ISSN # 0748-9331

THE COMPUTER JOURNAL®

For Those Who Interface, Build, and Apply Micros

Issue Number 22 January—February, 1986 $2.50 US

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, Montana
59912
406-257-9119

Editor/Publisher
Art Carlson

Production Assistant
Judie Overbeek

Circulation
Donna Carlson

Contributing Editors
Neil Bungard
C.Thomas Hilton
Jerry Houston
Bill Kibler
Rick Lehrbaum

The Computer Journal® is a bimonthly magazine for those who interface, build, and apply microcomputers.

The subscription rate is $14 for one year (6 issues), or $24 for two years (12 issues) in the U.S. Foreign rates on request.

To indicate a change of address, please send your old label and new address.

Postmaster: Send address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912.

Address all editorial, advertising and subscription inquiries to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912.

Editor's Page

Restating Our Objectives

When I started TCJ, I wanted to stress the fact that we would not be publishing page after page of product reviews on the latest spreadsheets and appliance-type office systems. My intention was to indicate that we would cover subjects of interest to those who had to implement and interface the systems, but not for the end-user who only wanted an appliance-type machine and who was not interested in the how and why of making the computer work. But, as pointed out in Wilkinson's letter in this issue, it sounded like we were ONLY interested in measurement and control.

This is definitely NOT the case! What we are interested in is understanding the hardware and software so that we can make the computer do whatever it is that we want it to do. I don't want a computer in a sealed box with canned programs that does something the way some 'experts' decided that I should do it. I want to tear into it and make it do what I want, the way I want to do it. For my applications this involves hardware construction, programming, and a lot of interfacing to physical devices. One of the big stumbling blocks has been the necessity of working around the limitations of an operating system—but with Hilton's series starting in this issue that problem will also be solved. I'll talk more about that a little later.

With this issue we are starting sections on the SCSI interface, programming in C, writing your own operating system, Ampro SBC user's support, and continuing the Turbo Pascal series with an article on ISAM files, plus The Computer Corner and other goodies. We have a number of excellent articles on hand and in progress for future issues, so our coverage will expand and improve.

Write Your Own Operating System

As Tom Hilton points out in his letter, what the user sees is the application program and not the operating system. In a well written program the user should never see the system prompt or have to deal with the system— if he does he should criticize the program and not the system. It is only those of us who program and implement systems who should have to deal with the operating system directly.

I have had a love/hate relationship with CP/M because it does some things so well while doing other things very poorly. I finally got ZCPR3 running (it came installed on the Ampro 122), and it eliminates most of CP/M's limitations. But I won't be satisfied until I fully understand exactly what the OS is doing and can modify it to do what I want. In order to control the computer we have to be able to control the OS, and Hilton's series on NEW-DOS starting in this issue is exactly what I was looking for.

"…. SCSI interface programming in C, writing your own operating system, AMPRO SBC user's support …."

Even those who use other systems should follow the series in order to understand what a system does—and they can envy CP/M users because we can modify our system. By the end of the series we will be able to write our own OS with the features we want, without paying any license fees or depending on an unresponsive company for support. TCJ will organize a user group to support NEW-DOS so that we can help each other. We realize that there are other systems, and MS-DOS may be the best choice for some uses, but a disk-based OS which we can create and modify gives us great opportunity to learn and grow.

(Continued on page 49)

Letters From Our Readers

Using C

I read with interest your editorial in Issue 21. It is possible to get most of the things you wish using some C compilers (plus add-ons) but unfortunately, the BDS C compiler does not support them. Many C compilers support the pre-processor directives #asm and #endasm. The use of these directives allows the programmers to include in-line assembler code in his or her C code. When the preprocessor finds these directives, the code is marked so that it will not be optimized during the optimization pass on the compiler (if this pass is present). This is a great improvement over the method which you have to use with BDS C. In terms of "flash compiling", it is my impression that this could only be done using a single pass C compiler (with, of course, a built in editor). The problem with single pass C compilers is that they are sensitive to the order in which #define's are listed in the code. If several #define statements refer to each other and are incorrectly ordered, a single pass compiler could end up with unresolved references, which would halt compilation. A good alternative would be the use of a C interpreter, several of which are now on the market (although I believe all of them are targeted at the IBM PC environment, rather than CP). These generally have built in editors which will point back to the source code if syntax errors are found (similar to Turbo Pascal's editor). They are also generally syntactically compatible with major PC C compilers (Lattice and Microsoft compilers and occasionally some others). The problem here is that you are set back the price of the interpreter (which can range from $100.00 to $500.00), as well as the cost of your compiler (which, for Lattice and Microsoft are not cheap).

Don Howes
Pullman, WA

Is CP/M Dead?

Is CP/M dead? Are hammers dead? Are nails dead? Is cooked food a thing of the past? There are some questions we writers should not dignify with an answer, and would not, if it weren't so much fun!

To be quite honest, I never saw pure, virgin, CP/M until I just had to see what it looked like. Pure CP/M is an option on the Ampro Series 100 systems. When I first booted it, I thought my terminal program had gone into high orbit (again). The point is, who uses pure CP/M? If CP/M is dead, then it has been dead for a long time, and will be dead for decades to come.

It is only we masochistic system programmers who ever see CP/M. We are the only ones who appreciate it, albeit in a love/hate relationship. What the user sees, and communicates with is the Console Command Processor, (CPP). While it is a part of the standard CP/M Disk Operating System, (DOS), it is seldom allowed to remain as Digital Reseach intended it. Perhaps the greatest gift to computerdom was Richard Conn's ZCPR, a CPP enhancement.

I began computing in an industrial world. The king of the space program was the RCA CDP1802 microprocessor. This was a CMOS chip, and has been available literally for decades. People are just now discovering CMOS technologies, though few really understand them. Hence, my perspective is that of machine intellect, robotics, satellites, and deep space probes, where the machine must fend for itself.

My world does not generally involve spiffy graphics displays, though I appreciate that type of programming genius. Nor does it generally involve complex mathematical process. I am, after all just a lowly chip mechanic.

As I entered into the world of consumer computers I was spellbound by all the nifty features of CP/M machines. After about an hour I had to say to myself, "this is neat, but how do I get to the system, and will all of this spiff get in my way?" With the exception of the Ampro Z-80 machines, all that spiff did get in the way. Ampro allows me, the operator/developer, to decide how much spiff I want.

These days I design systems for the disabled community. These are challenges greater than the space program, and demand the highest technology. I need a system that I can tailor to the specific needs of the individual. Not only must I be sure that the system may be used by the individual with ease, but it must be reliable.

Now then, for the casual operator, CP/M does not present a great deal of flash, nor pretty noises. It must be remembered that CP/M was designed as a business workhorse. For a person who just operates a computer, or perhaps plays with an assembler, CP/M type systems have little to offer. However, it is this type of computerist that is the most vocal in what has been termed, "The DOS WARS." When running an applications program, the operator never sees the operating system, only the applications program. These vocalists are judging the performance of these programs, not the operating systems. But, as I think of it, these people do not read TCJ either, but dough-files, or BC Weekly.

The best thing about CP/M type hardware, as opposed to the CP/M operating system, is that there is a reasonably standard way of doing things. If I didn't have a fully debugged set of routines to handle the disks, and terminal, I'd have to write them. That is just the reality of computing.

In my work I use equipment designed to run CP/M. There are a number of reasons for this choice. First, is the price. Thousands of people are maimed daily in automobile mishaps. When a disability strikes money is an issue. In other fields, it is the same concept. The boss wants the lowest cost technology that will do the job. A large amount of CP/M hardware is unuseable. Were it not for the way the Ampro systems allow you to modify the operating system, I would have designed a similar system to run MY operating system; not CP/M, an operating system to do a specific task. CP/M is no longer a mystery. Those who know hardware generate operating systems for special tasks. You can't do that with the PC clones without new ROMs, higher end costs, and complexity. As a matter of fact, TCJ will be doing a series on how to design your own operating system for Z-80 technologies. Why buy a DOS when you can write your own to do what you want it to?

As the American market turns further towards the 16, and 32 bit technologies, the Japanese will invade the CP/M world. They may call it something else, but has anyone been noticing the number of very low cost Japanese CP/M systems on the Market? As with most things the Japanese will take our left-overs, perfect them, and sell them back to us. This is like selling an Eskimo snowballs. But, they keep on doing it, and we keep going for it. Take the HD64180 superchip. It is nothing but a souped up Z-80. Some say its only real advantage is being able to access more memory. Fine, I can live whith that easily.

The bottom line is that, from the machine level, or "the other side of the screen," as I like to say, there just are not systems as easy to work with for the price asked.

Creative Computing published a list of "The World's Worst Computers," in an article by that name. The IBM PC headed this list, with PC clones coming in second, and the IBM PC JR. taking third. Comments ranged from "user hostile," to "an uninspired design..." I happen to agree with all of the negative comments about 16 bit machines, and agree with but two positive comments: they do crunch numbers, and they do have neat graphics. The prices being asked for these technologies are near criminal. The sophistication, for most board level projects, is like putting airbrakes on a turtle.

Now what would I want for Christmas, had I all the money I should want to spend on computers? A Sanyo MBC-775, (Japanese portable PC clone), the Borland "Turbo Jumbo Pack," and a program to do a "school newspaper." Now what would I do with this $5000 Christmas package? Why develop applications for the Ampro Z-80 Little Boards, and The Little Board '186! I need a compact portable for many personal and professional functions, to use as a terminal, and a 16 bit machine to run the full Borland Package.

Is this hyprcritical, or at best treason? I don't think so. Tools are tools. My personal, and applications programming productivity would be increased 4000% with the Borland package, especially when applied to the Little Board '186, if I could afford all this. Why a Japanese clone instead of buying American? The Sanyo, in my opinion is a superior implement, cheaper than any American model, and has a color monitor built-in.

The key concept is that tools are tools. For most of my work the Z-80 systems are the best for the job, and of the lowest cost. I can work with them. Just by the way they are designed, and constructed the IBM machines are not all that great for board level systems, except for building super computers. Their cost is nearly double that of the Z-80 systems. From an applications programming perspective, however, more is available for the IBM types, and I want the full Borland Toolbox series. So, on the "operator's side of the screen," I'd like the Sanyo Clone, for personal and program development. For the mainstay of my work, however, I prefer the single board Z-80 systems, especially the Ampro Little Board. Many clients, knowing no better, want the IBM systems, just because their neighbor, the used car salesman, said his brother-in-law's sister's cousin heard they were good. The key point is that each are tools with specific functions, and people who don't really know better are demanding systems that are IBM compatible.

Is CP/M dead? Perhaps, but the systems that run CP/M, and will tolerate a user's version of the DOS will be with us for decades to come. Is CP/M dead? Who cares, as long as the hardware that will run it keeps getting cheaper? The only problem with hardware that will run CP/M, and CP/M itself, is that the skill level of users is dropping, and the desire to learn is nonexistent. The popular trend is to serve the computer, not to have the computer to serve you. When viewed in this context, the entire issue is stupid, in my opinion.

Tom Hilton

More on Soldering

I was reading Mr. O'Connor's article about soldering in issue #20, and I'd like to mention a couple of points he missed. It's an excellent piece, with more useful information about soldering (and clearer explanations of what's going on) than I've ever seen in one place before; but ever so, there are a few more details that a potential kit builder might find useful.

For example, iron-plated tips for the soldering iron—why are they better than plain copper? They're a lot more expensive—are they worth it? Yes, because they don't have to be cleaned, scraped and re-tinned nearly as often. In fact, Mr. O'Connor hardly mentioned tinning the tip at all—and that can make a big difference in the efficiency of heat transfer.

About flux—first, the name. Mr. O'Connor's explanation was excellent, but he left out one important point: the reason it's called "flux" is because it makes the solder flow over the surfaces being soldered. Soldering flux is primarily a "wetting agent" for metals. Just as soap or detergent helps water to coat and cover a surface, instead of clumping into little beads and droplets, flux helps the solder to make a thin, penetrating film over the metals being soldered: this improves heat transfer during soldering, and provides for more and better metal-to-metal contact (which means better electrical conduction).

Rosi-core solder is very useful—but don't sneer at paste or liquid rosin fluxes (NOT acid fluxes), old-fashioned though they may be. For an experiment, try tinning the end of a piece of stranded wire (coating it with solder, to make it more manageable before connecting it to a terminal in a tight place). First the usual way, by simply heating the wire and applying rosin core solder; then do the same again, but put a little paste of liquid flux on the wire before you start. You'll find that the added flux makes for a much neater and quicker job, and also requires less heat from the iron.

Only a teeny tiny bit of flux is required (more will just make a mess); but the difference it makes is tremendous!

One final, but very important point: about why (and when) you need to clean the flux residue off afterward. They tell you that rosin is non-conductive...but that's not quite true. It's non-conductive, compared to a piece of wire; but compared to a ten-megohm resistor, it conducts quite nicely, thank you!

For instance, if you're trying to get a long time delay in a 555 timing circuit, by using a fairly small capacitor and a very high resistance, you had better clean off all the flux when you finish—or the conductivity of that "non-conducting" flux may upset your calculations considerably. Or if you're working with CMOS ICs, you may find that current leakage from one of the power suply pins to an adjacent input pin, through un-removed rosin flux residue, can shut down the circuit entirely!

In short, whenever you're working with high resistances or low currents, you had better clean all the old flux off the board before you power it up, or you may find a nasty bug in your circuit!

Jock Root
L.A., CA

FORTH

Bill Kibler: I have just read your Computer Corner column in issue #17 and I am interested in your idea of building a Z-80 FORTH unit. I am currently learning 64 FORTH from HES on my Commodore 64 and would like to put FORTH on a Z-80 board that I have already built.

I became interested in FORTH because I work with elevators which are rapidly becoming computerized and FORTH seems perfectly suited to this field.

Any columns on Z-80 FORTH would be greatly appreciated. As usual, you and everyone at The Computer Journal are doing a great job.

G.K.
New York

Data Acquisition and Control

I enjoy reading your informative journal. Mr Jerry Houston's article on analog data acquisition and control systems was especially interesting to me.

I wonder if he and/or others might be interested in elaborating on actual applications of ADC units, such as those mentioned in his article. As you know, some of these devices are relatively inexpensive and most can be interfaced through RS-232 with a number of different micros. In my opinion, they present a unique opportunity for us who interface micros with the real world. I am sure that a number of your current readers and potential new subscribers would be interested in this area.

Thanks for your efforts on editing and publishing The Computer Journal.

Matthew K. Rogoyski, Ph.D.
Hotchkiss, CO

(Continued on page 38)

NEW-DOS Write Your Own Operating System

Part 1: The Console Command Processor

by C. Thomas Hilton

We Listen to Our Readers

A number of professional readers have written to request more technically based articles. While these readers may be professionals, who use computers in their work, they may not always be computer professionals. Most often they use their systems to interface an experiment, or just require more control over their equipment.

We have all heard of the "DOS WARS." Of those professional computer users who have made comment, most have stated that the 16 bit systems "have more features." Most however resent the higher cost of 16 bit systems.

The point of issue is not which operating system is the best. The user does not communicate with the operating system directly, but through a Console Command Processor, (CCP), which translates human commands into computer based functions. Hence, the number of "features" seen is a product of the CCP, not the operating system.

In this series we will be discussing how to modify your system to meet your specific needs, or desires. Because of their price, and versatility, we will be targeting the Ampro LITTLE BOARD® series of Z-80® machines. Users of other, or existing, systems may follow these discussions and implement the projects. The only project series that non-Ampro users will not be able to implement will be the custom Basic In/Out System, (BIOS), which is hardware specific. The BIOS we discuss may, however, serve as a model for implementation on non-Ampro systems.

We will open our discussion with the CP/M® type CCP. Some users have either purchased the Ampro "FRIENDLY"® operating environment, or purchased an Ampro Series 100 system which has ZCPR3 installed for use. Others may have purchased the "Z" System® from Echelon. We will term the ZCPR3 systems the "top of the line," as far as enhancements are concerned. On the other end of the scale is standard CP/M, which is available as an option with all Ampro systems. It is with the stock CP/M system that we will begin our discussion. In this series we will develop a system whose function is mid-way between ZCPR3 and standard CP/M. The best part of our system will be that it is ours, not someone else's. Our system will require neither extra system memory, nor support files on disk for proper functioning.

The Standard CP/M System - Structure and Terminology

Figure 1 shows a standard CP/M memory map. (A memory map shows where various portions of the system are located in memory.) In hexadecimal, ("hex") notation, as applied to an 8 bit system, memory locations form a four character representation. In hex, memory is defined as a series of "lines," and "pages." A line is a single digit code, or "byte." A "page" of memory consists of 256 lines of code. The number of lines being referenced occupies the two "least significant digits" of the hex representation. Hex is read from right to left. The number of pages is represented by the left-most two digits, or 'most significant" digits. Each pair of digits is capable of a single byte value, of 256 elements. Each numeric representation begins with the number zero.

```
 ---------------------------------------
:                                       : FFFFH
:  BASIC IN/OUT SYSTEM  (BIOS)          :
:---------------------------------------: EE00H  __________My System
: STANDARD CP/M DISK OPERATING SYSTEM   :
:---------------------------------------: E000H  __________My System
: Console Command Processor  (CCP)      :
:---------------------------------------: D800H  __________My System
: Transient Program (User Program)      : D7FFH  (End of TPA)
: (Dependent Upon Amount of Memory)     :
:                                       :
:---------------------------------------: 0100H  (Start of TPA)
:  BIOS and CP/M Work Area, Page Zero   :
:---------------------------------------: 0000H  (Start Of Memory)
```

Figure 1: CP/M memory map.

Figure 2 shows the basic hexadecimal number system. For most people it is easier NOT to attempt to translate the hex system into decimal. The key thought is that, instead of ten fingers, we now have 16 fingers. The number system works the same as the more familiar base ten. We start at number zero and count to 15, or "F" before starting a new, left-most number column.

```
    ___ lines or bytes
  ::
0001h   equals the first line, or byte of page zero
::__ pages


The standard hex progression is as follows:

0 = 0       8 = 8
1 = 1       9 = 9
2 = 2      10 = A
3 = 3      11 = B      16 = 10
4 = 4      12 = C
5 = 5      13 = D
6 = 6      14 = E
7 = 7      15 = F
```

Figure 2: The HEX number system.

This brief introduction to the number system is inadequate, but will have to be enough. Supplemental reading is suggested for all parts of this series.

The structure of the operating system is very straightforward when related to the memory map in Figure 1. In "the attic," of our microworld, or top of memory, is the BIOS. The BIOS is responsible for all hardware dependent functions. That is to say that the BIOS handles all transfer of data, in and out, on a byte by byte, or character by character basis. As the CP/M type operating systems may be run on any number of different computers some means of compatibility was needed. The BIOS begins with a 'jump table" to each of its internal functions. Each of these functions is implemented in the fashion required by the hardware specific to the concerned computer. We will deal specifically with the BIOS in a future portion of this series.

Underneath the BIOS is the Disk Operating System, or 'DOS.'' The DOS may be said to handle the system in a form which is system independent. It does not know, nor care, what the system specific hardware is, nor looks like. All the DOS knows is whom to 'call'' to perform a specific function. It is dependent upon the BIOS for all of its tasks. The BIOS, again, works only in the most primitive terms. The DOS, however, works in multiples of these primitives to accomplish a task. As an example, to print a string of text the DOS sends the string to the BIOS a character at a time, who sends each individual character to the terminal, or other hardware. To assure that a program will run on other computers programmers perform tasks by making function calls, or requests, to the DOS, instead of the BIOS. As the DOS is the same for all machines, though the bios is not, compatibility between differing systems is had. A program may, however, make function requests of the BIOS by calling the jump table at the start of the BIOS. The jump table must remain in the same relative position for all systems. The DOS assumes function addresses to be in a given sequence, relative to a given starting address.

Beneath the DOS is the CCP. It is the job of the CCP to interpret the human's commands and perform simple tasks with files stored upon a disk. The standard CCP is a very simple minded fellow, at best. We will add a number of functions to the CCP. Our main focus will be, however, to show you how to add your own special commands and functions.

In "the basement" of our system is an area reserved for use by the BIOS, DOS, and CCP. This area is the first page of memory, and is called, "page zero." We will cover the use of this basement area in great detail, but later. Finding a place to begin is always the most difficult part of starting any project. And, as I often say, no matter where I begin, I should have covered something else first. With this in mind, let us begin.

Project Support Disk

Because many are not familiar with assembly language programming, nor the structure of the world on the other side of the screen, I have prepared a special assembler. This assembler has proven itself to be of value to the beginning assembly language programmer. We have discussed briefly bytes, lines, pages, and other terms. There are even more terms to learn. For example, a "word" is a 16 bit address constructed from two bytes. The hardest thing for a beginning assembly language programmer to understand is where a byte must be used, and where a word must be used. This is especially true when they are often represented by the same series of symbols such as DEFB, DB, DEFW, or DW. In the assembler we will be using a byte is called a "BYTE," and a word is called a "WORD." I realize this is at best treason, to the pundits of tradition, but it is easy to work with.

Additionally, which does one use when representing string, or character data? You guessed it, "DATA." The common approach to Z-80 system programming is the use of the 8080 assembler that came with your system. 8080 mnemonics have many and varied forms, which are confusing at best. We will use a Z-80 assembler for a Z-80 system, another attack against tradition. Most all of the instructions have a simple form, with only two variations, a marked improvement.

This assembler, originally written by Pat Crowe of England, is provided for this project as a user disk. It is available from TCJ at a very reasonable cost, as it is a public domain program. The source code file can be assembled with itself. No other assembler is needed for this project. Additionally, all the source code files to do all of the projects we will be discussing are provided in ready to modify and assemble formats.

Due to the fact that Ampro distributes the T/MAKER system, and bundles it with some system configurations, all source code files are presented in T/MAKER format. Word Star, and other editors may read these files without modification as the files are pure ASCII code.

Getting Started, (Finally)

The first thing that we must do is configure a system for our use. We do not want all of the spiff and reservations of memory space used by ZCPR3. Our first project will be to install standard CP/M in our systems, at the maximum possible memory image. Place a check mark in the box provided before each step. This is to assure that all steps are performed in the proper sequence.

☐ 1. Format and SYSGEN a blank disk.

