

```
                    MACHINE LANGUAGE ROUTINES

output routine:   06 40     LD B,40      [06 7D   LD B,7D]
                  0E 92     LD C,92
                  0A        LD A,(BC)
                  D3 00     OUT 00,A
                  C9        RET

input routine:    06 40     LD B,40      [06 7D    LD B,7D]
                  0E 93     LD C,93
                  DB 01     IN A,01
                  02        LD (BC),A
                  C9        RET

Note:  use the commands in brackets [ ] with the Spectrum and TS2068
       computers.
```

Figure 1

Outputting To the Latch

Lines 100 through 170 of the BASIC program work with the machine language output routine to write data to the eight bit latch in Figure 2 (reprinted from part 1). Line 120 asks for a value between 0 and 255D to be input through the keyboard. (255 is the limit because it is the largest decimal value which can be represented with 8 bits.) Line 130 inputs the value and line 140 temporarily stores it in a RAM memory location where it will later be retrieved by the machine language output routine. Line 150 may look strange, but it is the command which actually calls the machine language output routine. Branching to the machine language output routine, LD B,40 [LD B,7D] and LD C,92 set the BC register pair to point to the RAM memory location just filled by line 40 of the BASIC program (4092 hex is 16530 decimal) [7D92 hex is 32147 decimal]. LD A,(BC) moves the value which is in location 16530D [32147D] into the Z80's accumulator. OUT 00,A sends this value, which was input through the keyboard, to the latch in Figure 2. RET returns program execution back to the BASIC program where the entire process is repeated. To execute this program, type the command "RUN 100" and press the ENTER key. Follow the instructions given on the screen. Once a value has been entered, it should be present on the output pins of the latch in Figure 2. This can be verified by checking the output of the latch (pins 2, 5, 6, 9, 12, 15, 16, and 19 on IC3) with a logic probe.

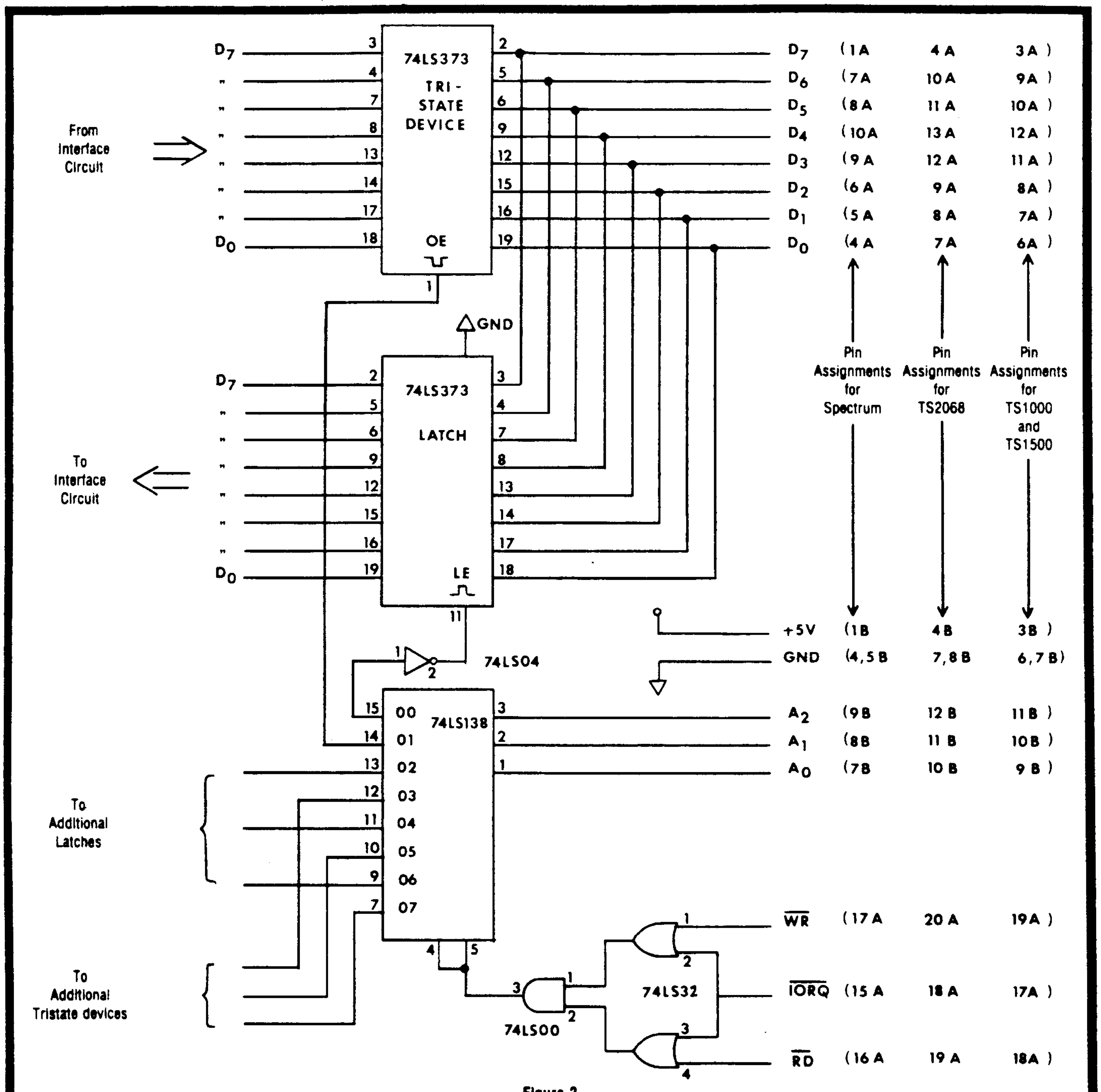
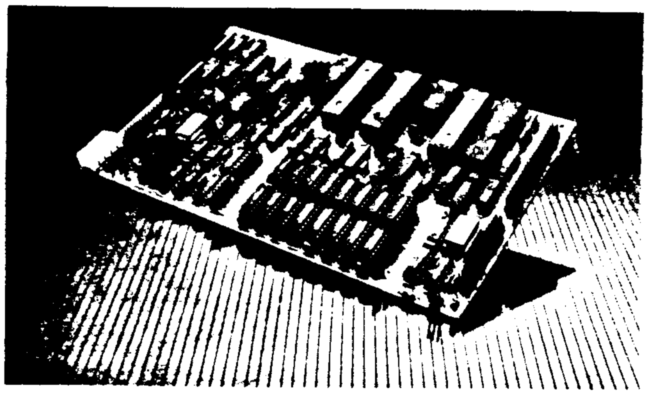


