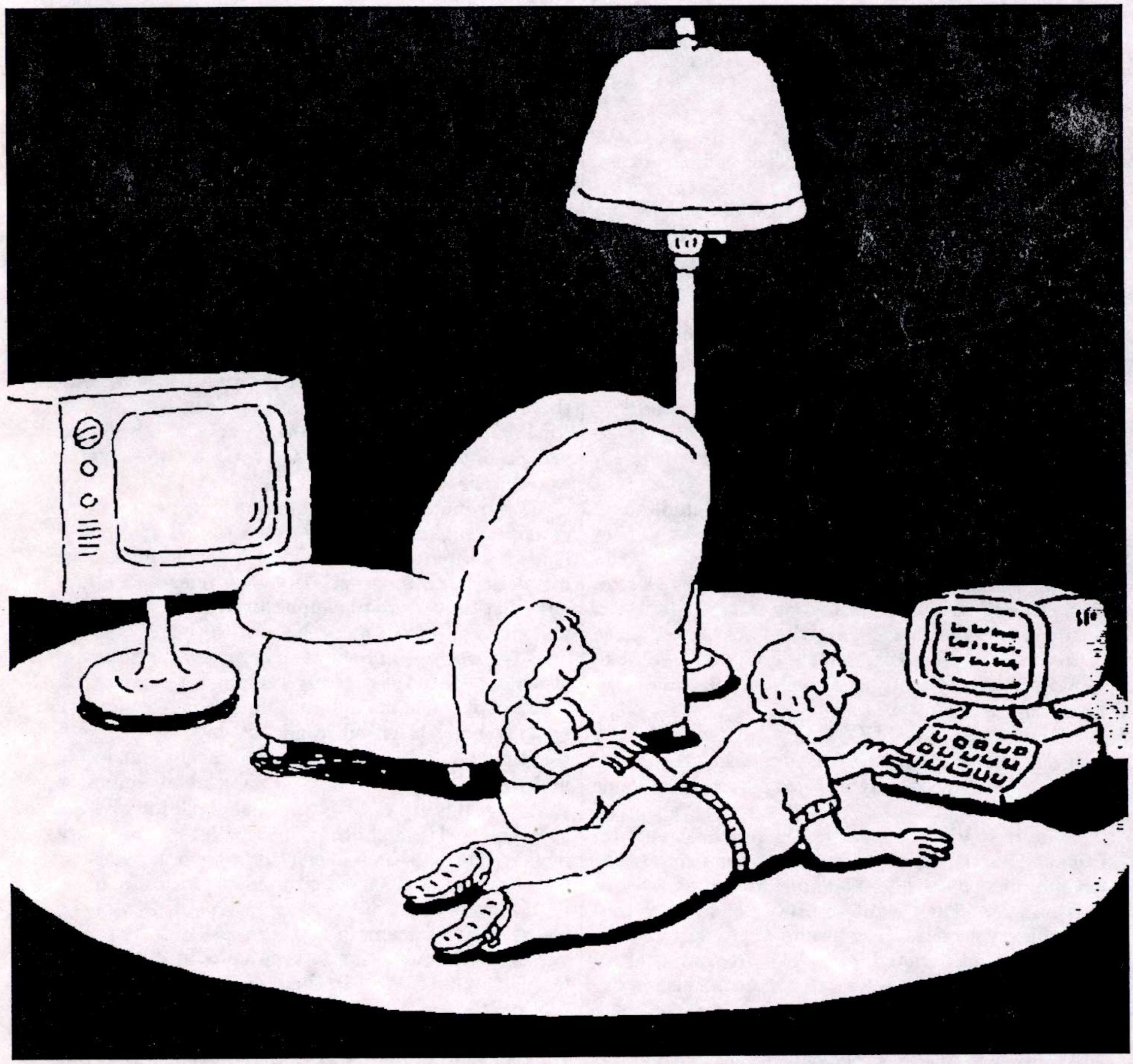
TRSTimes

Volume 7. No. 3 - May/Jun 1994 - $4.00



Model I, III & 4

LITTLE ORPHAN EIGHTY

I have recently had a somewhat interesting experience. My oldest son, Alan, is attending a junior college here in the San Fernando Valley, and one of his courses is Computer programming using Basic. Now, I always considered myself a fair programmer and, having taught Basic for some years at a competing college, I thought that I might be able to help him.

Sure enough, after a couple of classes, Alan asked his ol' Dad for assistance. The homework assignment was nothing special, but I realize, for a beginner, it wasn't easy. Anyway, I sat down with him in front of the computer and we proceeded to write the program step by step.

Since Alan understood what we had done, I felt really good about being able to help him with his homework. But, as it turned out, it was the kind of help he didn't need. He received an 'F' on the assignment. Why? Because we had used GOTO's. It seems that GOTO is, indeed, a modern four-letter word!

I hit the roof, and I am still angry about it. So angry, in fact, that I am writing about it in this column.

What happened to Alan is not an isolated case. I have talked to — make that argued with — teachers who are proponents of structured code. Now, if the language itself requires structure, such as Pascal, then so be it! But I resent it when someone tells me, or anyone else, to do a job, and then not allow the use of all the available tools. I mean, how would you like to write a novel — only one catch — you are not allowed to use the letter E.

My view is, that if GOTO is good enough for the program that wrote Basic (assembler has the JP instruction), then certainly it is good enough for Basic itself.

But that view is not shared by the structure freaks. They think that you, the programmer, are too stupid to write code. You must do it their way, or it is no good. They want to prevent you from programming yourself into a corner, or if you somehow should get lucky enough to write a program that actually works, it must be easy for someone else to understand it and maintain it.

This is pure BalderdaSh, and it is indicative of where we are heading, not only in computing, but also as a country. It has been defined by a famous radio personality as 'the Dumbing of America'.

Over the past couple of decades, our schools have changed drastically. No longer do we cater to the students who excel, we now slow the learning down to suit the least able. The result is that we now have graduates that can neither read nor write. For the first time in history we have a generation that is less educated than the one before them. Heaven help us!

You may find this paranoid, but I see this trend spill over to computers. We are now catering to the lowest common denominator. For the most part, the popular computers (PC and Mac) are being operated with pictures. Could this be because the users can't read?

Going further, programming is being watered down to cater to the unimaginative. In the old days we were given a set of commands and told to go to it, make something useful — anyway you want to; now, it seems, the emphasis is not on whether the program works, but rather if the code is easy to read. Maybe, if we could somehow manage to print the actual code as pictures....!!

Programming, like any other art, requires many years of study and practice. It is not easy. There are people who have the talent for programming and there are people who don't, and that's as it should be. I mean, you really don't want EVERYBODY to play professional baseball, do you? Going to a game wouldn't be much fun.

Programming classes should be taught to emphasize imagination. Forget structure. Let the students break every rule; let them program themselves hopelessly into a corner. The good ones will figure out what they can and cannot do, and the no-talents will give up.. I have no problem with that.

My favorite story about structured programming is really about music. Many years ago I played with a guitar player named Jimmy Amerson. Jimmy was an extremely talented musician, but he was self-taught. One night, Paul, a guitar teacher was in the audience. During a break he mentioned to Jimmy that his playing style was wrong. Hmm! Jimmy asked he he'd brought his guitar with him, and since he had, Paul was invited on stage to jam. If I remember correctly, we started on a medium-shuffle blues with Jimmy and Paul trading choruses. Obviously, before long it became a cutting session. Verse after verse found Paul trying to keep up with Jimmy, but he just couldn't do it. In the end, Jimmy had blown him off the bandstand. For all the correctness and style, Paul just couldn't compete.

The bottom line was not how the music was played, but rather, how it sounded.

The bottom line in programming is not how elegantly and structured the program is written, but DOES IT WORK!!

TRSTimes magazine

Volume 7. No. 3 - May/Jun 1994 - $4.00

PUBLISHER-EDITOR
Lance Wolstrup
CONTRIBUTING EDITORS
Roy T. Beck
Dr. Allen Jacobs
TECHNICAL ASSISTANCE
San Gabriel Tandy Users Group
Valley TRS-80 Users Group
Valley Hackers' TRS-80 Users Group

TRSTimes is published bi-monthly by TRSTimes Publications. 5721 Topanga Canyon Blvd., Suite 4. Woodland Hills, CA 91367. U.S.A. (818) 716-7154.

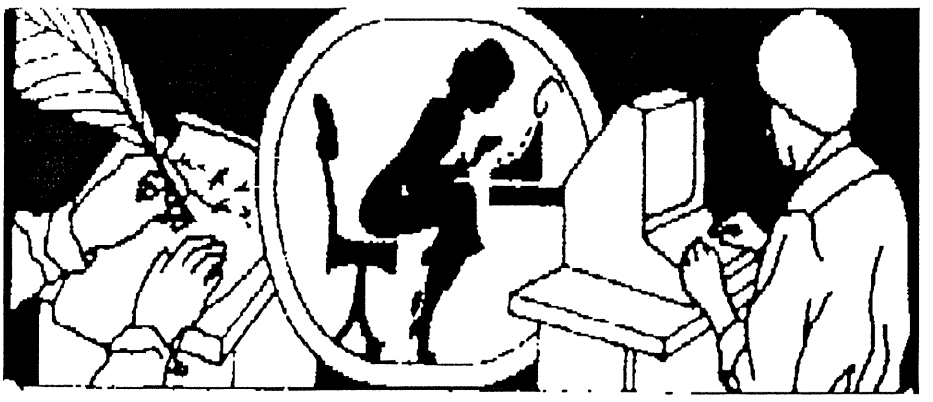
Publication months are January, March, May, July, September and November.

1994 subscription rates (6 issues):
UNITED STATES & CANADA:
$20.00 (U.S. currency)

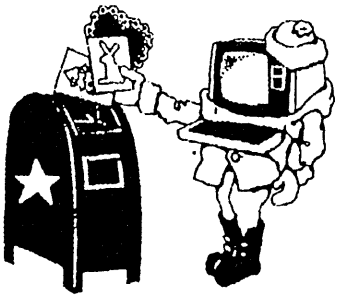
EUROPE, CENTRAL & SOUTH AMERICA:
$24.00 for surface mail or $31.00 for air mail. (U.S. currency only)

ASIA, AUSTRALIA & NEW ZEALAND:
$26.00 for surface mail or $34.00 for air mail. (U.S. currency only)

Article submissions from our readers are welcomed and encouraged. Anything pertaining to the TRS-80 will be evaluated for possible publication. Please send hardcopy and, if at all possible a disk with the material saved in ASCII format. Any disk format is acceptable, but please note on label which format is used.



THE MAIL ROOM



ISSUE 7.2

I read with complete absorption "Mrs. TRSTimes" riveting account of the earthquake. Boy, can she write a gripping narrative! It was almost like being there! The only time I'm more 'at the edge of my seat' is when I'm watching 'Star Trek: The Next Generation". That's the greatest compliment I can think of! I am glad that no one in your family was hurt.

Anyway, I read on the back cover that there are 4 volumes of the "Z-80 Tutor" available. I didn't know this! If mentioned in a past issue of TRSTimes, I sure missed it. Where can I get all four volumes? How much are they? Can they be gotten from you?

John E. Grant, Jr.
West Columbia, SC

I agree. Mrs. TRSTimes is, indeed, an excellent writer. She is on the Board of Advisors of 'Men's Fitness' magazine, where she writes frequent articles, using the name Sylvia Cary. She has also written several books, educational movies, as well as numerous articles for various national magazines. I'm proud of her.

We do not handle the Z-80 Tutor books, but they can be obtained directly from the author, Chris Fara of Microdex. He can be reached at 1212 Sawtelle, Tucson, AZ 85716.

Ed.

I would just like to say a thank you to Sylvia for her article on the 'quake'; it brings a little understanding to us who rarely feel the effects of these things, and even when we do, they are like that of a very heavy lorry passing and nothing more.

Another line of enlightenment was Sylvia's description of the layout of your house and of your families; it brings things that much closer.

I hope you were not too affected by the recent aftershocks we read about; it seems to me like living with a time bomb and something I can well do without.

Tom Ridge
Surrey, England

Sylvia says 'thank you' for the kind words about her article. We have had thousands of aftershocks since the 'Northridge Shake', but we are getting so used to them that we don't even pay attention anymore. On the other hand, there have been a few that, had it not been for the 6.8, they would have been considered quakes in their own right, rather than aftershocks - those we noticed with various degree of nervousness.

A time bomb - yes, I guess you're right. But it is sooo exciting. Hmmm!

Ed.

HIDEE

Just a few words on HIDEE (see ad elsewhere in this issue). The program was written by Jerry Gosmire, another TRS-80 nut here in South Dakota. We have been hammering on it for about a year, and it is now something that is really super fantastic. I told Jerry that the program is so nice that everyone should check it out!

It's truly amazing the way it runs - instead of flipping regular screens, as in LOW-RES using DIRECT by Chris, this can be used to call DMENU/X10....on and on.... to match the DIRECT/X10 files, so each file calls a Hi-Res screen that you can create, or use the ones that will be included in the package. It's really superb. We've also included a Westminster Chime, instead of the usual beep that's heard on the hour! Awesome - sometimes I don't know whether I'm using a Macintosh or a Model 4...!!

The program has been written into a self-run JCL file to install it into the DIRECT/CMD file. Then you install it, just like DIRECT on SYS13 and away you go! Real nice. Right now we are working on a Moving Screen Saver, instead of the BLANK screen when the screen times-out. Should be a real winner.

Andrew Miller
Sioux Falls, SD

It is always nice when there's new software for our TRS-80's. I wish you luck with the program.

Ed.

BIT FIDDLING

I just got a new pocket calculator from Texas Instruments, a 35X, as an "update" for my old SR-51-II, when TI was unable to replace its battery pack. Charge: Can$ 17.5 (approx. $13.50 U.S.), and one store in Windsor quoted me $33.54 for this model. Not a bad discount!