☐ 2. Place the following programs on your fresh disk:

a. MOVCPM.COM
b. SYSGEN.COM
c. DDT.COM
d. CROWE.COM
e. CCP.CRW
f. CCPA.CRW
g. CCPB.CRW
h. STATUS.COM

☐ 3. Place our working disk in drive 'A' and boot it.

☐ 4. Your system should send you to the CP/M command line and issue the "A0>" prompt. When the prompt appears enter:

```
A0>MOVCPM 61 *
```

What we have done is told MOVCPM, (do not use the ZMOVCPM program), to construct a CP/M system that is 61K in size. The "*" tells it to leave this new image in memory.

MOVCPM will, when it has done its work, instruct you as to the option to SYSGEN or SAVE the memory image. When the prompt reappears enter:

```
A0>SAVE 41 MYSYS.COM
```

☐ 5. When the A0> prompt returns, enter:

```
A0>SYSGEN MYSYS.COM
```

SYSGEN will then ask for a destination drive, enter "A" to place the standard CP/M image on drive 'A.' The next time SYSGEN asks for a destination drive, answer with only a RETURN.

□ 6. "Flip" the reset button. Your system should come back with the prompt: "A>'. Nothing is wrong, that is just standard CP/M!

□ 7. Now enter:

A>STATUS

and make note of the positions reported for the locations of the BIOS, DOS, and CCP on the memory map in Figure 1, if they are different than those shown.

It is very important to accurately determine the location of the CCP. If the new CCP does not begin in the same place as the old one, then the system will not function. Each portion assumes another portion to be in a given spot relative to itself.

□ 8. If you do not have my source disks and assembler, then enter the published source code, and convert the data representations to those of your assembler.

Be sure to set the equate to be used by the ORG statement to the proper value in file CCP.CRW, if the location of the CCP is other than D800H in your system.

If you do have the source disks you may now assemble the source code by entering:

A>CROWE CROWE.AAZ

This will cause the assembler to look for the source file on drive 'A,' place the ".HEX" file on drive 'A,' and omit a ".PRN" file. We do not want a PRN file due to the size of the files being assembled.

□ 9. When the source file has been assembled without error enter:

A>DDT MYSYS.COM

DDT will sign on and present its own prompt, "-', then enter:

-ICCP.HEX
-R3180

when the "-" prompt returns, enter

-G0

which will return you to the "A>" prompt. At this point enter:

A>SAVE 41 NEW.COM
A>SYSGEN NEW.COM

and place the new operating system on the 'A' drive as was done in step 5, above. Reset the system and the command prompt of "A0>" should be returned.

For a summary of your new system commands enter:

A0>HELP

and a help screen will appear.

A Bit Of Digression

An Overview of CCP Commands, Old And New

The standard CP/M CCP has the following commands:

a. DIR which returns a directory
b. TYPE which prints a text file
c. REN which renames a single file
d. ERA which deletes files
e. SAVE which we have already used
f. USER which changes subdirectories
g. ^P (control-P) which sends whatever is sent to the screen to the printer
h. ^C which resets the system
i. ^S which stops screen display when using the TYPE function

It would be best if you referred to your system manual to assure that you understand what these commands are, and do. Our new CCP has all of these commands, and more, some with expanded functions. The TYPE command, which is called "READ," now has a built-in single screen paging function. The basic command set is as follows:

a. DIR

This command functions in the same manner as the standard CP/M DIR command. The enhanced variations available are:

DIR *.* S will display files with the "SYSTEM" attribute

DIR *.* U will display files of any attribute

b. READ

This command functions in the same manner as the standard TYPE function, but has a built-in paging routine. That is, it will display 22 lines of text from a text file and stop, awaiting any keypress by the operator. This paging function may be disabled by the suffix "N," for no paging enter:

READ MYFILE.TXT N

c. LIST

The LIST function is a relative of the READ command. It reads a text file from the disk and sends it to the printer, instead of the terminal. No paging options are currently available for this command.

LIST MYFILE.TXT

d. REN

The REN, or rename command, renames one file at a time, and has the syntax of:

REN NEW.FIL=OLD.FIL

where NEW.FIL is the name that OLD.FIL is to be given.

e. ERA

ERA deletes files individually, or en masse. Wild-cards may be used. If the wild-card of *.* is used the system will ask you if you really want to erase the entire directory. Forms may include:

```
ERA MYFILE.TXT
ERA MYFILE.*
ERA *.TXT
ERA *.*
ERA MY????.*
```

and so forth.

f. SAVE

We have already used the SAVE command. However, DDT and the rest of microworld speaks hex. This has meant that we have had to translate the number of pages to save into decimal to use this command. This function now allows the option of specifying the number of pages to be saved in a hexadecimal number.

```
SAVE 41 MYFILE.COM
SAVE 12H HISFILE.COM
```

To inform the CCP that the number to be worked is in hex, the "H" suffix is required. If a file name specified already exists the system will ask you if you want to overwrite it.

g. USER

This command changes the currently assigned "USER AREA," or subdirectory. It has the form of:

```
USER 12
USER 8
```

which would be returned as:

```
A12>
A8>
```

in the prompt. The concept of a "USER AREA" is at best false, as there are no such "areas" on the disk. All this command does is assign special directory numbers. Without this ability to assign special directories large systems would have many screens of directory listings in response to the DIR command.

h.PATH

The PATH command modifies the search path the CCP uses in attempting to locate a file for us. In the standard CCP there is no search path. In this project's CCP the system will search the current drive, and current directory, then the current drive and the directory assigned by the PATH command, (normally zero), then drive 'A' current user, then drive 'A' PATH assigned directory before complaining that it cannot find the file. The search path is not as extensive as ZCPR3's, but doesn't require any extra memory or disk support files.

i. JUMP

The JUMP command allows the programmer to jump to any position in memory and execute a program at that location. The syntax for this command is:

```
JUMP EE00H
```

All address references are assumed to be in hex.

j. RUN

The RUN command will run any program which is currently in memory, without reloading it from disk.

```
        SYSGEN MYSYS.COM
(exit)
        DIR
        RUN
(system prompts:)
        Destination Drive?
```

k. DO

The DO function is similar to the RUN command, but allows the passing of parameters to the program residing in memory.

```
        STAT
(exit)
        DO *.* $SYS     (which sets all files with the
system attribute)
```

l. LOAD

The load command loads a named file into a given address:

```
        LOAD 2345H MYFILE.COM
```

in which the load address must be in hex, and the file must be assembled, or compiled, to run at the address specified.

m. HELP

The help command displays a user created HELP file, one screen at a time. The help file may be created by any text editor. The Help file must be named:

```
        SYS.HLP
```

and must take into consideration the paging effect of the READ command, which is used to print the HELP file. In general use the SYS.HLP file should contain an index of other help files. Once this Index is displayed the user may enter the command:

```
        A0>READ HELP.ERA
or
        A0>READ STORY.TXT
```

Only the name of the system base help file is predefined. The HELP command may also be used in either the cold, or warm boot autocommand structure for power-up and reset screens or menus.

n. ^P, ^C, and ^S

These three control codes operate in the standard manner.

We will deal with all of the commands in greater detail as we examine the CCP source code. I wanted to give you a basic overview of what we will be doing in case you wanted to order the source disks before we got too deeply involved. Of course these commands are only those which are installed for demonstration purposes. At the end of this series you will be able to design your own command structures for the work that you do.

The CCP, A Detailed Look

The CCP we will be examining has a long and varied history. If one had to trace its history it could probably be said that the original author was Richard Conn. The version shown here has been assembled from a number of "ZCPR" type CCP implements written for four or five different assemblers, and untold numbers of machines.

This CCP, as with all of its forefathers is a public domain program. All persons who claim copyrights to any given version do so to assure that the program will remain in the public domain, and will not be used for direct commercial purpose.

In Figure 3 we begin to look at the source code, as written for the CROWE Z-80 assembler. As with most assembler files we begin with a long list of equates for terms we will be using later in the program. As we progress I will excerpt from the source code so we do not have to keep turning pages to see what is being discussed. For now, make special note of Figure 3.

Note that the first line of our file holds a line of comments which are described as T/MAKER Tab Settings. The T/Maker editor is, again, supplied with many Ampro systems as the only alternative to the CP/M ED.COM program. Many users therefore will not have a copy of WordStar or other editor. The target system in this discussion is the Ampro Series 100. It therefore makes little sense to create files with an editor that is not commonly available to Ampro users. A T/Maker file may be read by any text editor as it produces a "pure ASCII code." This means that there are no flipped bits nor control codes hidden in the text file. T/Maker does have a quirk, however. There is a 300 character "first line" in the text file where tab settings are stored. The maximum length a line may assume is 300 characters, in the CP/M versions. This tab line gives most languages and assemblers a fit! When using T/Maker be sure to save your files without tabs. This is done by entering:

WHAT NEXT? NOTABS SAVE

In the alternative, and what I have done, is use the T/MODIFY program to change the default setting of my T/Maker system. When asked if tabs should be saved with the file, answer "NO." The top tab line in our program is to set the internal tab settings for assembly language programming. All that is required to set these tabs is to place the cursor on this tab line and enter:

ESC
S
^I or TAB key

all tabs represented by this model line will be installed in the system for as long as the power is on.

```
NO      EQU     0
YES     EQU     0FFH            ;conditional logic boolean declarations
```

In the two lines of code above we define the states of two boolean symbols. A boolean symbol is a logic operator which may have either of two states, either "true," or "false." A negative logic state is, in this program, called "NO," and a positive logic state "YES." These boolean elements are used for triggering conditional assemblies.

A conditional assembly is a section of program that either will be assembled, or will not be assembled, depending upon the result of an "IF" evaluation. We have a number of conditional assembly sections in this CCP. Some commands are considered hazardous in a Remote CP/M system, (RPCM), as callers may try to "crash our system," causing damage to files, and possibly equipment. For our personal use we will not inhibit these options.

```
CR      EQU     0DH             ;character: carriage return
LF      EQU     0AH             ;character: line feed
TAB     EQU     09H             ;character: tab
ESC     EQU     1BH             ;character: escape
CTRLC   EQU     03H             ;character: control-c
WBOOT   EQU     00H             ;cp/m warm boot address
UDFLAG  EQU     04H             ;user num in high nybble, disk in low
BDOS    EQU     05H             ;bdos function call entry pt
BIOS    EQU     0EE00H          ;industrial bios location
TFCB    EQU     5CH             ;default fcb buffer
TBUFF   EQU     80H             ;default disk i/o buffer
TPA     EQU     0100H           ;base of tpa
```

In the equates above we assign symbols for frequently used codes. The first five equates are for control characters we will recognize in the program. It is far easier to remember a symbol than the actual numeric codes. The next seven equates, starting at "WBOOT" will eventually tell the program where to go to perform a function, or find data.

```
MSIZE   EQU     61              ;ampro cp/m size
AMPCCP  EQU     0D800H          ;ccp location for ampro series 100
CBBUFF  EQU     BIOS+62H
```

The equates above give us a picture of what our memory map is to look like. Remember that we created a memory image of a 61K CP/M system early in our discussion. MSIZE equates to the size of the CP/M system we created.

Equate AMPCCP is set to D800H, which is where, in memory, this program is to begin. Note that there is a leading zero in the equate. Any time a hex number begins with a letter we have to place a leading zero. If we did not do this the assembler would not recognize the hex representation as a number, but as some kind of strange human symbol. All numbers must look like a number in some way.

```
RCPM    EQU     NO              ;set to true if ccp is for a BBS system
NLINES  EQU     24              ;number of lines on crt screen
PGDFLG  EQU     'N'             ;this flag reverses the default effect
MAXUSR  EQU     15              ;maximum user number accessable
SYSFLG  EQU     'U'             ;for dir command: list $sys and $dir
SOFLG   EQU     'S'             ;for dir command: list $sys files only
DEFUSR  EQU     0               ;default user number for com files
```

In the next series of equates, which have been simplified from many versions of the CCP, we tell the program what options we want to use, and what our terminal "looks like." In that our first version of this CCP will be for internal use, we have set the RCPM equate to a negative state. That is, we are telling the system that we do not want a secure remotely operated CP/M system.

We then answer questions, in compuspeak, about the maximum number of lines on our screen, (24), and the symbol we wish to use to disable the page scroll feature. These two equates, therefore, relate to the READ command of the CCP.

```
;--------------------------------------------------------------------------
;       ;       ;            ;            ;
; Tab Settings For T/Maker Editor
;
;
;                                   Hermit  Software's
;
;                              Modified  CROWE  Assembler
;                                   Source Code File
;
;                              (c) 1985 C.  Thomas Hilton
;
;
;
;       Primary
;       Hardware: Ampro Series 100, 1A CPU
;                 (Original Little Board)
;       System:   CP/M 2.2
;                 (Ampro Standard Version)
;
;       Function: A True Z-80 Replacement Console Command Processor To:
;
;                 1. Restore AUTOCOMMAND Function To Ampro 61K CP/M
;                 2. Enhance Standard CP/M Console Functions
;
;
;       Index:
;               CCP.CRW         CROWE Source Code File
;               CCPA.CRW        CROWE Chain File
;               CCPB.CRW        CROWE Chain FIle
;
        LIST
        TITLE   'Ampro Custom CCP Base File'
        NLIST
;
;             ___ terminal and 'type' customization equates ___
;
NO      EQU     0
YES     EQU     0FFH            ;conditional logic boolean declarations
CR      EQU     0DH             ;character: carriage return
LF      EQU     0AH             ;character: line feed
TAB     EQU     09H             ;character: tab
ESC     EQU     1BH             ;character: escape
CTRLC   EQU     03H             ;character: control-c
WBOOT   EQU     00H             ;cp/m warm boot address
UDFLAG  EQU     04H             ;user num in high nybble, disk in low
BDOS    EQU     05H             ;bdos function call entry pt
BIOS    EQU     0EE00H          ;industrial bios location
TFCB    EQU     5CH             ;default fcb buffer
TBUFF   EQU     80H             ;default disk i/o buffer
TPA     EQU     0100H           ;base of tpa
MSIZE   EQU     61              ;ampro cp/m size
AMPCCP  EQU     0D800H          ;ccp location for ampro series 100
CBBUFF  EQU     BIOS+62H
;
RCPM    EQU     NO              ;set to true if ccp is for a BBS system
NLINES  EQU     24              ;number of lines on crt screen
PGDFLG  EQU     'N'             ;this flag reverses the default effect
MAXUSR  EQU     15              ;maximum user number accessable
SYSFLG  EQU     'U'             ;for dir command: list $sys and $dir
SOFLG   EQU     'S'             ;for dir command: list $sys files only
DEFUSR  EQU     0               ;default user number for com files
;
        ORG     AMPCCP
;
ENTRY:  JP      CCP             ; process potential default command
        JP      CCP1            ; do not process potential default command
```

Figure 3

The MAXUSR equate identifies the legal number of user areas the operator may request in a USER command. The number of user areas in most systems is 16, due to the way that user areas are defined in low memory. Location 0004 contains a single byte which tells us what disk drive we are on, and what the current user number is. The format for this data is:

```
F1H
```

which would indicate that the system was in user area 15, (remember that hex starts with the number zero which is a valid number), and drive 'B' was the currently selected drive.

SYSFLG and SOFLG are options to display files of all attributes, or system files. When given the "SYS," or system attribute, the file will be displayed in a normal directory listing.

The DEFUSR equate is the number which is to be considered the default user, or directory number to be used in a directory search for an operator specified file. This value may be modified, once the system is running, by the PATH command.

```
        ORG     AMPCCP
```

Our ORG statement is set to the value we assigned to the AMPCCP symbol, which is D800H. In a larger program we could present a number of different values to the ORG statement by use of a conditional assembly.


```
        IF      KAYPRO
AMPCCP  EQU     0E000H
        ENDIF
;
        IF      AMPRO
AMPCCP  EQU     0D8000H
        ENDIF
```

A standard CCP has two entry points. The ORG statement defines where the program actually begins as the equate statements have no real meaning to the computer. Equates are just there to make life easier for us humans. Ampro chose not to use the standard means of entry into the CCP. This is primarily because their BIOS does not check to see if there is a valid cold boot, or "autocommand," installed in the position set aside for it. The Ampro BIOS assumes that the CCP, or other program will take care of this matter. In a normal system such a check would be made. If no command was noted for execution upon restart the BIOS would jump to location ENTRY+3. In this way no time would be wasted in determining whether to process this command. Additionally determinations must be made as to when to execute such a command.

This failure to check for the actual presence of a restart command causes the Ampro to always enter the CCP at ENTRY, or D800. We will retain this dual entry feature as other programs may try to use it, being written to be run on a standard CP/M machine. Again, this failure to check for a restart command is why the "autocommand" feature is lost in the Ampro machines when ZCPR3 is not being used. ZCPR3 makes these determinations. We will perform a test for a restart command ourselves, while allowing standard CCP entry points.

To summarize, most systems would enter the CCP at ENTRY+3 if they had no intention of processing a restart command. They would enter the CCP at ENTRY if they did want to supply a restart command to the CCP for processing. Ampro systems always enter at the ENTRY location, whether there is a restart command stored in the BIOS or not. We will have to make up for this oversight in BIOS design to allow the use of the standard Ampro "autocommand," while retaining compatibility with other CP/M systems.

```
ENTRY:  JP      CCP             ; process potential default command
        JP      CCP1            ; do not process potential default command
```

As is noted, the two different entry points are met with unconditional jumps to routines that either will process a restart command, or will not attempt to process a restart command.

Kindly note that our discussion does not follow the physical layout of the source code listing, but rather the logical path of program execution. That is to say that while we will now discuss the routine "CCP" it is not the next entry in the program listing.

```
;
; ccp starting points
;
; start ccp and don't process default command stored
;
CCP1:   XOR     A               ;set no default command
        LD      (CBUFF),A
```

From our previous discussion we learn that if the Ampro BIOS did its own check for a restart command, and

```
;----------------------------------------------------------------------
;
; ccp starting points
;
; start ccp and don't process default command stored
;
CCP1:   XOR     A               ;set no default command
        LD      (CBUFF),A
;
; start ccp and possibly process default command
;
CCP:    LD      SP,STACK        ;reset stack
        PUSH    BC
        LD      A,C             ;c=user/disk number (see loc 4)
        RRA                     ;extract user number
        RRA
        RRA
        RRA
        AND     0FH
        LD      E,A             ;set user number
        CALL    SETUSR
        CALL    RESET           ;reset disk system
        POP     BC
        LD      A,C             ;c=user/disk number (see loc 4)
        AND     0FH             ;extract default disk drive
        LD      (TDRIVE),A      ;set it
        JR      Z,NLOG          ;skip if 0...already logged
        CALL    LOGIN           ;log in default disk
NLOG:   LD      A,(CBBUFF)      ;is there system command to execute?
        OR      A
        JR      NZ,CBPROC       ;if not zero there is a command
        LD      A,(CBUFF)
        OR      A
        JP      NZ,RS1
        JR      RESTRT
CBPROC: LD      BC,9
        LD      HL,CBBUFF
        LD      DE,CBUFF
        LDIR
        XOR     A
        LD      (CBBUFF),A
        JP      RS1
;
```

Figure 4

found that there was none to execute, we would have entered the CCP at ENTRY+3. At ENTRY+3 we would have been ordered to jump to CCP1, which is the entry point when we do not want to process a restart command, or there is none to execute. Both circumstances have the same meaning.

When a command is entered at the prompt, it uses BDOS function 10, or input a line of text from the logical console device. This function call requires that a buffer be defined, and that the first character in the buffer tell the DOS what the maximum number of characters to accept should be. Function 10 will return to the caller when either this number of characters is reached, or the operator signals the end of a line by pressing the RETURN key, or control-M, (^ M).

```
BUFLEN  EQU     80              ;maximum buffer length
MBUFF:  BYTE    BUFLEN          ;maximum buffer length
CBUFF:  BYTE    0               ;number of valid characters in command line
CIBUFF: DATA    ' '             ;default (cold boot) command
CIBUF:  BYTE    0               ;command string terminator
        RSRV    BUFLEN-($-CIBUFF)+1     ;total is 'buflen' bytes
;
```

A type of buffer for DOS function 10 is shown above, and is from our CCP. The equate BUFLEN states that the byte stored in the BYTE at MBUFF should be 80, which represents the maximum length of the input command line. This could have just as easily been stated as:

```
MBUFF:   BYTE   80
```

With this byte as the first byte in our input buffer we tell DOS function 10 to look here for the maximum number of characters by saying:

```
LD   DE,MBUFF   ;load the address of buffer in DE pair
LD   BC,10      ;load function number into BC pair
CALL 0005H      ;make the DOS call by jumping through
                ;location 5, page 0
```

When DOS function 10 has done its job, returning only when the maximum number of characters has been input, (80), or the operator enters a Carriage Return, (CR), it will place the actual number of characters input in the byte at location CBUFF. For entry into the CCP at CCP1 this byte, which contains the actual number of characters in the returned command line is our focal point:

```
; start ccp and don't process default command
;
CCP1:   XOR     A               ;set no default command
        LD      (CBUFF),A
```

In this application of the exclusive OR logic function we are exclusive ORing the 'A' register with itself, which produces a zero figure. Any time a value is exclusive ored with itself a zero amount is returned. We could also have said:

```
LD      A,0
```

with the same result. Our next instruction states that we are to place this zero value into the byte at CBUFF. When we leave this instruction, a zero value will be in the byte at CBUFF. Why did we do this? Well, the quickest way to determine whether there are any characters in the command line is to check and see how many characters DOS function 10 says should be in the command line. Now if we put a zero value in CBUFF, whose function is to hold the number of characters in the command line, then when we check we will be told the command line is empty. If the command line is empty, as CBUFF tells us, then there is no use trying to execute whatever is in the command line. It is empty. Computer psychology at work friends, if you can't dazzle them with your brilliance, dazzle them with your bulls#$t.

Having gone 50 miles in a two line program segment we then drop into CCP, where we would have come if we wanted to execute a restart command. Yup, now we are at where we didn't want to go anyway, and took a two liner "short cut" to get there. Oh well, that's high-tech.

```
;
; start ccp and possibly process default command
;
CCP:    LD      SP,STACK        ;reset stack
        PUSH    BC
        LD      A,C             ;c=user/disk number (see loc 4)
        RRA                     ;extract user number
        RRA
        RRA
        RRA
        AND     0FH
        LD      E,A             ;set user number
```

Having arrived here we have to explain some assumptions we have made and thus far ignored. The first of these is that the BIOS is supposed to place the current disk and user area in the 'C' register before jumping to the CCP.