Figure 2



POWER THAT GOES ANYWHERE!

Single Board Computer

FAST	— 6MHz Z80B* CPU
POWERFUL	— 64K to 256K RAM, 2K to 64K ROM
	— 5¼" and 8" Floppy Controller, SASI
	— 2 RS-232, Centronics Port
FLEXIBLE	— 50-pin I/O Expansion Bus.
SMALL	— 5¾" x 10"

DAVIDGE CORPORATION
292 East Highway 246
P.O. Box 1869
Buellton, CA 93427 (805) 688-9598

*Z80 is a registered trademark of Zilog

Interfacing Workshop To Be Held

The Virginia Polytechnic Institute in Blacksburg will hold their popular interfacing workshop in the spring of 1985. The workshop, on "Personal Computer and STD Computer Interfacing for Scientific Instument Automation," will be directed by Mr. David E. Larsen and Dr. Paul E. Field. The three-day workshop will be held twice, March 14 through 16, and May 30, 31, and June 1, 1985. It will be held at the Virginia Polytechnic Institute in Blacksburg. The cost is $450.00.

This is a hands-on workshop, with each participant wiring and testing interfaces. For more information, call or write Dr. Linda Leffel, C.E.C., VA Tech, Blacksburg, VA, 24061, (703) 961-4848.