The calculator has three functions on each key (164 total), 3 memories, aside from the usual scientific functions, one and two variable statistics, fractions, 10 metric conversions, and best of all, Hex, Oct & Bin, with Boolean logic functions, NOT, AND, OR, XOR and XNOR (wherever that is).

Playing with it, I remembered Chris Fara's "BASIC IMPS" article in 6.2 page 24, and your "PEEKING & POKING" article in 1.1 page 4. It seems that the XNOR is the equivalent to Chris' EQV: 0 XNOR 0 = 1, 0 XNOR 1 = 0, 1 XNOR 1 = 1. No IMP on it, though.

But then it got confusing (probably one of the reasons I never got far into assembler). Trying some of the AND's and OR's in your article, and checking what the computer put at that address and what the calculator came up with, things got muddy. Except for the error in the Break key address (which I think I mentioned once before, it should have been 7CH, rather than 74H as printed) your POKE's all work.

But, for instance, at 74H (36H) AND'ing 223 gets 16H on the calc, but the computer shows a 4H at 74H?? Pencil and paper confirm the calc: 0011 0110 AND 1101 1111 gets 0001 0110, 16H. Why does that not show up at the 74H address? Jump somewhere else? The reverse OR 32 miraculously restores the 36H. As I said, too far out for me.

I also could never understand why go to the lengthy POKE/PEEK/AND/OR syntax, why not poke the proper Byte? It would make a Basic program shorter and simpler! Is that a hangover from working in binary, or does it have a valid reason? In the above, POKE 116,4 as well as POKE 116,16 switch to lower case, while POKE 116,36 restores upper case (which is my boot-up setting). Well, I am confused.

Maybe an article going a bit more into depth about Boolean logic, its use, what's in the FLAG table, etc. would be of interest to some of us, not just to me. I have no contact with anybody else to ask these questions - my BASIC knowledge came mainly from the childish looking, but very good, "Getting Started With TRS-80 BASIC" book, and nothing similar is around for Assembly Language. Maybe I should start another attempt to dig through Chris' Tutor.

Henry H. Herrdegen
LaSalle, Ontario, Canada

I have a calculator (Casio fx-115v) that is similar to yours. It also has the Dec/Hex/Bin/Octal conversions, as well as the Boolean logic functions NOT/AND/OR/XNOR - which is why I bought it in the first place. Sounds like you got a nice deal.

If you look a little closer at the XNOR function, you'll see that it does not produce a results of 0 and 1, rather:

0 XNOR 0 = -1
0 XNOR 1 = -2
1 XNOR 1 = -1

Just for the record, the Model 4 EQV function works exactly the same.

Henry, memory location 74H (on the Model 4), among other things control the status of the keyboard. If bit 5 is ON (1), the keyboard is in upper-case mode - if bit 5 is OFF (0), it is in lower-case mode.

You have configured your Mod 4 to boot-up in upper-case, so bit 5 of memory location 74H is ON. But this is where you make your mistake, and thus confuse yourself - if you check the contents of 74H (from BASIC typing PRINT PEEK(&H74), you will get 36. That is 36 **decimal**, *not 36 Hexadecimal. BASIC always returns PEEK values in decimal.*

Now that we have the correct starting value, the AND 223 (turning OFF bit 5) will produce 4 in the machine, as well as on the calculator. The reverse (turning ON bit 5), 4 OR 32 produce the number 36, again, that is 36 decimal, not 36H.

All operating systems that I have ever worked on have data tables that determine the configuration of the computer. In the case of the Model 4 and LS-DOS 6.x.x., one such table is the FLAG$, which begins at memory location 6AH(106) and continue for 26 consecutive memory locations. Many of these memory locations are bit-mapped; that is, each bit in the 8-bit byte, holds a piece of information about the machine configuration. A good example is memory location 7DH, known as DFLAG$ - it contains 8 pieces of crucial information, each depending on the status of a particular bit.

if set
bit 0 - SPOOL is active
bit 1 - TYPE ahead is active
bit 2 - VERIFY is on
bit 3 - SMOOTH active
bit 4 - MemDISK active
bit 5 - FORMS active
bit 6 - KSM active
bit 7 - accept graphics in screen print

On a virgin TRSDOS/LS-DOS 6.x.x disk, the value of this memory location is 0AH or 10 decimal. This translates to 0000 1010 in binary, which means that Type ahead and SMOOTH are active. But when you write programs that will be used by other people, you just cannot assume that this (or any other memory location for that matter) will contain a specific value. As you can tell, it is quite possible that 7DH has been changed to a different value; for example,

FORMS might be active - in which case the value will be 2AH - 42 in decimal - or 0010 1010 in binary. If we now simply POKEd 10 back into this location in order to make TYPE ahead and SMOOTH active, we also manage to turn off bit 5 (FORMS) whether or not we wanted to. I am sure you can see that simply POKEing a predetermined value anywhere is not a good idea.

Thus, in order to avoid chaos, we first pick up the value that's already stored in the particular location, then we turn on/off the desired bit(s) with AND/OR, and then we POKE in the new altered value. Yes, the syntax is a smite longer and a little more difficult to understand, but it is the only way to fiddle with the bits. Hope this answered all your questions - also see the 'BITS & PIECES article elsewhere in this issue.

Ed.

ELECTRIC WEBSTER

Rumaging through a box of diskettes (donated by a friend a year or so ago), I came across Electric Webster, the spelling checker and hyphen/grammar package. I copied the files to my 3 1/2" disk and discovered it had been set up to run with LeScript.

Looking further, I noted a CONF/CMD and a CONFGRAM/CMD. Aha! Configuration files, which allegedly will install Electric Webster to work with LazyWriter, AllWrite, or LeScript. It runs OK, except for one thing - I installed it to go back to AllWrite and it keeps asking me to put a floppy in with ESCRIPT/CMD. Got no such, and I figure it really is looking for LESCRIPT. But why? I installed it for AllWrite, I thought...

So the CONFGRAM/CMD program has a problem. Got my DED6/CMD loaded (my trusty DiskZapper) and asked it to find all occurrences of ESCRIPT on the 720K disk. Obviously, one (or more) of them needs to be changed to AL/CMD. I found it on track 59, sector 21. It was in the CORRECT2/EW file. Did an ASCII change (easier than doing HEX, right?), exited DED6 and ran EW (Electric Webster) for effect. Fantastic - exited EW and brought up AllWrite - no more errors!

I now have my Wordprocessor, DotWriter, and the Spelling-checker with the grammar/hyphen utilities all on the same floppy. Oh, the joys of having 720K disks.

The moral of this story is "you have to fix it yourself" since everybody went out of business. I had no idea that I would find the solution to the problem - but I was sure going to try!

Electric Webster can be used without your wordprocessor - just type EW and it comes up and asks what file you want to process. But it is so nice when you can hit the 'Hot Key' in AllWrite, check your work, and then return to AllWrite.

I now have all the above on TRSDOS/LS-DOS 6 on the Model 4, while in Model III mode on NEWDOS/80 I have AllWrite. DotWriter and Electric Webster (without the grammar/hyphen feature). I am pretty well set now using either of the machines.

It comes to mind that the reader might wonder why Electric Webster had to be installed in the first place if it will run without the word processor. Good question! Most word processors have formatting commands that are unique, so the spelling checker needs to know which you are using. Keeps confusion and crashes to a minimum...

Kelly Bates
Oklahoma City, OK

Yes, running into a problem, and then solving it, is a very satisfying experience. Congratulations. Having seen Roy Beck's demonstrations of AllWrite with Electric Webster at several club meetings, I recognize how powerful the programs are when used together. Indeed, the TRS-80 is still a very capable machine, blessed with superior software.

Ed.

BEAT THE GAME

by Daniel Myers

CUTTHROATS

Welcome aboard, matey! Dust off your scuba tank, shake out your flippers, and prepare to go treasure hunting. But first, a word from our sponsor. Cutthroats, like most Infocom games, has several solutions. This walkthrough will show you one way of completing the adventure. However, there are others, so when you've finished, you might want re-play the game, doing different things, to see if you can come up with another way of recovering the treasures successfully.

Also, you should be aware that you can only recover treasures from 2 of the ships, the Sao Vera and the S.S. Leviathan. The other wrecks are only red herrings, and you don't have to bother with them. Which of the two real wrecks you will dive for depends on the item you are shown by Johnny Red. If he shows you the gold coin, it's the Sao Vera; if it's the dinner plate, then the ship is the Leviathan.

Further, most of your actions up to the dive itself will be pretty much the same, so this section of the walkthrough will take you up almost to the dive itself. After that, consult either the Sao Vera section, or the Leviathan section, depending on which ship you're investigating.

Ok! The game starts with a long lead in, explaining how you came by the book of shipwrecks. You will have to sit through this on each boot-up; no way around it. After that, the game really begins, with you lying in bed in your scruffy room at the Red Boar Inn. The first thing to do is stand up, then wind your watch (time is important in the game, and if your watch runs down, you can't keep track of the time).

There's a note on the floor. Read that, then open your dresser. Inside are the shipwreck book, your bankbook, and a room key. Get the key, open the door, go out, and lock the door again. You don't want to leave the door open, or the Weasel will come by later and steal the shipwreck book. If that happens, the game is over before it even starts. You don't need to take the book with you, so locking the door is effective here.

Now, go downstairs and out to the Wharf Road. Follow the road East until you get to the Shanty. Enter the Shanty, and you will see Johnny Red and Pete the Rat already there. Sit down and order breakfast, then wait for Weasel to show. Order a glass of water when you get thirsty. While you're waiting, you might want to listen to the parrot. He doesn't have anything important to say, but you might get a chuckle out of him.

Eventually, Weasel will arrive, and Johnny will ask if you're interested in doing some treasure hunting. Say yes, and then Johnny will have you all meet again a little later at the lighthouse, in order to keep McGinty from finding out what you're up to. After that, leave the Shanty, go back West to the end of Wharf Road, and from there Southwest twice and Northwest once, which brings you to the lighthouse. Now, wait for Pete, who will be the last person to arrive.

Once Pete gets there, Johnny will show an object, either the coin or the plate. This indicates which wreck to dive for. After that, he'll give further instructions, which you should read carefully. When he's finished, go back to your room at the Red Boar. Get your passbook. If you're diving for the Leviathan, also get your scuba gear from the closet (scuba gear not needed for the Sao Vera).

Leave the room (lock the door behind you!), and go back out. Walk East along Wharf road to the end,

and go Southeast to the Ocean Road. If you're going to use your scuba gear for the dive, go Southwest into the alley, and drop your scuba gear there. You don't want McGinty to see you lugging it around.

Follow the Ocean Road south to the end, then go Southwest to the Ocean Road, and North into the bank. Make your withdrawal, then leave and return to Ocean Road, where you go Southeast to Point Lookout. Drop your passbook here (that Mcginty has sharp eyes, and you don't want him to see you with that, either), and wait for Johnny.

When Johnny arrives, show him the money you just took from the bank. He'll be satisfied, and then ask if the wreck is more than 200 feet underwater. Answer yes if it's the Sao Vera, no if it's the Leviathan. The two of you will then head back to International Outfitters to rent a ship and purchase supplies and equipment. McGinty will be in the store when you get there. However, just wait, and he'll leave eventually.

Johnny will make his purchases first, and you will have to chip in some of the cash you're carrying. However, you will have plenty of money left over to buy whatever you need. When it's your turn, buy the flashlight and the shark repellent. If you're diving for the Sao Vera, that's all you need. However, if you are diving for the Leviathan, also buy the following items: C battery, putty, and electromagnet, and also rent the small air compresser (so you can fill your tank). All these items will be delivered to the ship for you, so you don't have to take them with you.

Now, it's time to uncover a little double-dealing. Leave Outfitters, and go back East along Wharf Road to the end, then Southeast again to Ocean Road. Go along Ocean Road to the end, then Southwest to Shore Road, and continue West along Shore Road until you reach the Ferry dock. Wait around.

Soon McGinty will appear, and a short while later, Weasel. The two men will go off to a corner and talk. Then Weasel will hand something to McGinty, and board the Ferry (you can't get on it yourself, but you have other things to do, anyway). Ok, now you've seen that, go back to Ocean Road, and then into the alleyway.

The alley runs behind all the buildings, and it will come in very handy! Go West along the alley (pick up your scuba gear if you dropped it here earlier) , until you're standing behind the vacant lot, which is next door to McGinty's. Wait here, and McGinty will come by, heading from East to West. Continue waiting, and he will soon re-appear, going from West to East (he is walking along Wharf Road, of course).

Once you see him the second time, go West once, and you're behind his store. The door is locked, but you can open the window and get through into the place. Here you will find an envelope that proves the Weasel is out to double-cross you all. Get the envelope, then leave by the window.

Go back along the alley to the Vacant lot, then go straight North until you come to the dock where the rental ships are moored. Both ships have approximately the same layout; they are slightly different on the top deck, but below they are exactly the same. Enter whichever ship has been rented for the dive, and go below deck. Then go north until you reach the crews quarters, and hide your envelope under the bed. You don't want Weasel to know you have it (he'll kill you), and if you show it to Johnny now, you'll cancel the expedition.

Now you have to do some more waiting. The delivery boy will come around, and drop off the items you've bought. Then the others will start to arrive. When Johnny comes, go to the Captain's Cabin, and tell him the longitude and latitude of the wreck, which you can easily get by looking at the shipwreck book that came in the game package. Then go back to the crews quarters, and wait some more. Eventually, you'll reach the dive site. At this point, you should now read either the Sao Vera section or the Leviathan section, whichever is applicable.

LEVIATHAN

Ok, so it's time for the Leviathan. Get up, then go North to the storage locker. Here you will find all the things you bought at Outfitters. Put on your wet suit and flippers. Get the drill and the C battery, open the drill, put the battery inside, and close the drill. Get the remaining items, except the compresser. Fill your tank with the compresser, then go South. Along the way, get the envelope from under the bed.