However, the first thing we do when we actually get into the CCP is create our own STACK. A stack is a place where you stuff things just to get them out of the way, such as the place to return to after a subroutine CALL and data you want to preserve. A stack builds down, and you "PUSH" things onto it, and 'POP" things off of it. In our CCP we have memory reserved for these stack functions called, of course "STACK." The register 'SP," into which this 16 bit value is loaded is used by the processor as well. Having defined where our closet of values is, where to stuff things, we immediately stuff the disk data sent to us in the 'C' register for safekeeping.

Now remember when we were talking about USER areas a while ago. No? Well go back and look, I'll wait, I mean I've nothing better to do than wait on you, just so I can confuse the 'ell out of you again, so go ahead... go back and reread that section..... I'll wait......

I am assuming that you now know that the user area number is held in the upper portion of the disk/user byte at location four. Now if a byte is eight bits, then the upper four bits represent our user area. We first move the disk/user byte into the 'A' register, from the 'C' register and then use the command Rotate Right Accumulator, (the 'A' register), to extract the user number.

```
        LD      A,C             ;c=user/disk number (see loc 4)
        RRA                     ;extract user number
        RRA
        RRA
        RRA
        AND     0FH
```

What we have done is shift the user number bits four places to the right, so they are now the four least significant bits:

```
BEFORE:  UUUUDDDD     RRA
DURING:  XUUUUDDD     RRA
         XXUUUUDD     RRA
         XXXUUUUD     RRA
AFTER:   XXXXUUUU
```

where X means don't really care. We now do a logical AND with the lower four bits, which house the user area code, which has the effect of making all the X marks zero values.

```
        LD      E,A             ;set user number
```

Now we move the value of the disk/user byte, which now only has the user number, into the 'E' register from the accumulator, or 'A' register. We make this move in preparation of calling a subroutine to deal with the user number.

```
        CALL    SETUSR          ;set the user value in 0004H
        CALL    RESET           ;reset disk system
```

These two calls set the user number and disk number, which we received from the BIOS, into the storage area at 0004H and reset the disk system so that everything is at a starting point for further operations. This setting things up is called "initializing the system."

We then do almost the same thing, separating the disk number now from the user number. First we get the original value from off of the stack, put it in 'A,' and then do a logical and on the lower four bits. This is essentially the same way we did the user value except that the value we want is already in the lower four bit position.

```
        POP     BC
        LD      A,C             ;c=user/disk number (see loc 4)
        AND     0FH             ;extract default disk drive
        LD      (TDRIVE),A      ;set it
        JR      Z,NLOG          ;skip if 0...already logged
        CALL    LOGIN           ;log in default disk
```

Now it could be that the system drive, disk 'A,' whose number is zero, is already assigned. The action of ANDing out drive value would set the 'Z' or Zero Result flag if it was. The system, or default drive will always be the lowest drive number. If this is in fact what has happened, that the drive number is already zero, then we do not have to log in the system drive. If the value isn't zero then we have to assign the new drive, or 'log it in.'

Now then, remember all the discussion about the restart command? We have to deal with the possible occurence of a restart command now. From our discussion above, can you determine how we will deal with this determination?

```
NLOG:   LD      A,(CBBUFF)      ;is there system command to execute?
        OR      A
        JR      NZ,CBPROC       ;if not zero there is a command
        LD      A,(CBUFF)
        OR      A
        JP      NZ,RS1
        JR      RESTRT
```

The Ampro "autocommand" or restart command is located 62h bytes above the BIOS entry point. Just like

CBUFF it has a value of actual characters in the restart command line. Ampro has limited this amount to just eight characters. At NOLOG we sneak a peek at the byte which is to tell us how many characters are in the restart command. We load the accumulator, or 'A' register, with that value. We then do a logical OR with the accumulator, or OR it against itself. We do this to see if there is a zero value in it. If there is then the 'Z' flag will be set after the OR function.

If the 'Z' flag is not set then the value was not zero and we want to jump relative to where we are, upon a nonzero returned value, to the location represented by CBPROC. "CBBUFF" means "cold boot buffer," and "CBPROC" refers to cold boot processing.

Now if there is a zero value in the BIOS restart command buffer, (CBBUFF), we check to see if there is a command in CBUFF, which is the CCP command buffer. If there is a command in this buffer then we will jump to RS1, else we will jump to RESTRT for normal CCP processing.

It is important to note that we are checking two restart buffers, why? Well, in my system you can have a command in the cold boot buffer and a command in what I call the warm boot buffer. The BIOS is loaded from disk only upon a power-up or reset condition. The CCP is loaded upon every warm boot. In this way it is possible to configure the system, using either or both restart command potentials, to assure that the system never reaches the command line. That is to say that even if the power goes off the system can recover and reset itself. This is important where reliability is at issue.

From this point we again have a fork in the processing road. If there was a cold boot command lurking in CBBUFF then we have to discuss its processing. If there was a command in CBUFF then we have to discuss it, and then discuss what to do if there were no restart commands at all to deal with. Let us begin where we jumped off to do the cold boot command, which would have sent us to CBPROC.

```
CBPROC: LD      BC,9
        LD      HL,CBBUFF
        LD      DE,CBUFF
        LDIR
        XOR     A
        LD      (CBBUFF),A
        JP      RS1
```

At CBPROC we perform a transfer of the restart command string from CBBUFF down into CBUFF. We do this by using a Z-80 specific assembler code, "LDIR." We know that the Ampro "autocommand" in the BIOS allows only an eight character file name. We also know that, like the CBUFF format, it has a number which indicates how many characters are in the command line.In the LDIR instruction the 'BC' register pair, (think of them as the "byte counter"), is loaded with the maximum number of characters we want to move. In this case the number of characters is nine. The actual BIOS autocommand buffer looks like this:

```
          ORG BIOS+62H
AUTOCMD:  BYTE    Ø                 ;number of characters in command
          DATA    '        '        ;blank command file name
          BYTE    Ø                 ;terminating null character

         ORG AUTOCMD+1Ø
```

In this case we are ignoring the terminating null as we know how many characters there are we want to move.

```
LD      HL,CBBUFF
LD      DE,CBUFF
LDIR
```

We next load the 'HL' pair with the source address of the data we want to move, CBBUFF. We cannot use the AUTOCMD label as it is unique to the BIOS source code. We could have used it here as we had defined it with the value of EE62H instead of using CBBUFF. But, we are concerned with the Cold Boot BUFFer. The 'DE' register pair is used to specify the DEstination address for our move. The destination address is CBUFF. CBUFF is used for all command processes in our CCP.

Having defined the number of bytes to be moved, by placing this number into the 'BC' register pair, the source address of the nine bytes to be moved in the 'HL' pair, and the destination of these bytes in the 'DE' pair, we are ready to make the move. The move is made by uttering the magical incantation "LDIR." (Hey! Give me a break! After all that set up such an anticlimactic ending needs something. OK, so it isn't magic. I bet you are the type that pulls back the Wizard of Oz's curtains just to ruin the show!)

```
XOR     A
LD      (CBBUFF),A
JP      RS1
```

So, party poopers and all, the move has been made, and the command hidden away in the BIOS is now in CBUFF, complete with the number of characters in the line. We now zero the accumulator with the exclusive or command, and stuff the zero value into CBBUFF, where we just moved our restart command from.

Now why did we do that after we just went to so much trouble to move it down where we could work on it? Remember that the standard Ampro BIOS does not perform its own check for a restart command in the AUTOCMD slot. Because it does not do this checking itself it does not make the decision as to jump into the CCP at ENTRY, or ENTRY+3; "to process or not to process, that is the question." The Ampro BIOS always jumps into the CCP at ENTRY, which is the process a restart command.

The BIOS is loaded from disk only from a power on or reset. Yet it loads the CCP back into memory after every warm boot function. A warm boot may be executed after a program is run, or when we press control-C. Now if the Bios always jumps into the CCP at ENTRY, to process a restart command, our restart command will be detected and executed every time the CCP is entered.

The object of the CBBUFF processing is to execute a restart command only once, and that is only upon power-up or reset. To prohibit re-execution, called "reentry," we put a zero value in CBBUFF. The next time we come through and check this position we will be told that there is no restart command in the BIOS, hence we will not try to process it.

Having zeroed out the BIOS restart command we jump to RS1 where a normal command in CBUFF, the CCP restart command buffer, is processed. Before we go to that routine, however, let's keep in mind what we have just done with the BIOS restart command and examine the normal CCP restart functioning.

I heard that! Someone said, "why don't we just change the Ampro BIOS so it doesn't create all these problems with a simple restart command?" Well, Ampro had a reason for doing a direct ENTRY jump into the CCP. In the interests of compatibility we don't want to upset things anymore than we have to. When we do our industrial BIOS this is an option we may explore. But let's not get ahead of ourselves. For now this is the way we have to do things.

Let's review for a moment. At NOLOG we made a determinationas to whether there was a restart command in the BIOS. Review the code provided below.

```
NLOG:   LD      A,(CBBUFF)      ;is there system command to execute?
        OR      A
        JR      NZ,CBPROC       ;if not zero there is a command
        LD      A,(CBUFF)
        OR      A
        JP      NZ,RS1
        JR      RESTRT
```

If there was a restart command in the BIOS we would have jumped to CBPROC, which we just finished discussing. At CBPROC we moved the command from the BIOS into CBUFF, in the CCP, and jumped to RS1. Let us now assume that there was no command in the BIOS to be processed. In this case we drop into the code that makes a check to see if there is a command in CBUFF.

```
        LD      A,(CBUFF)
        OR      A
        JP      NZ,RS1
        JR      RESTRT
```

We do the same thing as we did for CBBUFF, that is check the byte holding the number of characters in the text string. If it is a zero then there is no command to process. A nonzero value indicates that there are characters present, a command to execute. At this point we would jump, if there were a command, to RS1, where all roads for execution of a command lead.

Now remember that I said that the CCP is reloaded after every warm boot, and the BIOS generally only once per session? Now if we placed a command line in CBUFF, the CCP command line, and wrote it to disk, then every time the CCP was loaded it would execute our command, wouldn't it? Yes it would. The first thing we check is to see if there is an initial restart command in the BIOS. If there is we move it into the CCP and execute it, over-writing any command that may currently be in CBUFF. Once we move the BIOS restart command, however, we zero out the character byte so we do not re-execute that command. But, every time the CCP is reloaded, had we a command written into the CBUFF command line in the CCP it would be executed after every warm boot. It would be impossible for the system to ever reach the "A0>" prompt as it would always take a priority command from the restart procedures.

Our priority for automation of our system could be stated as:

BIOS AUTOCOMMAND: Executed Once, Set By Ampro CONFIG.COM

CCP RESTART COMMAND: Executed after every possible programtermination except a reset or power loss. Set by a user program or DDT

It is clear that we could produce a system that was extremely reliable and able to reset itself after an interruption of power or other fatal error, as well as restart itself after program termination or process sequence. We are well on the road to a totally automated system that is both reliable, and intelligent.

Now then, if no restart commands are detected, (all roads lead to RS1 eventually), then we must get a command from the human. This is done by a jump over CB-PROC to RESTART, where all CCP roads return.

```
;
; prompt user and input command line from him
;
RESTRT: LD      SP,STACK        ;reset stack
        XOR     A
        LD      (CBUFF),A
```

RESTRT is the CCP's internal restart point. When all internal functions have been completed the CCP will return here to begin a new sequence. This sequence is a simple procedure of getting your command from the command line, processing the command, if it is an internal function, or loading the file you specify, and transferring control of the system to that program.

When we entered the CCP at CCP we set up a local stack, and repeat that process again here, why? Because the stack is kept small, and the processing of any restart command is a sequence all of its own. When we reach this point we are in the "inner sanctum" of the CCP where we start anew. We set up a new stack, (actually we just reset the old one as if it were new, and clear our character byte at CBUFF). In this way we have a fresh stack and tell anyone who may ask that there are no commands to execute, everything that needs to be done has been done. We may now begin a new day.

```
---------------------------------------------------------------------------
;
; prompt user and input command line from him
;
RESTRT: LD      SP,STACK        ;reset stack
        XOR     A
        LD      (CBUFF),A
;
; print prompt (du>)
;
        CALL    CRLF            ;print prompt
        CALL    GETDRV          ;current drive is part of prompt
        ADD     A,'A'           ;convert to ascii a-p
        CALL    CONOUT
        CALL    GETUSR          ;get user number
        CP      10              ;user < 10?
        JR      C,RS00
        SUB     10              ;subtract 10 from it
        PUSH    AF              ;save it
        LD      A,'1'           ;output 10's digit
        CALL    CONOUT
        POP     AF
RS00:   ADD     A,'0'           ;output 1's digit (convert to ascii)
        CALL    CONOUT
;
; read input line from user
;
RS000:  CALL    REDBUF          ;input command line from user
;
; process input line
;
RS1:    CALL    CNVBUF          ;capitalize command line, place ending 0,
                                ;and set cibptr value
        CALL    DEFDMA          ;set tbuff to dma address
        CALL    GETDRV          ;get default drive number
        LD      (TDRIVE),A      ;set it
        CALL    SCANER          ;parse command name from command line
        CALL    NZ,ERROR        ;error if command name contains a '?'
        LD      DE,RSTCCP       ;put return address of command
        PUSH    DE              ;on the stack
        LD      A,(TEMPDR)      ;is command of form 'd:command'?
        OR      A               ;nz=yes
        JP      NZ,COM          ;immediately
        CALL    CMDSER          ;scan for ccp-resident command
        JP      NZ,COM          ;not ccp-resident
        LD      A,(HL)          ;found it:  get low-order part
        INC     HL              ;get high-order part
        LD      H,(HL)          ;store high
        LD      L,A             ;store low
        JP      (HL)            ;execute ccp routine
;
; entry point for restarting ccp and logging in default drive
;
RSTCCP: XOR     A
        LD      (CBUFF),A
        CALL    DLOGIN          ;log in default drive
;
; entry point for restarting ccp without logging in default drive
;
RCCPNL: CALL    SCANER          ;extract next token from command line
        LD      A,(FCBFN)       ;get first char of token
        SUB     ' '             ;any char?
        LD      HL,TEMPDR
        OR      (HL)
        JP      NZ,ERROR
        JP      RESTRT
```

Figure 5

```
;
; print prompt (du>)
;
        CALL    CRLF            ;print prompt
        CALL    GETDRV          ;current drive is part of prompt
        ADD     A,'A'           ;convert to ascii a-p
        CALL    CONOUT
```

If there was a restart command we would have jumped over this program segment. We are assuming that we want to get a command from the human operator. Computers are so willing to please.....

The first thing we do is clear a line on the terminal, (our ego demands a clear work space). We clear a line by sending a RETURN, (CR, which sets the cursor to the beginning of a line), and a LINE FEED, (LF, which moves the cursor down a line), to the terminal. To make life easy for the human we include a LF every time he, or she, presses the RETURN key.

In the interest of brevity, as we are running out of space for this issue we will highlight what each subroutine does. We will discuss the various support routines in detail later. This time I only want you to understand the primary CCP function loop.

GETDRV returns the binary number of the current disk drive. We then add the value of a capital "A" character, (41H) to convert the disk drive number into a value that can be printed by the terminal. If the disk drive number is "0," then when we add the number for a capital "A" we have the value of the ASCII code for "A," as we have added nothing to it. If the drive number was a "1" for drive "B" then the base figure of 41H would have one added to it, which would be the code for "B," and so on. A standard CP/M system can have up to 16 disk drives, represented by the letters "A" through "P.'

At the assembler level, and unlike BASIC, just because we print something doesn't mean a CR/LF sequence is also printed. With the printable value of the current disk drive in the accumulator we call the subroutine CONOUT which sends our character to the terminal. At this point in time just the letter is printed. We process so fast, however, that when the prompt is printed it appears as if the entire prompt appears at once.

```
        CALL    GETUSR          ;get user number
        CP      10              ;user < 10?
        JR      C,RS00
        SUB     10              ;subtract 10 from it
        PUSH    AF              ;save it
        LD      A,'1'           ;output 10's digit
        CALL    CONOUT
        POP     AF
RS00:   ADD     A,'0'           ;output 1's digit (convert to ascii)
        CALL    CONOUT
```

In a similar manner we call a routine that returns only the current user number. For a hint at how this is done, basically, refer to our discussion of the entry point CCP early in this article.

What follows is a simple binary to decimal conversion routine. We may not just use an offset value to convert the user number into a printable form, as we did for the drive number, as the user area is represented as a decimal number, instead of a letter. Additionally, we have more than one digit to deal with. If the number is less than 10 we do not have to formulate the "tens" value, which must be printed first. (In the number 10 the "1" is the tens value.)

If the number is greater than 10, then we subtract ten from it, which will leave us with the "ones" value as the remainder. This remainder will be left in the accumulator, or 'A' register. As the ones value must be printed after the tens value, we will save this remainder on the stack.

```
        LD      A,'1'           ;output 10's digit
        CALL    CONOUT
        POP     AF
```

Having saved the ones value, and because we know that the maximum number of drives can never be more than 16, we just print the character "1" on the screen any time the user area is greater than 10. Having the prompt now appearing as "A1 " we must print the remainder, or ones value. So let's get it back, by popping it off the stack, and make it printable by adding the lowest possible number code to it. The ones value is converted into printable form in the same manner as the disk number, except that the offset is now the character "0."

```
RS00:   ADD     A,'0'           ;output 1's digit
                                ;(convert to ascii)
        CALL    CONOUT
```

Note that if the value of the user number was determined to be less than 10 we would have come here and made this single digit conversion.

```
;
; read input line from user
;
RS000:  CALL    REDBUF          ;input command line from user
;
```

Because we are assuming that there were neither a restart command in the BIOS, or in the CCP buffer, to execute, we just have to have a command to process. We have printed the "A0 " portion of the prompt on the screen. REDBUFF supplies the ">" portion of the prompt while it waits for our input. This uses the same DOS function 10 sequence we have already discussed. The buffer for this input is at MBUFF, whose byte contains the maximum number of characters to be accepted from the console. When this number of characters has been received, or a CR is entered by the operator, the number of characters actually in the command is placed in CBUFF. The characters themselves begin at CIBUFF.

Now then, whether we had a command in the BIOS, one in CBUFF when we entered the CCP, or we just received one from the operator, our varied paths gather at RS1.

```
; process input line
;
RS1:    CALL    CNVBUF          ;capitalize command line, place ending 0,
                                ;and set cibptr value
```

The first thing we do to our command is convert all characters into uppercase format. In this way the operator can enter command either in upper, or lower case. We also set a pointer, CIBPTR to point to the first character in our command. We are now ready to get to work, after a little bit of preparation.

```
        CALL    DEFDMA          ;set tbuff to dma address
```

The first step in preparation is to set up a buffer for any and all data from the disk, or terminal.

```
        CALL    GETDRV          ;get default drive number
        LD      (TDRIVE),A      ;set it
```

Then we save the value of the drive we are currently on, so we can "remember" where to return after we perform the command to be executed.

```
        CALL    SCANER          ;parse command name from command line
        CALL    NZ,ERROR        ;error if command name contains a '?'
```

All commands must conform to a very standard format for primitive interpreters. This is the format of VERB - OBJECT OF VERB. SCANER, we may assume for the moment, does a quick syntax check to assure that all commands are properly formed. It checks to see if there is a disk drive specifier, (if the command is to be found on a different drive), and assures that the user isn't trying to do something outrageous. If all is well SCANER returns with the 'Z' flag set. If there is an error, or an object appears before the verb, the 'Z' flag is not set, and we are routed to an error handling routine.

```
LD      DE,RSTCCP       ;put return address of command
PUSH    DE              ;on the stack
```

When we "CALL" a subroutine, the "way home" to the caller is placed on the stack. When a RET command is seen the top two values on the stack are assembled into a 16 bit address, and the processor executes a jump to that assembled address. The CCP also has to be able to find its way home so it can accept another command from us; this is, after all, its function in life. Internal commands are also called as subroutines, and have to find their way home. In the code above we set the return address on the stack so that when a RET command is encountered program control is sent to a recovery routine.

```
LD      A,(TEMPDR)      ;is command of form 'd:command'?
OR      A               ;nz=yes
JP      NZ,COM          ;immediately
```

SCANER will set TEMPDR, (TEMPorary command DRive), if the command, or verb to be executed is specified to be on another disk drive. To avoid needless processing, we can check for another drive assignment, and assume that the verb portion of the command is a file name. If the value of TEMPDR is other than the system drive, drive '0,' then we jump directly to the COM file load and execution routines.

```
CALL    CMDSER          ;scan for ccp-resident command
JP      NZ,COM          ;not ccp-resident
```

If we do not get a clue as to what type of command it may be, then we must check to see if the verb is an internal command. CMDSER, (CoMmanD SEaRch), will search a table of key words, or verbs for a match with the verb in the command. This table is constructed as follows:

```
DATA  'COMMAND'
WORD  COMMAND
DATA  'ANOTHER'
WORD  ANOTHER
```

The DATA contains the literal verb string. The WORD is a label representing the address of the concerned routine. Remember that a binary WORD is a 16 bit value which, in this case, represents an address in memory. If a match is found between the command verb and a verb string in the command table, the end of the character by character match will be the last verb character, plus one, (as if looking for another character to match). Hence, on return, if a matching verb was found, the 'HL' pair will be pointing at the first byte of the address of the verb's action routine. If no match is found then the command verb is assumed to be a file name to be fetched from the current disk drive, and executed. If an internal command verb is matched then we must jump to the verb's subroutine for execution.

```
LD      A,(HL)          ;found it:  get low-order part
INC     HL              ;get high-order part
LD      H,(HL)          ;store high
LD      L,A             ;store low
JP      (HL)            ;execute ccp routine
```

We then load the low order byte of the address into the 'A' register, from memory, increment 'HL' which then points to the high order byte of the 16 bit address, into the 'H' register. We then load the low order byte stored in the 'A' register into the 'L' register. Having loaded the address of the verb's subroutine into the 'HL' pair we jump to where HL is pointing. Because we have placed our "way home" on the stack we can return to the main loop of the CCP by executing a RET instruction.

```
;
; entry point for restarting ccp and logging in default drive
;
RSTCCP: XOR     A
        LD      (CBUFF),A
        CALL    DLOGIN          ;log in default drive
```

RSTCCP is where we return from most internal commands. When a file is used as the verb the program generally exits to the warm boot loop which rewrites the CCP and enters it where we originally entered. For those functions that return here, the first thing we do is make damn sure that the command line character byte is zeroed out. We don't really need to do this, as we will just reset it again when we get backup to RESTRT, but some programs return to the CCP, and do not terminate to the BIOS warm boot function, or have mystical, magical ways of trying to re-execute the restart command. In an over-kill mode I went in and put "dummy traps" everywhere to make sure that the two restart commands are only executed when and where they were supposed to, every time they were supposed to. When reliability is an issue, a little redundant code can sometimes help....