Inputting From the Tristate Device

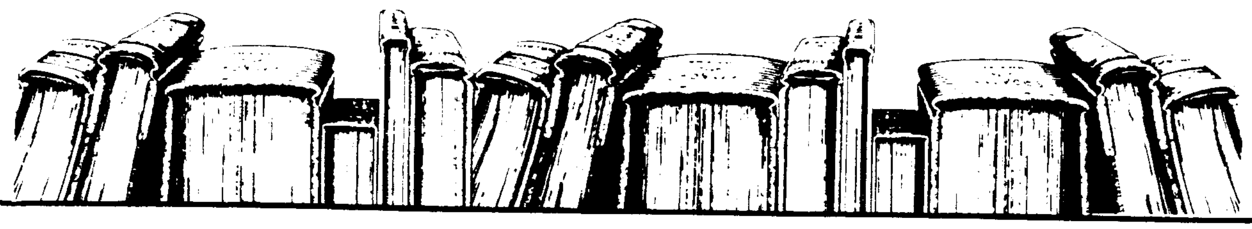
Lines 200 through 280 of the BASIC program work with the machine language input routine to read data from the eight bit tristate device in Figure 2. Line 220 informs the user that he/she will be inputting a value from the interface. Line 230 initiates the inputting and line 240 branches program execution to the machine language input routine. Branching to that routine, LD B,40 [LD B,7D] and LD C,93 are set to point to the RAM memory location where the value from the tristate device will temporarily be stored. Note: 4093H [7D93H] is 16531D [32148D]. IN A,01 inputs the tristate value. LD (BC),A stores the value in memory, and RET returns program execution to line 250 of the BASIC program. Line 250 assigns the variable "A" to the value just input from the tristate device, and line 260 prints that value to the screen. This program and the output program above will continue to cycle until a break is encountered. To execute the inputting routine, type the command "RUN 200" and press the ENTER key. Again follow the instructions given on the screen. Each time the ENTER key is pressed, a new value from the tristate device will be printed on the screen. This can be verified by grounding various inputs on the tristate device (pins 3, 4, 7, 8, 13, 14, 17 and 18 on IC2) and observing the changing values printed on the screen.

Conclusion

So there you have it; the hardware, the software, and the list of oddities required to get you started in interfacing your Sinclair computer. The hardware and the software presented in this article cover the basics, and with this information you should be able to add the level of sophistication required to accomplish any interfacing task. Bryan, I realize that I did not answer your question about the VIC-20 EPROM programmer directly, but by using the techniques presented in this article, I think that you can modify the EPROM programmer to be used with your Sinclair color computer. Good luck, and if you have additional questions, drop us a line here at *The Computer Journal*. We will be glad to help. ■

listing 2, continued from page 10

```
1630 DISP   JSR COUT           Display
1640        RTS
1650 STORE  LDA TEMP           Retrieve character
1660        LDY #$00
1670        STA (PTR),Y        Save character in buffer
1680        LDA TEMP
1690        CMP #$E0           Lowercase?
1700        BCC CHARTN         No-display on screen
1710        SEC
1720        SBC #$20           Convert to upper case
1730 CHARTN JSR COUT           Print to screen
1740        INC PTR            Increment buffer counter
1750        LDA PTR
1760        BNE SKIP
1770        INC PTR+1
1780 SKIP   LDA PTR+1
1790        CMP #$90           Buffer full?
1800        BCS FULL           Yes-print warning
1810        RTS                Done
```



The Bookshelf

Soul of CP/M: Using and Modifying CP/M's Internal Features

Teaches you how to modify BIOS, use CP/M system calls in your own programs, and more! Excellent for those who have read *CP/M Primer* or who otherwise understand CP/M's outer-layer utilities. By Mitchell Waite. Approximately 160pages, 8x9½, comb. ©1983. . . . . $18.95

The Programmer's CP/M Handbook

An exhaustive coverage of CP/M 80®, its internal structure and major components is presented. Written for the programmer, this volume includes subroutine examples for each of the CP/M system calls and information on how to customize CP/M – complete with detailed source codes for all examples. A dozen utility programs are shown with heavily annotated C-language source codes. An invaluable and comprehensive tool for the serious programmer. By Andy Johnson-Laird. 750 pages, 7½x9¼, softbound. $21.95

Interfacing to S-100 (IEEE 696) Microcomputers

This book is a must if you want to design a custom interface between an S-100 microcomputer and almost any type of peripheral device. Mechanical and electrical design is covered, along with logical and electrical relationships, bus interconnections and more. By Sol Libes and Mark Garetz. 322 pages, 6½x9¼, softbound. $16.95

Microprocessors for Measurement and Control

You'll learn to design mechanical and process equipment using microprocessor-based "real time" computer systems. This book presents plans for prototype systems which allow even those unfamiliar with machine or assembly language to initiate projects. By D.M. Auslander and P. Sagues. 310 pages, 7 3/8x9 1/4, softbound. $16.95

Understanding Digital Logic Circuits

A working handbook for service technicians and others who need to know more about digital electronics in radio, television, audio, or related areas of electronic troubleshooting and repair. You're given an overview of the anatomy of digital-logic diagrams and introduced to the many commercial IC packages on the market. By Robert G. Middleton, 392 pages, 5½x8½, softbound. . . . . $18.95.