Stop in the galley to eat and drink, then continue on South to the Captain's Quarters. Show Johnny the envelope. That will take care of Weasel! Now go North and up. Put on your tank and mask. Johnny will tell you about the orange line, but for this dive, it won't be needed.

You're all set, so dive in! Once underwater, turn on your flashlight, because it's going to get dark pretty soon. Oops! A shark just showed up! Good thing you have the repellent. Open the canister, and

the shark will take off. Now, just keep going down until you reach the wreck.

You're on the top deck of the Leviathan, with a hole at your feet. Go down through the hole, to the Middle Deck. Here, you can only go up or down, so go down again, to the Below Decks area. From there, go South, to the room with the closed door. You might want to read the sign on the door before you open it.

Once past the door, you're in a mine locker. All the mines are tied down, except for one loose one, floating in front of a hole. Fortunately, you can take care of that problem without difficulty. Touch the magnet to the mine, then turn on the magnet. Drop the magnet (why that doesn't blow you to bits, I don't know, but that's how it works). Now you can go up through the hole.

You're on the Middle Deck again, although a different part of it. The way South is narrow, so remove your tank, then go due South until you come to the room with the safe. This is the tricky part. Turn on the drill, drill the lock, and then *immediately* turn off the drill again. Otherwise, it will burn out, and you'll have a big problem later!

Ok, inside the safe is a glass case containing some valuable stamps. Alas, there is a crack in the case, and water is starting to seep in. However, don't be alarmed; you'll have enough time to fix that. Go back North to the room with the hole in it. Put your tank back on. Go through the hole into the mine locker, then North, then up through another hole.

Surprise! This room still has air in it. Good thing, too, because the water level in the case was starting to get too high for comfort! Now, turn on the drill, and drill a hole in the case. As the water drains out, the drill dies (lasted just long enough). Now, open the tube of putty, and put the glob of putty on the hole. The putty will seal both the hole and the crack.

And that's just about it for the Leviathan. All you have to do now is go back through the ship, and up to your own boat, where your comrades are waiting. Congratulations! You're now a very rich diver!

SAO VERA

So, it's off to the Sao Vera. This one has a few more obstacles than the Leviathan did, but none of them are particularly difficult. The first thing is to get off the bed, and head North to the Storage Locker. Here you'll find the flashlight and repellent, as well as a deap-sea diving outfit. There is also a small machine here, that you won't be needing (it's a locater box. If you really want to fiddle with it, you have to buy a dry cell to make it work).

Get everything but the box, then go back South. Get the envelope from under your bed, stop off in the Galley to eat and drink, then continue on to Johnny's cabin. Show him the envelope, which will put an end to Weasel's double-cross. Now wear the suit and go up on deck.

Johnny will be there, and will tell you about the orange line. Keep in mind what he says. If you look around, you'll see a large air compressor, with an air hose. Attach that to your suit, and then turn on the compressor. You're all set, so dive in!

Once underwater, turn on your flashlight. There's that pesky shark again! Open your canister to get rid of it, then keep on going down. It will be a long way down, but you'll get there.

Now you're on the top deck of the Sao Vera, with a hole at your feet. Go down the hole. Crash! Looks like the ladder broke. You may have a problem getting back up again! Then again, maybe not. Leave that for now, and make your way South, into the room with the iron bars. Get one, because it will come in handy soon.

Then keep going South, until you come to the room with the bunks barring the way. Move the bunks with the bar, then wedge the bar under the bunks to keep them from moving back. Now you can go South again, to another room, with a ladder leading down. Climb down that one.

Oops! Crash again! This time, though, the whole ladder didn't crumble. Still, it's going to be hard to reach it on your way back. No matter, you still have to find that treasure, so go North.

Uh Oh!! There's a giant squid here! Good thing for you it's asleep. And if you're smart, you won't wake it up! So, just go right on by, don't try doing anything to the squid at all. In the next room is an oak chest, along with a hole in the side of the ship. Leave that for now, and keep going North.

In the next room are some skeletons, remains of the crew. Examine them, and you'll see one wears a scabbard. In the scabbard is a sword. Get that, and go North again, to the last room. Here you will find a maple chest. The chest is to heavy to carry, so push it back South until you come to the oak chest (note: you must say "Push Maple Chest South").

Hmmmm, now, which chest to take? Let's try the oak chest. Push that out West through the hole (carefully! You don't want to cut your air supply!). Wait awhile, and the orange line will appear. Get that, tie it to the oak chest, and tug on the line. The chest will slowly make its way upward, while you return to the ship.

Now, push the maple chest south, past the sleeping squid, and south again into the room with the ladder. Climb on the chest, and you'll be able to reach the ladder and climb back up to the middle deck.

From there, go North until you reach the room with the cask in it. Now, push the cask north with you, until you come back to the room with the mast and the rope tied around it. Climb on the cask, then cut the rope with the sword. Drop the sword (you can't leave with it), and then make your way up and out.

Once on the top deck, just keep going up until you're back on the boat. The chest will be opened to display hundreds of gold coins. Congratulations, you're now a very rich diver!

UTILITY M.A.D.NESS

The M.A.D. Software Utilities Disk #1

Reviewed by Dr. Allen Jacobs

M.A.D. Software has produced a truly professional quality integrated set of TRS-80 utilities called the Utilities Disk #1. It is a set of DOS level UNIX-like programs that add a dimension of utility unique to LS-DOS/TRS-DOS in the TRS-80 Model 4 family of computers. However, the manual occasionally notes that some of the utilities can apparently be ordered for the Model III if it is run under LS-DOS 5.

The *NIX family of systems provide small and large network multitasking and multiprocessing the resources that single user systems simply can not utilize for most applications. Many of those "big system" features are largely invisible to the single user. In theory, the true location and processing of programs and data in a large networked system is supposed to be invisible to the individual user. Thus, if a task is actually running on some giant server in another city rather than on a user's desktop machine, how would the user know or care? The truth is, they wouldn't. Thus, what is really most important to the user is how the computer reacts to the user.

So, what makes the difference between a good DOS and a great DOS to a user? Aside from the applications available that run on the DOS, the most significant difference is the availability of utilities that give the user direct knowledge and control over both the performance of the operating system and its files, wherever they may be. In the case of most personal computers, those files reside in our own machines.

To that end, M.A.D. Software has written a set of utilities called the Utilities Disk #1 that give LS-DOS 6.3.1 many of the user relevant features that make the various members of the UNIX family of operating systems say that they love their DOS. In their documentation, M.A.D. Software refers to these systems as the *NIX Systems. The use "*" is a result of a legal dispute between Unix System Labs (USL) and University of California at Berkeley (who made Unix usable and got it out of the labs) over the ownership of all or part of the Unix system. This dispute was resolved in March 1994.

The M.A.D. in M.A.D. Software stands for Michael A. Durda, who is the brother of Frank Durda The Fourth. Frank is the author of the boot ROM running in every Model 4P TRS-80 machine ever made. I am told that his signature appears within the ROM code as "FDIV". Because Frank worked for Tandy, who does not like employees supporting or selling products for any type computers on the side, he created M.A.D., using his brothers name. While Michael ran things, Frank wrote nearly all the software and firmware that the company sells. However, he no longer works for Tandy, so Frank now takes care of nearly all aspects of M.A.D. Software. He has rewritten the ROMS to allow booting of all Model 4's, directly from a hard disk without the requirement of a boot floppy. He also included some other improvements. That is no small feat. I have included this background just so that you might become aware of who M.A.D. Software really is, and so you would have some idea of the truly professional level of the technical capabilities.

The disk utility set consists of nine major utilities and their attendant overlay and configuration files. There is also a well organized and extensive set of documentation files for each of the utilities. The documentation is available on two media. Both are included in the package. The documentation comes both professionally printed, and in ASCII (on disk). That is the best combination possible. Printed documentation is easier to read while ASCII files are easier to search.

What makes these utilities unique is the rich number of parameters each of them has and their modular interconnectivity. While some are "just" helpful, others virtually transform the essential character of LS-DOS into a quasi-multitasking environment. Thus, the TRS-80 begins to act much like a UNIX system, which is what seems to be the overall intent of the utilities set.

The utilities included on the disk are: LOOK, MAPMEM, MORE, LS, OOPS, WC, XLR8SET, FORCEHI, and PIPES. They are all called from the command line with parameters optionally entered after the command, separated by a space and a "-" preceding the first parameter, which is usually a single character. Thus, help is provided for a utility by typing in the utility's name followed by <SPACE><-><?><ENTER>. Using <-><h> will also work.

With the daunting array of specifications available to the user within each of the programs of this utility set, re-typing each desired option every time the utility is invoked could become tedious. Also, there will probably be a limited set of options for

each of the utilities that a single user will finally decide is useful for his or her purposes. In a file called DEFAULTS/MAD, the desired configuration for each of the utilities in the set can be saved. This file is consulted by each of the programs when they run, unless the "-!" option is specified. The file is an ASCII text listing of the name of each utility. After the name, the user may list the desired options to be invoked each time the program is run. Each utility can be saved under a number of different names in order to provide multiple sets of options for the same utility. This system provides a near "named macro" convenience mechanism for calling the programs in the utility set with pre-specified options. While I don't recall reading any specific warning in the manual against renaming a utility with the exact spelling of a DOS command, it is not advisable to do so. DOS will execute the command before it ever searches for a program with the same name.

Some of the included utilities work with flexible file specifications (filespecs) called wildspecs. When using wildspecs for filenames, an asterisk ("*") between two characters specifies that a variable number of characters in a filename will be recognized as being included within the specified group of files; whereas, a "%" will allow variability only in the character position in which it exists. A range of acceptable characters can be specified by a "-" between the limiting character values; whereas, specific acceptable characters can be specified for a filename position if they are surrounded in "[" and "]". Drivespecs can also be selected in ranges. Filenames are all converted to upper case. The examples shown in the manual make the true file specifying power of these wildspecs more apparent than can be presented here.

LOOK

Those of us who have used Super Utility Plus know that by pressing "!" and a drive number, the program will indicate which TRS-80 DOS was used to format an unknown disk placed into that drive. The problem with SU+ is that if the disk is not in a TRS-80 format, there is no simple means we have of identifying its DOS beyond TRS-80 systems. Thus, having floppies that include CP/M and IBM formats, we are forced to boot up almost every computer system we have at home, only to determine that the floppy is, in fact, not formatted. That determination can take about 15 to 20 minutes and we're never sure that our conclusion is correct. Alas, a TRS-80 program that can make that determination in one pass has been an unfilled void in my wish list for a long time.

LOOK addresses this deficiency by being able to determine the formats of TRS-80 (including Model II/12/16/6000 formats), MS-DOS format, and those formats of selected CP/M disks. Also, while it can not read them, LOOK can actually determine if a 5 1/4" floppy disk has been written in IBM AT High Density Format. The user can also supply additional formats which LOOK can thereinafter recognize. On an LDOS 5 or an LS-DOS/TRSDOS 6 disk, LOOK can map the location of files on both physical and logical drives (devices) including diskDISKS.

With LDOS 5 LS-DOS/TRSDOS 6 disks LOOK will optionally display allocation information by physical location on a disk or by any file or specified combination of files, including, of course, all of them. LOOK will accept drive (device), filename:drive, cylinder, granule, and sector specifiers. The display can always be paused with the familiar <SHIFT>@ combination and resumed by pressing any other key. With the addition of a ">" specifier, LOOK will send its output to a user specified file on any specified drive (device). If the ">>" specifier is used, the output will be appended to an existing file. The ";" command can be used to separate commands that will then be executed sequentially. While LOOK's multiple command execution capability from the command line is not quite as sophisticated as PIPES, the differences will be explained later. Actually, LOOK can optionally be run under PIPES.

Favored combinations of user specified options can be stored in the DEFAULTS/MAD file. New and previously unspecified disk formats can be added to those already present in the LOOK/INI file. Once specified, LOOK can recognize and identify these formats again, on other disks.

MAPMEM

MAPMEM does for memory what LOOK does for drives. It summarizes the amount of low and high ram that is currently available and how much low ram was available at the time the system booted. Also, it detects the presence, size, and location of all modules in memory and optionally displays the information in tables, in decimal or hex. MAPMEM categorizes the information into four tables. They are: available low and high memory, enabled disk drivers, character drivers and filters, and other modules not accessed as drives or character based devices. The tables contain the starting address of each module, its ending address, its length, its name, and a brief description of its purpose.

A maximum of 50 entries with each entry containing a 35 character description of a specifically named module may be placed in a file referred to by MAPMEM called MAPMEM/INI. This file is con-

sulted when the program runs. Newly developed memory modules can thus be included in subsequent memory mappings.

The options for MAPMEM are appropriate to its function and follow those available for other programs in the set. Each of the tables may be included or eliminated from the from the output. Additionally, the length of each module may optionally be displayed in decimal instead of hex. The normally displayed headers for each table may optionally be omitted. The display can be paused and resumed with the <SHIFT>@ combination, alternated with any other key to resume. The output can be redirected to a file and optionally appended to an existing file with the ">>" specifier previously described. As with the other utilities in this set, the output can also be optionally redirected through MORE.

MORE

MORE is best described as the closest TRS-80 program to Vernon Buerg's "List" in the MS-DOS world. It is a file display utility through which the output of any of the programs in this set that produce output can be redirected. MORE can also display the contents of any file or group of files, using the wildspec file specification options available in all the utilities in this set. Also, its output can be redirected to a file or other device.

MORE can display a file's contents a line at a time if the <ENTER> key is pressed or a screen at a time with the press of the <SPACE> bar. Pages are normally advanced by scrolling new lines from the bottom. However, new pages may optionally be started from the top by sequentially erasing lines from the old page before printing new lines. Pages are normally displayed with a --More--prompt after the last line of the page. If desired, MORE can display a brief help message each time the --More-- message is displayed. The "?" at the --More-- prompt will always display help.