Having once again managed the restart commands, we want to reassign the system disk drive, drive 'A' or zero, as the current disk drive. We then "fall into" RCCPNL, which may also be used as a CCP return point when a command does not wish to reassign the disk drive being used in the execution of a command. You will note that there are sections of code that may not always seem needed in my CCP. This is because I modify it for nearly every specialty system I create. When we begin designing our own commands, you may see why these sections of code are left here.

```
;
; entry point for restarting ccp without logging in default drive
;
RCCPNL: CALL    SCANER          ;extract next token from command line
        LD      A,(FCBFN)       ;get first char of token
        SUB     ' '             ;any char?
        LD      HL,TEMPDR
        OR      (HL)
        JP      NZ,ERROR
        JP      RESTRT
```

Well, this pretty well covers the main loop of the CCP. In Part Two we will discuss the various support routines called by the main loop. If we have the space we will also begin discussion of how to design your own CCP commands to suit your specific application.

I would recommend that you acquire the CCP source code, and do some snooping before we meet again. By the time we finish with the CCP section of our series you will have a great understanding of this module, and be able to modify it to suit yourself, with far more computing power than any standard system could have. Remember that, unlike the CCP's big brother ZCPR3, our system does not require any additional memory space, nor direct support files on your disk. When thinking of all the fun we can have reworking the CCP, just think of what can be done when we begin discussions upon the design of the BIOS and DOS systems! ■

* * *

NEW-DOS Disks Available

An AMPRO format 5¼ DSDD with the files for the Crowe assembler and the CCP is available from The Computer Journal for $10 postpaid. Inquire about other formats.

Additional disks with the BDOS and BIOS portions of NEW-DOS will be made available when these portions are published. Anyone making extentions to NEW-DOS or implementing it for other systems are urged to send their material to TCJ so that it can be shared with others.

Tom is preparing a user disk library for the AMPRO little board, and the disks will be distributed by TCJ. Watch for more details in the next issue!

Variability In The BDS C Standard Library
Porting BDS C To CP/M 86
By Donald Howes

This overview is aimed at C programmers who don't own a copy of the BDS C compiler, but still wish to be able to compile some of the large number of programs which are available from the group, and not become old before their time in doing so. If you are like me (I do most of my work in CP/M-86, using the SuperSoft C compiler), the following scenario has occured at least once (and possibly, many times). On getting your latest software disk from the group, you immediately try to compile a program. Everything works through the compiler, but then comes the link step (maybe an assemble step first, but why make things overly complicated). You think the machine is having a fit, but it's fascinating, who could have thought that a three hundred line program could have generated four pages of link error messages!!

Ok, maybe that is a little overblown, but it really can be a problem getting a BDS C program to link and there are some programs which I had given up on trying to get to work (if you want to know, Roff is one, I really wasn't lying about the four pages of linker errors). I've managed to solve the problem in a remarkably easy way. I've bought a copy of the BDS C compiler. This, however, may not be a viable alternative for people who are either short on cash, or don't have a machine which will run both eight and sixteen bit software (I use a CompuPro (Viasyn, who's Viasyn?) 8/16-A). Hopefully, this overview will help to alert those people who do not have access to a BDS C compiler to the variations in the "standard library". You will notice the quotes; one thing that I did find out is that there really isn't such a thing as a standard library. What I was able to do was compare the BDS C library functions to the two CP/M-86 compilers for which I have documentation (SuperSoft and Digital Research) and note the variability over the three compilers. The SuperSoft documentation states that they have attempted to stay as close as possible to the Unix library, while Digital Research is missing a number of Unix C functions and have implemented some specialized functions (there are three variations on creat(), for example) to take the place of a single Unix function. With this type of mix (admittedly, not a scientific sample, but you do the best with what you've got) I was able to break the library functions up into three types. First, there are the functions which all three compilers agree on. Given the differences between the compilers, I felt that these should represent as close to a "standard" function as there is. These functions will be noted by their conspicuous absence from the following list. Second, there are the functions which both BDS C and one of the other two compilers define in the same way (from my overview, this is generally the SuperSoft compiler), while the other compiler either does not support the function (the normal case) or the definition is different. Third, the cases where both the CP/M-86 compilers do not support the function, or the definition of the function is different. These are the ones which will cause the most trouble and I will flag the entry with two asterisks preceding the function name (i.e. **peek(n)).

Finally, a short note about syntax. The initial section heading where the function name is given will name the function with its list of parameters as they are given in the BDS C User's Guide (i.e. sleep(n)). If I refer to any function by name in the descriptive material following the section heading, no parameters will be given (i.e. sleep()). Please don't assume from this that there are no parameters for that function. Also, any parameters that are mentioned in text will be surrounded by single quotes (i.e. for sleep(n) the parameter 'n' would be quoted). In deference to those who will be rummaging through this listing in the small hours of the morning, I have taken the liberty of rearranging the functions from their categorical order as found in the User's Guide into alphabetical order.

A Note on Buffered I/O Functions

In the following list, only those functions which do not have the same number of parameters being passed are shown. However, there is a general difference between the way BDS C handles buffered I/O and the "standard" form of those functions. As is mentioned below, the BDS C version of fopen() does not pass a mode parameter when opening a file. The "standard" version of this function has the form: "fopen(filename,mode,iobuf)" and returns a valid file descriptor, which is used by all other buffered I/O functions to reference the opened file. BDS C buffered I/O functions do not use a file descriptor, but rather, directly reference the I/O buffer 'iobuf' (fopen() does return a file descriptor, but it is not used for other than error checking, since 'iobuf' itself maintains a copy of the file descriptor for use by other buffered I/O functions). It may be necessary, therefore, for you to place a 'mode' parameter in your buffered I/O calls, for them to operate correctly, check your compiler documentation.

alloc(n)

Returns a pointer to a block of memory 'n' bytes long. This is the dynamic memory (heap) allocation function used by BDS C. However, this function is obsolete and being dropped from the standard libraries of some compilers. You should use calloc() or malloc() instead of alloc(), if they are available in your compiler.

**call(addr,a,h,b,d), calla(addr,a,h,b,d)

Both of these functions are used to call a machine subroutine at location 'addr'. If used outside of the CP/M-80 environment, almost anything can happen, none of it good. The best that can be done is to try to determine what the routine was to do, and recode in standard C. The use of these functions makes the program essentially untransportable (at least, not easily).

**cfsize(fd)
The function calculates the exact number of sectors in the open file given by the descriptor 'fd', without affecting the associated R/W pointer.

**codend(),externs(),topofmem(),endext()
I have grouped these four functions together, since they all deal with the calculation of different areas of memory for dynamic use in a program. These functions could cause real problems, but they're so handy that they will almost invariably be used if the situation is appropriate.

Codend() and externs() are essentially equivalent functions. Codend returns the first byte following the program code and externs() returns the first byte of the external data area. These will normally be the same, unless the external data area has been explicitly moved (this could be done so the code could be ROMmed).

Topofmem() returns a pointer to the last byte of user available memory (generally the base of the BDOS in CP/M-80), while endext() returns a pointer to the byte following the external data area. You can see that the use of these two functions will allow for the calculation of the amount of space in the system which can be used as heap space.

Some compilers may not have any of these functions available, or some may be present but the action of the function may be different (for example, the SuperSoft compiler has a function named topofmem(), but it functions the same as endext() in BDS C). If your compiler has a way of determining the top of the external data area, and you are using a small memory model (for 16-bit compilers), the size of the heap area can be found by subtracting the top of the external data area from 0xFFFF (the top of the data segment in a small memory model). My thanks to John Johnson of Professional Microware, who pointed out this fix.

**creat(filename)
Creates the file of name 'filename', erasing any existing file which already has that name. Your compiler may require an additional parameter after 'filename', the mode in which the file has been opened. Check your compiler documentation for the parameter list.

**csw()
Returns the byte value of the console switch register.

**errno(), errmsg(errnum)
Does the same as the external variable ERRNO and the function perror(s) combination found in other compilers.

**execv(filename,argvector)
This function allows the passing of a variable number of arguments to the chained program 'filename', by passing 'argvector' a pointer to an array of string pointers. This could require a real software kludge to port a program.

**exit()
If there is one function that I would have thought would have been standard between compilers, it is exit(). No such luck, each of the three compilers handled the closing of files and flushing of buffers in a different way. In BDS C, exit() will close all open files, but does not flush any buffers. This means the a BDS C program will have a call to fflush() to empty any buffers before a call is made to exit(). This may not be necessary for you. Check your compiler documentation to see just how exit() functions in your compiler.

**fabort(fd)
The function frees the file descriptor 'fd' without closing the file. This function was present in the SuperSoft compiler, but only to maintain some compatability with BDS C. It's not a great idea to use this function even if it's present in your library, since some or all of the file input can be lost if the file had been opened for writing.

**fcreat(filename,iobuf)
Creates the file 'filename' and opens the file for buffered output using a buffer pointed to by 'iobuf'. The size of the buffer is determined from the BDSCIO.H variable BUFSIZ. This function is needed, since the BDS C version of fopen() does not support the mode parameter. A call to fopen of the form "fopen(filename,mode,iobuf)", where mode is declared as "w" (write only) would accomplish the same. See below for the BDS C version of fopen().

**fgets(str,iobuf)
Reads a line from the input buffer 'iobuf' and loads it into the string pointed to by 'str'. A third parameter 'n' (the number of bytes to be read) may be required by your compiler. The BDS C version reads the buffer until an end of line is found in the input stream, not until a specified number of bytes have been read. The alternate version of fgets() has the form "fgets(str,n,iobuf)".

**fopen(filename,iobuf)
Opens the file 'filename' for buffered input and initializes 'iobuf', the input buffer. This function does not implement the file I/O mode parameter and, therefore, may be a parameter short for your version of fopen(). The alternate version of fopen() has the form "fopen(filename,mode,iobuf)".

getchar()
The BDS C version of getchar() tests for ^C and reboots the operating system if found. My other compilers don't, so if you wish to do this type of interrupt test, it would have to be coded explicitly.

**getline(strbuf,maxlen)
Returns a text line of characters of maximum length 'maxlen' into the space pointed to by 'strbuf'. This seems to be a special case of gets(),with the maximum line length given as a parameter (there is an automatic return with getline() when 'maxlen' is reached). Gets() could be used in place of getline(), though you would have to watch that the length of the string did not exceed the size of the array into which it was being read, since gets() does not check this.

getval(strptr)
'Strptr' is a pointer to a pointer of a string of ASCII characters separated by comma's. This is the driving routine used by initw() and initb() to fill their arrays. This function probably won't be present in compilers which allow initialization (see the descriptions of initb() and initw(), below).

initb(array,string), initw(array,string)
These functions are used to perform the initialization of character and integer arrays respectively. They are not needed in compilers which allow the initialization of arrays at the time they are declared.

inp(n), outp(n)
The functions read and write 8-bit values to the port 'n'. If these functions are not present in your compiler, it would be possible to accomplish the same thing by the use of pointers.

isspace(c)
Tests whether the character 'c' is space, tab (\ t) or newline (\ n) character. This same functions may be called iswhite() in other compilers.

khbit()
Polls stdin to see if there is a character present, returns TRUE orFALSE.

movmem(source,dest,count)
Moves 'count' bytes of memory from location 'source' to destination 'dest'. The original memory is not modified, unless the destination area partially overlies the source.

**oflow(fd)
Quoting the manual "returns true (non-zero) if an overflow has occured into the high order (third) byte of the random-record field of the FCB". Good luck.

**open(filename,mode)
Opens the file specified by 'filename' for I/O as given by 'mode'.However, the meanings of the mode values are different in BDS C. BDS modes are: 0 = input (write only), 1 = output (read only), 3 = input/output (read/write).

pause()
Tests for console input, looping until a key is pressed.

**peek(n), poke(n,b)
These are equivalent to BASIC PEEK and POKE statments, and are not really necessary, since C supports indirection. Peek(n) can be simulated by initializing a char pointer:

```
char *bdosjmp = 0x05;
```

This won't work in BDS C or other compilers which don't support initialization. If the compiler doesn't support initialization use:

```
char *bdosjmp;
bdosjmp = (char *) 0x05;
```

(This will only work in the 8086 environment if the DS register points to the correct segment. This can't be guaranteed).
As is pointed out in the BDS C manual, poke() is better accomplished by using pointers:

```
*n = b;
```

putchar(c), putch(c)
The BDS C version of putchar() is able to detect the input of ^C (and ^S) during character output. Putch() does not detect these control characters. A call to putch(), therefore, is equivalent to putchar() in other compilers. If you want to be able to interrupt character output, you will have to code an explicit test into the output loop.

qsort(base,nel,width,compar)
This function is used by BDS C to conduct a shell sort. The type of sort that is conducted may be different for your compiler (Digital Research does a quick sort) even though the function name is the same. Check your compiler documentation.

**read(fd,buf,nbl)
Reads the number of blocks given by 'nbl' (1 block = 128 bytes) from the file given by the file descriptor 'fd' into the buffer 'buf'. Other versions of read() require the number of bytes to be read as the final parameter, rather than the number of blocks. To pass a valid parameter, multiply the value of 'nbl' by 128.

rename(old,new)
Renames the file given by filename 'old' to that given by filename 'new'. Although there are obvious advantages to this function, it is not supported by Digital Research, and possibly not by your compiler. Check your compiler documentation. This could be accomplished by a BDOS call to change the FCB.

**rsvstle(n)
The function limits the closest approach of the stack (which grows down from the top of memory) and th heap (which grows up from the end of the external dat area) to 'n' bytes. Stack/heap management of this type generally the responsibility of the programmer.

**setfcb(fcbaddr,filename),fcbaddr(fd)
Setfcb() initializes a CP/M FCB with the string pointe to by 'filename', while fcbaddr() returns the address c the FCB pointed to by 'fd'.

setmem(addr,count,byte)
The function sets 'count' contiguous bytes starting at 'addr' to the value of 'byte'. This function is used to initialize buffers and arrays. It is not needed in compilers which support initialization.

sleep(n)
Suspends the execution of a program for a variable amount of time. Since how the time delay is calculated is compiler and processor dependent you should consult your compiler documentation.

**ungetch(c)
The character 'c' is placed on the console buffer and is returned by the next call of getchar(). This function may

be called ugetchar() or ungetchar() by your compiler.

**write(fd,buf,nbl)
The function writes 'nbl' blocks from the memory location pointed to by 'buf' to the file pointed to by 'fd'. Other versions of write pass the number of bytes to be written, rather than the number of blocks. It would be necessary to multiply the value of 'nbl' by 128, to obtain a valid parameter. ■

The C Users' Group

The above article was reprinted from The C Users' Group (CUG)September, 1985 newsletter, with their permission. A language is only as good as what you can do with it, and CUG maintains a large library of very useful programs. We will be reviewing and commenting on some of their disks in the future, but you should contact Donna Stucky Ward at The C Users' Group, Box 97, McPherson, KS 67460 for details on joining the group and a catalog of their disks.

Starting with the next issue Donald Howes will be writing a column on C language programming, and we would appreciate your comments and suggestions on the topics to be covered. We would also like to include your tips, routines, and questions.

The SCSI Interface

Introductory Column To A Series

By Rick Lehrbaum

To research the history of SCSI, I had lunch with Larry Boucher, the founder of SCSI. It all began in the Spring of 1979 when Larry was the Director of Design Services at Shugart Associates, and Shugart was getting ready to announce another new Winchester disk drive product. As usual, it would take 1½ to 2 years before Shugart could enjoy widespread sales of the new drive since it took that long for all the controller designers to debug their phase-locked loops. But that was the way it had to be back then—there was no standardized I/O bus in the micro world. (Unless you want to count RS-232!)

Larry decided that what Shugart needed was a way to speed up controller design. Before coming to Shugart, Larry had worked at IBM, and it occurred to him that something like the IBM OEM Channel also made sense for micro's. So Larry, along with Bernie Nieman (who worked for Larry) and Jim Korpi (who worked for Bernie), wrote the spec for a new interface to be proposed for all future Shugart disk controllers. They called it the Shugart Associates Standard Interface, or "SASI." According to Larry, there were two major objectives in defining SASI: (1) Make it the cheapest possible interface with an 8080-like bus; and (2) Outperform the IBM OEM Channel bus, which ran around 500,000 bytes per second average throughput. Both goals were met.

The rest is history. Shugart officially adopted the new SASI interface in July 1979, and commissioned Data Technology Corporation (DTC) to do the first SASI disk controller in August 1979. DTC performed admirably, under the direction of Dave Tsang, delivering samples of the new SASI controller board to Shugart in December, followed by delivery of 25 production boards in January 1980. A new industry standard was born. But few knew it...

In July 1981, NCR began taking an interest in SASI. In December 1981, John Lohmeyer (NCR) and Hank Meyer (Shugart) formally proposed to the American National Standards Institute (ANSI) that it adopt SASI as a small computer intelligent peripheral interface. An ANSI committee called "ANSC X3T9.3" was in the process of defining "The next microcomputer I/O interface." They turned SASI down! It seemed all was lost.

But there was another ANSI committee, currently inactive, which had been chartered to define a microcomputer "peer-to-peer" protocol. "ANSC X3T9.2," as this other committee was called, took SASI under its wing, in February 1982. One of the first official orders of business was to change SASI's name, since it contained Shugart's name. They settled on "Small Computer System Interface."

In the months that followed, ANSC X3T9.2, under the able direction of Bill Burr (National Bureau of Standards), hammered out a thorough and percise specification (currently over 180 pages long!) which makes SCSI one of the bext documented interface standards in the computer industry.

Last summer, the members of ANSC X3T9.2 unanimously approved the standard, forwarding it to the ANSI parent organization for public review and final approval. Short of editorial and other minor corrections, the final SCSI specification is in hand.

And now, a few questions from readers...

What is SCIS?

SCSI stands for "Small Computer System Interface" and is quickly becoming the most popular interface for connecting hard disk drives to small computers of every type. But as you'll see later, SCSI can be used for a lot more than that.

Why should I use SCSI?

You should use SCSI if you need an easy way to make your computer or computerized device expandable. Today, SCSI is mostly used for adding hard disk and tape controllers and drives. But soon, there will be lots of other functions to choose from, including: optical storage, network interfaces, graphics displays, co-processors, and more.

SCSI has a number of important advantages over other ways of attaching add-on's to a small computer system. One of the biggest advantages of SCSI is that is is an easy and inexpensive interface to add to any computer. It also simplifies your software hassles: when implemented correctly, SCSI allows you to change from one brand of device (such as a hard disk controller and drive) to another, with little or no software modifications.

How do you pronounce "SCSI"?

I'm glad you asked that question! There has really been a lot of time and energy arguing about how to pronounce SCSI. Well, it pretty much boils down to either pronouncing the word SCSI as SKU-zzy, or spelling it out S-C-S-I. My own personal preference is "SCSI." On the other hand, surveys show that 78% of corporate marketing executives prefer spelling it out.

Where can I get more information?

You have two choices: (1) Read this column in The Computer Journal, or (2) Be the first one on your block to have an official copy of the "ANSC X3T9.2 SCSI Specification," by sending $20, along with a self-addressed mailing label to: X3 Secretariat, Computer and Business Equipment Manufacturers Assn., 311 First Street NW, Suite 500, Washington, DC 20001.

Watch for "An Introduction to SCSI" in the next issue of The Computer Journal, and send your questions, ideas, or comments to the Editor at TCJ or to Rick at the following address. ■

Rick Lehrbaum
VP of Engineering
AMPRO Computers, Inc.
PO Box 390427
Mountain View, CA 94039

SCSI NEWS

This section will bring you the latest news and information about SCSI related products and applications. Your input and experiences will be greatly appreciated.

SCSI RAM-DISK—The AMPRO SCSI/RAM suitable for use in both single- and multi-master system applications is available with either 512K bytes or 1 megabyte of RAM, and can be used in any system having an SCSI (SASI) interface. By adding one or more SCSI/RAM units to a host computer, system memory need not be used as a RAM disk, thereby freeing the computer's system RAM for program functions. Data can be transfered over the SCSI bus at up to 1.5 megabytes per second, and effective SCSI/RAM disk transfer rates are generally as fast as when using system RAM. In addition, the SCSI/RAM does not clear its memory on SCSI bus reset, thereby allowing the SCSI/RAM to be used for system memory backup on power-fail detection.

Inexpensive SCSI host adapters allow connection to a wide variety of host computers, including the IBM-PC and AT, VME-bus, Multibus, and S-100. The SCSI/RAM, with 512K bytes of on-board RAM occupies a single 5¼ × 7¾ inch circuit card, and has the same mounting holes and footprint as industry standard 5¼ inch disk drives. A daughter board may be added to expand the RAM disk size to 1 megabyte. Available for $395 from AMPRO Computers, 67 E. Evelyn Ave, Mountain View, CA 94039, (415) 962-0230

SCSI Printer Server—The AMPRO SCSI/PRN intelligent printer server is suitable for both single- and mulit-master systems. Up to seven SCSI hosts can connect to a single SCSI/PRN, sharing printer and spool buffer resources, and maximum data transfer rate is 1.5 megabytes per second. The SCSI/PRN's three printer interfaces allow simultaneous connection of one Centronics compatible printer and two RS232C serial printers, with host controlled baud rates of up to 19.2K baud. Onboard memory of 512K bytes, expandable to 1MB via daughter board, is used as a spool buffer. Available from AMPRO for $449.