Real Time Programming: Neglected Topics

This book presents an original approach to the terms, skills, and standard hardware devices needed to connect a computer to numerous peripheral devices. It distills technical knowledge used by hobbyists and computer scientists alike to useable, comprehensible methods. It explains such computer and electronics concepts as simple and hierarchical interrupts, ports, PIAs, timers, converters, the sampling theorem, digital filters, closed loop control systems, multiplexing, buses, communication, and distributed computer systems. By Caxton C. Foster. 190 pages, 6¼x9¼, softbound. $9.95

Interfacing Microcomputers to the Real World

Here is a complete guide for using a microcomputer to computerize the home, office, or laboratory. It shows how to design and build the interfaces necessary to connect a microcomputer to real-world devices. With this book, microcomputers can be programmed to provide fast, accurate monitoring and control of virtually all electronic functions – from controlling houselights, thermostats, sensors, and switches, to operating motors, keyboards, and displays. This book is based on both the hardware and software principles of the Z80 microprocessor (found in several minicomputers, Tandy Corporation's famous TRS-80, and others). By Murray Sargent III and Richard Shoemaker. 288 pages, 6¼x9¼, softbound. $15.55

Mastering CP/M

Now you can use CP/M to do more than just copy files. For CP/M users or systems programmers – this book takes up where our CP/M handbook leaves off. It will give you an in-depth understanding of the CP/M modules such as, CCP (Console Command Processor), BIOS (Basic Input/Output System), and BDOS (Basic Disk Operating System). Find out how to: incorporate additional peripherals with your system, use console I/O, use the file control block and much more. This book includes a speical feature – a library of useful macros. A comprehensive set of appendices is included as a practical reference tool. Take advantage of the versatility of your operating system! By Alan R. Miller. 398 pages, 6"x9", softbound. $16.95

FORTH Tools, Volume One

FORTH Tools is a comprehensive introduction to the new international FORTH-83 Standard and all its extensions. It gives careful treatment to the CREATE-DOES construct, which is used to extend the language through new classes of intelligent data structures. FORTH Tools gives the reader an in-depth view of input and output, from reading the input stream to writing a simple mailing list program. Each topic is presented with practical examples and numerous illustrations. Problems (and solutions) are provided at the end of each chapter. FORTH Tools is the required textbook for the UCLA and IC Berkeley extension courses on FORTH. By Anita Anderson and Martin Tracy. 218 pages, 5¼x8¼, softbound. $20.00

TTL Cookbook

Popular Sams author Dan Lancaster gives you a complete look at TTL logic circuits, the most inexpensive, most widely applicable form of electronic logic. In no-nonsense language, he spells out just what TTL is, how it works, and how you can use it. Many practical TTL applications are examined, including digital counters, electronic stopwatches, digital voltmeters, and digital tachometers. By Don Lancaster. 336 pages, 5½x8½, soft. ©1974. . . . . $12.95

The Computer Journal

PO Box 1697 Kalispell, MT 59903

Order Date ____________________

Print Name ____________________

Address ____________________

City ____________ State ________ Zip ________

☐Check ☐Mastercard ☐Visa

Card No. ____________ Expires ________

Signature for Charge ____________________

Qty	Title	Price	Total

Shipping charges are: $1.00 for the first book, and $.50 for all subsequent books. Please allow 4 weeks for delivery.

Book Total	
Shipping	
TOTAL	

Searching for Useful Information?

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

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

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

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

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

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

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

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

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

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

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

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

Volume 2, Number 8:
- Tricks of the Trade: Installing New I/O Drivers in a BIOS
- Write Your Own Threaded Language, Part Four: Conclusion
- Interfacing Tips and Troubles: Noise Problems, Part Three
- Multi-user: Cables and Topology
- LSTTL Reference Chart

Volume 2, Number 9:
- Controlling the Apple Disk][Stepper Motor
- Interfacing Tips and Troubles: Interfacing the Sinclair Computers, Part One
- RPM vs ZCPR: A Comparison of Two CP/M Enhancements
- AC Circuit Anaysis on a Micro
- BASE: Part One in a Series on How to Design and Write Your Own Database
- Understanding System Design: CPU, Memory, and I/O

The listing above includes only the major articles in each issue. The Computer Journal also contains regular features such as "New Products," "Books of Interest," "The Bookshelf," and "Classified."

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