MORE has other display options. It can be made to count logical lines rather than screen lines. Its effect is noticeable only with lines longer than 81 characters. Recognition of the new page character (^L) can be turned off to prevent MORE from prematurely paginating the output when it encounters a new page character. Optionally, control characters can be displayed with a caret (^) preceding a letter rather than be ignored. Multiple blank lines can be removed from the display of a file if desired. The default reverse video representation of underlined text can be canceled by the user. MORE can optionally ignore its default configuration in the DEFAULTS/MAD file by invoking the "-!" optional command. If specified as a number "-n", MORE will use that number as the line count to display before another --More-- prompt will appear. A "+n" number on the command line will cause MORE to skip that many lines before it begins to display the current file.

A "/" with a user specified pattern will cause MORE to skip the display of the lines in the current file until the specified pattern is found. If the pattern is preceded by a "^", the specified pattern will only be recognized if it begins a line. Pattern searches are case sensitive. Also, the patterns searched for may optionally be specified to be for a range of characters in any position. If a search is unsuccessful, the point of display is relocated to the beginning of the specified file. However, if the file was redirected from other programs such as PIPES, its display point resides at the end of the file.

As previously described, MORE can use wildspecs for filenames. As in other utilities, the output of MORE can be redirected to a specified file or appended to an existing file. When MORE is invoked by another program, the last screen of the last file sent to the screen might be cleared from view before it can be read as MORE terminates. To prevent this, a warning message that MORE is about to terminate can optionally be invoked. Thus, the prompt --No More-- will precede the exit of MORE. Also, <SHIFT>@ will pause the display, and an additional command separated by a ";" on the command line will be executed.

At the --More-- prompt, a number of commands can be specified and the default values these commands use can be altered. As previously noted, the <SPACE> key causes the next screen of a file to be displayed. While the default is 22 lines, a decimal number of lines optionally specified before any <SPACE> key will change the default number of lines displayed to the screen to the number specified. In a like manner, an optional decimal number of lines to advance through the file each time the <ENTER> key is pressed can be increased from its default value of one. An optional number of lines to advance instead of the default number of 11 for the <CONTROL-D> combination can be specified in the same manner. A decimal number preceding an "s" will cause the display of that next number of lines to be skipped. A decimal number preceding an "f" will skip that number of screens. A decimal number preceding a "/" followed by a character pattern which may include wildcards and the beginning of the line specifier ("^") will locate the next occurrence of that pattern in the file. After that, a decimal number preceding an "n" will skip the file to within four lines before the specified number of occurrences of the

last specified search pattern. An "=" will display the current line number while a "." at the --More-- prompt will repeat the last command. A "q" or a <BREAK> will both cause MORE to quit. However, the <BREAK> will cause MORE to generate an "abort" error.

If the file is not being read from a streaming device, then a "'" will reposition the file back to the starting location of the previous search. A :f will display the current filename and line number. MORE will advance to a specified number of files ahead of the current file with an ":n" preceded by a decimal number of files to advance. More will go to the same number of previous files with a ":p" command. If zero files are specified with the ":p" command (ie.: "0:p"), MORE relocates its display to the beginning of the current file.

In practical use, MORE allows printed files from a word processing program to be viewed from the command line. For example, it is very useful when you don't have any means of commenting a file and want to search through a bunch of letters you have written to find a piece of information such as an address. If you think about it, this is a way to redirect the addresses of everyone to whom you have ever written a letter to a single file without retyping. This could be done with a word processor, but it would be difficult to remember which files had been processed and which had not. With wildcard multiple file specs not being available on any word processor I have ever heard of, I know of no other way this chore can be accomplished in an automatic manner.

LS

LS is most certainly a natural companion to MORE. As you can almost guess by now, the output of LS can be redirected through MORE. What is LS? Those who know anything about UNIX systems recognize LS to be the illogically chosen two letter designation for the command commonly known in almost every other DOS as "DIR". However, those "in the know" also recognize that LS has far more options than any run-of-the-mill "DIR" command. LS is a virtual command line directory listing system. If LS is entered without options, a "CAT" like display with all visible files alphabetized in lower case on all active drives will appear on the screen. The optional wildspec and filespec specifiers previously described are fully operational with this utility. Additionally, just about every aspect of file listing can be optionally displayed in a number of ways. Namely, files can be listed one per line. Or, all files can be listed including those that are invisible. They can be sorted in regular or reverse order by name or by date. The drive number may optionally be appended to the filename or the file may be listed in long format similar to the standard DIR (A) command.

In this format, various attributes of each file are displayed. They indicate whether the file is a partitioned data set or diskDISK subdirectory, a system file, an invisible file, a fixed size file, an open file, a copy-protected file, a user password protected file, or a modified file that has not yet been backed-up. The level of protection of each file is also listed in this format. It is displayed as the level of access allowed, from full access to none without the correct user password.

Files may also be displayed in a stream separated by commas. They may also be sorted by time and date and displayed in uppercase or lowercase. They can also be sorted horizontally and displayed in columns. The time format may be switched between 12 and 24 hour format and the separator between the filename and its extension may be either a "/" or a ".". Of course, selective files may be displayed on the basis of wildcard and drive specifications.

OOPS

OOPS is to the DOS command what the line editor in basic is to Basic itself. It is so useful that it is simply essential. If you know how to use the line editing capabilities of the Basic interpreter and/or a word processor, then you know how to edit the DOS command line with OOPS. There are some additional options but the essential concept of the program can be no clearer than for the user to know that the command line can now be selected from any of those on the screen, and can be edited. What a pleasure!

WC

WC, the word counting utility, is the feature left out of most of the TRS-80 word processors. This is because program space is precious in the available TRS-80 ram space. Although useful, since the word count is not essential to text editing, it is often among the first features to be sacrificed to the limitations of space. However, since WC is free standing, it can do more than give a word count to the program being edited. It can optionally give a character count, a line count, and the word count of a file, in any optional order. It can also give a total of these counts across any wildcard group of files and drives the user specifies. This includes character counts of executable files. The added effectiveness of this command line level utility is that a wildcard set of files may be word counted, with the result being displayed on the screen. If the total is less than the free

space on a disk, OOPS can be used to edit the WC command line to copy the same wildcard set of files that were "WC'ed" onto another target disk, while documentation of the process is routed through MORE to a diskfile that can later be printed. We can begin to see the amazing level of integration possible with these utilities. They are actually reusable modules rather than a set of separate utilities.

PIPES

PIPES is just what the name implies. It is the basis of the "multiple process" concept for which UNIX systems are renowned. Apparently, the only difference between PIPES on the TRS-80 and an actual UNIX system is that true UNIX processes can be run simultaneously on multiple processors while our single Z-80 systems can only run one process at a time. What PIPES does is to automatically route the output of one program as input into the next program in the "PIPES" line. This allows a program to filter the output from the previous program in much the same manner as the "FILTER" command available in LS-DOS. The main difference is that the programs do not have to be written as filters, as they must be for LS-DOS/TRS-DOS. Indeed, the programs do not even have to "know" that they are filters at all! Also, they do not have to reside simultaneously, in memory modules.

Therefore, a number of programs can each have been written to run as a single utility. Yet, a sequence of these programs (or even a single program) can repeatedly be run with different arguments, on a single data file or on multiple files. This can produce powerful effects on your data. The ability to sequentially place files through multiple processes is what makes UNIX users wonder why anybody works with any other DOS. I can only imagine how convenient a conditionally controlled sequential search and replace function would be for text files, since Allwrite does not have one. If you have an external utility with that ability, then that power is automatically available in PIPES. Of course, the other single utilities on this disk can be run under PIPES because they were especially designed to do so.

For those programs and processes that are designed to be used with a keyboard to page their output or delimit input through single strokes from the keyboard, PIPES provides options that can substitute for prompted keyboard inputs. It can optionally be commanded to supply a <SPACE>, a <NULL>, an <ENTER>, or an end of file error whenever keyboard entry is requested by the program currently running. It does all this by taking over control of the standard DOS I/O devices and redirecting them.

FORCEHI and XLR8SET

These programs are improvements to the drivers for the XLR8ER adapter that many TRS-80 users have. I do not personally have one nor do I know anybody locally who has one installed. Thus, I do not have any means of evaluating these particular programs. However, many of these cards were sold, so those who have purchased used TRS-80's from their "local-leading-technology-hardware-hacker" may be pleasantly surprised to find a dormant XLR8ER board sitting in the card slot of a Model 4P where the internal modem is supposed to fit. Desktop owners will have to open their machines. Search through your disks to find the necessary software drivers. From what I have read, if your used Model 4 will mysteriously NOT run TRSDOS 1.3 due to its use of some undocumented Z-80 commands, you may discover a pleasant surprise in your machine.

SUMMARY

The concept of reusability is not new to computing. Programmers often reuse code. This is the basis of the concept of subroutines. Namely, arguments are passed to the subroutine and it processes data accordingly. Subroutines however normally exist within programs. The concept notable within UNIX (and now within LS-DOS/TRS-DOS) is that subroutines are available to the user at the level of the operating system (DOS). That is the unifying concept of the M.A.D. Utilities Disk #1.

How does it work? Basically, it works just as described in the manual, without surprises. The manual is as straight forward as are the programs. What they do, in practice, is easy to understand and use. They just become part of the operating system and I find them to be "sort of" key words in a DOS level "programming language" of its own. Namely, you don't have to call up a language and load a program. Rather, you just issue a command. If you don't remember the options, just type a <-><?> on the command line plus a space, after the program name. The DEFAULTS/MAD file flattens the learning curve to "intuitive".

The worst problem you will have with the M.A.D. Utilities Disk #1 is deciding which way you want to name and PIPE your utilities. It's the same kind of problem you have when you go the clothing store to buy a new ensemble. You sit there mixing and matching while going "M.A.D." trying to decide which items to buy. The only difference with this set of utilities is that everything fits, and you already own the entire store...

Order the Utility Disk #1 from: M.A.D. Software PO Box 331323, Ft. Worth, TX 76163 $25.00

NEW VERSION OF FOREM BBS SOFTWARE

Humor from the TRSTimes vault

A new release of FoReM ST arrived yesterday. Among the features is yet another new file transfer protocol, 'ZZZMODEM.' This new protocol transfers data in blocks of 16 Megabytes, giving it the largest block size of any file transfer protocol in the Known Universe. The checksum for each block in a ZZZMODEM transfer is sent via XMODEM, for greater accuracy. "This new protocol will allow us to transfer data at rates up to one one-hundredth of one percent FASTER than by any previous method," explained Phil "Compu" Dweeb, a FoReM aficionado, pausing occasionally to wipe the drool from his chin.

Industry insiders were quick to point out that using ZZZMODEM, it takes roughly 2 hours and 25 minutes to transfer a 20K file at 19,200 baud. Mr. Dweeb said that this problem has been dealt with. "Each block is padded with nulls, which take no time to send," he explained.

The new version of FoReM ST also has the new "Recursive ARCing" feature. As Mr. Dweeb explains: "All download files are recursively ARCed by FoReM before being put online. Our experience has shown that when you ARC a file, it gets smaller. Therefore, the approach we have taken is to repeatedly ARC the file until it reaches a size of roughly 10K. At that point, it's hardly worth the trouble, wouldn't you say?"

Reportedly in the works for a future release is the patented "One Length Encoding" process. Early reports suggest that this procedure can reduce the length of a file to just 1 bit. Mr. Dweeb takes up the story: "One day we were sitting around doing some hacken and phreaken, and one of us started thinking. All binary data is encoded into bits, which are represented by ones and zeros. This is because a wire can either carry a current or not, and wires can therefore be set up in a a series that can represent strings of ones and zeros. "Notice, however, that the real information is carried in the ones, since the others carry no current. I mean, what good does a wire do when it isn't carrying any current? So by dropping all the zeros, you can easily cut file sizes in half. So we decided that a cool way to speed up data transfer would be to only send the one bits. The results were phenomenal -- an average speed increase of 50%!! "After we finished the initial implementation, we kept finding ways to make the thing faster, and more efficient. But then we realised that we hadn't gone all the way. If you think about it, after you drop all the zeros, you're left with a string of ones. Simply count all the ones, and you're left with another binary string. Say you end up with 7541 ones. In binary, that's 1110101110101. So immediately we've reduced the number of bits from 7541 to 13. But by simply repeating the process, we can reduce it further. 1110101110101 becomes 111111111, or 9, which is 1001, which becomes 2, which is 10, or 1.

Once we reach a string length of 1, we have reached maximum file com-pression. We now have the capability to encode virtually unlimited amounts of information into a single digit! Long-distance bills will never be the same! "Now, that's not to say that there aren't a few problems. The biggest one we have encountered is that for some reason, there seems to be a certain amount of data loss during the re-conversion process. It seems that sometimes the file cannot be expanded into its original form. So, the solution we came up with was to have an encryption key associated with each file. When a One Length Encoded file is received and is undergoing decompression, the unique encryption key must be supplied. That way, we end up with a 100% success rate in our conversions!

"A problem which we are having difficulty resolving lies in the fact that to ensure a 100% success rate, the encryption key must be exactly as long as the original file. We are confident, however, that the use of our Recursive ARCing procedure will help to solve this problem..."

BITS & PIECES

Model I, III & 4

by Lance Wolstrup

Every so often I receive a letter that requires more than just a short answer on the mail pages, it requires an article of its very own. Such is the case with the correspondence from Henry Herrdegen, which is published in part on page 4 in this issue. Henry brings up several interesting topics which I hope are now cleared up. However, he also mentions that he is having a difficult time understanding Boolean logic. Don't feel bad - it is a topic that all of us have found troubling at one time or another. So, to help not only Henry, but also other interested readers, let's give it another shot and see if we can shed some light on this mysterious concept.

1. A memory location can hold 1 byte.
   This byte can have a value of:
   0 to 255 (decimal)
   0 to FF (hexadecimal)
   0000 0000 to 1111 1111 (binary).