SCSI Real-Time I/O—The AMPRO SCSI/IOP is an intelligent processor which adds read-time control and measurement capabilities to any computer system having an SCSI interface. The SCSI/IOP plugs into a normal STD bus card cage, and can control STD bus I/O boards such as analog-to-digital converters, video display controllers, speech synthesizers, network interfaces, etc. Low cost SCSI host adapters are available for many computer systems, and SCSI's bus arbitration feature permits up to eight host computers and SCSI/IOP's to share resources.

The SCSI/IOP includes a 4 or 6 mHz Z80 microprocessor, eight byte-wide memory sockets for up to 64K of EPROM/RAM memory, and a Z80 counter/timer controller. Fimrware on the SCSI/IOP supports a number of basic operations, and both the Initiator and Target functions of SCSI which include peer-to-peer message capability. The SCSI/IOP can be used in hierarchical real-time control system configurations, or where either host or I/O device redundancy is required. Available from AMPRO with preliminary pricing of $119.

SCSI Hard Disk Subsystem—The AMPRO hard disk subsystem consisting of an SASI S1610-4 controller, 10 Mbyte half height hard drive, box with fan & supply, and cables is available 100% complete and tested for $599 from Peripheral Land, 3400 El Camino, Suite 10, Santa Clara, CA 95051 (408) 248-5282. ■

Indexed Sequential Access Method Files

Using Turbo Pascal ISAM Files

By Jerry Houston

ISAM, or Indexed Sequential Access Method, files are the type most commonly used for business purposes on large computers. ISAM file records are accessed according to a KEY FIELD for reading, writing, or updating. This method provides random access to the file records, allowing simple updates and quick retrieval of a particular record.

Operating systems that support ISAM files directly are usually found on minicomputers and mainframes. There is certainly a cost to pay in terms of overhead (both in processing requirements and in media space), but the advantages are generally thought to be worth the cost.

In the simple example that accompanies this article, an ISAM-type file is set up as a phone directory. The key field in each record is the name of a person or a company, and the rest of the record contains just a phone number. Now, if the prospect of another phone list program doesn't exactly make your floppies quiver, consider that you can easily define more fields for the file, and the key field can be anything you want it to be—a name, employee number, stock number, or nearly any other information upon which you'd like the file to be organized.

Taking that modification a step further, the index for the file could be checked to see whether it INCLUDES a particular key word—easy to do in Turbo Pascal—and a large file of The Computer Journal articles could be searched for a particular content.

In the large computer systems that support ISAM files directly, the key fields are checked against two or more kinds of index on the disk drive itself. The desired KEY FIELD is compared to the entries in a short sequential file called a CYLINDER INDEX, which lists the upper limits for the key fields that are located on a particular CYLINDER (the same track extended through all the recording surfaces of the disk-pack). Once the head access mechanisms have positioned the read-write heads to access a particular cylinder, then a track index is consulted to determine the track on which the desired record can be found, and the proper read-write head is electronically switched into action. Then it's just a matter of waiting until the right record rotates under the head, a delay called latency, or rotational delay.

A big advantage of ISAM files over ordinary sequential files becomes apparent when it's necessary to update one. A sequential file must be re-created in its entirety just to change a single record. Records in an ISAM file can be re-written, making it easy to correct fields in individual records. It's almost too obvious to point out, but applications that require access to a limited number of records from a large file are better served with random-access files (such as ISAM), so that all the previous records don't have to be examined in order to find the one that's needed.

Random Access On Micros

Most small computers have random access capability, but somehow the program needs to determine which record is to be read (or written). Random files can be DIRECT ACCESS FILES, in which case the program identifies the exact physical location (track and sector) where the record is located, or they can be RELATIVE FILES, where each record number is relative to the beginning of the file. The computer already must know how long each record is, and it can find the 29th record in a relative file by passing over the first 28, then accessing the next.

The trouble with this is, how is the computer to know that the address and phone number for XYZ, Inc. is in record number 127, or that the record for product number C-12285-BR is located on track 2A, sector 09? The answer, of course, is that it needs an INDEX, as in ISAM.

Using the relative files that are available with every disk-equipped computer, and accessible from every language, it's possible to duplicate the ISAM function with applications programming. This requires keeping a small sequential file on the disk that contains the key fields and corresponding record numbers of the main (relative) file. At the beginning of the program, and whenever the file is updated, the index file is read into a table in memory which can be searched to find the location of whatever record is needed. If a file is designed for direct-access, then the index must contain the key field and the physical track location where the record can be found. Since relative files are easier to work with and more versatile in some respects, this article will concern itself with simulating ISAM with Turbo Pascal relative files.

Program ISAM.PAS

The Turbo Pascal program that accompanies this article maintains an ISAM file of names and phone numbers in such a way that the user can type in a name (the KEY FIELD) and the computer will deliver up the phone number associated with it in the file.

This example program was purposely kept simple, but it would be an easy matter to expand the records in the ISAM file to hold whatever information is required for a more complex application. Once we've covered the example program in detail, I'll mention a way to make it even better, if you're so inclined.

Portability of Code

In keeping with my philosophy that a programming process (design, coding, testing) shouldn't be done twice if once will do, I've written the procedures and functions of the accompanying program with local variables and parameter lists (also called argument lists). Thus, these sections can be stored separately in a Pascal library and used whenever a programming project requires ISAM files. They can be read into the source code being developed, and made a part of the new program with relatively little effort. If GLOBAL variables (rather than passed parameters) are used, each procedure and function will need to be modified extensively to agree with the variables used in every new program.

Run-Time Narrative

When ISAM.PAS is compiled and run, it will look for an index file that shares the name of the ISAM file being used. The ISAM file will have an extension of .ISA, and the matching index file will end with .KEY. The first time the program is run, of course, neither of these files will exist, and the file-read sections are coded to understand this.

The menu that appears at the beginning of the program will offer the choices:

Q=Quit R=Read A=Add
D=Delete C=Change

ADD will write new records to the ISAM file. The user is prompted for a name to be used as the key field, then the phone number to be stored with it. The name and PHONE number go into the PHONE.ISM file, the name and the RECORD number (from the ISAM file) go into the PHONE.KEY file, and also into the table of keys that's maintained in memory. Trying to add a new record that has the same key field as an existing record is a big no-no for ISAM file processing, so an appropriate error message is displayed if this is attempted, and the addition is not made.

DELETE will not remove a record physically from the ISAM file, nor its key from the index, but it will replace the phone number with a message that says it's been deleted. This function of the program could be changed so that the record actually IS deleted (at least, used for the next ADD), but at a cost of additional complexity that isn't warranted in such an example. Trying to delete a record that doesn't exist is taboo, and earns the user a scolding from the computer.

CHANGE allows the phone number to be edited. In a more complex application you would want to allow editing of all the fields EXCEPT the KEY FIELD. If that needed to be changed, a DELETE would be appropriate instead. Naturally, an attempt to change a record that doesn't exist is useless, and will be trapped as an error condition.

QUIT is just a graceful way out of the program, back to the operating system. From the menu part of the program there are no files to close or other End Of Job processing to do, so it just means a quick trip to the end of the main logic. All the other functions re-run the main logic, so that a variety of tasks can be carried out while the program is running.

The Main Flow of Logic

I wrote this program knowing that I would be explaining it in detail in this text, so it isn't as full of comments as it ordinarily would be. Remember that you'll probably want to use this code again and again, so it wouldn't hurt to add additional comments as you enter the source. A few extra minutes of typing now can save hours of debugging later, when you want to use this method to access complex files in a new application—and that goes for all source code. (End of Lecture, I promise.)

Let's take a look at the mainline logic first, that last chunk of code that begins with BEGIN and ends with END. Because this program is written modularly, the main logic is short and simple. (It's not easy to write spaghetti-code in Pascal but believe it or not, I've seen it done!)

LoadKeys is a procedure that reads the .KEY file from the disk and loads the values found there into the two arrays IndexAray[1..200] and KeyAray[1..200] that make up the file index table in memory. The array limits of 200 were chosen entirely arbitrarily, and might be much higher in an application program, depending on need and the availability of variable storage space. You can see one purpose for the ReturnCode that many of these procedures send back to their calling module—if the appropriate .KEY file can't be found on the disk, the LoadKeys procedure sends back a return code of −, and the main logic understands to print an error message to that effect and bail out to STOP:.

START: is a label that tells the program where to go when a restart is required. This is done after all the menu functions except QUIT, which sends control instead to STOP: at the bottom.

The line that writes the heading and menu items to the screen starts out with a #26 as the first item printed. That's a screen-clear character for many computers, and could have been written just as well with a control Z. Turbo Pascal also offers a screen clear procedure, but this is shorter to type. Other Write() statements in this program use control M and control J to produce a carriage-return and a line-feed, respectively. Alternately, those could be coded as #13 and #10.

After Choice is read, it's changed to upper case with the UPCASE() function that's built-in. This makes it easier to write the subsequent CASE STRUCTURE so that upper and lower case selections will both be honored. The arguments of the case structure, of course, match the offerings from the menu line at the top of the screen, and all except 'Q' will execute a particular procedure written as a demo.

That's all there is to the main logic. Thanks to structured modular programming, a problem that's too complex to grasp all at one time can be broken down into separate tasks that are easily comprehensible. Now those smaller tasks will be broken down further into individual steps with an explanation for each.

Of Types, Variables, Functions, and Procedures

Those who are new to Turbo Pascal (and I hope that many of you are reading this) may be a little unclear about the use of the TYPE declaration. It's pretty easy to understand why Turbo (or even BASIC) wants to know whether a variable is to be treated as a real number or an integer, as the two are stored differently in memory and in files. They require different numbers of bytes and are treated differently by the compiler. Fact is, a Pascal programmer isn't limited to the pre-declared types of variables that most languages provide, but is free to

define types that are consistent with the present needs.

In this program, the procedures that access the files will need to be passed a parameter that tells them what file name to look for. Since the parameter can be called by one variable name on the sending end, and by something entirely different on the receiving end, the procedure must be able to identify it according to (1) its location in the parameter list and (2) its type. Because I've defined a TYPE called AnyFile, a string that's able to contain ten characters, I can pass a file name along from a "calling" procedure to a "called" procedure in a list of arguments. The argument list specifies to the called procedure that the file name is first in the list, and also what type of variable it is. This helps to make the code "portable", in that it can be used in various programs with little or no customizing needed.

Similarly, I'll be using a lot of string variables that contain a 30-character name and a lot of string variables that contain a 20-character phone number. These are defined as type NAME and type PHONE.

Even entire records can be defined as a TYPE, as shown by the next two entries. The records in the ISAM file will all contain a NAME and a PHONE (which were previously defined as types of their own, so these don't need to be spelled out as to how many characters, etc.) and the INDEX file will contain records that each contain a NAME and a record number, which is of the ordinary type INTEGER.

To digress for just a minute, if the number of records were to be limited to less than 256 for an application like this, it would be appropriate to use type BYTE instead of INTEGER for all the variables that have to do with record numbers and counters. Each variable would then require only one byte of storage or memory instead of two.

Now comes the familiar declaration for all the GLOBAL variables in the program, the ones that are to maintain their values from one procedure to the next. As you can see, there are some variables that are declared as ordinary types, such as INTEGER, and others that are declared as the special types that were defined above.

FINDEX()

FINDEX is my shorthand for Find Index, a function that searches through the table of key fields, and for each key requested finds the appropriate record number in the ISAM data file. It's a function, rather than a procedure, because of the way it's written and used. When the appropriate parameters are passed to this function, referring to FINDEX() is all that's needed to get an answer. One way of thinking of it is that you RUN a procedure, but you MAKE USE OF a function. To find the record number for a variable called NameIn, for example, from among MaxKeys number of entries, all we need to say is:

```
RecordNumber := FINDEX(NameIn,MaxKeys);
```

and we can use a function anywhere a variable would be at home, such as:

```
Write(FINDEX(NameIn,MaxKeys));
```

FINDEX begins by initializing COUNT to 0. Since COUNT is declared as a variable within the function FINDEX, it is LOCAL to this function. Even though there are other variables called COUNT elsewhere in the program, they won't get confused. This is one of the reasons why functions and procedures that use local variables are so portable from one program to another. FINDEXCode and FOUND, a Boolean truth flag, are also declared here, and exist only within FINDEX.

From there on, it's a relatively simple function, and even TurBeginners (sorry, I just couldn't resist, and this IS supposed to be a tutorial...) won't have any problems with the logic. The only line that's not entirely clear is the one that actually compares the given name with the names in the table. It starts out with:

```
If Copy(KeyAray[Count],1,Length(IndexKey))
    = IndexKey then ...
```

Rather than requiring the given name to be an exact match for the key field, I've written the program to allow a partial—BUT CORRECT—name to be used instead. The comparison is made to the key fields in the table only up to the length of the given name. That way, I can find the phone number for "Remote Measurement Systems, Inc." by typing just "Rem", if I'm sure that none of the other records starts with the same three letters. If there aren't any others that start with "R", I could just type that, but that would be taking a fair-sized chance. I need to enter enough of the name that there is no question which record is required, as the program will read the record that corresponds to the first match it makes.

LOADKEYS

LoadKeys is a procedure, so it's something to RUN, not USE. The program doesn't expect LoadKeys to take on a value like a function, although this procedure does, in fact, pass a ReturnCode back to the calling module. The return code goes back under its own variable name, not the name LoadKeys. The object of this procedure is to read the .KEY file from the disk and load the values it finds there into the two arrays that make up the index table in memory.

Following the procedure name is a good example of a parameter list. I'll discuss just this one in detail, and you'll be able to see the similarity between this one and the parameter lists in the other procedures.

First of all, if the procedure is to load a file into memory, it needs to know from which file to read. The file name COULD have been declared as a global variable, but then it would have to be done that way in every program that makes use of this procedure, and the procedure would be workable only with one file name per program. Instead, the file takes on the name that this procedure associates with the parameter FileName, which is defined as type AnyFile. The calling logic can place a literal value, such as a file name of 'PHONE' in the first spot in the parameter list, and the called procedure will find it there.

The other way of passing these parameters is shown next, with two parameters passed as variables. There are a lot of procedures in this program that make use of the variables MaxKeys and ReturnCode, which have already been explained, and it's easiest to pass those back and forth as variables in the parameter list. I happen to use MaxKeys and ReturnCode for the very same purposes each time I use these procedures in other programs, so that's not difficult. You'll notice that the keyword VAR precedes these variables in the parameter list, just as it would in an ordinary variable declaration. Of course, if I intend to have a program work with multiple ISAM files, I have to be sure that MaxKeys is assigned properly before this procedure is called. ReturnCode gets its value within the procedure and passes it back to the calling module, so it doesn't have to be initialized before use.

Since this procedure accesses a disk file, it makes use of the Turbo compiler directives {$I−} and {$I+} which turn off system disk error handling and turn it back on again, respectively. If there's a problem like "file not found", I want to deal with it from within this program, not have the operating system crash the program on that account. After the error-checking is turned off, an attempt to open a buffer for the file using the statement Reset(IndexFile) is all that's needed to see whether the file exists. The Turbo Pascal function IOResult will return a value of zero if everything was all right, or an I/O error number if it wasn't. If things went well, this procedure turns error checking back over to the system and continues. If they didn't, a value of −1 is placed into ReturnCode, the parameter that will be passed back to the module that called LoadKeys, and control is passed to the label RETURN: at the bottom of this procedure. That calling module is written so as to understand that a ReturnCode of −1 means the .KEY file wasn't there.

About the only other obscure code in this procedure is the part that starts out with the line:

```
With IndexRec Do ...
```

IndexRec is defined as a variable of type INDEX, which means that each IndexRec actually contains the variables ISAMName and ISAMPhone, which are of type NAME and PHONE, respectively. In order to access these "buried" variables, we need to enclose all references to them between statements like the above, and an "End;". Otherwise, the compiler will swear that it doesn't know what variables we're talking about. Trust me, this is only confusing at first—it becomes pretty intuitive after you've used records this way a few times. The convenience of being able to refer to entire records by one variable name (such as when reading or writing to a file) is wonderful. By the way, the WITH statements can be nested, and if you write some lines that need to access variables that are shared by more than one record at a time (like a variable that's stored in two different files), you can write it such as:

```
With IndexRec, DataRec Do
    ...
    ...
    ...
End;
```

It's pretty clear, then, that this procedure reads the file of the same name as the ISAM file, but with an extension of

```
Program ISAM;

{ Demonstration of reading and writing an ISAM-type file with Turbo Pascal }

Label   Start, Stop;                    {labels used in main logic only}

Type    AnyFile = String[10];           {declaration of non-standard types}
        Name    = String[30];
        Phone   = String[20];
        ISAM    = Record
                    ISAMName  : Name;
                    ISAMPhone : Phone;
                  End;
        INDEX   = Record
                    KeyField : Name;
                    RecNum   : Integer;
                  End;


VAR     INDEXFile : File of INDEX;      {GLOBAL variables that retain their}
        ISAMFile  : File of ISAM;       {values throughout the program     }
        INDEXRec  : INDEX;
        ISAMRec   : ISAM;
        IndexAray : Array[1..200] of Integer;
        KeyAray   : Array[1..200] of Name;
        Choice    : Char;
        NameIn    : Name;
        PhoneIn   : Phone;
        MaxKeys   : Integer;
        ReturnCode: Integer;

Function FINDEX(INDEXKey:Name;Max:Integer) : Integer;
Label   Return;
VAR     Count : Integer;                {LOCAL variables that exist only}
        FINDEXCode : Integer;           {within the function FINDEX()   }
        Found : Boolean;
Begin
   Count      := 0;
   FINDEXCode := 0;
   Found      := False;
   With INDEXRec Do                     {'opens up' INDEXRec to get access}
      Begin                             {to the variables inside it       }
         Repeat
            Count := Count + 1;
            If Copy(KeyAray[Count],1,Length(IndexKey)) = IndexKey then
               Begin
                  FINDEXCode := IndexAray[Count];
                  Found      := True;
               End;
         Until ((Found = True) or (Count >= Max));
      End;
Return:
   If (Found = True) then FINDEX := FINDEXCode Else FINDEX := -2;
   End;


Procedure LoadKeys(FileName:AnyFile;Var MaxKeys:Integer;ReturnCode:Integer);
Label   Return;
Var     Count : Integer;
Begin
   Assign(IndexFile,FileName + '.KEY');
{$I-}                                   {turns off system error-checking}
   Reset(IndexFile);
{$I+}                                   {error-checking back on here}
   If IOResult <> 0 then                {checks whether any error happened}
      Begin
         ReturnCode := -1;
         Goto Return;
      End;
```

.KEY, and puts the names it finds there into an array called KeyAray[1..200] and the record numbers into another array called IndexAray[1.200]. These two arrays make up the table in memory that is searched to find the appropriate record number for any name that's requested.

From this point on, I'll offer less explanation for each procedure, just pointing out the logic that may not be obvious, and the syntax that might be unclear to someone starting out with Turbo.

READISAM()

ReadISAM is the procedure to read the ISAM file, and the arguments (parameters) passed to it include the file name and the key field being sought, with the third argument—the ReturnCode—that goes back to the calling module to identify an error.

First the function FINDEX() is called, and FINDEX() will contain either the value of the record number for the key field that's wanted or a return code that indicates an error. If it's an error (less than zero), ReadISAM just goes back where it came from with ReturnCode the same as what came from FINDEX(). If it's a legitimate record number instead, then the ISAMFile is accessed and a record read from ReturnCode −1. The reason for the −1 is that records are numbered in a Pascal file starting at zero, but we tend to think more logically when the first record is numbered "one". If this little irregularity is tolerated here, then all the rest of the logic can believe that the first record is #1, the second is #2, and so on.

If the ISAM file isn't on the disk, the ReturnCode is changed to −3 before control is returned to the calling module, letting it know what the problem was. (A ReturnCode of −2 from FINDEX() meant the key field wasn't found when the index was searched. This way, the original calling module can be coded to respond to whatever number of error conditions is necessary.)

WRITEISAM()

WriteISAM is the procedure to write a record to the ISAM file. The parameters it needs are the file name and the key field. The only error condition that's anticipated here is the possible absence of this file from the disk, and that's not really an error—it just means we haven't written the first record yet. Thus the code here tries to RESET the ISAM file, and if that doesn't work it will REWRITE the file, starting it from scratch. The same logic follows when the procedure writes the key field and record number to the .KEY file. If there's an error, the .KEY file hasn't been started yet and it's handled the same way.

Along with writing the record to the .KEY file, the same information is added to the index table arrays, after incrementing MaxKeys appropriately. Thus, whenever a new record is written to the ISAM file the index is updated in the .KEY file and in memory at the same time. The index table will remain valid without having to read the .KEY file into memory again.

REWRITEISAM()

RewriteISAM is very similar, but the differences are important. In this case an error that will need to be dealt with is very possible, so one of the parameters is again ReturnCode. Since RewriteISAM is used to update a record, not write one in the first place, it is indeed a problem if the file doesn't exist on the dis'-. Also, in this case we don't want to seek the end of the file and write a record there, we want to seek the position of the requested record and rewrite that same record. Remember to subtract 1 from the record number—obtained from FINDEX()—to account for records that start at zero.

Having written all the code that's required to accomplish these basic tasks, making the program do something useful is easy. The following procedures are the demonstrations, and I'll continue to point out ususual syntax and logic, but skip the simple stuff.

READ DEMO

Intended to be used in this program only, ReadDemo doesn't receive any parameters from the main logic which calls it, but it must pass some parameters to the procedures that it calls. First off, ReadDemo asks for the name that identifies the record to be read. Then it calls ReadISAM to read the right record from the file, and displays the phone number to the CRT. Along the way it is prepared to deal with values of the parameter ReturnCode that indicate the following problems in reading the file:

Return Code	Problem
-2	Key field not found when the index was searched.
-3	The ISAM file didn't contain a record for that record number.

(A return code of −1 was used by LoadKeys to indicate that the .KEY file wasn't found. Though these errors could have been called −1 and −2, I prefer to associate certain ReturnCode values with specific problems, so I don't usually re-use the same numbers, even in different procedures.)

The statement "Read(Choice);" is used just to hold th displayed phone number on the screen until the user e ters a RETURN.

ADD DEMO

Because all the nit-picky details are handled by th procedures that were coded earlier, this one to demor strate adding a record to the ISAM file is very short an sweet. It asks for the name (key field) and the phone number, then places them into the record for the ISAM file and writes it. Notice the section of code that starts out with the words "With ISAMRec Do".

The problem that's possible when a record is added, of course, is that there might be a duplicate of that key field in the index already. If we were to enter a phone number for someone, forgetting that there was already an old and incorrect one there, the computer would never deliver unto us the correct phone number. Each time the index was searched and a matching key field found, it would point to the record containing the old number.

Therefore, the proper way to change a field is with the CHANGE feature, not by adding another record with the same key field. This procedure does a quick check of the index—using FINDEX()—to be sure that a key field doesn't already exist before a record is added to the file.

```
   Count := 0;
   With IndexRec Do
      Begin
      Repeat
         Begin
            Read(IndexFile,IndexRec);
            Count := Count + 1;
            KeyAray[Count] := KeyField;    {load values from .KEY file into}
            IndexAray[Count] := RecNum;    {arrays used for INDEX table    }
         End;
      Until EOF(IndexFile);
   End;
   MaxKeys := FileSize(IndexFile);      {determine how many records in file}
   Return:
   Close(IndexFile);
   End;


Procedure ReadISAM(FileName:AnyFile;ReadKey:Name;Var ReturnCode:Integer);
Label  Return;
Begin
   ReturnCode :=  FINDEX(ReadKey,MaxKeys);
   If ReturnCode < 0 then Goto Return;
   Assign(ISAMFile,FileName + '.ISA');
{$I-}
   Reset(ISAMFile);
{$I+}
   If IOResult <> 0 then
      Begin
         ReturnCode := -3;              {file not found - bail out}
         Goto Return;
      End;
   Seek(ISAMFile,ReturnCode -1);        {locate the right record}
   Read(ISAMFile,ISAMRec);              {read it, then close file}
   Close(ISAMFile);
Return:
End;

Procedure WriteISAM(FileName:AnyFile;ISAMKey:Name);
Label  Return;
Begin
   Assign(ISAMFile,FileName + '.ISA');
{$I-}
   Reset(ISAMFile);
{$I+}
   If IOResult <> 0 then Rewrite(ISAMFile);
   Seek(ISAMFile,FileSize(ISAMFile));
   Write(ISAMFile,ISAMRec);
   Assign(IndexFile,FileName + '.KEY');
{$I-}
   Reset(IndexFile);
{$I+}
   If IOResult <> 0 then Rewrite(IndexFile);
   Seek(IndexFile,FileSize(INDEXFile));
   MaxKeys := MaxKeys + 1;
   With INDEXRec Do
      Begin
         KeyField := ISAMKey;
         RecNum := FileSize(ISAMFile);
         Write(IndexFile,INDEXRec);
         KeyAray[MaxKeys] := KeyField;
         IndexAray[MaxKeys] := RecNum;
      End;
   Close(ISAMFile);
   Close(IndexFile);
Return:
End;

Procedure ReWriteISAM(FileName:AnyFile;ISAMKey:Name;Var ReturnCode:Integer);
Label   Return;
```

```
Begin
   ReturnCode := 0;
   Assign(ISAMFile,FileName + '.ISA');
{$I-}
   Reset(ISAMFile);
{$I+}
   If IOResult <> 0 then
      Begin
         ReturnCode := -1;              {file not found - bail out}
         Goto Return;
      End;
   Seek(ISAMFile,FINDEX(ISAMKey,MaxKeys)-1);
   Write(ISAMFile,ISAMRec);
Return:
   Close(ISAMFile);
End;

Procedure ReadDemo;
Label    Return;
Var      ReturnCode : Integer;
Begin
   Write(^J^J^M,'Enter Name to Read: ');
   Readln(NameIn);
   ReadISAM('PHONE',NameIn,ReturnCode);
   Case ReturnCode Of
      -2 :  Begin
               Writeln(^J^J^M,'Error - KEY NOT FOUND IN INDEX');
               Goto Return;
            End;
      -3 :  Begin
               Writeln(^J^J^M,'Error - RECORD NOT FOUND in ISAM FILE');
               Goto Return;
            End;
      End;
   With ISAMRec Do Writeln(^J^J^M,'Phone Number:      ',ISAMPhone);
   Write(^J^J^M,'Press <RETURN> to Continue... ');
Return:
   Read(Choice);
End;
```

If a ReturnCode comes back that indicates the key wasn't found, it doesn't imply an error this time, it means everything's OK to proceed. Isn't it great the way we can use a function like that whenever it's needed!

CHANGE DEMO

ChangeDemo is very similar, but this one requires that a key field MUST exist before it's legitimate to continue. Obviously, it isn't possible to change a record that hasn't been written to the file yet, and any attempt to access a nonexistant record would certainly result in an I/ error that would crash the program but good (attempt to seek beyond EOF).

DELETE DEMO

By now, this one last procedure needs little or no explaining. It simply does a re-write of a record, replacing the phone number with a message that says "*** Deleted ***". In actual practice on large computers, records are not really deleted physically from an ISAM file by an application program, but simply marked with a user-selected "delete character" in a user-selected position in the record. A system utility program is occasionally used that will put the file back into physically-sequential order, working in all the records that have been added, and leaving out the ones that were marked for deletion.

Assuming that deleted records won't amount to enough disk space to cause problems, this procedure might be as far as you would want to go in your own applications. In fact, in some cases, it might even be an advantage to leave deleted keys in the index. In this program, for example, a phone number can be deleted if it turns out to be a wrong number—no point in continuing to use it—but then the deleted field can be replaced with the proper number at a later time by using the CHANGE feature.

Closing Comments

That's about all you might want to know about this program to simulate Indexed Sequential Access Files with your micro. If you're NOT a beginner, though, you may have spotted some possible improvements already, and I couldn't let you just sit there and smirk without pointing out that I, too, have thought of some. I'll take up just a little more room to point out one potential upgrade, and I welcome suggestions from anyone. This program is surprisingly fast and capable, but it can be made even more so at a cost of somewhat increased complexity.

Binary Search

If there are to be a limited number of records in the ISAM file, say, a couple of hundred friends' phone numbers and addresses, a sequential search of the index array (such as in FINDEX) will be tolerably fast, and requires very little room in the program. If this principle is to be applied to an application program that will keep track of thousands of records, then it would be worth the

```
Procedure DeleteDemo;
Label Return;
VAR   ReturnCode : Integer;
Begin
   Write(^J^J^M,'Enter Name to Delete:  ');
   Readln(NameIn);
   If FINDEX(NameIn,MaxKeys) <0 then
      Begin
         Writeln(^J^J^M'Error - KEY NOT FOUND IN INDEX - Cannot DELETE');
         Delay(3000);
         Goto Return;
      End;
      With ISAMRec Do
         Begin
            ISAMPhone := '*** Deleted ***';
            ReWriteISAM('PHONE',NameIn,ReturnCode);
         End;
Return:
End;


{ A BEGIN like this, without a procedure or function name, indicates the  }
{ start of the main program logic.  The program actually starts here, and }
{ calls previously-coded procedures and functions as needed.              }

Begin
   LoadKeys('Phone',MaxKeys,ReturnCode);
   If (ReturnCode = -1) then
      Begin
         Writeln('Unable to locate KEY file... returning to system.');
         Goto Stop;
      End;
Start:
   Write(#26'  ISAM Phone File Example   [ Q=Quit R=Read A=Add D=Delete C=Change
 ]   ');
   Readln(Choice);
   Choice := UpCase(Choice);
   Case Choice Of
      'Q' : Goto Stop;
      'A' : AddDemo;
      'R' : ReadDemo;
      'D' : DeleteDemo;
      'C' : ChangeDemo;
   End;
   Goto Start;
Stop:
End.

Procedure AddDemo;
Label Return;
Begin
   With ISAMRec Do
      Begin
         Write(^J^J^M,'Enter Name To Add: ');
         Readln(ISAMName);
         If FINDEX(ISAMName,MaxKeys) >0 then
            Begin
               Writeln(^J^J^M,'Error - DUPLICATE KEY - Cannot ADD ');
               Delay(3000);
               Goto Return;
            End;
         Write(^J^J^M,'Enter Phone:       ');
         Readln(ISAMPhone);
         WriteISAM('PHONE',ISAMName);
      End;
Return:
End;

Procedure ChangeDemo;
Label    Return;
VAR      ReturnCode : Integer;
```

```
Begin
   Write(^J^J^M,'Enter Name:  ');
   Read(NameIn);
   If FINDEX(NameIn,MaxKeys) <0 then
      Begin
         Writeln(^J^J^M,'Error - RECORD NOT FOUND - Cannot CHANGE');
         Delay(3000);
         Goto Return;
      End;
   ReadISAM('PHONE',NameIn,ReturnCode);
   With ISAMRec Do
      Begin
         Write(^J^J^M,'Old Number is:      ',ISAMPhone);
         Write(^J^J^M,'Enter New Number:  ');
         Read(ISAMPhone);
         ReWriteISAM('PHONE',NameIn,ReturnCode);
      End;
Return:
End;
```

extra programming cost to use a binary search of that index instead.

Actually, two major changes would be needed. The index would always need to be sorted into ascending or descending order (based on the key fields),to use a binary search. This sort would have to be done each time a record is added to the file, before the index is accessed the next time.

Second, the search itself would have to be re-written a little. The details of all this are a little beyond this particular article, but for those budding hackers who haven't already added the binary search to their bag of programming tricks, I'll give a few hints, at least.

Begin by determining how many entries are in the array (MaxKeys, in this case). Assign a variable called FINISH the value in MaxKeys to start with, and initialize a variable called START to 1. Add START to FINISH and divide by 2 to find the middle of the array. Compare the value in the middle to what you're looking for, and see if it matches. If it does, the search is over already.

If it doesn't, determine whether the value you found is HIGHER or LOWER than the one you're looking for (that's why the array has to be sorted!). If it's higher, set FINISH to a value that's one less than the subscript you just checked. If it's lower, set START to one more than the subscript you just checked. Now, start over again with these new values for START and FINISH. Sooner or later, that middle element you check will be the one you're looking for.

It usually turns out that it's sooner than you expected—a binary search is FAST. The reason is very simple (and you'll probably catch on real quick to why it's called what it is). After one comparison, you're able to eliminate fully one-half of the possible elements as being either higher or lower than the one you're looking for. After the second comparison, you're able to eliminate half of the ones that were left. It turns out that it takes only 16 comparisons (or fewer!) to find an element among an array of 65,536 elements. Using a linear search, it would take—on average—16 comparisons to find an element among an array of only 32 elements.

In actual practice, you won't find yourself getting bored waiting for this program to find the right record number for any reasonable size index, and the binary search really shows its stuff the best when the numbers are very large. On the other hand, a quick sort done only after adding a new record to the ISAM file doesn't add a lot of overhead, either. Just be sure to re-write the sorted index file as part of the EOJ (End Of Job) processing as you quit the program, so it won't have to be sorted again each time the program is used. After all, you might access a file thousands of times without adding new records to it, depending on the application.

(By the way, a QuickSort is a real jewel in Pascal, using recursion. That might be a good subject for another article...) ■

Letters

(Continued from page 4)

Still More On Soldering

James O'Connor:

I enjoyed your article in the September/October issue.

I think that soldering dates back to the Egyptians—for metals—and the application to electrical circuits probably dates to Faraday and his contemporaties. Electronics is just a minor off-shoot of electrical engineering.

Your differentiation between welding and soldering is a bit extreme. Most welding (not all) involves using a "welding rod" which is melted and flowed between the component parts to be joined. The rod usually melts at a lower temperature than the parts, like solder, but yes, the component parts melt too.

On the other hand, metals when soldered do dissolve into the solder, a process which is similar to melting. When thin films of solder are used to join metals structurally (brazing), you will find that desoldering requires a higher temperature than the initial joining because of the change in composition of the solder due to dissolved metals. The change is a couple hundred degrees in silver soldered/silver brazed stainless steel.

This effect is not very noticeable in most electrical connection soldering because the quantity of solder used is so great, compared to the surface area of the joint, that the concentration of dissolved metal from the joined surfaces never gets very high. However, it does become noticeable at times, such as desoldering a DIP or socket from a board, where the pins are reasonably tight in the holes in the board AND the plating/conductor on the board lines the walls of all holes.

Personally, I use a 7½ watt iron for all my electronics work. I have a high power iron (15 watt) for use on heavy duty connections—8 gauge wire to a lug on a transformer, etc.

I'm surprised that you made no mention of the gallium and indium based special electronic solders. While the hobbiest isn't liable to use these, he should know that they exist, and that there are problems in trying to bond lead-tin to the speciality solders or to parts "contaminated" with them.

In your review of "Sources", you might add a mention that Heath sells a kit for learning how to do elementary electronic soldering—a board, parts, and instruction manual. That kit was my starting point and I still have the manual.

Dave English
Orange, CA

Dave:

Thanks for the comments, Dave. Actually the Egyptians were also the first to do VLSI (Very Large Scale Integration) they just happened to use large cut stones instead of logic gates. But seriously, I wanted to delineate welding from soldering because I have encountered novices who were sure that soldering was just a wimpy form of welding. The result of that misconception was a tendency to overheat connections in the belief that insufficient heat was the cause of their problems. In the old days welding and forming were done concurrently so that the high heat aided both processes, for example the gun barrels of the famous Brown Bess muskets carried by Revolutionary War soldiers were formed and welded from a sheet of flat metal by means of the high temperature of the gunsmith's forge and a forming block. Today changes in shape would be a deformity so the welding rod which melts at a slightly lower temperature prevents that. As you correctly point out, it is still welding that occurs, but more on the surface than throughout the pieces. Supplying and controlling the precise amount of heat is very similar to soldering.

Yes, indeed there is a chemical bond formed during soldering and it can affect the temperature required to de-solder a connection. For electronics work my experience has been that this bond is not much of a problem if most of the residual solder can be removed. As you note there are situations where that is difficult due to tight quarters. As as aside, this bonding effect is also present with chemical 'glues' and shows up dramatically in that most glues produce the strongest possible bond with the thinnest possible glue layer. So-called 'Super Glues' (cynoacrylate) exhibit this to the extreme, remember the TV ad where a man is suspended from a beam with just one drop, if they had used two drops he might have fallen. This is directly opposite to the intuitive feeling that if a little is good then more is better.

7½ Watts!!! I had heard of such low power irons but had to check my catalogs to determine that, yes, they are available. Your successful use of them is testimony to the degree of proficiency you've developed. Size and shape of the iron's tip along with the wattage rating ultimately determine the tip's temperature so there is a wide range of suitable irons that can be used, still I would stick with the recommendations made in the article for anyone just getting into soldering.

Specialty solders are used by people who do craft type metalworking and in industrial applications but in my experience are very rare in electronics work. Fortunately, most articles that I have read about these solders do a good job of explaining where, when, why, what and how to use them and I endeavored to model the series on electronics soldering on this principle. The companies that sell these solders also provide good technical brochures about them.

Good suggestion about the Heath Soldering Course, Catalog No. EI-3133 list price $19.95, I was not familiar with this particular course. Heath's educational courses are invariably a good investment. In some respects the Soldering article and the two follow-on articles cover much the same material, so Computer Journal readers may want to use them as a psuedo course in soldering. Also, those familiar with Heath's kits will probably detect the fact that I diverge from and expand upon Heath's usual instructions about soldering. By all means, CJ readers should obtain a copy of Heath's catalog and feel confident about attempting any of the kits or educational products.

James O'Connor
Randolph, MA

Art: Your prompt response is very much appreciated.

Good Grief! Looking at your back issue list is like finding buried treasure; what a kick. I'm enclosing a check for all the listed back issues and a two year subscription.

May you prosper.

Interests, Equipment, etc. We're principally Z-80, CP based. There are two homebrew units and an Ithaca Intersystems system with front panel. Also, have a little Z-8 (Circia) and a yet-to-be-assembled MicroAce. There's a Godbout 68K system languishing in storage as a result of an abortive "bright idea"; the collapse of the oil business had much to do with it. I'm interested in wringing the maximum from the Z-80/S-100 system and am looking for a reason (excuse?) to make a foray into 68K country. Am determined to know what goes on inside and look to your journal to aid and abate tje quest.

Thanks.

Chuck Henderson
Midland, TX

Remax Drives

Bill Kibler:

I just read your column in the September-October issue and immediately picked up on a comment about Remex disk drives. You referred to modified drives. I bought a couple of them and have had nothing but trouble. I used them on an AMPRO Little Board and found that it was almost impossible to verify a disk. I concluded that there was something sloppy about either the mechanics or the electronics or both, but didn't quite know what to do about it.

Could you let me know what you did to modify the beasts? I enclose a SASE for a reply. I certainly will appreciate it.

Frank Oechsli
Richmond, CA

Dear Frank:

Thanks for the fan mail, and hopefully this information will get you going. Please send us any NEW information you might get after playing with your drives.

The Remex Disk drives that are currently available for as little as $39 have many problems, but can be made useable. In the July-August issue of Computer Journal I covered several important solutions to the problems. Other magazines have also commented on some solutions (Micro Cornucopia #26), which agree in part with my findings. These drives appear to be made rather marginally and without much quality control. A solution may involve one or more of the following:

A) The 12 volt supply needs are much larger than most, and will exceed the listed ratings. Limiting the number of drives per power supply, as well as adding more capacitance (2000ufd) has cured poor stepping problems.

B) Problems with noise have been noted, and in one case shielding with steel helped. I found that adding more bypass caps (.022) can also improve their operation (the PC board doesn't have enough).

C) Precompensation has been noted as causing problems. Use 125ns or none at all. The schematic shows some minor value changes in the read circuit design and I suspect the design is inadequate. Adjusting R27 can improve the operation, sometimes. **PLEASE NOTE** that most of my errors have been READ not WRITE errors.

D) Several signals are fed directly off of the 12V line and without proper filtering will cause false write triggering and possible speed problems. Adding 0.01 and 10-20ufd across the input power socket can help.

E) Speed regulation is controlled by an MJE210 and this device is insulated from the case by a mica washer. Excessively tightened screws will short this device and cause the drives to run full on (loosen screw). I had one of the transistors fail completely and replaced it.

F) The guides or rails on which the head rides can become sticky and may need cleaning and (lightly) oiling. Head alingment has checked out good (usually), with only azimuth being occasionally off (play with rails to correct).

G) Check all mechanical and electrical connections as several units have had broken, loose, and poor connections. I found one loose wire on a motor unit and had to "recrimp" it.

Remember, these units most likely have poor quality control, and any type of problem could be encountered. The design appears to be minimum, and external help (larger and cleaner supply voltages) will be necessary.

One of the most important points in fixing the problems with these drives is networking. This means talking and writing to others with both questions and solutions. Only through networking can both manufacturers and users keep poorly designed units either off the market or at least running.

Best of luck.

Bill Kibler
Sacramento, CA

Reconsider TCJ's Emphasis

Your letter seems to indicate an emphasis on robotics, real time data collection and process control, at the expense of "business" topics— yet your back issue topics suggest a much broader base. Please consider that "system integration" issues in the vertical market business world are also required for practical applications on the factory floor, and thus are a common area of interest that can expand your subscription base. As micros move into the sphere of large MIS departments, DP types are having to cope with systems subjects formerly handled by small outside systems houses. This is a big market for you! Subjects include multi-user, LANs, hardware integration and interfacing, micro multi-user operating systems, database, etc. The recent launch of Multi-User magazine was a disappointment. Maybe you can fill the gap.

L.A. Wilkinson
Van Nuys, CA

Editor's Note:

I appreciate this kind of feedback, see this month's editorial for my comments. ■

The AMPRO Little Board Column

by C. Thomas Hilton

Whether you are a hobbyist, or commercial developer, often the desire for a small, yet powerful, single board computer, (SBC), presents itself. Often the choice for a good SBC is between commercially available boards, or in-house manufacturing. This column will present my personal choice for many SBC requirements, the Ampro Series 100, or "Little Board®," microcomputer. This system is no larger than a 5.25 inch disk drive, so it can be mounted in the drive enclosure, and it operates with very low current demands.

The Little Board is available in a number of forms. It is available from Digital Research Computers as a kit, and in a number of different configurations from Ampro Computers. (See Figure One for vendor listing).

AMPRO Computers, Inc., 67 East Evelyn Avenue, Mountain View, CA 94039 (415) 962-0230. THE source for manuals, systems, and parts.
Digital Research Computers Of Texas, POB 461565, Garland, TX 75046 (214) 225-2309. Little Board kits and ZRT terminal card.
Integrand Research Corp., 8620 Roosevelt Ave., Visalia, CA 93291 (209) 651-1203 Cases, power supplies, and wiring harness.
Colonial Data Services Corp., 80 Picket District Road, New Milford, CT 06776 (203) 355-3178. Disk drives and accessories.

Figure 1: AMPRO Series 100 and Little Board source list

When considering SBCs for industrial, instrumentation, or other application, one must consider how he will develop the system. That is, what manner of "development system" is also required to use the board in question. I am using an Ampro Model 122 with my Kaypro 4-84 as a development system. The '122 is a dual 48 track per inch, (48 tpi), microcomputer. It is a "terminal system." This means that it requires a terminal to communicate with it. I use the Kaypro as a "dumb" terminal for development of Ampro projects.

The Little Board is a 64K, Z80, system capable of supporting up to four Double Sided, Double Density, (DSDD) disk drives with a standard Basic In/Out System, (BIOS), or up to 88 megabytes of hard-disk storage, with a modified bios. My Kaypro can handle only 10 megabytes of hard-disk storage. With a terminal the Little Board is far more powerful, and versatile, in my opinion, than the Kaypro. In time, if Art allows me to continue this column, we will explore the Little Board in great detail. At this point, however, we will assume that you are either looking for a well supported SBC, and have chosen the Ampro Series 100, or I have convinced you of how much computing enjoyment can be had with a Little Board. This column will begin with the assumption that you have either just received an Ampro Series 100, or Little Board. We will not repeat the material covered in this column. New Ampro users should be advised to obtain back issues of The Computer Journal to keep abreast of all projects. If reader interest warrants, user disks, with these articles and public domain software tools to perform programming projects, will be made available. By providing public domain software tools, at reasonable acquisition costs, for use with this series, we will have some consistency. I will not hide from my readers. I will review reader contributions for this column, and answer any questions I receive, quickly, and personally. Though I have no connection with Ampro Computers, I feel my function is to assist you in the understanding of, and use of your Ampro SBC.