2. Using the binary notation it can be seen that a byte is broken down into 8 separate pieces (called bits).

3. The bits are numbered from right to left:
   7 6 5 4 3 2 1 0
   and they have the following values:
   7 6 5 4 3 2 1 0
   128 64 32 16 8 4 2 1

Use the above table to see how values and bit settings correspond - for example, the value 129 would be represented as 128+1

```
                    7 6 5 4  3 2 1 0
129 = 128+1=        1 0 0 0  0 0 0 1
```

Another example might be the value 67. It can be represented as 64+2+1, or in binary:

```
                    7 6 5 4  3 2 1 0
67=64+2+1           0 1 0 0  0 0 1 1
```

The value 0 has all bits turned off, while 255 has all bits turned on:

```
0=                  7 6 5 4  3 2 1 0
                    0 0 0 0  0 0 0 0

255=128+64+32+16+8+4+2+1
                    7 6 5 4  3 2 1 0
                    1 1 1 1  1 1 1 1
```

Play with this until you are thoroughly familiar with how the values and bit settings relate.

OK, you're back with us, so let's assume we can continue on a somewhat quickened pace and we will now concentrate on the bit settings, rather than the values they produce.

As you can tell, a bit is either ON or OFF (ON=1, OFF=0). This means that each byte has 8 individual 'switches' that can be manipulated on or off, depending on the whim of the programmer. People, such as Randy Cook, Kim Watt, Roy Soltoff and other TRS-80 greats, have used this to advantage, saving precious memory for their programs. They set up data bit-tables where the setting of a bit would determine the status of a particular device - thus, each byte could handle up to 8 devices.

A handy example of bit-tables can be found in LS-DOS 6.3.1. Here Roy Soltoff has set up a series of data tables that are crucial to the workings of the DOS. It is called the FLAG$ table and it begins at memory location 6AH and continues through 83H and contain the following information:

6AH AFLAG$ Start CYL for Allocation search.

6BH BFLAG$;appears to do nothing

6CH CFLAG$;condition flag
- 0 - Can't change high$ via SVC-100
- 1 - @CMNDR in execution
- 2 - @KEYIN request from SYS1
- 3 - System request for drivers, filters, DCTs
- 4 - @CMNDR to only execute LIB commands
- 5 - Sysgen inhibit bit
- 6 - @ERROR inhibit display
- 7 - @ERROR to user (DE) buffer

6DH DFLAG$;device flag
- 0 - SPOOL is active
- 1 - TYPE ahead is active
- 2 - VERIFY is on
- 3 - SMOOTH active
- 4 - MemDISK active
- 5 - FORMS active
- 6 - KSM active
- 7 - accept graphics in screen print

```
6EH EFLAG$ ;This flag is for SYS13 usage.
           Use only bits 4, 5, and 6 to indicate
           user entry code to be passed to
           SYS13. SYS13 will be executed
           from SYS1 if this byte is non-zero,
           bit 4, 5, and 6 will be merged into
           the SYS13 (1000 1111) overlay
           request.

6FH FFLAG$ ;Port FE mask
70H GFLAG$ ;appears to do nothing
71H HFLAG$ ;appears to do nothing

72H IFLAG$ ;international flag
           0 - French
           1 - German
           2 - Swiss
           3 -
           4 -
           5 -
           6 - Special DMP mode ON/OFF
           7 - '7' bit mode ON/OFF
           This byte is 0 for US mode.

73H JFLAG$ ;appears to do nothing

74H KFLAG$ ;keyboard flag
           0 - BREAK latch
           1 - PAUSE latch
           2 - ENTER latch
           3 - reserved
           4 - reserved
           5 - CAPs lock
           6 - reserved
           7 - character in TYPE ahead

75H LFLAG$ ;LDOS (LS-DOS) feature inhibit
           0 - inhibit step rate question in
               FORMAT
           1 - reserved
           2 - reserved
           3 - reserved
           4 - inhibit 8" query in
               FLOPPY/DCT
           5 - inhibit # sides question in
               FORMAT
           6 - reserved for
           7 - IM 2 hardware
76H MFLAG$ ;MODOUT$ mask assignments
           0 - reserved
           1 - cassette motor on/off
           2 - mode select
               (0=80/64, 1=40/32)
           3 - enable alternate character set
           4 - enable external I/O
           5 - video wait states
               0=disable, 1=enable)
           6 - clock speed (0=2 mhz, 1=4mhz)
           7 - reserved

77H NFLAG$ ;network flag
           0 - allow setting of file open bit in
               DIR
           1 - reserved
           2 - reserved
           3 - reserved
           4 - reserved
           5 - reserved
           6 - set if in Task processor
           7 - reserved

78H OFLAG$ ;OPREG$ mem mgmt image port
           0 - SEL0 - select map overlay bit 0
           1 - SEL1 - select map overlay bit 1
           2 - 80/64 - 0=64, 1=80
           3 - inverse video
           4 - MBIT0 - memory map bit 0
           5 - MBIT1 - memory map bit 1
           6 - FXUPMEM - fix upper memory
           7 - PAGE - page 1K video RAM
               (set for 80x24)

79H PFLAG$ ;printer flag
           0 - reserved
           1 - reserved
           2 - reserved
           3 - reserved
           4 - reserved
           5 - reserved
           6 - reserved
           7 - printer spooler is paused

7AH QFLAG$ ;appears to do nothing

7BH RFLAG$ ;FDC retry count >=2
           set as 0000 1000
7CH SFLAG$ ;system flag
           0 - inhibit file open bit
           1 - set to 1 if bit 2 set & EXEC
               file opened
           2 - set by @RUN to permit load of
               EXEC file
           3 - SYSTEM (FAST)
           4 - BREAK key disabled
           5 - JCL active
           6 - force extended error messages
           7 - DEBUG to be turned on after
               load

7DH TFLAG$ ;type flag
            2 = Model 2
            4 = Model 4
            5 = Model 4P
           12 = Model 12
           16 = Model 16
```

```
7EH  UFLAG$ ;user defined flag

7FH  VFLAG$ ;video flag
             0 - set blink rate
             1 - 1=fastest
             2 - and
             3 - 7=slowest
             4 - display clock
             5 - cursor blink toggle bit
             6 - inhibit blinking cursor (user)
             7 - inhibit blinking cursor (system)

80H  WFLAG$ ;WRINT$ - interupt mask register
             0 - enable 1500 baud rising edge
             1 - enable 1500 baud falling edge
             2 - enable real time clock
             3 - enable I/O bus interrupts
             4 - enable RS-232 transmit
                 interrupts
             5 - enable RS-232 receive data
                 interrupts
             6 - enable RS-232 error interrupts
             7 - reserved

81H  XFLAG$ ;appears to do nothing
82H  YFLAGS ;appears to do nothing
83H  ZFLAG$ ;appears to do nothing
```

If you look closely at the FLAG$ table, you'll no doubt notice the many bit-tables filled with goodies that's just waiting to be taken advantage of.

The trick to effectively manipulate the data tables is to know how to turn a particular bit on or off. And this is where BOOLEAN logic comes in.

Boolean logic has several operators, but we shall only concern ourselves with two of them - AND and OR, as they are the ones that suit our purposes perfectly.

First, let's establish what each of the operators do - let's begin with AND:

The AND operator compares two numbers bit for bit. If the compared bits are both 1, then the result will be 1. Any other time the result will be 0.

```
1 AND 1 = 1
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0
```

The OR operator compares two numbers bit for bit. If both bits are 0, then the result will be 0. Any other time the result will be 1.

```
1 OR 1 = 1
0 OR 1 = 1
1 OR 0 = 1
0 OR 0 = 0
```

This allows us to turn bits on and off at will. If we wish to turn a particular bit ON, we simply OR the number with another number where only that bit is turned on. For example, imagine that we wish to turn on bit 6 of the value stored in memory location 500.

First, we would store the value from memory location 500 in, let's say, variable A.

```
A=PEEK(500)
```

Then we turn on bit 6 of variable A.

```
A=A or 64
```

Bit 6 is turned on if it was previously off - and kept on if the bit was already set. So, now we just need to store the new value back in memory location 500.

```
POKE 500,A
```

All of the above could, of course, been written more concisely with this command:

```
POKE 500,PEEK(500) OR 64
```

The following table shows the OR values needed to turn ON each bit in memory location 500.

```
turn on bit 0   POKE 500,PEEK(500) OR 1
turn on bit 1   POKE 500,PEEK(500) OR 2
turn on bit 2   POKE 500,PEEK(500) OR 4
turn on bit 3   POKE 500,PEEK(500) OR 8
turn on bit 4   POKE 500,PEEK(500) OR 16
turn on bit 5   POKE 500,PEEK(500) OR 32
turn on bit 6   POKE 500,PEEK(500) OR 64
turn on bit 7   POKE 500,PEEK(500) OR 128
```

To turn OFF a particular bit, you use the AND operator. You AND the number with another number where all the bits are turned on EXCEPT for the bit you wish to turn off. For example, imagine that we wish to turn off bit 6 of the value stored in memory location 500.

First, we would store the value in memory location 500 in, let's say, variable A.

```
A=PEEK(500)
```

At this point we have no idea what is stored in memory location 500, so we MUST make sure that we only turn off bit 6; therefore, use a number with all bits set except for bit 6 - that number is 1011 1111, also known as 191.

```
A=A and 191
```

Now, store the new value back in memory location 500.

POKE 500,A

Just as in the OR example, we could have done this in one simple step:

POKE 500,PEEK(500) AND 191

The following table shows the AND values. needed to turn OFF each bit in memory location 500.

turn off bit 0	POKE 500,PEEK(500) AND 254
turn off bit 1	POKE 500,PEEK(500) AND 253
turn off bit 2	POKE 500,PEEK(500) AND 251
turn off bit 3	POKE 500,PEEK(500) AND 247
turn off bit 4	POKE 500,PEEK(500) AND 239
turn off bit 5	POKE 500,PEEK(500) AND 223
turn off bit 6	POKE 500,PEEK(500) AND 191
turn off bit 7	POKE 500,PEEK(500) AND 127

Hope this lengthy piece has addressed at least some of your questions about Boolean logic and bit-fiddling, in particular. And thanks to Roy Soltoff's SOURCE for providing the information about the Model 4 LS-DOS FLAG$ tables.

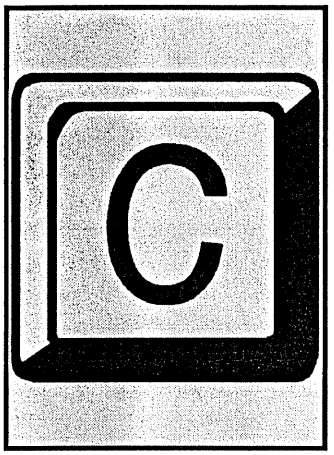
Finally, type in the program listing below, which I call BITS/BAS. It works on Models I/III & 4, and should help all to get a better understanding of the AND and OR functions.

BITS/BAS

```
10 IF PEEK(&H7D)=4 OR PEEK(&H7D)=5 THEN
SW=80 ELSE CLEAR 5000:SW=64
11 GOTO 100
20 H=0:GOTO 23
21 H=INT((SW-LEN(A$))/2):GOTO 23
22 H=SW-LEN(A$)
23 PRINT@SW*V+H,A$;:RETURN
30 A$=STRING$(ML,46):GOSUB 23:
A$=CHR$(14):GOSUB 23:L=0:FL=0:I$=""
31 A$=INKEY$:IF A$="" THEN 31
32 IF A$=CHR$(27) THEN FL=1:GOTO 39
ELSE IF A$=CHR$(13) THEN 39
33 IF A$=CHR$(8) AND L=0 THEN 31
34 IF A$=CHR$(8) THEN L=L-1:H=H-1:
I$=LEFT$(I$,L):A$=CHR$(46):GOSUB 23:
A$="":GOSUB 23:GOTO 31
35 IF A$<CHR$(48) THEN 31 ELSE IF A$>CHR$(57)
THEN IF A$<CHR$(65) OR A$>CHR$(122) THEN 31
36 IF L=ML THEN 31
37 I$=I$+A$:GOSUB 23:L=L+1:H=H+1:A$="":
GOSUB 23:GOTO 31
39 A$=CHR$(15)+CHR$(30):GOSUB 23:RETURN
40 I1=I:DV=128
41 FOR X=7 TO 0 STEP-1
42 A=INT(I1/DV)
43 A$=RIGHT$(STR$(A),1):GOSUB 23
44 IF X=4 THEN H=H+2 ELSE H=H+1
45 I1=I1-(DV)*A:DV=DV/2
46 NEXT:RETURN
60 FL=0:FOR X=1 TO LEN(I$):
IF MID$(I$,X,1)<CHR$(48) OR MID$(I$,X,1)> CHR$(57)
THEN FL=1
61 NEXT:RETURN
100 CLS:V=0:A$=CHR$(15)+"BITS/BAS":GOSUB 20:
A$="Boolean Math & Bit Settings":GOSUB 22:
V=V+1:A$="(c) Copyright 1994 by Lance Wolstrup":
GOSUB 22:V=V+1:A$="All rights reserved":GOSUB 22:
V=V+1:A$=STRING$(SW,140):GOSUB 20
110 V=4:A$=CHR$(31):GOSUB 20:V=5:H=55:
A$="BIT":GOSUB 23:V=6:H=52:FOR X=7 TO 0 STEP-1:
A$=RIGHT$(STR$(X),1):GOSUB 23:
IF X=4 THEN H=H+2 ELSE H=H+1
111 NEXT:V=7:H=52:A$=STRING$(9,131):GOSUB 23
120 V=8:H=5:
A$=CHR$(31)+"Enter first number (0-255):":
GOSUB 23:H=38
130 ML=3:GOSUB 30:IF FL THEN CLS:END
ELSE IF I$="" THEN 120 ELSE GOSUB 60:
IF FL THEN 120
140 I=VAL(I$):IF I>255 THEN 120 ELSE H=52:
GOSUB 40
145 N1=I:H=38:A$="":GOSUB 23:
PRINT USING"###";N1
150 V=9:
A$=CHR$(30)+"Enter Boolean Operator (AND/OR):":
GOSUB 20:H=33
160 ML=3:GOSUB 30:IF FL THEN 120 ELSE IF I$=""
THEN 150
170 FOR X=1 TO LEN(I$):M=ASC(MID$(I$,X,1)):
IF M>96 THEN M=M-32:MID$(I$,X,1)=CHR$(M)
180 NEXT
190 IF I$="AND" THEN B=0 ELSE IF I$="OR" THEN
B=1 ELSE 150
200 H=47:A$=I$:GOSUB 23
210 V=10:H=4:
A$=CHR$(30)+"Enter second number (0-255):":
GOSUB 23:H=38
220 ML=3:GOSUB 30:IF FL THEN 150 ELSE IF I$=""
THEN 210 ELSE GOSUB 60:IF FL THEN 210
230 I=VAL(I$):IF I>255 THEN 210 ELSE H=52:
GOSUB 40
235 N2=I:H=38:A$="":GOSUB 23:PRINT US-
ING"###";N2
240 V=11:H=37:A$=STRING$(5,131):GOSUB 23:H=52:
A$=STRING$(9,131):GOSUB 23
250 V=12:H=38:A$="":GOSUB 23
260 IF B THEN I=N1 OR N2 ELSE I=N1 AND N2
270 PRINT USING"###";I:H=52:GOSUB 40
280 V=14:A$="Press <ENTER> to continue ":
GOSUB 21:H=H+LEN(A$)
290 ML=0:GOSUB 30:IF FL THEN CLS:END ELSE 120
```

C Language Tutorial, Part I

By J.F.R. "Frank" Slinkman



Some hobby-level programmers I know, even good ones, have either shied away from, or given up on, the C programming language because it appears to be "too difficult" to learn.