The Ampro Series 100 systems have two serial ports, a printer port, and a disk expansion port. Owners of the "Little Board Plus" also have a SCSI port which can be used to interface to a hard disk and other devices. In this series we will deal primarily with the generic 1A CPU card as found in the inexpensive Little Board kits, and Series 100 microsystems.

Out of the box the Series 100 "Bookshelf Computer," the recommended development system, is ready to run once a terminal system has been prepared for its use. Most people interested in SBCs have a system that can be used as a terminal. Again, I use a Kaypro 4-84. In a like manner most people have an assortment of cables in reserve. I am a bit on the cheap side, and prefer to make my own cables. Constructing a cheap and simple cable will be the first project in this series. This is a bit pedestrian for you old salts, but there are a lot of beginners out there who need a little help as well.

1. DB-25 RS 232-C Male Connectors, 2 each (Radio Shack #276-1547)
2. Six Conductor Cable, (Or Ribbon Cable) (Radio Shack #278-722 is usable)
3. Soldering Iron & Solder
4. Small hand tools, tweezers, etc.

Figure 2: Materials required.

Cable Construction

Assemble all of the materials noted in Figure two. Before we get too carried away, let's force ourselves to tolerate a little theory. The Ampro communicates with the terminal via RS 232-C ports. In normal operation "Port A" is the terminal port. This port is wired as "Data Communications Equipment," (DCE). This is the format we will use in building the cable. The Ampro end of the cable must be wired as DCE, but your specific terminal may require a "Data Terminal Equipment," (DTE), format. Standard communications cables consist of 25 signal paths. Not all of these 25 lines of a standard cable are supported by the Ampro/Little Board. We need only six actual signal paths to get our system on-line:

1. Data Input
2. Data Output
3. Data Terminal Ready
4. Clear To Send
5. Signal Ground
6. Equipment Ground

In the list above, Data Input refers to serial data being received by the Ampro from the terminal. Data output refers to serial data being sent to the terminal. Data Terminal Ready and Clear To Send are "hand-shaking" signals to assure that the terminal is listening when it should, and vice versa with the computer. The signal ground is the common return path for all data signals. The equipment ground path is generally used to shield the cable, and assure that both the terminal and computer are of equal electrical potential.

In this construction project we will use an "A" to indicate the Ampro end of the cable. "K" will indicate the Kaypro end of the cable.

A statement such as "K2" refers to the Kaypro end of the cable, pin number two. In a like manner a reference to "A20" would refer to the Ampro end of the cable, pin 20. Place a mark in the space provided when you have completed each assembly sequence.

Prepare your connectors and cable for assembly at this time, marking one end with an 'A' for AMPRO END and the other end with a 'K' for KAYPRO END.

☐ 1. Connect K1 to A1 (One conductor to the same pin on each end, starting on the Kaypro end.)

☐ 2. Connect A2 to K2

☐ 3. Connect K3 to A3

☐ 4. Connect A5 to K5 (pin K4 and A4 are not connected)

☐ 5. Connect K6 to K8 (On Kaypro side ONLY, here we define the unused DCD input state as active)

☐ 6. Connect K7 to A7

☐ 7. Connect A20 to K20

At this stage of the assembly only the following pins should be connected, all others should be left vacant, unconnected.

Ampro End: 1,2,3,5,7,20

Kaypro End: 1,2,3,5,6,7,8,20

There are six connections on the Ampro end because there are two ground paths. While only one ground path needs to be used, we will use two. One is defined as equipment ground, the other as signal ground. In both the Ampro and the Kaypro these lines are tied together. Later we may wish to use this cable on another terminal where the two ground systems are different, hence we use the extra wire now, to avoid problems later.

The Kaypro has two extra connections as pin 6, signal power path, (+5 volts), is used to define the unused input DCD, pin 8, to its active state.

A good technician triple checks his work, even when he knows he has done the work properly. Triple check your connections now.

☐ 8. If you are using metal connector shields AND your cable has a metal braid type shield around the wires, connect the braid to the metal shield. If you are uncertain, ignore this step.

☐ 9. Insert the Kaypro end of the cable into the SERIAL DATA PORT in the rear of the Kaypro. Consult your Kaypro manuals if uncertain as to which connector this is.

☐ 10. Insert the Ampro end of the cable into Serial Port 'A' at the rear of the Ampro. Consult your Ampro manuals if uncertain of the port's location.

Assuming all of your cable connections are correct, and you are proud of your work, allowing no sloppy soldering, the assembly of your terminal cable is now complete.

Going On Line:

The Ampro Series 100, Serial Port 'A' is configured for 9600 baud without hand shaking, as a default assignment. Nearly all of the Kaypro communications programs to be found do not support hand shaking, nor do the programs found in Kaypro support magazines. Hand shaking is very important in data communications. Without it characters will be lost in transmission and data transmission speed will have to be reduced in an attempt to achieve reliability. The next segment of this article will concern itself with a high speed, but simple "dumb" terminal program.

Most Kaypro communications programs may not be used above 2400 baud. This is due to the programmer's assumption that higher speeds were not possible. In fact their programs did not implement the system's hand shaking facilities. KTERM may be run at 19200 baud. Higher speeds are possible, but the Kaypro CONFIG program only allows up to 19200 baud. This is more than fast enough to get the full power from your Ampro Series 100 microcomputer.

KTERM is available on disk preconfigured for easy installation of the Ampro Series 100. Lacking that, compile the Turbo Pascal, Version 3.0, program shown in Figure 3. Once you have assured that it has compiled properly, place it on a newly formatted, Kaypro disk. Add to this disk the Kaypro CONFIG program. From this point onward it is assumed that you have studied both the Kaypro and Ampro System Manuals. If you have not studied them, do so now.

The Kaypro "Serial Data Port" is connected to a Serial In/Out processing chip, (SIO). This SIO is referred to as SIO1 on the Kaypro Series 84 systems. SIO1, Channel 'A' is not supported by the Kaypro BIOS. It is, however, initialized on power-up as the keyboard uses Channel 'B' of this SIO. Our Program initializes Channel 'A' for its own use, without disturbing Channel 'B.'

The Kaypro's default modem data transmission rate is 300 baud. It will have to be changed to the rate of 9600 baud. This will allow us to communicate with the Ampro, whose default data rate is 9600 baud, long enough to reconfigure the Ampro for higher data speeds. Use the Kaypro CONFIG program, option 'M,' to set the Kaypro for a 9600 baud default data transmission rate. Press the Kaypro RESET button to assert the changes by writing them onto the system tracks of your disk.

Run KTERM.

Insert the Ampro SYSTEM DISK into drive 'A' of the Ampro. With KTERM running, turn on the Ampro. The Ampro log-on message may not be displayed properly, but it should be readable.

Enter CONFIG

Use the AMPRO CONFIG program, option '6,' to configure Serial Port 'A' as follows:

1. 8 data bits
2. 1 stop bit
3. even parity
4. 9600 baud
5. with hand shaking

When you have assured that the system is functioning properly you may increase the baud rate to 19200, though 9600 is fast enough for most operations.

Changing Baud Rates

To increase the baud rate, or lower it, enter the Ampro CONFIG program and set the desired baud rate. Install the changes ON DISK only. Do not install the changes in memory or communications will be lost. Exit KTERM and use the KAYPRO CONFIG program to set the desired baud rate. Press RESET on the Kaypro, and enter KTERM. RESET the Ampro and you should be on-line again.

The Terminal Program

KTERM is a very simple program. It has no features whatsoever. I have often been using the Ampro and found myself changing disks in the Kaypro, instead of the Ampro. This is the kind of terminal function I prefer. KTERM may be easily expanded to suit your own tastes.

We begin our discussion of the Pascal program with the first functional line of code. The entry:

```
{C-}
```

is a Turbo Pascal specific compiler option which inhibits the interpretation of control characters by the program. This feature is a must. Without it the host computer would attempt to act upon any control characters input at the console. We want the Ampro to to be sent control

Figure 3

```
{
                         H E R M I T   S O F T W A R E

                           Pascal Program Source File

 Program: KTERM

 Version: 1.0a

 Class  : Public Domain Utility

 Author : C. Thomas Hilton


 Date:    July 12th 1985

 Hardware Requirements:
                          Kaypro 4-84
                          Ampro Series 100, Model 122

 Program Comments:
                      The maximum speed that known Kaypro communications
                      programs have been able to interface with the Ampro 122 has
                      been 2400 baud.

                      This dumb terminal program corrects the defects of these
                      programs by allowing the AMPRO 122 to communicate with a
                      Kaypro 4-84 at 19200 baud, the maximum data transmission
                      rate allowed by the Kaypro CONFIG program.

                      Use the Kaypro CONFIG program to set the baud rate in the
                      KTERM working disk as the default data rate.

                      This program is in routine use, and shows no defects. See
                      the main text of this Application Note for the AMPRO 122
                      configuration set by the AMPRO CONFIG program.

                      Use the ^@, (null control code), to exit the program.
                      This is the only user command available, or needed for
                      particular application.
}


{$C-}               { Turn off control character interpretation by program  }

type
    workstring=string[80]; { define a utility variable                      }

const                      { define program constants                       }
      txrdy=4;             { transmitter buffer empty mask value            }
      rxrdy=1;             { receiver buffer full mask value                }
      dataport=4;          { SIO-1 'A' data I/O port assignment             }
      status=6;            { SIO-1 'A' control/status port assignment       }
      DtrWait=$68;         { 'Hey I'm Busy' DTR flag value                  }
      DtrRdy=$E8;          { 'Ready When You Are' DTR flag value            }

var                        { set system variables                           }
   finis:boolean;
   ch:char;

{------------------------- Functions & Procedures ---------------------------}

FUNCTION ReadStat:boolean;  { turns TRUE if character received              }
begin
     ReadStat:=((port[status] and rxrdy) <>0)
end; {of ReadStat}
```

codes from the terminal, and to act upon them. Essentially we want the terminal program to make the Ampro appear as if it were the only system in use.

Pascal TYPE and VARIABLE declarations, as shown in Figure 3, are generic, and need not be modified for systems other than the Kaypro.

The program CONSTANT declarations, however, may be redefined for non-kaypro users, or owners of differing models.

Function READSTAT does little more than determine the the status of the receiver register of the SIO. Function READCHAR polls READSTAT until the SIO indicates that a character has been received. It will then input the character. A logical AND is performed to mask-off any parity bit. In some systems, and terminals, the parity bit may trigger graphics characters.

Procedure INIT initializes the SIO (which the Kaypro ignores, save for setting the keyboard channel) for data communications. This set of code sets the data channel for our default data format of an 8 bit data word length, 1 stop bit, and even parity. Consult a ZILOG manual for the actual meanings of the codes shown. The operation of the SIO is beyond the scope of this small program.

Procedure WRITECHAR simply sends a keyboard character to the Ampro upon demand. Procedure CENTER centers the KTERM log-on message on the host 80 column terminal.

The main program organizes all of the above functions and procedures to perform the logical exchange of data between the two systems. This includes the setting and resetting of the hand shaking signals. A test is made for an escape character, control @, (^@). If the escape character is detected, as an input character from the keyboard, the program aborts to the operating system.

KTERM is one of those simple programs that hardly rate a comment, but are found in constant use. I use it exclusively with the Ampro, and cannot function without it. Other programs have features that do not allow me to interface with the Ampro as if it were actually the host system.

Configuring The System

If you purchase a Series 100 system, you will receive T/MAKER® , ZCPR3® , the "FRIENDLY OPERATING ENVIRONMENT® ," and standard CP/M® 2.2 as a "software bundle." Those who elect for other configurations will have these bundled programs as options. The T/MAKER package is a powerful set of applications programs, in a single system. WordStar® is not included in the bundle as T/MAKER has its own word processing system.

The Friendly system is configured with WordStar, (WS), cursor commands. T/Maker, uses a more standard approach. The CP/M version of TM is set to use cursor control codes which are compatible with the ADM 3-A terminal, and Kaypro computers. Three different sets of cursor commands is at best, frustrating, especially if your system has user definable keys, as does the Kaypro.

There are two options, either change the TM editing codes, which is allowed by the system, to those of the operating environment, or change the control codes of the operating environment. As a WS user this was a hard decision for me to make, especially since I use two dif-

```
FUNCTION ReadChar:char;  { read a character from SIO when char arrives   }
begin
     repeat until(ReadStat);
       ReadChar:=char(port[dataport] and $7f)
end; {of ReadChar}

PROCEDURE Init;             { set SIO-1 'A' for program use               }
begin
port[status]:=$18;          { reset Channel 'A'                           }
port[status]:=1;            { call SIO interrupt register 1               }
port[status]:=0;            { disable interrupt functions                 }
port[status]:=3;            { call SIO receiver parameters register 3     }
port[status]:=$0E1;         { set for 8 data bits, enable receiver        }
port[status]:=4;            { call SIO protocol register 4                }
port[status]:=$47;          { one stop bit, even parity, X16 clock        }
port[status]:=5;            { call transmit parameter register 5          }
port[status]:=$0e8;         { set for 8 data bits, enable transmitter     }
end; {of Init }             { and set Data Terminal Ready, (DTR)          }

PROCEDURE WriteChar(kc:char);  { send a character to Ampro                }
{ because no human could ever type so fast that the Ampro would miss a    }
{ character no output handshaking is supported in this program. Other     }
{ uses might require a treatment similar to that shown below for output   }
{ handshaking.                                                            }
begin
     repeat until(port[status] and txrdy) <>0; { is buffer empty?         }
     port[dataport] :=ord(kc);    { when buffer empty send character out  }
end; {of WriteChar}


PROCEDURE Center(S: WorkString);  { general utility for log-on, log-off   }
var R: integer;
begin
     for R:=1 to (80-Length(S)) div 2 do Write(' ');
     writeln(S);
end; { of center }

{--------------------------- Begin Main Program -----------------------}

begin
Init;                       { Initialize SIO-1 'A' for our use            }
clrscr;                     { begin log-on sequence for Kaypro screen only }
center('Hermit Software''s');
writeln;
center('K-TERM Version 1.0a');
writeln;
center('A Kaypro 4-84 / Ampro Series 100 Dumb Terminal Program');
center('Released Into The Public Domain July 1985');
writeln;
center('Press ^@ to Exit');
writeln; writeln; writeln;
writeln('System Ready'); writeln;  { End of log-on sequence           }
finis:=false;               { set initial boolean escape value            }
repeat                      { begin main loop                             }
      if (keypressed) then             { has a key been pressed?          }
       begin
         read(kbd,ch);                 { get it if so                     }
         if (ch=^@) then finis :=true  { if exit code then exit without   }
      else                             { sending to Ampro                 }
         WriteChar(ch);           { if valid character then send to Ampro }
      end;

      if (ReadStat) then { check to see if a character has been received }
        begin                               { set DTR BUSY flag line }
        port[status]:=5; port[status]:=DtrWait;
        write(ReadChar);                    { print the character    }
        port[status]:=5; port[6]:=DtrRdy;     { reset DTR for next character }
        end;
until (finis);                    { when 'finis' turns TRUE comes here }
writeln; writeln; writeln;
writeln('Exiting KTERM....... Have A Nice Day! ');  { log-off message    }
end.                                                { termination        }
```

ferent computers on a daily basis. I chose to alter the operating environment codes. The size, and number of commands in the TM system made the choice a little easier.

ZCPR3 has a portion of the system BIOS set aside for terminal control code definitions. We will discuss these stored terminal codes, and how to use them, in a later article. If you are using a popular terminal, or computer as a terminal, installation is a simple matter. At the command prompt enter:

A0>TCSELECT MYTERM<RET>

a menu of supported terminals will be displayed to select from. The "TC" in the command name stands for "terminal cap." This 'cap" is the set of terminal control codes stored in the system BIOS. The support for these terminals, and systems, are very generic in nature. I found that I received better performance from the system by installing ZCPR3 manually, using the "TC-MAKE.COM" utility.

One of the very nice things about the Ampro systems is the fact that they may be adapted easily to most any kind of terminal, or display system. In my work for the visually impaired we have used a serial keyboard as the input device, and a voice sythesizer as the output device. While we made extensive BIOS modifications, the actual hardware interface took only moments to accomplish. Due to the wide range of devices that may be used as a terminal, specific installation discussion would be meaningless to most readers. Everyone would feel that their system was not covered. The Ampro documentation is well done, and guides the first time user through the installation steps required for non- standard terminal systems.

As TM is the only full featured text editor, (ED.COM is also provided), I do have a comment to make for those who will be using TM for programming. TM produces sterile ASCII code. Unlike some editors, who use flipped bits and other trash for system use, TM does not put its flags in the general text. It does have a quirk, however.

TM has a maximum line length of 300 characters in a CP/M system, 400 for 16 bit systems. Yes, 300 characters! The screen can scroll left and right, as well as up and down! An off-screen "first line" stores the user's tab settings. A wide number of assemblers and languages get very upset when the first thing they see in the program file is a line of 300 characters. This "tab line" is composed of nothing but spaces and tab characters.

When installing TM answer NO to the question "Should TABs be stored with the file?" By placing the cursor on a line of text and entering <ESC> S <TAB> TM will record tab characters, for the current editing session from this "model line." As TM is the bundled editor with the Series 100 systems, I will provide a tab line for all programs. The tab line problem is offset by the ability to produce extremely high quality printed listings, and documents with T/Maker.

For those who wish to leave the tab line in their text files, there is an option. When saving programs enter:

NOTABS SAVE

at the "WHAT NEXT?" prompt. Remember that, like ZCPR3, TM allows command lines of up to 255 characters.

My only complaint with TM, other than having to learn new editing keystrokes, is the inability to execute a .COM file from the command line. Learning to use TM as a programming tool will be included in future articles as well.

While I didn't get into very much "good stuff" this time, it is hoped that this information will help you to get your system functional. Between the time you read this, and the next issue of TCJ is published, make friends with your system. In this way you will be ready for the projects to come. There is a great deal planned for the future. A new CCP, (console command processor), for a larger TPA, a voice output enhanced BIOS, and a close look at the schizoid 'E' drive which allows reading, writing and formatting of other system's disks, to name but a few. I didn't know where to start a column. So, I began at the beginning, and there is a great deal I would like to cover that I didn't even get close to this time. Of course, if you have a project you are interested in seeing covered here, let me know. This is your column, as an Ampro user. ■

* * *

AMPRO User Support

Tom and I are both impressed with the AMPRO Little Board and their Bookshelf 100 computer series, and we making a major user support effort for these systems.

In addition to this series of articles (plus the NEW-DOS series which is aimed at the AMPRO), Tom is preparing an AMPRO user disk library of 25 to 30 DSDD disks. These disks will be distributed thru the TCJ office, and we will provide an on-line RBBS when we can obtain an additional phone line. New disks will be added to the library as the material is available.

I am especially interested in SCSI related software, and plan on establishing separate disk volumns for this topic. Any SCSI material (whether for the AMPRO or not) will be greatly appreciated.

Editor

(Continued from page 1)

The SCSI Interface

Interfacing to microcomputers has been limited by the lack of a satisfactory high speed port. Most Centronics® ports are implemented as only a printer driver with eight lines for data output and a few lines of input for printer status. The RS-232 configuration dates back to the slow Teletype® days and is slow with a lot of non-standard implementations, and the IEEE-488 doesn't really fill the needs of the micro market. The SCSI interface (originally SASI) was developed by Shugart as a high speed parallel interface for hard disks, and is being adopted by many manufacturers as a general interface.

Almost every issue of the technical design magazines tell about new designs incorporating the SCSI interface, and you'll be dealing with it in the very near future. Some of the reported implementations are the expanded Macintosh from Apple, the next micro from IBM, and the Apple LaserWriter II (which has 8 Mbytes of RAM and ROM). The Ampro little boards (both the Z-80 and the 80186 versions) provide SCSIplus, which is one thing which makes these such great development systems. Starting with this issue we'll have a regular section on SCSI with technical information, applications, and product news.

Communications, User Disks, Odds & Ends

We finally have a 300/1200 baud modem running on MDM740 (the communications program came installed an the Ampro), and can receive and send files over the phone. We would like to establish a bulletin board, but the phone company can not provide a second line and we can't tie up the existing line. For now we prefer to receive articles on disk, but arrangements can be made to transfer by modem if you call by voice in the evening. Short messages and subscriptions by charge card can be left for Art Carlson on the Kalispell BBS (406) 257-6117 during the hours of 6pm to 8am Monday through thursday, 6pm Friday to 11am Saturday, and all day on Sunday Mountain Time. These facilities are provided by The Computer Place and the storage is limited, so please don't abuse the privilage. Would it help if we had a BBS on line during the limited hours of 10pm to 6am mountain time until we can obtain a second line? Leave a message and let me know.

Tom Hilton will be the librarian for CP/M user's disks in the Ampro 5¼ inch DSDD format, and the orders will be filled from the TCJ office. We'll be starting with about 25 disks of general CP/M files, and will add more as they develop. These files can be provided on some additional formats on request. Watch for more details in the next issue.

I've been plagued with erratic read and write errors on one set of Shugart 8008" single sided drives. They would almost always read and write OK on the outer tracks, but would frequently give problems on the tracks above about number 60. I checked the voltages, changed the head load pads, cleaned the heads, and tried different disks, but the problem would alway reappear. I finally tried the drives in another cabinet with a linear power supply, and they worked perfectly so I figured that there just had to be something wrong with the power supply.

This is a very strange set up with a low height switching type power supply mounted directly beneath the drives, and I have to set the drives outside the case in order to get at the supply. I put my old Heath scope on the output to check for 60 cycle ripple (again), and it looked clean—but then I noticed that the line looked a little fuzzy. When I cranked up the intensity I could see that there was a lot of high frequency noise riding on the DC. I put 20,000 mfd on each of the outputs and the drives performed flawlessly while the drives were out of the case. After reassembling everything I still get occasional problems from RFI picked up by the drives or the wiring. I'll have to pull the supply and put it in a separate well shielded enclosure with filtered output or just replace it with a linear supply.