Yes, there IS a lot to learn. But think back to when you first started out with BASIC or assembler. If you don't remember the frustration of programs that wouldn't run, error messages you didn't understand, and anger at the computer and at yourself for adding or leaving out a single comma or parenthesis in a complex line of code, either you're a programming genius, or one of your cerebral RAM chips has died.

Even with all the frustrations -- all the nights you sat at the keyboard 'til 3 a.m. because you were inches away from finding that bug, or you'd just thought of a much better way to solve a programming problem -- you stuck with it because it's "fun!" Well, as much "fun" as it is to program in BASIC and assembler, it's even MORE fun to program in C.

O.K. You already know BASIC, and maybe even assembler. So why learn C?

First, C has pretty much become the lingua franca of professional programmers. It is the main language they use to communicate programming ideas, algorithms and procedures to each other.

But more importantly, even at the hobbyist programming level, once you learn C, you can do many more things than you can do with pure assembler, such as work with floating point numbers and trigonometric functions.

Also, the data organization capabilities built in to the language will let you expand your programming horizons, presenting exciting new ways to solve old problems, and allowing you to tackle new kinds of projects that would be very difficult using only BASIC and/or assembler.

And, when the program is complete, it will run much faster than the same program written in BASIC with less code and, once you learn the language, with less work.

In short, C allows you to do more things, and do them far more productively, than other more limited languages.

I forget where I read it, but I once saw a comparison of the relative speed of programs written in various computer languages. The fastest, of course, is pure assembler. To perform the same task, interpreter BASIC takes 30 to 50 times longer to run; COBOL takes about 20 times longer; and compiled BASIC 5 to 10 times longer, and C only about two times longer.

Speaking of assembler, the output of a C compiler is not an executable program, but an assembly language source code listing which you can edit or optimize the same way you can with any assembly language program.

This listing is then assembled like any other relocatable assembly language listing to create the actual executable /CMD (or, in the case of MeSs-DOS,.EXE) program.

Thus, in a sense, C could be considered to be a "shortcut" way of writing assembler, while retaining the advantages of a higher level language like BASIC.

One of the biggest differences between BASIC and C is that the actual language contains very few "statements," while BASIC includes a large number of "commands".

In C, these simple building blocks are used to construct often elaborate "functions" to perform desired tasks. These functions can be saved in groups called "libraries". Thus, when you write a function you particularly like, you can give it a name and save it. Thereafter, if you need the same function in another program, all you have to do is "include" the name of the library in which your function is stored, and reference the function in your program by its name.

This helps you avoid constantly "re-inventing the wheel", and makes the time you spend programming far more productive.

There are also "standard libraries" of functions provided with C compilers, the contents of which are closely regulated by standards committees to help ensure C stays reasonably portable across all hardware platforms.

Thus, the programs you write on your TRS-80 will, with only very minor modification, compile and run on MeSs-DOS machines, Macs, Sun work stations, and even big mainframes. Likewise, C programs written for those machines can be easily modified to compile and run on your TRS-80.

To program in C, you need to learn the various

ways data can be stored and represented, the concept of "indirection", C's "statements" and "operators", and the "standard library" of "functions" and "header" files.

You will also need a good text editor to write your programs, and become familiar with your compiler's commands and options.

These articles will be written assuming the reader has a Model III or 4, some knowledge of BASIC, and the Pro-MC (Version 1.6b) and Pro-MRAS packages from Misosys, Inc. The Pro-MC compiler implements the original "K&R" C, and it's standard libraries are UNIX System V compatible.

But C is C; so even if you have another compiler and/or relocatable editor-assembler, most of what you read here will be usable on your system, with only minor adjustments and modifications.

Rather than give you long lists of things to memorize right from the start, these articles will take a more gradual approach.

They will start with very simple programs, which will become more and more complex as we go along, each of which will demonstrate various features of the language.

So let's start with our very first, and very simplest, program:

```
/* prog01.c */
#include<stdio.h>
main()
{  puts( "This is a message string" );
}
```

Note the code is in lower case (small) letters. This is the normal way of doing things in C. There are conventions regarding the use of upper case (capital) letters, but we'll worry about that later.

With C, tab stops are put every 4th column, not the usual 8. The SAID text editor included with the Pro MRAS package will automatically do this for you if you invoke it with its (C) parameter. Also, the compiler ignores all "white space" characters, such as tabs, spaces, carriage returns, etc. Instead, it relies on the presence of a semicolon at the end of each program statement, and on braces to define related blocks of program code.

On the Model 4 keyboard, the left and right braces are obtained by pressing <CLEAR><SHIFT> "<" and <CLEAR><SHIFT> ">", respectively.

Now use your text editor to type in the program above.

The first line is a comment, equivalent to a REMark in BASIC. In C, anything and everything between the begin-comment symbol ("/*") and the end-comment symbol ("*/"), is ignored by the compiler. In other words, it appears in the listing strictly for the convenience and/or edification of a human reading the program listing.

The next line "includes" the header file, STDIO.H. This file contains "prototypes" or "forward declarations" for most of the standard library functions you're likely to use, and loads other header files which define certain types of variables.

When the compiler's preprocessor sees an #include "preprocessor directive," it will search for the named file, read it, and put all declarations, definitions, directives, etc., found in that file into your program. This feature saves you having to re-invent the wheel each time you want to write a program.

In the future, you'll be writing your own header files, but for now we'll just use the "standard" ones.

The third line, "main()", indicates the start of a function. In C, the main() function is special. It is always the starting point of the program, no matter where it appears in the program listing.

Functions are indicated by opening and closing parentheses immediately following their names, and the code included in the function is placed within opening and closing braces.

Unless a "return" statement is encountered first, program control will return to the routine which called the function after the last line of code within the defining braces is executed. In this case, since this is the main() function, control will return to LS-DOS.

The one line of code in this function invokes ("calls") the standard library puts() function.

Puts() outputs a string to the "standard output "which, on our systems, is the monitor screen.

If you look in your manual for the documentation of puts(), you'll see where puts() takes one "argument", namely the RAM address of the string to be output.

In other words, the argument passed to puts() isn't the string itself, but a 16-bit POINTER to the string. This is a very simple introduction to the concept of "indirection" -- namely variables which point to data, rather than contain the data.

The puts() function returns a 16-bit code to the calling routine to indicate whether or not it was able to successfully execute. If successful, it returns "NULL" (zero). If unsuccessful, it returns "EOF" (-1).

Our little program assumes puts() is always going to work, so does not check the return code to test for failure.

The argument(s) sent to a function are listed, in order, inside the parentheses following the function name. They must be listed in the same order in which the function expects them. In this case, there is only one argument, so there's no argument order to worry about. But many other functions are more involved.

Now save the program with the name PROG01/CCC. (Some other compilers need it saved with the namePROG01/C.)

Now, from LS-DOS Ready, invoke the preprocessor. Under Pro-MC, you would do this with the command:

```
mcp prog01 +o=:d
```

where ":d" is the drive number where you want the PROG01/TOK file to be written.

This file is simply a tokenized version of the program. Now invoke the compiler itself. Under Pro-MC, you would use the command:

```
mc prog01 +o=:d
```

This reads the /TOK file, and outputs an assembly language listing of the program which, in this case, will be named PROG01/ASM.

Now load PROG01/ASM into your text editor, and find the label "MAIN:".

You'll see code something like:

```
MAIN:
            DSEG
$?1:        DB 'This is a message string',0
            CSEG
            LD    HL,$?1
            PUSH  HL
            CALL  PUTS
            POP   AF
            RET
```

Note the string at label "$?1:" ends with a zero (a" null character", equivalent to a CHR$(0) in BASIC). This is the way C determines the end of a character string. For this reason you cannot have null characters imbedded in ASCII strings as you can in BASIC.

Note also the "DSEG" and "CSEG" in the listing. These identify the start of "data segments" and "code segments" of the program which tell the linker program how to handle the code.

What this program does is create the string in RAM, load the RAM address of the string into the HL register, push the contents of HL onto the stack, call the puts() function, clear the stack by popping the first stacked value into the AF register, and return to whatever routine called it.

In C, arguments are passed to functions on the stack, and become the property of the called function, which uses them as variables which can be altered without affecting any variables in the calling routine. This automatically protects variables in the calling routine from being unintentionally altered, making the programmer's life a little easier, since he no longer has to keep track of such mundane details.

Now exit your text editor and get back to LS-DOS Ready, and use your assembler program to assemble PROG01/ASM to PROG01/REL. If using Pro-MRAS, do NOT use the "-gc" switch, since you want to generate a /REL file, not a /CMD file.

If using Pro-MRAS, use the command:

```
mras mc +i=prog01 +o=prog01:d -nl
```

which tells MRAS to assemble Pro-MC's MC/ASM file, include PROG01/ASM, and send the output to a file named PROG01/REL on the desired drive, and not to list the programs on the monitor screen. Now invoke the linker to produce the final /CMDprogram.

If you're using MLINK, the command is:

```
mlink prog01 -n=:d -e.
```

The linker searches the libraries included with your compiler (as well as other libraries you may have specified) for the functions utilized by PROG01, and link them together to create the final executable /CMDfile.

Of course, all the above steps could have been automated using Pro-MC's MC/JCL file, using the command:

```
do mc (n=prog01)
```

For this reason, we won't go through all the individual compilation steps again, but simply use MC/JCL. But it's important for you to know what's going on as MC/JCL goes through its various steps:

1. Use the pre-processor to create the tokenized file;
2. Compile the /TOK file to an assembly language listing (/ASM) file;
3. (Optional, and not discussed above) Optimize the /ASM file to an /OPT file;

4. Assemble the /ASM (or /OPT) file to a /REL file; and
5. Link the /REL file and the functions from the included libraries into the final executable /CMD file.

Now, at LS-DOS ready, enter PROG01, and the message, "This is a message string," should appear on your screen.

O.K. Assuming you got that little program to compile and run, you can REMOVE the files PROG01/TOK, PROG01/REL, and PROG01/ASM.

A "DIR PROG01" command will reveal you still have PROG01/CCC and PROG01/CMD. You may remove these too, at your option, as we won't be using them again.

Now it's time to write a similar, but slightly more involved program

```
/* prog02.c */
#include <stdio.h>
main()
{
   char   *msg1, *msg2;
   *msg1 = "First message";
   *msg2 = "Second message";

   puts( msg1 );
   puts( msg2 );
}
```

In C, all variables must be "declared" before they can be used. There are several "types" and "classes" of variables. In this example, the first line inside main() declares two variables, "msg1" and "msg2."

The "char" at the start of the declaration line tells the compiler these variable(s) deal with "char" data.

Char data is stored with each value contained in one 8-bit byte, and an "array of chars" or "char array" is the usual way to store strings of ASCII characters.

The asterisk ("*") before each variable name in this declaration line indicates that the variable does not contain the data itself, but is a "pointer" to the data.

Because these variables were not declared to be of some other class, they are of the default class, "auto". Auto variables are stored on the program stack, are created anew each time the function is invoked, and cease to exist when the function is exited.

At this point, even though the two pointer variables have been declared, they don't yet point to anything useful. That's why the next two lines "assign" values to them.

The first of these two lines causes the string, "First Message" to be created in RAM, and then puts the RAM address of the first character of the array into the variable "msg1."

In this case, the asterisk has a different meaning than it does on the declaration line. In declarations, the asterisk means "pointer." In all other cases, it means "the object being pointed to" or, more simply, the "object at."

In other words, the line

```
*msg1 = "First message";
```

means "the object at 'msg1' is the string 'First message'" or, more simply, "point 'msg1' to the string, 'First message'."

The next line both creates a second char array, and causes the "msg2" variable to point to it.

The next two lines cause the two messages to be displayed on the monitor screen, one after the other, by calling puts() twice; first with the argument equal to the value stored in "msg1" (i.e., the RAM address of the first string), and then with the argument equal to the value stored in "msg2" (the RAM address of the second string).

The important thing to understand here is that "msg1" and "msg2" are not themselves passed to puts(). The arguments passed to puts() are COPIES of the specified variables, not the variables themselves.

Now compile the program via the command:

```
do mc (n=prog02,k)
```

The MC/JCL "[K]ill" parameter removes all preliminary files (i.e., the /TOK, /REL and /ASM files), leaving only the original /CCC file and the executable /CMD file.

Now run PROG02. It should put the two strings on your screen on two separate lines, just below where you entered "PROG02."

Now load PROG02/CCC back into your text editor, and go to the line:

```
*msg1 = "First message";
```

Edit or replace this line, to make it read:

```
*msg1 = "\x1c\x1fFirst message";
```

On the Model 4 keyboard, the backslash is obtained by pressing <CLEAR> "/". This character has a special meaning in C, and is called the "escape" character.

The escape character has many uses. In this case it is used in conjunction with "x" to define two characters by their hexadecimal values. If we chose, we could have specified the same values as "\034\037," the octal equivalents of \x1c and \x1f. Remember, the "\x" combination followed by exactly two hexadecimal digits, and the "\" followed by exactly three octal digits (which should never be larger than "377," or 255 decimal), establishes an 8-bit (char) value. Since octal numbers are rather passe, I suggest you stick to hex when defining non-ASCII char values.

The new, edited line is equivalent to the BASIC line:

```
MSG1 = CHR$(28) + CHR$(31) +
"First message"
```

Now save out the modified program, recompile it, and run it. You'll notice that this time the screen is cleared before the two messages are displayed. That's because the codes for "home cursor" (28) and "clear to end of frame" (31) were imbedded in the first message string.

O.K. Assuming the modified PROG02 compiled and ran right, we're finished with it, and you can REMOVE or PURGE all PROG02 files and get to the next step -- doing some simple math.

C has four types of math variables, namely short (16-bit) integers, long (32-bit) integers, "floats" (single precision floating point numbers), and "doubles" (double precision floating point numbers).

Short integers are referred to as "ints," and long integers are referred to as "longs" or "long ints."

Both kinds of integers can be either signed or unsigned. Signed ints range in value from -32768 to+32767. Unsigned ints range from zero to 65535.

Signed longs range in value from -2,147,483,648 to 2,147,483,647, and unsigned longs range from zero to 4,294,967,295.

Signed short ints are exactly like BASIC integers, and floats and doubles are exactly like BASIC's single- and double precision floating point numbers.

However, when it comes to floating point numbers, the default mode in C is double, not single precision. Unfortunately, double precision math is slow; so most programmers, even when working on much faster computers than ours, go to great lengths to avoid floating point math unless it's absolutely necessary.

Pro-MC does have a feature, and some special non-standard math functions, which allow the default to be changed to much faster single precision math. But this is NOT standard C, and should NEVER be used if the portability of the code to other hardware platforms is a consideration.

If you'll recall, all variables must be declared before use. Here is how the various types of math variables can be declared:

```
Signed short integers:
int                 variable_name;
short               variable_name;
short int           variable_name;

Unsigned short integers:
unsigned            variable_name;
unsigned int        variable_name;

Signed long integers:
long int            variable_name;
long                variable_name;

  Unsigned long integers:
unsigned long       variable_name;
unsigned long int   variable_name;

Floats:
float               variable_name;

Doubles:
double              variable_name;
```

Of course, wherever "variable_name" appears in the table above, you would substitute the actual variable name.

In these articles, we will be using the shortest forms of the above, namely "int," "unsigned," "long," and "unsigned long."

In a straight rip-off of Radio Shack's old BASIC instruction manual, let's tackle solving the old "time, rate and distance" problem, namely "distance = time *rate."

```
/* prog03.c */
#include<stdio.h>
main()
{
  int    rate, time;

  rate = 55;
  time = 6;
```

```
    printf(
       "Rate = %d, Time = %d, Distance = %d\n",
          rate, time, rate * time );
}
```

In this program, the first line in the main() function declares two auto variables, "rate," and "time", to be of type short integer.

The next two lines assign the values 55 and 6 to "rate" and "time", respectively.

The last line invokes the standard library printf() function to display the result in a formatted manner.

Unlike puts(), printf() takes multiple arguments: a control string which tells the function how to format the data, and a list of the data items to display. In this case, there are three items in the list -- the two variables we have declared and their product.

In the control string are two codes. One is "%d," which tells the function to display the data in decimal form in the minimum number of characters necessary.

Note there are exactly as many data items in the list as there are "%d" codes. This is required for the function to work correctly.

The other code is "\n," which is the code for the "newline" character. You may have noticed that the puts() function we used earlier didn't require this. Puts() automatically adds its own newline character. Printf(), however, doesn't; so if we want the next data printed to be on the next line, we have to include it in the control string.

If we omitted the newline character, printf() would behave rather like a BASIC PRINT command with a semicolon at the end.

Actually, the whole "printf()" line should be on one program line, but the narrow column width here forced me to break it into three lines. It doesn't matter, though, because, if you'll remember, all white space characters are ignored; so the compiler will still see these three lines as one.

The only exception is that you can't break a string across two lines unless you use the escape character to tell the compiler what you want it to do.

For example, the string:

```
"Now is the time for all qui\
 ck foxes and lazy dogs to party"
```

will be interpreted as being on one line. However, if the escape character were omitted, the compiler would report an "unterminated string" error because it got to the end of a line without finding a closing quote.

Now type in, compile and run prog03.c to see how printf() formats and displays the data supplied to it.

O.K. Now it's time to write a program which makes some decisions; so we're going to radically modify prog03.c to create a new program, prog03a.c.

Before you start, be reminded that, on the Model 4, the "|" symbol is obtained via <CLEAR> <SHIFT> "/".

```
/* prog03a.c */
#include<stdio.h>

void   clr_scr();

char   inbuf[81];

main()
{
   int    rate = 0, time = 0, distance = 0;

   clr_scr();

   rate = get_val( "Input rate: " );

  do
     time = get_val( "Input time: " );
   while ( !rate && !time );

   while ( ( !rate || !time ) && !distance )
      distance = get_val( "Input distance: " );

   if ( !rate )
      rate = distance / time;
   else if ( !time )
      time = distance / rate;
   else
      distance = rate * time;

   printf(
      "Rate = %d, Time = %d, Distance = %d\n",
         rate, time, distance );
}

void clr_scr()
{  printf( "\x1c\x1f" ); }

int get_val( msg )
char   *msg;
{
   printf( msg );

   if ( !gets( inbuf ) )
      return NULL;
   else
      return atoi( inbuf );
}
```

This program introduces lots of new things; so we need to go slow here.

The first New Thing is the "forward declaration." In C, functions can return either no value or exactly one value. The default type of return values (or "return codes") is a short signed integer.

Functions which return no value (i.e., a "void" function), or return a type other than "int," should (and in most cases MUST) be declared in advance, to tell the compiler how to handle the values they return.

In this example, we're telling the compiler that the function "clr_scr()" returns no value.

It is not necessary to make a forward declaration for "get_val()" because this function returns an int.

The next line introduces a new kind of variable, the "global" variable. Our previous program examples only used auto variables local to the main() function.

But because "inbuf" is declared before the first line of executable code, it can be accessed anywhere in the program. By contrast, the "rate", "time", and "distance" variables in main() exist ONLY within main(), and CANNOT BE ACCESSED by ANY code outside the main() function.

The same is true of "msg" in the get_val() function. It exists ONLY in get_val().

This intentional limitation of the scope of variables is a boon to the programmer. One of the big irritations of programming in BASIC or assembler is keeping track of a large number of variables in a big program.

But in C, you can use the SAME variable names in different functions in the SAME program, and the variables stay entirely independent of each other!

Here, "inbuf" is declared as an array of 81 elements, each of type "char". In C, brackets ("[" and "]") are used to reference array elements.

Why 81? Well, in C, a normal line of input from the keyboard, or output to the monitor screen is 80 characters.

O.K, you say. That accounts for the first 80, so what's the 81st character there for?

Well, all data read from the keyboard is ASCII char data, and all strings in C are terminated by a null character; so we've got to make the array large enough to hold 80 characters plus the terminating "\0" (null character).

In the first line of main(), we introduce a new way to "initialize" variables. In the previous examples, we declared them and initialized them (put values in them) in separate steps. But here, we both declare and initialize them in the same step.

The next line of code calls the clr_scr() function. When we cleared the screen before, we imbedded the clear screen codes in a string. Here, we have built a separate function to perform that process. This is actually a more efficient way to do it if your program will be clearing the screen more than once. Every time you put "printf("\x1c\x1f");" in your program, the compiler is going to generate 11 bytes of data and machine language instructions:

```
            DSEG
$?LABEL:
            DB      28,31,0
            CSEG
            LD      HL,$?LABEL
            PUSH    HL
            CALL    PRINTF
            POP     AF
```

But when the above code is put into a separate function (with a RET instruction added at the end), calling it generates only three bytes of machine code:

```
            CALL    CLR_SCR
```

True, the call overhead will add a few millionths of a second to the screen clearing process, but that's a very small price to pay for saving 8 bytes of final program size each and every time you clear the screen.

In general, unless program speed is critical, the best way to structure C programs (or any program, for that matter) is to put all processes which will be used more than once into their own separate functions.

In the next line of code, we also do something we haven't done before, namely call a function we wrote ourselves: the get_val() function. Note that there is absolutely no difference between calling one of our own functions than calling one of the standard library functions. Functions are functions, and they are all called the same way.

In this case, the argument passed to get_val() is a pointer to the string, "Input rate: ". Now skip down the listing to the get_val() function.

The first line, "int get_val(msg)" says three things:

1. This function returns a short signed int;
2. It's name is "get_val;" and
3. It takes one argument.

The next line describes the argument by declaring it. In this case, it says "msg" is a pointer to char.

In some versions of C, the two lines would be put together as, "int get_val(char *msg)".

Note that the left brace indicating the start of the code for this function appears AFTER the function argument(s) is/are defined.

The first line of executable code passes "msg" to the printf() function to display it on the screen. Since there is no newline character in the string, the cursor will stay to the immediate right of the last character in the string.

Now we introduce one of the ten "statements" in the C language: the "if" statement.

The general format for this statement is:

```
if ( expression )
   program_statement;
```

In pseudo-code, its logical sequence is:

1. is "expression" TRUE?
   YES:
      execute "program_statement"
   NO:
      skip "program_statement."

Also, one or more "else" clauses can be added, and each "else" clause refers to the immediately preceding "if" statement.

Now, if you'll refer to the documentation for the standard library gets() function, you'll see it accepts a string up to 80 characters long from "stdin" (the standard input device -- the keyboard, in our case), and it does not add a newline character to it. It returns NULL if there is an error, or a pointer to the string which has been input if there is no error.

It requires an argument which is a pointer to a char buffer large enough to hold the data.

Notice the exclamation point ("!") before "gets." This is the NOT operator. In C, all values are either TRUE or FALSE. If a value is zero or NULL, it is FALSE. All non-zero values are TRUE.

The first thing that will happen is that gets() will be called to get some keyboard input. When the input is complete, the value returned by gets() will be either NULL (zero) or a pointer, which cannot be zero.

If gets() returns NULL, which equals FALSE, the "!" will logically reverse that to TRUE. In this event, the whole expression, "!gets(inbuf)," will evaluate to TRUE; so "return NULL;" will be executed.

This introduces another C statement, the "return" statement.

In void functions, "return" simply causes program control to return to the calling routine, much as the RETURN command does in BASIC.

However, "return" is often unnecessary in void functions, our clr_scr() function being one example. As I mentioned earlier, when a function runs out of code, it automatically returns to it's caller.

In functions which return values, "return" is required, and does two things at the same time: it specifies the value which will be returned by the function and it returns program control to the calling function.

In this case, "return NULL;" will cause our get_val() function stop execution, and return a value of zero to the calling routine.

If gets() returns a pointer, which must be non-zero and therefore TRUE, the "!" will logically reverse that to FALSE, which will cause the whole expression to evaluate to FALSE. In this case, the "return NULL;" will be skipped and the "else" clause executed.

The first thing the "else" clause does is pass a pointer to the keyboard data at "inbuf" to the standard library function atoi(). Atoi() converts ASCII data to an integer value (somewhat similar to the VAL command in BASIC), and returns that value. The return statement causes this value to be returned to the calling routine.

Now, going back to the main() function where we first called get_val(), the value returned by get_val() will be loaded into ("assigned" to) the variable "rate". In other words, if nothing was input, "rate" will be zero. Otherwise, "rate" will hold the value of the string which was input from the keyboard.

Now, to get a value for "time", we introduce the "do" statement. "Do" is one of three ways to set up program loops in C. Its general format is:

```
do
   program_statement;
while ( expression );
```

In pseudo-code, its logical sequence is:

1. execute "program_statement"
2. is "expression" TRUE?
   YES:
      goto 1
   NO:
      exit.

Here, the user will be prompted to input a value for the "time" variable. Then both "rate" and "time" will be evaluated via the expression "!rate && !time."

The "&&" is the "logical AND" operator; so what this expression really says is, "NOT 'rate' AND NOT 'time'."

In other words, this code will not accept a situation where both "rate" and "time" are equal to zero. At most, only one of the two can be zero. As long as both "rate" and "time" are zero, the program will keep going through the "do" loop, prompting the user to input a value for "time," until some non-zero value is entered.

The BASIC equivalent of this "do" loop is:

```
50 INPUT"Time", TIME :
IF ( RATE = 0 AND TIME = 0 ) THEN GOTO 50
```

DO NOT read any further until you FULLY understand how this "do" loop works.

Next, we use C's "while" statement to set up a loop to get a value for "distance". The format for "while" is:

```
while ( expression )
  program_statement;
```

In pseudo-code, its logical sequence is:

1. is "expression" TRUE?
   YES:
   A. execute "program_statement"
   B. goto 1
   NO:
   exit.