It's little things like this that drive you nuts! Share your experiences and problems with others, perhaps the Kalispell BBS will make it easier to communicate.

The Magazine Marketplace

Now that Creative Computing and Popular Computing have ceased publishing, we need to think about what we want in a magazine. Who should support a magazine, the readers or the advertisers? Who should a magazine support, its readers or the advertisers? Are readers only necessary as numbers to justify high advertising rates?

I have just received a mailing from Ziff-Davis Professional List Services offering to rent the mailing list for Creative Computing (which Ziff-Davis shut down last month), and they say that Creative Computing has (or rather had) 233,772 subscribers. If my memory serves me right, one of the other magazines had over 500,000 subscribers when they shut down. These magazines all gave the same reason for folding—falling advertising income. It didn't matter that they had several hundred thousand subscribers—advertising dollars were what kept them open.

You would think that the subscriptions from a half-million readers should be enough to support a magazine without ANY advertisers, but producing a slick magazine with lots of four color illustrations is very expensive. Magazine rack sales are another drain. I have seen audited reports which stated that 60% of the magazines sent to the magazine racks were unsold and sent to the shredders. The specifics in one case were 90+ thousand sent out and 60+ thousand shredded for that month. Someone has to pay for the 60,000 magazines turned into scrap paper—and it isn't the subscriber.

Everthing seems to cost too much, and subscriptions to the better magazines are not cheap, but in many cases the price of a subscription does not even cover the cost of printing and mailing the issues—and I understand that when subscriptions are placed through the national subscription agencies all of the money may go to the agency without one penny to the publisher. We subscribers have come to expect large, slick, four-color magazines at low cost, or even below cost, subscription fees. Then we complain about the large percentage of advertising (up to 75% of the space) and the weak editorial content! As Frank said in issue 402 of Echelon's Z-News "Many computer magazine editors have little or no investment in or emotion for our industry or its hardware...You seldom notice anything written that offends anyone; you notice they remind of melba toast...Remember, their revenue comes mainly from advertisers, not from you their readers." It is obvious that if profit and continued existence depend on the advertisers, then the readers will be treated merely as numbers to justify high advertising rates—and the subscription rates will have to very low to attract large numbers of readers who don't gain much from the magazine. I know that I've subscribed to magazines which didn't have much useful content, but they were so thick and cheap that it seemed a bargain.

We publish TCJ for the readers, and limit the advertising to products that we feel our readers should know about. The limited funds restrict the amount of promotion we can afford, and we can't pay the authors as much as we would like, but we are free to publish what our readers want without worrying about offending any advertisers. If you like an article, write the author and let him know, because they aren't doing it to get rich but rather because they want the information published! ■

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II +, IIc, IIe, Macintosch, DOS 3.3, ProDOS; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. MBASIC; Microsoft. Wordstar; MicroPro International Corp. IBM-PC, XT, and AT; IBM Corporation. Z-80, Zilog. MT-BASIC, Softaid, Inc. Turbo Pascal, Borland International.

Where these terms (and others) are used in The Computer Journal they are acknowledged to be the property of the respective companies even if not specifically mentioned in each occurence.

New Products

Bootable Z-System for H89/90

Analytical Products offers a Bootable Z-System disk for Heath and Zenith models 89 and 90 machines with no installation required. Just place the disk in the drive and press reset and the full Z-System, completely replacing CP/M, is up and running. The price for the bootable disk is $98.00 plus $3.00 shipping and handling. Source code for ZCPR3 and its utilities, and utilities of ZRDOS are also available at additional cost. The ready-to-run H89/90 Z-System can be ordered from Analytical Products, 20663 Avenue 352, Woodlake, CA 93286, or call Mr. Peter Shkabara at (209) 564-3687 for literature and more information on Heath and Zenith products.

Amiga-Lint C Diagnostic Facility

Gimple Software announced their Amiga-Lint, a diagnostic facility for the C programming language, running on the Commodore Amiga. They state that Amiga-Lint will analyze C programs and report on bugs, glitches and inconsistencies, in effect, providing a strong typing facility for C. Amiga-Lint looks across multiple modules and so enjoys a perspective that a compiler doesn't have. It aids considerably in developing reliable programs and in porting programs from other machines and operating systems. Amiga-Lint resembles the Lint that runs on the UNIX O.S. but has more feature and is better tuned to the 68000 environment.

Among the many errors reported on by Amiga-Lint are: type inconsistencies across modules, parameter-argument mismatches, library usage irregularities, uninitialized variables, value-return inconsistencies, variables declared but not used, suspicious use of operators and unreachable code. Amiga-Lint has many features, including full K&R support, one-pass very fast operation, no fixed-size tables to overflow, configurable to arbitrary architectures and special Lint-style comments to suppress errors. Amiga-Lint is delivered with user-modifiable standard library descriptions for several C compilers.

Amiga-Lint runs under Amiga's CLI interface. It will use all the memory available. Amiga-Lint is available for the special introductory price of $98.00, including shipping within the continental U.S. directly from Gimple Software, 3207 Hogarth Lane, Collegeville, PA 19426 (215) 584-4261.

CP/M-80 Emulation for MS-DOS

The ICU Group has announced CP/EM-CP/M 80 Emulation which they claim gives IBM PC/XT/AT and compatible computers the ability to run thousands of CP/M 80 programs without the expense of additional coprocessor boards.

CP/EM efficiently emulates the CP/M 8080 and Z80 environments on an MS-DOS based personal computer. CP/EM allows MS-DOS redirection of input and output devices used to alter device assignment allowing CP/M access to all standard MS-DOS devices and any installed device drivers. CP/EM uses the standard MS-DOS file system allowing data files to be shared between CP/M and MS-DOS applications. The Command Interrpreter provides all of the standard commands provided by the CP/M console command processor.

CP/EM version 1.2 provides terminal emulations for the Kaypro 10, ADM 3A/5 and Televideo 950. Serial communications programs are included with CP/EM to aid in the transfer of programs and data between the CP/M and MS-DOS computers. CP/EM runs on any MS-DOS, version 2.0 or later, based personal computer with at least 32K of memory available for application programs.

CP/EM can be ordered from The ICU Group, PO Box 10118, Rochester, NY 14610 (716) 425-2519

DOS 3.3 Compatibility with Apple's 800K UniDisk

Apple Computer's new 800K UniDOS 3.5 drive gives five times the storage capacity of floppy disks, but no program support for Apple's DOS 3.3 operating system. MicroSPARC's new UniDOS 3.3 operating system fills this gap for programs and data files that exist under DOS 3.3 by providing big 800K disk capacity and complete Applesoft compatibility with Apple's DOS 3.3.

Key UniDOS features are: (1) Two 400K volumes per disk; (2) Supports up to two UniDisk 3.5 drives addressable as drives 1-4; (3) Allows intermixing 5.25 inch and 3.5 inch drives; (4) Allows up to 217 catalog names per disk; (5) Uses only 1K of user memory (in addition to normal DOS 3.3 memory space).

UniDOS 3.3 comes with a user manual and Technical Data Sheet showing the modified DOS 3.3 address for systems programming. It runs on the Apple II Plus, Apple IIe, and Apple IIc, and software developer licenses are available.

UniDOS 3.3 is available for $49.95 postpaid from MicroSparc Inc., 45 Winthrop Street, Concord, MA 01742 (617) 371-1660

MasterFORTH Supports 8087

MicroMotion MasterFORTH 1.2 for the IBM PC family now supports the 8087/80287 math co-processor. This 8087 extension includes a complete macro assembler with local labels supporting all precisions, opcodes, and synchronization. The floating-point package includes a full complement of transcendental and high-level functions, as well as formatted input and output routines. Both the assembler and the floating-point package are provided as source files and as relocatable overlays. A software version of this package, which completely matches the hardware version, is also available. Applications can test for the presence

(Continued on page 54)

Back Issues Available:

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

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

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

Volume 1, Number 4 (Issue #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 (Issue #5):
- *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 (Issue #6):
- *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 (Issue #7):
- *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 (Issue #8):
- *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 (Issue #9):
- *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 (Issue #10):
- *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 (Issue #11):
- *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 (Issue #12):
- *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 (Issue #13):
- *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*

Issue Number 14:
- *Hardware Tricks*
- *Controlling the Hayes Micromodem II From Assembly Language*
- *S-100 8 to 16 Bit RAM Conversion*
- *Time-Frequency Domain Analysis*
- *BASE: Part Two*
- *Interfacing Tips and Troubles: Interfacing the Sinclair Computers, Part Two*

Issue Number 15:
- *Interfacing the 6522 to the Apple][and][e*
- *Interfacing Tips and Troubles: Building a Poor-Man's Logic Analyzer*
- *Controlling the Hayes Micromodem II From Assembly Language, Part Two*
- *The State of the Industry*
- *Lowering Power Consumption in 8" Floppy Disk Drives*
- *BASE: Part Three*

Issue Number 16:
- *Debugging 8087 Code*
- *Using the Apple Game Port*
- *BASE: Part Four*
- *Using the S-100 Bus and the 68008 CPU*
- *Interfacing Tips and Troubles: Build a "Jellybean" Logic-to-RS232 Converter*

Issue Number 17:
- *Poor Man's Distributed Processing*
- *Base: Part Five*
- *FAX-64:Facsimile Pictures on a Micro*
- *The Computer Corner*
- *Interfacing Tips and Troubles: Memory Mapped I/O on the ZX81*

Issue Number 18:
- *Interfacing the Apple II: Parallel interface for the game port.*
- *The Hacker's MAC: A letter from Lee Felsenstein*
- *S-100 Graphics Screen Dump*
- *The LS-100 Disk Simulator Kit: A product review.*
- *BASE: Part Six*
- *Interfacing Tips & Troubles: Communicating with Telephone Tone Control*
- *The Computer Corner*

Issue Number 19:
- Using The Extensibility of FORTH
- Extended CBIOS
- A $500 Superbrain Computer
- Base: Part Seven
- Interfacing Tips & Troubles: Part Two Communicating with Telephone Tone Control
- *Multitasking and Windows with CP/M:* A review of MTBASIC
- The Computer Corner

Issue Number 20:
- Build the Circuit Designer 1 MPB: Designing a 8035 SBC
- Using Apple II Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K
- The Computer Corner

Issue Number 21:
- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Build the Circuit Designer 1 MPB: Part 2 - Programming the 8035 SBC
- The Computer Corner

Ordering Information: Back issues are $3.25 in the U.S. and Canada. Send payment with your complete name and address to The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912. Allow 3 to 4 weeks for delivery.

Classified

ORDER FORM

Enter my subscription to The Computer Journal for the period checked. Payment in U.S. funds is enclosed.
☐ one year (6 issues) $14 in U.S. ☐ two years (12 issues) $24 in U.S. ☐ new subscription ☐ renewal

Back Issues #'s________________________@$3.25 ea. ________

☐ Check enclosed ☐ VISA ☐ MasterCard Card # ________________________

Expiration date ________ Signature ________________________

Name ________________________

Address ________________________

City ________________________ State ________ ZIP ________

The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

Advertiser's Index

New Products

(Continued from page 51)

of the 8087 co-processor and select the appropriate overlay.

MicroMotion MasterFORTH 1.2 runs on all members of the IBM family and includes a full file interface to MS DOS 2.1-3.1. It is also available for the Macintosch, the Apple family, the Commodore 64 and CP/M. Software can be written on one system and run on all the others. MasterForth retails for $125, and several additional extensions are available. MicroMotion, 8726 Sepulveda Fl. #A171, Los Angeles, CA 90045 (213) 821-4340

Computer Voice For The Blind

Artic Technologies announces the release of SynPhonix 200-VIP, a low cost voice synthesizer for Blind and Visually Impaired users of the IBM-PC/XT/AT personal computers. This product works in conjuction with other programs to speak the video screen and text files. Blind individuals can use SnyPhonix in applications such as word processing, spread sheet calculators, programming, and other marketable computer skills. As a personal tool SynPhonix can be used to access computer bulletin boards, data bases, write letters, etc.

A variety of voices and features can be controlled by the user from the computer keyboard. Speech can be set to an extremely fast rate to rapidly scan information and slowed down to examine details. Volume, pitch, tone, pausing and number pronuncation can also be controlled. A voice type can be selected according to the users preference, and the time can also be spoken on demand.

Complete installation and start-up instructions are provided on an audio tape so that a blind user can perform the installation without a sighted companion to read the manual.

SynPhonix 200-VIP retails for $295 and is available from Artic Technologies, 2234 Star Court, Auburn Heights, MI 48057 (313) 852-8344

Computer Corner

32 Bit System Bus

On a different note, I would like to propose a new standard for using 32 bit systems. One of the topics of considerable discussion is how to handle the new 32 bit wide data structures. For the hobbyist it is a real problem of cost versus speed. Most new systems are being designed as standalone units which do not use a bus system. This idea doesn't make use of a hobbyist's already existing supply of computer stuff. In my case, I have quite a few S-100 boards and would like a system that used them. Currently, S-100 will handle 16 bits by multiplexing the data bus after flagging a sixteen bit move. S-100 also has some known speed limits, abour 10 to 12 MHz at present.

I find the speed and use of the sixteen bit flag an acceptable solution, but the bus doesn't have room for more data lines. My proposal is to use two S-100 buses, side by side. The 32 bit only card would be a double width S-100 card. This would require a standard spacing between the bus systems but for 32 bits that's OK. This newer use of the extra S-100 bus would not have to be the standard pinout. However, if the same pinout was used on both buses it would make it possible to use the same memory cards in each side for 16 or 32 bit wide data. The use of different S-100 pinouts on the other hand, allows for some special functions and signals not now supported by the standard.

Another option to the use of two buses side by side would be splitting a 22 slotter in half, by cutting all the traces (except power and ground) at the half way point. A special 32 bit master card would be two separate cards connected together (by cable or socket connection) that spanned the cut traces. In either case, the signals and pinouts could either be the same or different, depending on the standard developed. I know that Viasyn was looking for a way to handle preregulated power supply signals, and this could also be handled with two different buses.

This concept isn't new, as several of the big boys already have systems in which a standard number of pins is used, but the number of sockets per card varies depending on how

```
9000   00 00     NOP NOP       ;NO OPS TO HELP SET THINGS UP RIGHT
9002   16 1A     MVI D,1AH     ;LOAD NUMBER OF SECTORS IN D REGISTER
9004   21 00 01  LXI HL,0100H  ;ADDRESS TO START WRITING TRA K DATA TO
9007   3E 2D     MVI A,7D      ;LOAD A WITH DRIVE B NUMBER AND DENSITY
9009   D3 63     OUT 63        ;OUTPUT A TO DISK CHIP, SELECT DRIVE
990B   F3        DI            ;TAKE CONTROL AND STOP ANY INTERRUPTS
900C   06 00     MVI B,00      ;SET UP LOOP COUNT VALUES
900E   0E 67     MVI C,67      ;DATA PORT ADDRESS IN C FOR INDIR
9010   3E E4     MVI A,34H     ;DISK CHIPS FULL TRACK COMMAND
9012   D3 64     OUT 64        ;TELL DISK CHIP READ A FULL TRACK NOW
9014   18 00     JMP R,0       ;WAIT A FEW CLOCK CYCLES
9016   18 00     JMP R,0       ;WAIT SOME MORE FOR CHIP TO GET DATA
9018   ED B2     INIR          ;Z80 GET DATA FROM I/O AND WRITE TO HL
901A   15        DCR D         ;COUNT DOWN ON LOOP COUNTER
901B   C2 0C 90  JNZ 900C      ;STILL LOOPING GO BACK AND GET MORE
901E   FF FF FF  RST 7         ;GET BACK TO DDT PROMPT.

NOTE: USE G9000 TO RUN PROGRAM, MUST BE ON DESIRED TRACK FIRST
      9008 WILL CHANGE FOR DIFFERENT DRIVES AND DENSITY. CHECK
      BOOK FIRST AND WATCH OUT FOR INVERTED LOGIC.


TYPICAL ID MARK OF FORMATTED DISK

FF FF FF 00 00 00 00 00 00 FE 02 00 14 00 C3 2D FF FF FF FF FF
   :___________________:__:__:__:__:_____:
                                     CRC
                                 SECTOR LENGTH
                              SECTOR NUMBER
                           SIDE NUMBER
                        TRACK NUMBER
              THIS TELLS CONTROLLER ID FOLLOWS
```

many hardware extensions are needed to handle the project. What is missing from the other ways, which the S-100 would provide, is the cost of this solution. This solution makes use of the already existing supply of 8 and 16 bit boards, disk controllers and I/O cards. Most 32 or 16 bit systems still write/read the disk in 8 bits and do their I/O in 8 bits, which leaves only memory, graphics, and CPU functions needing a larger data path.

A project then for the future will be a 68020, doing 16 bit data transfers. The CPU will be spread over two cards for using two halves of a 22 slot bus. Regular 8 bit memory and I/O cards will be in either half, only the data path will be split, lower 8 bits in front, upper in back. The need for such a system would be fast 16 bit wide video mapping, and this video would make up the other half of the CPU card. The 24 bit S-100 address would be the same for front and back, allowing generic cards to be used throughout.

The operating system for the above would be FORTH, and my project on putting FORTH in ROM is still moving along. A few good books for understanding FORTH are:

Threaded Interpretive Languages by R.G. Loelinger, BYTE Books. This book discusses how a FORTH type system is designed and implemented. The sample is a Z-80 FORTH. A good book for those who want to know more, somewhat entertaining, also.

Inside F83 by C.H. Ting Phd, OFFETE Enterprises, Inc. For those using FORTH 83 this book is a must. I was stumbling along until I got it. The introduction and details will answer almost all possible questions.

Don't forget the traditional sources for help on FORTH:

FORTH INTEREST GROUP, PO Box 8231, San Jose, CA 95155. They publish FORTH DIMENSIONS which is included with the $15 membership fee.

MOUNTAIN VIEW PRESS, INC., PO Box 4656, Mountain View, CA 94040. These people were the originl publisher of FORTH Dimensions, but now are strictly a resource center for FORTH publications.

To see and read about some different applications try one of the special group publications such as FORML Proceedings and The Rochester Proceedings, or The Journal of Forth Applications and Research. I got a couple of these books and found some interesting stuff. The degree of help is limited however as these were papers presented at conferences. ■

THE COMPUTER CORNER

A Column by Bill Kibler

Several things have happened recently to cause me to review some disk format fundamentals. It seems that most people have some idea about how disks are formatted and used, but some misconceptions abound about a few simple but important concepts. What must be kept in mind when dealing with dissimilar formats are the software and hardware parameters. The hardware part is how your computer's disk controller formats, writes, and reads disks. The software part is the tricking of hardware to read or write unusual formats.

Let's review things by looking at standard eight inch disks, using the single density format called IBM 3740. This single density arrangement was established way back in the beginning and is the only true standard of any disk format. The disk has 26 sectors of 128 bytes of data and plenty of overhead information. It is this overhead information that can cause some trouble with varied formats. The disk controller will check the "ID marks" (a part of the overhead) to find out if it is on the right track, and then find the correct sector. Other information in the ID is the sector size (00=80h, 01=100h bytes) and a CRC flag. A good way to see what a proper disk format is, is to do a full track read. I have supplied a simple listing that will allow the SDSystems Versa Floppy II to read a full track (read the comments and change for other systems). My procedure is to use SID (or DDT) and clear memory to A000h and assemble the listing at 9000h. This allows you to exit SID, change tracks or whatever, and return to SID to reread a new track.

Now that you have seen a full track, what do you look for if this is a disk you have been unable to read? I look first for the order of sectors, are they in sequence or skewed? Skewing is done to get more sectors read during each rotation of the disk. Disk read time is not fast enough to read one sector after the other. The delay needed to change pointers and set registers for the next read, is usually enough that a skew of six (skipping five sectors and reading the sixth) works fine. This is part of the standard and will be handled in most BIOSs. Some systems however do it during formatting and will need the BIOS's skew turned off.

This is actually one of the points of this whole discussion, as most people have trouble understanding the differences. Looking at a directory track might help. A non-skewed system (no skewing of any kind) would have the sectors numbered in order and the directory sectors would correspond with the sector numbers. A standard disk would have the sector numbers in order but the second directory sector would not be found in sector number two but in sector number seven. A hard skewed track would find sector number two in the seventh sector from the beginning location. The standard system will use a skew table and the hard skew will not. What most people have problems with is how the system performs this operation. The operating system will ask for a logical sector number. The BIOS will change (or not) the sector number into the skewed value (2 becomes a 7 in standard systems). This value is then stuffed into the disk controller sector register and used to find the actual sector. You must remember that it is you that tells the controller which sector to get and it reads the ID mark to find that sector. A common misunderstanding is the use of the index hole. Some early systems counted sectors starting at the index hole and the recorded sector number was unused. Most new disk controller chips however use the formatted sector ID mark to find the proper sector.

It is my understanding that on most new systems you will find the index hole used only for full track reads and writes. Some chips like the NEC 765 also provide their own format options. The WD179X series uses a full track write for formatting (you supply the data string). The most common use for the Index timing mark is to check disk size (five and eight inch timing is different) and for disk status (ready) condition.

We can conclude this discussion by reviewing how to copy or use a different formatted disk. First is to do a full track read through a hand assembled program or your system's monitor. This will give you the sector size, skewing information, and density (single or double). For double sided drives you will need to check the track order as some systems go from side to side (all odd number tracks on one side of the disk) while others fill one side before going to the other side (the ID mark tells which side, hopefully). The next operation is to use the listing of your BIOS (the PRN file) and find the disk's DPBA table (can be found by using DDT if you know what to look for). This table has the address for the DPB (parameter blocks which tell the system block size, number, etc.) and the XLT (translation table for skewing). For no skew systems the XLT can have all zeros, for others it will point to a table of skew values. The translation program steps through the table based on the logical skew number. The value in the table is then sent to the disk controller. Patching the table with DDT will work for different skews.

Many little things must be kept in mind however. Your BIOS must support double density or larger blocks, to be able to read double density disks. If the number of sectors or blocks is different than before, new values will be needed in the DPB. Systems can have a different number of system tracks and the offset in the DPB must match. Side differences will need to be checked as system programmers do it differently.