In this case, we first use the "| |" (logical OR) operator to tell us if either "rate" or "time" is zero. If neither is zero, then the "(!rate | | !time)" expression will evaluate to FALSE, and no attempt will be made to either evaluate "!distance" or to call get_val() to get a value for "distance".

If this expression evaluates to TRUE (i.e., if one or both of "rate" and "time" is zero), then, and only then, will the current value of "distance" be evaluated.

Remember, "distance" was initialized to zero (FALSE); so, at least the first time through this "while" loop, "!distance" will evaluate to TRUE.

Assuming one of "rate" and "time" is zero, then the only way this "while" loop can be exited is for the user to input some non-zero value for "distance".

The BASIC equivalent of this "while" loop is:

```
50 IF ( ( RATE = 0 OR TIME = 0 ) AND
    DISTANCE = 0 ) THEN INPUT"Distance",
    DISTANCE : GOTO 50
```

An overview of what main() has done so far may help your understanding. First, we got a value for "rate". We allow "rate" to be either zero or non-zero.

Second, we asked the user to input a value for "time". If "rate" is non-zero, we will accept a zero value for "time". Otherwise, we keep prompting the user to enter a non-zero value for "rate."

Third, because we only need values for two of the three variables, we checked the values for both "rate" and "time". If both are non-zero, we make no attempt to get a value for the third variable, "distance".

But if either "rate" or "time" are zero, then we keep badgering the user to input a value for "distance" until he enters some non-zero value.

Do NOT read any further until you understand this logic, and how it is implemented in both the "do" and "while" loops.

You understand it? Good!

Then this is a good time to point out the difference between "do" and "while" loops.

A "do" loop will always be executed at least once, regardless of whether it's contingent expression is TRUE or FALSE.

A "while" loop will be executed only if its contingent expression is TRUE. It will be skipped if the contingent expression initially evaluates to FALSE.

The rest of main() uses "if" statements with "else" clauses to determine which one of the three variables is unknown (zero).

When it finds the unknown variable, its value is calculated, then all three variables are displayed on the screen via a printf() call very much like the one in the previous program example.

The BASIC equivalent of this if-then-else chain is:

```
50 IF (RATE = 0) THEN RATE = DISTANCE /
    TIME ELSE IF (TIME = 0) THEN TIME =
    DISTANCE / RATE ELSE DISTANCE =
    RATE * TIME
```

Now compile and run PROG03A. Run it several times, trying different combinations of unknowns and values. Also, if you're "into" assembler, you might want to study PROG03A/ASM to see how the program is structured and how values are passed to and returned from functions.

O.K. That's all for this time. You've got two months to chew on all this, and get ready for the next exciting episode, when we'll get into floating point math and some other "fun" stuff. Don't remove or kill PROG03A/CCC -- we'll be using it again.

In the meantime, I suggest you beg, borrow (try your local public library), or buy (stealing is a no-no) a copy of "The New C Primer," written by the Waite Group, published by SAMS. This is the best C tutorial book I have seen so far which deals strictly with the pure "K&R" C we use on our Model III's and 4's. Don't get a book which deals with ANSI C or C++ at this point -- it'll only confuse and confound you.

Some Memory Meanderings

part 2

by Roy T. Beck

I have been doing a little looking at my favorite hate; a 486-50 which resides next to my trusty 4P and shares a LaserJet with it. Kind of ironic; the Model 4P drives a Model 4 Laser, and operates under LS-DOS version 6.3 in competition with the 486 which operates under MS-DOS version 6.2. I guess it is inevitable that MS-DOS will eventually have a higher version number than LS-DOS!

As I mentioned last time, the IBM family has a lot of memory, far more than the Z-80 equipped Model 4 and 4P possess. The original IBM PC typically had 64 K to 256 K of RAM, depending upon the depth of the owner's pocket book. The memory map was actually much larger, 1 Meg to be exact. The controlling factor, as always, is the number of address lines the CPU can manage. In the case of the 8088, 8086 and 80186, the address lines were 20 in number. The memory map is 2 to the power of 20, which is 1,048,576, decimal. In terms of K = 1024, this number is 1024 x 1024.

Now, if the original PC had a memory map of 1 Meg, and only 64 K or 256 K of RAM, what was the rest of the memory used for? Or was it used at all? The answer is that the original designer of the PC was thinking BIG, and he decided 10 x 64 K, or 640 K would obviously be more than ample space for user programs. Thus he reserved the lower 640 K of the 1 Meg as user program space, and then assigned blocks of the remaining 384 K of memory for various necessary housekeeping functions.

In the original layout, the memory assignment was as follows:

Conventional Memory	640 K, 00000H to 9FFFFH, into which up to 640K of DRAM could be addressed.
Upper Memory	384 K, A0000h to FFFFFH, reserved for system devices, as follows:

A0000H to BFFFFH was available for video display cards, the exact location and amount depending upon the type of video display in use. VGA and EGA require all of this area, but some older systems, (Hercules, other monochrome and CGA cards) used only a portion of it.

C0000H to DFFFFH was reserved for Hard Drive controllers and their ROMs and other adapters and devices which could be expected to be developed.

E0000H to FFFFFH was intended for the ROM BIOS and BASIC ROMs. The BASIC ROMs were addressed from F6000H to FDFFFH, and the BIOS runs from FE000H to FFFFFH.

As a practical matter, the C0000H to DFFFFH block is usually not fully populated, resulting in blank spots in the memory..

Similarly, the The E0000H to FFFFFH area is also seldom fully used. The ROM BIOS is usually located at or above F0000H. As an aside, it is interesting to note that the CPU begins execution at FFFF0H at power-on or when the hardware RESET is actuated. The design of the CPU and the memory map obviously had to be coordinated in choosing this value. The good ol' Z-80, (and the 8080 and 8085) CPU jumped to 0000H when RESET was actuated.

The memory map of the present PC's is a strange and wonderful arrangement. The original layout consisted of just two areas, the 640 K assigned for program storage, and the 384 K reserved for housekeeping purposes. Today, the 640 K is known as "Conventional Memory", and the 384 K is termed "Upper Memory". All memory in excess of the original 1 Meg is now known as "Extended Memory". So far, so good. But there is yet another kind of memory which actually came along before "Extended Memory" was implemented. This was known as "Expanded Memory", and it is really spooky. I say that because it appears and disappears, and is only accessible through holes, or windows, in Upper Memory. More on this later.

As time marched along and newer CPU's were designed '286, '386, etc, the world according to Garp (Intel, that is) became larger, which was accomplished by adding additional memory address lines to the newer CPU's. The original 20 address lines of the 8086 family were increased to 24 in the 286, which thereby increased the memory map to 16 Megs. 2 to the 24th power = 16,777,216, which, divided by 1024 = 16,384 K. Dividing a second time by 1024 yields 16 Megs. This new extension of the memory map was all well and good, but there were inevitably several flies in the PC ointment.

First, the problem of compatibility had to be faced, and a decision made as to whether the new memory was to be contiguous with the original 640 K, which would require relocating all the ROMs and adapters located in Upper Memory, or whether the system area would be left as is, and the new memory made non-contiguous, starting at 100000H and running upwards from there. Upward compatibility of existing software dictated that all the system stuff remain unchanged, with new memory starting from 100000H. I am sure the designers had begun to anticipate further growth of memory, and realized relocation of system stuff would only have to be done again the NEXT time the memory map was expanded, whereas if it was left at its original location, the question would never arise. For whatever reason, the system stuff was left in Upper Memory, and all new memory above 100000H became a new world known as "Extended Memory"

Fly #2 was that the 286 chip had certain "limitations" in its behavior. It had several operating modes, the first of which was the REAL mode, which could only access the original 1 Meg memory map. It has a second mode known as the VIRTUAL mode, in which it could access the Extended Memory from 100000H to 1000000H. Sounded great; here was this shiny new CPU which could access 16 whole Megs of memory! But I said there was a fly in the ointment... Intel had provided the necessary instructions to shift from REAL mode to VIRTUAL mode, but for whatever reason, deliberately omitted any instructions to shift it back to REAL mode. Why? Dunno, you'll have to ask Intel what that was all about. But, you say, the 286 really can use all 16 Megs. Yes, it's true it can get into the VIRTUAL mode easily, and can then access the additional 15 Megs of memory using the additional 4 address lines. But how to get back to the REAL mode where the lower 1 Meg is situated? The answer was that the CPU had to do a warm RESET, which required a relatively long period of initialization time. Since in the normal course of program execution of large programs, the chip has to toggle between modes very often, the result is that the chip spends an inordinate amount of time doing warm RESETs. That is the reason the 286, even at 33 mHz is such a poor performer in large programs.

Other CPU versions came along, including the 386 DX, the 486, and now the Pentium. All of these chips can shift modes under software control, thus eliminating the need for a soft RESET to return to the REAL mode. Further all of these have 32 address lines, allowing Extended Memory up to 4096 Megs, or 4 Gigabytes. Wow!

Besides the problem of the inherently "crippled" 286 CPU, there was another area of concern. There were numerous machines in daily operation with only the original 1 Meg of memory, but users of spreadsheets such as Lotus were demanding more than 640 K of memory as their size increased. Lotus and Intel got together (later joined by AST, Microsoft and others) to develop a new form of memory, named "Expanded Memory" which could be utilized by the earlier chips, including the 8088 and its cousins. The first widely used version of this concept was identified as the LIM EMS 3.2 standard, which spelled out how this new "Expanded Memory" would be software interfaced by all players. This scheme allowed for the addition of 8 Megs of Expanded Memory to all the existing PC's. In the case of the 8086 family, an additional card had to be added to the machine which housed the additional memory. (Note, this is not the memory added in the old "6 Pack" and similar cards; that was only part of the original 640 K). A later version of the LIM standard was numbered 4.0, and this allowed for a still larger Expanded Memory, 32 Megs.

So how does Expanded Memory function? You will remember I mentioned above that there are "holes", or unoccupied spaces, in the Upper Memory.

The Expanded Memory really consists of bank-switched blocks of memory, and makes four 16 K blocks available through a 64 K window, the window to be established somewhere in Upper Memory. The original 8 Megs of Expanded Memory is divided into 512 pages of 16 K each, any 4 of which can be "viewed" through the "window" in Upper Memory. The 4 pages need not be contiguous. I told you this was spooky, didn't I? A further problem is that not all machines could be expected to have a 64 K window in exactly the same place in Upper Memory as every other PC. I believe the installation program takes care of finding a suitable 64 K block in any specific machine, and I also think this can be overruled by a knowledgeable operator, who can specifiy where the window is to be located.

I don't care about these mechanics, only the concept is important. If you think back to the Model 4 and 4P I discussed in Part 1, you will see that this bank switching of multiple 16 K pages in a window is closely analogous with switching of single 32 K pages in the upper half of the Z-80's memory map, except that the PC is inherently more complex. The programmers claim anything can be done in software, and in this case they surely made it work.

Note that in the 8086 family, an extra card was required to house the LIM memory chips. In the case of the 286 and later chips, the memory used to form the 16 K pages is actually part of the Extended Memory of the machine, with the CPU performing

memory readdressing to bring the 16 K pages down to the selected window in the Upper Memory. By so doing, the software concepts developed for use by large programs in the 8086 era remain applicable to the 286 and later chips, thus avoiding obsolescence of the application programs. Due to the changed CPU configurations, a separate driver program is required in each case, usually named something like EMM286, etc. The later Microsoft DOS's include this driver. Where a board was supplied, the board vendor had to provide the necessary software driver to implement Expanded Memory on his board.

All of the above, so far, relates to memory mapping concepts. What about types of memory chips? The PC world has popularized some new types in addition to the dynamic RAMS (DRAMs) and ROMS used in the Model 4 world. One type is the EEPROM, which is an electrically erasable and writeable PROM. This is used in the PC family to hold semipermanent data which might have to be changed at some time in the life of the machine. Examples might include hard drive parameters on a controller card.

Another very important type is CMOS memory, which is a form of static RAM which will hold its memory contents indefinitely so long as a low DC voltage is applied to one pin on the chip. The current draw is microscopic, and can be neglected as insignificant. CMOS memory is used to hold all the startup parameters in 286 and later machines. When the machine is turned off this voltage is supplied by a small primary cell or nickel-cadmium storage cell mounted on the motherboard. Since nothing is forever, this battery will eventually fail, and the CMOS contents will be lost. It is for this reason you must periodically replace the battery, and certainly you must record and save (on paper) the data normally present in the CMOS memory.

As a curiousity, many lap-top computers including the TRS Models 100, 102 and 200 all have CMOS main memory. It is for this reason that many files are kept in memory even when the machine is turned off, and are instantly available when the machine is turned on. It is a nice feature, but it does tend to clutter up the memory with programs and data not presently in use.

Those of you who go back to the Model I may remember the presence of "snivvies" on the screen when the Z-80 was writing to the screen. These were the result of competition by the CPU and the screen scanning circuitry for the attention of the screen memory. The CPU had priority, so whenever it wanted to write to the screen, the scan circuit was prevented from accessing the screen memory, so was simply given a blank character to display. Dan Dresselhaus of SAGATUG created a small circuit hack to overcome this. His circuit required the CPU to defer its access until the scanning circuit was doing a horizontal or vertical retrace. The result was to slightly slow the CPU's effective speed and eliminate the snivvies on the screen. The hack worked beautifully, and several club members installed it in their machines. Radio Shack also thought about the snivvies, and in the Model 4 they added an equivalent of Dan's circuit. Dan had a switch to turn his circuit on or off, Radio Shack implemented it by way of a bit in a port, with the default turning off the snivvies. If you have special needs, you can flip the bit to the other position.

All of the above paragraph is by way of introduction to another memory chip. There now exists a dual-ported screen memory chip which will allow unrestricted access by the CPU while simultaneously allowing the scanning circuit to access the stored byte. This automatically eliminates the snivvies without slowing the CPU. Here is a good example of necessity being the mother of invention.

I believe I have about exhausted my store of memory about memories, and so I will wrap this article up and send it to Lance. Good night, all.